UNIT - III

3.1. SOA delivery lifecycle phases

The lifecycle of an SOA delivery project is simply comprised of a series of steps that need to be completed to construct the services for a given service-oriented solution.

3.1.1. Basic phases of the SOA delivery lifecycle

Development projects for service-oriented solutions are, on the surface, much like other custom development projects for distributed applications. Web services are designed, developed, and deployed alongside standard components and the usual supporting cast of front- and back-end technologies.

Fig 3.1. Common phases of an SOA delivery lifecycle.

3.1.2. 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.

3.1.3. Service-oriented design

It is a design phase where we know how the web service 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.

3.1.4. Service development

This is an 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.

3.1.5. 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:

- What types of service requestors could potentially access a service?
- Can all service policy assertions be successfully met?
- What types of exception conditions could a service be potentially subjected to?
- How well do service descriptions communicate service semantics?
- Do revised service descriptions alter or extend previous versions?
- 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?
- Have all compensation processes been fully tested?
- What happens if exceptions occur within compensation processes?
- 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?

3.1.6. 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:
• How will services be distributed?
• Is the infrastructure adequate to fulfill the processing requirements of all services?
• How will the introduction of new services affect existing services and applications?
• How should services used by multiple solutions be positioned and deployed?
• How will the introduction of any required middleware affect the existing environment?
• Do these services introduce new versions of service descriptions that will need to be deployed alongside existing versions?
• What security settings and accounts are required?
• How will service pools be maintained to accommodate planned or unforeseen scalability requirements?
• How will encapsulated legacy systems with performance or reliability limitations be maintained and monitored?

3.1.7. 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:
• How will service usage be monitored?
• What form of version control will be used to manage service description documents?
• How will messages be traced and managed?
• How will performance bottlenecks be detected?

3.2. 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

Three common strategies have emerged, each addressing this problem in a different manner.
• top-down
3.2.1 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.

3.2.1.1. Process

The top-down approach will typically contain some business requirements have already been collected and defined.

**Fig 3.2. Common top-down strategy process steps.**

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
<th>Step 5</th>
<th>Step 6</th>
<th>Step 7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Define relevant enterprise-wide ontology</strong></td>
<td><strong>Align relevant business models</strong></td>
<td><strong>Perform service-oriented analysis</strong></td>
<td><strong>Perform service-oriented design</strong></td>
<td><strong>Develop services</strong></td>
<td><strong>Test service operations</strong></td>
<td><strong>Deploy services</strong></td>
</tr>
</tbody>
</table>

**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. If such a business vocabulary does not yet exist for whatever
information sets a solution is required to work with, then this step requires that it be defined. A significant amount of up-front information gathering and high-level business analysis effort may therefore be required.

**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.

**Step 3: Perform service-oriented analysis**

A service-oriented analysis phase is the initial stage for SOA.

**Step 4: Perform service-oriented design**

The service layers are formally defined as part of a service-oriented design process.

**Step 5: Develop the required services**

Services are developed according to their respective design specifications and the service descriptions created in Step 4.

**Step 6: Test the services and all service operations**

The testing stage requires that all service operations undergo necessary quality assurance checks. This typically exceeds the amount of testing required for the automation logic being implemented because reusable services will likely need to be subjected to testing beyond the immediate scope of the solution.

**Step 7: Deploy the services**

The solution is finally deployed into production. An implementation consideration beyond those we originally identified as part of this step is the future reuse potential of the service. To facilitate multiple service requestors, highly reusable services may require extra processing power and may have special security and accessibility requirements that will need to be accommodated.

3.2.1.2. 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.

3.2.2. 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.

3.2.2.1. Process

A typical bottom-up approach follows a process where business requirements have already been collected and defined.

**Fig 3.3. Common bottom-up strategy process steps.**

1. **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.

**Step 3: Develop the required application services**

Application services are developed according to their respective service descriptions and applicable design specifications.

**Step 4: Test the services**

Services, their associated solution environment, and underlying legacy logic are tested to ensure that processing requirements can be met. Performance and stress testing measures often are used to set the processing parameters of legacy systems exposed via wrapper services. Security testing is also an important part of this stage.

**Step 5: Deploy the services**

The solution and its application services are deployed into production. Implementation considerations for application services frequently include performance and security requirements.

**3.2.2.2. 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. Although the bottom-up design allows for the efficient creation of Web services as required by applications, implementing a proper SOA at a later point can result in a great deal of retro-fitting or even the introduction of new
standardized service layers positioned over the top of the non-standardized services produced by this approach.

3.2.3. The agile strategy

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.

3.2.3.1. Process

The process steps demonstrate an example of how an agile strategy can be used to reach the respective goals of the top-down and bottom-up approaches.

