"Service Oriented Architecture and Cloud Computing"

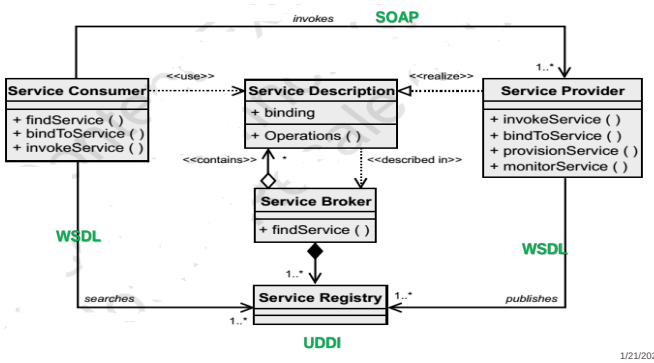
Course by
Dr. S. K. Rath

NIT Rourkela, Odisha

Research Agenda -SOA

- **Gartner Group** Stamford, Connecticut, United States, 1979 (http://en.wikipedia.org/wiki/Gartner) **2007 –50 % new mission critical operational application**
- **Business processes in 2007 were designed around SOA and that number was projected to be more than 80 % by 2010.**
- **In may 2009, Forrester Research** Cambridge, US 1983(http://en.wikipedia.org/wiki/Forrester_Research) **claim that 75% of IT executive at global 2000 organizations plan to use SOA by the end of year 2010.**
- **Only 1% reported SOA as negative**

Basic SOA Architecture



The 7 steps to SOA are:

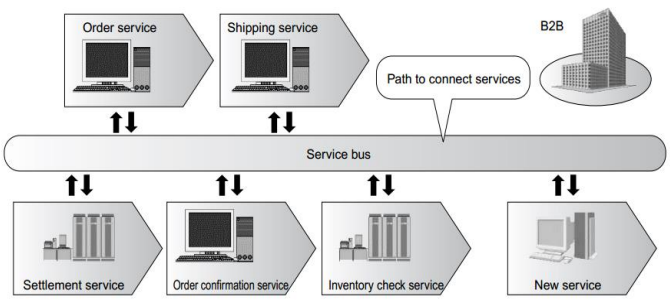
1. Create/Expose Services
2. Register Services
3. Secure Services
4. Manage (monitor) Services
5. Mediate and Virtualize Services
6. Govern the SOA
7. Integrate Services (ESB)

- ESB--------

"*An Enterprise Service Bus (ESB) is a new architecture pattern that exploits Web Services, messaging, middleware, intelligent routing, and transformation.*

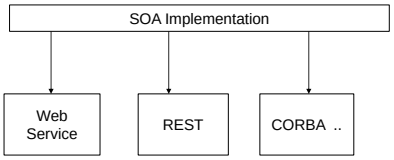
ESBs act as a lightweight, ubiquitous integration backbone through which software services and application components flow"

ESB



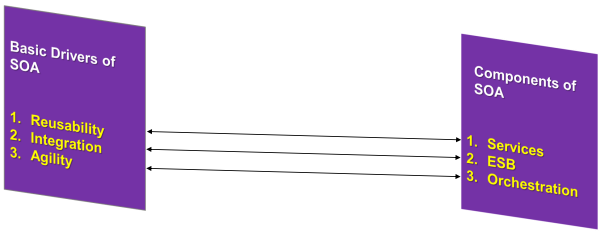
Strongest effects of SOA is that it makes a system robust against change.

Realization / Implementation of SOA Concept



Many different possible approaches to SOA
– CORBA
– J(2)EE
• JMS
• Two most popular approaches are
– WS-*
– REST

Drivers → Components of SOA



SOA

SOA is an **architecture style** for building **business applications** as a set of loosely coupled **black-box components** orchestrated to deliver a well-defined **level of service** by linking together **business processes**

(SOA)

- **Service-oriented architecture** (**SOA**) is a software design and software architecture design pattern based on discrete pieces of software providing application functionality as services to other applications. This is known as Service-orientation. It is independent of any vendor, product or technology.
- Services can be combined by other software applications to provide the complete functionality of a large software application.
- SOA makes it easy for computers connected over a network to cooperate.

SOA

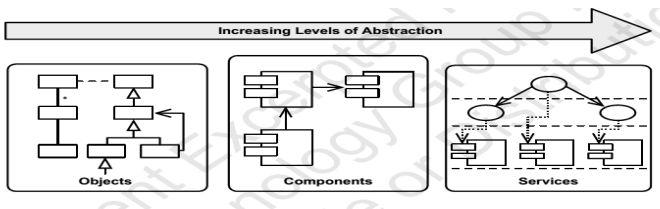
- Every computer can run an arbitrary number of services, and each service is built in a way that ensures that the service can exchange information with any other service in the network without human interaction and without the need to make changes to the underlying program itself.
- SOA design principles are used during software development and integration.

What Are the Fundamental Constructs (the DNA) of SOA?

The most basic construct or building block of SOA is a service

Software engineering over the years has evolved from procedural to structured programming to object-oriented programming to component-based development and now to service oriented.

Different levels of abstraction



Service consumer, service provider, service description, service broker, and a registry are all part of the DNA of SOA.

Service Classification

- **Elementary services**: The smallest possible components that qualify as services. For example, a service that can be used for zip code lookups.
- **Composite services**: Services that result from combining two or more other services into one service that provides more value. Composite services are executed in a single transaction and their execution time is relatively short. An example is a service to book a hotel and flight together.
- **Process services**: Longer running services that can take a couple of hours, days, or even more to complete and span multiple transactions. An example is a service for ordering a book online. The entire process (order, pay, ship, deliver) involves multiple transactions and takes a couple of hours at least.

Component vs Service

Difference Between SaaS and SOA

- SOA is a manufacturing model which deals with designing and building software by applying the service oriented computing principles to software solutions, while SaaS is a model for sales and distribution of software applications.
- In simpler terms, SaaS is a means of delivering software as services over the internet to its subscribers, while SOA is an architectural model in which the smallest unit of logic is a service. So, SOA (an architectural strategy) and SaaS (a business model) cannot be directly compared.
- However, to get the maximum benefits of cost reduction and agility, it is highly recommended that enterprises integrate SOA and SaaS together.

- **SAAS provides service To-----End users**
- **SOA provides service To-------Applications**

When Should SOA be Used?

- Service oriented architecture is not the right answer for all technology problems. **Some of the common situations where SOA lends value are:**
- New IT Solutions requiring integration of Enterprise capability from disparate IT systems and programming models.
- Increased efficiency in working with business partners for driving revenue, saving costs and for off-loading non-core functions Improving employee productivity by providing information they need when and where they need it;
- Leveraging existing enterprise assets by making them accessible for reuse outside their original purpose;
- Permitting organizations to change quickly in response to changes in the marketplace or competitive Maintaining cross-institution academic records

Understanding the Problem

What problems people who apply Service Oriented Architecture(SOA) are trying to solve. The problems can be categorized into two major areas:

- The mismatch between the business and IT
- Duplication of functionality and process silos

One discipline that can help solve these issues is the application of architecture in an organization and in projects.

Mismatch between business and IT

- As organizations are so dependent on information, it is very important that the technology that provides this information and is used to support these processes is **in line** with the ***needs of*** the organization.
  - **This is what we call business and IT alignment.**
  - **Henderson and Venkat raman can be seen as the founding fathers of business/IT alignment**

Objective of business and IT alignment is to manage three separate risks associated with IT projects:

- **Technical risk**: will the system function, as it should?
- **Organizational risk**: will individuals within the organization use the system as they should?
- **Business risk**: will the implementation and adoption of the system translate into business value?

Business value is *jeopardized/adventure* unless all three risks are managed successfully.

Technical misalignment of business and IT manifests in two ways:

- IT is not able to change fast enough along with the business
- IT is not able to deliver the functionality the business needs correctly

The first item, IT not being able to change fast enough, is becoming more and more important in today's market.

It is one of the problems that SOA can help you solve, if applied correctly

Duplication of functionality and data

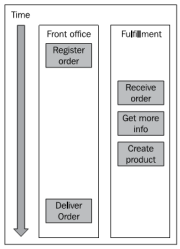
Traditionally, companies are organized functionally

- All these departments use their own IT systems that keep track of the data that is needed.
- Because all the departments use their own IT systems, and these systems are not connected to each other, information is duplicated within an organization.
- This can lead to differences between departments, because the information is not only stored, but also changed in these systems.
  - This leads to inconsistencies across the organization, unless the information is synchronized between all the systems

Process silos

- Departments that are self sufficient and isolated from the other departments are called **organizational silos**.
- These silos not only lead to duplication of functionality and data, but also to suboptimal process execution.
- The processes are divided based on organizational structure, not based on the most efficient end-to-end process. These processes are often referred to as **process silos**.

So even though the front office optimized its processing time, the total end-to-end client process has become slower because of the organizational silos.



SOA

- Enterprise Architecture (EA) is tasked to ensure Business-IT alignment.
- Service Oriented Architecture (SOA) is about bridging the gap between **Business and IT through** well defined, business aligned services developed by subscribing to established design principle, frameworks, pattern and methods.
- The objectives of EA and SOA are quite similar. However, whereas EA is a framework that covers all dimensions of enterprise IT architecture, SOA provides an architectural strategy that uses the concept of a “Service” as the underpinning business-IT alignment entity.

SOA Myths and Facts

There are several myths associated with SOA which are very important to understand before digging deeper into it.

Myth	Fact
SOA is a technology	**SOA is a design philosophy independent of any vendor, product, technology or industry trend. No vendor will ever offer a "complete" SOA "stack" because SOA needs vary from one organization to another. Purchasing your SOA infrastructure from a single vendor defeats the purpose of investing in SOA.**
SOAs require Web Services	SOAs may be realized via Web services but Web services are not necessarily required to implement SOA
SOA is new and revolutionary	EDI, CORBA and DCOM were conceptual examples of SOA
SOA ensures the alignment of IT and business	SOA is not a methodology
A SOA Reference Architecture reduces implementation risk	SOAs are like snowflakes – no two are the same. A SOA Reference Architecture may not necessarily provide the best solution for your organization
SOA requires a complete technology and business processes overhaul	SOA should be incremental and built upon your current investments
We need to build a SOA	SOA is a means, not an end

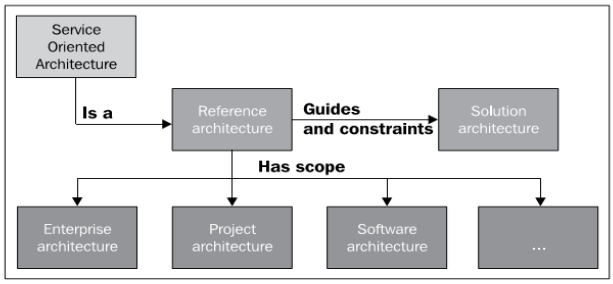
Focus on delivering a solution, not an SOA. SOA is a means to delivering your solution and should not be your end goal

Architecture Ontology

- Now we will learn about the different types of architecture that you can apply or use for your organization to ensure business and IT alignment.
- This will put Service Oriented Architecture in perspective
- There are two different axes on which you plot architecture;
  - one is scope of the system, and the other one is generalization.
  - Scope can be really big, or specific and small.

- **Ontology** is the philosophical study of the nature of *being*, *becoming*, *existence*, or *reality*, as well as the basic categories of being and their relations.
- **ontology** - (computer science) a rigorous and exhaustive organization of some knowledge domain that is usually hierarchical and contains all the relevant entities and their relations

Architecture Ontology



Enterprise Architecture

- If the scope of the architecture consists of the entire **company or organization** (enterprise), or the architecture scope spans multiple organizations, we practice enterprise architecture.

"Enterprise Architecture is the organizing logic for business processes and IT infrastructure, reflecting the integration and standardization requirements of the company's operating model.

The enterprise architecture provides a long term view of a company's processes, systems, and technologies so that individual projects can build capabilities, not just fulfill immediate needs".

Reference Architecture

- There are a lot of similarities between different organizations in the same industry.
- For this reason, reference architectures are developed.

"A reference architecture in the field of software architecture or enterprise architecture provides a template solution for an architecture for a particular domain. It also provides a common vocabulary with which to discuss implementations, often with the aim to stress commonality".

- **Generalized Enterprise Reference Architecture and Methodology** (GERAM) is a generalized Enterprise Architecture framework for enterprise integration and business process engineering
- This framework was developed in the 1990s by an International Federation of Automatic Control**(IFAC)**Task Force on Architectures for Enterprise Integration

Using reference architecture has several advantages

- Standardization of terminology, taxonomy, and services working with suppliers and partners.
- It reduces the cost of developing enterprise architecture; the organization can focus on what sets it apart from the reference architecture and other organizations in the industry instead of reinventing the wheel.
- It makes it easier to implement commercial **off-the-shelf (COTS)** software, because common terminology and processes are used

Obviously, RA also has some disadvantages:

- It takes some time to learn industry reference architecture; they are often (over) complete
- The reference architecture is typically written by a group of people from different organizations and so is the result of a compromise

Solution architecture

- Solution architecture is a detailed (technology) specification of building blocks to realize a business need.
- Open Group recognizes different types of solutions in the solution continuum:
  - **Foundation solutions**: This can be a programming language, a process or other highly generic concepts, tools, products, and services; an example of a process is Business Information Services Library (BISL)- a framework used for business information management.

Solution architecture

- **Common systems solutions**: For example CRM systems, ERP systems and security solutions.
- **Industry solutions:** These are solutions for a specific industry. They are built from foundation solutions and **common systems** solutions, and are **augmented** with industry-specific components.
- **Organization-specific solutions**: An example of this is the solution for the health insurance companies that want to offer self-service to prospective clients. The solution architecture describes the multi-channel solution for the organization, the tools and products that are used to implement it, and the relationship between the different layers.

The difference between Reference Architecture and Solution Architecture

- is that reference architecture is a **generic reference** where common problems are described together with principles and constraints on how to solve these.
- Solution architectures describe a specific solution to solve a specific problem or problems.
  - For example, the Dutch reference architecture NORA is used as a reference for the solution architecture in a ***municipality***.

- **NORA-(Nederlandse Overheid Referentie Architectuur)**

Project Architecture

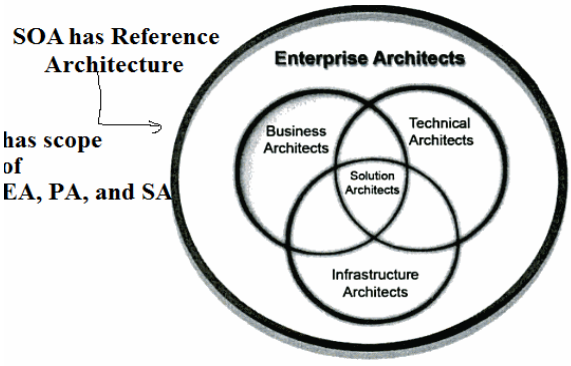
- Defines what part of a solution architecture will be realized by the project, or is in scope. This can overlap with a solution architecture if there is only one project needed to realize the solution architecture.
- If the solution is too big for one project, or there are different parts that are assigned to different projects, the project architecture describes what part of the solution architecture will be realized in the project.
- Project architecture serves two purposes
  - **Guarding the enterprise or solution architecture:**
  - **Provide input for, or change the enterprise or solution architecture:**

Software Architecture

- Software Architecture focused on the **structure** of the software.
- It can be viewed as a special type of solution architecture, project architecture, or part of the technology architecture.
- The target audience of these types of architectures are developers. Often these are not referred to as architecture, but rather as software design.

Service Oriented Architecture

- Service Oriented Architecture is a reference architecture that can be applied in organizations that are part of a **(supply-) chain or network**, and organizations that have to deal with a lot of **regulatory changes, or fast-changing markets.**
- It makes it possible to change parts of the IT landscape, and processes, without affecting other parts.
- Reusing existing functionality (services) makes sure that it can be changed quickly because you don't have to start from scratch every time, and also makes it cheaper.



Summary

"Service Oriented Architecture is a specific reference architecture that helps solve the data and functionality duplication, thus making the companies that apply this more flexible, and operate more efficiently"

Definition

"SOA is a conceptual business architecture where business functionality, or application logic, is made available to SOA users, or consumers; as shared, reusable services on an IT network. "Services" in an SOA are modules of business or application functionality with exposed interfaces, and are invoked by messages."

What Is the Difference Between a Web Service and an SOA Service?

- SOA services can be realized as web services, but not all web services are equal to SOA services.
- Web Services are self describing services that will perform well defined tasks and can be accessed through the web.
- (SOA) is an architecture paradigm that focuses on building systems through the use of different Web Services, integrating them together to make up the whole system.
- SOA services are services that fulfill a key step or activity of a business process and can be described as business services and are often exposed as web services.

Conti..

In an SOA, business processes, activities, and workflow are broken down into constituent functional elements called services.

They can be accessed and used directly by applications, or they can be mixed and matched with other services to create new business capabilities. **Business services or SOA services are reusable business capabilities.**

Examples in banking include open account or change address.

Why to adopt SOA?

- In 1917, **Forbes** compiled first 100 largest American companies list.
  - In 1987, Forbes published its 100 list & compared it to its 1917 list
    - Of the original group, 61 had **ceased** to exist
    - Of the remaining 39, only 18 had managed to stay in the top 100.
- Reason- Lack of maintaining agility
  - **Solution- SOA**

Why...

Why..

Today, the very survival of a business hinges on its ability to adapt its IT to meet ever-changing business challenges.

SOA helps business and IT to unify goals and bridge the gaps between their very separate worlds by establishing a common language and creating a more flexible infrastructure to support change

Why...

Service oriented architecture (SOA) is a business approach to building IT systems that allows businesses to

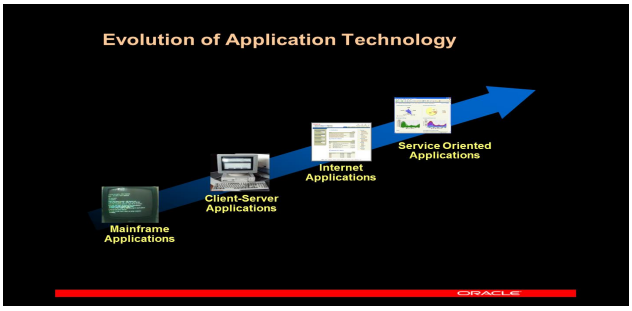
- _ **Leverage existing assets**
- _ **Create new ones**
- _ **easily enable the inevitable changes required to support the business**

"SOA Means Smarter Business and Smarter IT"

- SOA can make it easier and faster to build and deploy IT systems that directly serve the goals of a business.
- SOA integrates business requirements with an IT framework that simultaneously leverages existing systems and enables business change.
- A SOA enables the business to keep its focus on business and allows IT to evolve and keep pace in a dynamically changing world.

SOA enables businesses to make business decisions *supported* by technology instead of making business decisions *determined* by or *constrained* by technology. And with SOA, the folks in IT finally get to say "yes" more often than they say "no.

Evolution



- **ERP**: ERP stands for Enterprise Resource Planning. An ERP system, when implemented, would be helpful in doing business transactions related to Human Resource, Payroll, Absence Management, Accounting, Purchasing etc.
- Main functionality of ERP is to integrate all the roles/functions in an organization so that they share information with each other.
- **SOA**: In general, SOA could be termed as a tool for developing software. The main functionality behind using SOA is re-usability.

ERP using SOA:

- Service Oriented Architecture can be found at the heart of an ERP system.
- ERP systems often store similar data, that is shared by various modules or functional areas, in a single place.
- They are not always in the same environment but SOA is more likely to be found in integrated systems.

Differences:

- SOA is a tool that is transparent to system users–
- ERP is an architecture linking multiple functions together.
- Comparing the two is like comparing a **house** with a ***hammer***–
- one is used to build the other but you can't live in a hammer. You can't assume a hammer was used because there are other tools, such as nail guns
- SOA is a subset of ERP, having a core functionality of re-usability
- On the hand, ERP is more like a connecting the functionalities themselves like Payroll, Absence Management, Finance etc.

SOA and Cloud

"It's important to have a service-oriented architecture in place before you move to the cloud," says Tierney.

Though few companies will shift quickly over to 100-percent SOA, the aggressive introduction of the cloud brings urgency to the SOA model.

Addressing the challenges of the hybrid cloud requires a cultural shift for IT as well as a forward-looking SOA platform.

IT must take the initiative to get executive buy-in in this evolving hybrid cloud model or risk being marginalized initially and blamed later. –SOA a necessary **precursor** to an optimal hybrid cloud.

Another important point about the **SOA horse** before the **cloud cart:** "As you form partnerships with cloud vendors, it's important that you have a low barrier to exit," **Tierney says**. "You want the flexibility to pull back or to move to another cloud vendor.

One of the key tenants to **SOA** is that there are no direct connections between applications. Likewise, you don't want it to be difficult to replace the connections to your various cloud service providers. You want **loose coupling** between you and your service providers."

Business Process Management:

BPM is the next wave following on SOA. "We layer business process management on top of SOA, using the same set of application adapters, database adapters," Tierney points out.

BPM can be a great benefit to the emergence and development of the hybrid cloud.

And once you get executive buy-in for your COE, you could create a business process around cloud-service requests, approval, and confirmation, Tierney advises.

BPEL

BPEL is a language for defining ***executable* business** processes

- enables:
  - Definition of Business Processes using Web Services
  - Coordination of a set of Web service interactions
  - Degree of interoperability at the process level (describe process and use it in different runtime infrastructures)

where it comes from:

- Builds on XML and Web Services
- Convergence of two workflow.

BPEL Process Definition

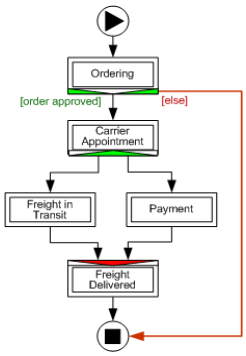
- BPEL defines an executable process by specifying
  - ***Activities*** and their execution order
  - ***Partners*** interacting with the process
  - ***Data necessary*** for and resulting from the execution
  - ***Messages*** exchanged between the partners
  - ***Fault handing*** in case of errors and exceptions
- Example: a simplified structure of a BPEL process

```
1 <process name="..."
2    targetNamespace="http://www..." >
3    ...
4    <partnerLinks> ...
5    <messageExchanges> ...
6    <variables> ...
7    ...
8    <faultHandlers> ...
9    <eventHandlers> ...
10   ...
11   activity
12 </process>
```


BPEL Activities

- ***invoke***: invoking operations offered by partner Web services
- ***receive***: waiting for messages from partner Web services
- ***reply*:** for capturing interactions
- ***wait*:** delaying the process execution
- ***assign*:** updating variables
- ***throw***: signalling faults
- ***Re-throw*:** propagating the faults that are not solved
- ***compensate***: triggering a compensation handler
- ***empty*:** doing nothing
- ***exit*:** ending a process immediately

Example: Order Fulfillment Process




```
1  <sequence>
2     <scope name = "Ordering"> ..... </scope>
3     <if>
4        <condition>
5           $POApprovalResult = "approved"
6        </condition>
7        <sequence name ="continue">
8           <scope name = "CarrierAppointment"> ..... </scope>
9              <flow>
10                <scope name = "FreightInTransit"> ..... </scope>
11                <scope name = "Payment"> ..... </scope>
12             </flow>
13          <scope name = "FreightDelivered"> ..... </scope>
14       </sequence>
15       <else>
16          <exit />
17       </else>
18    </if>
19 </sequence>
```


Orchestration

- **Orchestration** describes the automated arrangement, coordination, and management of complex computer systems, middleware, and services.

1. Conductor
2. Four groups of related musical instruments.
   1. Woodwinds: **Flute, Obes, Saxophone, Clarinets, Bassoons**
   2. Brass: **Trumpet, French horns, Trombones, Tuba**
   3. Percussion: **Timpani, Snare drum, Cymbals, Bells**
   4. Strings: **Violin, Cello, Double bass**

What is an Orchestra?

- An **orchestra** (/ˈɔːrkɪstrə/ or US /ˈɔːr.kɛstrə/; Italian: [orˈkɛstra]) is a large instrumental ensemble typical of classical music, which features string instruments such as violin, viola, cello and double bass, as well as brass, woodwinds, and percussion instruments, grouped in sections. Other instruments such as the piano and celesta may sometimes appear in a fifth keyboard section or may stand alone, as may the concert harp and, for performances of some modern compositions, electronic instruments.
- The term *orchestra* derives from the Greek ὀρχήστρα (*orchestra*), the name for the area in front of a stage in ancient Greek theatre reserved for the Greek chorus.[1]
- A full-size orchestra may sometimes be called a *symphony orchestra* or *philharmonic orchestra*.
  - The actual number of musicians employed in a given performance may vary from seventy to over one hundred musicians, depending on the work being played and the size of the venue. The term *chamber orchestra* (and sometimes *concert orchestra*) usually refers to smaller-sized ensembles of about fifty musicians or fewer.

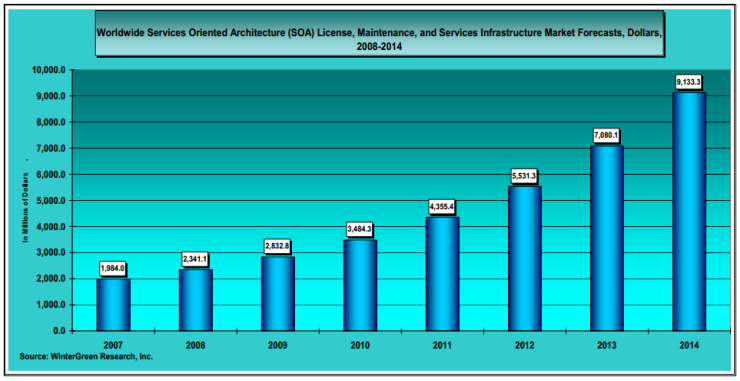
SOA

Chapter 2

Case Studies

Table ES-3

SOA Market Forecasts, 2008-2014



Case Studies

- Case studies are developed for two organizations
  - RailCo Ltd.
  - Transit Line Systems
- 125 examples related to the case studies are presented

RailCo Ltd

- Parts supplier for railways
- Specialized in airbrakes, tools
- International sales 10%, Domestic 90%- North America
- Primary **line of business**: resale of parts
- Secondary LOB: technician hire out for installations and repair from local

RailCo Ltd - History

- Established: Early 90s
- Staff has grown from 12 to 40
- Started as a **brokerage** for wholesalers
- Specialized in **air brakes**
- Became a **wholesaler** by dealing directly with air brake parts manufacturers

RailCo Ltd – Technical Infrastructure

- **Five employees** and **one manager** dedicated to full-time IT duties
- Maintain client workstations and back-end servers
- Custom development is outsourced
- Periodic upgrades and maintenance is in-house

RailCo Ltd – Automation Solutions

- Two-tier **client server** for all accounting and inventory control
- Two administrative clerks feed this solution with transaction document data (purchase orders and invoices)
- Receipt and submission of these documents initiates corresponding inventory receiving and order shipping processes

RailCo Ltd – Automation Solutions

- A **contact management system** in which customer and business partner profiles are maintained
- Consists of a database fronted by web-based data entry and reporting interfaces
- Users: **managers, admin assistants, accounting personnel**

RailCo Ltd – Situation,Business Goals, Obstacles

- Profit margins are **declining**
- Business process **overhead** is limiting the ability to be competitive
- Clients are switching to another company providing same products more **efficiently and at lower cost**
- Competitor has implemented a **B2B** solution for certain transactions online.
- **Business-to-business** (**B2B**) describes commerce transactions between businesses, such as between a manufacturer and a wholesaler, or between a wholesaler and a retailer.
- RailCo's primary client, **Transit Line Systems** has started an online relationship with the competitors as well

RailCo Ltd – Situation,Business Goals, Obstacles

- Outdated technology must be overhauled
- Must upgrade its automation environment
- **Top priority**: participate in online transactions with TLS
- RailCo. has hurriedly built a pair of Web services that connect it with the existing TLS B2B solution

RailCo's Initial Web Services

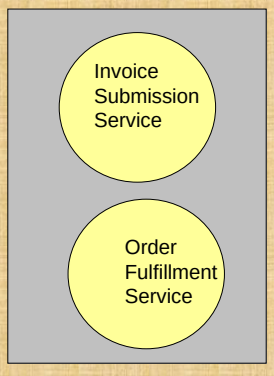


Figure 2.1: RailCo's initial set of Web services, designed only to allow RailCo to connect to TLS's B2B solution.

Note:

- Further RailCo. Realizes that it must also seek new clients to make up lost.
- This new req. ends up also affecting the design of its SOA

Type of Ecommerce

- **B2B (Business-to-Business)**
  Companies doing business with each other such as manufacturers selling to distributors and wholesalers selling to retailers. Pricing is based on quantity of order and is often negotiable.
- **B2C (Business-to-Consumer)**
  Businesses selling to the general public typically through catalogs utilizing shopping cart software. By dollar volume, B2B takes the prize, however B2C is really what the average Joe has in mind with regards to ecommerce as a whole.
- **C2B (Consumer-to-Business)**
  A consumer posts his project with a set budget online and within hours companies review the consumer's requirements and bid on the project. The consumer reviews the bids and selects the company that will complete the project. Elance empowers consumers around the world by providing the meeting ground and platform for such transactions.
- **C2C (Consumer-to-Consumer)**
  There are many sites offering free classifieds, auctions, and forums where individuals can buy and sell thanks to online payment systems like PayPal where people can send and receive money online with ease. eBay's auction service is a great example of where person-to-person transactions take place everyday since 1995.

Transit Line Systems

- Prominent corporation in private transit sector
- Employs over 1800 people, offices in 4 cities
- Primary Line of Business: providing private railroad transit
- Secondary LOB:
  - Maintenance and Repair, outsourcing technicians
  - Parts manufacturing for other businesses
  - Tourism branch that partners with Airlines and Hotels

Transit Line Systems - History

- Existed as a mid-size corporation centered around a single private railway
- The bankruptcy of a competitor sparked an era of growth and acquired two other private railways
- Over 10 years a made a series of acquisitions:
  - G&R Tracks, Ltd. – private railway.

Transit Line Systems - History

- Over 10 years a made a series of acquisitions:
  - G&R Tracks, Ltd. – private railway, provided TLS with enough rail assets to launch a new railway in a neighboring city. Reduced G&R employees from 180 to 10
  - Sampson Steel, Corp.- manufacturing plant that produced parts for clients in automobile and airline industries. TLS acquired Sampson when it was on the brink of liquidation. TLS partners with Sampson who now produces parts and assembly for TLS. Sampson management still runs the company

Transit Line Systems – Technical Infrastructure

- Large IT department of 200 professionals. 50% of staff are contractors hired on a per-project basis
- Contemporay eBusiness solutions are hosted in a clustered server environment
- High transaction volume, robust backup and disaster recovery
- Legacy systems are isolated. Mainframes and Windows servers house older, specialized client-server applications and databases

Transit Line Systems – Automation Solutions

- Legacy systems
  - **Distributed enterprise accounting solution** for 400 users. Partially custom developed, but mostly out-of-the-box accounting features. Sophisticated front-ends for intranet and remote access. Vendor offers Web service adapters
  - **Third party time tracking system** used by employees that are outsourced to record time spent at client sites. Information is manually entered into accounts receivable

Transit Line Systems – Business goals and obstacles

- TLS has undergone much change.
- The IT department has dealt with a volatile business model and regular additions to its solutions.
- Riddled with custom developed applications and third-party products
- Cost of business automation has skyrocketed

Transit Line Systems – Business goals and obstacles

- Integration of systems is complex
- Maintenance of solutions is expensive
- Complexity of systems and inflexibility have slowed IT response time to business process changes
- IT directors have decided to adopt SOA as the standard architecture for new solutions
- SOA will be used to unite legacy systems

Transit Line Systems – Business goals and obstacles

- TLS has already built its first SOA solution Fig-2.2
- B2B system to which RailCo and other vendors connect to conduct transactions online

TLS – B2B Solution

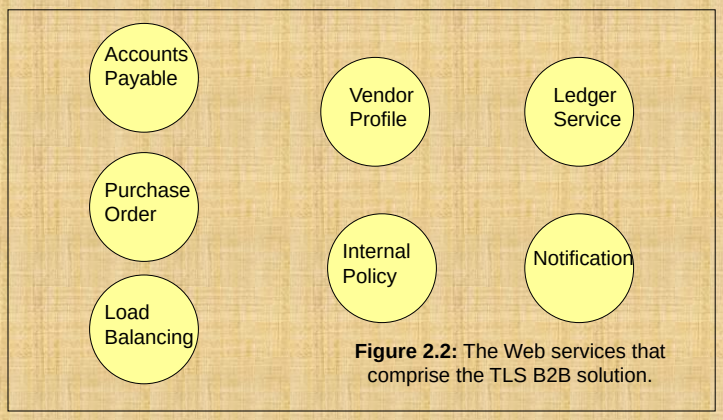


Figure 2.2: The Web services that comprise the TLS B2B solution.

SOA

Chapter 3

Introducing SOA

Fundamental SOA

- A distinct approach for separating concerns
- Logic for large problems are decomposed into smaller, related pieces
- What distinguishes SOA in separating concerns is the manner in which it achieves separation
- SOA is a term that represents a model in which automation logic is decomposed into smaller. Distinct units of logic.

Fundamental SOA

- Individual units of logic can be distributed
- Why is **Service Oriented** separation different?
  - Units exist autonomously, but not isolated
  - Units maintain a degree of commonality and standardization but can evolve independently
  - Units of logic are known as services

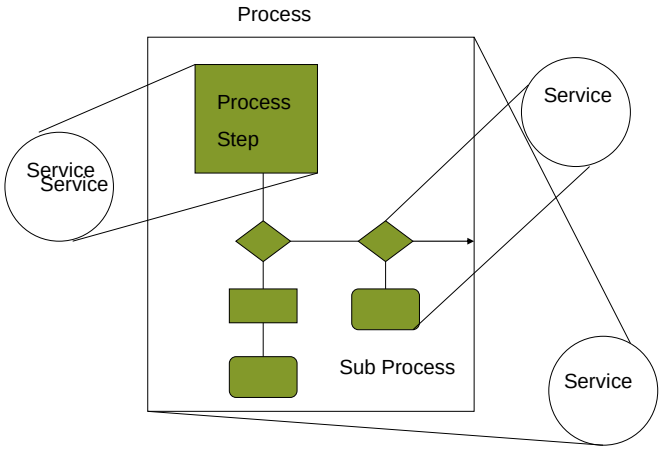
Services Encapsulate Business Logic

- Services encapsulate logic within a distinct context
- This context is directly related to a business task, business entity, or other logical grouping
- Service task can be small or large
- Service logic can encompass the logic provided by other services

Services Encapsulate Business Logic

- Services can be composed to produce other services
- Business processes drive the design
- Services execute in predefined sequences that match business logic and runtime conditions

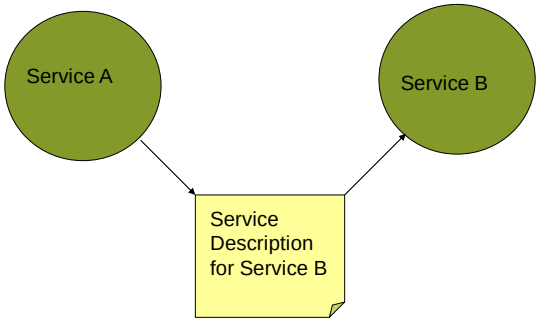
Services Encapsulate Logic



How Services Relate

- In order for services to interact, they must be aware of each other
- Awareness is achieved through service descriptions
- Description: name, location, data exchange requirements
- Services are loosely coupled through service descriptions

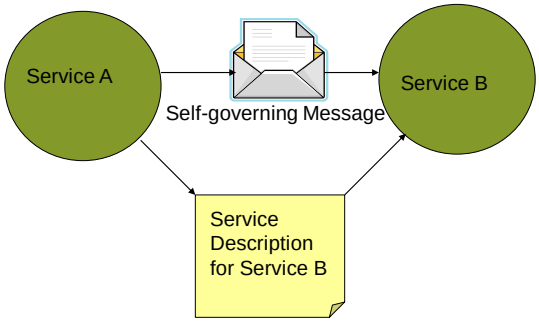
How Services Relate



How Services Communicate

- Services need to cooperate and so they must communicate
- One communication framework is messaging
- Messages, like services, are autonomous
- After a message is sent, the service loses control of what happens to the message
- Messages carry enough intelligence to self-govern their parts of processing logic

How Services Communicate



SO concept

- Service that provides service description and communication via message from a basic architecture.
- This is similar to distributed architecture that supports messaging and a separation of interface from processing logic.
- What distinguishes ours is how its three core components (Service, description and messages) are design.
- This is where service orientation comes in.

How Services Are Designed

- Like Object-orientation, service-orientation is a distinct design approach with a set of design principles
- Application of principles results in a standardized service-oriented processing logic
- A solution comprised of units of service-oriented processing logic is said to be a service-oriented solution

How services are Designed?

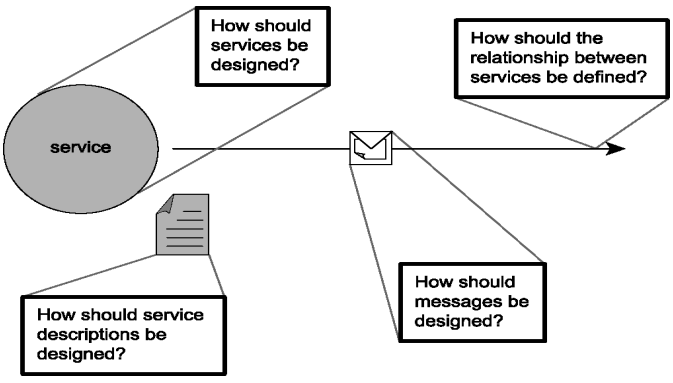
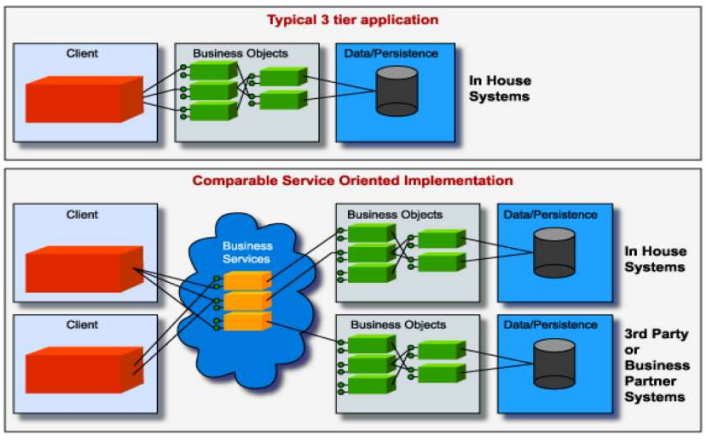


Figure : Service-orientation principles address design issues.

3-Tier vs. Service Oriented Implementation



Principle of SOA

- **Loose coupling :** Services maintain a relationship that minimizes dependencies and only require that they retain an awareness of each other.
- **Service Contract :** Services adhere to communication agreement, as defined collectively by one or more service description and related documents.
- **Autonomy:** Services have control over the logic they encapsulate.
- **Abstraction:** Beyond what is described I service contract, services hide logic form the outside world.
- **Reusability:** Logic is divided into services with the intention of promoting reuse.
- **Composability:** Collection of services can be coordinated and assembled to form composite services.
- **Statelessness:** Services minimize retaining information specific to an activity.
- **Discoverability:** Services are designed to be outwardly descriptive that they can be found and assessed via available discovery mechanism.

SOA Design Principles

- **Loose Coupling** – Services maintain a relationship that minimizes dependencies.

Three types of coupling are considered

1. **Technology coupling**, which is related to how much a service depends on particular technology, product and development platform. A general guideline here is to avoid relying on features that are proprietary to a specific vendor, and to use open technology.
2. **Interface coupling**, which refers to the coupling between service requesters and service providers, that is, dependencies that the service provider imposes on the requester. The requester should not require any information about the internals of the services.

SOA Design Principles

3. **Process Coupling,**

which refers to how much the service is tied to a particular process. Ideally, the service can be reused across many different processes and applications.

SOA Design Principles

- **Service Contract** – Services adhere to a communications agreement defined by one or more service descriptions and related documents.

This is applied to three areas of service contracts:

-**Data model standardization** : standardized data models are used to avoid transformation and improve interoperability,

–**Policy standardization** : terms of usage for a service are expressed in a consistent manner using standardized policy expressions that are based on industry standard vocabularies,

SOA Design Principles

-**Functional expression standardization** : Service's operations, input and output message names and their corresponding type names are defined using standardized naming conventions.

SOA Design Principles

Autonomy -Services have control over the logic they encapsulate.

Services have control over their logic and can execute their functionality without relying on external resources. Design-time as well as runtime autonomies are taken into account.

SOA Design Principles

- **Abstraction** –To the outside, world services only expose what is described in the service contract. Four types of abstractions are taken into account:
  1. Functional abstraction,
  2. Logic abstraction,
  3. Technology information abstraction, and
  4. Quality of service abstraction.

SOA Design Principles

- **Reusability:** Services are designed in a manner so that their solution logic is independent of any particular business process or technology thus promoting reuse.

  Note that a common pitfall is to create overly complex service logic with extra capabilities that will suit possible future service usage scenarios instead of reusable service's core logic.

SOA Design Principles

- **Composability** – Collections of services can be coordinated and assembled to form composite services

SOA Design Principles

- **Statelessness** –In computing, a **stateless protocol** is a communications protocol that treats each request as an independent transaction that is unrelated to any previous request so that the communication consists of independent pairs of requests and responses.
  - SOA services neither remember the last thing they were asked to do nor do they care what the next is.

  Services are not dependent on the context or state of other services, only on their functionality.

  Exa….Talking on the telephone is stateful, whereas posting a letter is stateless.

SOA Design Principles

- **Discoverability** – Services are designed to be outwardly descriptive so they can be found and accessed via discovery mechanisms

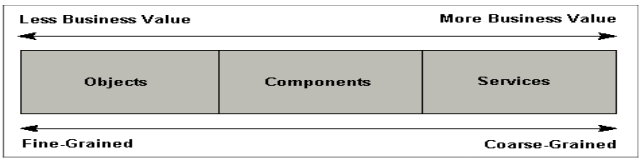
SOA Design Principles

- **Granularity** :Services are designed in a way which provides optimal scope of functionality.
  - The key design factors are:
    - performance,
    - message size,
    - transaction, and
    - business functionality

Coarse-grained systems consist of fewer, larger components than **fine-grained** systems

Granularity is the extent to which a system is broken down into small parts, either the system itself or its description or observation. **Coarse vs fine grained**

Degree of Granularity



- Fine-grained services might be services that provide a small amount of business-process usefulness, such as basic data access.
- Slightly more coarse-grained services might provide rudimentary operations, which are valuable to system experts, but are not of much value to a business-process expert.

How Services Are Built

- **The arrival of Web service technology has promoted SOA**
- **All major vendor platforms support SOA solutions with the understanding that SOA support is based on Web services**
- **SOA can be achieved without Web services, but the book's focus uses Web service technology**

SOA can be implemented using following technologies

- SOAP,
- CORBA
- RPC (remote procedure call)
- WCF (Windows communication foundation),
- REST (Representational state transfer)
- Web services, --- In this course

Primitive SOA

- The **previously** discussed ideas comprise "primitive SOA"
- These ideas represent a baseline technology architecture supported by all major vendors
- All other forms of SOA are based on extending primitive SOA

Case Study

- RailCo accounting solution exists as a **two-tier client-server** application
- Most of the application logic resides within an executable deployed on client workstations
- Two primary tasks:
  - Enter Customer Purchase Order
  - Create Customer Order

Case Study

- In an SO business model, the logic behind each process would be partitioned into one or more services
- The entire process might be a composition of multiple smaller services
- Create Customer Order Process might look like this:
  - Retrieve Purchase Order Data
  - Check Inventory Availability
  - Generate Back Order
  - Publish Customer Order

Case Study

- A business process can be viewed and modeled as a service
- One or more processes can be combined to form an even larger process
- **Create Customer Order** and **Generate Customer Invoice** could be combined to form **Order Processing Process**

Case Study

We need a technical architecture with the following :

- The ability to partition business automation logic into units represented by services
- The ability for these units to be relatively independent so that they can participate in different compositions
- The ability for these units to communicate with each other in a way that preserves unit independence

Contemporary SOA …

- Fosters inherent reusability
  - Service-oriented design principles encourage reuse of software
  - Services can be composed into larger services which in turn can be reused
  - **Services are agnostic** in regard to business processes and automation solutions

1. Agnostic, in an information technology (IT) context, refers to something that is **generalized** so that it is interoperable among various systems. The term can refer not only to software and hardware, but also to business processes or practices.

2. Agnostic services are **not aware of the context in which they are being called**, nor are they aware of how the service is implemented, which platform, technology etc.

Some examples of agnosticism in IT:

- **Platform-agnostic** software runs on any combination of operating system and underlying processor architecture. Such applications are sometimes referred to as "cross-platform."
- **Device-agnostic** software operates across various types of devices, including desktop computers, laptops, tablet PCs and smartphones .
- **Database-agnostic** software functions with any vendor's database management system (DBMS). Typical database-agnostic products include business analytics (BA) and enterprise resource planning (ERP) software.

Contemporary SOA

- The basic ideas in SOA are continually expanded upon by vendors
- Current platforms include powerful XML and Web services support
- This includes new Web services specifications
- **We refer to this extended variation of SOA as Contemporary SOA**

Contemporary SOA Is …

- At the **core of the service-oriented platform**
  - The term SOA has come to have several meanings including a new computing platform as well as an architectural approach
  - This book includes the notion of SOA as a contemporary service-oriented platform

Contemporary SOA …

- **Increases quality of service** but there is more yet to be done in this area
  - Tasks need to be carried out in a secure manner, protecting the contents of messages
  - Tasks to need to be carried out reliably so that message delivery or notification of failed delivery is guaranteed
  - Performance needs to be enhanced for SOAP and XML processing
  - Transaction processing enhancement for task failure

Contemporary SOA Is …

- **Fundamentally autonomous**
  - Individual services need to be as independent and self-contained as possible
  - This is realized through message-level autonomy
  - Messages are "intelligence-heavy" and control the way they are processed by recipients
  - Application that are comprised of autonomous services can be viewed a composite, self-reliant services

Contemporary SOA Is …

Based on open standards

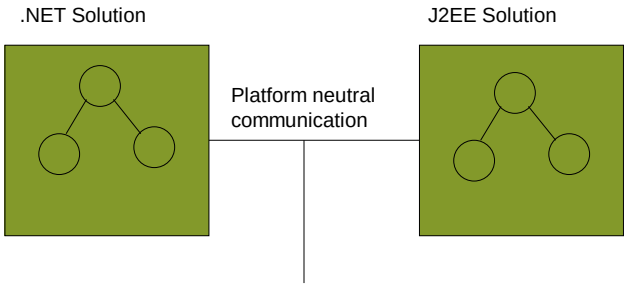
- Messages travel between services via a set of protocols that is globally standardized and accepted
- Message format is standardized, too.
- SOAP, WSDL, XML, and XML Schema allow messages to be fully self-contained
- For services to communicate, they only need to know of each other's service description. This supports loose-coupling
- This limits the role of proprietary technology

Contemporary SOA …

- Supports vendor diversity
  - The communications framework bridges the heterogeneity within and between corporations
  - Any development environment that supports web services can be used to create a non-proprietary service interface layer
  - Integration technologies encapsulate legacy logic through service adapters

Contemporary SOA …

- Supports vendor diversity



Contemporary SOA …

- Promotes discovery
  - Universal Description Discovery and Integration (UDDI) provided for service registries
  - Few early SOA systems used UDDI

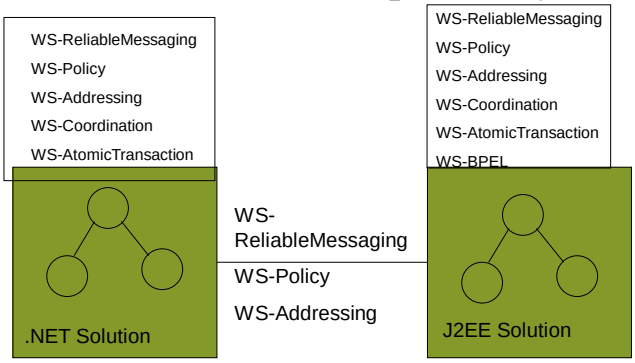
Contemporary SOA …

- Promotes **federation**
  - Establish SOA in an enterprise doesn't require replacement of what you have
  - SOA helps establish unity across non-federated environments
  - Legacy logic is exposed via a common, open, and standardized communication network

Contemporary SOA …

- **Promotes architectural composability**
  - Supports the automation of flexible, adaptable business process by composing loosely coupled services
  - Web service framework is evolving with the release of numerous WS-* specifications that can be composed
  - WS-* specification leverage SOAP messaging

Architectural Composability



Different solutions can be composed of different extensions and can be continue to interoperate as long as they support the common extensions required

Contemporary SOA fosters inherent reusability

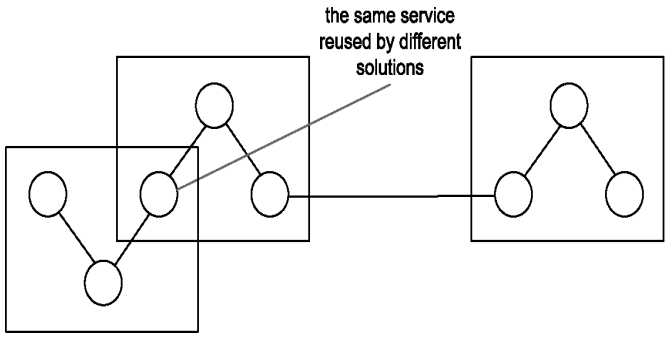
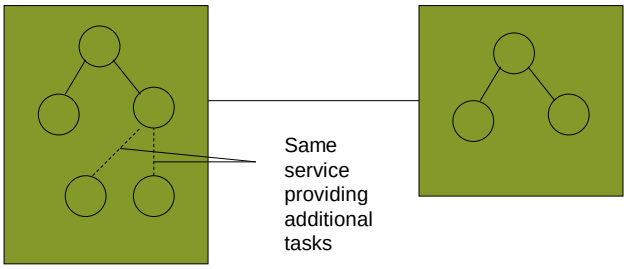


Figure : Inherent reuse accommodates unforeseen reuse opportunities.

Contemporary SOA …

- Emphasizes extensibility
  - When encapsulating functionality through a service description, you are encouraged to think beyond a point-to-point solution
  - With appropriate granularity, the scope of a service can be extended without breaking an established interface

Extensibility

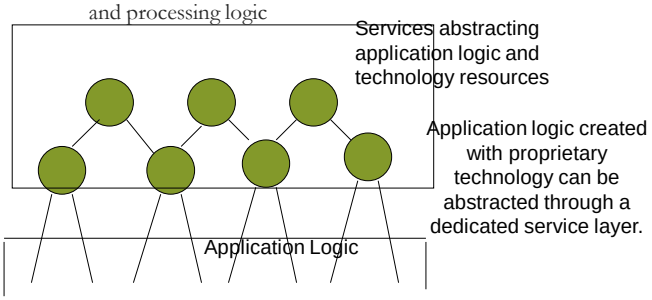


Contemporary SOA …

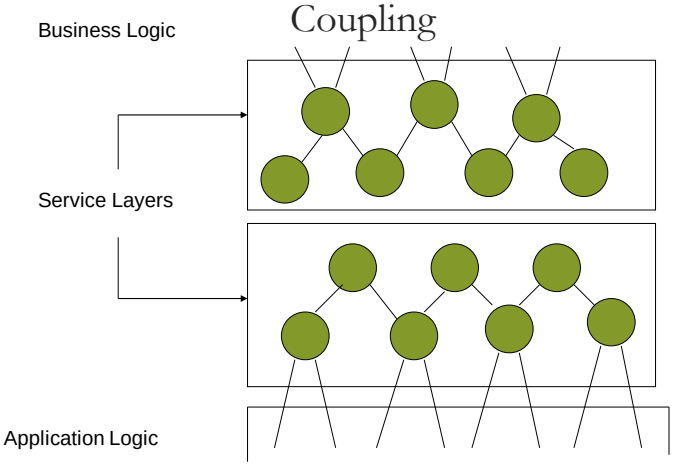
- Supports a service-oriented business modeling paradigm
  - Business processes are modeled with services and cut vertically through business logic
  - BPM models, entity models and other forms of business intelligence can be accurately represented through coordinated composition of business-centric services

Contemporary SOA …

- Implements layers of abstraction
  - SOAs introduce layers of abstraction by positioning services as the sole access points to a variety of resources and processing logic



Contemporary SOA Promotes Loose Coupling



Contemporary SOA …

- Promotes organizational agility
  - High dependency between parts of an enterprise means that changing software is more complicated and expensive
  - Leveraging service business representation, service abstraction, and loose coupling promotes agility

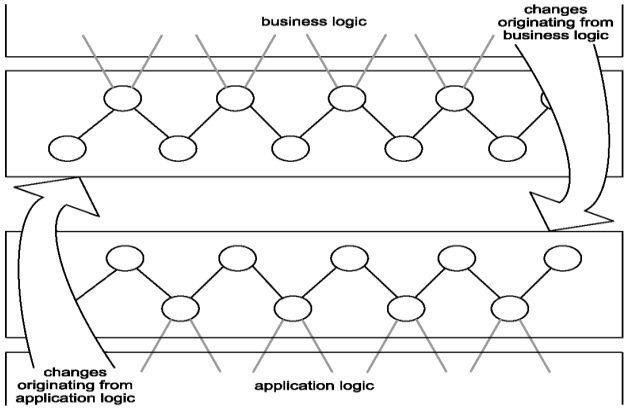


Figure: A loosely coupled relationship between business and application technology allows each end to more efficiently respond to changes in the other.

Contemporary SOA …

- Is a building block
  - Services are composed into larger services
  - Multiple SOA applications can be pulled into service-oriented integration technologies to help build a Service-Oriented Enterprise (SOE)

Appending the Definition

- SOA can establish an abstraction of business logic and technology, resulting in a loose coupling between these domains. These changes foster service-orientation in support of a service-oriented enterprise

Contemporary SOA …

- Is an evolution
  - SOA is a distinct architecture from its predecessors
  - Different from client-server technology in that it is influenced by concepts in service-orientation and Web services
  - Promotes reuse, encapsulation, componentization, and distribution of application logic like previous technologies

Contemporary SOA …

- Is still maturing
  - Standards organization and vendors are continuing to develop new SOA technologies

Contemporary SOA …

- Is an achievable ideal
  - Moving an enterprise toward SOA is a difficult and enormous task
  - Many organizations begin with a single application and then begin leveraging services into other applications
  - Changing to SOA requires cultural changes in an organization

SOA Definition

- Contemporary SOA represents an open, extensible, federated, composable architecture that promotes service-orientation and is comprised of autonomous, QoS-capable, vendor diverse, interoperable, discoverable, and potentially reusable services, implemented as Web services

Misconceptions about SOA

- An application that uses Web services is service-oriented
- SOA is just a marketing term used to re-brand distributed computing with Web services
- SOA simplifies distributed computing
- An application with Web services that uses WS-* extensions is service-oriented

Misconceptions about SOA

- If you understand Web services you won't have a problem building SOA
- Once you go SOA, everything becomes interoperable

Benefits of SOA

- Improved integration and intrinsic interoperability
- Inherent reuse
- Streamlined architectures and solutions
- Leveraging of legacy investment
- Establishing standardized XML data representation
- Focused investment on communication infrastructure
- Best of breed alternatives
- Organizational agility

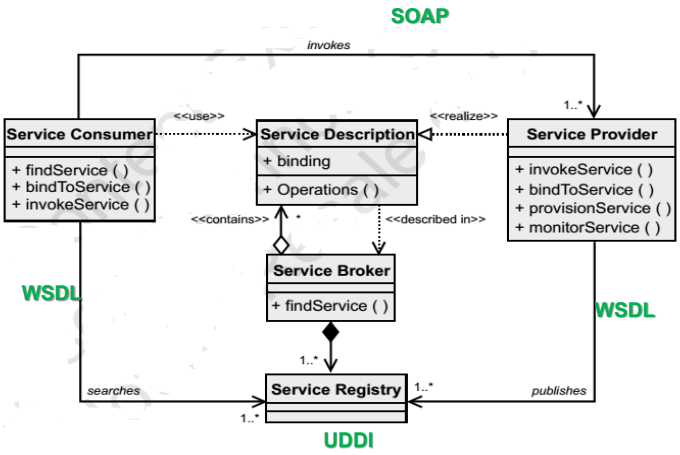
Pitfalls of adopting SOA

- Building service-oriented architectures like traditional distributed architectures
- Proliferation of RPC-style service descriptions (increased volumes of fine-grained message exchanges)
- Inhibiting the adoption of features provide by WS-* specifications
- Improper partitioning of functional boundaries within services
- Creation of non-composable services
- Entrenchment of synchronous communications
- Creation of non-standardized services

Pitfalls of adopting SOA

- Not standardizing SOA in the enterprise
- Not creating a transition plan
- Not starting with an XML foundation architecture
- Not understanding SOA performance requirements
- Not understanding Web services security
- Not keeping in touch with product platforms and standards development

SOA Architecture



SOA

CHAPTER 4
THE EVOLUTION OF SOA

An SOA Timeline XML to Web Services to SOA

What is XML?

