Arunai Engineering College

Tiruvannamalai

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

IT8074 - SERVICE ORIENTED ARCHITECTURE

PART B - 13 MARKS

UNIT I XML

XML document structure – Well-formed and valid documents – DTD – XML Schema – Parsing XML using DOM, SAX – XPath - XML Transformation and XSL – Xquery

Part - B

1. Explain in detail about the basic terms of XML Document Structure?

Markup Languages and Self-Describing Data:

- The markup languages in general and XML in particular really contain only two kinds of information: **markup elements and actual data** that are given meaning by these markup elements. Together, these are known as ***XML text.***

Goal of XML and its intended use:

- XML can be used to **encode any structured information**, XML is good at representing information that has an ***extensible, hierarchical* format** and requires **encoding of *metadata***. These three concepts form the basis of the XML language's structure and data model.

XML Syntax

Markup text rules:

- These special characters are known as ***delimiters.*** The XML language has four special delimiters.

TABLE 2.1 XML Delimiter Characters

Character	*Meaning*
<	The start of an XML markup tag
>	The end of an)> less than angle brackets> XML markup tag
&	The start of an XML entity
;	The end of an XML entity

Example: A simple XML document that demonstrates the self-describing property of XML.

LISTING 2.1 XML As a Self-Describing Language

```
<?xml version="1.0"?>
<the_following_text_is_my_first_name>Ron</the_following_text_is_my_first_name>
```

Example: A Simple XML Document

<xml version="1.0">

```
<book category="Comics">
<title lang="en">Everyday Italian</title>
<author>Sherlock</author>
<year>2005</year>
<price>30.00</price>
</book>
```

XML Document Structure

The major portions of an XML document include the following:

- The XML declaration
- The Document Type Declaration
- The element data
- The attribute data
- The character data or XML content

XML Declaration:

- The **first part** of an **XML document** is the declaration. The XML declaration is a **processing instruction of the form <?xml ...?>.**
- XML document and indicates the version of XML also indicates the presence of external markup declarations and character encoding.
- The XML declaration consists of a number of components. Table 2.2 lists these various components and their specifications

TABLE 2.2 Components of the XML Declaration

Component	*Description*
`<?xml`	Starts the beginning of the processing instruction (in this case, for the XML declaration).
`Version="xxx"`	Describes the specific version of XML being used in the document (in this case, version 1.0 of the W3C specification). Future iterations could be 2.0, 1.1, and so on.
`standalone="xxx"`	This standalone option defines whether documents are allowed to contain external markup declarations. This option can be set to `"yes"` or `"no"`.
`encoding="xxx"`	Indicates the character encoding that the document uses. The default is `"US-ASCII"` but can be set to any value that XML processors recognize and can support. The most common alternate setting is `"UTF-8"`.

Standalone document declaration:

This defines whether an external DTD will be processed as part of the XML document. When standalone is set to "yes", only internal DTDs will be allowed. When it is set to "no", an external DTD is required and an internal DTD becomes an optional feature.

Valid XML Declarations:

LISTING 2.3 Valid XML Declarations

```
<?xml version="1.0" standalone="yes"?>
<?xml version="1.0" standalone="no"?>
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

XML document complies with version 1.0.The specification requires external markup declarations that are encoded in UTF-8.

- The first declaration defines a **well-formed XML document**,
- The second declaration defines a **well-formed and valid XML document.**
- The third declaration shows a more **complete definition** that states a typical use-case for XML.

Document Type Declaration:

- The Document Type Declaration **(DOCTYPE)** gives a **name to the XML content** and provides guarantee for the document's validity, either **by including or specifying** a link to **a Document Type Definition (DTD).**
  - **Valid XML documents-** We must declare the document type to which they fulfill.
  - **Well-formed XML documents –** We can include the DOCTYPE to simplify the task of the various tools that will be manipulating the XML document.
- A **Document Type Declaration** names the document type and **identifies** the **internal content by** specifying the ***root element.***

LISTING 2.4 General Forms of the Document Type Declarations