Fig 3.4. A sample agile strategy process.
Step 1: Begin the top-down analysis, focusing first on key parts of the ontology and related business entities

The standard top-down analysis begins but with a narrower focus. The parts of the business models directly related to the business logic being automated receive immediate priority.

Step 2: When the top-down analysis has sufficiently progressed, perform service-oriented analysis

While Step 1 is still in progress, this step initiates a service-oriented analysis phase. Depending on the magnitude of analysis required to complete Step 1, it is advisable to give that step a good head start. After the top-down analysis has sufficiently progressed, model business services to best represent the business model with whatever analysis results are available. This is a key decision point in this process.

Step 3: Perform service-oriented design

The chosen service layers are defined, and individual services are designed as part of a service-oriented design process.

Steps 4, 5, and 6: Develop, test, and deploy the services

Develop the services and submit them to the standard testing and deployment procedures.

Step 7: As the top-down analysis continues to progress, revisit business services

Perform periodic reviews of all business services to compare their design against the current state of the business models. Make a note of discrepancies and schedule a redesign for those services most out of alignment. This typically will require an extension to an existing service for it to better provide the full range of required capabilities. When redesigned, a service will need to again undergo standard development, testing, and deployment steps. To preserve the integrity of services produced by this approach, the concept of immutable service contracts needs to be strictly enforced. After a contract is published, it cannot be altered. Unless revisions to services result in extensions that impose no restrictions on an existing contract (such as the addition of new operations to a WSDL definition), Step 7 of this process likely will result in the need to publish new contract versions and the requirement for a version management system.

3.2.3.2. 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 retasking 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.

3.3. 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.

3.3.1. 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 extent to which these questions are answered is directly related to the amount of effort invested in the analysis. 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.

3.3.2. The service-oriented analysis process

Service-oriented analysis can be applied at different levels, depending on which of the SOA delivery strategies are used to produce services. The strategy will determine the layers of abstraction that comprise the service layers of a solution environment. From an analysis perspective, each layer has different modeling requirements. 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?

**Fig 3.5. A high-level service-oriented analysis process.**

![Service-oriented analysis process diagram]

### 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. This business process documentation will be used as the starting point of the service modeling process described in Step 3.

**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.

**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.

**Case Study**

The RailCo's ultimate goal is to standardize on SOA to solve its current automation problems and increase its online clientele. Here is the familiar list of RailCo services:

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

Construction of these services was based purely on a bottom-up approach, resulting in the creation of "hybrid" application services (application services that also contain business logic, intended for narrowly focused business tasks). These services were assembled quickly to accommodate a single client. These benefits appear to provide the potential to fulfill key business goals of the organization. Specifically, preliminary analysis results indicate that:

- The application of service-orientation principles and the standardization introduced by SOA will allow a single set of services to accommodate interaction with different online partners. This would free RailCo from its ties to TLS and allow it to pursue new customers without necessarily having to build a new set of services each time.
- SOA facilitates the integration of internal legacy systems by establishing standardized application endpoints. In particular, the abstraction provided by service layers will enable RailCo to create integration channels that do not need to be removed if underlying legacy technology is replaced.
3.3.3. Deriving business services

There are set of approaches, some of which have been more accepted than others. Perhaps there should be no single approach to deriving services. Organizations employ different methodologies, business entity relationships, and vocabularies, resulting in vastly diverging business model structures. Further, there are cultural preferences and vendor platform influences that result in expression of the business models through different sets of modeling tools and languages.

3.3.3.1. Sources from which business services can be derived

The inner workings 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 logical unit of work, and pretty much anything any organization does consists of units of work. What differs, though, is how organizations structure and document the work they perform.

Business Process Management (BPM) models

The advent of BPM has resulted in an industry-wide flurry of process modeling and remodeling activity. Process models have therefore become a central form of business analysis documentation in many organizations. Business services can be derived from process workflow logic. Deriving a business service from a business process requires a thorough knowledge of the underlying workflow logic. This is because defining the scope of the business logic to be represented is a judgment call that can have significant implications when the business service is implemented as part of solution environments; hence, the better the judgment, the better the quality of the service. And, of course, the better quality services you end up with, the better quality your service-oriented environment will be.

Case Study

Following are descriptions of two existing RailCo processes that are currently being partially automated by its services. The descriptions provided here document the steps that comprise internal RailCo processes because these are the processes that eventually will be affected by the introduction of SOA. First, we’ll explore the existing Invoice Submission Process. It consists of a series of steps that describe how invoices are generated specifically for TLS, as follows:

1. Accounting clerk creates and issues an electronic invoice using the legacy accounting system.
2. The save event triggers a custom script that exports an electronic copy of the invoice to a network folder.
3. A custom developed component, which polls this folder at ten-minute intervals, picks up the document and transforms it into an XML document.
4. The invoice XML document is then validated. If it is deemed valid, it is forwarded to the Invoice Submission Service. If validation fails, the document is rejected, and the process ends.
5. Depending on when the last metadata check was performed, the service may issue a Get Metadata request to the TLS B2B solution.
6. If the Get Metadata request is issued and if it determines that no changes were made to the relevant TLS service descriptions, the Invoice Submission Service transmits the invoice document to the TLS B2B solution using the ExactlyOnce delivery assurance. If the Get Metadata request identifies a change to the TLS service descriptions, the invoice is not submitted, and the process ends.