- XML stands for EXtensible Markup Language
- XML is a **markup language** much like HTML
- XML was designed to carry data, not to display data
- XML is designed to be self-descriptive
- HTML was designed to display data.

- XML is derived from SGML- **Standard Generalized Markup language.**
- Generalized markup is based on two novel postulates
  - Markup should be declarative: it should describe a document's structure and other attributes, rather than specify the processing to be performed on it.
  - Markup should be rigorous so that the techniques available for processing rigorously-defined objects like programs and database can be used for processing documents as well.
- XML is designed to expose the structure of a document for processing
- XML gained in popularity in the e-Business movement of the 90's

XML

- XML allowed developers to attach meaning and context to a document that was transmitted across internet protocols
- XML was used as the basis for additional standards including XSD (schema definition) and XSLT (transformation language)
- XML data representation is the foundation layer of SOA
- XML establishes the structure of messages traveling between services
- XSD schemas preserve the integrity and validity of message data
- XSLT is used to enable communication between disparate data representations by schema mapping

How Can XML be Used?

1. **XML Separates Data from HTML**
   - If you need to display dynamic data in your HTML document, it will take a lot of work to edit the HTML each time the data changes.
   - With a few lines of **JavaScript** code, you can read an external XML file and update the data content of your web page.

XML Simplifies Data Sharing

- In the real world, computer systems and databases contain data in incompatible formats.
- XML data is stored in plain text format. This provides a software- and hardware-independent way of storing data.
- This makes it much easier to create data that can be shared by different applications.

XML Simplifies Data Transport

- One of the most time-consuming challenges for developers is to exchange data between incompatible systems over the Internet.
- Exchanging data as XML greatly reduces this complexity, since the data can be read by different incompatible applications.

XML Simplifies Platform Changes

- Upgrading to new systems (hardware or software platforms), is always time consuming. Large amounts of data must be converted and incompatible data is often lost.
- XML data is stored in text format. This makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

XML Namespaces

- XML Namespaces provide a method to avoid element name conflicts.
- In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications.
- The namespace is defined by the **xmlns attribute** in the start tag of an element..

Namespaces in Real Use

- XSLT is an XML language that can be used to transform XML documents into other formats, like HTML.
- In the XSLT document below, you can see that most of the tags are HTML tags.
- The tags that are not HTML tags have the prefix xsl, identified by the namespace xmlns:xsl="http://www.w3.org/1999/XSL/Transform":

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<body>
  <h2>My CD Collection</h2>
  <table border="1">
    <tr>
      <th style="text-align:left">Title</th>
      <th style="text-align:left">Artist</th>
    </tr>
    <xsl:for-each select="catalog/cd">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="artist"/></td>
    </tr>
    </xsl:for-each>
  </table>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```


XML Encoding

- XML documents can contain international characters, like Norwegian æøå, or French êèé.
- To avoid errors, you schould specify the encoding used, or save your XML files as UTF-8.
- **Character Encoding**
- Character encoding defines a unique binary code for each different character used in a document.
- In computer terms, character encoding are also called character set, character map, codeset, and code page.

Unicode

- Unicode is an industry standard for character encoding of text documents. It defines (nearly) every possible international character by a name and a number.
- Unicode has two variants: UTF-8 and UTF-16.
- UTF = **U**niversal character set **T**ransformation **F**ormat.
- UTF-8 uses a single byte (8-bits) to represent commonly used characters and two (or three) bytes for the rest.
- UTF-16 uses two bytes (16 bits) for most characters, and three bytes for the rest.

XML Encoding

- The first line in an XML document is called the **prolog**
- <?xml version="1.0"?>
- The prolog is optional. Normally it contains the XML version number. It can also contain information about the encoding used in the document. This prolog specifies UTF-8 encoding:
- **<?xml version="1.0" encoding="UTF-8"?>**

Web Services

- W3C receives Simple Object Access Protocol (SOAP) in 2003
- SOAP messages now represent the communications layer for SOA
- This provided a proprietary-free Internet communications layer
- This led to the idea of creating a pure, Web-based, distributed technology – Web services
- This helped bridge the enormous disparity between and within organizations

Web Services

- The most important part of a Web service is its public interface
- This is the central piece of information that assigns a service an identity and enables its invocation
- WSDL (Web Services Description Language) was one of the first initiatives in support of Web services
- WSDL was submitted to W3C in 2001

Web Services

- Web services required an Internet-friendly and XML-compliant communications format
- Other alternatives were considered (XML-RPC), but SOAP was the winning technology
- W3C responded by releasing newer versions of SOAP that allow RPC-style and document-style messages

UDDI

- Universal Description, Discovery and Integration
- Submitted to OASIS (Organization for the Advancement of Structured Information Standards)
- Allows for the creation of standardized service description registries
- Services can be registered in a central location and discovered by service requestors
- Unlike WSDL and SOAP, UDDI hasn't yet attained industry-wide acceptance
- Remains an optional extension to SOA

Custom Web Services

- Custom services were developed to accommodate specialized business requirements
- Existing messaging platforms (Message Oriented Middleware – MOM) incorporated services to support SOAP
- Some organizations incorporated Web services for B2B data exchange – replacing EDI (Electronic Data Interchange)

Brief History of SOA

- It became clear that Web services could be the basis for a separate architectural platform
- Early SOA is modeled around three basic components:

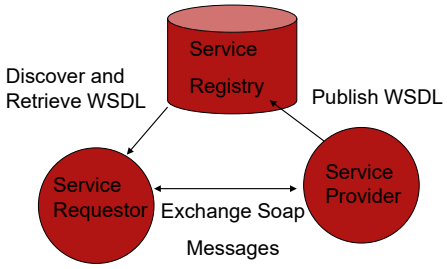


Fig: An early incarnation of SOA

SOA Extensions

- This early model has been extended by WS-* specifications
- The goal was to elevate Web services technology to an enterprise level
- There was also an interest in applying service-oriented concepts to business analysis
- This vision was supported by the rise of business process definition languages – WS-BPEL

Retrofitting Issues while SOA reshaping XML and web-services

- Data Representation and Service Modeling standards must be aligned
- SOAP is used for all inter-service communication. XML documents and XSD schemas are modeled with SOAP in mind
- Document-style messaging is standard. Changing from RPC-style imposes changing on services
- SOAP promotes a content and intelligence-heavy message model. This supports statelessness, autonomy and minimizes message sending

Standard Organizations contributing to SOA

- **The World Wide Web Consortium (W3C)**
  - Founded by Tm Berners-Lee in 1994
  - Began with HTML
  - XML, XML Schema, XSLT, SOAP, WSDL, and Web Services
  - Formal and rigorous with many public reviews
  - Two to three years for standards to be adopted

Standard Organizations contributing to SOA

- **Organization for the Advancement of Structured Information Standards (OASIS)**
  - Created in 1993 as SGML Open
  - Renamed when their scope changed from SGML to XML standards
  - Over 600 organizations follows it.
  - WS-BPEL, ebXML, contributions to UDDI, SAML (security), XACML
  - Focuses on core, industry-agonostic standards, leveraging standards to support vertical industries
  - Shorter development times than W3C

Standard Organizations contributing to SOA

- **The Web Services Interoperability Organization (WS-I)**
  - Does not create new standards but ensures the goal of open interoperability
  - 200 organizations, all major SOA vendors
  - Basic Profile – a recommendation-based document that establishes which available standards form the most desirable interoperability architecture
  - Positions specific versions of WSDL, SOAP, UDDI, XML, XML Schema
  - Basic Security Profile

Some Major Vendors

- Microsoft, IBM, BEA Systems, Sun Microsystems, Oracle, Tibco, Hewlett-Packard, Canon, Commerce One, Fujitsu, Software AG, Nortel, Verisign, WebMethods

Vendor Influence

- Each vendor has its own vision for SOA
- IBM uses WebSphere
- Microsoft supports .NET and its operating system
- Vendors try to influence decisions using proprietary designs

Vendor Alliances

- Vendors form loose alliances for common goals
- IBM, Microsoft have collaborated on several WS-* extensions
- OASIS supported WS-Reliable Messaging. Microsoft and IBM announced their own specification – WS-Reliability.

Roots of SOA

- With the rise of multi-tier applications, the variations with which applications could be delivered began to increase
- A standardized definition for a baseline application becomes important
- The definition is abstract but describes the technology, boundaries, rules, limitations and design characteristics that apply to all solutions – an **application architecture**

Application Architecture

- An application architecture is a blueprint
- Different levels can be specified, depending on the organization
- Some keep it high level, providing abstract physical and logical representation of the technical blueprint.
- Other include more detailed like data models, communication flow diagrams, security requirements and aspect of infrastructure
- Several application architectures may exist in an enterprise and kept in alignment by an enterprise architecture

Enterprise Architecture

- Enterprise architectures provide high-level views of all forms of heterogeneity
- “Urban plan for a city” and “blueprint of a building” is comparable to enterprise architecture and application architecture.
- Contain a long-term vision of how an organization will evolve its technologies

Service Oriented Architecture

- Spans both enterprise and application architecture domains
- Benefits of SOA are realized when applied across multiple solution environments
- Because SOA is a compostable architecture, a company may have multiple SOAs

SOA vs Client Server

- Mainframes provided the first "client-server" computing with synchronous and asynchronous communication
- 1980s – two-tier client server with fat clients, GUI, database. Dominated the 90s
- In general, any environment in which one piece of software request or receive information from another can be referred to as **Client-server**

Client Server Characteristics

- Bulk of the application logic resides with the client
- Business rules were maintained in stored procedures and triggers on the database
- This abstracted a set of business logic from the client and simplified data access programming
- Overall, the client ran the show

Single-Tier Client-Server Architecture

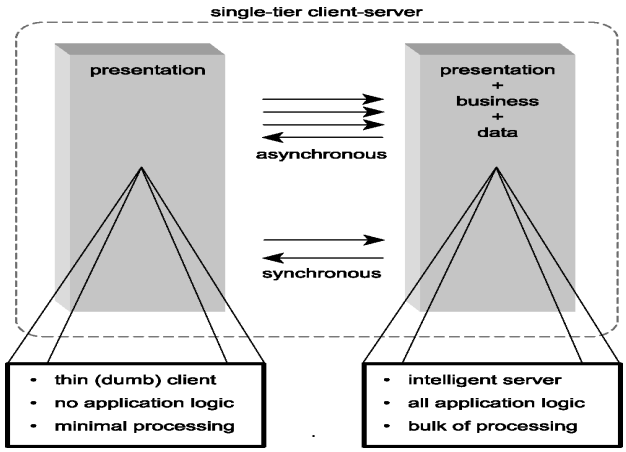
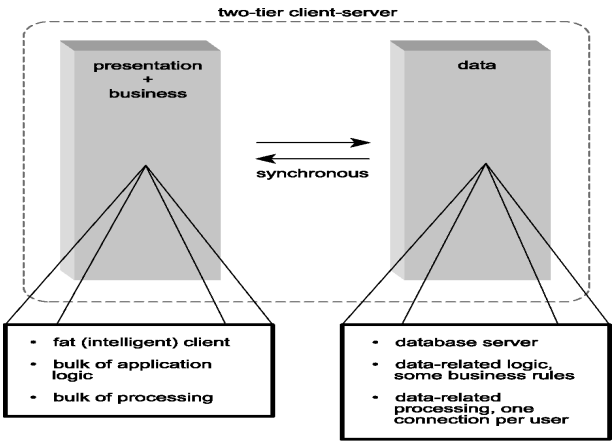


Figure 4.2: A typical single-tier client-server architecture

Figure 4.3: A typical two-tier client-server architecture



Client-Server vs. SOA Application Technology

- The technology set for client-server applications included 4GLs like VB and PowerBuilder, RDBMSs
- The SOA technology set has expanded to include Web technologies (HTML, CSS, HTTP, etc)
- SOA requires the use of XML data representation architecture along with a SOAP messaging framework

Client-Server vs. SOA Application Security

- Centralized at the Server level
- Databases manage user accounts and groups
- Also controlled within the client executable
- Security for SOA is much more complex
- Security complexity is directly related to the degree of security measures required
- Multiple technologies are required, many in WS-Security framework

Client-Server vs. SOA Application Administration

- Significant maintenance costs associated with client-server
- Each client housed application code
- Each update required redistribution
- Client stations were subject to environment-specific problems
- Increased server-side demands on databases

Client-Server vs. SOA Application Administration

- SOA solutions are not immune to client-side maintenance challenges
- Distributed back-end supports scalability, but new admin demands are introduced
- Management of server resources and service interfaces may require new admin tools and even a private registry
- Commitment to services and their maintenance may require cultural change in an organization

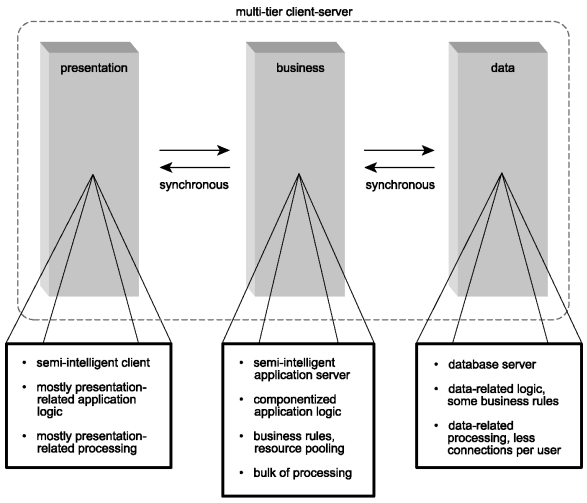


Figure 4.4: A typical multi-tier client-server architecture.

SOA vs Traditional Distributed Internet Architecture

- Multiple client-server architectures have appeared
- Client-server DB connections have been replaced with Remote Procedure Call connections (RPC) using CORBA or DCOM (Distributed Component Object Model)
- Multi-tiered client-server environments began incorporating internet technology in 90s.
- The browser shifted 100% of application logic to the server
- Distributed Internet architecture introduced the Web server as a new physical tier
- HTTP replaced RPC protocols

SOA vs Traditional Distributed Internet Architecture

- Distributed Internet application put all the application logic on the server side
- Entire solution is centralized
- Emphasis is on:
  - How application logic is partitioned
  - Where partitioned units reside
  - How units of logic should interact

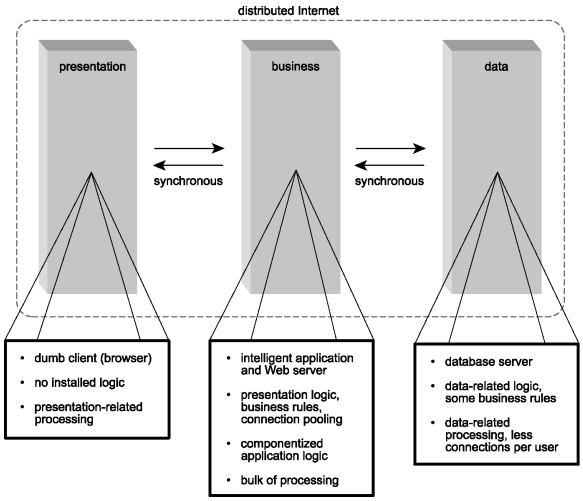


Figure 4.5: A typical distributed Internet architecture.

SOA vs Traditional Distributed Internet Architecture

- Difference lies in the **principles** used to determine the three primary design considerations
- Traditional systems create components that reside on one or more application servers
- Components have varying degrees of functional granularity
- Components on the same server communicate via proprietary APIs.
- RPC protocols are used across servers via proxy stubs

SOA vs Traditional Distributed Internet Architecture

- Actual references to other physical components can be embedded in programming code (tight coupling)
- SOAs also rely on components
- Services encapsulate components
- Services expose specific sets of functionality
- Functionality can originate from legacy systems or other sources

SOA vs Traditional Distributed Internet Architecture

- Functionality is wrapped in services
- Functionality is exposed via open, standardized interface, irrespective of technology providing the solution
- Services exchange information via SOAP messages. SOAP supports RPC-style and document-style messages
- Most applications rely on document-style

SOA vs Traditional Distributed Internet Architecture

- Messages are structured to be self-sufficient
- Messages contain meta information, processing instructions, policy rules
- SOA reuse a deep level concept by promoting **solution-agnostic services**

Application Processing

- Distributed Internet architecture promotes the use of proprietary communication protocols (DCOM, CORBA)
- SOA relies on message-based communication
- Messages use serialization, transmission, de-serialization of SOAP messages containing XML payloads
- RPC communication is faster than SOAP and SOAP processing overhead is a significant design issue

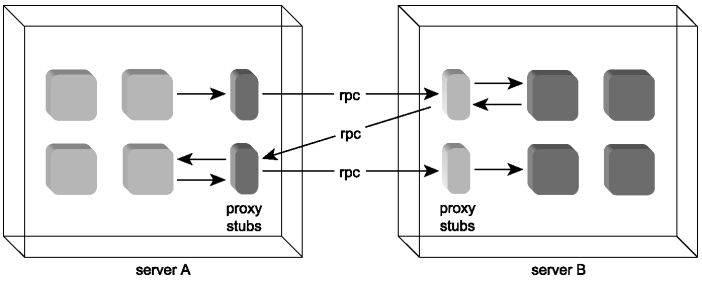


Figure 4.6: Components rely on proxy stubs for remote communication.

Application Processing

- Messaging framework supports a wide range message exchange patterns
- Asynchronous patterns encouraged
- Support for stateless services is supported by context management options (WS-Coordination, WS-BPEL

Technology

- Distributed Internet architecture now includes XML data representation
- XML and Web services are optional for distributed Internet architecture but not for SOA

Security

- When application logic crosses physical boundaries, security becomes more difficult
- Traditional security architectures incorporate delegation and impersonation as well as encryption
- SOAs depart from this model by relying heavily on WS-Security to provide security logic on the messaging level
- SOAP messages carry headers where security logic can be stored

Administration

- Maintaining component-base applications involves:
  - keeping track of individual components
  - tracing local and remote communication problems
  - Monitoring server resource demands
  - Standard database administrative tasks
- Distributed Internet Architecture introduces the Web server and its physical environment

Administration

- SOA requires additional runtime administration:
  - Problems with messaging frameworks
  - Additional administration of a private or public registry of services

Web service as component wrapper

- This introduce an integration layer which consists of wrapper services that enable synchronous communication via SOAP – compliant integration channel.
- They are also used to enable communication with other solutions and take advantages of features offered by third party utility web services.

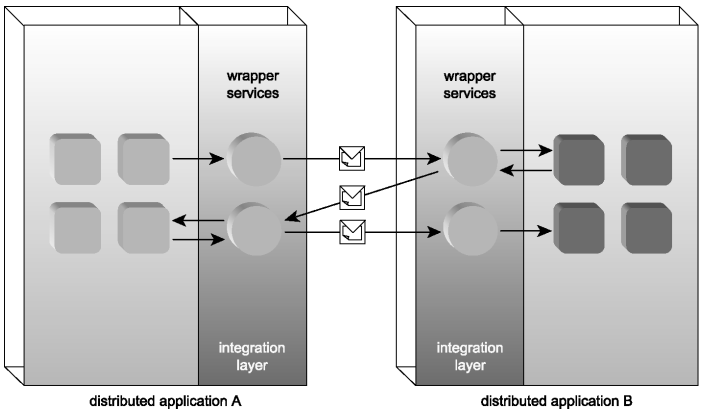


Figure 4.7: Wrapper services encapsulating components.

Service Orientation and Object Orientation

SOA BASED

1. Loose coupling (services)
2. Encourage coarse grained interface (Service Description)
3. Processing logic vary significantly
4. Creates activity agnostic unit of processing
5. Units of processing login designed to be stateless

Object oriented

1. Tight coupling (objects)
2. Fine grained interface
3. Objects are smaller and more specific
4. Binding processing logic with data
5. It provides binding of data and logic thus have stateful units (objects)

THANK YOU

Chapter 5

WEB SERVICES AND PRIMITIVE SOA

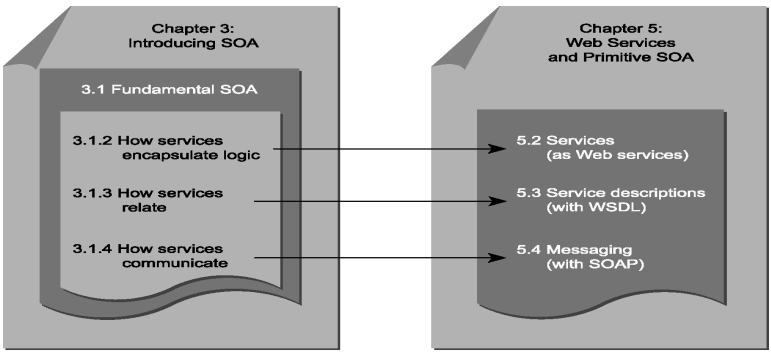


Figure 5.1: The structural relationship between sections in Chapters 3 and 5.

The Web services framework

"A TECHNOLOGY FRAMEWORK IS A COLLECTION OF THINGS. IT CAN INCLUDE ONE OR MORE ARCHITECTURES, TECHNOLOGIES, CONCEPTS, MODELS, AND EVEN SUB-FRAMEWORKS".

SPECIFICALLY, THIS FRAMEWORK IS CHARACTERIZED BY:

1. AN ABSTRACT (VENDOR-NEUTRAL) EXISTENCE DEFINED BY STANDARDS ORGANIZATIONS AND IMPLEMENTED BY (PROPRIETARY) TECHNOLOGY PLATFORMS.
2. CORE BUILDING BLOCKS THAT INCLUDE WEB SERVICES, SERVICE DESCRIPTIONS, AND MESSAGES
3. A COMMUNICATIONS AGREEMENT CENTERED AROUND SERVICE DESCRIPTIONS BASED ON WSDL
4. A MESSAGING FRAMEWORK COMPRISED OF SOAP TECHNOLOGY AND CONCEPTS
5. A SERVICE DESCRIPTION REGISTRATION AND DISCOVERY ARCHITECTURE SOMETIMES REALIZED THROUGH UDDI
6. A WELL-DEFINED ARCHITECTURE THAT SUPPORTS MESSAGING PATTERNS AND COMPOSITIONS (COVERED IN CHAPTER 6)
7. A SECOND GENERATION OF WEB SERVICES EXTENSIONS (ALSO KNOWN AS THE WS-*SPECIFICATIONS) CONTINUALLY BROADENING ITS UNDERLYING FEATURE-SET (COVERED IN CHAPTERS 6 AND 7)

SERVICES (AS WEB SERVICES)

Fundamentally every web services are associated with

- A **temporary classification** based on the roles it assumes during runtime processing of a message.
- **Permanent classification** based on the application logic it provides and the roles it assumes within a solution environment

Service Roles

- A Web service is capable of assuming different roles, depending on the context within which it is used.
- A service is therefore not labeled exclusively as **a client or server**, but instead as a unit of software capable of altering its role, depending on its processing responsibility in a given scenario.
- **Service provider**
  - The Web service is invoked via an external source, such as a service requestor (Figure 5.2).
  - The Web service provides a published service description offering information about its features and behavior
  - The service provider role is synonymous with the server role in the classic client-server architecture.

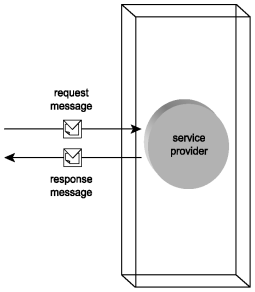


Figure 5.2: As the recipient of a request message, the Web service is classified as a service provider.

Some Terms used in service context.

- **service provider entity** (the organization or individual providing the Web service)
- **service provider agent** (the Web service itself, acting as an agent on behalf of its owner)

SERVICE REQUESTOR

- Any **unit of processing logic** capable of issuing a request message that can be understood by the service provider is classified as a service requestor .
- A Web service is always a service provider but also can act as a service requestor.

Web service takes on the service requestor role under the following circumstances:

- The Web service invokes a service provider by sending it a message
- The Web service searches for and assesses the most suitable service provider by studying available service descriptions. (Service descriptions and service registries are covered in the Service descriptions

service requestor entity *(the organization or individual requesting the Web service)*

service requestor agent *(the Web service itself, acting as an agent on behalf of its owner)*

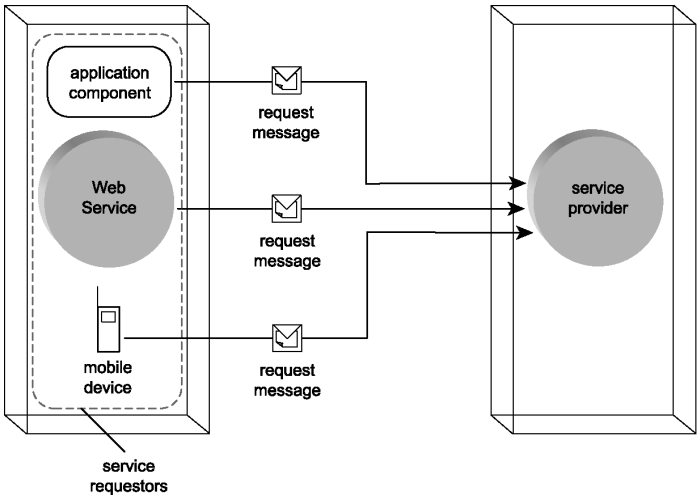


Figure 5.3: The sender of the request message is classified as a service requestor.

- **TLS's** Purchase order service submits electronic POs that are received by **RailCo's** Order fulfillment service
- Upon shipping the order, **RailCo's** Invoice Submission Service sends an electronic invoice to **TLS's** Account Payable Service

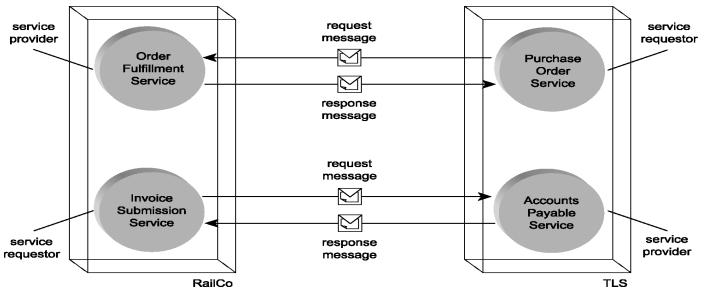


Figure 5.4: TLS and RailCo services swapping roles in different but related message exchanges

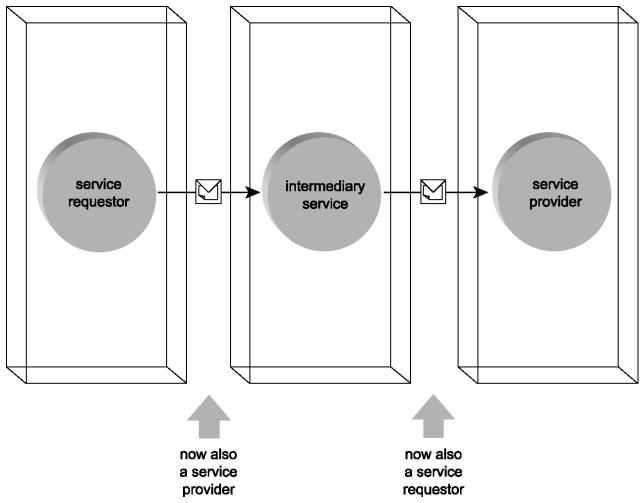


Figure 5.5: The intermediary service transitions through service provider and service requestor roles while processing a message.

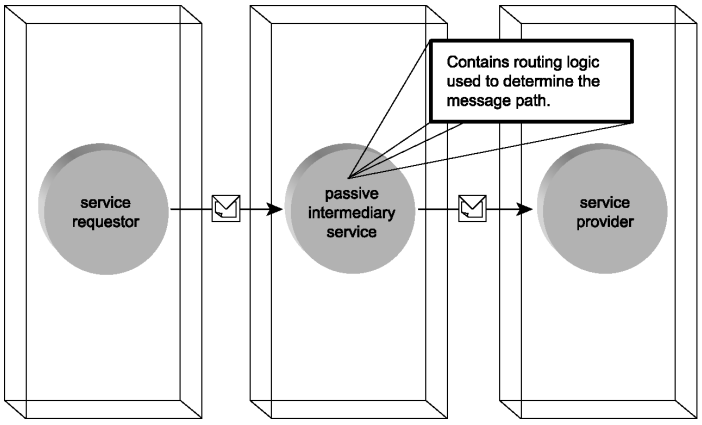


Figure 5.6: A **passive intermediary** service processing a message without altering its contents.

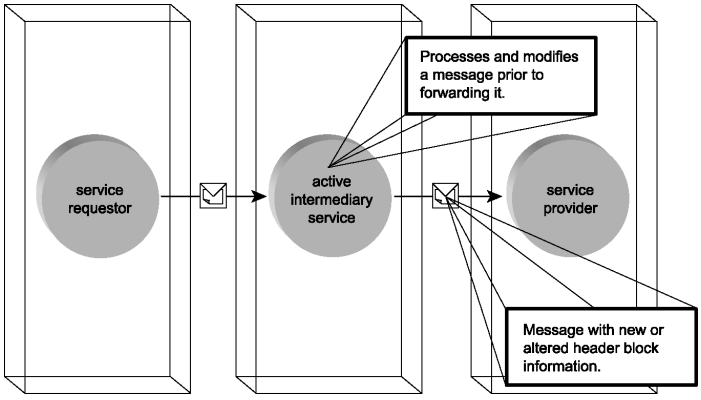


Figure 5.7: An active intermediary service.

Initial Sender and Ultimate Receiver

- Initial sender are simply service requestors that initiate the transmission of message
- Initial sender always the first web service in a message path.
- The counterpart to this role is the ultimate receiver.
- It is the service providers that exist as the last web service along a message paths

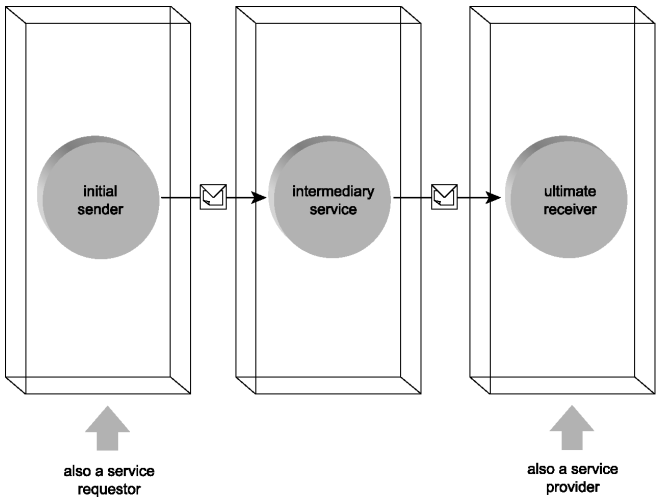


Figure 5.8: Web services acting as initial sender and ultimate receiver.

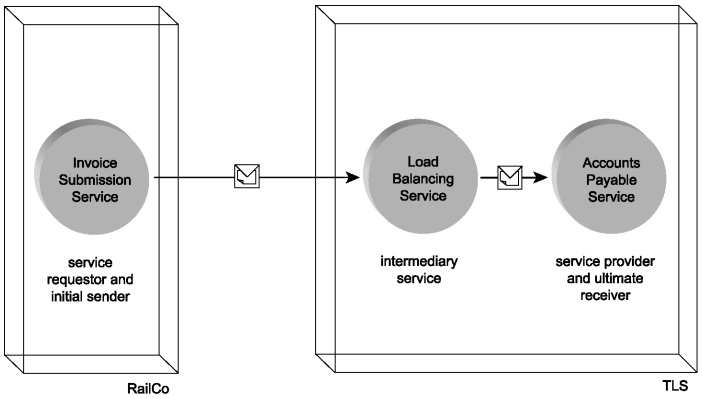


Figure 5.9: The TLS Load Balancing Service acting as an intermediary between the RailCo initial sender and the TLS ultimate receiver.

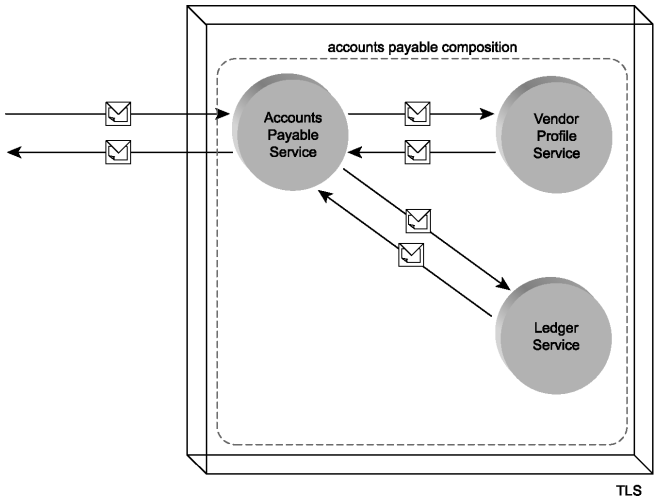


Figure 5.11: The Accounts Payable Service enlisting other TLS services in a service composition.

Utility service model

"Any generic Web service or service agent designed for potential reuse can be classified as a utility service" .

Utility services are used within SOAs as follows:

- as services that enable the characteristic of reuse within SOA
- as solution-agnostic intermediary services
- as the services with the highest degree of autonomy

- Web service need to be designed with service composition in mind to be effective composition members.
- It is very important to SO environment. It is govern by WS*-composition extensions, such as WS-BPEL, which introduce the related concepts of **orchestration and choreography.**

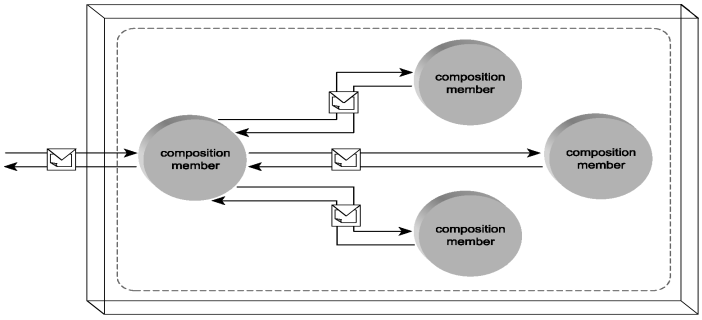


Figure 5.10: A service composition consisting of four members.

SERVICE MODELS

"Type of Service Model"

Business service model,

Within an SOA, the business service represents the most fundamental building block. It encapsulates a distinct set of business logic within a well-defined functional boundary.

It is fully **autonomous** but still not limited to executing in isolation, as business services are frequently expected to participate in service compositions.

Business services are used within SOAs as follows:

- as fundamental building blocks for the representation of business logic
- to represent a corporate entity or information set
- to represent business process logic
- as service composition members

Case Study

In the examples we've gone through so far in this chapter, we've described eight web services. six of these are business services, while the other two are utility services, as follows:

Accounts Payable Service	***= Business Service***
Internal Policy Service	***= Utility Service***
Invoice Submission Service	***= Business Service***
Ledger Service	***= Business Service***
Load Balancing Service	***= Utility Service***
Order Fulfillment Service	***= Business Service***
Purchase Order Service	***= Business Service***
Vendor Profile Service	***= Business Service***

The load balancing and internal policy services are classified as utility services because they provide generic functionality that can be reused by different types of applications.

The application logic of the remaining services is specific to a given business task or solution, which makes them business-centric services.

CONTROLLER SERVICE MODEL

- Service compositions are comprised of a set of independent services that each contribute to the execution of the overall business task.
- The assembly and coordination of these services is often a task in itself and one that can be assigned as the primary function of a dedicated service or as the secondary function of a service that is fully capable of executing a business task independently.
- The controller service fulfills this role, acting as the parent service to service composition members.

Controller services are used within SOAs as follows:

- to support and implement the principle of composability
- to leverage reuse opportunities
- to support autonomy in other services

Note that controller services themselves can become subordinate service composition members. In this case the composition coordinated by a controller is, in its entirety, composed into a larger composition. In this situation there may be a master controller service that acts as the parent to the entire service composition, as well as a sub-controller, responsible for coordinating a portion of the composition (Figure 5.12).

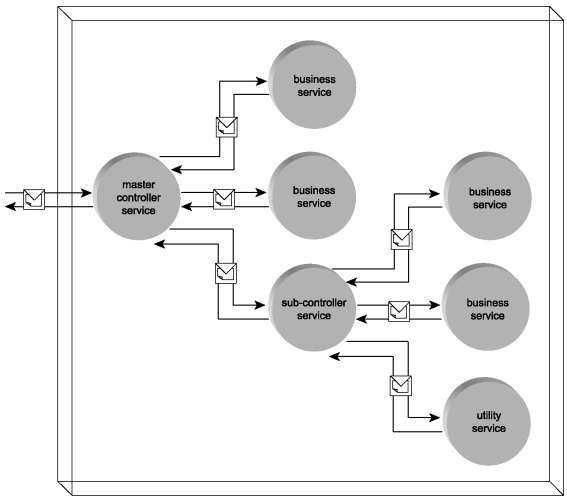


Figure 5.12: A service composition consisting of a master controller, a sub-controller, four business services, and one utility service.

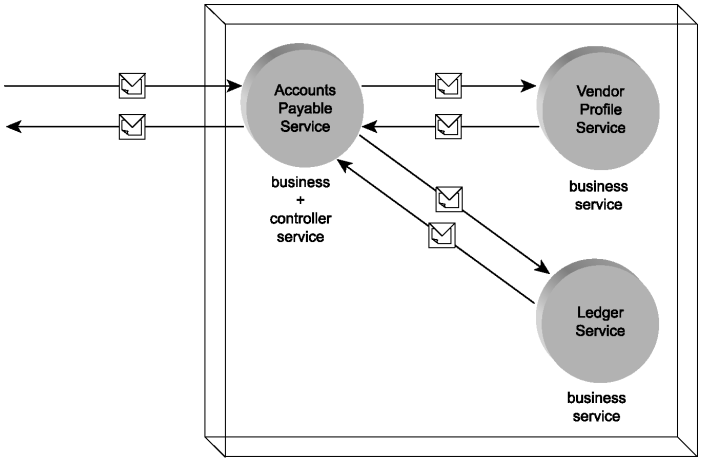


Figure 5.13: The Accounts Payable Service acting as a business and controller service, composing two other business services.

SERVICE DESCRIPTIONS (WITH WSDL)

❖This part of SOA provides the key ingredient to establish a consistently loosely coupled form of communication between services implemented as Web services.

- For this purpose, description documents are required to accompany any service wanting to act as an ultimate receiver. The primary service description document is the WSDL definition (Figure 5.14).

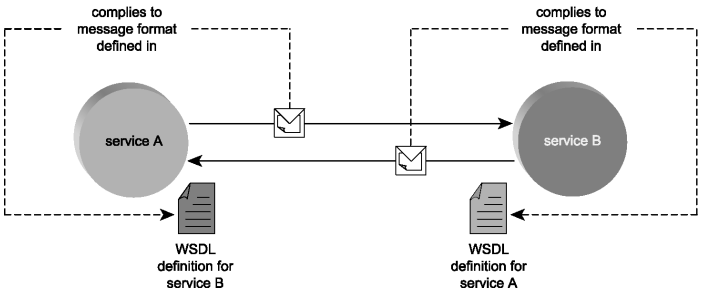


Figure 5.14: WSDL definitions enable loose coupling between services.

CASE STUDY

- For RailCo to design its B2B Web services in full compliance with the TLS services, RailCo acquires the WSDL service description published by TLS for their Accounts Payable Service.

- This definition file then is used by developers to build the **Invoice Submission Service** so that it can process SOAP messages in accordance with the service interface requirements defined in the TLS service descriptions.

- Further, RailCo provides TLS with a copy of the WSDL definition for the RailCo Order Fulfillment Service. TLS registers this service description and adds it to the list of vendor endpoints that will receive electronic purchase orders. (Figure 5.15 illustrates both scenarios.)

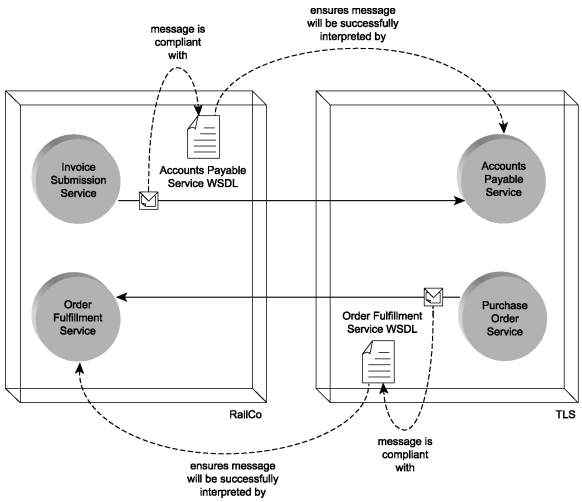


Figure 5.15: Each service requestor is using the WSDL of a service provider to ensure that messages sent will be understood and accepted.

Service endpoints and service descriptions/WSDL service definition:

A WSDL describes the point of contact for a service provider, also known as the service endpoint or just endpoint .
It provides a formal definition of the endpoint interface (so that requestors wishing to communicate with the service provider know exactly how to structure request messages) and also establishes the physical location (address) of the service.

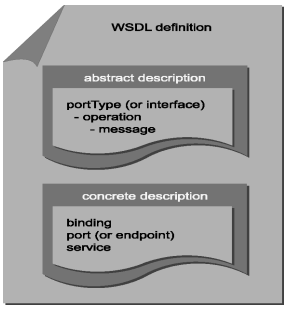


Figure 5.16: WSDL document consisting of abstract and concrete parts that collectively describe a service endpoint.

ABSTRACT DESCRIPTION

An abstract description establishes the interface characteristics of the Web service without any reference to the technology used to host or enable a Web service to transmit messages.

By separating this information, the **integrity of the service description** can be preserved regardless of what changes might occur to the underlying technology platform.

portType, operation, and message

The parent **portType** section of an abstract description provides a **high-level view of the service interface**
Web services rely exclusively on messaging-based communication, parameters are represented as messages. Therefore, an operation consists of a set of input and output messages .

The term "portType" is being renamed to "interface" in version 2.0 of the WSDL specification.

CONCRETE DESCRIPTION

- For a Web service to be able to execute any of its logic, it needs for its abstract interface definition to be connected to some real, implemented technology.
- Because **the execution of service application logic always involves communication, the abstract Web service interface needs to be connected to a physical transport protocol.**

binding, port, and service

- A WSDL description's binding describes the requirements for a service to establish physical connections or for connections to be established with the service.
- In other words, a binding represents one possible transport technology the service can use to communicate.
- SOAP is the most common form of binding, but others also are supported. A binding can apply to an entire interface or just a specific operation.

The term "port" is being renamed "endpoint" in version 2.0 of the WSDL specification.

METADATA AND SERVICE CONTRACTS

We have up to three separate documents that each describe an aspect of a service:
- WSDL definition
- XSD schema
- Policy

❑ Each of these three service description documents can be classified as service metadata , as each provides information about the service.

❑ **service contract** can refer to additional documents or agreements not expressed by service descriptions. For example, a Service Level Agreement (SLA) agreed upon by the respective owners of a service provider and its requestor can be considered part of an overall service contract (Figure 5.17).

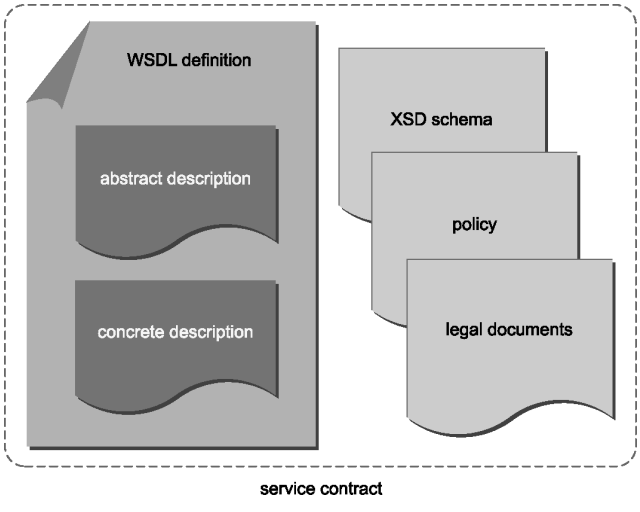


Figure 5.17: A service contract comprised of a collection of service descriptions and possibly additional documents.

SEMANTIC DESCRIPTIONS

***Examples of** service semantics include:*

- how a service behaves under certain conditions
- how a service will respond to a specific condition
- what specific tasks the service is most suited for

❑ Semantic information is usually of greater importance when dealing with external service providers, where your knowledge of another party's service is limited to the information the service owner decides to publish.

SERVICE DESCRIPTION ADVERTISEMENT AND DISCOVERY

As the amount of services increases within and outside of organizations, mechanisms for advertising and discovering service descriptions may become necessary.
For example, central directories and registries become an option to keep track of the many service descriptions that become available. These repositories allow humans (and even service requestors) to:

- locate the latest versions of known service descriptions
- discover new Web services that meet certain criteria
- When the initial set of Web services standards emerged, this eventuality was taken into account.

This is why **UDDI** formed part of the first generation of Web services standards. Though not yet commonly implemented, UDDI provides us with a registry model worth describing.

PRIVATE AND PUBLIC REGISTRIES

UDDI specifies a relatively accepted standard for structuring registries that keep track of **service descriptions** (Figure 5.18).

These registries can be searched manually and accessed programmatically via a standardized API.

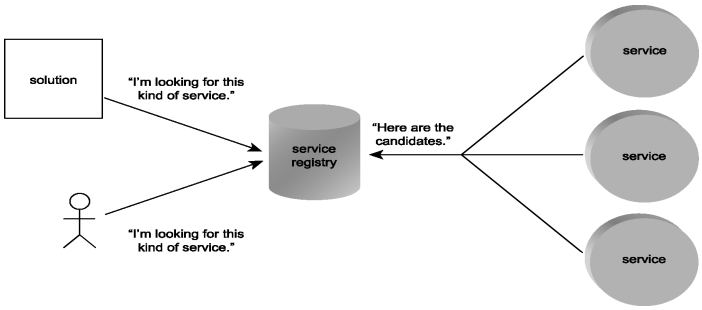


Figure 5.18: Service description locations centralized in a registry.

Public registries accept registrations from any organizations, regardless of whether they have Web services to offer. Once signed up, organizations acting as service provider entities can register their services.

Private registries can be implemented within organization boundaries to provide a central repository for descriptions of all services the organization develops, leases, or purchases.

Business entities and business services

Each public registry consists of a business entity containing basic profile information about the organization (or service provider entity).

Models are core components of UDDI.
tModels represent unique concepts or constructs and are used to describe compliance with a specification, a concept, a category or identifier system, or a shared design

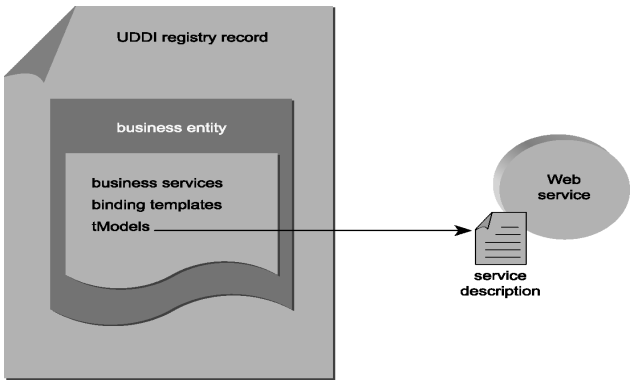


Figure 5.19: The basic structure of a UDDI business entity record.

Case Study

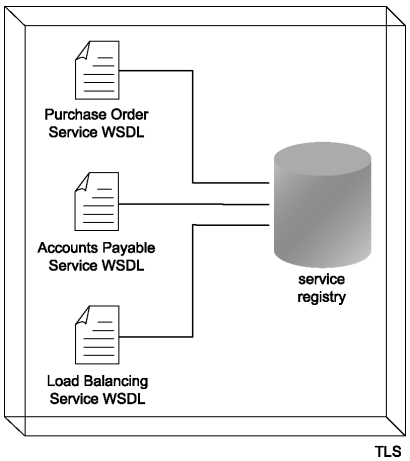


Figure 5.20: The TLS service registry containing pointers to current TLS WSDL definitions.

MESSAGING (WITH SOAP)

- Because all communication between services is message-based, the messaging framework chosen must be standardized so that all services, regardless of origin, use the same format and transport protocol.

Structure of message :>Envelope, header, and body

Every SOAP message is packaged into a container known as an envelope . The envelope is responsible for housing all parts of the message (Figure 5.21).

- Each message can contain a **header** , an area dedicated to hosting meta information.
- In most service-oriented solutions, this header section is a vital part of the overall architecture, and though optional, it is rarely omitted.
- The actual message contents are hosted by the **message body** , which typically consists of XML formatted data.

- A primary characteristic of the SOAP communications framework used by SOAs is an emphasis on creating messages that are as intelligence-heavy and self-sufficient as possible.

- Message independence is implemented through the use of **header blocks** , packets of supplementary meta information stored in the envelope's header area.

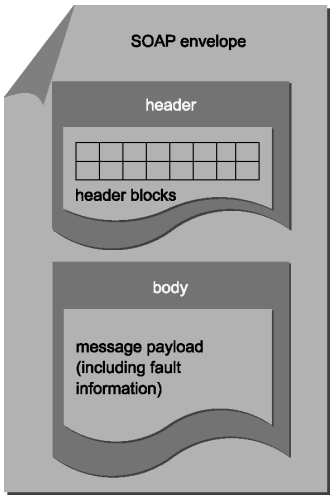


Figure 5.21: The basic structure of a SOAP message

HEADER BLOCKS

Examples of the types of features a message can be outfitted with using header blocks include:

- processing instructions that may be executed by service intermediaries or the ultimate receiver
- routing or workflow information associated with the message
- security measures implemented in the message
- reliability rules related to the delivery of the message
- context and transaction management information
- correlation information (typically an identifier used to associate a request message with a response message)

.

Case Study

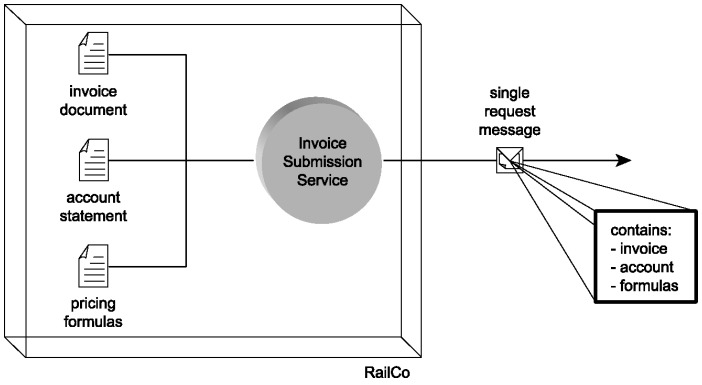


Figure 5.22: The RailCo Invoice Submission Service packaging the contents of three documents into one SOAP message.

NODES

- Although Web services exist as self-contained units of processing logic, they are reliant upon a physical communications infrastructure to process and manage the exchange of SOAP messages.

- Every major platform has its own implementation of a SOAP communications server, and as a result each vendor has labeled its own variation of this piece of software differently.

- In abstract, the programs that services use to transmit and receive SOAP messages are referred to as SOAP nodes (Figure 5.23).

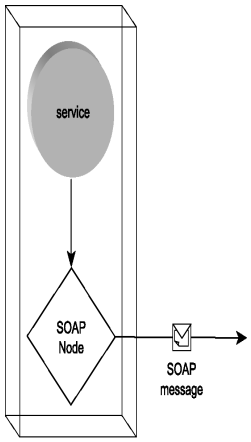


Figure 5.23: A SOAP node transmitting a SOAP message received by the service logic.

NODE TYPES

1. SOAP sender: a SOAP node that transmits a message
2. SOAP receiver: a SOAP node that receives a message
3. SOAP intermediary: a SOAP node that receives and transmits a message, and optionally processes the message prior to transmission
4. Initial SOAP sender: the first SOAP node to transmit a message
5. Ultimate SOAP receiver: the last SOAP node to receive a message

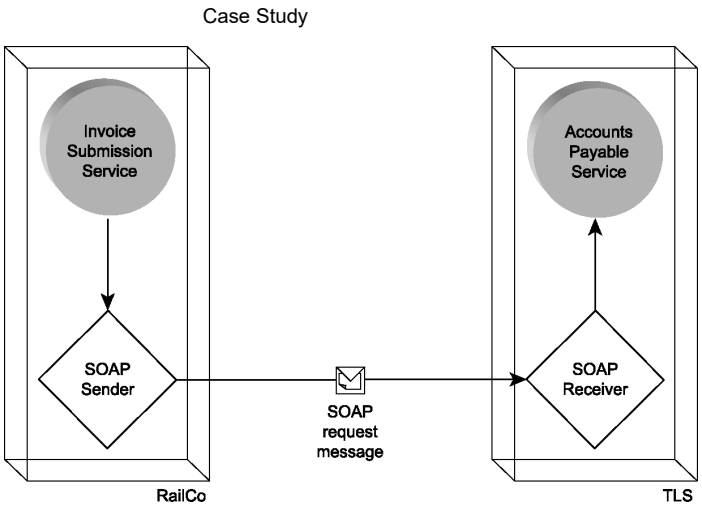


Figure 5.24: The positioning of SOAP nodes within a message transmission between RailCo and TLS.

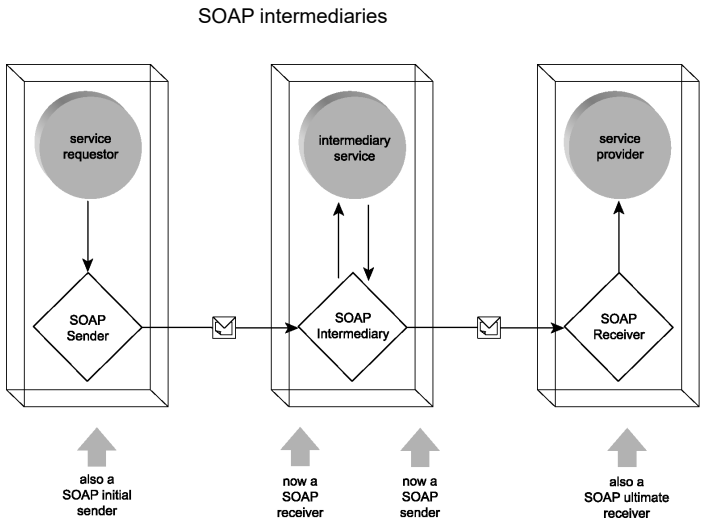


Figure 5.25: Different Types of SOAP nodes involved with processing a message.

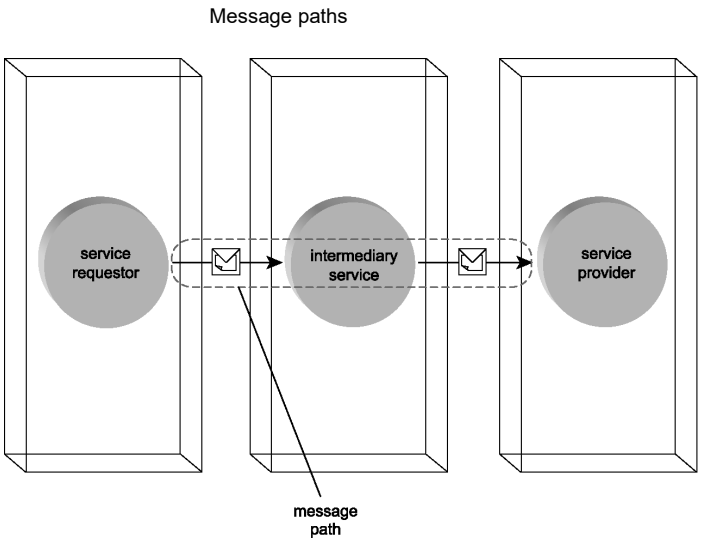


Figure 5.26: A message path consisting of three Web services.