```
<!DOCTYPE NAME SYSTEM "file">
<!DOCTYPE NAME [ ]>
<!DOCTYPE NAME SYSTEM "file" [ ]>
```

- **The first declaration** - use of an **externally defined DTD subset**.
- **The second declaration** - **internally defined subset** within the document.

- **The third declaration** - provides a place for **inclusion of an internally defined DTD** subset between the square brackets while also making use of an **external subset**.

  Eg: <!DOCTYPE shirt SYSTEM "shirt.dtd">

- DTD is saved to a file named shirt.dtd, which saved in the same path as the XML document.

Difference between External and Internal DTD subset:

- The only real difference between internally and externally defined DTD subsets is that the DTD content itself is contained within the square brackets, in the case of internal subsets, whereas external subsets save this content to a file for reference, usually with a .dtd extension.

Component	*Description*
<	The start of the XML tag (in this case, the beginning of the Document Type Declaration).
!DOCTYPE	The beginning of the Document Type Declaration.
NAME	Specifies the name of the document type being defined. This must comply with XML naming rules.
SYSTEM	Specifies that the following system identifier will be read and processed.
"file"	Specifies the name of the file to be processed by the system.
[	Starts an internal DTD subset.
]	Ends the internal DTD subset.
>	The end of the XML tag (in this case, the end of the Document Type Declaration) .

2. Explain in detail the various XML mark up contents (or) the basic elements of XML.(Nov/Dec2016)

Markup and Content In general, six kinds of markup can occur in an XML document:

- Elements
- Entity references
- Comments
- Processing instructions

- Marked sections
- Document type declarations.

2.1 Elements:

- XML elements are either a matched pair of XML tags or single XML tags that are "self-closing."

  For example : <shirt> </shirt>.
- When elements do not come in pairs, the element name is suffixed by the forward slash. **For example :** <on_sale/>
- If no other matching element of the same name used in a different manner. These "unmatched" elements are known as **empty elements.** The trailing "/>" in the modified syntax indicates to a program processing the XML document that the element is empty and no matching end tag should be required.

2.2 Attributes: Within elements, additional information can be communicated to XML processors that modify the nature of the encapsulated content.

LISTING 2.5 Attribute Examples

```
<price currency="USD">…</price>
<on_sale start_date="10-15-2001"/>
```

Attributes can be

- **Required , Optional** - contain freeform text or contain one of a set list of enumerated values
- **Fixed value**- contain a specific value

2.3 Entity References:

Each **entity has a unique name** that is **defined as part of an entity declaration in a DTD or XML Schema.** Entities are used by simply referring to them by name. **Entity references are delimited by an ampersand** at the beginning and a semicolon at the ending.

LISTING 2.6 Sample Entity References

```
<description>The following says that 8 is greater than 5</description>
<equation>4 &gt; 5</equation>
<prescription>The Rx prescription symbol is &#8478;
        which is the same as &#x211E;</prescription>
```

Difference between Internal and External Entities

Internal entities	External entities

Defined and used within the context of a document	Defined in a source that is accessible via a URI
Consists of simple string replacements	Consists of entire XML documents or non-XML text, such as binary files.
No need to define the type of file	Must define the type of the file

The Internal and External Entities can be general or parameter entities.

Character reference - Used to insert arbitrary Unicode characters into an XML document. This allows international characters to be entered even if they can't be typed directly on a keyboard.

For example: &#8478, U+211

2.4 Comments:

Comments can be placed anywhere in a document and are not considered to be part of the textual content of an XML document.

The character sequence <!-- begins a comment and --> ends the comment

LISTING 2.7 A Sample Comment

```
<!-- The below element talks about an Elephant I once owned... -->
<animal>Elephant</animal>
```

2.5 Processing Instructions:

Processing instructions (PIs) **perform a similar function as comments** in that **they are not a textual part of an XML document** but **provide information** to applications as to **how the content should be processed**.

Processing instructions have the following form:

<?instruction options?>

The instruction name, **called the *PI target***, is a special identifier that the processing application is intended to understand.

LISTING 2.8 Example of a Processing Instruction

```
<?send-message "process complete"?>
```

2.6 Marked CDATA Sections:

Some **documents** will **contain a large number of characters and text** that an **XML processor should ignore and pass** to an application. These are **known as character data (or CDATA) sections**. The **CDATA section instructs the parser** to **ignore all markup characters except the end of the CDATA** markup instruction.

CDATA sections follow this general form:

<![CDATA[content]]>

LISTING 2.9 A Sample CDATA Section