The internal Order Fulfillment Process is similar in that it establishes the same type of relationship between the accounting system and a Web service only this time, the data flow is reversed, as described here:
1. The RailCo Order Fulfillment Service receives a SOAP message from TLS, containing a payload consisting of a TLS purchase order document.
2. The service validates the incoming document. If valid, the document is passed to a custom component. If the TLS PO fails validation, a rejection notification message is sent to TLS, and the process ends.
3. The component has the XML document transformed into a purchase order that conforms to the accounting system's native document format.
4. The PO then is submitted to the accounting system using its import extension.
5. The PO ends up in the work queue of an accounting clerk who then processes the document.

3.3.3.2. Entity models

Primary entities represent the main business documents and transaction areas of an enterprise. For example, Invoice, Purchase Order, Customer, and Claim are all milestone entities within different types of businesses. Further, organizations model entities according to proprietary rules and business policies. This results in entities having relationships with each other. Entity-centric services mirror the entity model by containing a set of generic operations that facilitate the different types of functions associated to the processing of the entity. Communication between different entity-centric services also can be governed by constraints relating to the inherent relationship between entities.

Case Study

RailCo has decided to build services centered around specific tasks and is therefore not using entity models as a source for deriving business services. However, to demonstrate the concept of
entities, let's briefly look at what entities exist within the RailCo business areas we've been exploring so far. Based on the business processes, RailCo entities of relevance are:

- Invoice
- Purchase Order

Additional entities involved with these processes could include:

- Employee
- Order
- Back Order
- Customer

3.3.3.4. Types of derived business services

Deriving services from the two sources we just identified results in the creation of distinct types of 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. Typical examples of task-centric business services are:

- VerifyInvoice
- GetHistoryReport

Each of these services contains operations that relate to a particular task within the context of a process. Task-centric services usually result from modeling exercises that are focused on meeting immediate business requirements.

**Case Study**

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. Their inherent generic nature makes them highly reusable by numerous business processes. Even though entity-centric
business services often are built as part of application development projects centered around a particular business process, they differ from task-centric services in that they do not provide an interface specific to that process. Instead, the source of inspiration for these types of services is entity models. When compared to task-centric services, entity-centric services significantly increase the agility with which service-oriented processes can be remodeled. This is because task-centric services often are built to help automate one business process and can therefore get tied to that process.

**Case Study**

Although we have identified logical entities within RailCo, no entity-centric business services currently exist. Having followed the top-down SOA delivery strategy, though, TLS already has a set of entity-centric business services in place.

**Fig 3.5 Different organizations, different approaches to building services.**

These are:

- Accounts Payable Service
- Purchase Order Service
- Ledger Service
- Vendor Profile Service
3.4 Introduction to service-oriented design

Service-oriented design is the process by which concrete physical service designs are derived from logical service candidates and then assembled into abstract compositions that implement a business process.

3.4.1. Objectives of service-oriented design

The primary questions answered by this phase are:

- How can physical service interface definitions be derived from the service candidates modeled during the service-oriented analysis phase?
- What SOA characteristics do we want to realize and support?
- What industry standards and extensions will be required by our SOA to implement the planned service designs and SOA characteristics?

To address these questions, the design process actually involves further analysis. This time our focus is on environmental factors and design standards that will shape our services. 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.

3.4.2. "Design standards" versus "Industry standards"

The term "standards" is used frequently in this chapter. It is easy to confuse its context, so we often qualify it. 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.

3.5. 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. Although most SOAs will implement a common set of shared technologies based on key XML and first-generation Web services specifications, the modular nature of the WS-* specification landscape allows for extensions to this core architecture to be added as required. This step consists of the following three further steps.

1. Choose service layers
2. Position core SOA standards.
3. Choose SOA extensions.
These steps are represented by the following three separate processes:

- Entity-centric business service design process.
- Application service design process.
- Task-centric business service design process.

**Step 5: Design service-oriented business process**

Upon establishing an inventory of service designs, we proceed to create our orchestration layer—the glue that binds our services with business process logic. This step results in the formal, executable definition of workflow logic, which translates into the creation of a WS-BPEL process definition.