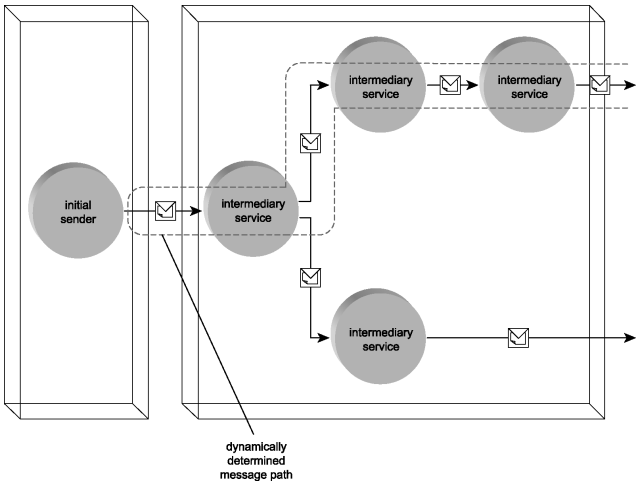


Figure 5.27: A message path determined at runtime.

Chapter 6

WEB SERVICES AND CONTEMPORARY SOA (PART I: ACTIVITY MANAGEMENT AND COMPOSITION)

This chapter explains key concepts that define the messaging-based services activity model and also discusses the role and context of concepts derived from WS-* specifications.

Message Exchange Patterns (MEP)

Basic:

- Every task automated by a Web service can differ in both the nature of the application logic being executed and the role played by the service in the overall execution of the business task.
- Regardless of how complex a task is, almost all require the transmission of multiple messages.
- The **challenge lies** in coordinating these messages in a particular sequence so that the individual actions performed by the message are executed properly and in alignment with the overall business task

Message exchange patterns (MEPs) represent a set of templates that provide a group of already mapped out sequences for the exchange of messages.

The most common example is a **request and response pattern**. Here the MEP states that upon successful delivery of a message from one service to another, the receiving service responds with a message back to the initial requestor

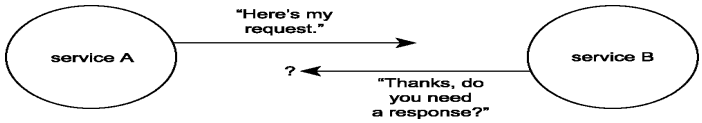


Figure 6.2: Not all message exchanges require both requests and responses.

An MEP is like a type of conversation.
It's not a long conversation; it actually only covers one exchange between two parties.

Primitive MPEs

Request-response

This is the most popular MEP in use among distributed application environments and the one pattern that defines synchronous communication (although this pattern also can be applied asynchronously).

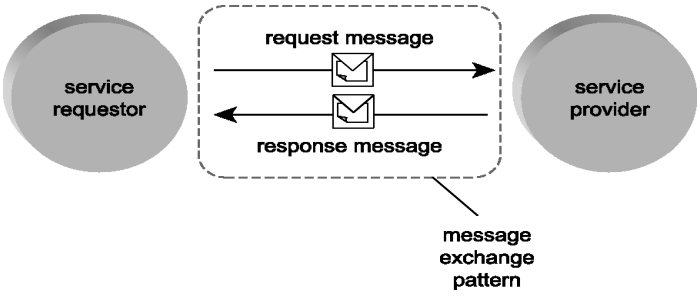


Figure 6.3: The request-response MEP.

Case Study – Request-Response Message

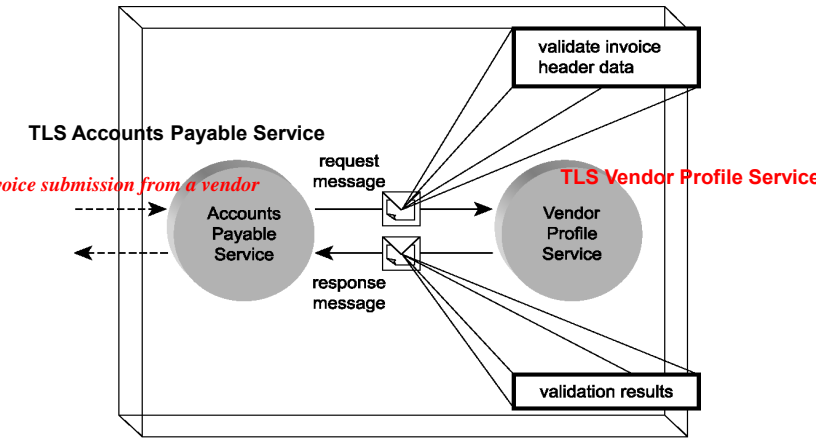


Figure 6.4: A sample request-response exchange between the TLS Accounts Payable and Vendor Profile Services.

Fire and Forget MEP

- This simple **asynchronous pattern** is based on the unidirectional transmission of messages from a source to one or more destinations
- The fundamental characteristic of this pattern is that a **response to a transmitted message is not expected.**

A number of variations of the fire-and-forget MEP exist, including:

- The single-destination pattern, where a source sends a message to one destination only.
- The multi-cast pattern, where a source sends messages to a predefined set of destinations.
- The broadcast pattern, which is similar to the multi-cast pattern, except that the message is sent out to a broader range of recipient destinations.

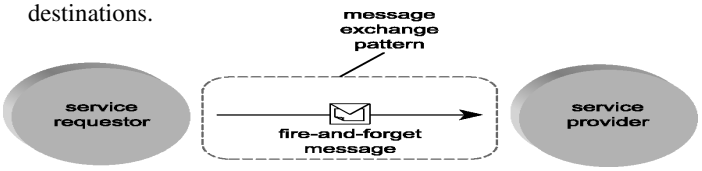


Figure 6.5: The fire-and-forget MEP.

Case Study

- The TLS Accounts Payable Service contains a rule that when an invoice header **fails validation**, an **e-mail notification** is generated.
- To execute this step, the Accounts Payable Service sends a message to the Notification Service.
- This utility service records the message details in a notification **log database**.

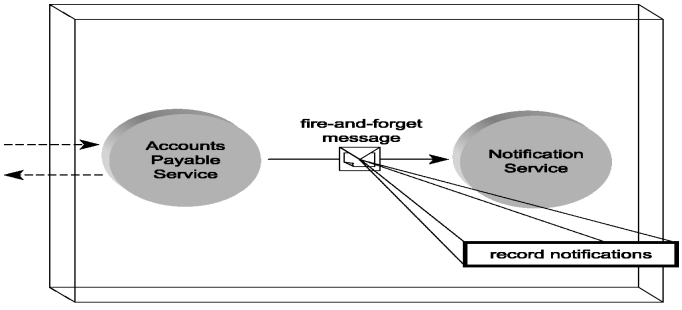


Figure 6.6: The TLS Accounts Payable Service sending off a one-way notification message.

Complex MEPs

A classic example is the publish-and-subscribe model.

Step 1.	The subscriber sends a message to notify the publisher that it wants to receive messages on a particular topic. Publish-and-subscribe MEP implemented.
Step 2.	Upon the availability of the requested information, the publisher broadcasts messages on the particular topic to all of that topic's subscribers. fire-and-forget patterns.

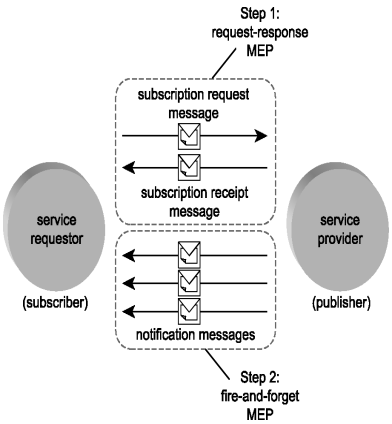


Figure 6.7: The publish-and-subscribe messaging model is a composite of two primitive MEPs.

MEPS AND SOAP

- On its own, the SOAP standard provides a messaging framework designed to support single-direction message transfer.
- The extensible nature of SOAP allows countless messaging characteristics and behaviors to be implemented via SOAP header block.
- The SOAP language also provides an optional parameter that can be set to identify the MEP associated with a message

MEPS AND WSDL

- Operation defined within **service description** are comprised, in part, of message definition.
- The exchange of these message constitute the execution of a task represented by an operation.
- MEPs plays an important role in WSDL service descriptions as they can coordinate the input and output messages associated with an operation

The association of MEPs to WSDL operations thereby embeds expected conversational behavior into the interface definition.

Patterns are applied to service operations from the perspective of a **service provider or endpoint**. ***In WSDL 1.1 terms***, they are represented as follows :

1. ***Request-response operation*:** Upon receiving a message, the service must respond with a standard message or a fault message.

2. ***Solicit-response operation***: Upon submitting a message to a service requestor, the service expects a standard response message or a fault message.
3. ***One-way operation***: The service expects a single message and is not obligated to respond.
4. ***Notification operation***: The service sends a message and expects no response.

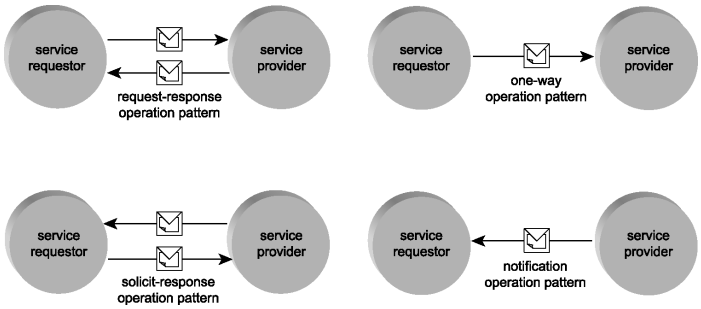


Figure 6.9: The four basic patterns supported by WSDL 1.1.

Specifically, release ***2.0 of the WSDL*** specification extends MEP support to eight patterns (and also changes the terminology) as follows.

- ***The in-out pattern*** , comparable to the **request-response** MEP (and equivalent to the WSDL 1.1 request-response operation).

- ***The out-in pattern*** , which is the reverse of the previous pattern where the service provider initiates the exchange by transmitting the request. (Equivalent to the WSDL 1.1 solicit-response operation.)

- ***The in-only pattern*** , which essentially supports the standard fire-and-forget MEP. (Equivalent to the WSDL 1.1 one-way operation.)

- ***The out-only pattern*** , which is the reverse of the in-only pattern. It is used primarily in support of event notification. (Equivalent to the WSDL 1.1 notification operation.)

- ***The robust in-only pattern*** , a variation of the in-only pattern that provides the option of launching a fault response message as a result of a transmission or processing error.

- ***The robust out-only pattern*** , which, like the out-only pattern, has an outbound message initiating the transmission. The difference here is that a fault message can be issued in response to the receipt of this message.
- ***The in-optional-out pattern*** , which is similar to the in-out pattern with one exception.
  - This variation introduces a rule stating that the delivery of a response message is optional and should therefore not be expected by the service requestor that originated the communication. This pattern also supports the generation of a **fault message**.
- **The out-optional-in pattern** is the reverse of the in-optional-out pattern, where the incoming message is optional. Fault message generation is again supported.

MEPS AND SOA

MEPs & SOA: MEPs are highly generic and abstract in nature. They are fundamental and essential part of any web service based environment, including SOA.

COMMUNICATION PATTERN IN REST

In REST web services its individual operation are accessed through uniform resource identifier(URI).

REST uses HTTP protocol for transmission.

These are the method used by REST to support transmission.
1. GET
2. POST

❑ GET is used to provide parameter to URI and response is returned.
❑ POST is used to modify some value.

CONTINUED...

❑ REST is lightweight process as it does not require the exchange of heavy SOAP messages.

❑ SOAP message are written in XML which is verbose in nature. So it creases the Network traffic and load on server.

❑ REST uses XML as well as JSON for representation of data. It is lightweight in Nature.

❑ As the REST is lightweight access of resources over the network is fast. So the response time is less than that of SOAP.

❑ Throughput is also high in case of REST as compared to SOAP.

Service activity

- The completion of business tasks is an obvious function of any automated solution.
- Tasks are comprised of processing logic that executes to fulfill a number of business requirements.
- In service-oriented solutions, each task can involve any number of services. The interaction of a group of services working together to complete a task can be referred to as a service activity

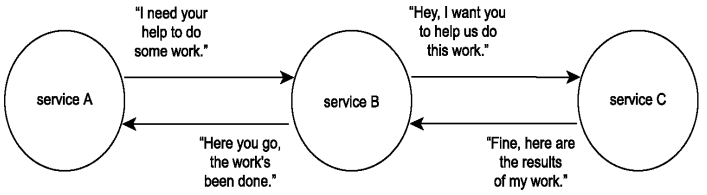


Figure 6.10: In an activity, multiple Web services collaborate to do a specific piece of work.

Primitive and Complex Service activities

The scope of an activity can drastically vary.
A simple or primitive activity is typified by synchronous communication and therefore often consists of two services exchanging information using a standard **request-response MEP**

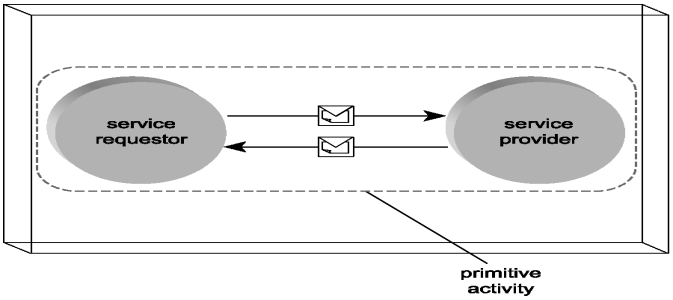


Figure 6.11: A primitive service activity consisting of a simple MEP.

Complex activities: Can involve many services (and MEPs) that collaborate to complete multiple processing steps over a long period of time .
These more elaborate types of activities are generally structured around extension-driven and composition-oriented concepts, such as choreography and orchestration.

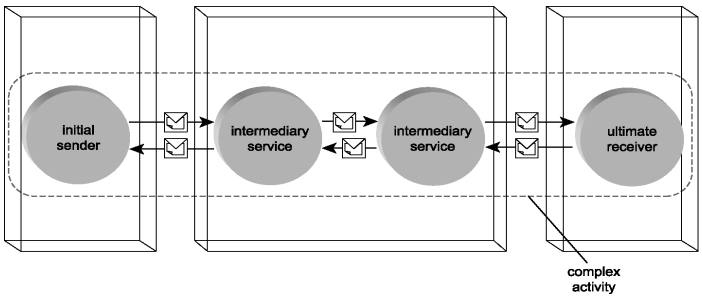


Figure 6.12: A complex activity involving four services.

Case Study

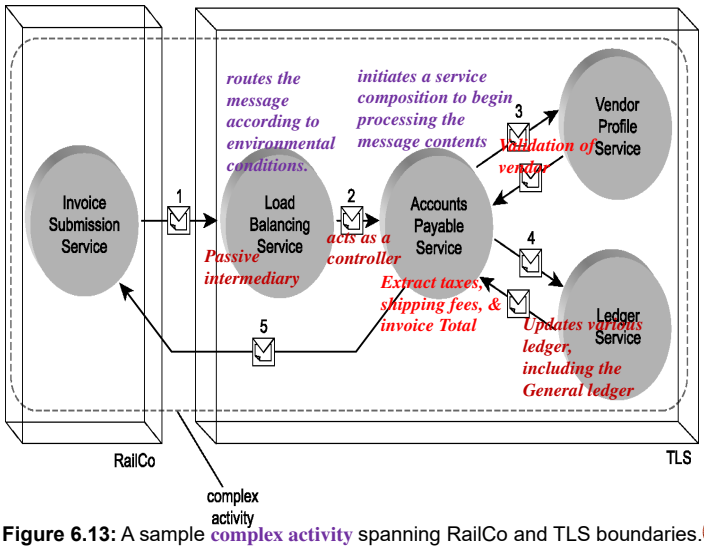


Figure 6.13: A sample **complex activity** spanning RailCo and TLS boundaries.

21

Coordination

A framework is required to provide a means for context information in complex activities to be managed, preserved and/or updated, and distributed to activity participants . Coordination establishes such a framework.

complexity of an activity can relate to a number of factors, including:

- the amount of services that participate in the activity
- the duration of the activity
- the frequency with which the nature of the activity changes
- whether or not multiple instances of the activity can concurrently exist

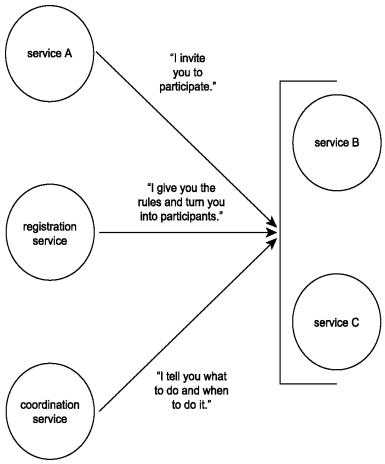


Figure 6.14: Coordination provides services that introduce **controlled structure into activities.**

22

Coordinator composition Services

The coordinator composition consists of the following services:

- Activation Service Responsible for the creation of a new context and for associating this context to a particular activity.
- Registration Service Allows participating services to use context information received from the activation service to register for a supported context protocol.
- Protocol-Specific Services These services represent the protocols supported by the coordinator's coordination type.
- Coordinator The controller service of this composition, also known as the coordination service .

23

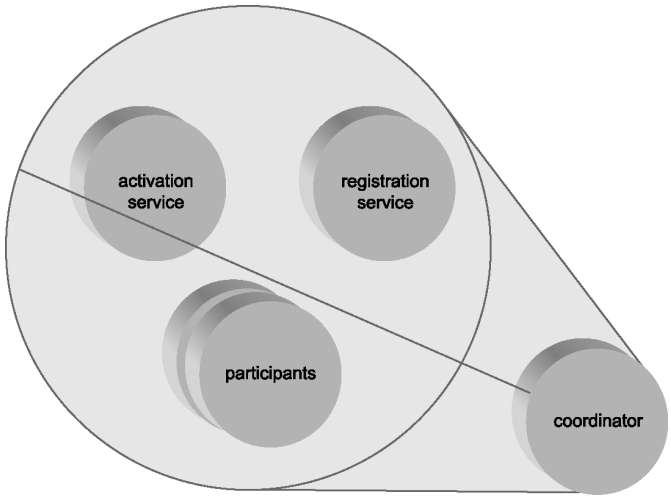


Figure 6.15: The coordinator service composition.

24

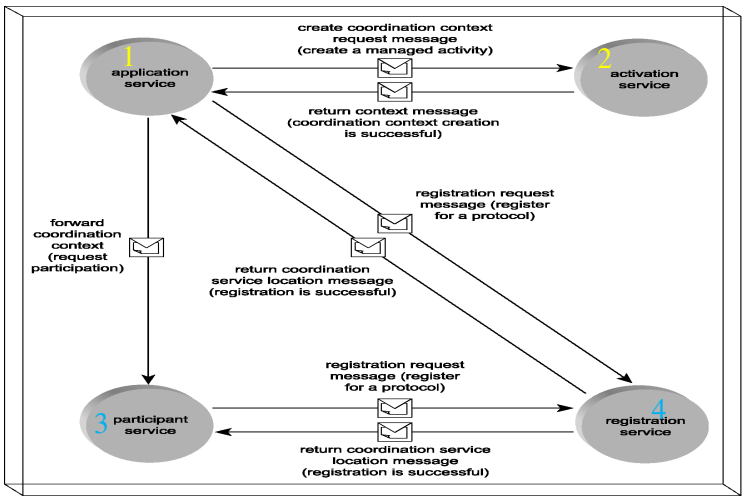


Figure 6.16: The WS-Coordination registration process.

25

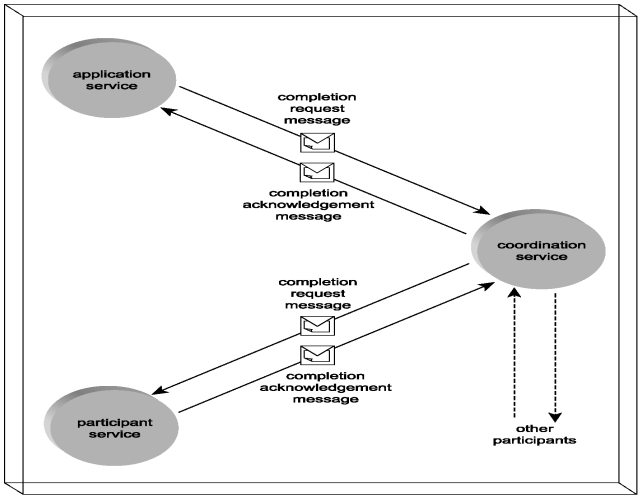


Figure 6.17: The WS-Coordination completion process.

26

Coordination and SOA

A coordinator based context management framework, as provided by WS-Coordination and its supporting types, introduces a layer of composition control to SOAs.

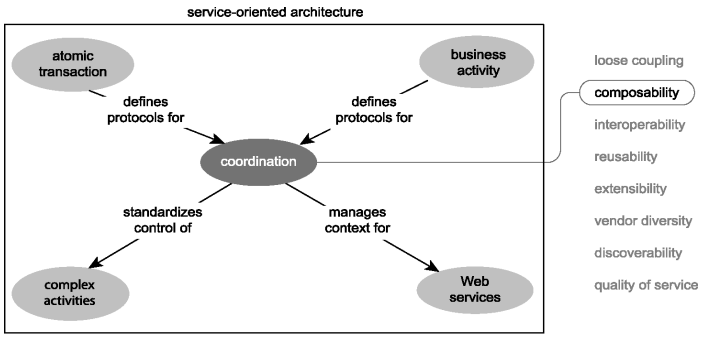


Figure 6.18: Coordination as it relates to other parts of SOA.

Case Study

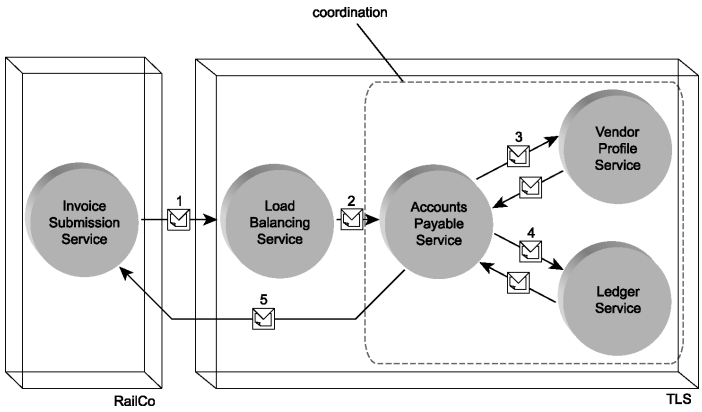


Figure 6.19: The TLS Accounts Payable, Vendor Profile, and Ledger Services being managed by a coordination.

Atomic Transaction

Atomic transactions implement the familiar commit and rollback features to enable cross-service transaction support

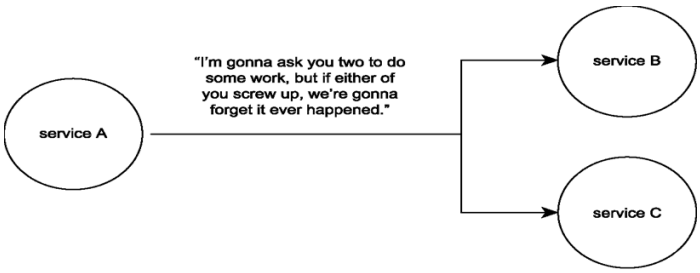


Figure 6.20: Atomic transactions apply an all-or-nothing requirement to work performed as part of an activity.

ACID transactions

The protocols provided by the WS-Atomic Transaction specification enable cross-service transaction functionality comparable to the ACID-compliant transaction features found in most distributed application platforms.

Atomic: Either all of the changes within the scope of the transaction succeed, or none of them succeed. This characteristic introduces the need for the rollback feature that is responsible for restoring any changes completed as part of a failed transaction to their original state.

Consistent :None of the data changes made as a result of the transaction can violate the validity of any associated data models. Any violations result in a rollback of the transaction.

Isolated: If multiple transactions occur concurrently, they may not interfere with each other. Each transaction must be guaranteed an isolated execution environment.

Durable: Upon the completion of a successful transaction, changes made as a result of the transaction can survive subsequent failures.

The atomic transaction Protocol

- A Completion Protocol: which is typically used to initiate the commit or abort states of the transaction.
- Durable 2PC protocol for which services representing permanent data repositories should register
- The Volatile 2PC protocol to be used by service managing non-persistent data.

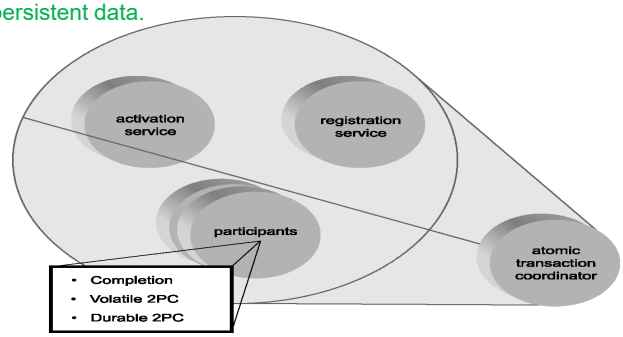


Figure 6.21: The atomic transaction coordinator service model.

Atomic Transaction Process for feedback

Phase1

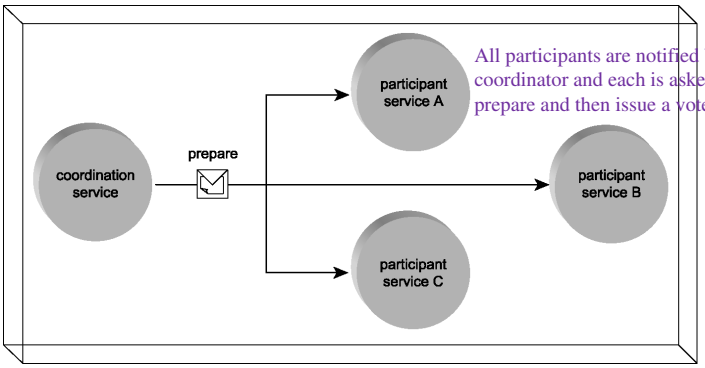


Figure 6.22: The coordinator requesting that transaction participants prepare to vote.

Each participant's vote consists of either a commit or abort request

Phase1

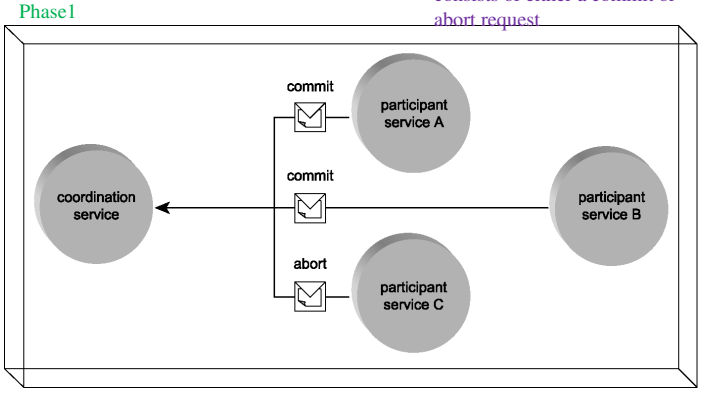


Figure 6.23: The transaction participants voting on the outcome of the atomic transaction.

After collecting the vote than start phase 2.

Phase2 Commit/Rollback

Commit: when all votes received and if all participant's voted to commit- Coordinator declare the transaction successful.

Rollback: If any one vote request an abort, or any one of participants fail to response

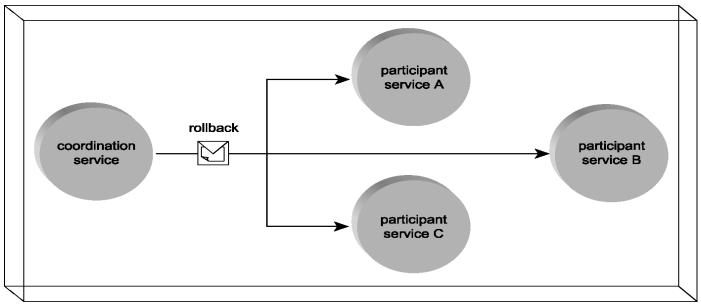


Figure 6.24: The coordinator aborting the transaction and notifying participants to rollback all changes.

Atomic transactions and SOA

How atomic transition support/promote these aspects of SOA

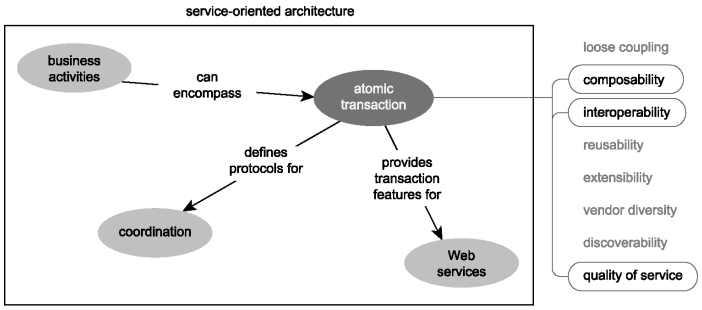


Figure 6.25: Atomic transaction relating to other parts of SOA.

Case Study

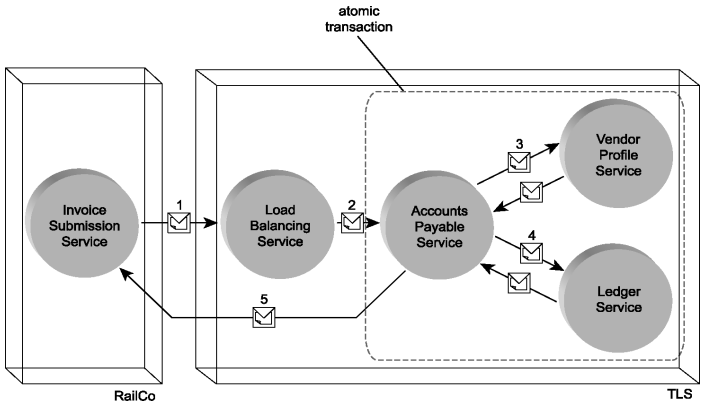


Figure 6.26: All changes made by the TLS Accounts Payable, Vendor Profile, and Ledger Services are under the control of an atomic transaction.

Business activities

Business activities govern long-running, complex service activities. Hours, days, or even weeks can pass before a business activity is able to complete.

What distinguishes a business activity from a regular complex activity is that its participants are required to follow specific rules defined by protocols. Business activities primarily ***differ*** from the also *protocol-based atomic* transactions in how they deal with exceptions and in the nature of the constraints introduced by the protocol rules.

For instance, **business activity** protocols do not offer **rollback capabilities.** Given the potential for business activities to be long-running, it would not be realistic to expect ***ACID-type*** transaction functionality. Instead, business activities provide an optional compensation process that, much like a "plan B," can be invoked when exception conditions are encountered

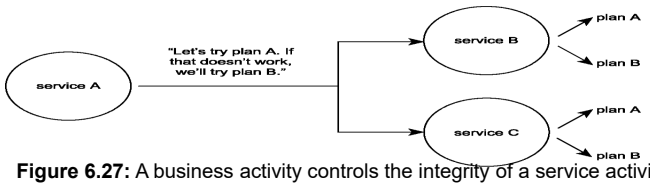


Figure 6.27: A business activity controls the integrity of a service activity by providing participants with a "plan B" (a compensation).

Business activity protocols

As with ***WS-Atomic Transaction,*** **WS-Business Activity** is a coordination type designed to leverage the WS-Coordination context management framework. It provides two very similar protocols, each of which dictates how a participant may behave within the overall business activity.

1. The ***BusinessAgreementWithParticipantCompletion*** protocol, which allows a participant to determine **when** it has completed its part in the business activity.

2. The ***BusinessAgreementWithCoordinatorCompletion*** protocol, which requires that a participant rely on the business activity coordinator to notify it that it has no further processing responsibilities.

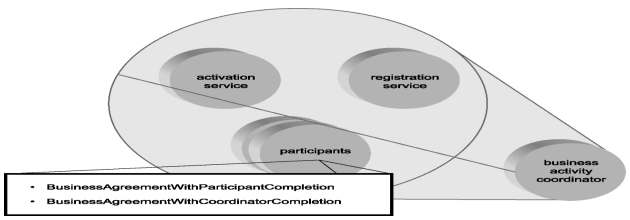


Figure 6.28: The business activity coordinator service model.

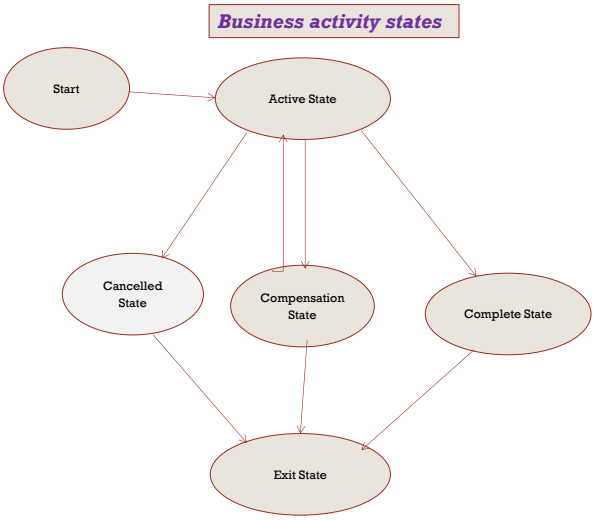


Figure 6.29: A business activity States.

Business activities and atomic transactions

1. It is important to note that the use of a business activity does not exclude the use of atomic transactions.
2. In fact, it is likely that a long-running business activity will encompass the execution of several atomic transactions during its lifetime

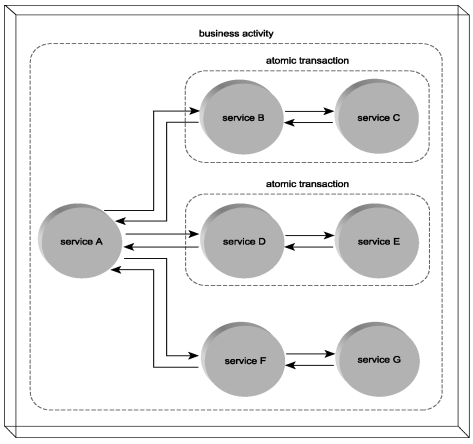


Figure 6.30: Two atomic transactions residing within the scope of a business activity.

Business activities and SOA

Business activities fully complement the composable nature of SOA by tracking and regulating complex activities while also allowing them to carry on for long periods of time.
The use of the compensation process, business activities increase SOA's quality of service by providing built-in fault handling logic

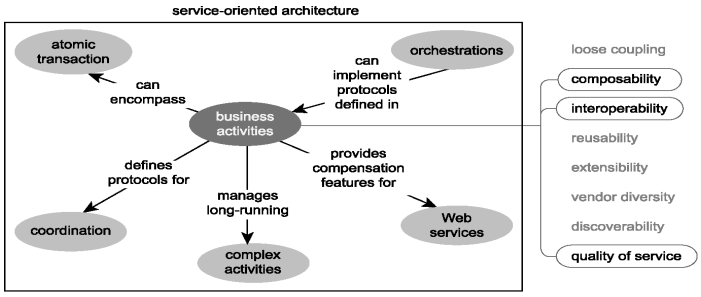


Figure 6.31: A business activity relating to other parts of SOA.

Case Study

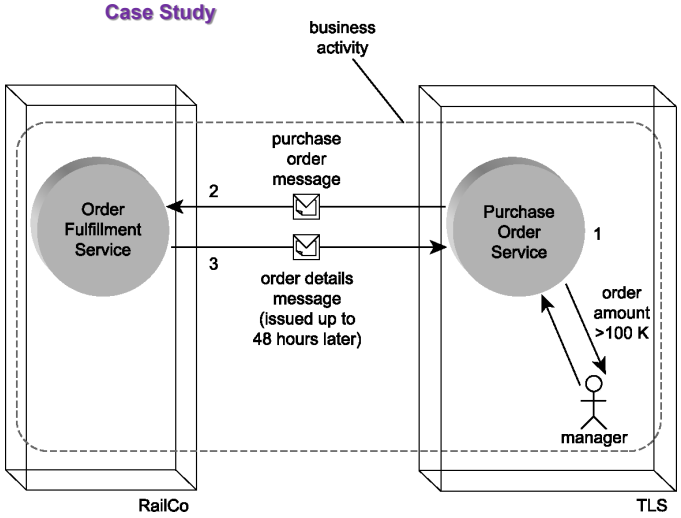


Figure 6.31: The TLS Purchase Order Submission Process wrapped in a long-running business activity and spanning two organizations (two participants).

Orchestration

- A common implementation of orchestration is the **hub-and-spoke** model that allows multiple external participants to interface with a central orchestration engine.
- **Orchestration** describes the automated arrangement, coordination, and management of complex computer systems, middleware, and services.
- With orchestration, different processes can be connected without having to redevelop the solutions that originally automated the processes individually. Orchestration bridges this gap by introducing new workflow logic.
- This is also treated as service
- A primary industry specification that standardizes orchestration is **WS-BPEL**

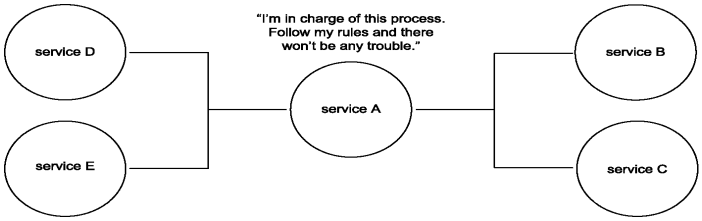


Figure 6.33: An orchestration controls almost every facet of a complex activity.

Business protocols and process definition

The workflow logic that comprises an orchestration can consist of numerous business rules, conditions, and events.

Collectively, these parts of an orchestration establish a business protocol that defines how participants can interoperate to achieve the completion of a business task.

The details of the workflow logic encapsulated and expressed by an orchestration are contained within a process definition .

Process services and partner services

Identified and described within a process definition are the allowable process participants. First, the process itself is represented as a service, resulting in a process service (which happens to be another one of our service models)

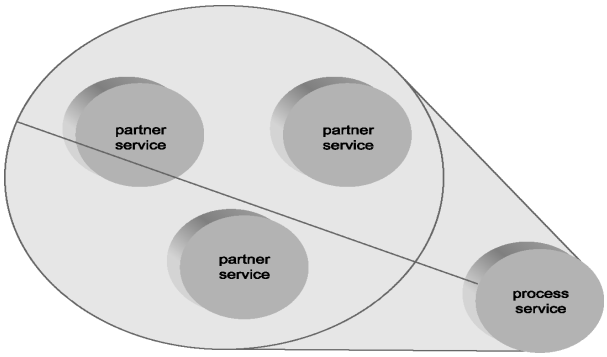


Figure 6.34: A process service coordinating and exposing functionality from three partner services.

Orchestration and SOA

Business process logic is at the root of automation solutions. Orchestration provides an automation model where process logic is centralized yet still extensible and composable. Through the use of orchestrations, service-oriented solution environments become inherently extensible and adaptive.

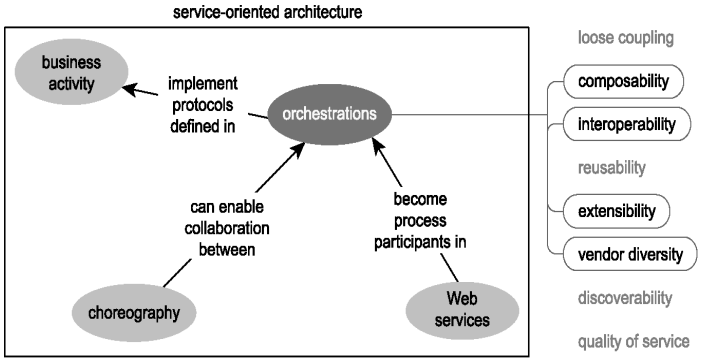


Figure 6.35: Orchestration relating to other parts of SOA.

These qualities lead to increased Organizational agility because:

- The workflow logic encapsulated by an orchestration can be modified or extended in a central location.
- Positioning an orchestration centrally can significantly ease the merging of business processes by abstracting the glue that ties the corresponding automation solutions together.
- By establishing potentially large-scale service-oriented integration architectures, orchestration, on a fundamental level, can support the evolution of a diversely federated enterprise.

Orchestration is a key ingredient to achieving a state of federation within an organization that contains various applications based on disparate computing platforms.
Advancements in middleware allow orchestration engines themselves to become fully integrated in service-oriented environments.

Case Study

The orchestration establishes comprehensive process logic that encompasses the business activity and extends it even further to govern additional interaction scenarios with multiple vendor services.

TLS employs a WS-BPEL orchestration

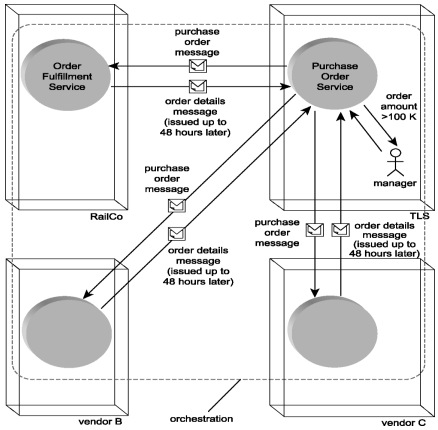


Figure 6.37: The extended TLS Purchase Order Submission Process managed by an orchestration and involving numerous potential partner organizations.

Choreography

A choreography is essentially a collaboration process designed to allow organizations to interact in an environment that is not owned by any one partner.

The Web Services Choreography Description Language (WS-CDL) is one of several specifications that attempts to organize information exchange between multiple organizations

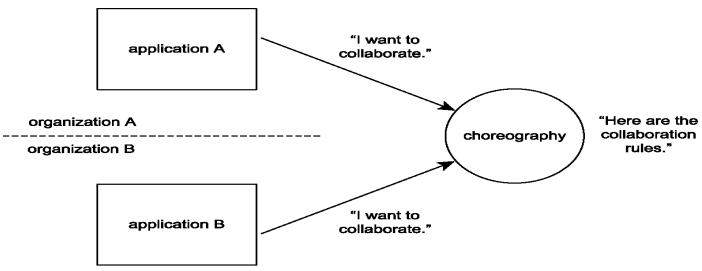


Figure 6.38: A choreography enables collaboration between its participants.

Collaboration

- An important characteristic of **choreographies** is that they are intended for public message exchanges. The goal is to establish a kind of organized collaboration between services representing different service entities, only no one entity (organization) necessarily controls the collaboration logic.
- **Choreographies** therefore provide the potential for establishing universal interoperability patterns for common inter-organization business tasks .

Reusability, composability, and modularity

- Each choreography can be designed in a reusable manner, allowing it to be applied to different business tasks comprised of the same fundamental actions.
- Further, using an import facility, a choreography can be assembled from independent modules .
- These modules can represent distinct sub-tasks and can be reused by numerous different parent choreographies

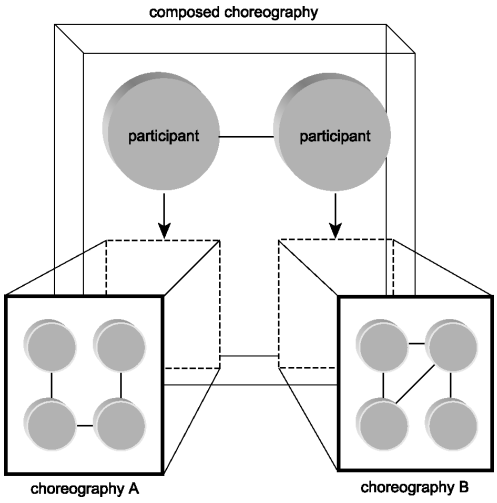


Figure 6.39: A choreography composed of two smaller choreographies.

Orchestrations and choreographies

An **orchestration** expresses organization-specific business workflow

A **choreography**, on the other hand, is not necessarily owned by a single entity. It acts as a community interchange pattern used for collaborative purposes by services from different provider entities

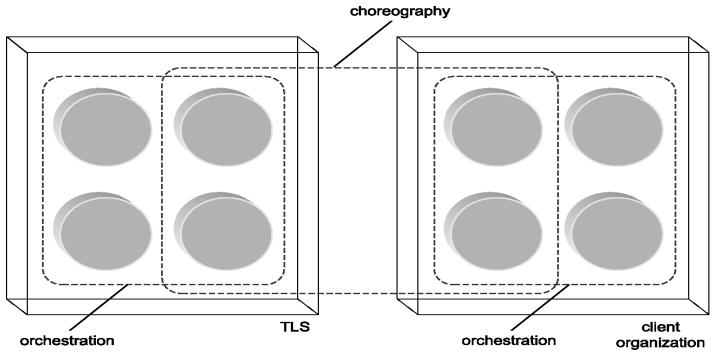


Figure 6.40: A choreography enabling collaboration between two different orchestrations.

Choreography and SOA

Choreography can assist in the realization of SOA across organization boundaries . While it natively supports composability, reusability, and extensibility, choreography also can increase organizational agility and discovery.

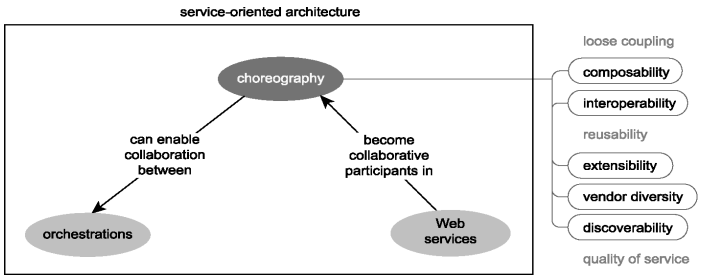
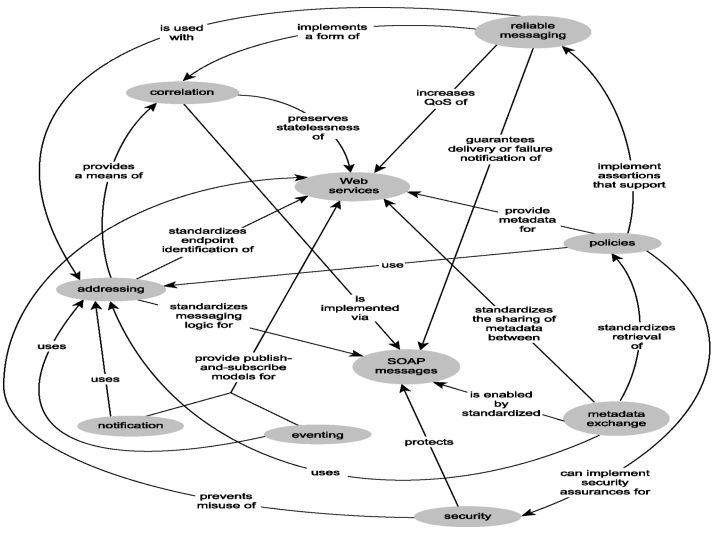


Figure 6.41: Choreography relating to other parts of SOA.

THANK YOU

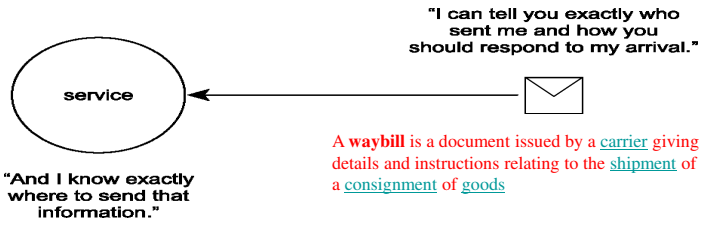
Chapter 7

Web Services and Contemporary SOA (Part II: Advanced Messaging, Metadata, and Security)



Addressing

- What addressing brings to SOAP messaging is much like what a waybill brings to the shipping process.
- Regardless of which ports, warehouses, or delivery stations a package passes through en route to its ultimate destination, with a waybill attached to it.
- Addressing keeps track of
  - where it's coming from
  - the address of where it's supposed to go
  - the specific person at the address who is supposed to receive it
  - where it should go if it can't be delivered as planned



A **waybill** is a document issued by a carrier giving details and instructions relating to the shipment of a consignment of goods

Figure 7.2: Addressing turns messages into autonomous units of communication.

Endpoint references

- An endpoint reference (EPR) is a combination of Web services (WS) elements that define the address for a resource in a Simple Object Access Protocol (SOAP) header.
- In this context, an endpoint is any user device connected to a network. Endpoints can include personal computers (PCs), personal digital assistants (PDAs) and specialized equipment such as inventory scanners and point-of-sale terminals.
- In other words, all that is required for a service requestor to contact a service provider is the provider's WSDL definition. This document, among other things, supplies the requestor with an address at which the provider can be contacted.
- The concept of addressing introduces the endpoint reference , an extension used primarily to provide identifiers that pinpoint a particular instance of a service (as well as supplementary service metadata).
- The endpoint reference is expected to be almost always dynamically generated and can contain a set of supplementary properties.

An endpoint reference consists of the following parts :

1. Address- The URL of the Web service.
2. Reference properties- A set of property values associated with the Web service instance
3. Reference parameters- A set of parameter values that can be used to further interact with a specific service instance.
4. Service port type and port type- Specific service interface information giving the recipient of the message the exact location of service description details required for a reply.
5. Policy- A WS-Policy compliant policy that provides rules and behavior information relevant to the current service interaction

7.1.1. Endpoint references

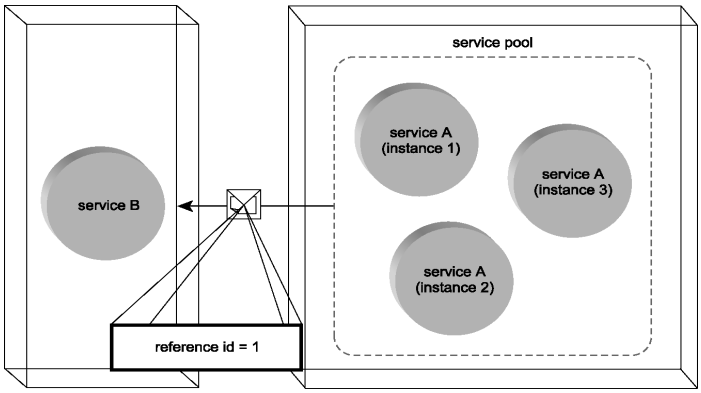


Figure 7.3: A SOAP message containing a reference to the instance of the service that sent it.

Message information headers

This collection of standardized headers is known as the message information (or MI) headers.
The MI headers provided by WS-Addressing include:

- Destination The address to which the message is being sent.
- Source endpoint An endpoint reference to the Web service that generated the message.
- Reply endpoint This important header allows a message to dictate to which address , its reply should be sent.
- Fault endpoint Further extending the messaging flexibility is this header, which gives a message the ability to set the address to which a fault notification should be sent.
- Message id A value that uniquely identifies the message or the retransmission of the message.
- Relationship Most commonly used in request-response scenarios, this header contains the message id of the related message to which a message is replying
- Action A URI value that indicates the message's overall purpose (the equivalent of the standard SOAP HTTP action value).

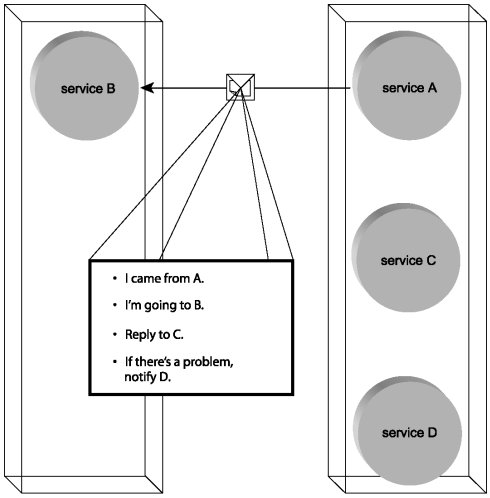


Figure 7.4: A SOAP message with message information headers specifying exactly how the recipient service should respond to its arrival.

7.1.3 Addressing and SOA

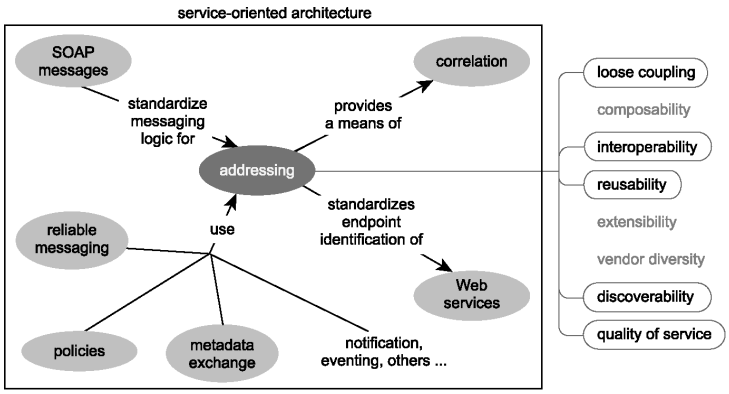


Figure 7.5: Addressing relating to other parts of SOA.

Case Study

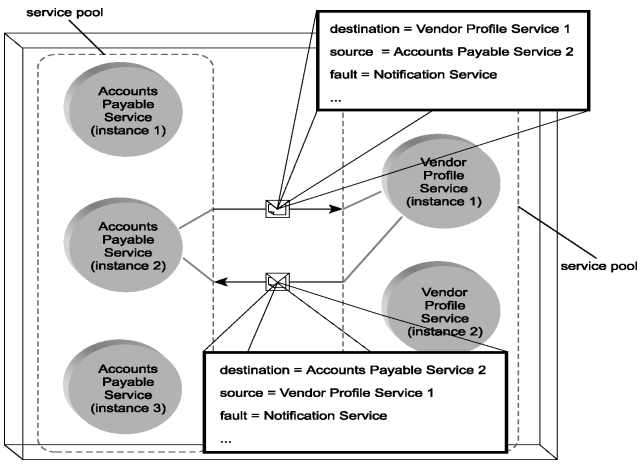


Figure 7.6: Separate service instances communicating using endpoint references and MI headers across two pools of Web services within TLS.

Reliable messaging

The benefits of a **loosely coupled** messaging framework come at the cost of a loss of control over the actual communications process. After a Web service transmits a message, it has no immediate way of knowing:

- whether the message successfully arrived at its intended destination
- whether the message failed to arrive and therefore requires a retransmission
- whether a series of messages arrived in the sequence they were intended to

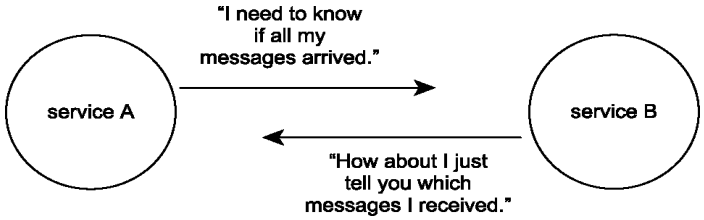


Figure 7.7: Reliable messaging provides a guaranteed notification of delivery success or failure.

RM Source, RM Destination, Application Source, and Application Destination

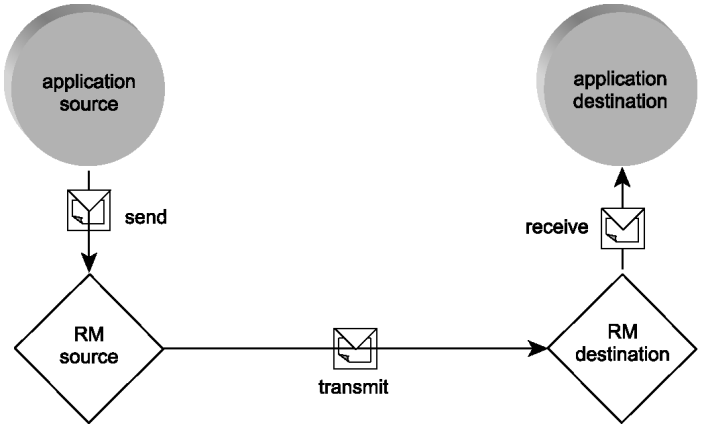


Figure 7.8: An application source, RM source, RM destination, and application destination.

Sequences

- A **sequence** establishes the order in which messages should be delivered.
- Each message that is part of a sequence is labeled with a message number that identifies the position of the message within the sequence.
- The final message in a sequence is further tagged with a last message identifier.