```
<object_code>
    <![CDATA[
    function master(poltice integer) {
  if poltice<=3 then {
    intMaster=poltice+IntToString(FindElement("<chicken>"));
  }
}
]]>
</object_code>
```

2.7 Document Type Definitions:

Document Type Definitions (DTDs) provide a means for **defining what XML markup can occur in an XML documen**t. It provides a mechanism to guarantee that a given XML document complies with a well-defined set of rules for document structure and content.

2.8 XML Content

The content between XML elements is where most of the value lies in an XML document. The **XML content** can **consist of any data at all, including binary data**.

3. Compare XML with SGML, HTML and Database files

Advantages of XML over SGML

XML	SGML
XML permits well-formed documents to be parsed without the need for a DTD	SGML implementations require some DTD for processing
XML has much simpler syntax	SGML syntax complex
The XML specification is very small,	The SGML specification is very large with large codes

Advantages of XML over HTML

- HTML was designed as a language to present hyperlinked, formatted information in a Web browser.
- It has no capability to represent metadata, provide validation, support extensibility by users, or support even the basic needs of e-business.
- The difference is that HTML is intended for consumption by humans, whereas XML is meant for both machine and human consumption.

Advantages of XML over Databases and Flat Files

- XML is a structured document format that includes not only the data but also metadata that describes that data's content and context.
- They either represent simply the information to be exchanged without metadata

- Relational and object-oriented databases and formats can represent data as well as metadata, but their formats are not text based.
- Most databases use binary format to represent their information.

4. Explain about the various XML content models

A **content model provides a framework** around which the **extensibility features of XML can be taken into advantage.**

Types of XML Content models

- Open Content Model
- Closed Content Model
- Mixed Content Model

Open Content Model

- An **"open" content model enables** a **user to add additional elements** and **attributes** to a document without them having to be explicitly declared in a DTD or schema.
- An open content model, users can take full advantage of the extensibility of XML without having to make changes to a DTD.
- In an **open content model**, **all required elements must be present**, but it is not invalid for additional elements to also be present.

Closed Content Model

- A "**closed" content model restricts elements and attributes** to only those that are specified in the DTD or schema.
- A **DTD is a closed content model** because it **describes what may appear** in the **content of the element.**
- Closed models are helpful when implementing strict document exchange and provide a means to guarantee that incoming data fulfill with data requirements.

Mixed Content Model

- A **"mixed" content model**, which **enables individual elements to** allow an **arbitrary mixture of text and additional** elements.
- These mixed elements are useful when freeform fields, with possible XML or other markup data are to be included.

5. Explain in detail about the Well formed and valid XML documents

An XML document with correct syntax is called "Well Formed".

An XML document validated against a DTD is both "Well Formed" and "Valid".

Rules of XML Structure (or) Well formed XML documents

a) All XML Elements Must Have a Closing Tag

- XML requires all tags to be closed.
- They can be **closed by matching a beginning element tag with a closing tag**, or they can be closed by the use of empty elements.

LISTING 2.11 Incorrect XML Due to Unclosed Tags

```
<markup>This is not valid XML
<markup>Since there is no closing tag
```

b) XML Tags Are Case Sensitive

XML elements and attributes are case sensitive. This means that differences in capitalization will be interpreted as different elements or attributes

In XML, the elements <shirt> and <Shirt> are different

LISTING 2.12 Incorrect XML Due to Capitalization Mismatch

```
<Markup>These two tags are very different</markup>
```

c) All XML Elements Must Have Proper Nesting

XML requires that **elements be nested in proper hierarchical order**. Tags must be closed in the reverse order in which they are opened.

<oxygen> <nitrogen>These tags improperly nested </oxygen> </nitrogen> **Incorrect XML**	<oxygen> <nitrogen>These tags are properly nested </nitrogen> </oxygen> **Correct XML**

d) All XML Documents Must Contain a Single Root Element

- XML documents **must contain a single root element**. All other **elements** in the XML document are then **nested within this root element**.
- Once the root element is defined, any number of child elements can branch.
- The root element is the most important one in the document because it contains all the other elements and reflects the document type as declared in the Document Type Declaration.