**3.5.1. Specification associated with service design**

![Diagram: Three core specifications associated with service design.](image)

**3.5.1.1 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. 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. Therefore, many environments must go through a mapping process to select the XSD schema types best suited to represent data originating from or being delivered to proprietary application logic and data sources.
"Elements" vs. "Constructs"

Each of the specifications we explore in this and subsequent chapters provides a markup language that is expressed as an XML dialect. This means that the language itself is written in XML and is comprised of XML elements. Our focus is on describing key elements that provide features relevant to the topics discussed in this book. Sometimes we refer to language elements as 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.

Example 3.1. The most basic form of schema declaration.

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema">
Other important attributes include targetNamespace, used to assign a namespace to the custom elements and attributes declared in the schema document, and the element-FormDefault attribute, which when set to a value of "qualified," requires that all elements in the XML document be associated
with their corresponding namespace.

The **element** element

Using this element, you can declare a custom element that is then referenced by its name within an XML document instance.

**Example 3.2. An element declaration in an XSD schema.**

```
<element name="InvoiceNumber" type="xsd:integer"/>
```

**Example 3.3. The usage of this element in an XML document instance.**

```
<InvoiceNumber>12345</InvoiceNumber>
```

...where the value in between the opening and closing InvoiceNumber tags is required to be an integer.

The **type** attribute of an element can be set to one of the predefined data types established by the XML Schema specification, or it can be assigned a complex type, as explained next.

The **complexType** and **simpleType** elements

With a **complexType** you can group elements and attributes into a composite type that can be used to represent a set of related data representations. The following example groups two elements named ID and WeeklyHoursLimit into a **complexType** named EmployeeHours.

**Example 3.4. A complexType containing two element declarations.**

```
<complexType name="EmployeeHours">
  <sequence>
    <element name="ID" type="xsd:integer"/>
    <element name="WeeklyHoursLimit" type="xsd:short"/>
  </sequence>
</complexType>
```

The EmployeeHours complexType can be assigned to one or more elements. This facilitates standardization and reuse of commonly grouped information and avoids redundant element declarations.

The **import** and **include** elements

XSD schemas can be modularized. This allows for one schema document to import the contents of another. Both the import and **include** elements are used to point to the location of the XSD schema file that will be pulled in when the schema is processed at runtime.

**Example 3.5. The import and include elements.**

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" ...
```
The difference between these two elements is that **include** is used to reference schemas that use the same target namespace as the parent schema, whereas **import** is used to point to schemas that use a different target namespace.

**Other important elements**

The XML Schema Definition Language is large and complex and provides numerous options for structuring and validating XML document data. There are many other important parts of the language that are not used in the examples provided in this book, including:

- additional type definition elements (**attribute**, **complexContent**, **simpleContent**)
- constraint related elements (**restriction**, **enumeration**, **pattern**)
- element indicators (**maxOccurs**, **minOccurs**, **group**)
- extensibility elements (**any**, **extension**, **redefine**)
- elements for simulating relationships between elements (**unique**, **key**, **keyref**)

**3.5.1.2. WSDL language basics**

The Web Services Description Language (WSDL) is the most fundamental technology standard associated with the design of services. The WSDL document consists of separate abstract and concrete definitions. The abstract definition contains a series of parts that include types, message, and port type (or interface), whereas the concrete definition is comprised of binding and service parts.

![Fig 3.9. The structure of a WSDL definition.](image)
The **definitions** element

This is the root or parent element of every WSDL document. It houses all other parts of the service definition and is also the location in which the many namespaces used within WSDL documents are established.

**Example 3.6** A **definitions** element of the Employee Service, declaring a number of namespaces.

```
<definitions name="Employee"
    targetNamespace="http://www.xmltc.com/tls/employee/wsd/"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    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/wsd/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
</definitions>
```

The service definition is started with a **definitions** element that contains a series of attributes in which the service is assigned the name of "Employee" and in which a number of namespaces are declared.

**The types element**

The **types** construct is where XSD schema content is placed. This part of the WSDL can consist of actual XSD schema markup (an entire schema construct containing type definitions), or it can contain **import** elements that reference external schema definitions (or it can contain both embedded and imported XSD types). The WSDL definition are used to represent the XML content of message bodies. The **message** element references these types and associates them with messages. The SOAP message body contains XML content that can represent anything from simple parameter data to complex business documents. This content can be formally defined through types provided by the WSDL **types** area. Therefore, XSD schema **complexType** elements are commonly provided here, as they consist of groups of related types that can represent entire message body structures. In the following example, an entire **schema** construct is embedded within the WSDL **types** construct.
Fig 3.10. The WSDL types construct populated by XSD schema types used by the message construct to represent the SOAP message body.

Example 3.7. A types construct containing an XSD schema construct in which a complexType is defined.