Acknowledgements

- A core part of the reliable messaging framework is a notification system used to communicate conditions from the RM destination to the RM source.
- Upon receipt of the message containing the last message identifier, the RM destination issues a sequence acknowledgement
- The acknowledgement message indicates to the RM source which messages were received.
- It is up to the RM source to determine if the messages received are equal to the original messages transmitted.
- The RM source may retransmit any of the missing messages, depending on the delivery assurance used

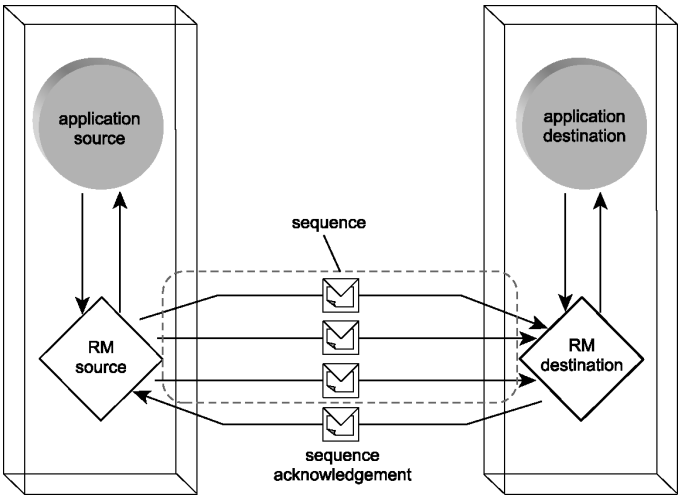


Figure 7.9: A sequence acknowledgement sent by the RM destination after the successful delivery of a sequence of messages.

An RM source does not need to wait until the RM destination receives the last message before receiving an acknowledgement. RM sources can request that additional acknowledgements be transmitted at any time by issuing request acknowledgements to RM destinations.

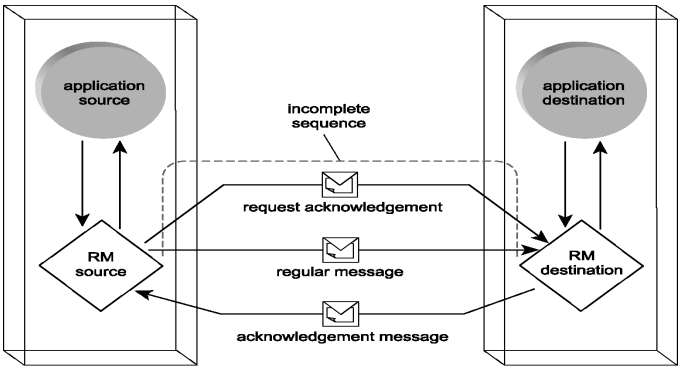


Figure 7.10: A request acknowledgement sent by the RM source to the RM destination, indicating that the RM source would like to receive an acknowledgement message before the sequence completes.

Additionally, RM destinations have the option of transmitting negative acknowledgements that immediately indicate to the RM source that a failure condition has occurred

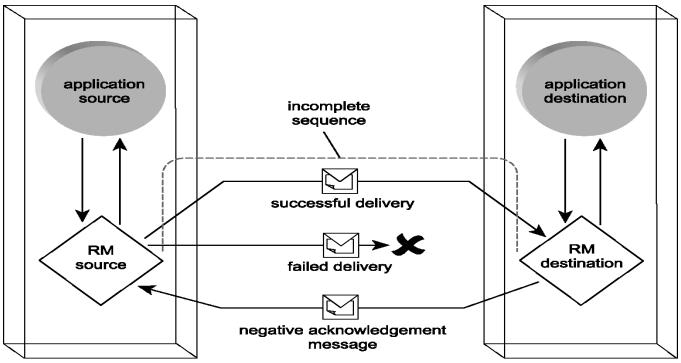


Figure 7.11: A negative acknowledgement sent by the RM destination to the RM source, indicating a failed delivery prior to the completion of the sequence.

Delivery assurances

- The nature of a sequence is determined by a set of reliability rules known as **delivery assurances**. Delivery assurances are predefined message delivery patterns that establish a set of reliability policies.
- The following delivery assurances are supported:
  - The **AtMostOnce** delivery assurance promises the delivery of one or zero messages. If more than one of the same message is delivered, an error condition occurs
  - The **AtLeastOnce** delivery assurance allows a message to be delivered once or several times. The delivery of zero messages creates an error condition.
  - The **ExactlyOnce** delivery assurance guarantees that a message only will be delivered once. An error is raised if zero or duplicate messages are delivered
  - The **In Order** delivery assurance is used to ensure that messages are delivered in a specific sequence . The delivery of messages out of sequence triggers an error. Note that this delivery assurance can be combined with any of the previously described assurances.

Reliable messaging and SOA

Reliable messaging brings to service-oriented solutions a tangible quality of service.
It introduces a flexible system that guarantees the delivery of message sequences supported by comprehensive fault reporting.

This elevates the robustness of SOAP messaging implementations and eliminates the reliability concerns most often associated with any messaging frameworks.

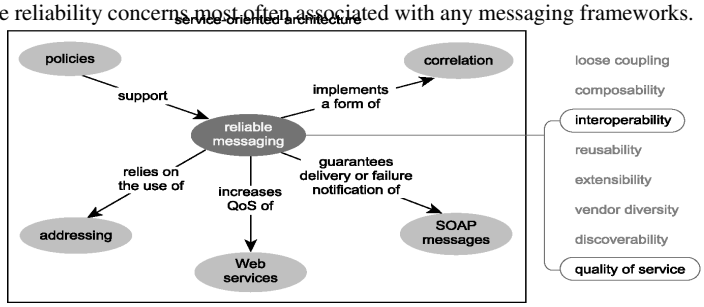


Figure 7.16: Reliable messaging relating to other parts of SOA.

Case Study

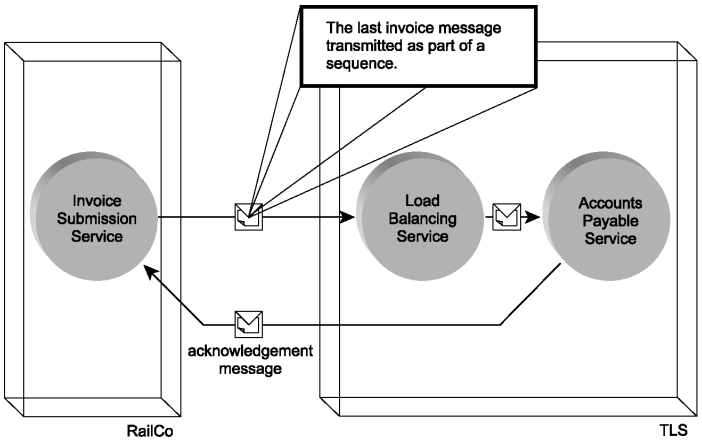


Figure 7.17: After transmitting a series of invoice messages, the last message within the sequence triggers the issuance of an **acknowledgement message** by TLS.

Correlation

- A service-oriented communications framework is inherently loosely coupled , there is **no intrinsic** mechanism for associating messages exchanged under a common context or as part of a common activity.
- Even the execution of a simple **request-response message** exchange pattern provides no built-in means of **automatically associating the response message** with the original request.
- **Correlation addresses** this issue by requiring that related messages contain **some common** value that services can identify to establish their relationship with each other or with the overall task they are participating in.
- The specifications that realize this simple concept provide different manners of implementation.

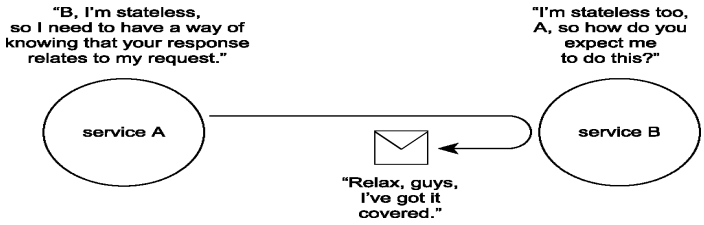


Figure 7.18: Correlation places the association of message exchanges into the hands of the message itself.

Correlation in orchestration

- **WS-BPEL orchestrations** need to concern themselves with the correlation of messages between **process and partner services.**
- This involves the added complexity of representing specific process instances within the correlation data.
- Further complicating this scenario is the fact that a single message may participate in multiple contexts, each identified by a separate correlation value.
- To facilitate these requirements, the WS-BPEL specification defines specific syntax that allows for the creation of extensible correlation sets .
- These message properties can be dynamically added, deleted, and altered to reflect a wide variety of message exchange scenarios and environments.

Policies

- The use of policies allows a service to express various characteristics and preferences and keeps it from having to implement and enforce **rules and constraints** in a custom manner.
- It adds an important layer of abstraction that allows service properties to be independently managed.

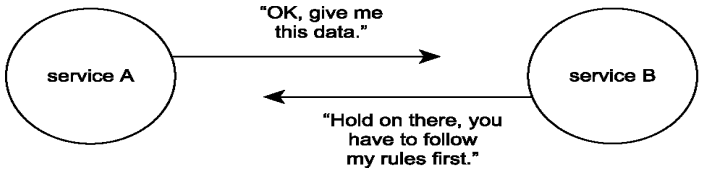


Figure 7.19: Policies can express a variety of service properties, including rules.

The WS-Policy framework

The WS-Policy framework establishes extensions that govern the assembly and structure of policy description documents, as well as the association of policies to Web resources. This framework is comprised of the following three specifications:

- **WS-Policy**
- **WS-Policy Attachments**
- **WS-Policy Assertions**

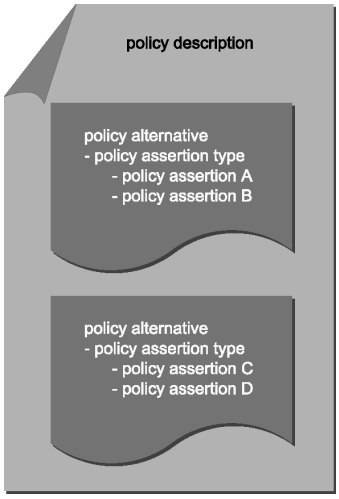


Figure 7.20: The basic structure of a policy description.

- Policies can be **programmatically accessed** to provide service requestors with an understanding of the requirements and restrictions of service providers at runtime.
- Alternatively, policies can be studied by humans at design time to develop service requestors designed to interact with specific service providers.

Policy assertions and policy alternatives

- The service properties expressed by a policy description are represented individually by policy assertions . A policy description therefore is comprised of one or more policy assertions.
- Examples of policy assertions include service characteristics, preferences, capabilities, requirements, and rules. Each assertion can be marked as optional or required.
- Policy assertions can be grouped into policy alternatives . Each policy alternative represents one acceptable (or allowable) combination of policy assertions.
- This gives a service provider the ability to offer service requestors a choice of policies.

Policy assertion types and policy vocabularies

- Policy assertions can be further categorized through policy assertion types.
- Policy assertion types associate policy assertions with specific **XSD schemas**.
- In the same manner as XML vocabularies are defined in XSD schemas, policy vocabularies simply represent the collection of policy types within a given policy.
- Similarly, a policy alternative vocabulary refers to the policy types contained within a specific policy alternative.

Policies in orchestration and choreography

- Policies can be applied to just about any subjects that are part of orchestrations or choreographies.
- For example, a policy can establish various requirements for **orchestration partner services and choreography** participants to interact.

Policies and SOA

- **If an SOA is a city, then policies are certainly the laws, regulations, and guidelines that exist to maintain order among inhabitants.**
- Policies are a necessary requirement to building enterprise-level service-oriented environments, as they provide a means of communicating constraints, rules, and guidelines for just about any facet of service interaction.
- As a result, they improve the overall quality of the loosely coupled arrangement services are required to maintain

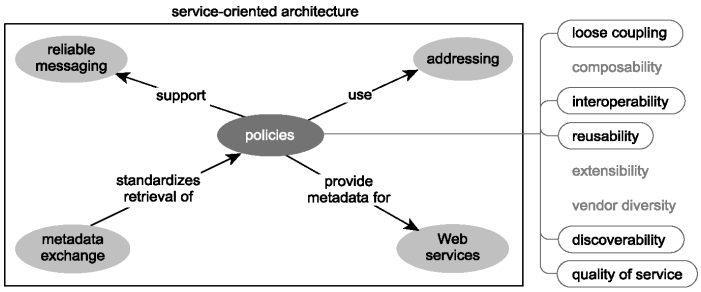


Figure 7.21: Policies relating to other parts of SOA.

Metadata Exchange

- Web services use **metadata** to describe what other **endpoints** need to know to **interact** with them.
- Specifically, WS-Policy describes the **capabilities, requirements**, and general **characteristics** of Web services;

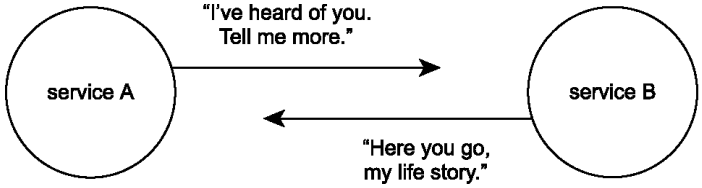


Figure 7.22: Metadata exchanges let service requestors ask what they want to know about service providers.

The WS-Metadata Exchange specification

- This specification essentially allows for a service requestor to issue a standardized request message that asks for some or all of the meta information relating to a specific endpoint address.
- In other words, if a service requestor knows where a service provider is located, it can use metadata exchange to request all service description documents that comprise the service contract by sending a metadata exchange request to the service provider.
- Originally the WS-Metadata Exchange specification specified the following three types of request messages:
  - Get WSDL
  - Get Schema
  - Get Policy
  - Get Metadata

Get Metadata request and response messages

- A Service Requestor can use metadata exchange to programmatically request available metadata documents associated with a Web service.
- To do so, it must issue a **Get Metadata** request message. This kicks off a standardized request and response MEP resulting in the delivery of a Get Metadata response message.

Here's what happens for a metadata retrieval activity to successfully complete:

- A service requestor issues the **Get Metadata** request message. This message can request a specific type of service description document (WSDL, XSD schema, policy), or it can simply request that all available metadata be delivered.
- The Get Metadata request message is received at the endpoint to which it is delivered. The requested meta information is documented in a Get Metadata response message.
- The Get Metadata response message is delivered to the service requestor. **The contents of this message can consist of the actual metadata documents, address references to the documents, or a combination of both**.

```
(01) <s12:Envelope
(02)     xmlns:s12='http://www.w3.org/2003/05/soap-envelope'
(03)     xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'
(04)     xmlns:wsx='http://schemas.xmlsoap.org/ws/2004/09/mex' >
(05)   <s12:Header>
(06)     <wsa:Action>
(07)       http://schemas.xmlsoap.org/ws/2004/09/mex/GetMetadata/Request
(08)     </wsa:Action>
(09)     <wsa:MessageID>
(10)         uuid:73d7edfc-5c3c-49b9-ba46-2480caee43e9
(11)     </wsa:MessageID>
(12)     <wsa:ReplyTo>
(13)       <wsa:Address>http://client.example.com/MyEndpoint</wsa:Address>
(14)     </wsa:ReplyTo>
(15)     <wsa:To>http://server.example.org/YourEndpoint</wsa:To>
(16)     <ex:MyRefProp xmlns:ex='http://server.example.org/refs' >
(17)         78f2dc229597b529b81c4bef76453c96
(18)     </ex:MyRefProp>
(19)   </s12:Header>
(20)   <s12:Body>
(21)     <wsx:GetMetadata>
(22)       <wsx:Dialect>
(23)         http://schemas.xmlsoap.org/ws/2004/09/policy
(24)       </wsx:Dialect>
(25)     </wsx:GetMetadata>
(26)   </s12:Body>
(27) </s12:Envelope>
```

```
(01) <s12:Envelope
(02)     xmlns:s12='http://www.w3.org/2003/05/soap-envelope'
(03)     xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'
(04)     xmlns:wsp='http://schemas.xmlsoap.org/ws/2004/09/policy'
(05)     xmlns:wsx='http://schemas.xmlsoap.org/ws/2004/09/mex' >
(06)   <s12:Header>
(07)     <wsa:Action>
(08)       http://schemas.xmlsoap.org/ws/2004/09/mex/GetMetadata/Response
(09)     </wsa:Action>
(10)     <wsa:RelatesTo>
(11)         uuid:73d7edfc-5c3c-49b9-ba46-2480caee43e9
(12)     </wsa:RelatesTo>
(13)     <wsa:To>http://client.example.com/MyEndpoint</wsa:To>
(14)   </s12:Header>
(15)   <s12:Body>
(16)     <wsx:Metadata>
(17)       <wsx:MetadataSection
(18)         Dialect='http://schemas.xmlsoap.org/ws/2004/09/policy' >
(19)         <wsp:Policy
(20)           xmlns:wsse='http://schemas.xmlsoap.org/ws/2002/12/secext' >
(21)           <wsp:ExactlyOne>
(22)             <wsse:SecurityToken>
(23)               <wsse:TokenType>wsse:Kerberosv5TGT</wsse:TokenType>
(24)             </wsse:SecurityToken>
(25)             <wsse:SecurityToken>
(26)               <wsse:TokenType>wsse:X509v3</wsse:TokenType>
(27)             </wsse:SecurityToken>
(28)           </wsp:ExactlyOne>
(29)         </wsp:Policy>
(30)       </wsx:MetadataSection>
(31)     </wsx:Metadata>
(32)   </s12:Body>
(33) </s12:Envelope>
```

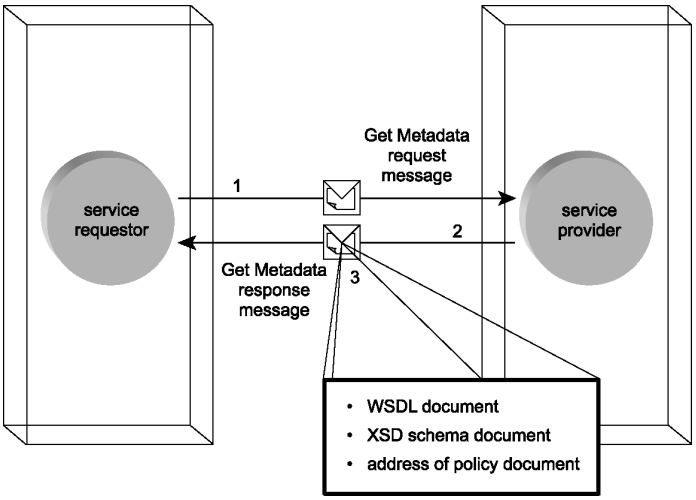


Figure 7.23: Contents of a sample Get Metadata response message.

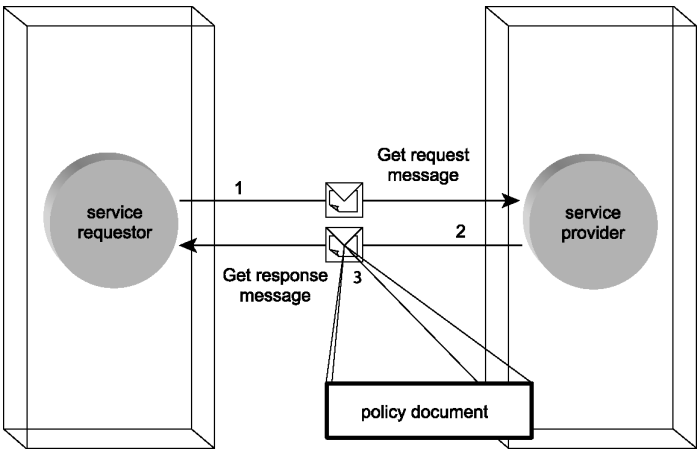


Figure 7.24: Contents of a sample Get response message.

Metadata exchange and SOA

- Its ability to automate the retrieval of meta information reinforces loose coupling between services, and increases the ability for service requestors to learn more about available service providers.
- By standardizing access to and retrieval of metadata, service requestors can programmatically query a multitude of candidate providers. Because enough service provider metadata can more easily be retrieved and evaluated, the overall discovery process is improved, and the likelihood for services to be reused is increased

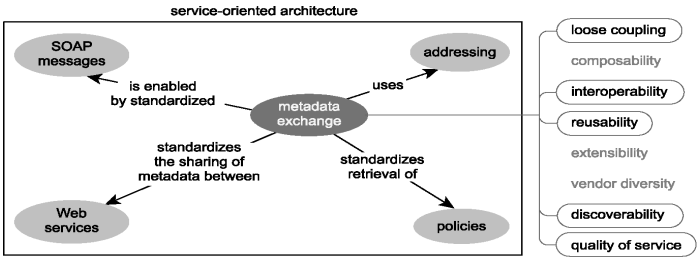


Figure 7.25: Metadata exchange relating to other parts of SOA.

Case Study

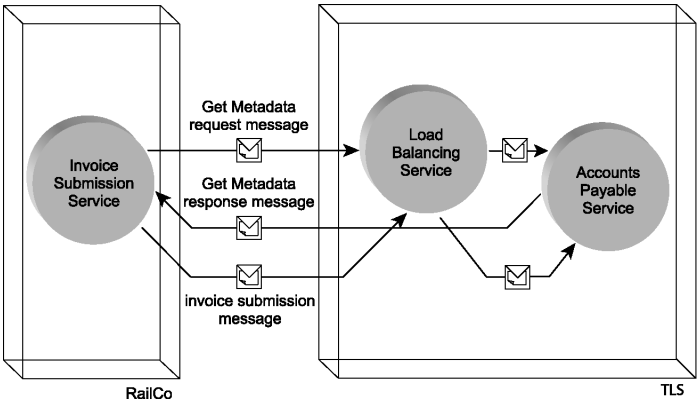


Figure 7.26: The revised RailCo Invoice Submission Process now includes a periodic metadata exchange with TLS.

Security

SOAP messaging communications framework, upon which contemporary SOA is built, emphasizes particular aspects of security that need to be accommodated by a security framework designed specifically for Web services.

Sidebar

A list of security specifications that may be used as part of SOA. For more information regarding these specifications, visit: www.specifications.ws .

- WS-Security
- WS-SecurityPolicy
- WS-Trust
- WS-SecureConversation
- WS-Federation
- Extensible Access Control Markup Language (XACML)
- Extensible Rights Markup Language (XrML)
- XML Key Management (XKMS)
- XML-Signature
- XML-Encryption
- Security Assertion Markup Language (SAML)
- .NET Passport
- Secure Sockets Layer (SSL)
- WS-I Basic Security Profile

Identification, authentication, and authorization

For a service requestor to access a secured service provider, it must first provide information that expresses its origin or owner.
This is referred to as making a claim .
Claims are represented by identification information stored in the SOAP header. WS-Security establishes a standardized header block that stores this information, at which point it is referred to as a token .

Identity

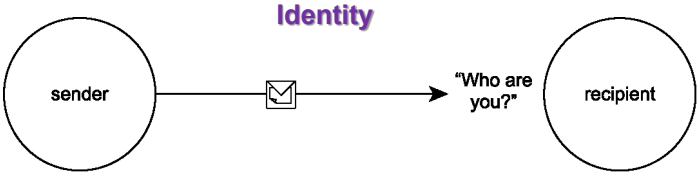


Figure 7.27: An identity is a claim made regarding the origin of a message.

Authentication

Authentication requires that a message being delivered to a recipient prove that the message is in fact from the sender that it claims to be.
In other words, the service must provide proof that its claimed identity is true.

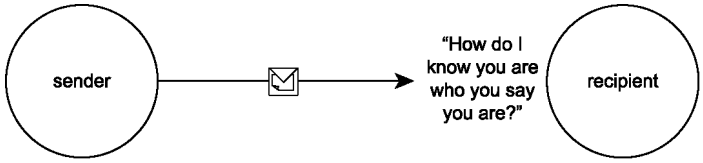


Figure 7.28: Authentication means proving an identity.

Authorization

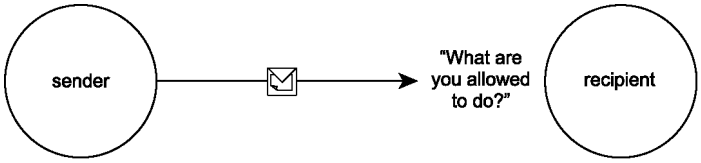


Figure 7.29: Authorization means determining to what extent authentication applies.

Single sign-on- a form of central security

- Because services are autonomous and independent from each other, a mechanism is required to persist the security context established after a requestor has been authenticated.
- Otherwise, the requestor would need to re-authenticate itself with every subsequent request.
- The concept of single **sign-on addresses this issue**. The use of a single sign-on technology allows a service requestor to be **authenticated once** and then have its security context information shared with other services that the requestor may then access without further authentication.

There are three primary extensions that support the implementation of the single sign-on concept:

- **SAML (Security Assertion Markup Language)**
- **.NET Passport**
- **XACML (XML Access Control Markup Language)**

- As an example of a single sign-on technology that supports centralized authentication and authorization, let's briefly discuss some fundamental concepts provided by **SAML**.
- SAML implements a **single sign-on system** in which the point of contact for a service requestor can also act as an **issuing authority** . This permits the underlying logic of that service not only to authenticate and authorize the service requestor, but also to assure the other services that the service requestor has attained this level of clearance.
- Other services that the **service requestor contacts**, therefore, do not need to perform authentication and authorization steps. Instead, upon receiving a request, they simply contact the issuing authority to ask for the authentication and authorization clearance it originally obtained.
- The issuing authority provides this information in the form of assertions that communicate the security details

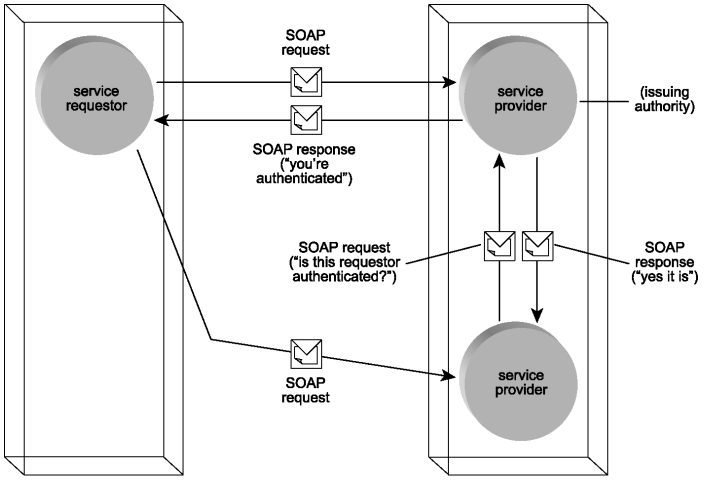


Figure 7.30: A basic message exchange demonstrating single sign-on (in this case, as implemented by SAML).

Confidentiality

Confidentiality is concerned with protecting the privacy of the message contents.

A message is considered to have remained confidential if no service or agent in its message path not authorized to do so viewed its contents.

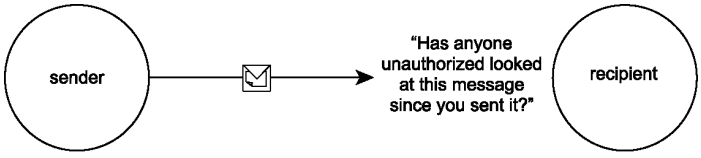


Figure 7.31: Confidentiality means that the privacy of the message has been protected throughout its message path.

Integrity

Integrity ensures that a message has not been altered since its departure from the original sender.

This guarantees that the state of the message contents remained intact from the time of transmission to the point of delivery.

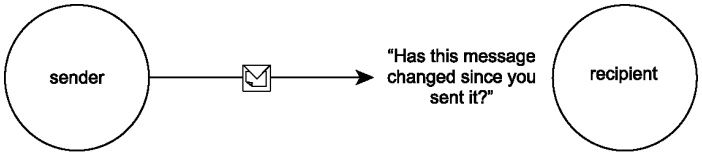
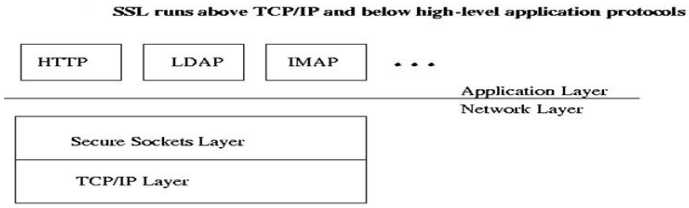


Figure 7.32: Integrity means ensuring that a message's contents have not changed during transmission.

Transport-level security and message-level security



The type of technology used to protect a message determines the extent to which the message remains protected while making its way through its message path.

Secure Sockets Layer (SSL), for example, is a very popular means of securing the HTTP channel upon which requests and responses are transmitted.

However, within a Web services-based communications framework, it can only protect a message during the transmission between service endpoints. Hence, SSL only affords us **transport-level security**

What Does SSL Concern?

- SSL server authentication.
- SSL client authentication. (optional)
- An encrypted SSL connection or Confidentiality. This protects against electronic eavesdropper.
- Integrity. This protects against hackers.

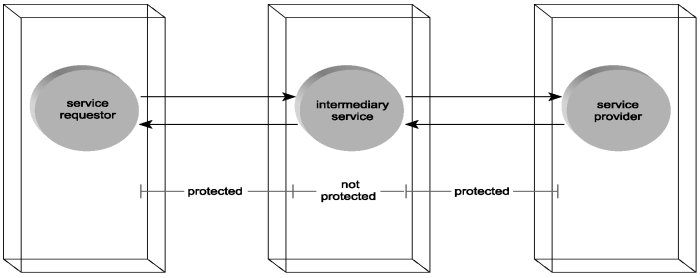


Figure 7.33: Transport-level security only protects the message during transit between service endpoints.

If, for example, a service intermediary takes possession of a message, it still may have the ability to alter its contents. To ensure that a message is fully protected along its entire message path, **message-level security is required**.

In this case, security measures are applied to the message itself (not to the transport channel on which the message travels). Now, regardless of where the message may travel, the security measures applied go with it.

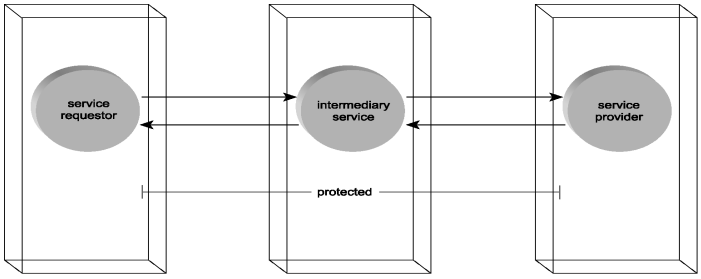


Figure 7-34: Message-level security guarantees end-to-end message protection

Encryption and digital signatures

Message-level confidentiality for an XML-based messaging format, such as SOAP, can be realized through the use of specifications that comprise the WS-Security framework.

In this section we focus on XML-Encryption and XML-Signature, two of the more important WS-Security extensions that provide security controls that ensure the confidentiality and integrity of a message.

XML-Encryption, an encryption technology designed for use with XML, is a cornerstone part of the WS-Security framework. It provides features with which encryption can be applied to an entire message or only to specific parts of the message (such as the password).

To ensure message integrity, a technology is required that is capable of verifying that the message received by a service is authentic in that it has not been altered in any manner since it first was sent.

XML-Signature provides features that allow for an XML document to be accompanied by a special algorithm-driven piece of information that represents a digital signature.

This signature is tied to the content of the document so that verification of the signature by the receiving service only will succeed if the content has remained unaltered since it first was sent.

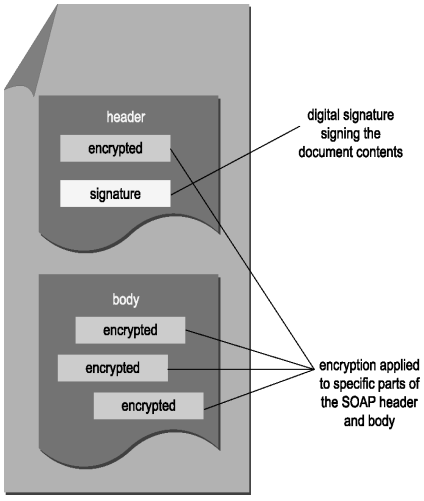


Figure 7.35: A digitally signed SOAP message containing encrypted data.

Security and SOA

Message-level security can clearly become a core component of service-oriented solutions. Security measures can be layered over any message transmissions to either protect the message content or the message recipient. The WS-Security framework and its accompanying specifications therefore fulfill fundamental QoS requirements that enable enterprises to: utilize service-oriented solutions for the processing of sensitive and private data restrict service access as required

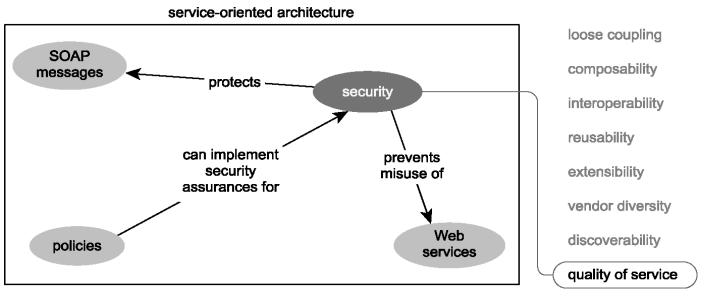


Figure 7.36: Security, as it relates to policies, SOAP messages, and Web services.

Notification and eventing

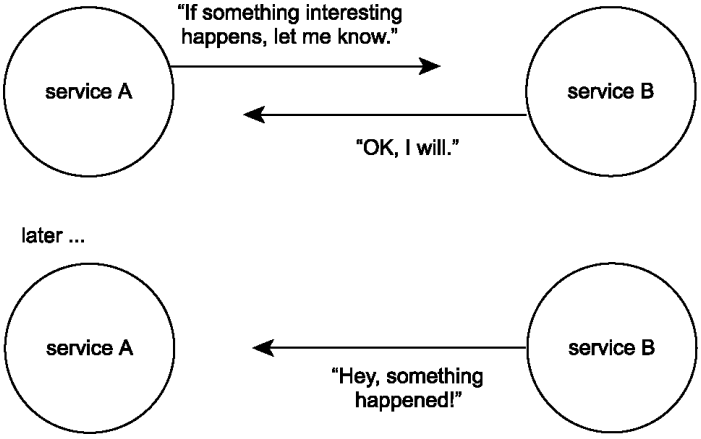


Figure 7.37: Once subscribed, service A is notified of anything service B publishes that is of interest to service A.

A subscriber is the part of the application that submits the subscribe request message to the notification producer.

This means that the subscriber is not necessarily the recipient of the notification messages transmitted by the notification producer.

The recipient is the notification consumer , the service to which the notification messages are delivered. A subscriber does not need to exist as a Web service, but the notification consumer is a Web service.

Both the subscriber and notification consumer roles can be assumed by a single Web service.
The subscriber is considered the service requestor .

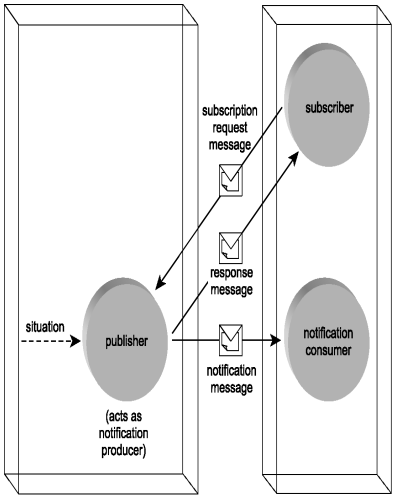


Figure 7.38: A basic notification architecture.

Notification broker, publisher registration manager, and subscription manager

- The **notification broker** A Web service that acts on behalf of the publisher to perform the role of the **notification producer**. This isolates the publisher from any contact with subscribers. Note that when a notification broker receives notification messages from the publisher, it temporarily assumes the role of notification consumer.

- The **publisher registration manager** A Web service that provides an interface for subscribers to search through and locate items available for registration. This role may be assumed by the notification broker, or it may be implemented as a separate service to establish a further layer of abstraction.
- The subscription manager A Web service that allows notification producers to access and retrieve required subscriber information for a given notification message broadcast. This role also can be assumed by either the notification producer or a dedicated service.

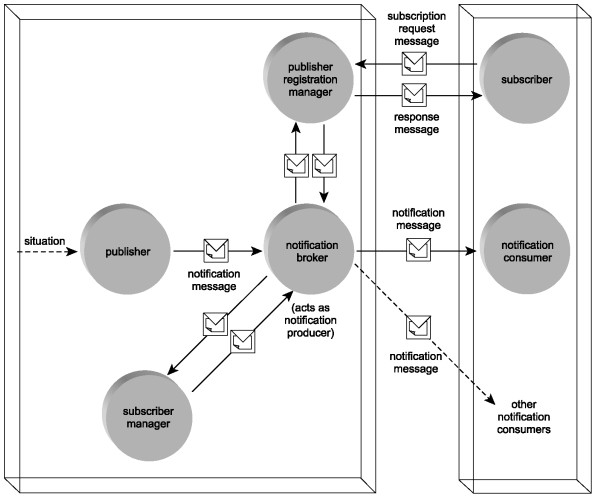
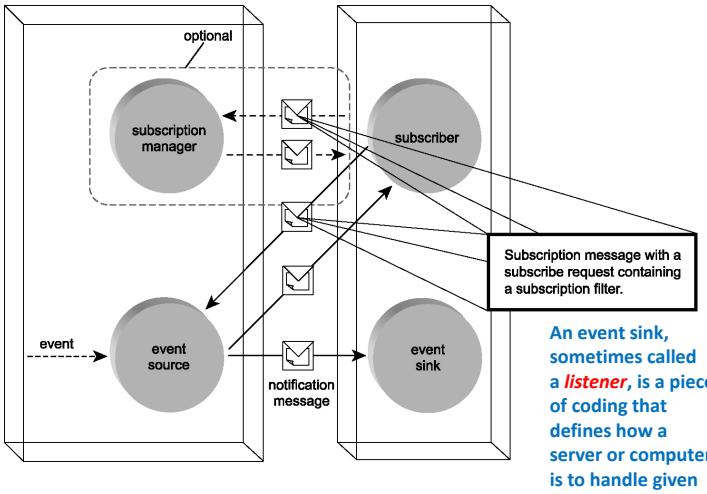


Figure 7.39: A notification architecture including a middle tier.



An event sink, sometimes called a *listener*, is a piece of coding that defines how a server or computer is to handle given events.

Figure 7.40: A basic eventing architecture.

By implementing a messaging model capable of supporting traditional publish-and-subscribe functionality, corresponding legacy features now can be fully realized within service-oriented application environments

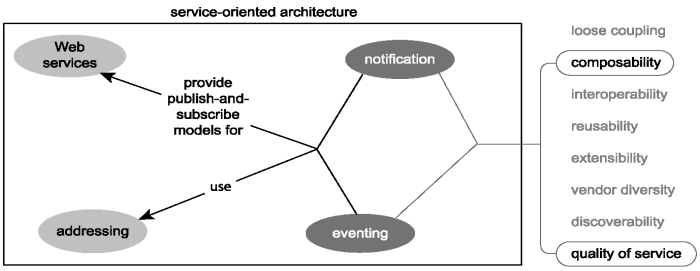


Figure 7.41: Notification and eventing establishing standardized publish-and-subscribe models within SOA.

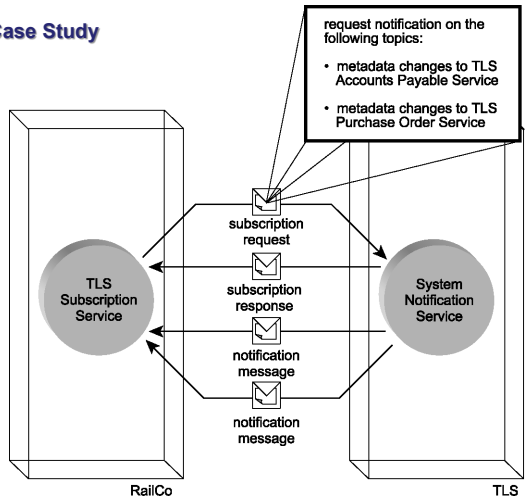


Figure 7.42: The new RailCo subscription service allows RailCo to receive notifications from the TLS System Notification Service.

Chapter 8

"Principles of Service Orientation"

8.1. Service-orientation and the enterprise

The collective logic that defines and drives an **enterprise** is an ever-evolving entity constantly changing in response to external and internal influences. From an IT perspective, this **enterprise logic** can be divided into two important halves:

1. **Business logic (BL)**
   Business logic is a documented implementation of the business requirements that oriented from an **enterprise's business** area. BL is generally structured into processes that express these requirements, along with any associated constraints, dependencies and outside influences.

2. **Application logic**
   Application logic is automatic implementation business logic organized into various technology solutions. **AL express business process workflows** through purchase customs-developed system within the confines of an organization`s IT infrastructure, security constraints, technical capabilities, and vendor dependencies .

- The business logic layer (BLL) contains logic specific to the business domain
- The **business layer** implements the **Domain Model** in a **boundary-technology-neutral** way. In other words, it doesn't depend on any particular UI or service interface-related technology, such as web libraries or windowing APIs. You should be able to consume the business layer from any type of application - web, rich client, web service, etc.
- The **application layer** bridges the gap between the business layer and the **boundary technology**.
- The application layer consists of those elements that are specific to this application. So that would contain the UI, back-end processing for the UI, and any bindings between the application and your business logic layer. In a perfect world, this layer **would not contain any logic of the business** domain.

The business and application logic domains

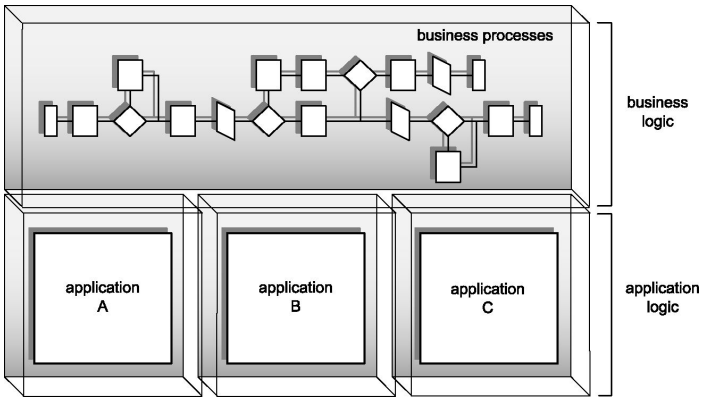


Figure 8.1: The business and application logic domains.

Enterprise Logic

Service Establish a high form of abstraction wedged between traditional business and application layers

Service can encapsulate **physical application** logic as well as **business process logic**

Service Establish a high form of abstraction wedged between traditional business and application layers

Service can encapsulate physical application logic as well as business process logic

Freed from proprietary ties, services can communicate via open protocols.

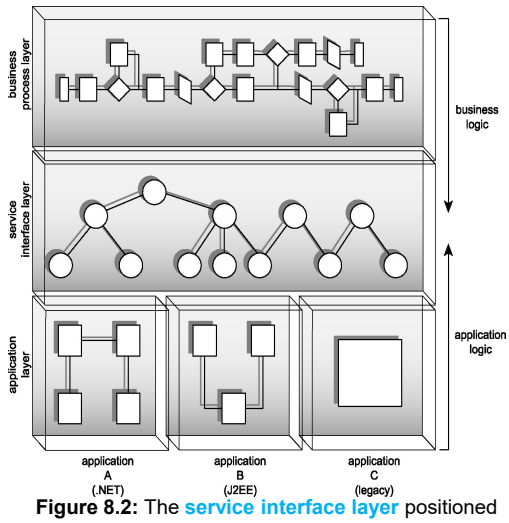


Figure 8.2: The **service interface layer** positioned between enterprise layers that promote application and business logic.

On a physical level, services are developed and deployed in proprietary environments, wherein they are individually responsible for the encapsulation of specific application logic.

Figure 8.3 shows how individual services, represented as service interfaces within the service interface layer, represent application logic originating from different platforms.

Individual service represents application logic originating from different platform

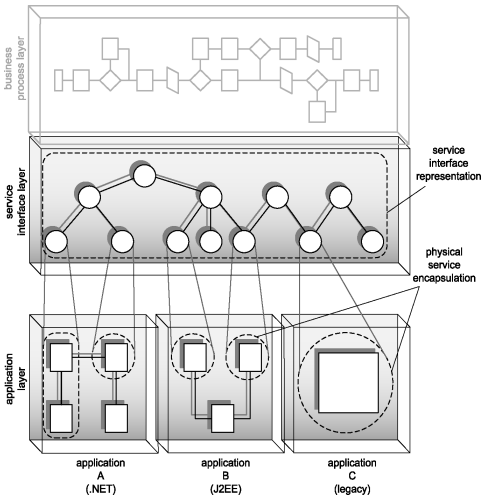


Figure 8.3: The service interface layer abstracts connectivity from service deployment environments.

8.2. Anatomy of a service-oriented architecture

1. Each Web service contains one or more operations.
2. Each operation governs the processing of a specific function the Web service is capable of performing.

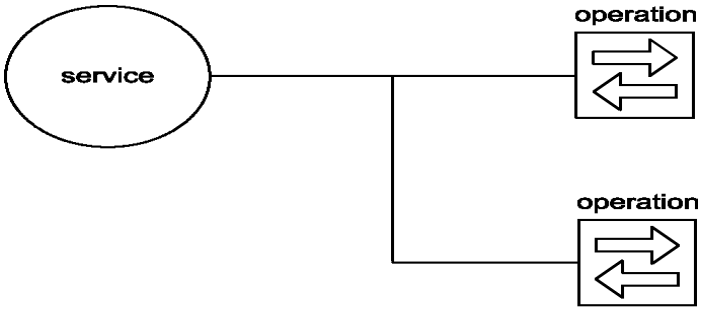


Figure 8.4: A Web service sporting two operations.

The processing consists of sending and receiving SOAP messages,

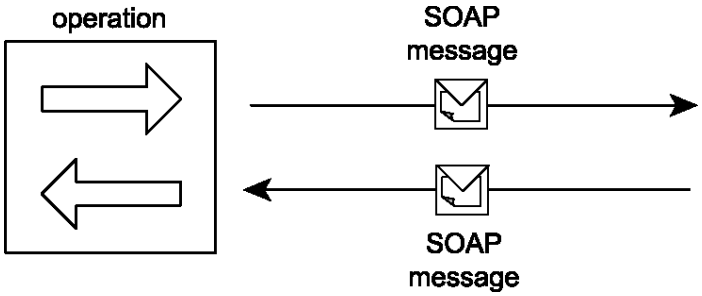


Figure 8.5: An operation processing outgoing and incoming SOAP messages.

By composing these parts (*service, operation and messaging*), Web services form an activity through which they can collectively automate a task

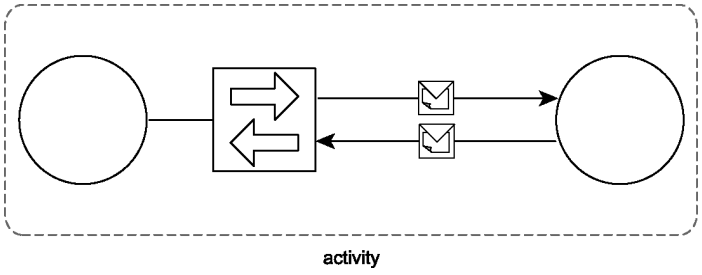


Figure 8.6: A basic communications scenario between Web services.

Logical components of automation logic

Fundamental parts of the framework	Rename Term
SOAP messages	messages
Web service operations	operations
Web services	services
activities	processes (and process instances)

messages = units of communication
operations = units of work
services = units of processing logic (collections of units of work)
processes = units of automation logic (coordinated aggregation of units of work)

We establish that **messages** are a suitable means by which all units of processing logic (services) communicate.

This illustrates that regardless of the scope of logic a service represents, no actual processing of that logic can be performed without issuing units of communication (in this case, messages).

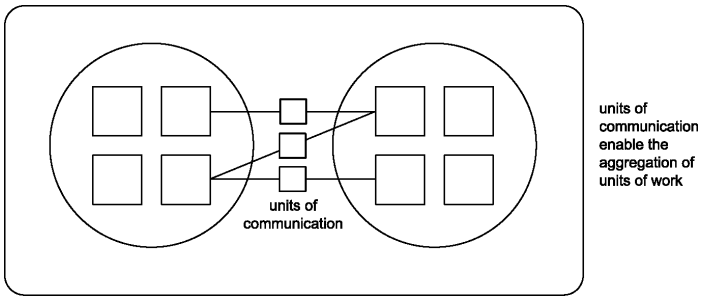


Figure 8.8: A primitive view of how units of communication enable interaction between units of logic.

Logic Component of Automation Logic

- The Purpose of these view is to express the process, service and operations.
- It also provides a flexible means of partitioning and modularizing the logic.
- These are the most basic concepts that underlies service orientation.

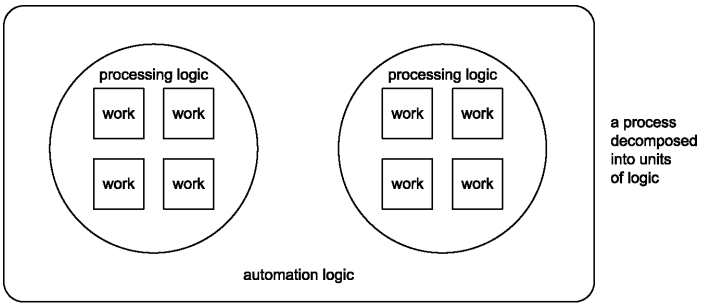


Figure 8.7: A primitive view of how SOA modularizes automation logic into units.

Components of an SOA

- A ***message*** represents the data required to complete some or all parts of a unit of work.
- An ***operation*** represents the **logic required to process** messages in order to complete a **unit of work**
- A ***service*** represents a logically grouped set of operations capable of performing related units of work.

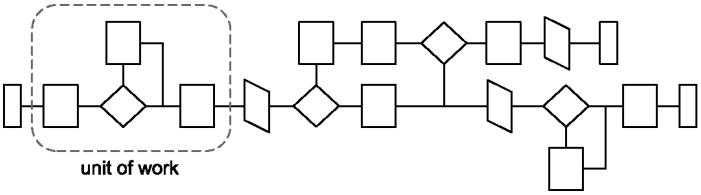


Figure 8.9: The scope of an operation within a process.

Process

A process contains the business rules that determine which service operations are used to complete a unit of automation.

In other words, a process represents a large piece of work that requires the completion of smaller units of work

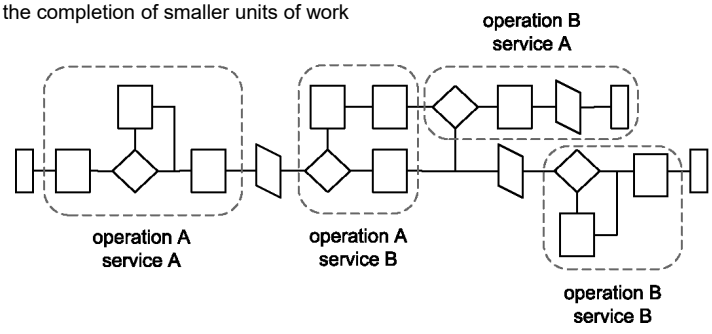


Figure 8.10: Operations belonging to different services representing various parts of process logic.

How components in an SOA inter-relate

Having established the core characteristics of our SOA components, let's now look at how these components are required to relate to each other:

- An operation sends and receives messages to perform work.
- An operation is therefore mostly defined by the messages it processes.
- A service groups a collection of related operations.
- A service is therefore mostly defined by the operations that comprise it.
- A process instance can compose services.
- A process instance is not necessarily defined by its services because it may only require a subset of the functionality offered by the services.
- A process instance invokes a unique series of operations to complete its automation.
- Every process instance is therefore partially defined by the service operations it uses.

How components in an SOA inter-relate

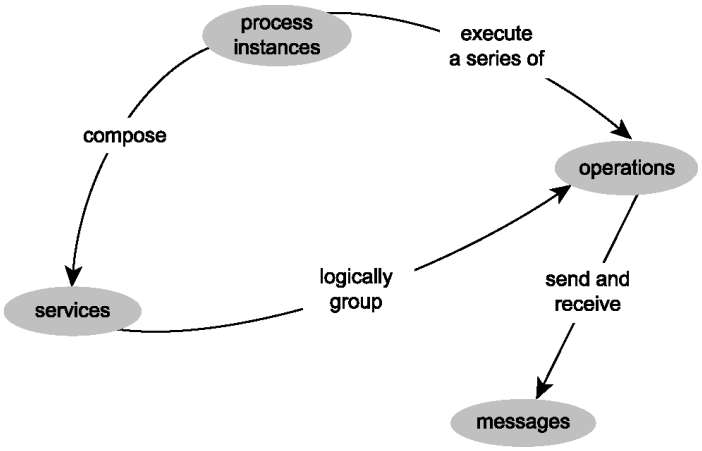


Figure 8.11: How the components of a service-oriented architecture relate.

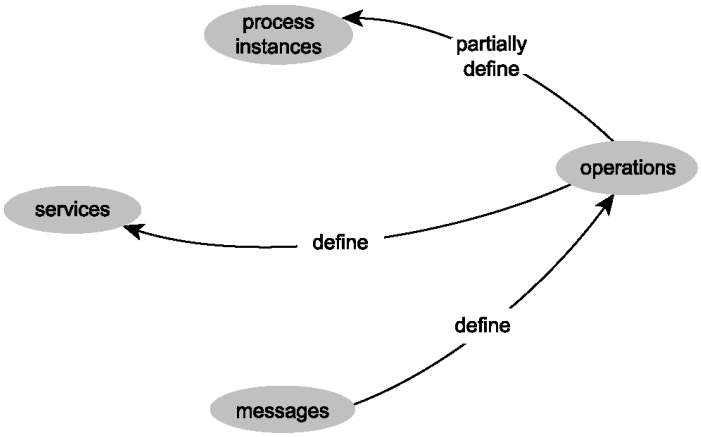


Figure 8.12: How the components of a service-oriented architecture define each other.

8.3. Common principles of service-orientation

MTTR MTBF

Principles of service-orientation

1. **Services are reusable** - Regardless of whether immediate reuse opportunities exist, services are designed to support potential reuse.
2. **Services share a formal contract** -For services to interact, they need not share anything but a formal contract that describes each service and defines the terms of information exchange.
3. **Services are loosely coupled-** Services must be designed to interact without the need for tight, cross-service dependencies.
4. **Services abstract underlying logic** -The only part of a service that is visible to the outside world is what is exposed via the service contract. Underlying logic, beyond what is expressed in the descriptions that comprise the contract, is invisible and irrelevant to service requestors.

A **service-level agreement** (**SLA**) is a part of a service contract where a service is formally defined. In practice, the term *SLA* is sometimes used to refer to the contracted delivery time (of the service or performance).

6. **Services are composable** -Services may compose other services. This allows logic to be represented at different levels of granularity and promotes reusability and the creation of abstraction layers .

7. **Services are autonomous**-The logic governed by a service resides within an explicit boundary. The service has control within this boundary and is not dependent on other services for it to execute its governance.

8. **Services are stateless**-Services should not be required to manage state information, as that can impede their ability to remain loosely coupled. Services should be designed to maximize statelessness even if that means deferring state management elsewhere.

9. **Services are discoverable** -Services should allow their descriptions to be discovered and understood by humans and service requestors that may be able to make use of their logic.

Services are reusable

Service-orientation encourages reuse in all services, regardless if immediate requirements for reuse exist.

By applying design standards that make each service potentially reusable, the chances of being able to accommodate future requirements with less development effort are increased.

Inherently reusable services also reduce the need for creating wrapper services that expose a generic interface over top of less reusable services.

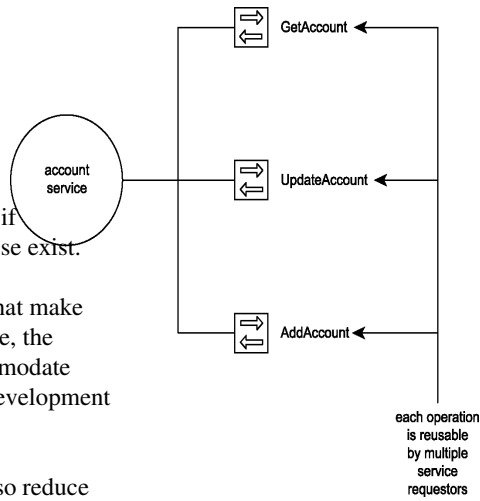


Figure 8.13: A reusable service exposes reusable operations.

Services share a formal contract

Service contracts provide a formal definition of:
- the service endpoint
- each service operation
- every input and output message supported by each operation
- rules and characteristics of the service and its operations

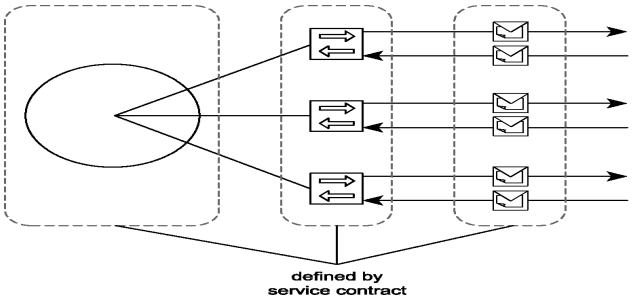


Figure 8.15: Service contracts formally define the service, operation, and message components of a service-oriented architecture.

Services are loosely coupled

Service must be designed to interact without the need for tight, cross-service dependencies.

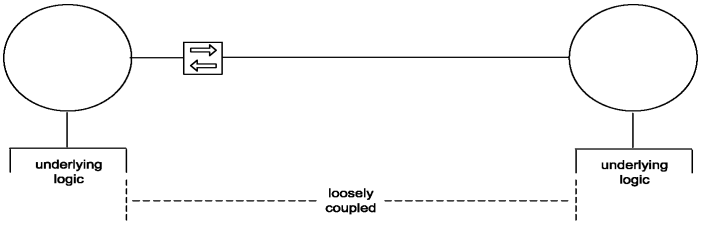


Figure 8.16: Services limit dependencies to the service contract, allowing underlying provider and requestor logic to remain loosely coupled.

Case Study

- TLS services are designed to communicate with multiple online venders make it loosely coupled
- Rail-Co services are designed to communicate only with TLS B2B system so is considered Less loosely coupled

Services abstract underlying logic

The only part of a service is visible to the outside world is what is exposed via service contract. Underlying logic, beyond what is expressed in the descriptions that comprise the contract, is invisible and irrelevant to service requestor.

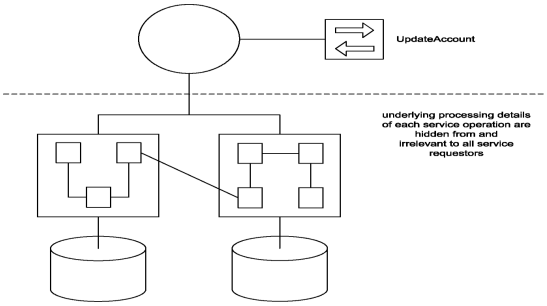


Figure 8.17: Service operations abstract the underlying details of the functionality they expose.

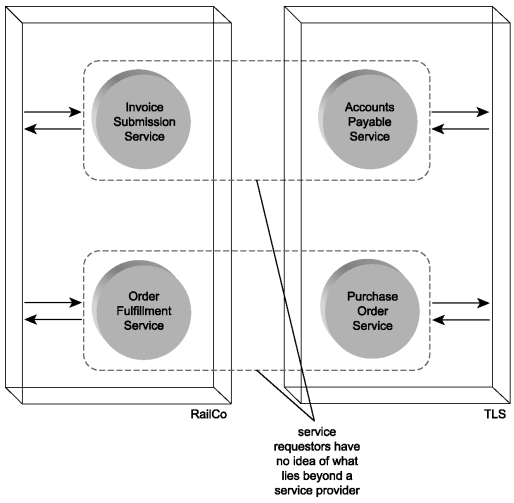


Figure 8.18: Neither of RailCo's or TLS's service requestors require any knowledge of what lies behind the other's service providers.

Services are composable

Service may be compose of other services. This allows logic to be represented at different levels of granularity and promotes reusability and the creation of abstraction layers.

Composability is simply another form of reuse, and therefore operations need to be designed in a standardized manner and with an appropriate level of granularity to maximize composition opportunities.

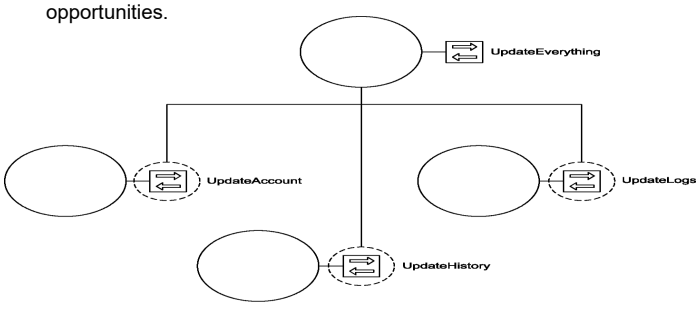


Figure 8.19: The UpdateEverything operation encapsulating a service composition.

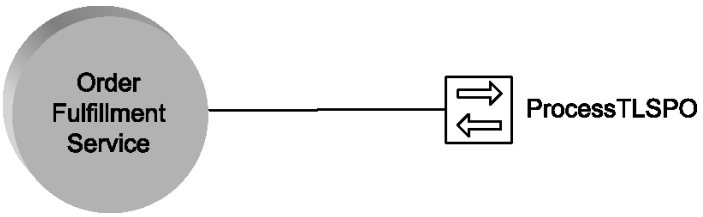


Figure 8.20: The RailCo Order Fulfillment Service with its one operation.

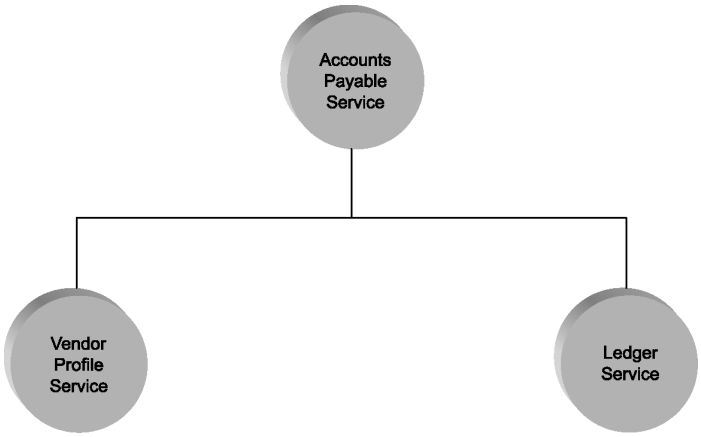


Figure 8.21: The TLS Accounts Payable Service composition.

Services are autonomous

Autonomy requires that the range of logic exposed by a service exist within an explicit boundary.

This allows the service to execute self-governance of all its processing.

It also eliminates dependencies on other services, which frees a service from ties that could inhibit its deployment and evolution

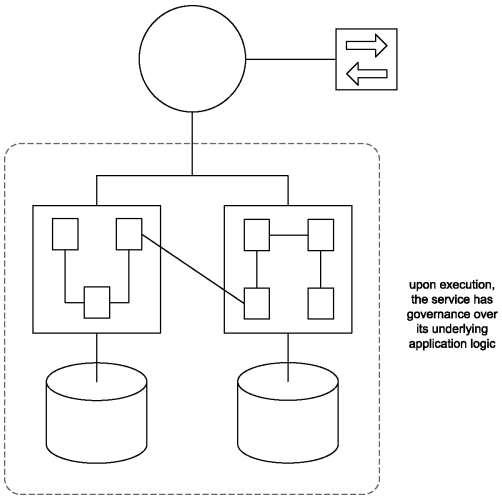


Figure 8.22: Autonomous services have **control** over underlying resources.

Case Study

Given the distinct tasks they perform, the following three RailCo services all are autonomous:

- Invoice Submission Service
- Order Fulfillment Service
- TLS Subscription Service.

The Invoice Processing and Order Fulfillment Services encapsulate legacy logic.
The legacy accounting system also is used by clients independently from the services, which makes this service-level autonomy.

The TLS Notification Service achieves pure autonomy, as it represents a set of custom components created only in support of this service.

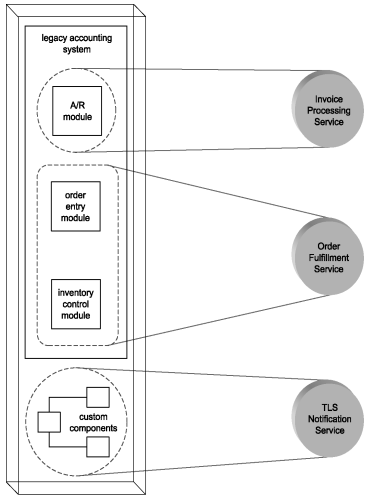


Figure 8.23: RailCo's services luckily encapsulate explicit portions of legacy and newly added application logic.

Services are stateless

Services should minimize the amount of state information they manage and the duration for which they hold it.

State information is data-specific to a current activity.

While a service is processing a message, for example, it is temporarily stateful.

If a service is responsible for retaining state for longer periods of time, its ability to remain available to other requestors will be impeded.

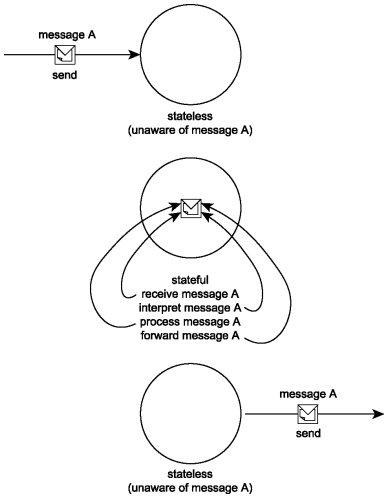


Figure 8.24: Stateless and stateful stages a service passes through while processing a message.

Services are discoverable

Case Study

Discovery helps avoid the accidental creation of redundant services or services that implement redundant logic.
On an SOA level, discoverability refers to the architecture's ability to provide a discovery mechanism, such as a service registry or directory.

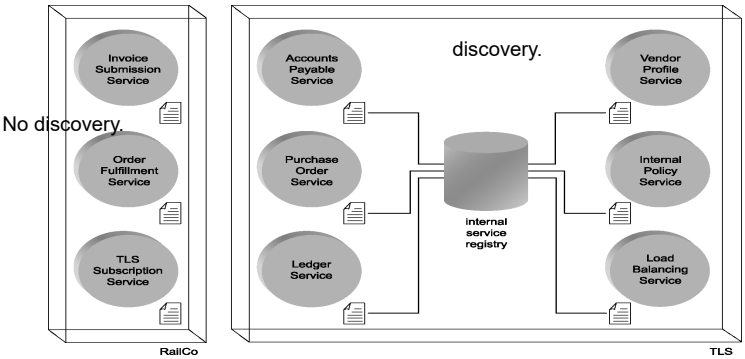


Figure 8.25: RailCo's services are not discoverable, but TLS's inventory of services are stored in an internal registry.

Service reusability

When a service encapsulates logic that is useful to more than one service requestor , it can be considered reusable.

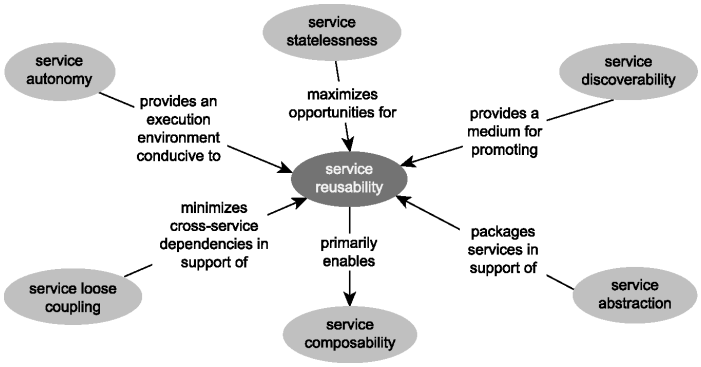


Figure 8.26: Service reusability and its relationship with other service-orientation principles.

Service reusability

The concept of reuse is supported by a number of complementary service principles, as follows.

- Service autonomy establishes an execution environment that facilitates reuse because the service has independence and self-governance. The less dependencies a service has, the broader the applicability of its reusable functionality.
- Service statelessness supports reuse because it maximizes the availability of a service and typically promotes a generic service design that defers activity-specific processing outside of service logic boundaries.
- Service abstraction fosters reuse because it establishes the black box concept, where processing details are completely hidden from requestors. This allows a service to simply express a generic public interface.
- Service discoverability promotes reuse, as it allows requestors (and those that build requestors) to search for and discover reusable services.
- Service loose coupling establishes an inherent independence that frees a service from immediate ties to others. This makes it a great deal easier to realize reuse.

The principle of service reuse itself enables the following related principle:

Service composability is primarily possible because of reuse. The ability for one service to compose an activity around the utilization of a collection of services is feasible when those services being composed are built for reuse.

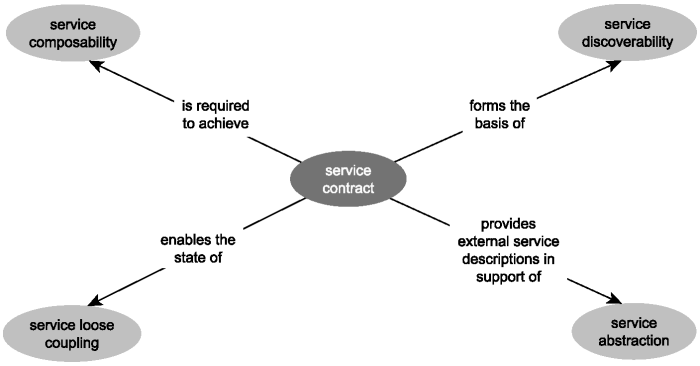


Figure 8.27: The service contract and its relationship with other service-orientation principles.

Service loose coupling

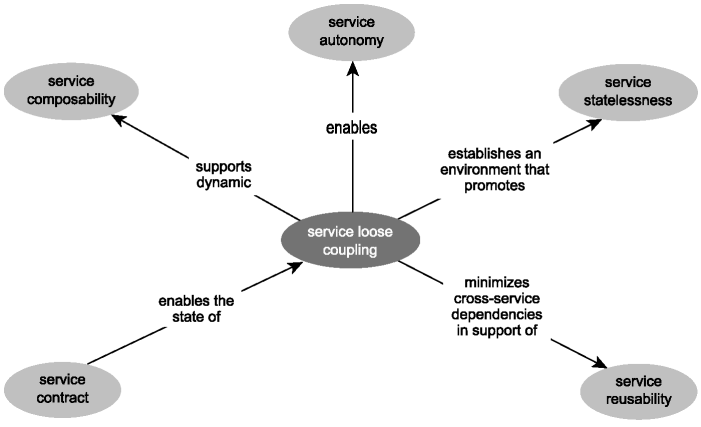


Figure 8.28: Service loose coupling and its relationship with other service-orientation principles.

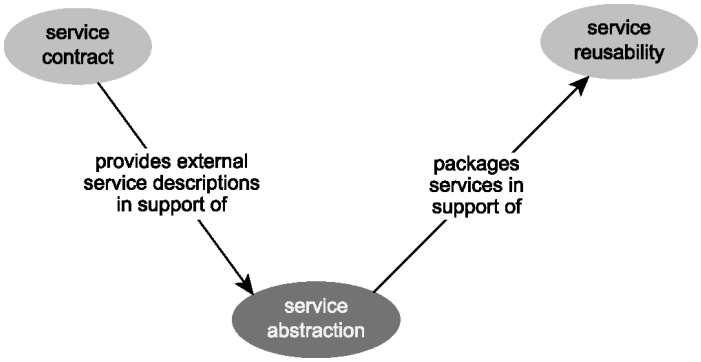


Figure 8.29: Service abstraction and its relationship with other service-orientation principles.

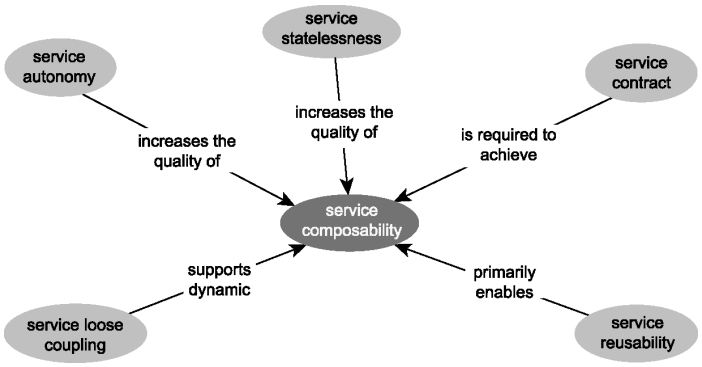


Figure 8.30: Service composability and its relationship with other service orientation principles.

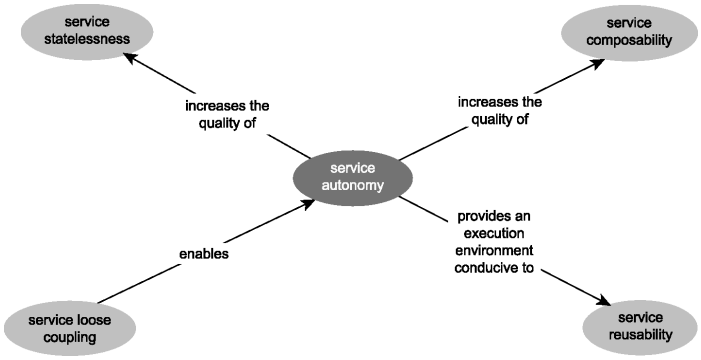


Figure 8.31: Service autonomy and its relationship with other service-orientation principles.

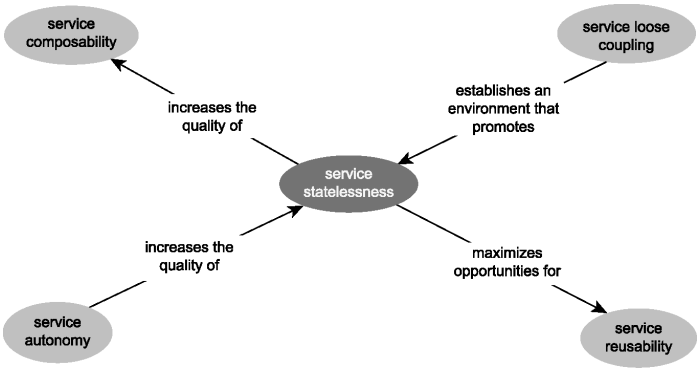


Figure 8.32: Service statelessness and its relationship with other service-orientation principles.

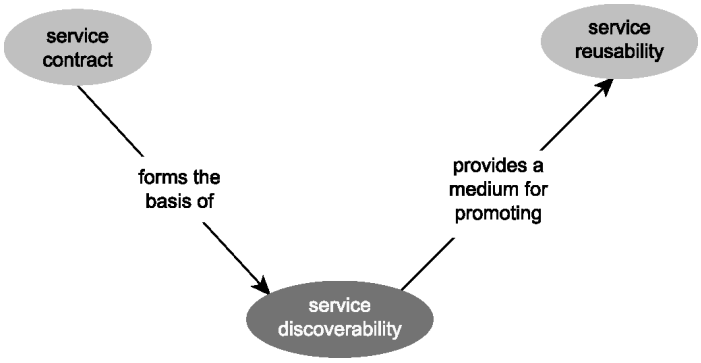


Figure 8.33: Service discoverability and its relationship with other service-orientation principles.

Service-orientation and object-orientation

- OOP is a fine way to create applications that perform specific tasks. However, it is a *terrible* architecture for creating robust network applications. Every **OOP-based architecture** for **distributed applications** (using CORBA, DCOM, or EJB) has failed to deliver on its promises... and made network programming vastly more difficult than it needed to be.
- A **service-oriented approach**, on the other hand, gives the developers and architects an **easy way to integrate systems**. They are far **less** complex, faster, **more robust**, more scalable, more easily maintained, and they are much easier to for designers to conceptualize.
- Does this mean an **end to OOP**? Good heavens, no. We still need OOP -- or at least significant chunks of it -- to create these systems. However, we should stay as far away from OOP as possible when creating interfaces to distributed applications.
- It will probably take several more years for SOAs to become common practice across the industry. And until then, people will demand an object-based API to **network applications**. The number of SOA converts has not yet reached a critical mass, so creating OOP interfaces still has value... if for no other reason than that's what people expect.

Service-orientation and object-orientation

Indeed, service-orientation owes much of its existence to object-oriented concepts and theory Table 8.1 provides a look at which common object-orientation principles are related to the service-orientation principles we've been discussing.

Service-Orientation Principle	Related Object-Orientation Principles
service reusability	Much of object-orientation is geared toward the creation of reusable classes. The object-orientation principle of modularity standardized decomposition as a means of application design. Related principles, such as abstraction and encapsulation, further support reuse by requiring a distinct separation of interface and implementation logic. Service reusability is therefore a continuation of this goal.
service contract	The requirement for a service contract is very comparable to the use of interfaces when building object-oriented applications. Much like WSDL definitions, interfaces provide a means of abstracting the description of a class. And, much like the "WSDL first" approach encouraged within SOA, the "interface first" approach also is considered an object-orientation best practice.

service loose coupling **multiple inheritance.jpg**	**Although the creation of interfaces somewhat decouples a class from its consumers, coupling in general is one of the primary qualities of service-orientation that deviates from object-orientation.** **The use of inheritance and other object-orientation principles encourages a much more tightly coupled relationship between units of processing logic when compared to the service-oriented design approach.**
service abstraction	**The object-orientation principle of abstraction requires that a class provide an interface to the external world and that it be accessible via this interface. Encapsulation supports this by establishing the concept of information hiding, where any logic within the class outside of what is exposed via the interface is not accessible to the external world.** **Service abstraction accomplishes much of the same as object abstraction and encapsulation. Its purpose is to hide the underlying details of the service so that only the service contract is available and of concern to service requestors.**
service composability	**Object-orientation supports association concepts, such as aggregation and composition. These, within a loosely coupled context, also are supported by service-orientation.** **For example, the same way a hierarchy of objects can be composed, a hierarchy of services can be assembled through service composability.**

service autonomy	**The quality of autonomy is more emphasized in service-oriented design than it has been with object-oriented approaches. Achieving a level of independence between units of processing logic is possible through service-orientation, by leveraging the loosely coupled relationship between services.** **Cross-object references and inheritance-related dependencies within object-oriented designs support a lower degree of object-level autonomy.**
service statelessness	**Objects consist of a combination of class and data and are naturally stateful. Promoting statelessness within services therefore tends to deviate from typical object-oriented design.** **Although it is possible to create stateful services and stateless objects, the principle of statelessness is generally more emphasized with service-orientation.**
service discoverability	**Designing class interfaces to be consistent and self-descriptive is another object-orientation best practice that improves a means of identifying and distinguishing units of processing logic. These qualities also support reuse by allowing classes to be more easily discovered.** **Discoverability is another principle more emphasized by the service-orientation paradigm. It is encouraged that service contracts be as communicative as possible to support discoverability at design time and runtime.**

Native Web service support for service-orientation principles

Service-Orientation Principle	Web Service Support
service reusability	Web services are not automatically reusable. This quality is related to the nature of the logic encapsulated and exposed via the Web service.
service contract	Web services require the use of service descriptions, making service contracts a fundamental part of Web services communication.
service loose coupling	Web services are naturally loosely coupled through the use of service descriptions.
service abstraction	Web services automatically emulate the black box model within the Web services communications framework, hiding all of the details of their underlying logic.

service composability	**Web services are naturally composable. The extent to which a service can be composed, though, generally is determined by the service design and the reusability of represented logic.**
service autonomy	**To ensure an autonomous processing environment requires design effort. Autonomy is therefore not automatically provided by a Web service.**
service statelessness	**Statelessness is a preferred condition for Web services, strongly supported by many WS-* specifications and the document-style SOAP messaging model.**
service discoverability	**Discoverability must be implemented by the architecture and even can be considered an extension to IT infrastructure. It is therefore not natively supported by Web services.**

Principles not supported by Web Service automatically

Need special attention when building service-oriented solutions. The four principles identified as ***not* being automatically** provided by Web services are:

- service reusability
- service autonomy
- service statelessness
- service discoverability

THANK YOU

Chapter 9. Service Layers

9.1. Service-orientation and contemporary SOA

Contemporary SOA is a complex and sophisticated architectural platform that offers significant potential to solve many historic and current IT problems.

Three of the primary influences of contemporary SOA.

- first-generation Web services concepts
- second-generation (WS-*) Web Services concepts
- principles of service-orientation

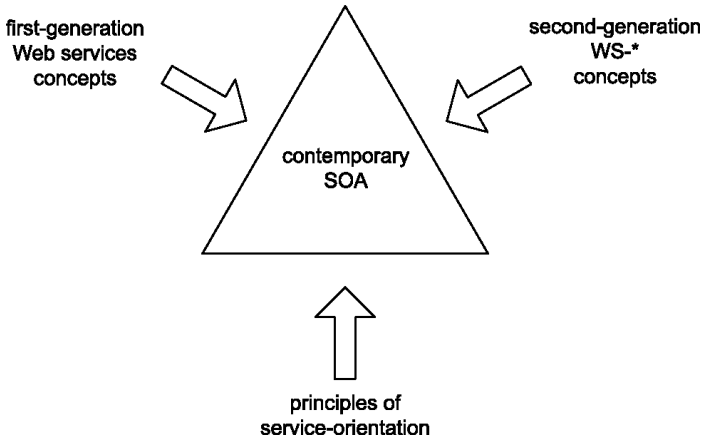


Figure 9.1: External influences that form and support contemporary SOA.

SOA characteristics are influenced by Web services specifications and service-orientation principles.

Characteristic	Origin/Supporting Source
fundamentally autonomous	**Autonomy is one of the core service-orientation principles that can be applied to numerous parts of SOA. Pursuing autonomy when building and assembling service logic supports other SOA characteristics.**
based on open standards	This is a natural by-product of basing SOA on the Web services technology platform and its ever-growing collection of WS-* specifications. The majority of Web services specifications are open and vendor-neutral.
QoS capable	The quality of service improvements provided by contemporary SOA are, for the most part, realized via vendor implementations of individual WS-* extensions.
Architecturally composable	While composability, on a service level, is one of our service-orientation principles, for an architecture to be considered composable requires that the technology from which the architecture is built support this notion.

vendor diversity	**This is really more of a benefit of SOA than an actual characteristic. Regardless, it is primarily realized through the use of the open standards provided by the Web services platform.**
intrinsic interoperability	The standardized communications framework provided by Web services establishes the potential to foster limitless interoperability between services. This is no big secret. To foster intrinsic interoperability among services, though, requires forethought and good design standards. Although supported by a number of WS-* specifications, this characteristic is not directly enabled by our identified influences.
discoverability	Service-level discoverability is one of our fundamental principles of service-orientation. Implementing discoverability on an SOA level typically requires the use of directory technologies, such as UDDI (one of the first-generation Web services specifications).

promotes federation	**Federation is a state achieved by extending SOA into the realm of service-oriented integration. A number of key WS-* extensions provide feature-sets that support the attainment of federation. Most notable among these are the specifications that implement the concepts of orchestration and choreography.**
inherent reusability	Reusability is one of the primary principles of service-orientation and one that can be applied across service-oriented solution environments. SOA promotes the creation of inherently reusable service logic within individual services and across service compositionsa benefit attainable through quality service design.
extensibility	Given that Web services are composable and based on open standards, extensibility is a natural benefit of this platform. Several WS-* extensions introduce architectural mechanisms that build extensibility into a solution. However, for the most part, this is a characteristic that must be intentionally designed into services individually and into SOA as a whole.

service-oriented business modeling	**This key characteristic is supported by orchestration, although not automatically. WS-* specifications, such as WS-BPEL, provide a dialect capable of expressing business process logic in an operational syntax resulting in a process definition. Only throughdeliberate design, though, can these types of process definitions actually be utilized to support service-oriented business modeling.**
layers of abstraction	Service-orientation principles fully promote black box-type abstraction on a service interface level. However, to coordinate logic abstraction into layers, services must be designed and organized according to specific design standards.
enterprise-wide loose coupling	Loose coupling is one of the fundamental characteristics of Web services. Achieving a loosely coupled enterprise is a benefit expected from the coordinated proliferation of SOA and abstraction layers throughout an organization's business and application domains.

organizational agility	**Though the use of Web services, service-orientation principles, and WS-* specifications support the concept of increasing an organization's agility, they do not directly enable it. This important characteristic requires dedicated analysis and design and relies on the realization of other SOA characteristics.**

9.1.2. Unsupported SOA characteristics

Having removed the concrete SOA characteristics that receive support from our identified external influences, we are now left with the following six:

1. intrinsic interoperability
2. extensibility
3. enterprise-wide loose coupling
4. service-oriented business modeling
5. organizational agility
6. layers of abstraction

The first two are somewhat enabled by different WS-* extensions.

Section 9.2. Service layer abstraction

In our familiar enterprise model, the service interface layer is located between the business process and application layers . This is where service connectivity resides and is therefore the area of our enterprise wherein the characteristics of SOA are most prevalent . To implement the characteristics we just identified in an effective manner, some larger issues need to be addressed.

Specifically , we need to answer the following questions:

- What logic should be represented by services?
- How should services relate to existing application logic?
- How can services best represent business process logic?
- How can services be built and positioned to promote agility?

Problems solved by layering services

- **What logic should be represented by services?**
  - In the previous chapter we established that enterprise logic can be divided into two primary domains: **business logic** and **application logic**. Services can be modeled to represent either or both types of logic, as long as the principles of service-orientation can be applied.
  - However, to achieve enterprise-wide loose coupling (the first of our four outstanding SOA characteristics) physically separate layers of services are, in fact, required. When individual collections of services represent corporate business logic and **technology-specific** application logic, each domain of the enterprise is freed of direct dependencies on the other.
  - This allows the automated representation of business process logic to evolve independently from the technology-level application logic responsible for its execution. In other words, this establishes a loosely coupled relationship between business and application logic.

How should services relate to existing application logic?

- Much of this depends on whether existing legacy application logic needs to be exposed via services or whether new logic is being developed in support of services. Existing systems can impose any number of constraints, limitations, and environmental requirements that need to be taken into consideration during service design.
- Applying a service layer on top of legacy application environments may even require that some service-orientation principles be compromised. This is less likely when building solutions from the ground up with service layers in mind, as this affords a level of control with which service-orientation can be directly incorporated into application logic.
- Either way, services designed specifically to represent application logic should exist in a separate layer. We'll therefore simply refer to this group of services as belonging to the *application service layer*.

• How can services best represent business logic?

- Business logic is defined within an organization's business models and business processes.
  - When modeling services to represent business logic, it is most important to ensure that the service representation of this logic is in alignment with existing business models.
- It is also useful to separately categorize services that are designed in this manner. Therefore, we'll refer to services that have been modeled to represent business logic as belonging to the *business service layer*.
  - By adding a business service layer, we also implement the second of our four SOA characteristics, which is support for service-oriented business modeling.

• How can services be built and positioned to promote agility?

- The key to building an agile SOA is in minimizing the dependencies each service has within its own processing logic. Services that contain business rules are required to enforce and act upon these rules at runtime.
- This limits the service's ability to be utilized outside of environments that require these rules. Similarly, controller services that are embedded with the logic required to compose other services can develop dependencies on the composition structure.

- Abstraction is the key
  The three layers of abstraction we identified for SOA are:

  1. the application service layer
  2. the business service layer
  3. the orchestration service layer

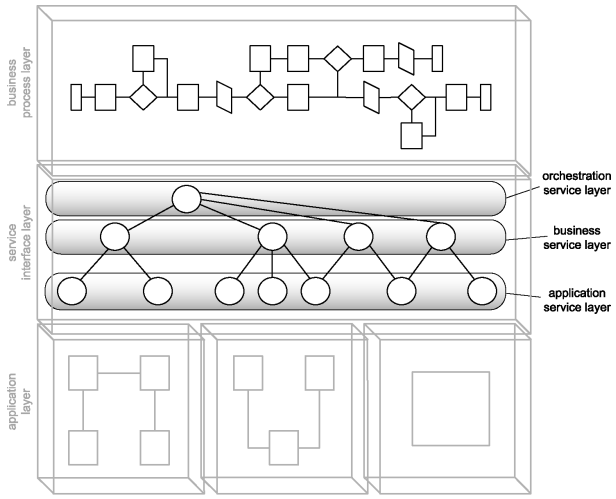


Figure 9.2: The three primary service layers.

9.3. Application service layer

- The application service layer establishes the ground level foundation that exists to express **technology-specific functionality**.
- Services that reside within this layer can be referred to simply as application services.
- Their purpose is to provide reusable functions related to processing data within new or legacy application environments.

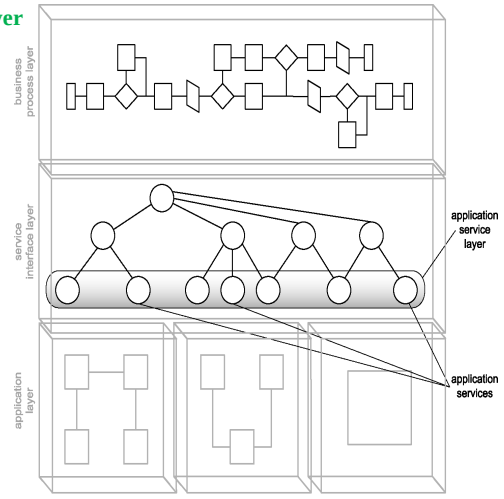


Figure 9.3: The application service layer.

Application services commonly have the following characteristics:

1. They expose functionality within a specific processing context
2. they draw upon available resources within a given platform
3. they are solution- agnostic
4. they are generic and reusable
5. they can be used to achieve point-to-point integration with other application services
6. they are often inconsistent in terms of the interface granularity they expose
7. they may consist of a mixture of custom-developed services and third-party services that have been purchased or leased
8. Typical examples of service models implemented as application services include the following:
9. utility service
10. wrapper service

Utility computing is the packaging of computing resources, such as computation, storage and services, as a **metered service**. This model has the advantage of a low or no initial cost to acquire computer resources; instead, computational resources are essentially rented.

Arguably the entire point about SOA is that everything is a wrapper. That is , a web service is just a facade, and there is no way to tell whether it fronts a "legacy application" or "a set of complex services".

a wrapper service is a service that wraps *something*, be it another service, a set of services, or a legacy application.

"wrapper service" which is also more widely known as "legacy wrapper" is used to wrap a legacy component to prevent coupling issues in soa.

Case Study

TLS has a well-defined application services layer. Of the TLS services we've discussed so far in our case study, the following are considered application services:

1. Load Balancing Service
2. Internal Policy Service
3. System Notification Service

RailCo has following application services layers:

1. Invoice Submission Service
2. Order Fulfillment Service
3. TLS Subscription Service

Section 9.4. Business service layer

While application services are responsible for representing *technology and application logic*, the business service layer introduces a service concerned solely with representing *business logic*, called the *business service.*

Business services are the lifeblood of contemporary SOA.

They are responsible for expressing business logic through service-orientation and bring the representation of corporate business models into the Web services arena.

Business services, are always an implementation of the business service model.
The sole purpose of business services intended for a separate business service layer is to represent business logic in the purest form possible.
This does not, however, prevent them from implementing other service models. For example, a business service also can be classified as a **controller service and a utility service**.

In fact, when application logic is abstracted into a separate application service layer, it is more than likely that business services will act as controllers to compose available application services to execute their business logic.

Business service layer abstraction leads to the creation of two further business service models:
- ***Task-centric business service*** A service that encapsulates business logic specific to a task or business process. This type of service generally is required when business process logic is not centralized as part of an orchestration layer. Task-centric business services have limited reuse potential.
- ***Entity-centric business service*** A service that encapsulates a specific business entity (such as an invoice or timesheet). Entity-centric services are useful for creating highly reusable and business process-agnostic services that are composed by an orchestration layer or by a service layer consisting of task-centric business services (or both).

When a separate application service layer exists, these two types of business services can be positioned to compose application services to carry out their business logic.

HYBRID SERVICE

hybrid service is actually a service that contains both business and application logic.
It is therefore often referred to as a type of business service.
For the purpose of establishing specialized service layers, we consider the business service layer reserved for services that abstract business logic only.
We therefore classify the **hybrid service** as a **variation of an application service, making it a resident of the application service layer**.

Case Study

Of the TLS services we've discussed so far in our case study examples, the following are true business services:

1. Accounts Payable Service
2. Purchase Order Service
3. Ledger Service
4. Vendor Profile Service

Each represents a well-defined boundary of business logic, and whenever one of these service's operations needs to perform a task outside of this boundary, it reuses functionality provided in another business or application service.

RailCo's Invoice Submission and Order Fulfillment Services are hybrid in that they contain both business rules and application-related processing.

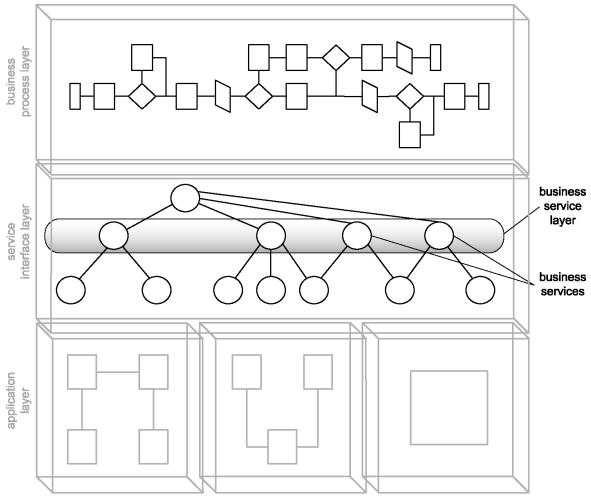


Figure 9.4: The business service layer.

Section 9.5. Orchestration service layer

- The orchestration service layer introduces a parent level of abstraction that alleviates the need for other services to manage interaction details required to ensure that service operations are executed in a specific sequence.
- Within the orchestration service layer, process services compose other services that provide specific sets of functions, independent of the business rules and scenario-specific logic required to execute a process instance.

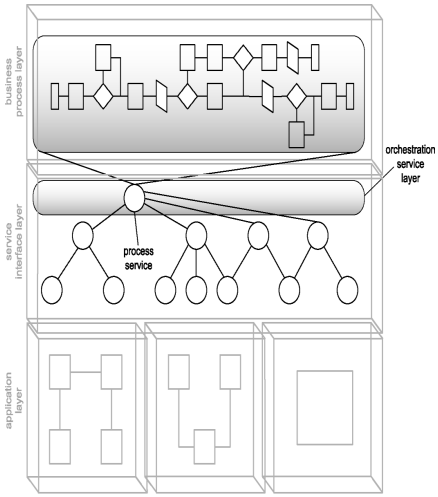


Figure 9.5: The orchestration service layer.

Agnostic services

key aspect of delivering reusable services is that they introduce service layers that are not limited to a single process or solution environment. It is important to highlight this one point, as it can blur the architectural boundary of a service-oriented solution.

An application-level SOA containing solution-agnostic services does, in fact, extend beyond the application. And, in the same manner, an application-level SOA that depends on the use of existing solution-agnostic services also does not have a well defined application boundary.

Entity-centric business services are designed to provide a set of features that provide data management related only to their corresponding entities. They are therefore business process-agnostic. The same entity-centric business services can (and should) be reused by different process or task-centric business services..

Application services ideally are built according to the utility service model. This makes them highly generic, reusable, and very much solution-agnostic. Different service-oriented solutions can (and should) reuse the same application services.

services can be process- and solution-agnostic while still being used as part of a service layer that connects different processes and solutions.

An enterprise that invests heavily in agnostic services easily can end up with an environment in which a great deal of reuse is leveraged. This is the point at which building service-oriented solutions can become more of a modeling exercise and less of an actual development project.

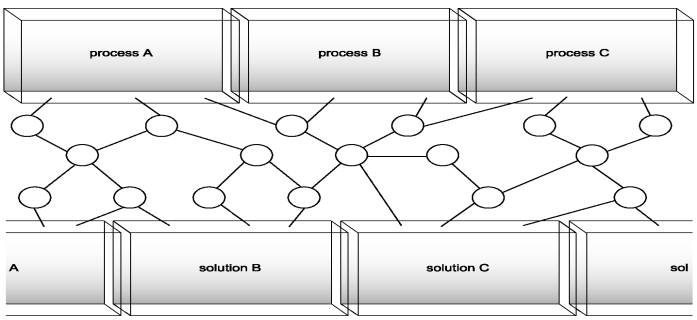


Figure 9.6: Services uniting previously isolated business processes and solution environments.

9.7. Service layer configuration scenarios

we will explore scenarios based on the use of the following types of services:

- **Hybrid application** services (services containing both business process and application logic)
- **Utility application** services (services containing reusable application logic)
- **Task-centric business** services (services containing business process logic)
- **Entity-centric business** services (services containing entity business logic)
- **Process services** (services representing the orchestration service layer)

Scenario #1: Hybrid application services only

When Web services simply are appended to existing **distributed** application environments, or when a **Web services-based solution** is built without any emphasis on **reuse or service-oriented business modeling**, the resulting architecture tends to consist of a set of ***hybrid application services***

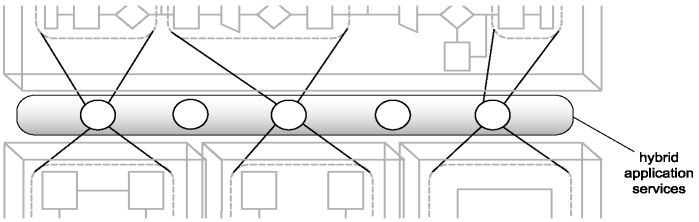


Figure 9.7: Hybrid services encapsulating both business and application logic.

Scenario #2: Hybrid and utility application services.

A Web services-based architecture consisting of hybrid services and reusable application services.
In this case, the hybrid services may compose some of the reusable application services. This configuration achieves an extent of abstraction, as the utility services establish a solution- agnostic application layer

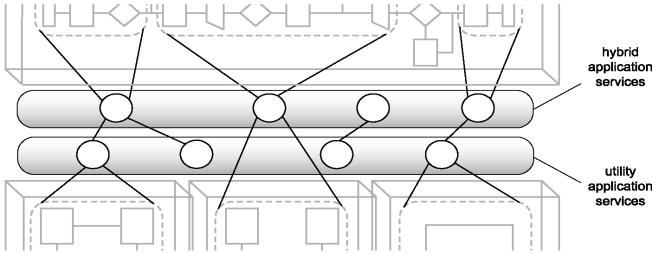


Figure 9.8: Hybrid services composing available utility application services.

Scenario #3: Task-centric business services and utility application services.

- This approach results in a more coordinated level of abstraction, given that business process logic is entirely represented by a layer of task-centric business services.
- These services rely on a layer of application services for the execution of all their business logic

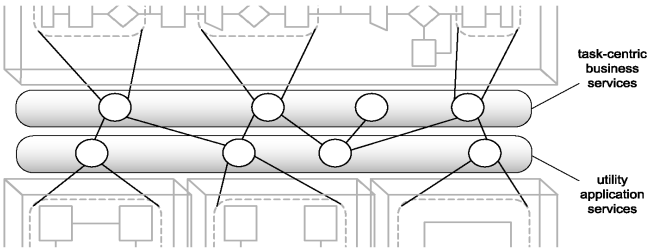


Figure 9.9: Task-centric business services and utility application services cleanly abstracting business and application logic.

Scenario #4: Task-centric business services, entity-centric business services, and utility application services.

Here we've added a further layer of abstraction through the introduction of entity-centric business services. This positions task-centric services as parent controllers that may compose both entity-centric and application services to carry out business process logic

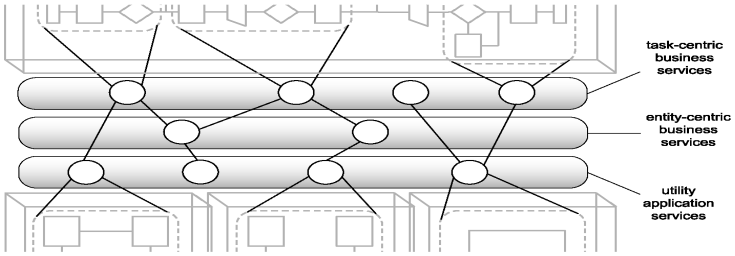


Figure 9.10: Two types of business services composing application services.

Scenario #5: Process services, hybrid application services, and utility application services.

In this scenario, a parent process service composes available hybrid and application services to automate a business process.
This is a common configuration when adding an orchestration layer over the top of an older distributed computing architecture that uses Web services

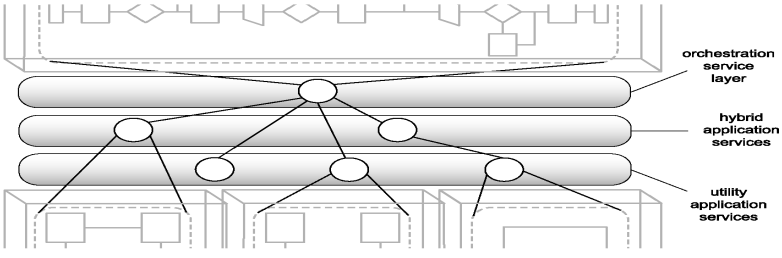


Figure 9.11: An orchestration layer providing a process service that composes different types of application services.

Scenario #6: Process services, task-centric business services, and utility application services

Even though task-centric services also contain business process logic, a parent process service can still be positioned to compose both task-centric and utility application services.
Though only partial abstraction is achieved via task-centric services, when combined with the centralized business process logic represented by the process services, business logic as a whole is still abstracted

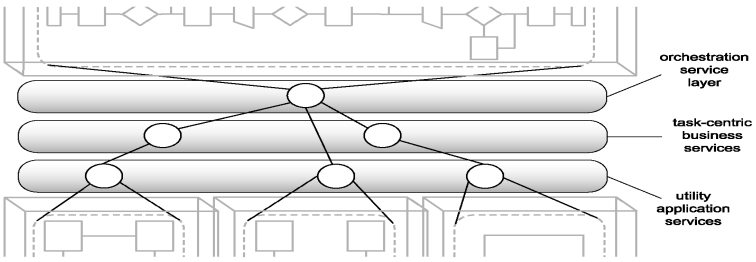


Figure 9.12: A process service composing task-centric and utility application services

Scenario #7: Process services, task-centric business services, entity-centric business services, and utility application services.

This variation also introduces entity-centric business services, which can result in a configuration that actually consists of four service layers. Sub-processes can be composed by strategically positioned task-centric services, whereas the parent process is managed by the process service, which composes both task-centric and entity-centric services

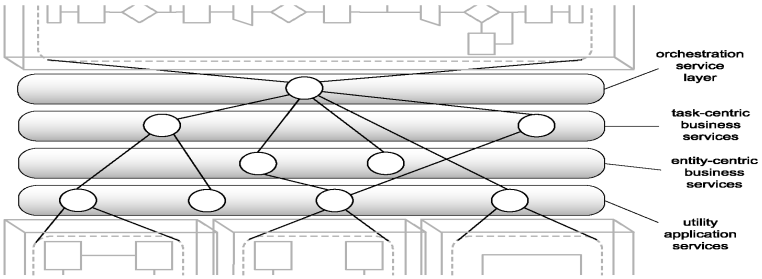


Figure 9.13: Process and business services composing utility application services.

Scenario #8: Process services, entity-centric business services, and utility application services

This SOA model establishes a clean separation of business and application logic, while maximizing reuse through the utilization of solution and business process-agnostic service layers.

The process service contains all business process-specific logic and executes this logic through the involvement of available entity-centric business services, which, in turn , compose utility application logic to carry out the required tasks

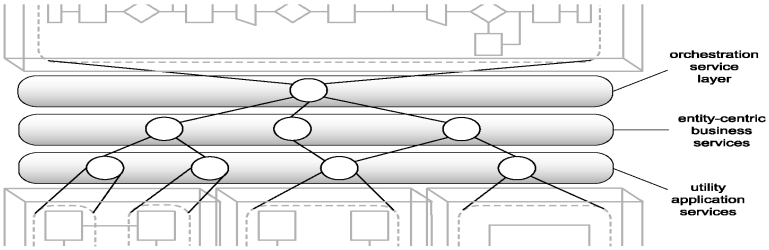


Figure 9.14: A process service composing entity-centric services, which compose utility application services.

Part IV: Building SOA (Planning and Analysis)

Chapter 10. SOA Delivery Strategies

Chapter Overview

SOA delivery lifecycle phases

Typical System Development Phases

- ❑Service-oriented analysis
- ❑Service-oriented design
- ❑Service development
- ❑Service testing
- ❑Service deployment
- ❑Service administration

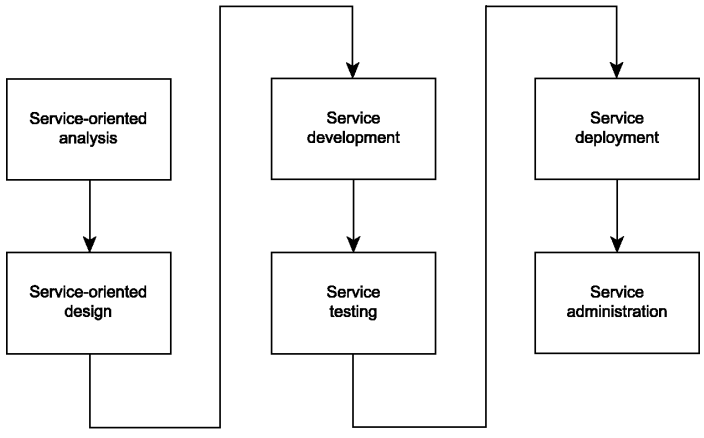


Figure 10.1: Common phases of an SOA delivery lifecycle.

Service-oriented analysis

It is in this initial stage that we determine the potential scope of our SOA. Service layers are mapped out, and individual services are modeled as service candidates that comprise a preliminary SOA.

A formal step-by-step service modeling process is provided as part of the two chapters (11(See 11.2) and 12(See 11.3)) dedicated to the service-oriented analysis phase.

Service-oriented design

- When we know what it is we want to build, we need to determine how it should be constructed. Service-oriented design is a heavily standards-driven phase that incorporates industry conventions and service-orientation principles into the service design process.
- This phase, therefore, confronts service designers with key decisions that establish the hard logic boundaries encapsulated by services. The service layers designed during this stage can include the **orchestration layer**, which results in a formal business process definition.
- Four formal step-by-step design processes are provided within the four chapters (13(See 12.1) to 16(See 12.4)) dedicated to the service-oriented design phase.

Service development

- Next, of course, is the actual construction phase. Here development platform-specific issues come into play, regardless of service type.
- Specifically, the choice of programming language and development environment will determine the physical form services and orchestrated business processes take, in accordance with their designs.
- As part of our coverage of SOA platforms, we explore development and runtime technologies associated with the .NET and J2EE platforms in Chapter 18

Service testing

Given their generic nature and potential to be reused and composed in unforeseeable situations, services are required to undergo rigorous testing prior to deployment into a production environment.

Below is a sampling of some of the key issues facing service testers:

1. What types of service requestors could potentially access a service?
2. Can all service policy assertions be successfully met?
3. What types of exception conditions could a service be potentially subjected to?
4. How well do service descriptions communicate service semantics?
5. Do revised service descriptions alter or extend previous versions?
6. How easily can the services be composed?

- How easily can the service descriptions be discovered?
- Is compliance to WS-I profiles required?
- What data typing-related issues might arise?
- Have all possible service activities and service compositions been mapped out?
- Do all new services comply with existing design standards?
- Do new services introduce custom SOAP headers? And, if yes, are all potential requestors (including intermediaries) required to do so, capable of understanding and processing them?
- Do new services introduce functional or QoS requirements that the current architecture does not support?

Service deployment

The implementation stage brings with it the joys of installing and configuring distributed components, service interfaces, and any associated middleware products onto production servers.
Typical issues that arise during this phase include:

1.How will services be distributed?
2.Is the infrastructure adequate to fulfill the processing requirements of all services?
3.How will the introduction of new services affect existing services and applications?
4.How should services used by multiple solutions be positioned and deployed?
5.How will the introduction of any required middleware affect the existing environment?
6.What security settings and accounts are required?
7.How will service pools be maintained to accommodate planned or unforeseen scalability requirements?

SERVICE ADMINISTRATION

After services are deployed, standard application management issues come to the forefront. These are similar in nature to the administration concerns for distributed, component-based applications, except that they also may apply to services as a whole (as opposed to services belonging to a specific application environment).

Issues frequently include:

1. How will service usage be monitored?
2. What form of version control will be used to manage service description documents?
3. How will messages be traced and managed?
4. How will performance bottlenecks be detected?

- The last item on this list poses the greatest challenge.
- The success of SOA within an enterprise is generally dependent on the extent to which it is standardized when it is phased into business and application domains.
- However, the success of a project delivering a service-oriented solution generally is measured by the extent to which the solution fulfills expected requirements within a given **budget and timeline.**

SOA delivery strategies

The lifecycle stages identified in the previous sections represent a simple, sequential path to building individual services.
We now need to organize these stages into a process that can:

- accommodate our preferences with regards to which types of service layers we want to deliver
- coordinate the delivery of application, business, and process services
- support a transition toward a standardized SOA while helping us fulfill immediate, project-specific requirements

To address this problem, we need a strategy. This strategy must be based on an organization's priorities to establish the correct balance between the delivery of long-term migration goals with the fulfillment of short-term requirements.

Three common strategies have emerged, each addressing this problem in a different manner.

- **top-down**
- **bottom-up**
- **agile (or meet-in-the-middle)**

The top-down strategy

This strategy is very much an "analysis first" approach that requires not only business processes to become service-oriented, but also promotes the creation (or realignment) of an organization's overall business model.

This process is therefore closely tied to or derived from an organization's existing business logic.

The top-down strategy supports the creation of all three of the service layers we discussed in the previous chapter. It is common for this approach to result in the creation of numerous reusable business and application services.

Process

The top-down approach will typically contain some or all of the steps illustrated and described in Figure 10.2.
Note that this process assumes that business requirements have already been collected and defined.

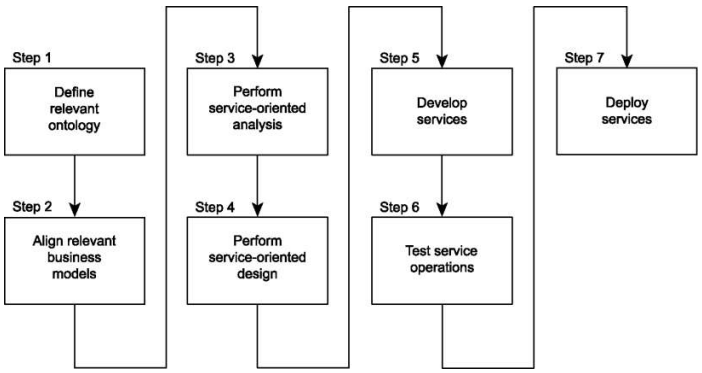


Figure 10.2: Common top-down strategy process steps.

Step 1: Define relevant enterprise-wide ontology

Part of what an ontology establishes is a classification of information sets processed by an organization. This results in a common vocabulary, as well as a definition of how these information sets relate to each other.

Larger organizations with multiple business areas can have several ontologies, each governing a specific division of business. It is expected that these specialized ontologies all align to support an enterprise-wide ontology.

Step 2: Align relevant business models (including entity models) with new or revised ontology

After the ontology is established, existing business models may need to be adjusted (or even created) to properly represent the vocabulary provided by the ontology in business modeling terms. Entity models in particular are of importance, as they can later be used as the basis for entity-centric business services.

Pros and cons

- The top-down approach to building SOA generally results in a **high quality** service architecture. The design and parameters around each service are thoroughly analyzed, maximizing reusability potential and opportunities for **streamlined compositions**.
- All of this lays the groundwork for a standardized and federated enterprise where services maintain a state of adaptability, while continuing to unify existing heterogeneity.
- The **obstacles** to following a top-down approach usually are associated with time and money. Organizations are required to invest significantly in up-front analysis projects that can take a great deal of time (proportional to the size of the organization and the immediate solution), without showing any immediate results.

The bottom-up strategy

- This approach essentially encourages the **creation of services** as a means of **fulfilling application-centric requirements.**
- Web services are built on an "as needed" basis and modeled to encapsulate application logic to best serve the immediate requirements of the solution.
- **Integration** is the primary motivator for **bottom-up designs**, where the need to take advantage of the open SOAP communications framework can be met by simply appending services as wrappers to legacy systems.

A typical bottom-up approach follows a process similar to the one explained in Figure 10.3. Note that this process assumes that business requirements have already been collected and defined.