```xml
<types>
  <schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace= "http://www.xmltc.com/railco/transform/schema/"
>
    <complexType name="ReturnCodeType">
      <sequence>
        <element name="Code" type="xsd:integer"/>
        <element name="Message" type="xsd:string"/>
      </sequence>
    </complexType>
  </schema>
</types>
```

Use of the types construct is common in WSDL definitions with substantial content. However, it is not a required element. Native XSD schema types can be referenced directly within the message
element, as explained next.

**The message and part elements**

For every message a service is designed to receive or transmit, a message construct must be added. This element assigns the message a name and contains one or more part child elements that each are assigned a type. message elements later are associated to operation elements to establish the input and output messages of the operation. part elements use the type or element attributes to identify the data type of the message part.

**Example 3.8. Two message constructs likely representing the input and output messages for an operation.**

```xml
<message name="getEmployeeWeeklyHoursRequestMessage">
  <part name="RequestParameter" element="act:EmployeeHoursRequestType"/>
</message>
<message name="getEmployeeWeeklyHoursResponseMessage">
  <part name="ResponseParameter" element="act:EmployeeHoursResponseType"/>
</message>
```

**Example 3.9. A simple parameter message requiring just a single integer value.**

```xml
<message name="getID">
  <part type="xsd:integer"/>
</message>
```

**The portType, interface, and operation elements**

Service operations are defined within the portType area of the WSDL definition. Hence, portType constructs simply represent collections of operations. Individual operations are defined using the aptly named operation element.

**Example 3.10. The portType construct hosting two operation constructs.**

```xml
<portType name="EmployeeInterface">
  <operation name="GetWeeklyHoursLimit">
    ...
  </operation>
  <operation name="UpdateHistory">
    ...
  </operation>
</portType>
```
The input and output elements (when used with operation)

Each operation construct contains input and/or output child elements that represent the request and response messages the operation is capable of processing.

Example 3.11. operation elements with child input and output elements.

```xml
<operation name="GetWeeklyHoursLimit">
  <input message="tns:getWeeklyHoursRequestMessage"/>
  <output message="tns:getWeeklyHoursResponseMessage"/>
</operation>
<operation name="UpdateHistory">
  <input message="tns:updateHistoryRequestMessage"/>
  <output message="tns:updateHistoryResponseMessage"/>
</operation>
```

WSDL supports predefined message exchange patterns (MEPs). The presence of input and output elements and the sequence in which they are displayed generally establishes the MEP of the operation.


```xml
<operation name="Submit">
  <input message="tns:receiveSubmitMessage"/>
</operation>
```

The binding element

The binding element begins the concrete portion of the service definition, to assign a communications protocol that can be used to access and interact with the WSDL. Upon first glance of the following example, the binding element appears similar in structure to the portType element. As with portType, the binding construct contains one or more operation elements. However, you'll notice the additional soap:binding and soap:operation elements interspersed within the construct syntax. These are what establish the SOAP protocol as the manner in which this WSDL can be communicated with.

Example 3.13. The binding construct hosting concrete operation definitions.

```xml
<binding name="EmployeeBinding" type="tns:EmployeeInterface">
```
<soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="GetWeeklyHoursLimit">
    <soap:operation soapAction="..."/>
    ...
</operation>
<operation name="UpdateHistory">
    <soap:operation soapAction="..."/>
    ...
</operation>
</binding>

Further, the style attribute of the soap:binding element defines whether the SOAP messages used to support an operation are to be formatted as document or RPC-style messages.

The input and output elements (when used with binding)

Each operation element within a binding construct mirrors the input and output message child elements defined in the abstract definition. Within a binding construct, however, the input and output elements do not reference the message elements again.

Example 3.14. input and output elements providing message processing information.

<operation name="GetWeeklyHoursLimit">
    <soap:operation soapAction="..."/>
    <input>
        <soap:body use="literal"/>
    </input>
</operation>
<operation>
    <output>
        <soap:body use="literal"/>
    </output>
</operation>
<operation name="UpdateHistory">
    <soap:operation soapAction="..."/>
    <input>
        <soap:body use="literal"/>
    </input>
</operation>
The service, port, and endpoint elements

The service element simply provides a physical address at which the service can be accessed. It hosts the port element that contains this location information.

Example 3.15. The service and port elements establishing the physical service address.

```xml
<service name="EmployeeService">
  <port binding="tns:EmployeeBinding" name="EmployeePort">
    <soap:address location="http://www.xmltc.com/tls/employee/"/>
  </port>
</service>
```

The import element

WSDL definitions support a similar form of modularity as XSD schemas do. The import element can be used to import parts of the WSDL definition as well as XSD schemas.

Example 3.16. The import element referencing a schema document.