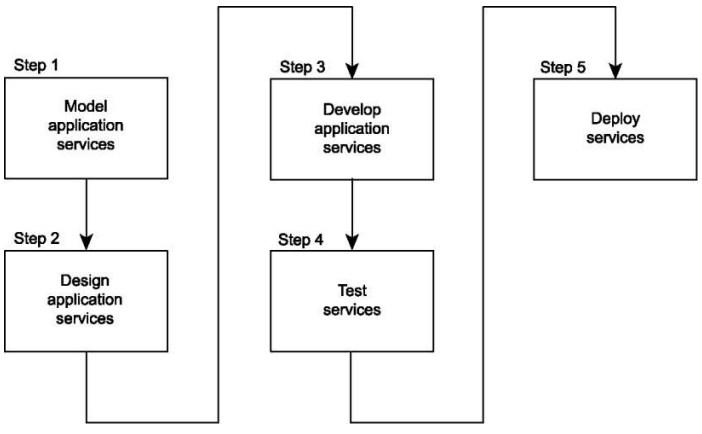


Figure 10.3: Common bottom-up strategy process steps.

Step 1: Model required application services

- This step results in the definition of application requirements that can be fulfilled through the use of Web services.
- Typical requirements include the need to establish point-to-point integration channels between legacy systems or B2B solutions.
- Other common requirements emerge out of the desire to replace traditional remote communication technology with the SOAP messaging communications framework.

- For solutions that employ the bottom-up strategy to deliver highly **service-centric solutions**, application services also will be modeled to include specific business logic and rules.
- In this case, it is likely that two application service layers will emerge, consisting of hybrid and utility services.
- Those services classified as reusable may act as generic application endpoints for integration purposes, or they may be composed by parent hybrid services.

Step 2: Design the required application services

Some of the application services modeled in Step 1 may be delivered by purchasing or leasing third-party wrapper services or perhaps through the creation of auto-generated proxy services.
These services may provide little opportunity for additional design. Custom application services, though, will need to undergo a design process wherein existing design standards are applied to ensure a level of consistency

Pros and cons

The majority of organizations that currently are building Web services apply the bottom-up approach.

The primary reason behind this is that organizations simply add Web services to their existing application environments to leverage the Web services technology set.

The architecture within which Web services are added remains unchanged, and service-orientation principles are therefore rarely considered .
As a result, the term that is used to refer to this approach "the bottom-up strategy "is somewhat of a misnomer.

The bottom-up strategy is really not a strategy at all. Nor is it a valid approach to achieving contemporary SOA. This is a realization that will hit many organizations as they begin to take service-orientation, as an architectural model, more seriously.

Section 10.4. The agile strategy

The combination of the Two: ***Top Down and Bottom up***

The challenge remains to find an acceptable balance between incorporating service-oriented design principles into business analysis environments without having to wait before integrating Web services technologies into technical environments.
For many organizations it is therefore useful to view these two approaches as extremes and to find a suitable middle ground.

This is possible by defining a new process that allows for the business-level analysis to occur concurrently with service design and development.

Also known as the meet-in-the-middle approach, the agile strategy is more complex than the previous two simply because it needs to fulfill two opposing sets of requirements.

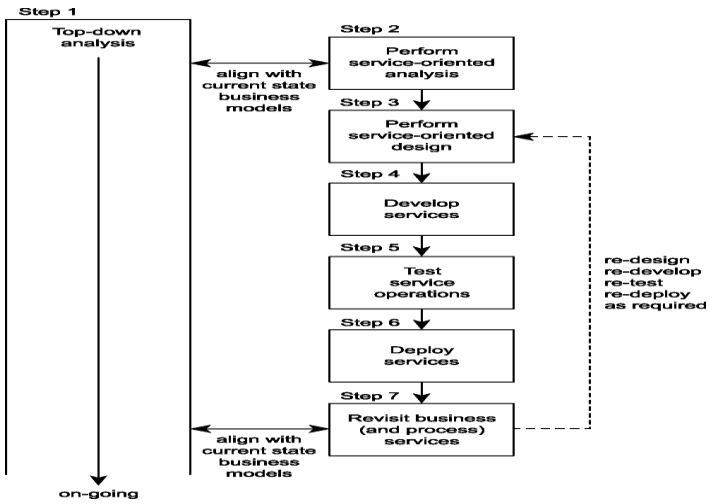


Figure 10.4: A sample agile strategy process.

Pros and cons

- This strategy takes the best of both worlds and combines it into an approach for realizing SOA that meets immediate requirements without jeopardizing the integrity of an organization's business model and the service-oriented qualities of the architecture.
- While it fulfills both short and long- term needs, the net result of employing this strategy is increased effort associated with the delivery of every service. The fact that services may need to be revisited, redesigned, redeveloped, and redeployed will add up proportionally to the amount of services subjected to this re-tasking step.

- Additionally, this approach imposes maintenance tasks that are required to ensure that existing services are actually kept in alignment with revised business models.
  Even with a maintenance process in place, services still run the risk of misalignment with a constantly changing business model.

Chapter 11. Service-Oriented Analysis (Part I: Introduction)

Introduction to service-oriented analysis

The process of determining how business automation requirements can be represented through service-orientation is the domain of the service-oriented analysis.

Objectives of service-oriented analysis

The primary questions addressed during this phase are:

- What **services** need to be built?
- What **logic** should be encapsulated by each service?

The overall goals of performing a service-oriented analysis are as follows :

- Define a preliminary set of service operation candidates.
- Group service operation candidates into logical contexts. These contexts represent service candidates.
- Define preliminary service boundaries so that they do not overlap with any existing or planned services.
- Identify encapsulated logic with reuse potential.
- Ensure that the context of encapsulated logic is appropriate for its intended use.
- Define any known preliminary composition models.

The service-oriented analysis process

- Introducing a new analysis process into an existing IT environment can be a tricky thing. Every organization has developed its own approach to analyzing business automation problems and solutions, and years of effort and documentation will have already been invested into well-established processes and modeling deliverables.
- Service-oriented analysis can be applied at different levels, depending on which of the SOA delivery strategies are used to produce services.
- From an analysis perspective, each layer has **different** modeling **requirements**. For example, the nature of the analysis required to define application services is different from what is needed to model the business service layer.

- Therefore, as previously mentioned, a key prerequisite of this process is the choice of SOA delivery strategy.
- Other questions that should be answered prior to proceeding with the service-oriented analysis include:
  - **What outstanding work is needed to establish the required business model(s) and ontology?**
  - **What modeling tools will be used to carry out the analysis?**
  - **Will the analysis be part of an SOA transition plan?**
- The service-oriented analysis process is a sub-process of the overall SOA delivery lifecycle. The steps shown in Figure 11.1 are common tasks associated with this phase and are described further in the following sections.

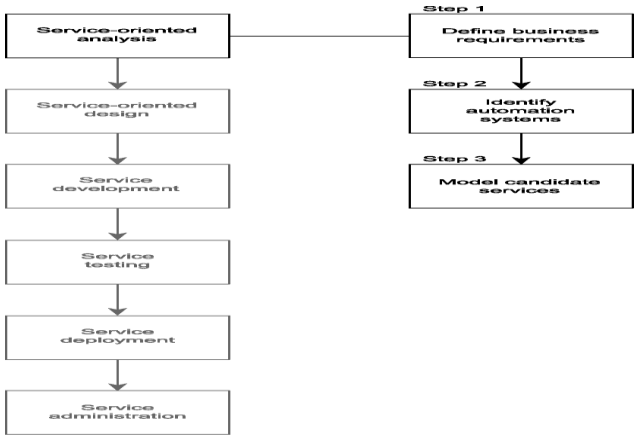


Figure 11.1:
A high-level service-oriented analysis process.

Step 1: Define business automation requirements

- Through whatever means business requirements are normally collected, their documentation is required for this analysis process to begin.
- Given that the scope of our analysis centers around the creation of services in support of a service oriented solution, only requirements related to the scope of that solution should be considered.
- Business requirements should be sufficiently mature so that a high-level automation process can be defined.

Step 2: Identify existing automation systems

- Existing application logic that is already, to whatever extent, automating any of the requirements identified in Step 1 needs to be identified.
- While a service-oriented analysis will not determine how exactly Web services will encapsulate or replace legacy application logic, it does assist us in scoping the potential systems affected.
- The details of how Web services relate to existing systems are ironed out in the service-oriented design phase.
- Note that this step is more geared to supporting the modeling efforts of larger scaled service-oriented solutions. An understanding of affected legacy environments is still useful when modeling a smaller amount of services, but a large amount of research effort would not be required in this case.

Step 3: Model candidate services

- A service-oriented analysis introduces the concept of service modeling a process by which service operation candidates are identified and then grouped into a logical context.
- These groups eventually take shape as service candidates that are then further assembled into a tentative composite model representing the combined logic of the planned service-oriented application
- Here is the familiar list of RailCo services:
  - Invoice Submission Service
  - Order Fulfillment Service
  - TLS Subscription Service

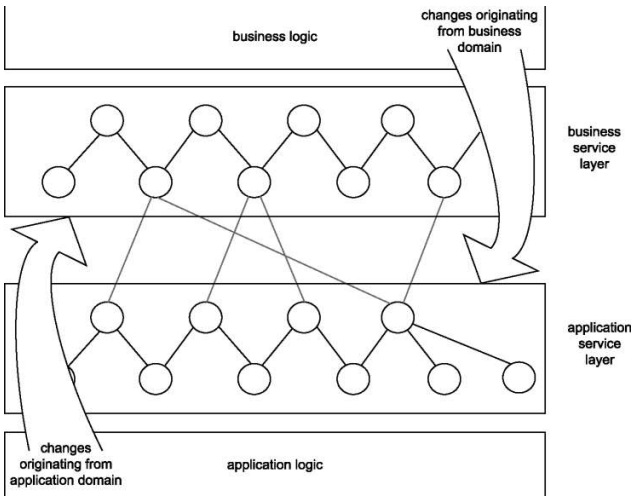


Figure 11.2: Changes originating in the business logic domain are accommodated by the application logic domain and vice versa.

11.2. Benefits of a business-centric SOA

Business services build agility into business models

•Service-orientation brings to business process models a structure that can significantly improve the flexibility and agility with which processes can be removed led in response to change.

•When properly designed, business service can establish a highly responsive information technology environment; responsive changes in that in an organization's business area can be efficiently accommodated through re-composition of both business process and its supporting technology architecture.

• In other words, applying service layer abstraction to both business and technology ends establishes the potential for an enterprise to achieve a form of two-way agility

Business services prepare a process for orchestration

- Whether or not you will be moving to an orchestration-based service-oriented environment anytime soon, it is becoming increasingly important to be ready for this transition.
- Orchestration brings with it concepts that, when implemented, lie at the core of SOA. Therefore, modeling current processes so that they eventually can be more easily migrated to an orchestration-driven environment is recommended.

Business services enable reuse

The creation of a business service layer promotes reuse in both business and application services, as follows :

1. By modeling business logic as distinct services with explicit boundaries, business process-level reuse can be achieved. Atomic sub-process logic or even entire processes can be reused as part of other process logic or as part of a compound process.

2. By taking the time to properly align business models with business service representation, the resulting business service layer ends up freeing the entire application service layer from assuming task or activity-specific processing functions. This allows application services to be positioned as and to evolve into pure, reusable utility services that facilitate business services across solution boundaries.

Only business services can realize the service-oriented enterprise

- Business service modeling marries the principles of service-orientation with an organization's business model. The resulting perspective can clearly express how services relate to and embody the fulfillment of business requirements.

- Applying business services forces an organization to view and reinterpret business knowledge in a service-oriented manner. Altering the perspective of how business processes can be structured, partitioned, and modeled is an essential step to achieving an environment in which service-orientation is standardized, successfully consistent, and naturally commonplace

11.3 Deriving business services

- As much as no industry-standard definition of SOA exists and as much as service orientation principles have not been globally standardized, there is also no standardized means of modeling business services.

- As with all other aspects of SOA, there are plenty of opinions, and though many have ideas, few concrete methodologies have emerged. Instead, there are a select set of approaches, some of which have been more accepted than others.

- Perhaps there should be no single approach to deriving services. It is not unusual for the business model behind a typical enterprise to have undergone thousands of revisions, shaped through years of adapting to the organization's surrounding business climate.

- The bottom line is that every business model is unique. Therefore, up-front analysis cannot be avoided to properly derive business services that best represent an organization as a cohesive business entity.

11.3.1. Sources from which business services can be derived

- The inner working of any organization, regardless of structure or size, can be decomposed into a collection of business services. This is because a business service simply represents a logic of works, and a pretty much anything any organization does consists of units of work.

- What differs, though, is how organizations structure and document the work they perform. At the beginning of this section we stressed the fact that every corporate environment is unique in the shape and size of its business models and in how it.

Business Process Management (BPM) model

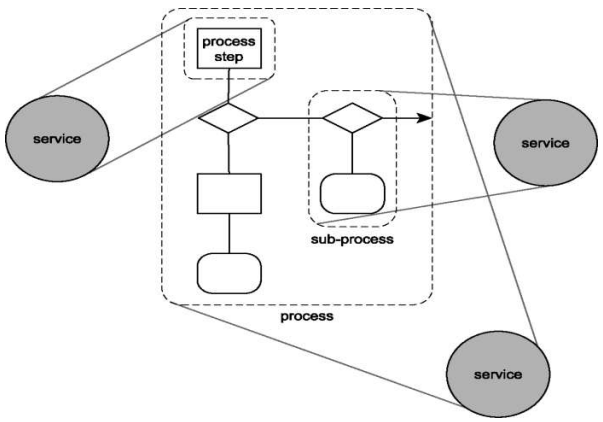


Figure 11.3: Parts of a process that can be encapsulated by a business service.

Types of derived business services

- **Task-centric business services**

These are Web services that have been modeled to accommodate a specific business process. Operations are grouped according to their relevance to the execution of a task in support of a process.

Although they are hybrid (application + business) in design, the following RailCo services follow a **task-centric** model:

- Invoice Submission Service
- Order Fulfillment Service
- TLS Subscription Service

- **Entity-centric business services**

Entity-centric business services generally are produced as part of a long-term or on-going analysis effort to align business services with existing corporate business models.

- Accounts Payable Service
- Purchase Order Service
- Ledger Service
- Vendor Profile Service

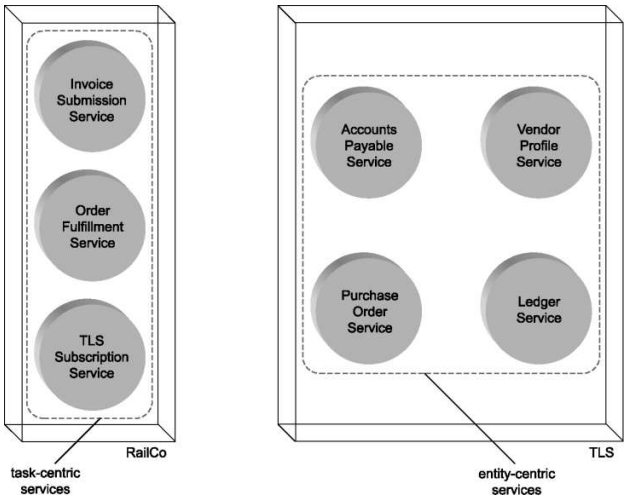


Figure 11.5: Different organizations, different approaches to building services.

- The process service is a form of business service
- It is business centric as it reside at the top of service layer hierarchy.
- It is responsible for composing business services.
- The core business model is represented by entity-centric services while business logic related task can be implemented in task centric services.

Chapter 12.

Service-Oriented Analysis

(Part II: Service Modeling)

12.1. Service modeling (a step-by-step process)

12.1.1. "Services" versus "Service Candidates

- Before we begin, let's first introduce an important modeling term : candidate . The primary goal of the service-oriented analysis stage is to figure out what it is we need to later design and build in subsequent project phases.
- It is therefore helpful to continually remind ourselves that we are not actually implementing a design at this stage. We are only performing an analysis that results in a proposed separation of logic used as input for consideration during the service-oriented design phase.
- In other words, we are producing abstract candidates that may or may not be realized as part of the eventual concrete design.
- So, at this analysis stage, we do not produce services; we create service candidates . Similarly, we do not define service operations; we propose service operation candidates .
- Finally, service candidates and service operation candidates are the end-result of a process called service modeling

12.1.2. Process description

Up next is a series of 12 steps that comprise a proposed service modeling process. Specifically, this particular process provides steps for the modeling of an SOA consisting of

- ***application,***
- ***business,***
- ***orchestration*** service layers .

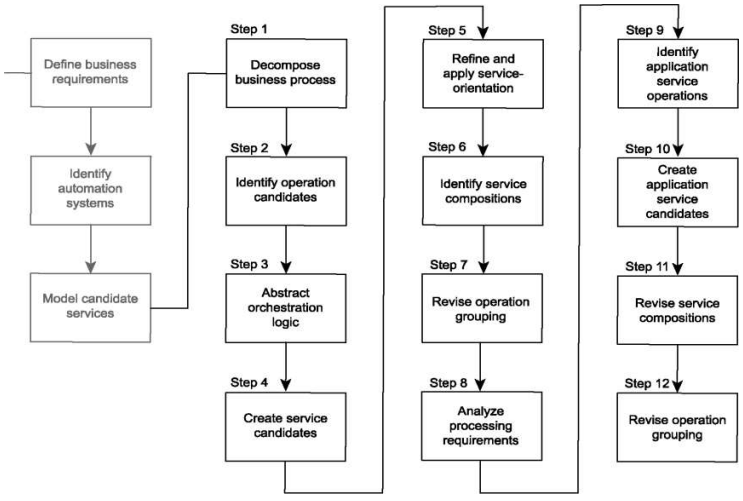


Figure 12.1: A sample service modeling process.

Step 1: Decompose the business process

- Take the documented business process and break it down into a series of granular process steps.
- It is important that a process's workflow logic be decomposed into the most granular representation of processing steps, which may differ from the level of granularity at which the process steps were originally documented.

Ex:
Decomposing the Invoice Submission Process

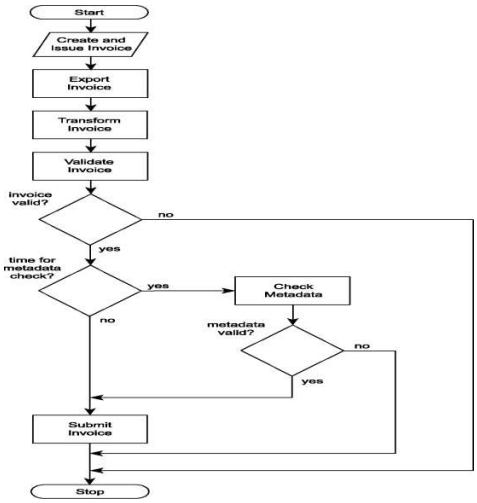


Figure 12.2: The RailCo Invoice Submission Process.

Decomposed Order Fulfillment Process

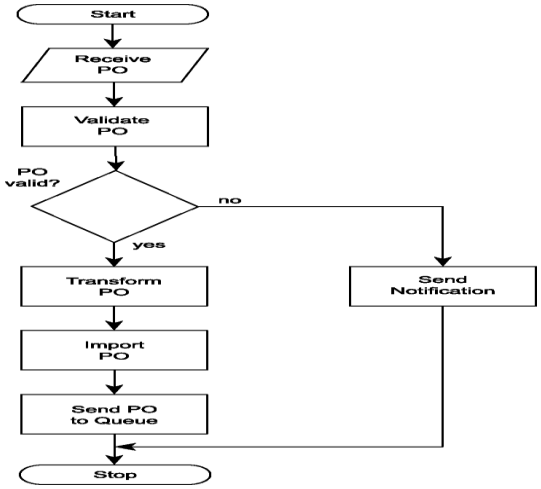


Figure 12.3: The RailCo Order Fulfillment Process.

Step 2: Identify business service operation candidates

- Some steps within a business process can be easily identified as not belonging to the potential logic that should be encapsulated by a service candidate.

Examples include:

- Manual process steps that cannot or should not be automated.
- Process steps performed by existing legacy logic for which service candidate encapsulation is not an option.
- By filtering out these parts we are left with the processing steps most relevant to our service modeling process.

Step 3: Abstract orchestration logic

- If you have decided to build an orchestration layer as part of your SOA, then you should identify the parts of the processing logic that this layer would potentially abstract. (If you are not incorporating an orchestration service layer, then skip this step.)
- Potential types of logic suitable for this layer include:

1. business rules
2. conditional logic
3. exception logic
4. sequence logic

Step 4: Create business service candidates

- Review the processing steps that remain and determine one or more logical contexts with which these steps can be grouped.
- Each ***context*** represents a service candidate. The contexts you end up with will depend on the types of business services you have chosen to build.
- It is important that you do not concern yourself with how many steps belong to each group. The primary purpose of this exercise is to establish the required set of contexts.
- Also it is encouraged that entity-centric business service candidates be equipped with additional operation candidates that facilitate future reuse.
- Therefore, the scope of this step can be expanded to include an analysis of additional service operation candidates not required by the current business process, but added to round out entity services with a complete set of reusable operations.

Going through the RailCo steps we listed from both processes, here is how they could be grouped:

Case Study

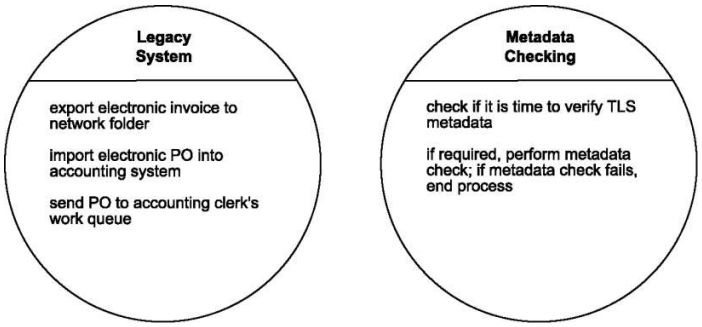


Figure 12.4.1: Our first set of agnostic service candidates.

Continue..

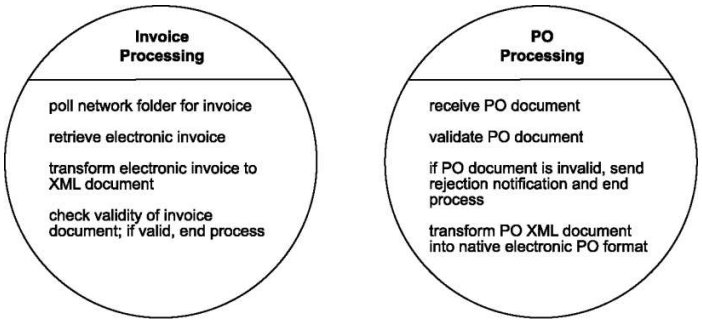


Figure 12.4.2: The first versions of two task-centric business service candidates.

Step 5: Refine and apply principles of service-orientation

- So far we have just grouped processing steps derived from an existing business process.
- To make our service candidates truly worthy of an SOA, we must take a closer look at the underlying logic of each proposed service operation candidate.

four key principles as those not intrinsically provided through the use of Web services:

- reusability
- autonomy
- statelessness
- discoverability

Figure 12.5: The revised set of RailCo service candidates.

Step 6: Identify candidate service compositions

- Identify a set of the most common scenarios that can take place within the boundaries of the business process.
- For each scenario, follow the required processing steps as they exist now.

This exercise accomplishes the following:

- It gives you a good idea as to how appropriate the grouping of your process steps is.
- It demonstrates the potential relationship between orchestration and business service layers.
- It identifies potential service compositions.
- It highlights any missing workflow logic or processing steps.

Case Study

Let's recap some of the service candidates established so far.

- Legacy System Service
- Polling Notification Service
- Transform Accounting Documents Service
- Metadata Checking Service

Each of these service candidates represents generic, reusable, and business- agnostic logic.

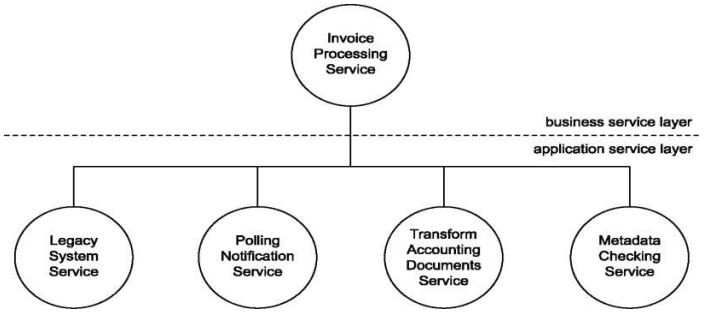


Figure 12.6: A sample composition representing the Invoice Submission Process.

Step 7: Revise business service operation grouping

- Based on the results of the composition exercise in Step 6, revisit the grouping of your business process steps and revise the organization of service operation candidates as necessary.
- It is not unusual to consolidate or create new groups (service candidates) at this point.

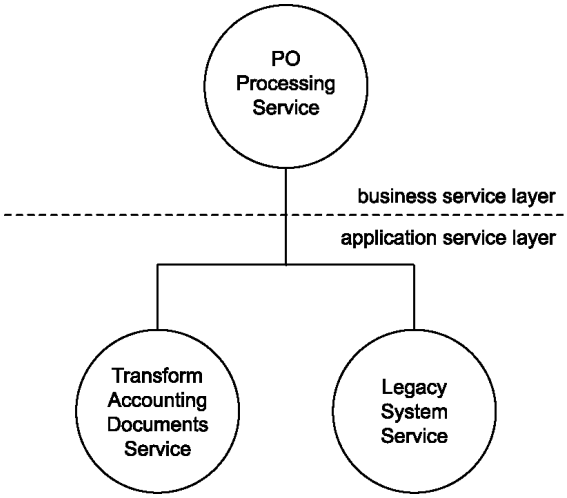


Figure 12.7: A sample composition representing the Order Fulfillment Process.

Step 8: Analyze application processing requirements

- By the end of Step 6, you will have created a business-centric view of your services layer. This view could very well consist of both application and business service candidates, but the focus so far has been on representing business process logic.
- This next series of steps is optional and more suited for complex business processes and larger service-oriented environments.

Step 9: Identify application service operation candidates

- Break down each application logic processing requirement into a series of steps.
- Be explicit about how you label these steps so that they reference the function they are performing.
- Ideally, you would not reference the business process step for which this function is being identified.

Step 10: Create application service candidates

Group these processing steps according to a predefined context. With application service candidates, the primary context is a logical relationship between operation candidates. This relationship can be based on any number of factors, including:

1. association with a specific legacy system
2. association with one or more solution components
3. logical grouping according to type of function

Step 11: Revise candidate service compositions

- Revisit the original scenarios you identified in Step 5 and run through them again. Only, this time, incorporate the new application service candidates as well.
- This will result in the mapping of elaborate activities that bring to life expanded service compositions. Be sure to keep track of how business service candidates map to underlying application service candidates during this exercise.

Step 12: Revise application service operation grouping

- Going through the motions of mapping the activity scenarios from Step 11 usually will result in changes to the grouping and definition of application service operation candidates.
- It will also likely point out any omissions in application-level processing steps, resulting in the addition of new service operation candidates and perhaps even new service candidates.

Section 12.2. Service modeling guidelines

- Provided here is a set of guidelines that supplement the previous modeling process with some additional considerations.
- These guidelines help ensure that our service candidates attain a balance of proper logic encapsulation and adherence to the service-orientation principles.

Take into account potential cross-process reusability of logic being encapsulated (task-centric business service candidates)

Identifying a real opportunity for reuse is an important consideration when grouping operation candidates into service candidates.

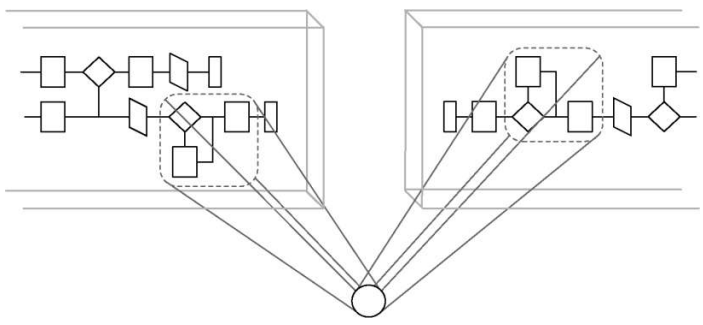


Figure 12.8: Service logic being reused across processes.

12.2.2. Consider potential intra-process reusability of logic being encapsulated (task-centric business service candidates)

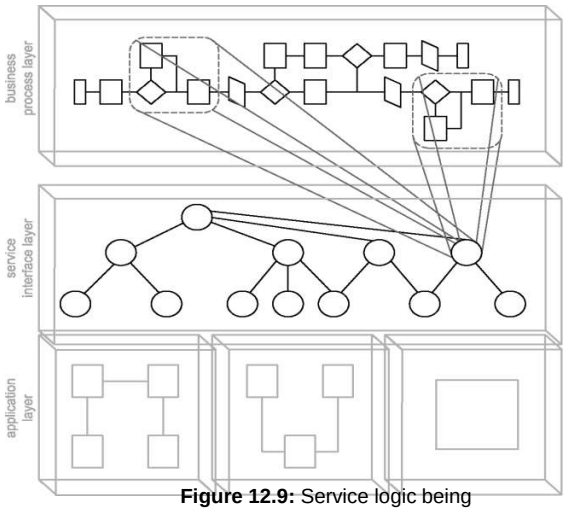


Figure 12.9: Service logic being reused within a process.

12.2.3. Factor in process- related dependencies (task-centric business service candidates)

- After identifying process steps that represent business logic we want to encapsulate, we have to check for any dependencies the current process have with legacy system.
- To check this the process is to be broken down to granular processing steps to check the dependency
- The dependency of service on external information determines its potential mobility and reusability.

12.2.4. Model for cross-application reuse (application service candidates)

- Each business process candidate designed to represent a specific part of business model. So, these are highly customized and carefully model.
- On the other hand, Application service candidates do not model to specific business requirement.
- They provide an business-agnostics design.
- So, when application service operation grouped with service candidate, they are completely neutral of service oriented solution.

12.2.5. Speculate on further decomposition requirements

- During this step, it is checked whether the service candidate can be further decomposed.
- If the service can be decomposed, it is decomposed up to the minimum level.

12.2.6. Identify logical units of work with explicit boundaries

- The Step checks that whether the services are autonomous.
- Logic encapsulated by service candidate have a boundary that informs the level of its independence from underlying business or application logic.

12.2.7. Prevent logic boundary creep

Boundary creep can happen in following circumstances

- The logic encapsulated by both services are same
- The logic they encapsulate overlaps
- Services are derived form same business process.

The steps to reduce the risk are as follows

- Check available metadata before creating new one
- Implement a set of standard from all model services
- Raise an awareness to those involved in Business process and service modeling.

12.2.8. Emulate Process services when not using orchestration (task centric business service candidate)

- Introduction of orchestration service layer increases the complexity of business process by providing one or more control services.
- In order to solve this issue a master control service needed that simulate process service and manage the orchestration composition model.

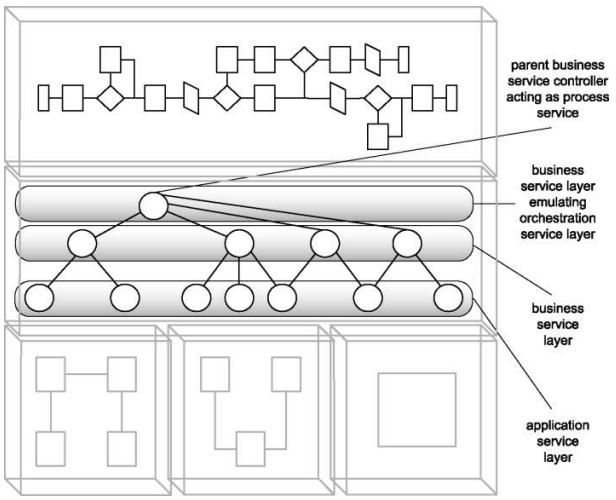


Figure 12.10: A parent business service layer acting as an orchestration layer.

12.2.9. Allocate appropriate modeling resources

- A service-oriented enterprise is further characterized by how the business end of an organization relates to the technology responsible for its automation.
- Service-orientation fully supports and enforces the vision of business-driven solutions, where automation technology is designed to be inherently adaptive so that it can respond to changes in the governing business logic.
- Limiting the application of service-orientation principles to technical IT staff can inhibit SOA's potential of realizing this vision.
- Technical architects and developers typically do not possess the depth of business knowledge required to model services with quality business logic representation. Therefore, business analysts often need to get involved in the service modeling process

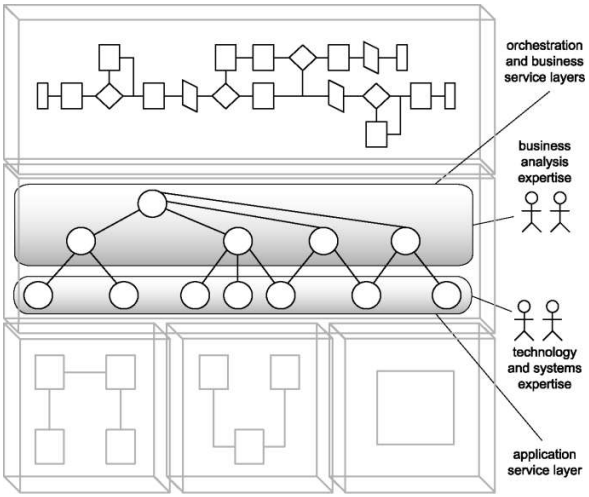


Figure 12.11: This intentionally simplistic diagram highlights the type of expertise recommended for modeling service layers.

12.2.10 Create and publish business service modeling standards

- The guidelines supplied by this section can only provide you with a direction on how to implement service modeling within your organization.
- Depending on the modeling tools and methodologies already in use, you will need to incorporate those that fit within your current business modeling environments.

Section 12.3. Classifying service model logic

- Service-oriented encapsulation allows a single operation to express a potentially broad range of logic.
- While modeling business logic, it is useful to understand the scope of logic represented by candidate operations, services or process.
- Building blocks also known as service modeling units are labels applied to units of business logic that helps in composition or decomposition of service oriented enterprise.

.

The SOE model

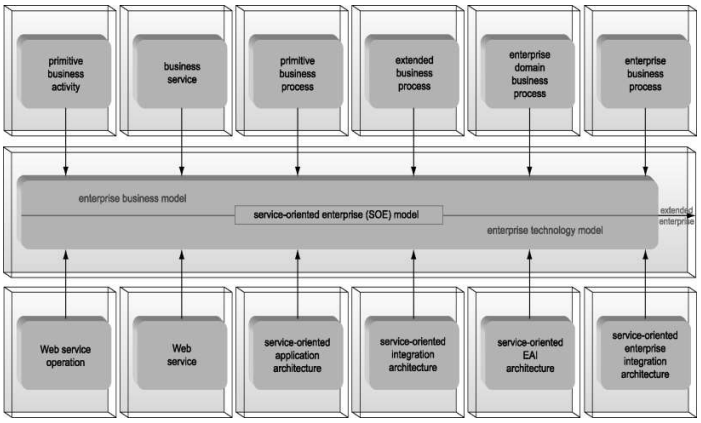


Figure 12.12: The SOE model.

The enterprise business model

- The building blocks in this first layer classify logic encapsulated by business service candidates only.
- They provide an abstract representation of a service-oriented enterprise's business intelligence, independent from the underlying technology platform with which it is implemented.
- If we ever want to replace our technology platforms, we will be able to do so while still preserving our abstract, service-oriented perspective of enterprise business logic.
- For our purposes, these building blocks help us label and categorize logic that resides in the orchestration and business service layers.

"Building Blocks" versus "Service Models"

- ***Building blocks*** are not an independent means of classifying service logic. They are complemented by existing classifications, such as the service models .
- ***Service models*** are useful for classifying the nature of logic encapsulated by a service, building blocks classify the scope of service logic.

Basic modeling building blocks

- Everything that exists within a modeled view of a service-oriented environment can be broken down into a collection of building blocks,
- Each of which falls into one of the following three categories:
  - Activity
  - Service
  - Process
- These terms establish an overall context. Hence, the names of our building blocks are derived from these three categories.
- The following represents the list of building blocks that form the enterprise business model part of the SOE model.
  - Primitive Business Activity
  - Primitive Business Service
  - Primitive Business Process
  - Extended Business Process
  - Enterprise Domain Business Process
  - Enterprise Business Process

Primitive business activities

- A primitive business activity represents the *smallest piece of definable and executable business logic* within a service-oriented environment.
- Typically this means that to whatever extent it makes sense to break down a business process, primitive business activities are the smallest parts.
- The assumption, therefore, is that its logic either cannot be decomposed or will not require further decomposition.
- The physical implementation of a primitive business activity can be compared to the functionality provided by a granular service operation.
- Coarse-grained operations tend to expose business logic that can be decomposed into numerous individual primitive business activities.
- Therefore, a granular operation exposing application logic that automates a single action of an overall business process is a suitable measure of implementation

Process activities

- Related to a primitive business activity is the process activity .
- A process activity is not a building block; it is simply a term used to represent an executable step within a business process's workflow logic.
- Unlike primitive business activities, process activities do not have a fixed scope.
- The range of business logic represented by a process activity is determined by the granularity of its governing business process.
- Therefore, a process activity may or may not be comprised of multiple primitive business activities.

Business services

- The business service (or business service candidate) category represents the familiar business service candidate.
- Within the context of this classification system, each business service is comprised of one or more primitive business activities.
- These activities can reside atomically within the service, or they can inter-relate.
- In the latter case, primitive business activities may form a logical algorithm that can establish independent workflow logic and associated business rules.

Primitive business services

- A primitive business service (or primitive business service candidate) is a type of business service that encompasses functionality limited to a simple business task or function.
- In other words, this variation of the business service building block represents the most granular type of service within service-oriented solutions.

Primitive business process

- A primitive business process represents a body of workflow logic comprised of a set of related process activities.
- A primitive business process is defined by a distinct functional boundary typically related to a specific business task (such as Submit Invoice or Process Purchase Order).
- A primitive business process can be represented by a process service or task-centric business service

THANK YOU

Part V: Building SOA (Technology and Design)

Chapter 13. Service-Oriented Design (Part I: Introduction)

13.1	Introduction to service-oriented design
13.2	WSDL- related XML Schema language basics
13.3	WSDL language basics
13.4	SOAP language basics
13.5	Service interface design tools

Introduction to service-oriented design

- Service-oriented design is the process by which concrete **physical service** designs are derived from ***logical service candidates***
- then assembled into abstract compositions that implement a business process.

Objectives of service-oriented design

The primary questions answered by this phase are:

1. How can physical service interface definitions be derived from the service candidates modeled during the service-oriented analysis phase?
2. What SOA characteristics do we want to realize and support?
3. What industry standards and extensions will be required by our SOA to implement the planned service designs and SOA characteristics?

The overall goals of performing a service-oriented design are as follows :

- Determine the core set of architectural extensions.
- Set the boundaries of the architecture.
- Identify required design standards.
- Define abstract service interface designs.
- Identify potential service compositions.
- Assess support for service-orientation principles.
- Explore support for characteristics of contemporary SOA.

"Design standards" versus "Industry standards"

- ***Design standards*** represent custom standards created by an organization to ensure that services and SOAs are built according to a set of consistent conventions.
- ***Industry standards*** are provided by standards organizations and are published in Web services and XML specifications

The service-oriented design process

- As with the service-oriented analysis, we first establish a parent process that begins with some preparatory work.
- This leads to a series of iterative processes that govern the creation of different types of service designs and, ultimately, the design of the overall solution workflow

Step 1: Compose SOA

A fundamental quality of SOA is that each instance of a service-oriented architecture is uniquely composable

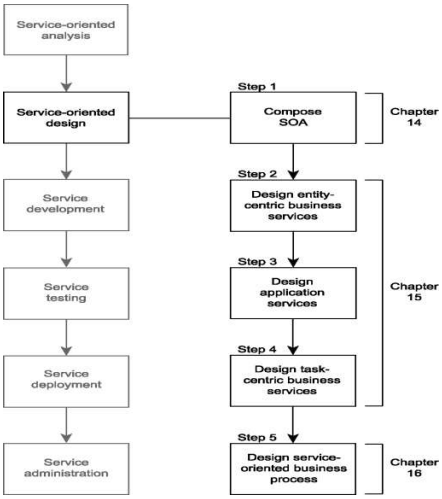


Figure 13.1: A high-level service-oriented design process.

Prerequisites

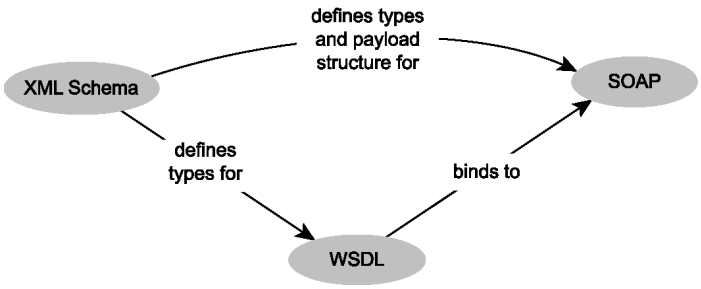


Figure 13.2: Three core specifications associated with service design.

Section 13.2. WSDL-related XML Schema language basics

- The XML Schema Definition Language (XSD) has become a central and very common part of XML and Web services architectures.
- The hierarchical structure of XML documents can be formally defined by creating an XSD schema hence an XML document is considered an instance of its corresponding schema.
- Further, the structure established within an XSD schema contains a series of rules and constraints to which XML document instances must comply for parsers and processors to deem them valid.

- The fundamental data representation rules provided by XSD schemas are related to representing data according to type.
- As with data types used in programming languages, XSD schemas provide a set of non-proprietary data types used to represent information in XML document instances.
- The data types supported by XSD schemas are extensive , but they do not always map cleanly to the proprietary types used by programming languages.

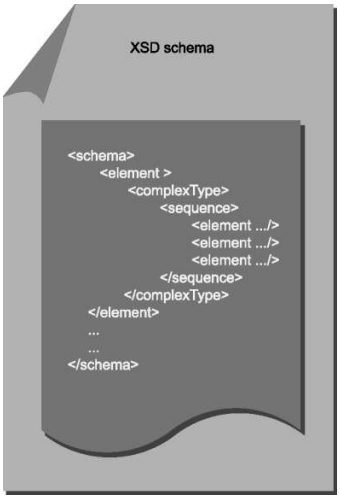


Figure 13.3: An XSD schema document.

- XSD schemas can exist as separate documents (typically identified by .xsd file extensions),
- They may be embedded in other types of documents, such as XML document instances or WSDL definitions.
- XML document instances commonly link to a separate XSD schema file so that the same schema can be used to validate multiple document instances.
- WSDL definitions can import the contents of an XSD file, or they also can have schema content embedded inline.
- Because almost all XML and Web services specifications are themselves written in XML, XSD schemas have become an intrinsic part of XML data representation and service-oriented architectures.
- Regardless of whether you explicitly define an XSD schema for your solution, your underlying processors will be using XSD schemas to execute many tasks related to the processing of XML documents through Web services.

"Elements" vs. "Constructs"

- A construct simply represents a key parent element likely to contain a set of related child elements.

The schema element

- The **schema** element is the **root** element of every XSD schema. It contains a series of common attributes used primarily to establish important namespace references.
- The xmlns attribute, for example, establishes a default namespace on its own, or can be used to declare a prefix that acts as an identifier to qualify other elements in the schema.
- The http://www.w3.org/2001/XMLSchema namespace always is present so that it can be used to represent content in the schema that originates from the XSD specification and the elements in the schema document itself.
- Ex. *<schema xmlns="http://www.w3.org/2001/XMLSchema">*

The ***element*** element

Using this element, you can declare a custom element that is then referenced by its name within an XML document instance.

For example, the following element declaration in an XSD schema:

Example 13.2. An element declaration in an XSD schema.
<element name="InvoiceNumber" type="xsd:integer"/>

Example 13.3. The usage of this element in an XML document instance.
<InvoiceNumber>12345</InvoiceNumber>

Section 13.3. WSDL language basics

The Web Services Description Language (WSDL) is the most fundamental technology standard associated with the design of services.

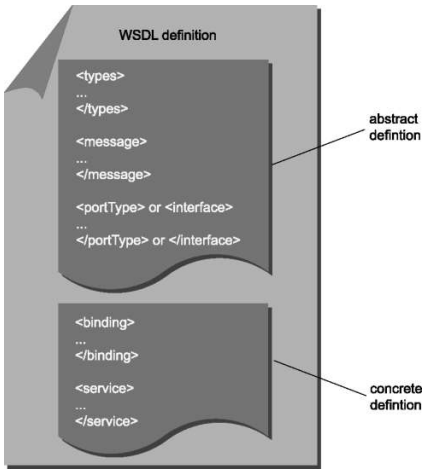


Figure 13.4: The structure of a WSDL definition.

The definition **element**

- This is the **root** or parent element of every WSDL document.
- It houses all other parts of the service definition
- and also the location in which the many namespaces used within WSDL documents are established.

Example 13.6. A definitions element of the Employee Service, declaring a number of namespaces.

< definitions name="Employee"
targetNamespace="http://www.xmltc.com/tls/employee/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:act="http://www.xmltc.com/tls/employee/schema/accounting/"
xmlns:hr="http://www.xmltc.com/tls/employee/schema/hr/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.xmltc.com/tls/employee/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"> ... </ definitions >

The types element

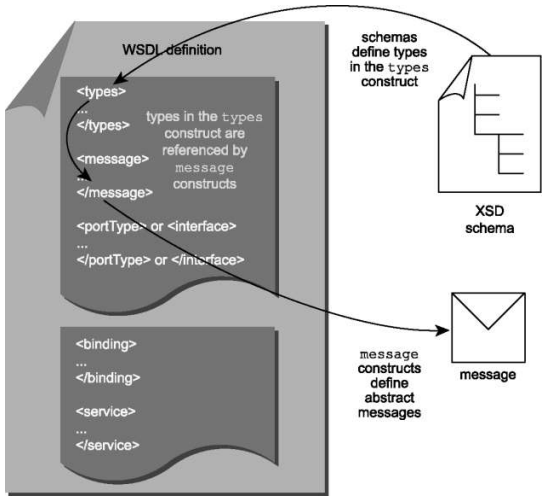


Figure 13.5: The WSDL types construct populated by XSD schema types used by the message construct to represent the SOAP message body.

Section 13.4. SOAP language basics

The Envelope element

- The Envelope element represents the root of SOAP message structures.
- It contains a mandatory Body construct and an optional Header construct.

Example 13.18. The root Envelope construct hosting Header and Body constructs.
< Envelope xmlns ="http://schemas.xmlsoap.org/soap/envelope/">
<Header> ... </Header>
<Body> ... </Body>
</ Envelope >

The Header element
< Header > <x:CorrelationID xmlns:x="http://www.xmltc.com/tls/headersample/"
mustUnderstand="1">
0131858580-JDJ903KD
</x:CorrelationID>
</ Header >

The Body element

SOAP message Body constructs are defined within the WSDL message constructs which, as we've already established, reference XSD schema data type information from the WSDL types construct

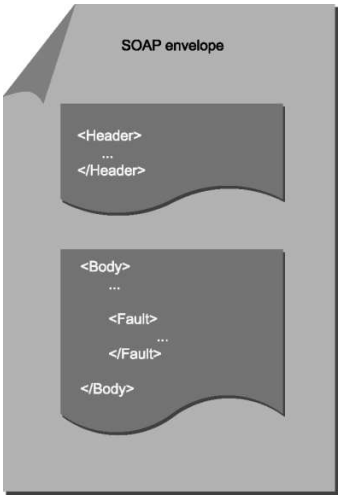


Figure 13.6: The structure of a SOAP message document.

<Body>
<soa:book xmlns:soa="http://www.serviceoriented.ws/">
<soa:ISBN>0131858580</soa:ISBN>
<soa:title> Service-Oriented Architecture
Concepts, Technology, and Design </soa:title>
</soa:book>
</Body>

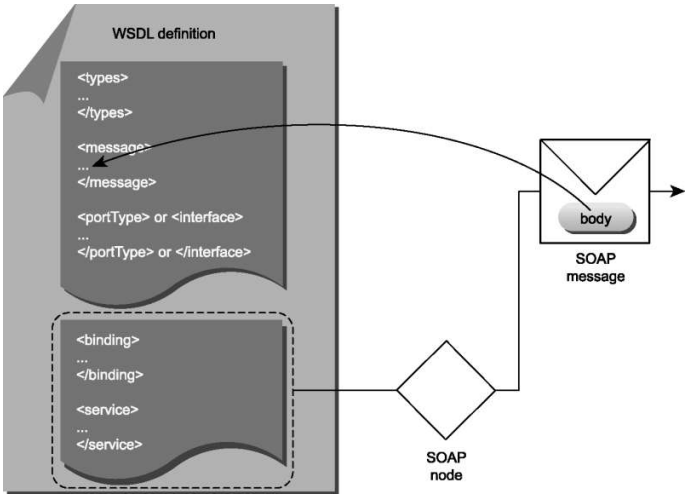


Figure 13.7: A SOAP message body defined within the WSDL message construct. The actual processing of the SOAP message via a wire protocol is governed by the constructs within the concrete definition.

The Fault Element

- The optional Fault construct provides a ready made error response that is added inside the Body construct.
- In the example that follows, this fault information is further sub-divided using additional child elements.
- The fault code element contains one of a set of fault conditions predefined by the SOAP specification. Both the
- Fault string and detail elements provide human readable error messages, the latter of which can host an entire XML fragment containing further partitioned error details.

```
<Body>
   <Fault>
      <faultcode>
         MustUnderstand
      </faultcode>
      <faultstring>
         header was not recognized
      </faultstring>
      <detail>
         <x:appMessage
            xmlns:x="http://www.xmltc.com/tls/faults">
               The CorrelationID header was not
               processed by a recipient that was
               required to process it. Now a fault's
               been raised and it looks like this
               recipient is going to be a problem.
         </x:appMessage>
      </detail>
   </Fault>
</Body>
```

Chapter 14. Service-Oriented Design (Part II: SOA Composition Guidelines)

Steps to composing SOA

- Regardless of the shape or size of your SOA, it will consist of a number of technology components that establish an environment in which your services will reside.
- The fundamental components that typically comprise an SOA include:
  - an XML data representation architecture
  - Web services built upon industry standards
  - a platform capable of hosting and processing XML data and Web services

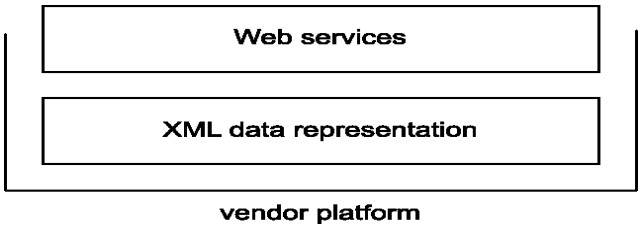


Figure 14.1: The most fundamental components of a Web services-based SOA.

Steps to composing SOA

- However, to support and realize the principles and characteristics we've explored as being associated with both the primitive and contemporary types of SOA requires some additional design effort.
- *Common questions that need to be answered at this stage include:*
  - What types of services should be built, and how should they be organized into service layers ?
  - How should first-generation standards be positioned to best support SOA?
  - What features provided by available extensions are required by the SOA?
- These issues lead to an exercise in composition, as we make choices that determine what technologies and architectural components are required and how these parts are best assembled.

Suggested steps for composing a preliminary SOA

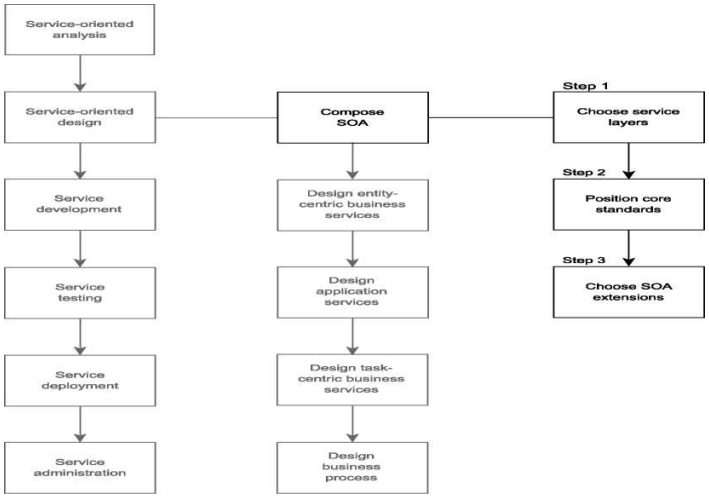


Figure 14.2: Suggested steps for composing a preliminary SOA.

Step 1: Choose service layers

- Composing an SOA requires that we first decide on a design configuration for the service layers that will comprise and standardize logic representation within our architecture.
- This step is completed by **studying the candidate service layers produced during the service-oriented analysis phase and exploring service layers and service layer configuration scenarios**.

Step 2: Position core standards

- Next, we need to assess which core standards should comprise our SOA and how they should be implemented to best support the features and requirements of our service-oriented solution.
- The Considerations for positioning core SOA standards section provides an overview of how each of the core XML and Web services specifications commonly is affected by principles and characteristics unique to SOA.

Step 3: Choose SOA extensions

- This final part of our "pre-service design process" requires that we determine which contemporary SOA characteristics we want our service-oriented architecture to support.
- This will help us decide which of the available WS-* specifications should become part of our service-oriented environment. The Considerations for choosing SOA extensions section provides some guidelines for making these determinations.

Section 14.2. Considerations for choosing service layers

- The service-oriented analysis process likely will have resulted in a preliminary identification of a suitable service layer configuration.
- The first step to designing SOA is deciding how you intend to configure service layers within your environment, if at all.
- Depending on the scope of your planned architecture, this step may require an analysis process that is highly organization-specific.
- Immediate and long- term goals need to be taken into account because when you choose a configuration, you essentially are establishing a standard means of logic and data representation.

High-level guidelines

- **Existing configurations** If service layers already have been standardized within your enterprise, you should make every attempt to conform new service designs to these layers.
- The exception to this is if a need to alter the current service layer configuration has been identified. Then the scope of your project will include a change to the overall complexion of your enterprise's standard SOA.
- **Required standards** If you are building new types of services or service layers, ensure that these are delivered along with accompanying design standards.
- **Service composition** performance Service compositions can impose a significant amount of processing overhead, especially when intermediary services are required to process the contents of SOAP messages.
- **Service deployment** When designing service layers that will produce solution- agnostic services, deployment can become a concern.
- **Business services and XSD schema designs:** If out enterprise already established a comprehensive XML architecture, it is worth taking look at existing set of XSD schema. But here there might be a compatibility issue comes into picture.
- **Business Service maintenance:** If we proceeds with agile delivery system, the on going maintenance of business service needs to be planned.

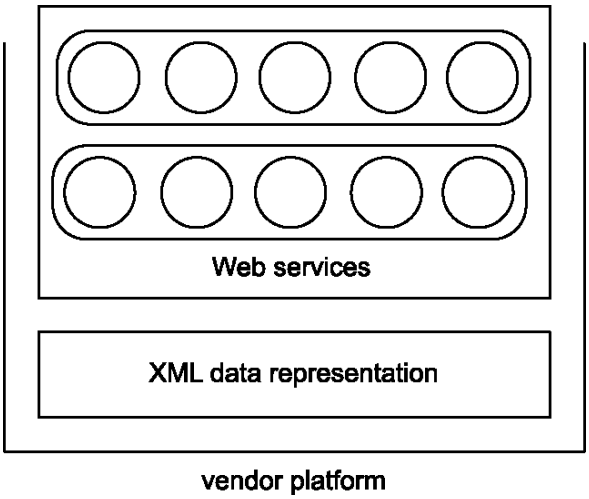


Figure 14.3: Designated service layers organize and standardize Web services within SOA.

Section 14.3. Considerations for positioning core SOA standards

- However, this step is not limited to simply picking and choosing what we want. We are required to properly position these standards so that they end up supporting the key characteristics we need our SOA to deliver.
- Therefore, this section consists of a series of discussions about how the utilization of core XML and Web services standards is commonly affected when utilized within the design parameters of a service-oriented architecture.

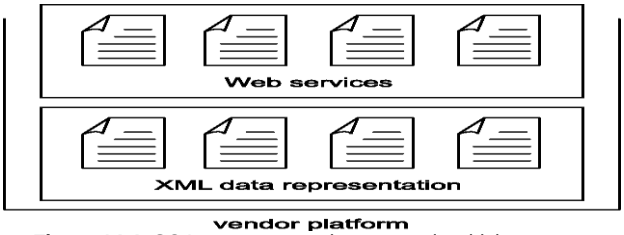


Figure 14.4: SOA can structure the manner in which common XML and Web services standards are applied.

Industry standards and SOA

- Even though we are creating abstract service designs, they still are realized in a physical implementation through the use of specific Web services markup languages.
- These languages originate from published specifications of which different versions in different stages of maturity exist.
- New versions of a specification can alter and extend the feature set provided by previous versions.
- It therefore is important to ensure that your SOA is fully standardized with respect to the specification versions that establish a fundamental layer of your technology architecture.
- This not only ensures standardization within an organization, it also expresses consistent metadata to any services with which external partners may need to interface. This, of course, promotes interoperability.

core specifications that commonly are found in SOA.

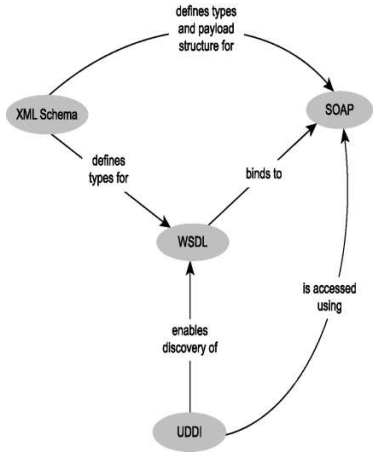


Figure 14.5: The operational relationship between core SOA specifications.

XML and SOA

- Fundamental to everything that comprises a contemporary SOA is data representation via XML.
- Practically all of the specifications within the Web services platform are expressed through XML and natively rely on the support and management of XML documents to enable all forms of communication.

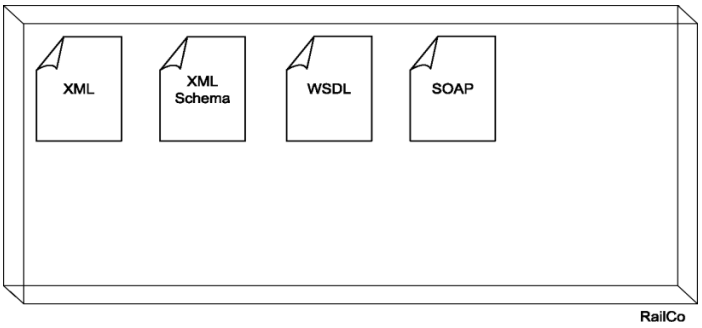


Figure 14.6: A composed view of the core standards that comprise part of RailCo's planned SOA.

RPC style vs. document style SOAP message

- To accommodate RPC communication, traditional data representation architectures tend to shape XML documents around a parameter data exchange model.
- This results in a fine-grained communications framework within distributed environments that inevitably leads to the use of RPC-style SOAP messages.

Auto-generated XML

- Many development tools offer the ability to provide auto-generated XML output.
- XML may be derived in numerous ways resulting in an instant XML data representation of a data model or data source.
- While useful to fulfill immediate conversion or data sharing requirements, the persistent use of auto-generated XML can lead to the proliferation of non-standardized data representation.

Fitting SOA on the top of an established XML data representation architecture

- Even if the standards are not in complete alignment with how your services will need to express corporate data, it is important that the existing data representation be consistent and predictable.
- After standardization has been achieved, service abstraction can be used to bridge representation disparity.
- The most effective way to coordinate this type of migration is to create a transition plan.
- Such a plan would include transition phases that specifically address XML compatibility issues.

WS – I Basic profile

Listed here are some examples of the design standards provided by the Basic Profile:

- SOAP envelopes cannot contain Document Type Declarations or processing instructions.
- The use of the SOAP encodingStyle attribute within SOAP envelopes is highly discouraged.
- Required SOAP header blocks must be checked prior to proceeding with the processing of other header blocks and the message contents.
- When a WSDL part construct (within the message construct) uses the element attribute, it must reference a global element declaration.
- The sequence of elements within the SOAP Body construct must be identical to the sequence established in the corresponding WSDL parts construct.
- The WSDL binding element can only use the WSDL SOAP binding.

WSDL and SOA

- The creation of WSDL definitions is an important part of the SOA delivery lifecycle.
- The service interface is the focal point of the service-oriented design phase and the primary deliverable of each of the service design processes.
- WSDL definitions are the end result of business and technology analysis efforts and establish themselves as the public endpoints that form a new layer of infrastructure within a service oriented enterprise.
- Some of the key design issues that relate to WSDL design within SOA are as follows
  - **Standardized use of namespaces :** Namespace standardization always be a primary concern.
  - **Modular service definitions :** modularize your WSDL documents to facilitate reuse and centralized maintenance
  - **Compatibility of granularity :**

XML schema and SOA

- XML Schema definitions (or XSD schemas) establish data integrity throughout service-oriented architectures.
- They are used intrinsically by many WS-* specifications but are most prominent in their role as defining a service's public data model.
- Following are some considerations as to how XSD schemas can be positioned and utilized in support of SOA.
- **Modular XSD schemas :**
  - XSD schemas can be broken down into individual modules that are assembled at runtime using the include statement (for schemas with the same target namespace) or the import statement (for schemas with different target namespaces).
  - Because WSDL documents also can be modularized, the XSD schema contents of the WSDL types construct can reside in a separate document.
- **Document-style messages and XSD schemas**
  - The use of extensible or redefined schemas may, therefore, be required when building documents that represent multiple data contexts.
  - However, even the advanced features of the XML Schema Definition Language may be insufficient.
  - Supplementary technologies (XSLT, for example) may be required to implement extensions, such as conditional validation.

SOAP and SOA

- SOAP messages are what fuel all action within contemporary SOA.
- They are therefore considered just as fundamental to service-oriented environments as WSDL definitions.

Following are two primary areas in which SOAP messaging can be affected.

SOAP message style and data types

- Probably the biggest impact SOA can have on an existing SOAP framework is in how it imposes a preferred payload structure and data type system.
- The use of schema modules may be required to accommodate the assembly of unique SOAP message payloads from differently structured XML data sources.
- In the end, though, standardization is key. Consistent XML document structures will accommodate the runtime assembly of document-style SOAP payloads.

SOAP Header

- it only provides a partial description of the SOAP messages required for services to communicate within contemporary SOA.
- While the XSD types used by WSDL definitions define and validate the contents of a SOAP message's Body construct,
- Understanding the extent of metadata abstraction allows us to adjust service operations accordingly.

Namespaces and SOA

- They must be viewed as identifiers of business or technology domains and accordingly applied as qualifiers for corresponding WSDL elements.
- The WS-I Basic Profile provides a set of best practices for implementing namespaces within WSDL definitions. However, additional design standards are required to ensure that namespaces are used properly in XML documents outside of WSDL definition boundaries..

UDDI and SOA

- UDDI provides an industry standard means of organizing service description pointers to accommodate the process of discovery through service registries.
- When implemented, UDDI typically represents an enterprise-wide architectural component positioned to provide a central discovery mechanism within and across SOAs.

Chapter 15. Service-Oriented Design (Part III: Service Design)

Before we can develop a service, we need to have defined the interface of that service.

This is the mantra of the commonly accepted "WSDL first" approach to designing Web services. And we will follow it in SOA service development.

Defining the service interface prior to development is important to establishing a highly standardized service-oriented architecture and required to realize a number of the characteristics we identified as being part of contemporary SOA.

Service design overview

The ultimate goal of these processes is to achieve a balanced service design. Typically this constitutes a Web service that accommodates real-world requirements and constraints, while still managing to:

1. encapsulate the required amount of logic
2. conform to service-orientation principles
3. meet business requirements

This sequence is actually more of a guideline, as in reality, the service design process is not always that clear cut.

For example, after creating an initial set of application service designs, you proceed to build task-centric services. Only while incorporating various operations, you realize that additional application service-level features are required to carry them out

Section 15.2. Entity-centric business service design (a step-by-step process)

Entity-centric business services represent the one service layer that is the least influenced by others.

Its purpose is to accurately represent corresponding data entities defined within an organization's business models.

These services are strictly solution- and business process-agnostic , built for reuse by any application that needs to access or manage information associated with a particular entity.

Because they exist rather atomically in relation to other service layers , it is beneficial to design entity-centric business services prior to others. This establishes an abstract service layer around which process and underlying application logic can be positioned.

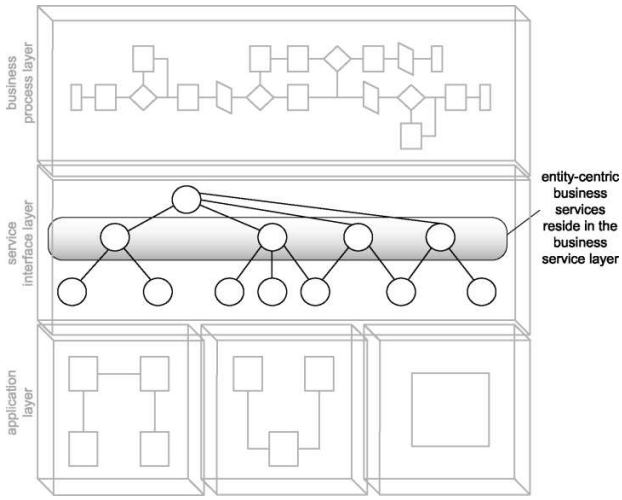


Figure 15.1: Entity-centric services establish the business service layer.

Process description

Provided next is the step-by-step process description wherein we establish a recommended sequence of detailed steps for arriving at a quality entity-centric business service interface.

Note that the order in which these steps are provided is not set in stone.
For example, you may prefer to define a preliminary service interface prior to establishing the actual data types used to represent message body content.

Or perhaps you may find it more effective to perform a speculative analysis to identify possible extensions to the service before creating the first cut of the interface.

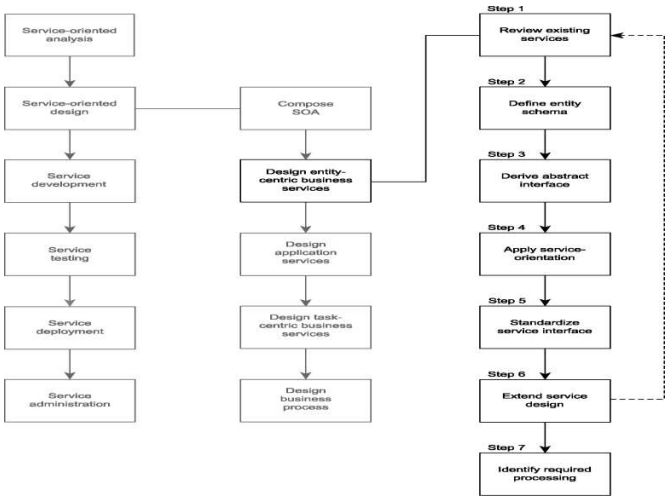


Figure 15.2: The entity-centric business service design process.

Case Study

The examples provided alongside this process description revisit the TLS environment.

The Employee Service was modeled intentionally to facilitate an entity-centric grouping of operations.

As part of the Timesheet Submission Process, this service is required to contribute two specific functions……>

Figure 15.3: The Employee Service candidate.

Step 1: Review existing services

Ideally, when creating entity-centric services, the modeling effort resulting in the service candidates will have taken any existing services into account.
However, because service candidates tend to consist of operation candidates relevant to the business requirements that formed the basis of the service-oriented analysis, it is always worth verifying to ensure that some or all of the processing functionality represented by operation candidates does not already exist in other services.

Therefore, the first step in designing a new service is to confirm whether it is actually even necessary. If other services exist, they may already be providing some or all of the functionality identified in the operation candidates or they may have already established a suitable context in which these new operation candidates can be implemented

Case Study

The investigation reveals that the following entity-centric business services were delivered as part of the B2B project:

1. Accounts Payable Service,
2. Purchase Order Service,
3. Ledger Service, and
4. Vendor Profile Service

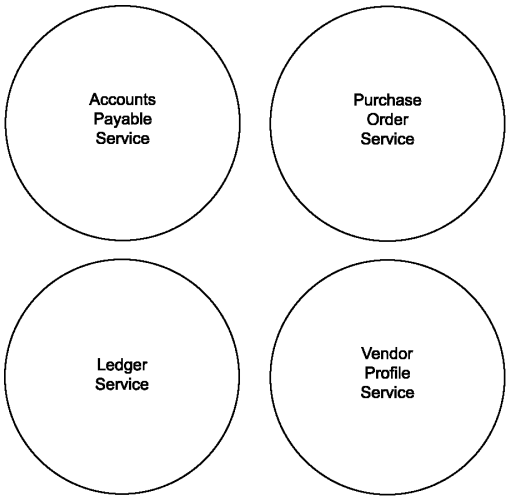


Figure 15.4: The existing inventory of TLS services.

Step 2: Define the message schema types

It is useful to begin a service interface design with a formal definition of the messages the service is required to process.
To accomplish this we need to formalize the message structures that are defined within the WSDL types area.

SOAP messages carry payload data within the Body section of the SOAP envelope. This data needs to be organized and typed. For this we rely on XSD schemas.

A standalone schema actually can be embedded in the types construct, wherein we can define each of the elements used to represent data within the SOAP body.

In the case of an entity-centric service, it is especially beneficial if the XSD schema used accurately represents the information associated with this service's entity.

This "entity-centric schema" can become the basis for the service WSDL definition, as most service operations will be expected to receive or transmit the documents defined by this schema.

Note that there is not necessarily a one-to-one relationship between entity-centric services and the entities that comprise an entity model.

Case Study
TLS invested in creating a standardized XML data representation architecture (for their accounting environment only) some time ago.

As a result, an inventory of entity-centric XSD schemas representing accounting- related information sets already exists.

However, upon closer study, it is discovered that the existing XSD schema is very large and complex. After some discussion, TLS architects decide for better or for worse that they will not use the existing schema with this service at this point.
Instead, they opt to derive a lightweight (but still fully compliant) version of the schema to accommodate the simple processing requirements of the Employee Service.

Continue..

They begin by identifying the kinds of data that will need to be exchanged to fulfill the processing requirements of the "Get weekly hours limit" operation candidate.

They end up defining two complex types: one containing the search criteria required for the request message received by the Employee Service and another containing the query results returned by the service.

The types are deliberately named so that they are associated with the respective messages. These two types then constitute the new Employee.xsd schema file.

Continue..

The Employee schema providing complexType constructs used to establish the data representation anticipated for the "Get weekly hours limit" operation candidate.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace=
"http://www.xmltc.com/tls/employee/schema/accounting/"> <xsd:element
name="  EmployeeHoursRequestType  "> <xsd:complexType>
<xsd:sequence>
 <xsd:element name="ID" type="xsd:integer"/> </xsd:sequence>
</xsd:complexType>
 </xsd:element> <xsd:element name="  EmployeeHoursResponseType  ">
<xsd:complexType>
<xsd:sequence>
 <xsd:element name="ID" type="xsd:integer"/> <xsd:element
name="WeeklyHoursLimit" type="xsd:short"/>
 </xsd:sequence>
 </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

However, just as the architects attempt to derive the types required for the "Update employee history" operation candidate, another problem presents itself.

They discover that the schema from which they derived the Employee.xsd file does not represent the *EmployeeHistory* entity, which this service candidate also encapsulates.

Another visit to the accounting schema archive reveals that employee history information is not governed by the accounting solution. It is, instead, part of the HR environment for which no schemas have been created.

Not wanting to impose on the already standardized design of the Employee schema, it is decided that a second schema definition be created. Analysts get involved and produce the following EmployeeHistory.xsd schema: fig-15.5

The EmployeeHistory schema, with a differenttargetNamespace to identify its distinct origin.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace=

"http://www.xmltc.com/tls/employee/schema/hr/">

 <xsd:element name="EmployeeUpdateHistoryRequestType">

<xsd:complexType> <xsd:sequence>
<xsd:element name="ID" type="xsd:integer"/>
 <xsd:element name="Comment" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
 <xsd:element name="EmployeeUpdateHistoryResponseType">
<xsd:complexType>
 <xsd:sequence>
<xsd:element name="ResponseCode" type="xsd:byte"/>
 </xsd:sequence>
</xsd:complexType>
 </xsd:element>
</xsd:schema>
```

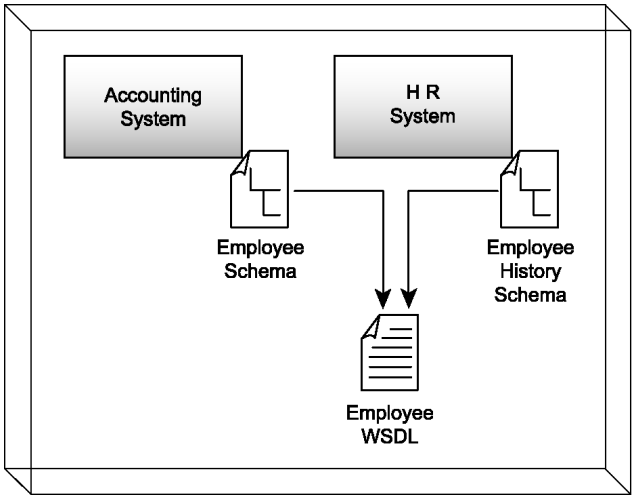


Figure 15.5: Two schemas originating from two different data sources.

To promote reusability and to allow for each schema file to be maintained separately from the WSDL, the XSD schema import statement is used to pull the contents of both schemas into the Employee Service WSDL types construct.

```
<types> <xsd:schema targetNamespace=
"http://www.xmltc.com/tls/employee/schema/">
 <xsd: import namespace=
"http://www.xmltc.com/tls/employee/schema/accounting/"
schemaLocation="Employee.xsd"/>
<xsd: import namespace=
"http://www.xmltc.com/tls/employee/schema/hr/"
schemaLocation="EmployeeHistory.xsd"/>
 </xsd:schema>
 </types>
```

Step 3: Derive an abstract service interface

Next, we analyze the proposed service operation candidate and follow these steps to define an initial service interface:

1.Confirm that each operation candidate is suitably generic and reusable by ensuring that the granularity of the logic encapsulated is appropriate. Study the data structures defined in Step 2 and establish a set of operation names .

2.Create the portType (or interface) area within the WSDL document and populate it with operation constructs that correspond to operation candidates.

3.Formalize the list of input and output values required to accommodate the processing of each operation's logic. This is accomplished by defining the appropriate messageconstructs that reference the XSD schema types within the child part elements.

Case Study

The TLS architects decide on the following operations names: GetEmployeeWeeklyHoursLimit and UpdateEmployeeHistory.

They subsequently proceed to define the remaining parts of the abstract definition, namely the ***message*** , and ***portType*** constructs.

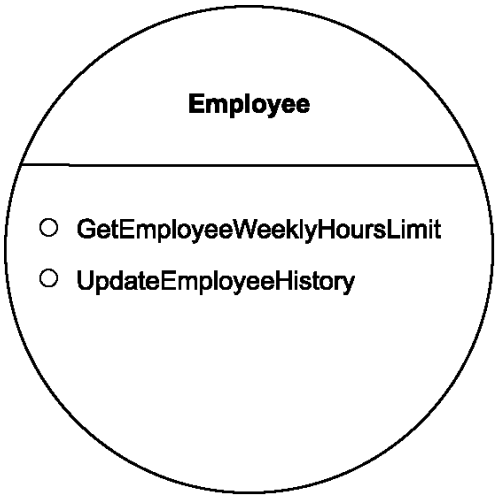


Figure 15.6: The Employee Service operations.

The message and portType parts of the Employee Service definition that implement the abstract definition details of the two service operations.

```
<message name="getEmployeeWeeklyHoursRequestMessage">
<part name="RequestParameter" element="act:EmployeeHoursRequestType"/>
</message> <message name="getEmployeeWeeklyHoursResponseMessage">
<part name="ResponseParameter"
element="act:EmployeeHoursResponseType"/> </message> <message
name="updateEmployeeHistoryRequestMessage"> <part
name="RequestParameter"
element="hr:EmployeeUpdateHistoryRequestType"/> </message>
 <message name="updateEmployeeHistoryResponseMessage">
<part name="ResponseParameter"
element="hr:EmployeeUpdateHistoryResponseType"/>
 </message> <portType name="EmployeeInterface">
<operation name="  GetEmployeeWeeklyHoursLimit  ">
 <input message= "tns:getEmployeeWeeklyHoursRequestMessage"/> <output
message= "tns:getEmployeeWeeklyHoursResponseMessage"/> </operation>
 <operation name="  UpdateEmployeeHistory  ">
 <input message= "tns:updateEmployeeHistoryRequestMessage"/> <output
message= "tns:updateEmployeeHistoryResponseMessage"/> </operation>
</portType>
```

Step 4: Apply principles of service-orientation

Here's where we revisit the four service-orientation principles.

- service reusability
- service autonomy
- service statelessness
- service discoverability

Reusability and autonomy, the two principles we already covered in the service modeling process, are somewhat naturally part of the entity-centric design model in that the operations exposed by entity-centric business services are intended to be inherently generic and reusable

Because entity-centric services often need to be composed by a parent service layer and because they rely on the application service layer to carry out their business logic, their immediate autonomy is generally well defined.
Unless those services governed by an entity-centric controller have unusual processing requirements or impose dependencies in some manner, entity-centric services generally maintain their autonomy.

Case Study

Upon a review of the initial abstract service interface, it is determined that a minor revision can be incorporated to better support fundamental service-orientation.
Specifically, meta information is added to the WSDL definition to better describe the purpose and function of each of the two operations and their associated messages.
The service interface, supplemented with additional metadata documentation.

```
<portType name="EmployeeInterface"> <  documentation  >
GetEmployeeWeeklyHoursLimit uses the Employee ID value to retrieve the
WeeklyHoursLimit value. UpdateEmployeeHistory uses the Employee ID
value to update the Comment value of the EmployeeHistory.
 </  documentation  > <operation name="GetEmployeeWeeklyHoursLimit">
<input message= "tns:getEmployeeWeeklyHoursRequestMessage"/>
<output message= "tns:getEmployeeWeeklyHoursResponseMessage"/>
</operation> <operation name="UpdateEmployeeHistory">
 <input message= "tns:updateEmployeeHistoryRequestMessage"/>
<output message= "tns:updateEmployeeHistoryResponseMessage"/>
</operation>
 </portType>
```

Step 5: Standardize and refine the service interface

Depending on your requirements, this can be a multi- faceted step involving a series of design tasks . Following is a list of recommended actions you can take to achieve a standardized and streamlined service design:
Review existing design standards and guidelines and apply any that are appropriate. (Use the guidelines and proposed standards provided at the end of this chapter as a starting point.)

In addition to achieving a standardized service interface design, this step also provides an opportunity for the service design to be revised in support of some of the contemporary SOA characteristics we identified in the Unsupported SOA characteristics.

If your design requirements include WS-I Basic Profile conformance, then that can become a consideration at this stage. Although Basic Profile compliance requires that the entire WSDL be completed, what has been created so far can be verified .3

Case Study

The TLS architect in charge of the Employee Service design decides to make adjustments to the abstract service interface to apply current design standards. Specifically, naming conventions are incorporated to standardize operation names,

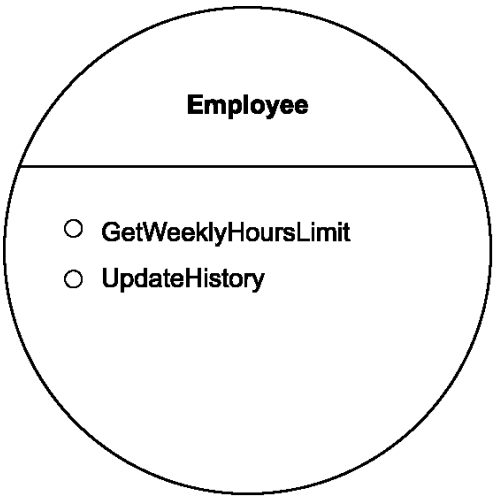


Figure 15.7: The revised Employee Service operation names.

The two operation constructs with new, standardized names.

```
<operation name="  GetWeeklyHoursLimit  ">
<input message="tns:getWeeklyHoursRequestMessage"/>
 <output message="tns:getWeeklyHoursResponseMessage"/>
</operation> <operation name="  UpdateHistory  ">
 <input message="tns:updateHistoryRequestMessage"/>
 <output message="tns:updateHistoryResponseMessage"/>
</operation>
```

Step 6: Extend the service design

The service modeling process tends to focus on evident business requirements.

While promoting reuse always is encouraged, it often falls to the design process to ensure that a sufficient amount of reusable functionality will be built into each service.

This is especially important for entity-centric business services, as a complete range of common operations typically is expected by their service requestors.

This step involves performing a speculative analysis as to what other types of features this service, within its predefined functional context, should offer.
There are two common ways to implement new functionality:

- add new operations
- add new parameters to existing operations

Case Study

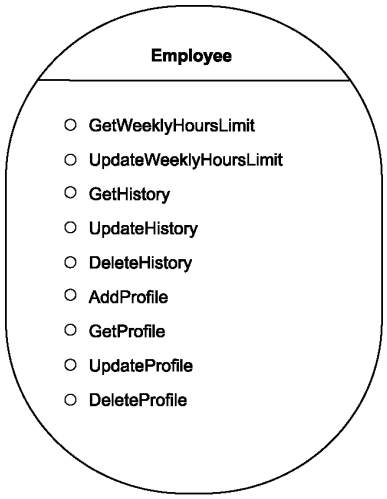


Figure 15.8: An Employee Service offering a full range of operations.

An expanded portType construct.

```
<portType name="EmployeeInterface"> <operation name="GetWeeklyHoursLimit">
 <input message="tns:getWeeklyHoursRequestMessage"/>
<output message="tns:getWeeklyHoursResponseMessage"/> </operation> <operation
name="  UpdateWeeklyHoursLimit  "> <input message=
"tns:updateWeeklyHoursRequestMessage"/> <output message=
"tns:updateWeeklyHoursResponseMessage"/> </operation>
 <operation name="  GetHistory  "> <input message="tns:getHistoryRequestMessage"/>
<output message="tns:getHistoryResponseMessage"/> </operation>
<operation name="UpdateHistory"> <input message="tns:updateHistoryRequestMessage"/>
 <output message="tns:updateHistoryResponseMessage"/>
 </operation> <operation name="  DeleteHistory  ">
 <input message="tns:deleteHistoryRequestMessage"/>
<output message="tns:deleteHistoryResponseMessage"/>
 </operation> <operation name="  AddProfile  "> <input
message="tns:addProfileRequestMessage"/> <output
message="tns:addProfileResponseMessage"/> </operation>
<operation name="  GetProfile  "> <input message="tns:getProfileRequestMessage"/>
<output message="tns:getProfileResponseMessage"/> </operation>
 <operation name="  UpdateProfile  "> <input message="tns:updateProfileRequestMessage"/>
 <output message="tns:updateProfileResponseMessage"/> </operation> <operation name="
DeleteProfile  ">
<input message="tns:deleteProfileRequestMessage"/>
<output message="tns:deleteProfileResponseMessage"/>
 </operation> </portType>
```

Step 7: Identify required processing

While the service modeling process from our service-oriented analysis may have identified some key application services, it may not have been possible to define them all.

Now that we have an actual design for this new business service, you can study the processing requirements of each of its operations more closely.

In doing so, you should be able to determine if additional application services are required to carry out each piece of exposed functionality. If you do find the need for new application services, you will have to determine if they already exist, or if they need to be added to the list of services that will be delivered as part of this solution.

Case Study

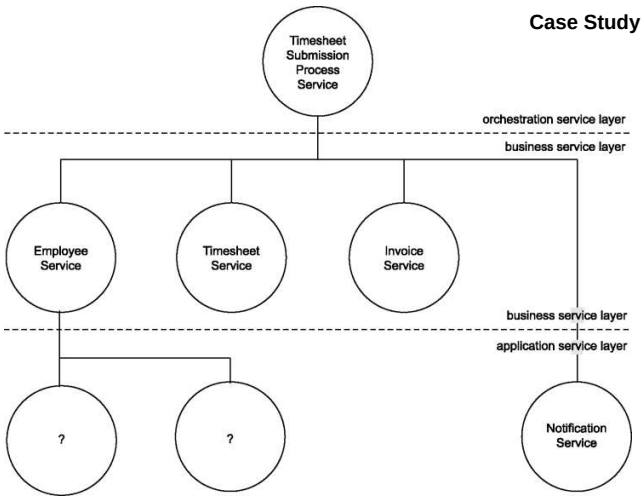


Figure 15.9: The revised composition hierarchy identifying new potential application services.

Case Study (Process Results)
The final abstract service definition.

```
<definitions name="Employee" targetNamespace="http://www.xmltc.com/tls/employee/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:act=
"http://www.xmltc.com/tls/employee/schema/accounting/"
xmlns:hr="http://www.xmltc.com/tls/employee/schema/hr/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.xmltc.com/tls/employee/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"> <types> <xsd:schema targetNamespace=
"http://www.xmltc.com/tls/employee/schema/"> <xsd:import namespace=
"http://www.xmltc.com/tls/employee/schema/ accounting/" schemaLocation="Employee.xsd"/>
<xsd:import namespace= "http://www.xmltc.com/tls/employee/schema/hr/"
schemaLocation="EmployeeHistory.xsd"/> </xsd:schema> </types> <message
name="getWeeklyHoursRequestMessage"> <part name="RequestParameter"
element="act:EmployeeHoursRequestType"/> </message> <message
name="getWeeklyHoursResponseMessage"> <part name="ResponseParameter"
element="act:EmployeeHoursResponseType"/> </message> <message
name="updateHistoryRequestMessage"> <part name="RequestParameter"
element="hr:EmployeeUpdateHistoryRequestType"/> </message> <message
name="updateHistoryResponseMessage"> <part name="ResponseParameter"
element="hr:EmployeeUpdateHistoryResponseType"/> </message> <portType
name="EmployeeInterface"> <documentation> GetWeeklyHoursLimit uses the Employee ID value
to retrieve the WeeklyHoursLimit value. UpdateHistory uses the Employee ID value to update the
Comment value of the EmployeeHistory. </documentation> <operation
name="GetWeeklyHoursLimit"> <input message= "tns:getWeeklyHoursRequestMessage"/> <output
message= "tns:getWeeklyHoursResponseMessage"/> </operation> <operation
name="UpdateHistory"> <input message= "tns:updateHistoryRequestMessage"/> <output
message= "tns:updateHistoryResponseMessage"/> </operation> </portType> ... </definitions>
```

Application services are the workhorses of SOA. They represent the bottom sub-layer of the composed service layer (Figure 15.10), responsible for carrying out any of the processing demands dictated to them by the business and orchestration layers .

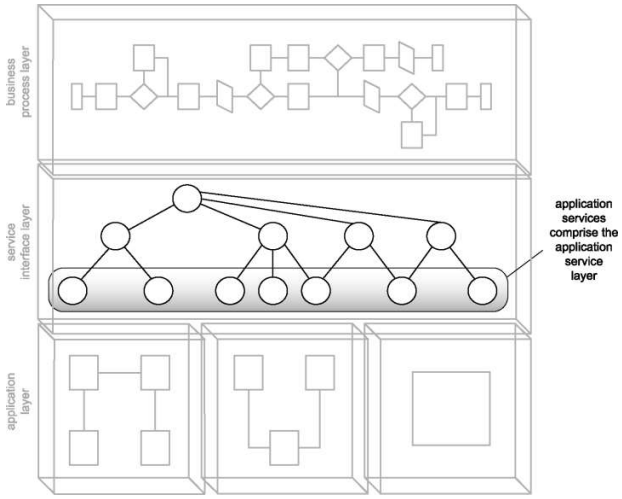


Figure 15.10: Application services establish the bottom sub-layer of the service layer.

Process Description

you'll notice that this process shares a number of steps with the previous entity-centric business service process.
This is because both ***application and entity-centric*** services establish reusable service logic and therefore rely on parent controllers to compose them into business process-specific tasks

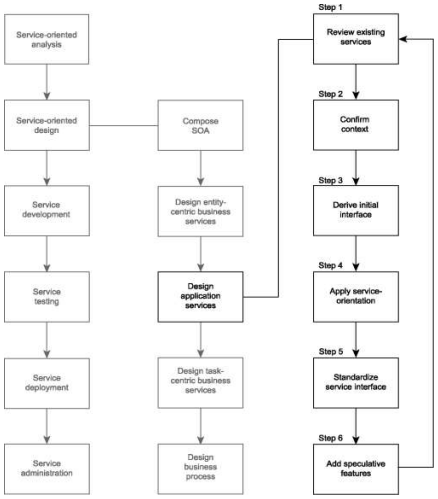


Figure 15.11: The application service design process.

Case Study

We now switch over to the RailCo environment, where the focus is on the design of the Transform Accounting Documents application service candidate.

This candidate establishes a "document transformation context," which justifies the grouping of its two very similar operation candidates:

- transform XML documents to native format
- transform native documents to XML

These two lines of information establish a base from which we can derive a physical service design via the steps in the upcoming design process.

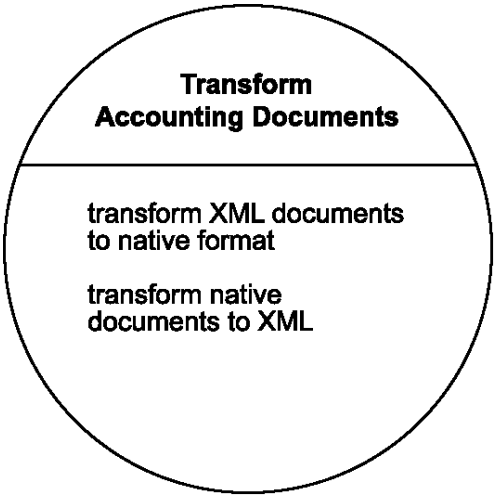


Figure 15.12: The Transform Accounting Documents Service candidate.

Step 1: Review existing services

More so with application services than with other types of reusable services, it is important to ensure that the functionality required, as per the application service candidate, does not, in some way, shape, or form, already exist.

So it is very necessary to review your existing inventory of application services in search of anything resembling what you are about to design.
Additionally, because these services provide such generic functionality, it is worth, at this stage, investigating whether the features you require can be purchased or leased from third-party vendors .

Because application services should be designed for maximum reusability, third-party Web services (which typically are built to be reusable) can make a great deal of sense, as long as required quality of service levels can be met.

Case Study

RailCo is delivering this service as part of a solution that is replacing their original hybrid Invoice Submission and Order Fulfillment Services.

The only other service that exists within the RailCo environment is the TLS Subscription Service, used to interact with the TLS publishing extension.

Therefore, this step is completed rather quickly, as little effort is required to determine that functionality planned for the Transform Accounting Document service will not end up being redundant.

Step 2: Confirm the context

When performing a service-oriented analysis it's natural to be focused on immediate business requirements.

As a result, application service candidates produced by this phase will frequently not take the contexts established by existing application services into account.

Therefore, it is important that the operation candidate grouping proposed by service candidates be re-evaluated and compared with existing application service designs.

Upon reassessing the service context, you may find that one or more operations actually belong in other application services.

Note
This step was not required as part of the entity-centric business service design, as the context of entity-centric services is predefined by the corresponding entity models.

Case Study

A review of the one existing RailCo service and the additional services planned as part of this solution confirms that the grouping context proposed for the two operation candidates of the Transform Accounting Documents service candidate is valid.

Step 3: Derive an initial service interface

Analyze the proposed service operation candidates and follow the steps below to define the first cut of the service interface:

- Using the application service candidate as your primary input, ensure that the granularity of the logic partitions represented by the operation candidates are appropriately generic and reusable.
- Document the input and output values required for the processing of each operation candidate and define message structures using XSD schema constructs (which essentially establishes the WSDL types construct).
- Complete the abstract service definition by adding the portType (or interface) area (along with its child operation constructs) and the necessary message constructs containing the part elements that reference the appropriate schema types.

Examples include:

1. Create a set of operations that are generic but still document specific. For example, instead of a single Add operation, you could provide separate AddInvoice, AddPO, and AddClaim operations.
2. Application services can be outfitted to support SOAP attachments, allowing a generic operation to issue a generic SOAP message containing a specific business document.

Case Study

RailCo begins by deriving the two operation names

It then moves on to define the types construct of its service definition to formalize the message structures. First, it tackles the request and response messages for the TransformToNative operation.

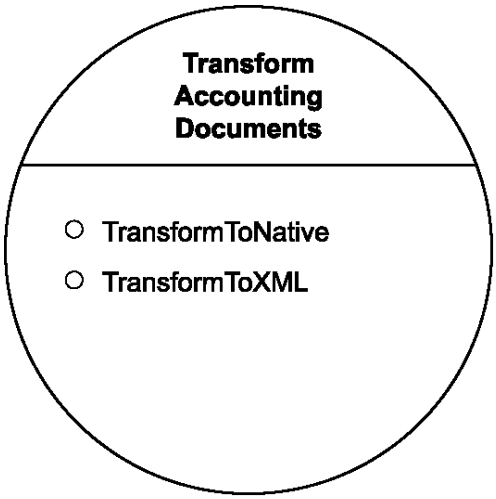


Figure 15.13: The first cut of the Transform Accounting Documents Service.

The XSD schema types required by the TransformToNative operation.

```
<xsd:schema targetNamespace=
"http://www.xmltc.com/railco/transform/schema/">
<xsd:element name="  TransformToNativeType  ">
 <xsd:complexType>
 <xsd:sequence> <xsd:element name="SourcePath"
type="xsd:string"/>
 <xsd:element name="DestinationPath"
type="xsd:string"/>
 </xsd:sequence>
 </xsd:complexType> </xsd:element>
<xsd:element name="
TransformToNativeReturnCodeType  ">
<xsd:complexType>
<xsd:sequence> <xsd:element name="Code"
type="xsd:integer"/>
 <xsd:element name="Message" type="xsd:string"/>
 </xsd:sequence> </xsd:complexType>
 </xsd:element> </xsd:schema>
```

The additional tpes for use by the TransformToXML operation.

```
<xsd:element name="TransformToXMLType">
<xsd:complexType> <xsd:sequence>
<xsd:element name="SourcePath"
type="xsd:string"/> <xsd:element
name="DestinationPath" type="xsd:string"/>
</xsd:sequence> </xsd:complexType>
</xsd:element>
<xsd:element
name="TransformToXMLReturnCodeType">
<xsd:complexType>
 <xsd:sequence> <xsd:element name="Code"
type="xsd:integer"/>
<xsd:element name="Message"
type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

The message and portType constructs of the abstract Transform Accounting Documents Service definition.

```
<message name="transformToNativeRequestMessage">
 <part name="RequestParameter" element="trn:TransformToNativeType"/>
</message> <message name="transformToNativeResponseMessage">
 <part name="ResponseParameter"
element="trn:TransformToNativeReturnCodeType"/> </message>
 <message name="transformToXMLRequestMessage"> <part
name="RequestParameter" element="trn:TransformToXMLType"/>
</message> <message name="transformToXMLResponseMessage">
<part name="ResponseParameter"
element="trn:TransformToXMLReturnCodeType"/> </message>
<portType name="TransformInterface">
 <operation name="TransformToNative">
<input message= "tns:transformToNativeRequestMessage"/>
 <output message= "tns:transformToNativeResponseMessage"/> </operation>
<operation name="TransformToXML"> <input message=
"tns:transformToXMLRequestMessage"/>
<output message= "tns:transformToXMLResponseMessage"/>
 </operation>
</portType>
```

Step 4: Apply principles of service-orientation

This step highlights the four principles of service-orientation we listed in Chapter 8, as being those that are not intrinsically provided by the Web services platform (service reusability, service autonomy, service statelessness, and service discoverability).

Autonomy is of primary concern when designing application services. We must ensure that the underlying application logic responsible for executing the service operations does not impose dependencies on the service, or itself have dependencies.

This is where the information we gathered in Step 2 of the service-oriented analysis process provides us with a starting point to investigate the nature of the application logic each service operation needs to invoke. Step 6 provides an analysis that covers this and other technology-related issues.

Statelessness also may be more difficult to achieve with application services. Because they are required to interface with a variety of different application platforms, these services are subject to highly unpredictable implementation environments.

Sooner or later, application services are bound to encounter challenges that impose unreasonable or inconsistent performance requirements .

Therefore, the best way to promote a stateless application service design is to carry out as much up-front analysis as possible. Knowing in advance what the performance demands will be will allow you to investigate alternatives before you commit to a particular design.

The Transform Accounting DocumentsportType construct with supplemental metadata documentation.

```
<portType name="TransformInterface">
 <  documentation  > Retrieves an XML document and
converts it into the native accounting document format.
 </  documentation  >
 <operation name="TransformToNative"> <input
message= "tns:transformToNativeRequestMessage"/>
<output message=
"tns:transformToNativeResponseMessage"/>
</operation>
 <  documentation  >
 Retrieves a native accounting document and converts it
into an XML document. </  documentation  >
<operation name="TransformToXML">
 <input message=
"tns:transformToXMLRequestMessage"/>
<output message=
"tns:transformToXMLResponseMessage"/> </operation>
</portType>
```

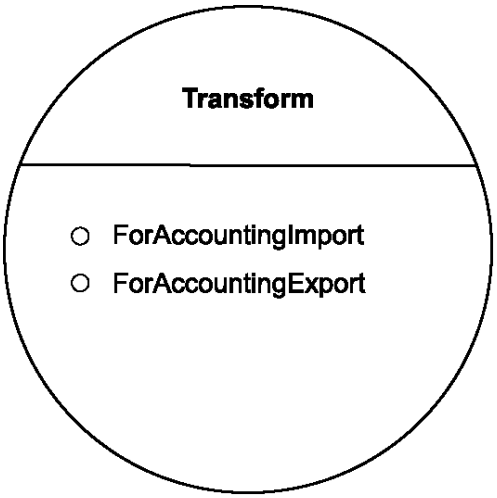


Figure 15.14: The final design of the Transform Service.

Step 5: Standardize and refine the service interface

Even though the role and purpose of application services differs from other types of services, it is important that they be designed in the same fundamental manner. We accomplish this by ensuring that the resulting application service WSDL definition is based on the same standards and conventions used by others.

Following is a list of recommended actions you can take to achieve a standardized and streamlined service design:

- Apply any existing design standards relevant to the service interface. (For a list of suggested standards, review the guidelines provided at the end of this chapter.)
- Review any of the contemporary SOA characteristics you've chosen to have your services support and assess whether it is possible to build support for this characteristic into this service design.
- Optionally incorporate WS-I Basic Profile rules and best practices to whatever extent possible.

The revised types construct.

```
<types>
 <xsd:schema targetNamespace=
"http://www.xmltc.com/railco/transform/schema/">
 <xsd:element name="  ForImportType  ">
<xsd:complexType> <xsd:sequence> <xsd:element name="SourcePath"
type="xsd:string"/> <xsd:element name="DestinationPath"
type="xsd:string"/> </xsd:sequence> </xsd:complexType> </xsd:element>
<xsd:element name="  ForImportReturnCodeType  ">
<xsd:complexType> <xsd:sequence> <xsd:element name="Code"
type="xsd:integer"/> <xsd:element name="Message" type="xsd:string"/>
 </xsd:sequence> </xsd:complexType> </xsd:element>
<xsd:element name="  ForExportType  "> <xsd:complexType>
<xsd:sequence>
<xsd:element name="SourcePath" type="xsd:string"/>
<xsd:element name="DestinationPath" type="xsd:string"/>
</xsd:sequence> </xsd:complexType> </xsd:element>
<xsd:element name="  ForExportReturnCodeType  ">
<xsd:complexType>
<xsd:sequence> <xsd:element name="Code" type="xsd:integer"/>
<xsd:element name="Message" type="xsd:string"/>
 </xsd:sequence> </xsd:complexType> </xsd:element>
</xsd:schema>
</types>
```

Step 6: Outfit the service candidate with speculative features

If you are interested in delivering highly reusable application services, you can take this opportunity to add features to this service design.

These new features can affect existing operations or can result in the addition of new operations. For application services, speculative extensions revolve around the type of processing that falls within the service context.

Of course, before actually adding speculative extensions to the application service, you should repeat Step 1 to confirm that no part of these new operations already exists within other services.

Additionally, when adding new extensions, Steps 2 through 5 also need to be repeated to ensure that they are properly standardized and designed in alignment with the portion of the service interface we've created so far.

Step 7: Identify technical constraints

At this point we have created an ideal service interface in a bit of a vacuum . Unlike our business services, application services need to take low-level, real-world considerations into account.

The types of details we are specifically looking for are:

- The physical connection point of the particular function. (In other words, whatcomponents need to be invoked, what API functions need to be called, or which adapters need to be activated.)
- Security constraints related to any part of the processing.
- Response time of each processing function.
- Availability of the underlying system performing the processing function.
- Environmental factors relating to service deployment location.
- Technical limitations of underlying application logic (especially when exposing legacy systems).
- Administration requirements imposed by the service.
- Potential SLA requirements.

Case Study (Process Results)

Following is the final version of the Transform Service definition, incorporating the changes to element names and all of the previous revisions.

The final abstract service definition.

<definitions name="Transform" targetNamespace= "http://www.xmltc.com/railco/transform/wsdl/" xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://www.xmltc.com/railco/transform/wsdl/" xmlns:trn="http://www.xmltc.com/railco/transform/schema/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"> <types> <xsd:schema targetNamespace= "http://www.xmltc.com/railco/transform/schema/"> <xsd:element name="ForImportType"> <xsd:complexType> <xsd:sequence> <xsd:element name="SourcePath" type="xsd:string"/> <xsd:element name="DestinationPath" type="xsd:string"/> </xsd:sequence> </xsd:complexType> </xsd:element> <xsd:element name="ForImportReturnCodeType"> <xsd:complexType> <xsd:sequence> <xsd:element name="Code" type="xsd:integer"/> <xsd:element name="Message" type="xsd:string"/> </xsd:sequence> </xsd:complexType> </xsd:element> <xsd:element name="ForExportType"> <xsd:complexType> <xsd:sequence> <xsd:element name="SourcePath" type="xsd:string"/> <xsd:element name="DestinationPath" type="xsd:string"/> </xsd:sequence> </xsd:complexType> </xsd:element> <xsd:element name="ForExportReturnCodeType"> <xsd:complexType> <xsd:sequence> <xsd:element name="Code" type="xsd:integer"/> <xsd:element name="Message" type="xsd:string"/> </xsd:sequence> </xsd:complexType> </xsd:element> </xsd:schema> </types> <message name="ForAccountingImportRequestMessage"> <part name="RequestParameter" element="trn:ForImportType"/> </message> <message name="ForAccountingImportResponseMessage"> <part name="ResponseParameter" element="trn:ForImportReturnCodeType"/> </message> <message name="ForAccountingExportRequestMessage"> <part name="RequestParameter" element="trn:ForExportType"/> </message> <message name="ForAccountingExportResponseMessage"> <part name="ResponseParameter" element="trn:ForExportReturnCodeType"/> </message> <portType name="TransformInterface"> <documentation> ForAccountingImport retrieves an XML document and converts it into the native accounting document format. ForAccountingExport retrieves a native accounting document and converts it into an XML document. </documentation> <operation name="ForAccountingImport"> <input message= "tns:ForAccountingImportRequestMessage"/> <output message= "tns:ForAccountingImportResponseMessage"/> </operation> <operation name="ForAccountingExport"> <input message= "tns:ForAccountingExportRequestMessage"/> <output message= "tns:ForAccountingExportResponseMessage"/> </operation> </portType> ... </definitions>

Section 15.4. Task-centric business service design (a step-by-step process)

The process for designing task-centric services usually require less effort than the previous two design processes, simply because reuse is generally not a primary consideration.

Therefore, only the service operation candidates identified as part of the service modeling process are addressed here.

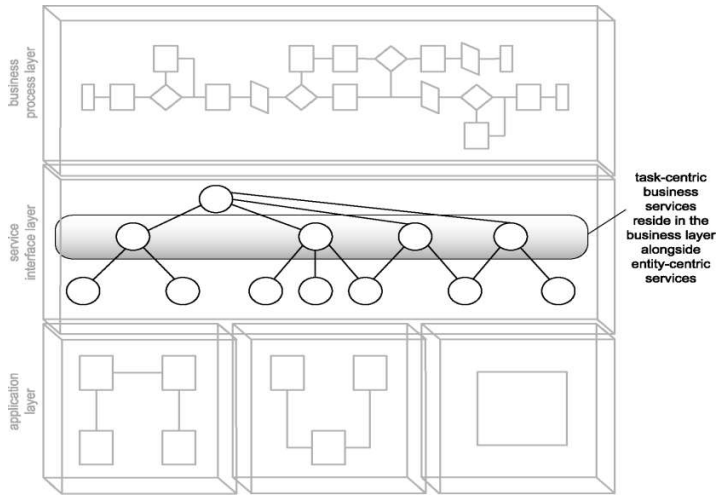


Figure 15.15: Task-centric business services can comprise the business service layer, along with entity-centric neighbors.

Process description

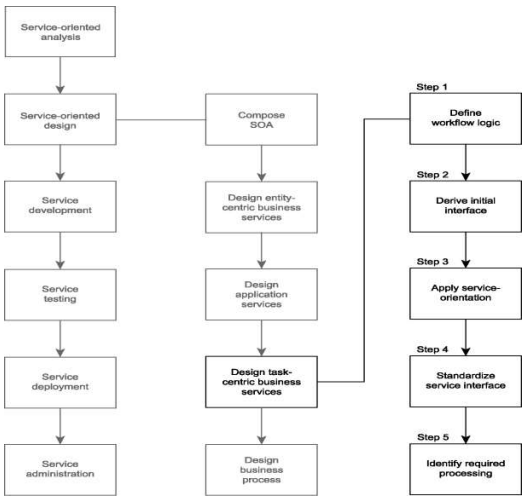


Figure 15.16: The task-centric business service design process.

Case Study

The RailCo service modeling process identified the need for a task-centric business service to govern the processing of invoices produced by the legacy accounting system.

This resulted in the Invoice Processing Service candidate

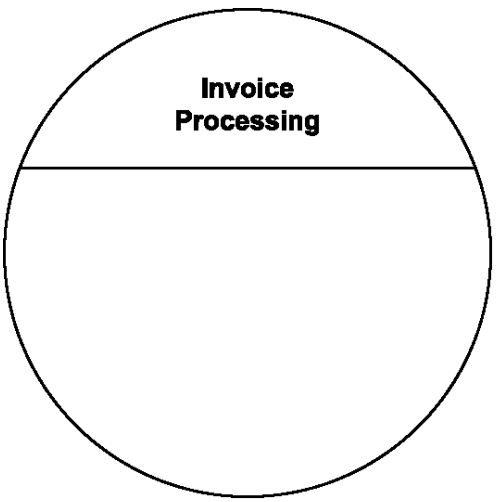


Figure 15.17: The Invoice Processing Service candidate.

It looks like the Invoice Processing Service actually will be quite busy, as it needs to compose up to four separate services to process a single invoice submission.

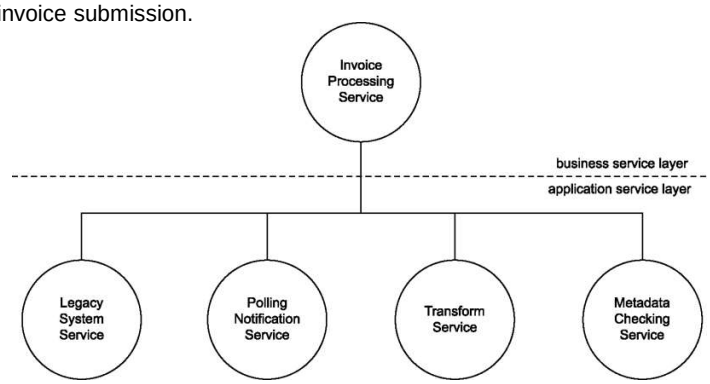


Figure 15.18: The Invoice Processing Service composition.

Step 1: Define the workflow logic

Task-centric services typically will contain embedded workflow logic used to coordinate an underlying service composition.

Our first step, therefore, is to define this logic for every possible interaction scenario we can imagine.

If you performed the mapping exercise in the Identify candidate service compositions step of the service modeling process in Chapter 12, then you will have preliminary composition details already documented.

Because we are designing our task-centric business service after our entity-centric and application service designs have been completed, we will need to revisit these scenario documents and turn them into concrete service interaction models.
Different traditional modeling approaches can be used to accomplish this step (we use simple activity diagrams in our case study examples). The purpose of this exercise is to document each possible execution path , including all exception conditions. The resulting diagrams also will be useful input for subsequent test cases.

Case Study

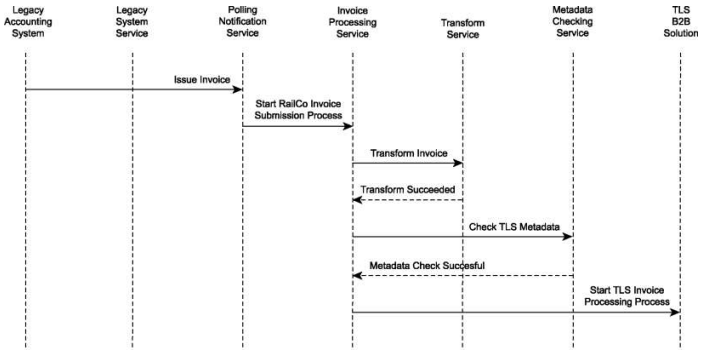


Figure 15.19: A successful completion of the Invoice Submission Process.

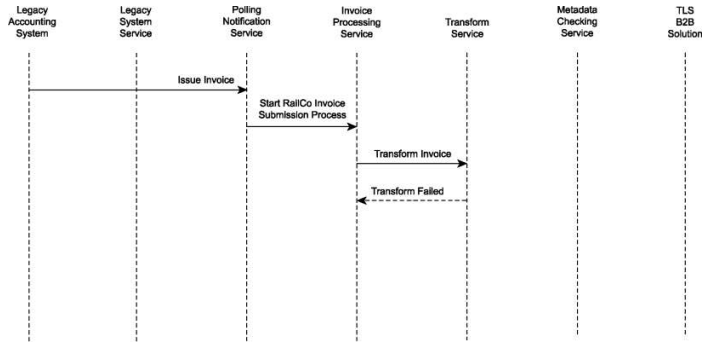


Figure 15.20: A failure condition caused by an error during the transformation step.

Step 2: Derive the service interface

Follow these suggested steps to assemble an initial service interface:

1.	Use the application service operation candidates to derive a set of corresponding operations.
2.	Unlike previous design processes, the source from which we derive our service interface this time also includes the activity diagrams and the workflow logic we documented in Step 1. This information gives us a good idea as to what additional operations our task-centric service may require.
3.	Document the input and output values required for the processing of each operation and populate the types section with XSD schema types required to process the operations.
4.	Build the WSDL definition by creating the portType (or interface) area, inserting the identified operation constructs. Then add the necessary messageconstructs containing the part elements that reference the appropriate schema types.

Case Study

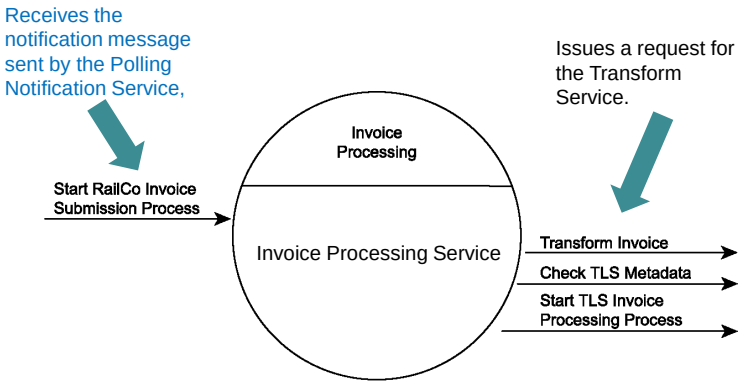


Figure 15.21: Identified requests and responses for the Invoice Processing Service.

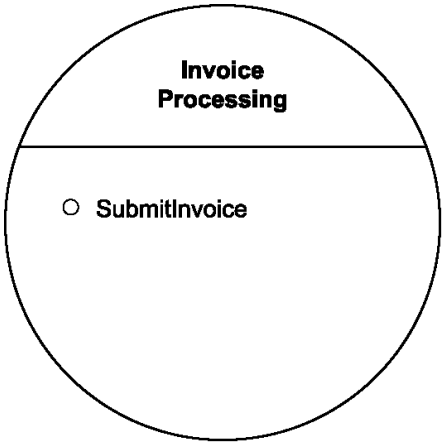


Figure 15.22: The Invoice Processing Service with a new operation.

Step 3: Apply principles of service-orientation

Task-centric services contain workflow logic that may impose processing dependencies in service compositions.

This can lead to the need for state management. However, the use of document-style SOAP messages allows the service to delegate the persistence of some or all of this state information to the message itself.

It is always useful for services to be discoverable, but the need for task-centric services to be discovered is not as pressing as with other, more generically reusable services.