```xml
<import namespace="http://www.xmltc.com/tls/schemas/"
        location="http://www.xmltc.com/tls/schemas/employee.xsd"/>
```

The documentation element

This optional element simply allows you to add descriptive, human-readable annotations within a WSDL definition. Developers can use this information when building service requestors or it can be programmatically retrieved through a service registry to aid the discovery of the service.

Example 3.17. The documentation element providing a description of the overall service interface.

```xml
<portType name="TransformInterface">
  <documentation>
    Retrieves an XML document and converts it into the native accounting document format.
  </documentation>
</portType>
```
3.5.1.3. SOAP language basics

Within the service-oriented design process, we place a great deal of emphasis on hand crafting the WSDL definition, along with required XSD schema types. SOAP messages generally do not require as much hands on attention. The structure of SOAP messages is relatively simple. They consist of header, body, and fault sections, all encased in an envelope.

Fig 3.11. The structure of a SOAP message document.

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 3.18. The root Envelope construct hosting Header and Body constructs.**

```xml
<Envelope xmlns ="http://schemas.xmlsoap.org/soap/envelope/">
  <Header>
    ...
  </Header>

  <Body>
    ...
    <Fault>
      ...
    </Fault>
  </Body>
</Envelope>
```
The header portion of the SOAP message has become a key enabler of the feature set provided by WS-* specifications. Most of these extensions are implemented on a message level and introduce new standardized SOAP header blocks destined to be embedded in the Header construct.

Example 3.19. The Header construct hosting a header block.

```xml
.setHeader
  <x:CorrelationID xmlns:x="http://www.xmltc.com/tls/headersample/" mustUnderstand="1">
    0131858580-JDJ903KD
  </x:CorrelationID>
</Header>
```

The Body element

This is the one required child element of the SOAP Envelope construct. It contains the message payload formatted as well-formed XML. The structure and naming used to define this part of the SOAP message relates to the style and use attributes discussed in the previous WSDL binding element description. SOAP message Body constructs are defined within the WSDL message constructs.

Fig 3.12. 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.
Example 3.20. The contents of a sample **Body** construct.

```xml
<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>
```

While SOAP header blocks can be processed actively during the transmission of a SOAP message, the SOAP body should not be touched. However, if allowed, intermediary services can still read and derive information from body content.

**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 faultcode element contains one of a set of fault conditions predefined by the SOAP specification. Both the **faultstring** and **detail** elements provide human readable error messages, the latter of which can host an entire XML fragment containing further partitioned error details.

**Example 3.21. The Fault construct residing within the Body construct.**

```xml
<Fault>
  <faultcode>
    MustUnderstand
  </faultcode>
  <faultstring>
    header was not recognized
  </faultstring>
</Fault>
```
<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>

3.5.2. **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.

**Fig.3.13 Entity-centric services establish the business service layer.**

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.
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.

**Fig 3.14. 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. The result of the TLS service modeling process was to express these two functions through the assignment of the following two operation candidates:

- get weekly hours limit
• update employee history

This service candidate now provides us with a primary input from which we derive a service design by following the steps in the following entity-centric business service design process.

Fig.3.15. 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.

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.

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. At first, this appears to make this step rather simple.

Example. 3.22. The Employee schema providing complexType constructs used to establish the data representation anticipated for the "Get weekly hours limit" operation candidate.

```xml
  <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>
```

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.
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 message constructs 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.

Fig 3.16. The Employee Service operations.

They subsequently proceed to define the remaining parts of the abstract definition, namely the message, and portType constructs.

Example 3.23 The message and portType parts of the Employee Service definition that implement the abstract definition details of the two service operations.

```xml
<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">
   
</message>
```

M.Senthil Kumar, Asst. Prof. Department of Information Technology, SMEC
<part name="ResponseParameter" element="hr:EmployeeUpdateHistoryResponseType"/>
</message>

<brType 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>
</brType>

Step 4: Apply principles of service-orientation
Here's where we revisit the four service-orientation principles which, being those not provided by the Web services technology set:

- 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. Discoverability is an important part of both the design of entity-centric services and their post-deployment utilization.

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.

Example.3.24 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.
- 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.
- 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.

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.
Fig.3.17 The revised Employee Service operation names.

Example 3.25. The two operation constructs with new, standardized names.

```xml
<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

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

Adding operations is a straight-forward means of providing evident functions associated with the entity. The classic set of operations for an entity-centric service is:

- GetSomething
- UpdateSomething
• AddSomething
• DeleteSomething

Security requirements notwithstanding, establishing these standard operations builds a consistent level of interoperability into the business service layer, facilitating ad-hoc reusability and composition.

**Case Study**

TLS is under time pressure to deliver the Timesheet Submission solution. It is therefore decided that they will not extend the service design at this point. The standards applied so far have guaranteed them an easily extensible service design, where additional operations can be added without breaking the original service interface.