Regardless, task-centric services can achieve reuse, and their existence should be known to others. Therefore, the Document services with metadata guideline provided at the end of this chapter also is recommended.

Case Study

There is no requirement for the Invoice Processing Service to be reusable, and autonomy and statelessness are also not considered immediate concerns.

As with the RailCo Transform Service that was designed previously, this service design is supplemented with additional metadata documentation to support discoverability.

The portType construct with an additionaldocumentation element.

```
<portType name="InvoiceProcessingInterface">
 <  documentation  > Initiates the Invoice Submission Process.
 </  documentation  > <operation name="SubmitInvoice"> <input
message="tns:receiveSubmitInvoiceMessage"/>
 </operation>
</portType>
```

Step 4: Standardize and refine the service interface

Although task-centric business services will tend to have more creative operation names , existing conventions still need to be applied.
Here is the standard list of recommended actions you can take to achieve a standardized and streamlined service design:

- Incorporate existing design standards and guidelines.
- Ensure that any chosen contemporary SOA characteristics are fully supported by the service interface design.
- Take WS-I Basic Profile standards and best practices into account.

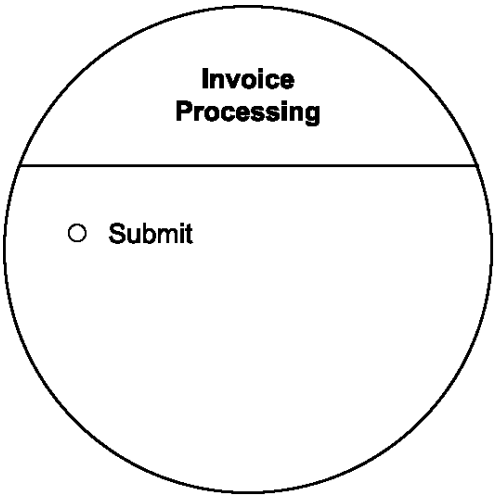


Figure 15.23: The Invoice Processing Service with a revised operation name.

Step 5: Identify required processing

To carry out their share of a solution's process logic, task-centric services can compose application and both entity-centric and additional task-centric business services.

Therefore, the implementation of a task-centric service interface requires that any needed underlying service layers are in place to support the processing requirements of its operations.

The final abstract service definition.

```
<definitions name="Transform" targetNamespace=
"http://www.xmltc.com/railco/transform/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.xmltc.com/railco/transform/wsdl/"
xmlns:trn="http://www.xmltc.com/railco/transform/schema/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"> <types>
 <xsd:schema targetNamespace=
"http://www.xmltc.com/railco/transform/schema/">
<definitions name="InvoiceProcessing" targetNamespace=
http://www.xmltc.com/railco/transform/wsdl/
 xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:inv="http://www.xmltc.com/railco/invoice/schema/"
xmlns:invs= "http://www.xmltc.com/railco/invoiceservice/schema/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.xmltc.com/railco/transform/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <types> <xsd:schema targetNamespace=
"http://www.xmltc.com/railco/invoiceservice/
 schema/"> <xsd:import namespace=
"http://www.xmltc.com/railco/invoice/schema/"
 schemaLocation="Invoice.xsd"/> <xsd:element name="SubmitInvoiceType"> <xsd:complexType>
<xsd:sequence> <xsd:element name="ContextID" type="xsd:integer"/> <xsd:element
name="InvoiceLocation" type="xsd:string"/> <xsd:element name="InvoiceDocument"
type="inv:InvoiceType"/> </xsd:sequence> </xsd:complexType> </xsd:element> </xsd:schema> </types>
<message name="receiveSubmitMessage"> <part name="RequestParameter"
element="invs:SubmitInvoiceType"/> </message> <portType name="InvoiceProcessingInterface">
<documentation> Initiates the Invoice Submission Process. Requires either the invoice document location
or the document. </documentation> <operation name="Submit"> <input
message="tns:receiveSubmitMessage"/> </operation> </portType> ... </definitions>
```

Section 15.5. Service design guidelines

1. Apply naming standards

Labeling services is the equivalent to labeling IT infrastructure. It is therefore essential that service interfaces be as consistently self-descriptive as possible.
Naming standards therefore need to be defined and applied to:

- service endpoint names
- service operation names
- message values

2. Apply a suitable level of interface granularity
3. Design service operations to be inherently extensible
4. Identify known and potential service requestors
5. Consider using modular WSDL documents
6. Use namespaces carefully

Chapter 16.

Service-Oriented Design (Part IV: Business Process Design)

16.1. WS-BPEL language basics

The ***orchestration service layer*** provides a powerful means by which contemporary service-oriented solutions can realize some key benefits.

The most significant contribution this sub-layer brings to SOA is an abstraction of logic and responsibility that alleviates underlying services from a number of design constraints.

For example, by abstracting business process logic:

- Application and business services can be freely designed to be process- agnostic and reusable.
- The process service assumes a greater degree of statefulness, thus further freeing other services from having to manage state.
- The business process logic is centralized in one location, as opposed to being distributed across and embedded within multiple services.

Before we can design an orchestration layer, we need to acquire a good understanding of how the operational characteristics of the process can be formally expressed .

We uses the WS-BPEL language to demonstrate how process logic can be described as part of a concrete definition that can be implemented and executed via a compliant orchestration engine.

Although you likely will be using a process modeling tool and will therefore not be required to author your process definition from scratch, a knowledge of WS-BPEL elements still is useful and often required.

WS-BPEL modeling tools frequently make reference to these elements and constructs, and you may be required to dig into the source code they produce to make further refinements.

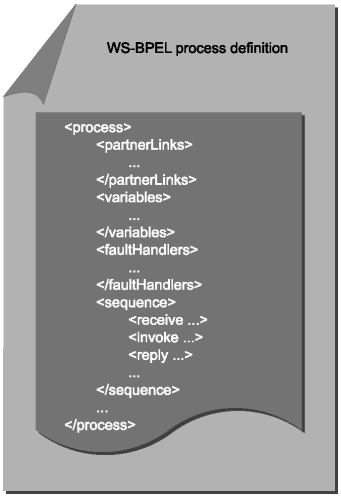


Figure 16.1: A common WS-BPEL process definition structure.

The process element

Let's begin with the root element of a WS-BPEL process definition. It is assigned a name value using the name attribute and is used to establish the process definition- related namespaces.

A skeleton process definition

```
< process name="TimesheetSubmissionProcess"
targetNamespace="http://www.xmltc.com/tls/process/" xmlns=
"http://schemas.xmlsoap.org/ws/2003/03/ business-process/"
xmlns:bpl="http://www.xmltc.com/tls/process/"
xmlns:emp="http://www.xmltc.com/tls/employee/"
xmlns:inv="http://www.xmltc.com/tls/invoice/"
xmlns:tst="http://www.xmltc.com/tls/timesheet/"
xmlns:not="http://www.xmltc.com/tls/notification/">
<partnerLinks> ... </partnerLinks> <variables> ... </variables>
<sequence> ... </sequence> ...
</ process >
```

The partnerLinks and partnerLink elements

A partnerLink element establishes the port type of the service (partner) that will be participating during the execution of the business process.

Partner services can act as a client to the process, responsible for invoking the process service. Alternatively, partner services can be invoked by the process service itself.

The contents of a partnerLink element represent the communication exchange between two partnersthe process service being one partner and another service being the other.
Depending on the nature of the communication, the role of the process service will vary. For instance, a process service that is invoked by an external service may act in the role of "TimesheetSubmissionProcess."

However, when this same process service invokes a different service to have an invoice verified , it acts within a different role, perhaps "InvoiceClient." ThepartnerLink element therefore contains
the myRole and partnerRole attributes that establish the service provider role of the process service and the partner service respectively.

The partnerLinks construct containing onepartnerLink element in which the process service is invoked by an external client partner and
four partnerLink elements that identify partner services invoked by the process service.

```
< partnerLinks > < partnerLink name="client"
partnerLinkType="tns:TimesheetSubmissionType"
myRole="TimesheetSubmissionServiceProvider"/>
 < partnerLink name="Invoice" partnerLinkType="inv:InvoiceType"
partnerRole="InvoiceServiceProvider"/>
 < partnerLink name="Timesheet" partnerLinkType="tst:TimesheetType"
partnerRole="TimesheetServiceProvider"/>
 < partnerLink name="Employee"
partnerLinkType="emp:EmployeeType"
partnerRole="EmployeeServiceProvider"/>
 < partnerLink name="Notification"
partnerLinkType="not:NotificationType"
partnerRole="NotificationServiceProvider"/>
 </ partnerLinks >
```

The partnerLinkType element

For each partner service involved in a process, partnerLinkType elements identify the WSDLportType elements referenced by the partnerLink elements within the process definition.

Therefore, these constructs typically are embedded directly within the WSDL documents of every partner service (including the process service).

The partnerLinkType construct contains one role element for each role the service can play, as defined by the partnerLink myRole and partnerRole attributes. As a result, apartnerLinkType will have either one or two child role elements.

A WSDL definitions construct containing apartnerLinkType construct.

```
<definitions name="Employee"
targetNamespace="http://www.xmltc.com/tls/employe
e/wsdl/" xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:plnk=
"http://schemas.xmlsoap.org/ws/2003/05/partner-
link/" … > ... <plnk: partnerLinkType
name="EmployeeServiceType" xmlns=
"http://schemas.xmlsoap.org/ws/2003/05/partner-
link/">
<plnk:role name="EmployeeServiceProvider">
<portType name="emp:EmployeeInterface"/>
</plnk:role>
</plnk: partnerLinkType > ...
 </ definitions >
```

The variables element

WS-BPEL process services commonly use the variables construct to store state information related to the immediate workflow logic.

Entire messages and data sets formatted as XSD schema types can be placed into a variable and retrieved later during the course of the process.

The type of data that can be assigned to a variable element needs to be predefined using one of the following three attributes: messageType , element , or type .

The messageType attribute allows for the variable to contain an entire WSDL-defined message, whereas the element attribute simply refers to an XSD element construct.

Thetype attribute can be used to just represent an XSD simpleType , such as string or integer.

The variables construct hosting only some of the childvariable elements used later by the Timesheet Submission Process.

```
< variables >
 < variable name="ClientSubmission"
messageType="bpl:receiveSubmitMessage"/>
< variable name="EmployeeHoursRequest"
messageType="emp:getWeeklyHoursRequestMessage"/> < variable
name="EmployeeHoursResponse"
messageType="emp:getWeeklyHoursResponseMessage"/>
 < variable name="EmployeeHistoryRequest"
messageType="emp:updateHistoryRequestMessage"/>

 < variable name="EmployeeHistoryResponse"

messageType="emp:updateHistoryResponseMessage"/>

 ... </ variables >
```

Typically, a variable with the messageType attribute is defined for each input and output message processed by the process definition. The value of this attribute is the message name from the partner process definition.

The getVariableProperty and getVariableDatafunctions

WS-BPEL provides built-in functions that allow information stored in or associated with variables to be processed during the execution of a business process.

1. getVariableProperty(variable name, property name)
2. getVariableData(variable name, part name, location path)

Ex.
getVariableData ('InvoiceHoursResponse', 'ResponseParameter')
getVariableData ('input','payload','/tns:TimesheetType/Hours/...')

The sequence element

The sequence construct allows you to organize a series of activities so that they are executed in a predefined, sequential order.

WS-BPEL provides numerous activities that can be used to express the workflow logic within the process definition.

The remaining element descriptions in this section explain the fundamental set of activities used as part of our upcoming case study examples.

A skeleton sequence construct containing only some of the many activity elements provided by WS-BPEL.

```
< sequence >
 <receive> ... </receive>
 <assign> ... </assign>
 <invoke> ... </invoke>
 <reply> ... </reply>
</ sequence >
```

The invoke element

This element identifies the operation of a partner service that the process definition intends to invoke during the course of its execution. The invoke element is equipped with five common attributes, which further specify the details of the invocation

Attribute	Description
partnerLink	This element names the partner service via its correspondingpartnerLink .
portType	The element used to identify the portType element of the partner service.
operation	The partner service operation to which the process service will need to send its request.
inputVariable	The input message that will be used to communicate with the partner service operation. Note that it is referred to as a variable because it is referencing a WS-BPEL variable element with amessageType attribute.
outputVariable	This element is used when communication is based on the request-response MEP. The return value is stored in a separate variable element.

The invoke element identifying the target partner service details.

```
< invoke name=
"ValidateWeeklyHours" partnerLink="Employee"
portType="emp:EmployeeInterface" operation=
"GetWeeklyHoursLimit" inputVariable=
"EmployeeHoursRequest" outputVariable=
"EmployeeHoursResponse"/>
```

The receive element

The receive element allows us to establish the information a process service expects upon receiving a request from an external client partner service.
In this case, the process service is viewed as a service provider waiting to be invoked.
The receive element contains a set of attributes, each of which is assigned a value relating to the expected incoming communication.

Attribute	Description
partnerLink	The client partner service identified in the correspondingpartnerLink construct.
portType	The process service portType that will be waiting to receive the request message from the partner service.
operation	The process service operation that will be receiving the request.
variable	The process definition variable construct in which the incoming request message will be stored.
createInstance	When this attribute is set to "yes," the receipt of this particular request may be responsible for creating a new instance of the process.

The receive element used in the Timesheet Submission Process definition to indicate the client partner service responsible for launching the process with the submission of a timesheet document.

```
< receive name="receiveInput" partnerLink="client"
portType="tns:TimesheetSubmissionInterface" operation="Submit"
variable="ClientSubmission" createInstance="yes"/>
```

The reply element

Where there's a receive element, there's a reply element when a synchronous exchange is being mapped out. The reply element is responsible for establishing the details of returning a response message to the requesting client partner service. Because this element is associated with the same partnerLink element as its corresponding receive element, it repeats a number of the same attributes

Attribute	Description
partnerLink	The same partnerLink element established in the receiveelement.
portType	The same portType element displayed in the receive element.
operation	The same operation element from the receive element.
variable	The process service variable element that holds the message that is returned to the partner service.
messageExchange	It is being proposed that this optional attribute be added by the WS-BPEL 2.0 specification. It allows for the reply element to be explicitly associated with a message activity capable of receiving a message (such as the receive element).

A potential companion reply element to the previously displayed receive element.

```
< reply partnerLink=
"client" portType=
"tns:TimesheetSubmissionInterface" operation=
"Submit" variable=
"TimesheetSubmissionResponse"/>
```

Section 16.2. WS-Coordination overview

The CoordinationContext element

This parent construct contains a series of child elements that each house a specific part of the context information being relayed by the header.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsc=
"http://schemas.xmlsoap.org/ws/2002/08/wscoor" xmlns:wsu=
"http://schemas.xmlsoap.org/ws/2002/07/utility">

<Header>
 <wsc: CoordinationContext >
 <wsu:Identifier> ... </wsu:Identifier>
<wsu:Expires> ... </wsu:Expires>
 <wsc:CoordinationType> ... </wsc:CoordinationType>
<wsc:RegistrationService> ... </wsc:RegistrationService>
</wsc: CoordinationContext >
 </Header>
 <Body> ... </Body>
 </Envelope>
```

Section 16.3. Service-oriented business process design (a step-by-step process)

Designing the process of a service-oriented solution really just comes down to properly interpreting the business process requirements you have collected and then implementing them accurately.

The trick, though, is to also account for all possible variations of process activity. This means understanding not just what can go wrong, but how the process will respond to unexpected or abnormal conditions.

This gap is being addressed by operational business modeling languages, such as WS-BPEL.

Modeling tools exist, allowing technical analysts and architects to graphically create business process diagrams that represent their workflow logic requirements, all the while auto-generating WS-BPEL syntax in the background.

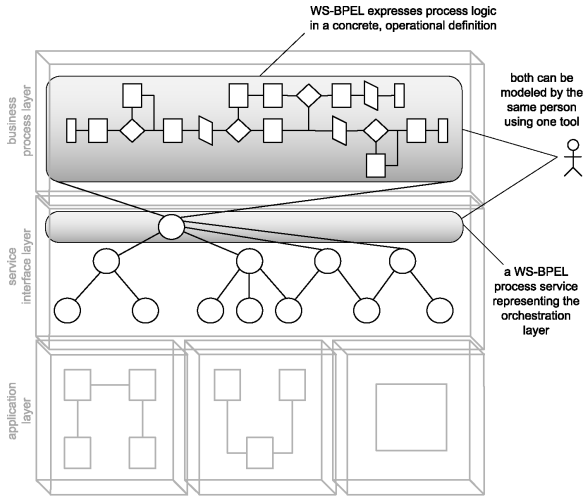


Figure 16.2: A concrete definition of a process service designed using a process modeling tool.

Process Description

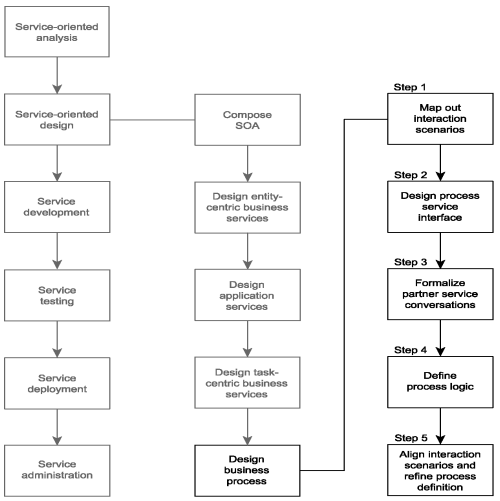


Figure 16.3: A high-level process for designing business processes.

Case Study

The original workflow logic for the TLS Timesheet Submission Process

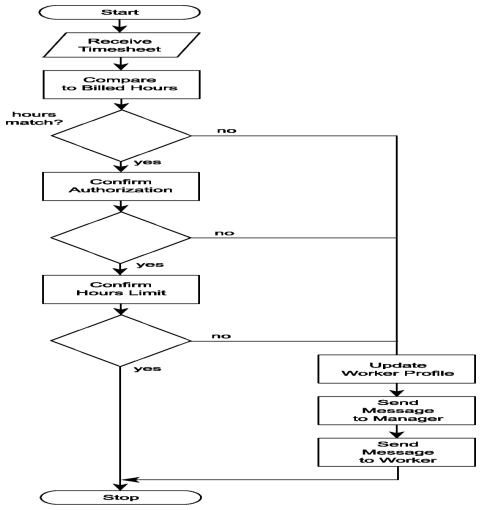


Figure 16.4: The original TLS Timesheet Submission Process.

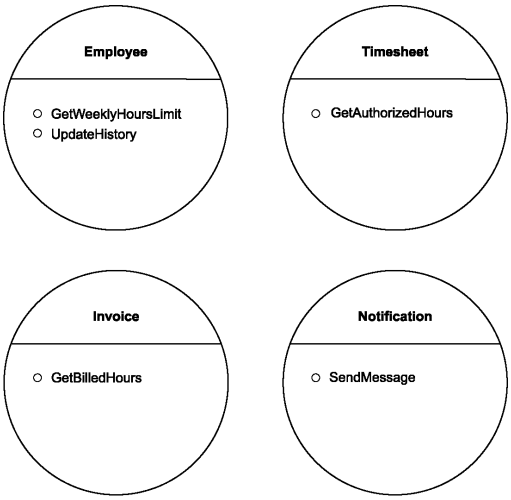


Figure 16.5: Service designs created so far.

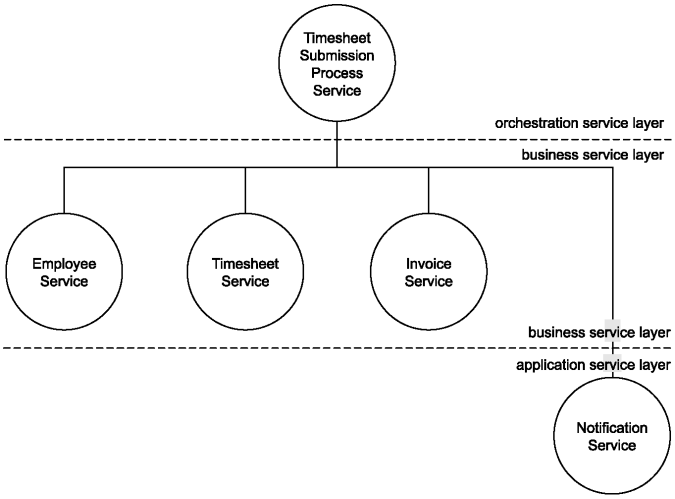


Figure 16.6: The original service composition defined during the service modeling stage.

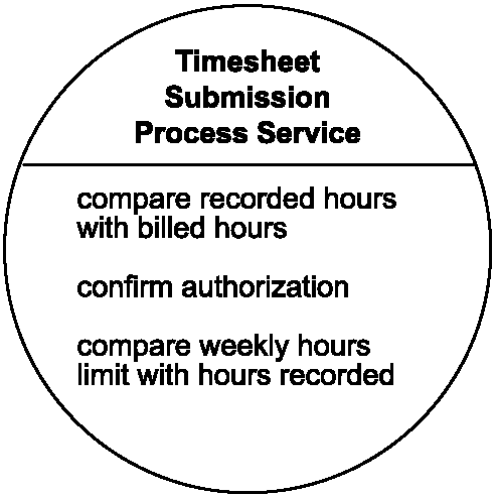


Figure 16.7: The Timesheet Submission Process Service candidate.

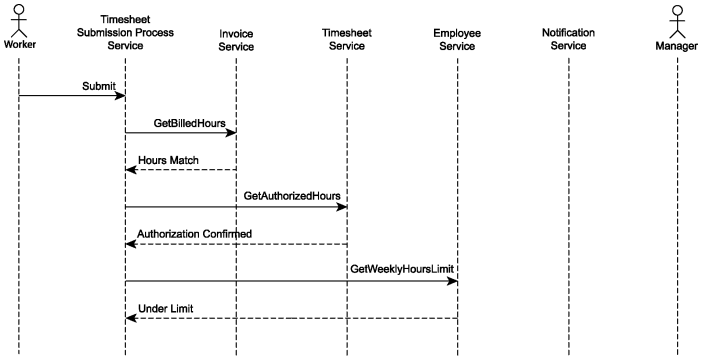


Figure 16.8: A successful completion of the Timesheet Submission Process.

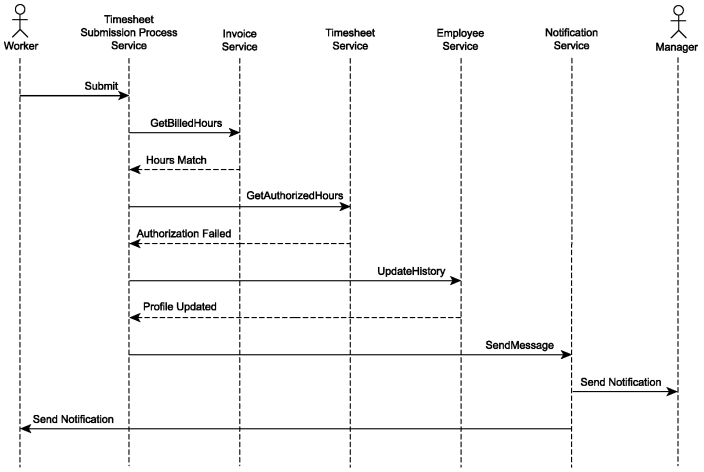


Figure 16.9: A failure condition caused by an authorization rejection.

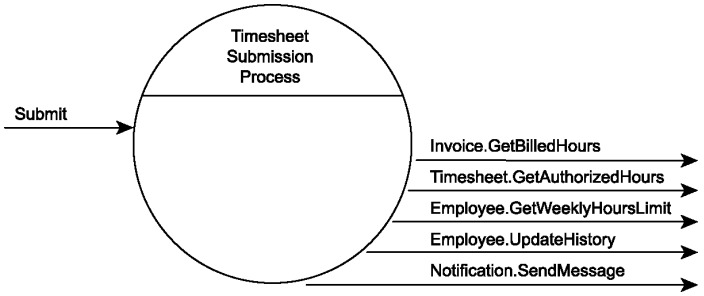


Figure 16.10: The incoming and outgoing request messages expected to be processed by the Timesheet Submission Process Service.

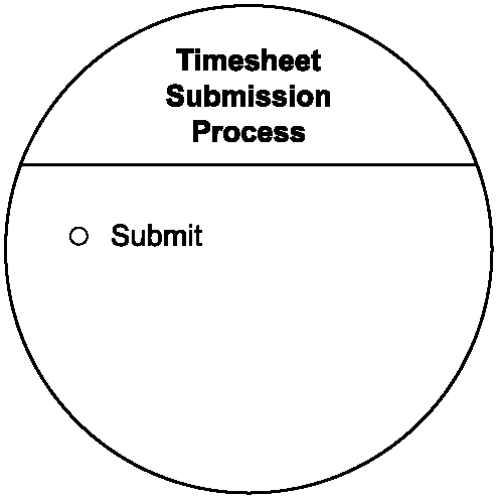


Figure 16.11: Timesheet Submission Process Service design.

The abstract service definition for the Timesheet Submission Process Service.

```
<definitions name="TimesheetSubmission"
targetNamespace="http://www.xmltc.com/tls/process/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:ts="http://www.xmltc.com/tls/timesheet/schema/" xmlns:tsd=
"http://www.xmltc.com/tls/timesheetservice/schema/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.xmltc.com/tls/timesheet/wsdl/" xmlns:plnk=
"http://schemas.xmlsoap.org/ws/2003/05/partner-link/"> <types> <xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace=
"http://www.xmltc.com/tls/ timesheetsubmissionservice/schema/"> <xsd:import namespace=
"http://www.xmltc.com/tls/timesheet/schema/" schemaLocation="Timesheet.xsd"/>
<xsd:element name="Submit"> <xsd:complexType> <xsd:sequence> <xsd:element
name="ContextID" type="xsd:integer"/> <xsd:element name="TimesheetDocument"
type="ts:TimesheetType"/> </xsd:sequence> </xsd:complexType> </xsd:element>
</xsd:schema> </types> <message name="receiveSubmitMessage"> <part
name="Payload" element="tsd:TimesheetType"/> </message> <portType
name="TimesheetSubmissionInterface"> <documentation> Initiates the Timesheet
Submission Process. </documentation> <operation name="Submit"> <input
message="tns:receiveSubmitMessage"/> </operation> </portType> <plnk:partnerLinkType
name="TimesheetSubmissionType"> <plnk:role name="TimesheetSubmissionService">
<plnk:portType name="tns:TimesheetSubmissionInterface"/> </plnk:role>
</plnk:partnerLinkType> </definitions>
```

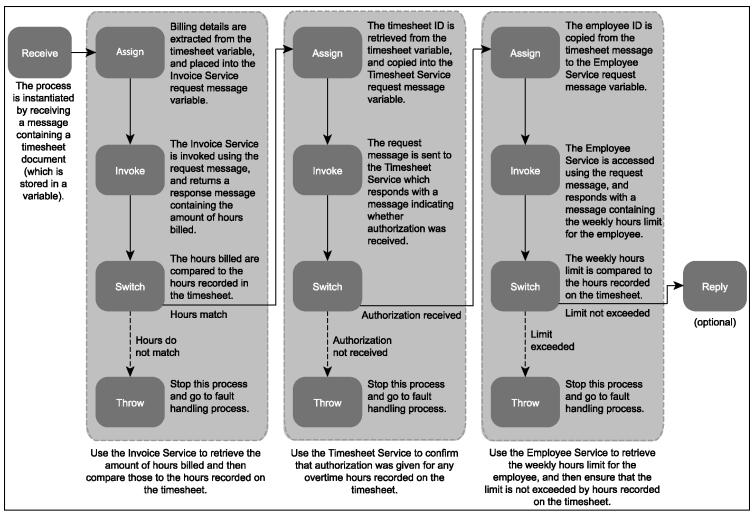


Figure 16.12: A descriptive, diagrammatic view of the process definition logic.

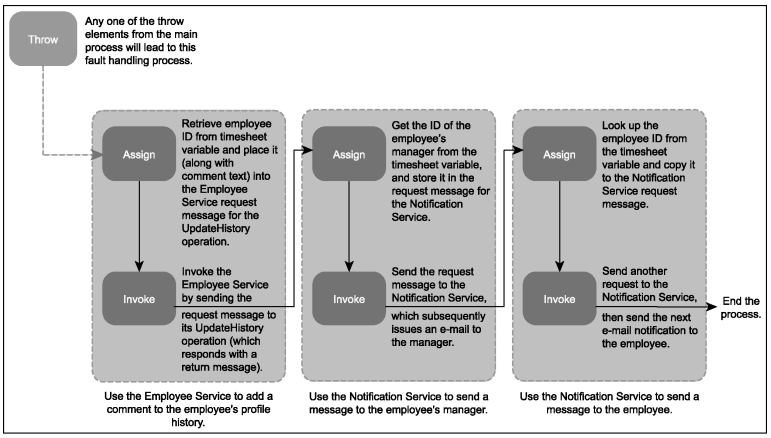


Figure 16.13:: A visual representation of the process logic within the faultHandlers construct.

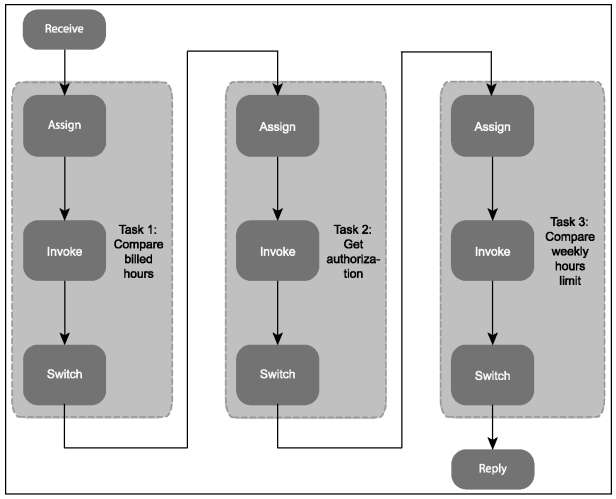


Figure 16.14: Sequential, synchronous execution of process activities.

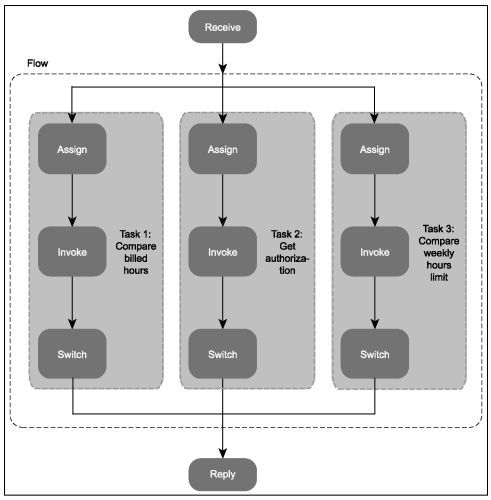


Figure 16.15: Concurrent execution of process activities using the flow construct.

Service-Oriented Architecture

Chapter 17

Fundamental WS-* Extensions

WS-Addressing language basics

WS-Addressing is a core WS-* extension providing features that can be used intrinsically or alongside features offered by other WS-* specifications.

- The *EndpointReference* element
  - This element is used by the From, ReplyTo, and FaultTo elements described in the *Message information header elements* section.

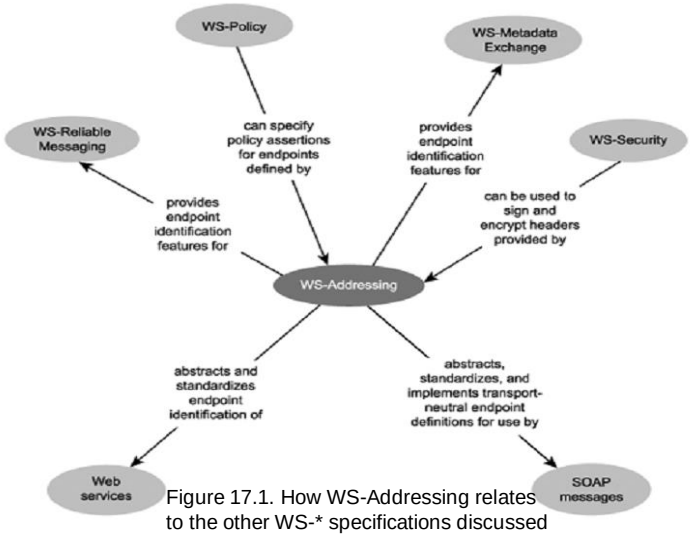


Figure 17.1. How WS-Addressing relates to the other WS-* specifications discussed in this chapter

WS-Addressing language basics

– WS-Addressing *endpoint* reference elements.

Elements	Description
Address	The standard WS-Addressing Address element used to provide the address of the service. This is the only required child element of the EndpointReference element.
ReferenceProperties	This construct can contain a series of child elements that provide details of properties associated with a service instance.
ReferenceParameters	Also a construct that can supply further child elements containing parameter values used for processing service instance exchanges.
PortType	The name of the service portType.
ServiceName and PortName	The names of the service and port elements that are part of the destination service WSDL definition construct.
Policy	This element can be used to establish related WS-Policy policy assertion information.

WS-Addressing language basics

- Message information header elements
  - This collection of elements can be used in various ways to assemble metadata-rich SOAP header blocks.
  - WS-Addressing message information header elements

WS-Addressing language basics

– WS-Addressing message information header elements

Elements	Description
MessageID	An element used to hold a unique message identifier, most likely for correlation purposes. This element is required if the ReplyTo or FaultTo elements are used.
RelatesTo	This is also a correction header element used to explicitly associate the current message with another. This element is required if the message is a reply to a request.
ReplyTo	The reply endpoint (of type EndpointReference) used to indicate which endpoint the recipient service should send a response to upon receiving the message. This element requires the use of MessageID.
From	The source endpoint element (of type EndpointReference) that conveys the source endpoint address of the message.
FaultTo	The fault endpoint element (also of type EndpointReference) that provides the address to which a fault notification should be sent. FaultTo also requires the use of MessageID.
To	The destination element used to establish the endpoint address to which the current message is being delivered.
Action	This element contains a URI value that represents an action to be performed when processing the MI header.

```
<Envelope
   xmlns="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:wsa=
      "http://schemas.xmlsoap.org/ws/2004/08/addressing"
   xmlns:app="http://www.xmltc.com/railco/...">
   <Header>
      <wsa:Action>
         http://www.xmltc.com/tls/vp/submit
      </wsa:Action>
      <wsa:To>
      </wsa:To>
      <wsa:From>
         <wsa:Address>
            http://www.xmltc.com/rail
         </wsa:Address>
         <wsa:ReferenceProperties>
            <app:id>
               unn:AFJK32311ws
            </app:id>
         </wsa:ReferenceProperties>
         <wsa:ReferenceParameters>
            <app:sesno>
               22322447
            </app:sesno>
         </wsa:ReferenceParameters>
      </wsa:From>
      <wsa:MessageID>
         uuid:243234234-43gf433
      </wsa:MessageID>
      <wsa:ReplyTo>
         <wsa:Address>
            http://www.xmltc.com/rail
         </wsa:Address>
         <wsa:ReferenceProperties>
            <app:id>
               unn:AFJK32311ws
            </app:id>
         </wsa:ReferenceProperties>
         <wsa:ReferenceParameters>
            <app:sesno>
               22322447
            </app:sesno>
         </wsa:ReferenceParameters>
      </wsa:ReplyTo>
      <wsa:FaultTo>
         <wsa:Address>
            http://www.xmltc.com/rail
         </wsa:Address>
         <wsa:ReferenceProperties>
            <app:id>
               unn:AFJK32311ws
            </app:id>
         </wsa:ReferenceProperties>
         <wsa:ReferenceParameters>
            <app:sesno>
               22322447
            </app:sesno>
         </wsa:ReferenceParameters>
      </wsa:FaultTo>
   </Header>
   <Body>
      ...
   </Body>
</Envelope>
```

Example 17.2. A SOAP header with WS-Addressing message information header elements, three of which contain Endpoint Reference elements.

WS-Addressing language basics

- WS-Addressing reusability
  - Established generic set of extensions useful to custom service-oriented solutions.
  - Also reusable by other WS-* specifications

WS-Addressing Summary

- WS-Addressing provides a collection of message header elements that can supplement a message with various (mostly routing-related) meta information.
- The WS-Addressing specification defines a set of reusable extensions that are becoming intrinsically commonplace among other WS-*specifications.
- It is worth remembering that some of the message information header elements established by WS-Addressing are of type EndpointReference and can therefore be contain a variety of endpoint metadata.

WS-ReliableMessaging language basics

WS-ReliableMessaging introduces critical quality of service features for the guaranteed delivery or failure notification of SOAP messages.

It also positions itself as a fundamental-* extension, as show in Figure 17.2

Key WS-ReliableMessaging language elements:

- *Sequence* element
- *MessageNumber* element
- *LastMessage* element
- *SequenceAcknowledgement* element
- *AcknowledgementRange* element
- *Nack* element
- *AckRequested* element

WS-ReliableMessaging language basics

WS-ReliableMessaging language basics

- The Sequence, MessageNumber, and LastMessage elements
  - *Sequence* construct resides in the SOAP message header to represent the location of the current message.
    - In relation to the overall sequence of messages within which it is being delivered.
    - *Sequence* relies on a set of child elements.
    - *Identifier* element is used to contain an ID value associated with the sequence itself

WS-ReliableMessaging language basics

  - *MessageNumber* element contains a number that is the position of the message within the overall sequence order.
  - *LastMessage* element can be added to the *Sequence* construct to communicate the fact that the current message is the final message of the sequence.

WS-ReliableMessaging language basics

- The SequenceAcknowledgement and AcknowledgementRange elements
  - *SequenceAcknowledgement* is issued by the recipient service upon the arrival of one or more messages with a sequence.
    - The SequenceAcknowledgement header construct to communicate that the original delivery was successful.
  - *AcknowledgementRange* element contains the Upper and Lower attributes that indicate a range of messages that were received.
    - The Range is based on the *MessageNumber* values of the message.
    - One A*cknowledgementRange* element communicates each consecutive set of messages received.
    - Sequence construct resides in the SOAP message header to represent the location of the current message.

WS-ReliableMessaging language basics

- The *Nack* element
  - Communicates the delivery failure of a message.
  - It's show which messages were not received.
- The *AckRequested* element
  - RM source service can request that the RM destination send out a sequence acknowledgement message on demand by using the *AckRequested* header construct.

WS-ReliableMessaging language basics

- Other WS-ReliableMessaging elements
  - Additional WS-ReliableMessaging elements

Elements	Description
SequenceRef	This construct allows you to attach policy assertions to a sequence, which introduces the ability to add various delivery rules, such as those expressed in the delivery assurances explained in Chapter 7.
AcknowledgementInterval	Specifies an interval period that an RM destination can use to automatically transmit acknowledgement messages.
BaseRetransmissionInterval	An interval periond used by the RM source to retransmit messages (for example, if no acknowledgements are received).
InactivitiyTimeout	A period of time that indicates at what point a sequence will time out and subsequently expire.
Expires	A specific date and time at which a sequence is scheduled to expire.
SequenceCreation	Sequences are genarally created by the RM Source, but the RM Destination my use this element to force the creation of its own sequence.

WS-ReliableMessaging Summary

- WS-ReliableMessaging introduces a set of elements that govern the processing of message sequences and subsequent delivery acknowledgments.
- WS-ReliableMessaging establishes the concept of a sequence, as implemented via the *Sequence* construct. This represents a group of messages to which additional delivery rules can be applied.
- WS-ReliableMessaging further introduces the concept of acknowledgements, as established by the *SequenceAckowledgement* and *Nack* elements, which provide an inherent mechanism for a guaranteed notification of successful and failed deliveries.

WS-Policy language basics

- The *WS-Policy* framework establishes a means of expressing service metadata beyond the WSDL definition
- The *WS-Policy* framework is comprised of the following three specifications:
  - *WS-Policy*
  - *WS-PolicyAssertions*
  - *WS-PolicyAttachments*

WS-Policy Relation to other WS-* Specifications

Policy Element

- Establishes the root construct to contain the various policy assertions that comprise the policy
- The *WS-PolicyAssertions* specification supplies the following set of common, predefined assertions elements:
  - **TextEncoding** – Dictates the use of a specific text encoding format

Policy Element cont…

- **Language** – Expresses the requirement or preference for a particular language
- **SpecVersion** – Communicates the need for a specific version of a specification
- **MessagePredicate** – Indicates message processing rules expressed using Xpath statements

ExactlyOne Element

- Surrounds multiple policy assertions and indicates that there is a choice between them, but that one must be chosen

All Element

- States that all of the policy assertions within the construct must be met

Usage Attribute

Attribute Value	Description
Required	The assertion requirements must be met, or an error will be generated.
Optional	The assertion requirements may be met, but an error will not be generated if they are not met.
Rejected	The assertion is unsupported.
Observed	The assertion applies to all policy subjects.
Ignored	The assertion will intentionally be ignored.

Preference Attribute

- Used to rank policy assertions in order of preference
- Assigned an integer value
- Higher value means a more preferred assertion
- Default value is “0”

PolicyReference Element

- A way to simply link an element with one or more policies
- Contains a URI attribute that points to one policy document or a specific policy assertion within the document
- Policy documents are merged at runtime when multiple *PolicyReference* elements are used within the same element

PolicyURIs Attribute

- Can be used to link to one or more policy documents
- Added to an element and can be assigned multiple policy locations
- Policies are merged at runtime
- Example:

<Employee wsp:**PolicyURIs**=
“http://www.xmltc.com/tls/policy1.xml#Emp1”
“http://www.xmltc.com/tls/policy2.xml#Emp2” />

PolicyAttachment Element

- Associates a policy with a subject
- The child *AppliesTo* construct is positioned as the parent of the subject elements
- The *PolicyReference* element the follows the *AppliesTo* construct to identify the policy assertions that will be used

Additional Types of Policy Assertions

- Policy assertions can be incorporated into WSDL definitions through the use of a special set of policy subjects that target specific parts of the definition structure
- A separate *UsingPolicy* element is provided for use as a WSDL extension
- *WS-ReliableMessaging* defines and relies on *WS-Policy* assertions to enforce some of its delivery and acknowledgement rules

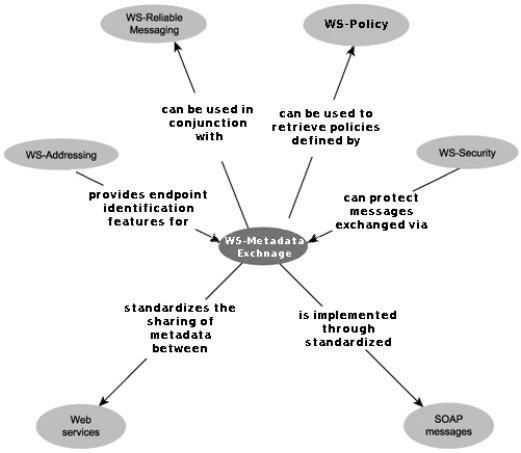
Additional Types cont…

- *WS-Policy* assertions can be created to communicate that a Web service is capable of participating in a business activity or an atomic transaction
- A policy assertion can be designed to express a service's processing requirements in relation to other WS-* specifications
- *WS-Policy* assertions commonly are utilized within the *WS-Security* framework to express security requirements

WS-MetadataExchange Language Basics

- Provides a standardized means by which service description documents can be requested and supplied.
- Supports interoperability and quality of service (QoS)
- Standardized metadata requests are:
  - GetMetadata
  - Get

WS-MetadataExchange Relation to other WS-* Specifications



GetMetadata Element

- Placed in the *Body* area of a SOAP message
- Can be turned into a construct that hosts child *Dialect* and *Identifier* elements

Dialect Element

- Specifies the type and version of the metadata specification requested
- Guarantees that the metadata returned to the service requesting it will be understood

Identifier Element

- Narrows the criteria by asking for a specific part of the metadata

Metadata, MetadataSection, and MetadataReference Elements

- Used to organize the content of the message sent in response to a GetMetadata request
- Metadata
  - construct resides in the SOAP message Body area
- MetadataSection
  - constructs that each represent a part of the returned metadata
  - Holds the contents of the metadata document returned

Metadata, MetadataSection, and MetadataReference Elements

- MetadataReference
  - Holds the pointer to the document if that is all that is returned

Get Message

- Used to explicitly request one of the documents from the location provided in the *MetadataReference* contstruct

WS-Security Language Basics

- Provides extensions that can be used to implement message-level security measures
- Designed to work with any of the WS-* specifications talked about so far
- Comprised of numerous specifications, many in different stages of acceptance and maturation

WS-Security Relation to other WS-* Specifications

Security Element

- Represents the fundamental header block provided by *WS-Security*
- Can have a variety of child elements
  - *XML-Encryption*
  - *XML-Signature*
  - *WS-Security* specification itself
- Can be outfitted with *actor* attributes that correspond to SOAP *actor* roles

UsernameToken, Username, and Password Elements

- Can be used to host token information for authentication and authorization purposes
- Custom elements can be added, but the *Username* and *Password* child elements are typical children to this construct

BinarySecurityToken Element

- Tokens stored as binary data
  - Example:
    - Certificates

SecurityTokenReference Element

- Allows a pointer to be provided to a token that exists outside of the SOAP message document

EncryptedData Element

- Parent construct that hosts the encrypted portion of an XML document
- The *Type* attribute indicates what is included in the encrypted content

CipherData, CipherValue, and CipherReference Elements

- *CipherData* construct is required and must contain either a *CipherValue* element hosting the characters representing the encrypted text or a *CipherReference* element that provides a pointer to the encrypted values

XML-Signature Elements

Element	Description
CanonicalizationMethod	This element identifies the type of "canonicalization algorithm" used to detect and represent subtle variances in the document content (such as the location of white space)
DigestMethod	Identifies the algorithm used to create the signature
DigestValue	Contains a value that represents the document being signed. This value is generated by applying the *DigestMethod* algorithm to the XML document
KeyInfo	This optional construct contains the public key information of the message sender
Signature	The root element, housing all of the information for the digital signature

XML-Signature Elements

Element	Description
SignatureMethod	The algorithm used to produce the digital signature. The digest and canonicalization algorithms are taken into account when creating the signature
SignatureValue	The actual value of the digital signature
SignedInfo	A construct that hosts elements with information relevant to the *SignatureValue* element, which resides outside of this construct.
Reference	Each document that is signed by the same digital signature is represented by a *Reference* construct that hosts digest and optional transformation details.

SOA

Chapter 18
SOA Platforms

SOA Platforms

- SOA platform basics
- SOA support in J2EE
- SOA support in .NET
- Integration considerations

SOA platform basics

- Basic platform building blocks
- Common SOA platform layers
- Relationship btw SOA layers and tech
- Fundamental service tech architecture
- Vendor platforms

Basic platform building blocks

- Development environment
- Runtime
- APIs
- Operating system

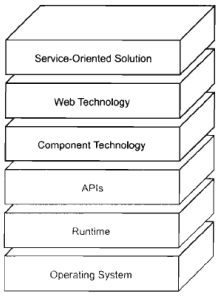


Figure 18.2
The common layers required by a development and runtime platform for building SOA.

Common SOA platform layers

- Self-contained and composable units
- Encapsulate and expose application logic

Relat. btw SOA layers & tech

- Web technology layer
  - Support for first gen Web services
  - Support for WS-* specs
  - Able to make Web services
- Component technology layer
  - Support encapsulation

Relat. btw SOA layers & tech (cont.)

- Runtime layer
  - Capable of hosting components and Web services
  - Provide APIs
- APIs layer
  - Support development and processing of components and Web services

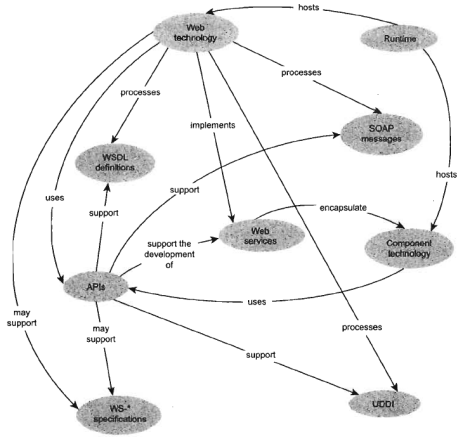


Figure 18.3
A logical view of the basic relationships between the core parts of a service-oriented architecture.

Fund. Service tech architecture

- Service processing task
- Service processing logic
  - Message processing logic
  - Business logic
- Service agents

Service processing task

- Send/Receive SOAP messages

Service processing logic

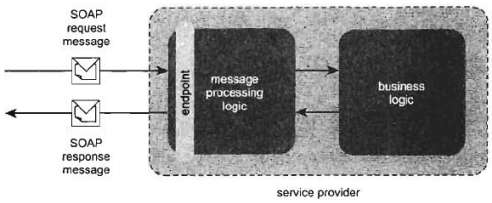


Figure 18.5
A revised service provider model now including an endpoint within the message processing logic.

Message processing logic

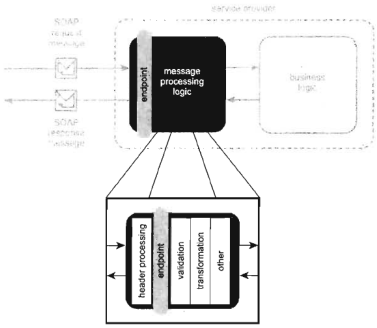


Figure 18.7
An example of the types of processing functions that can comprise the message processing logic of a service.

Business processing logic

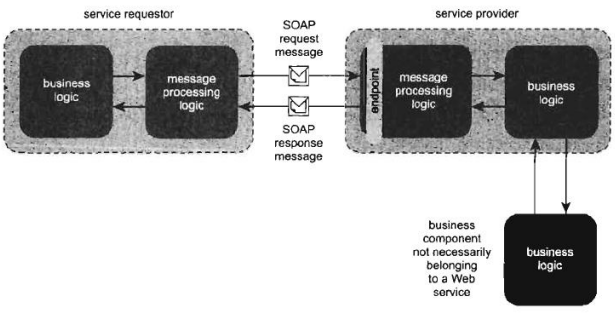


Figure 18.11
The same unit of business logic facilitating a service provider and acting on its own by communicating independently with a separate component.

Service agents

- SOAP header processing
- Filtering
- Authentication and content-based validation
- Logging and auditing
- Routing

Vendor platforms

- Architecture components
- Runtime environment
- Programming languages
- APIs
- Service providers
- Service requestors
- Service agents
- Platform extensions

SOA support in J2EE

- Platform overview
- Primitive SOA support
- Support for service-orientation principles
- Contemporary SOA support

Platform overview

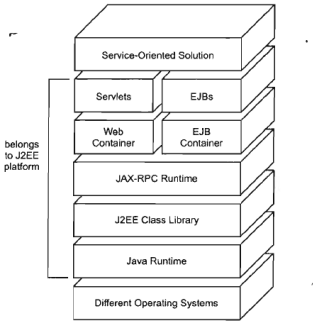


Figure 18.13
Relevant layers of the J2EE platform as they relate to SOA.

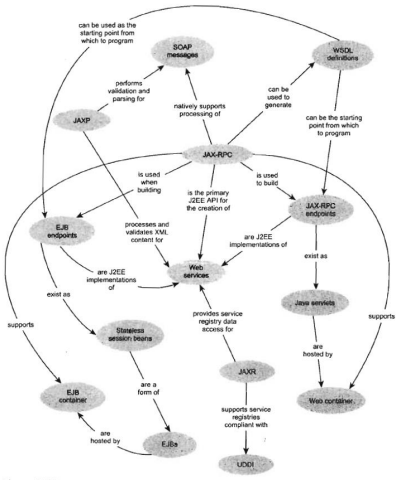


Figure 18.14
How parts of the J2EE platform inter-relate.

Key Java specifications

- Java 2 Platform Enterprise Edition Spec
- Java API for XML-based RPC (JAX-RPC)
- Web Services for J2EE

Architecture components

- Java Server Pages (JSPs)
- Struts
- Java Servlets
- Enterprise JavaBeans (EJBs)

Runtime environments

- EJB container
- Web container

Programming languages

- Java
- Various development tools

APIs

- Java API for XML Processing (JAXP)
- Java API for XML-based RPC (JAX-RPC)

Service providers

- JAX-RPC Service Endpoint
- EJB Service Endpoint
- Port Component Model
  - Service Endpoint Interface (SEI)
  - Service Implementation Bean

Service requestors

- Generated stub
- Dynamic proxy and dynamic invocation interface

Service agents

- Tasks
- Handlers

Platform extensions

- IBM Emerging Technologies Toolkit
- Java Web Services Developer Pack

Primitive SOA support

- Service encapsulation
- Loose coupling
- Messaging

Support for service-orientation principles

- Autonomy
- Reusability
- Statelessness
- Discoverability

Contemporary SOA support

- Based on open standards
- Supports vendor diversity
- Intrinsically interoperable
- Promotes federation
- Architecturally composable
- Extensibility
- Supports service-oriented business modeling
- Logic-level abstraction
- Organizational agility and enterprise-wide loose coupling

SOA support in .NET

- Platform overview
- Primitive SOA support
- Support for service-orientation principles
- Contemporary SOA support

Platform overview

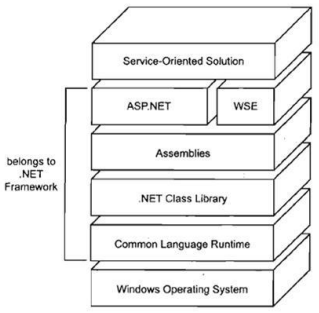


Figure 18.18
Relevant layers of the .NET framework, as they relate to SOA.

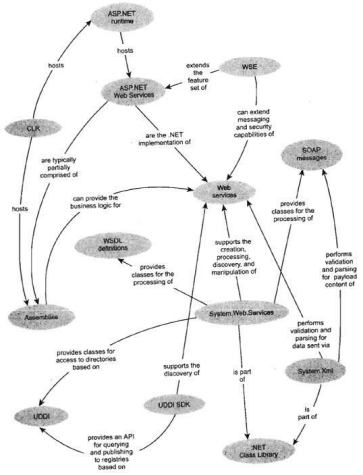


Figure 18.19
How parts of the .NET framework inter-relate.

Architecture components

- ASP.NET Web Forms
- ASP.NET Web Services
- Assemblies

Runtime environments

- Common Language Runtime (CLR)
- HTTP Pipeline

Programming languages

- Visual Basic, C++, C#
- Microsoft Intermediate Language (MSIL)

APIs

- System.Xml
- System.Web.Services
- As well as:
  - System.Xml.Xsl
  - System.Xml.Schema
  - System.Web.Services.Discovery

Service providers

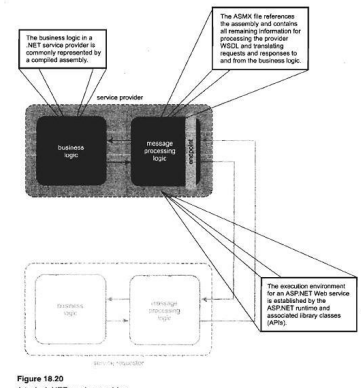


Figure 18.20
A typical .NET service provider.

Service requestors

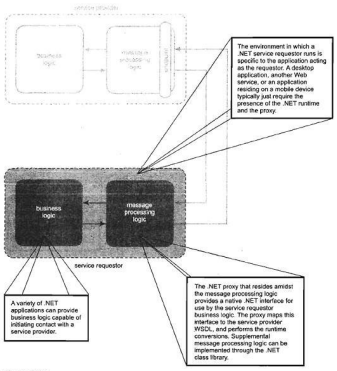


Figure 18.21
A typical .NET service requestor.

Service agents

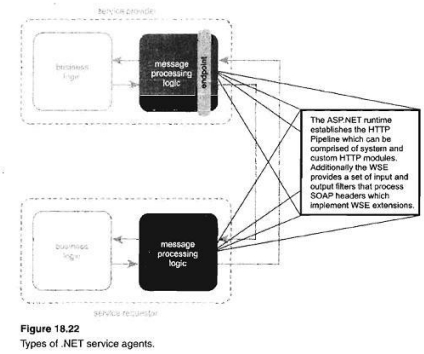


Figure 18.22
Types of .NET service agents.

Platform extensions

- Web Services Enhancements

Primitive SOA support

- Service encapsulation
- Loose coupling
- Messaging

Support for service-orientation principles

- Autonomy
- Reusability
- Statelessness
- Discoverability

Contemporary SOA support

- Based on open standards
- Supports vendor diversity
- Intrinsically interoperable
- Promotes federation
- Architecturally composable
- Extensibility
- Supports service-oriented business modeling
- Logic-level abstraction
- Organizational agility and enterprise-wide loose coupling

Integration considerations

- Reasons for new requirements
- Cross-platform interoperability
- Changes to cross-platform interoperability requirements
- Application logic abstraction