*Fig 3.18 An Employee Service offering a full range of operations.*

Following is an example that shows how the `portType` construct could be expanded with supplementary operations (*documentation elements have been omitted to save space)*:

**Example 3.26. An expanded `portType` construct.**

```xml
<portType name="EmployeeInterface">
  <operation name="GetWeeklyHoursLimit">
    <!-- Additional operations go here -->
  </operation>
</portType>
```
<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.

Case Study

Let's take another look at the two operations we designed into the Employee Service:
- GetWeeklyHoursLimit
- UpdateHistory

The first requires that we access the employee profile. At TLS, employee information is stored in two locations:
- Payroll data is kept within the accounting system repository, along with additional employee contact information.
- Employee profile information, including employee history details, is stored in the HR repository.

When an XML data representation architecture was first implemented at TLS, entity-centric XSD schemas were used to bridge some of the existing disparity that existed among the many TLS data sources. Being aware of this, the service architect investigates the origins of the Employee.xsd schema used as part of the Employee.wsdl definition to determine the processing requirements for the GetWeeklyHoursLimit operation.
3.5.3. Application service design (a step-by-step process)

Application services are the workhorses of SOA. They represent the bottom sub-layer of the composed service layer, responsible for carrying out any of the processing demands dictated to them by the business and orchestration layers.

Fig 3.20. Application services establish the bottom sub-layer of the service layer.
Unlike services in business-centric layers, the design of application services does not require business analysis expertise. The application service layer is a pure, service-oriented abstraction of an organization’s technical environments, best defined by those who understand these environments the most.

**Process description**

Process provides a proposed service design process for creating application service interfaces. Note that all references made to "application services" in this and remaining chapters imply that they are reusable utility application services.

**Fig 3.21. The application service design process.**

The 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.

**Case Study**

The focus of RailCo environment 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.

**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. 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.

**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:

1. 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.
2. 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).
3. 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.

**Case Study**

RailCo begins by deriving the two operation names

![Fig 3.23. The first cut of the Transform Accounting Documents Service.](image)
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.

**Example 3.27. The XSD schema types required by the TransformToNative operation.**

```xml
<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>
```

Upon assessing the message requirements of the TransformToXML operation, RailCo discovers that the required types are identical. Streamlining the schema design with shared complex types is considered, but RailCo decides against it. It chooses instead to create a second set of elements with redundant complex types because it would like the freedom to change these types independently in the future.

**Example 3.28. The additional types for use by the TransformToXML operation.**

```xml
<xsd:element name="TransformToXMLType">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="SourcePath" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```
Next, the initial version of the abstract service definition for the RailCo Transform Account Documents Service is completed by providing the remaining message and portType constructs.

**Example 3.29.** The message and portType constructs of the abstract Transform Accounting Documents Service definition.

```xml
<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">
  
  </operation>
</portType>
```
<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, as being those that are not intrinsically provided by the Web services platform service reusability, service autonomy, service statelessness, and service discoverability. Discoverability mechanism is useful. This becomes more of an infrastructure requirement that can be planned as part of an SOA implementation.

Case Study

The Transform Accounting Documents Service undergoes a review to ensure that it is properly incorporating service-orientation principles. First, the reuse potential of its two operations is assessed:

- TransformToNative
- TransformToXML

After some discussion around whether these two operations should be combined into one generic Transform operation, it is decided to leave them as they are. The descriptive nature of the operations is preferred, and RailCo would like the option of evolving each operation separately in the future.

Example 3.30. The Transform Accounting Documents portType 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>

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.
- Review any of the contemporary SOA characteristics
- Optionally incorporate WS-I Basic Profile rules and best practices to whatever extent possible.

Case Study

Some changes are made to the service as a result of the considerations taken into account as part of this step. Upon reviewing naming standards, it is determined that the chosen names for the operations are in line with existing conventions. The name given to the service itself, however, is not. Therefore, the service is renamed from "Transform Accounting Documents" to "Transform Accounting." Upon a review of the service design, increased extensibility is achieved with the following adjustments:

- The service name is changed again, this time shortened to just "Transform."
- The TransformXMLToNative and TransformNativeToXML operations are renamed to something more generic. The new names are ForAccountingImport and ForAccountingExport. This naming change trickles down to the element and message names as well.
Fig 3.24. The final design of the Transform Service.

Example 3.31. The revised types construct.

```xml
<types>
  <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"/>
</types>
```
<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>

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.

Case Study

Although RailCo would like to build highly reusable application services, it cannot afford to do so at this time. Having designed the Transform Service to be sufficiently generic and reusable so that it can be used by both the Invoice Submission and Order Fulfillment Processes, RailCo is confident that its immediate requirements are being fulfilled. Further, because extensibility is being emphasized as well, RailCo feels that it can continue to evolve this service if future reusability opportunities present themselves.

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, what components 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

The one issue that comes up when assessing the technical constraints of the Transform Service is a potential performance bottleneck. A review of existing metrics suggests that the conversion process can be time consuming when processing invoice or purchase order documents with more than twenty line items. Because the service will be performing its own processing, it is expected that an instance of the service will be tied up for the duration of the transformation process.

3.5.4. 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 steps. Fig.3.25 Task-centric business services can comprise the business service layer, along with entity-centric neighbors.
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.

**Process description**

This process starts off with a new kind of step in which workflow logic is mapped out. This is because task-centric business services are expected to contain and govern portions of business processes.

**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.
At first it appears as though this service does not contain much of anything. This is actually not unusual for smaller scale task-centric services. A service candidate with no operation candidates simply means that this service is a pure controller, solely dedicated to coordinating the underlying service composition. It also means that RailCo will need to define its service interface from scratch during this design process.

**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.

Case Study

RailCo generates activity diagrams for all foreseeable interaction scenarios involving the Invoice Processing Service. Let's have a look at two of these diagrams.

**Fig.3.29. A successful completion of the Invoice Submission Process.**

![Invoice Submission Process Diagram](image)

**Fig.3.30 A failure condition caused by an error during the transformation step.**

![Invoice Submission Process Diagram](image)

Step 2: Derive the service interface

Follow these suggested steps to assemble an initial service interface:

Use the application service operation candidates to derive a set of corresponding operations.

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. 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. Build the WSDL definition by creating the portType (or interface) area, inserting the identified operation constructs. Then add the necessary message constructs containing the part elements that reference the appropriate schema types.

Case Study

Because our service candidate provided us with no operation candidates, RailCo turns to the activity diagrams it created to derive the set of actions the service is required to perform.

- Start RailCo Invoice Processing Receives the notification message sent by the Polling Notification Service, which kicks off the RailCo Invoice Submission Process.
- Transform Invoice Issues a request for the Transform Service to retrieve the invoice document from the network folder and transform it into XML.
- Check TLS Metadata Issues a request to the Metadata Checking Service for it to determine whether it’s time to perform a metadata check (and then perform the metadata check, if required).
- Start TLS Invoice Processing Forwards the invoice document to TLS, which initiates the separate TLS invoice processing process.

Fig.3.31. Identified requests and responses for the Invoice Processing Service.

Step 3: Apply principles of service-orientation

Before we get too far ahead in our service design, it is beneficial to take another look at the four...
service-orientation principles, which are not automatically provided to us through the use of Web services (service reusability, service autonomy, service statelessness, and service discoverability). Reuse opportunities for task-centric services are much more rare than for entity-centric and application services. This is because task-centric services represent a portion of workflow logic specific to a business process.

**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.

**Example.3.32. The portType construct with an additional documentation element.**

```xml
<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. (A set of recommended guidelines is provided at the end of this chapter.)
- 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.

**Case Study**

Supporting the characteristic of extensibility is key to RailCo, as they are uncertain as to how their SOA will grow and evolve over time. In reviewing the service interface definition for the Invoice...
Processing Service, they realize that they are tailoring this interface to a single service requestor: the Polling Notification Service. It is foreseeable that the process logic encapsulated by this service could need to be invoked differently in the future.

For example:

- A change to the overall invoice submission process may require that this service be sent a pre-transformed XML version of the actual invoice document.
- If new services continue to be added to the RailCo technical environment, it would also be beneficial for this service to accept an invoice document so that it can participate in larger service compositions.

To address these extensibility requirements, RailCo makes an adjustment to the schema markup within the `types` construct. Because an XSD schema representing the invoice document already exists (as a result of building the underlying processing logic for the Transform Service), it is decided to incorporate this schema as part of this WSDL.

**Example.3.33. The revised `types` construct of the Invoice Processing Service definition.**

```xml
<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>
```

**Example. 3.34 The `documentation` element contents also have been changed.**

```xml
<documentation>
```

M.Senthil Kumar, Asst. Prof.
Department of Information Technology, SMEC
Initiates the Invoice Submission Process. Requires either the invoice document location or the document.

Example. 3.35. The revised message and portType constructs.

```xml
<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="receiveSubmitMessage"/>
  </operation>
</portType>
```

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.

Case Study

In our original workflow logic we established a step in the process that positioned the Transform Service as taking care of both validation and transforming tasks. The workflow logic therefore needs to be altered to make the interaction with the Transform Service optional if the Submit operation of the Invoice Processing Service receives an invoice document as part of its input (because it is already transformed and because validation of the document will occur at the time it is received). This change does not affect other services; it only requires that a new conditional processing step be added to the application logic encapsulated by the Invoice Processing Service.