```
<?xml version="1.0" encoding="UTF-8"?>
<note>     //single root element
```

```
  <from>John</from>
  <heading>Reminder</heading>
  <body>Wake up!</body>
</note>
```

e) Attribute Values Must Be Quoted

When **attributes are used** within XML elements, **their values must be surrounded by quotes**. Although most systems accept single or double quotes for attribute values, it is generally accepted to use double quotes around attribute values.

INCORRECT	**CORRECT**
<note date=12/11/2007> <to>Tove</to> <from>Jani</from> </note>	**<note date="12/11/2007">** <to>Tove</to> <from>Jani</from> </note>

f) Attributes May Only Appear Once in the Same Start Tag

Even though **attributes may be optional**, when they are present, **they can only appear once.** By only allowing a single attribute name/value pair to be present, the system avoids any conflicts or other errors.

LISTING 2.16 Incorrect XML Due to Multiple Attribute Names in Start Tag

```
<shirt size="large" size="small">Zippy Tee</shirt>
```

g) Attribute Values Cannot Contain References to External Entities

The **attribute values can make use of internally defined entities** and generally available entities, such as < and ".

h) All entities except amp, lt, gt, apos, and quot must be declared before they are used.

The entities cannot be used before they are properly declared. Referring to an undeclared entity would obviously result in an XML document that is not well formed and proper.

XML Well formed and Valid Document: A well-formed XML document may in addition be valid if it meets certain further constraints. **An XML document is valid if it has an associated document type declaration** and if the document complies with the constraints expressed in it. Only documents that refer to a DTD can be checked for validity.

Example for well formed and Valid XML documents:

XML DTD(document type definition) (book.dtd)

```
<!ELEMENT   Books    (title,author,pages,price,example)>
<!ELEMENT   title    (#PCDATA) >
<!ELEMENT   author   (#PCDATA)
<!ELEMENT   pages    (#PCDATA) >
<!ELEMENT   price    (#PCDATA) >
<!ELEMENT   example  (#PCDATA) >
```

Well formed not valid XML document (bookdetails.xml)

```
<?xml version='1.0' standalone='yes'?>
 <Books>
  <title>java book </title>
  <author>nick bore </author>
  <pages> 1020 </pages>
  <price> $20 </price>
  <example>xml - valid xml file</example>
</Books>
```

This appears to be valid because it follows the structural rules of the DTD. However, because **it does not contain a document type declaration**, the parser has no DTD to compare the document instance against in order to determine validity. Therefore, **it is well-formed, but not valid,** or at least its validity cannot be determined.

Well formed not valid XML document (bookdetails1.xml)

```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE Books SYSTEM "book .dtd" >
 <Books>
 <title>java book </title>
 <author>nick bore </author>
 <pages> 1020 </pages>
 <example>xml - valid xml file</example>
 </Books>
```

With the inclusion of a reference to a DTD, a validating parser can check this instance. It will conclude, however, that **there is a missing element, price** , so **this document is well-formed but invalid.**

Well formed and valid XML document (bookdetails2.xml)

```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE Books SYSTEM "book .dtd" >
 <Books>
 <title>java book </title>
 <author>nick bore </author>
 <pages> 1020 </pages>
```

```
 <price> $20 </price>
<example>xml - valid xml file</example>
</Books>
```

This document is well-formed and valid because it **matches the structure defined by the DTD**, which **is referenced in the document type declaration**.

5. Explain about the various namespaces in XML. (Nov/Dec 17)

XML is an open standard in which XML authors are free to create whatever elements and attributes they wish, it's unavoidable that multiple XML developers will choose the same element and attribute names for their standards. This often results in a conflict when trying to mix XML documents from different XML applications. To overcome this XML namespaces can be useful to overcome this problem.

Namespace: A **Namespace** is a **set of unique names**. Namespace is a mechanism by which **element and attribute name can be assigned to group**. The **Namespace** is **identified by URI(Uniform Resource Identifiers - string of characters which identifies an Internet Resource).**

Namespace Declaration:

Within an XML document, namespaces can be **declared using** one of **two methods: a default declaration or an explicit declaration**.

A Namspace is declared using reserved attributes. Such an attribute name must either be **xmlns** or begin with **xmlns:** shown as below:

<element xmlns:name="URL">

Syntax

- The Namespace starts with the keyword **xmlns.**
- The word **name** is the **Namespace prefix.**
- The **URL** is the **Namespace identifier.**

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<cont:contact xmlns:cont="www.mec.com/profile">
 <cont:name>John</cont:name>
 <cont:company>MEC</cont:company>
 <cont:phone>123-4567</cont:phone>
</cont:contact>
```

Using XML Namespaces:

The following sample XML document:

```
<Customer>
<Name>John Smith</Name>
</Customer>
```

This sample document contains the root element <Customer>, which contains a child element called <Name>. We can clearly determine that the <Name> element contains the name of the customer referred to by the <Customer> element.

Let's have another sample XML document.

```
<Product>
<Name>Memory cards</Name>
</Product>
```

This document contains a <Product> element as the root element and a <Name> element, which contains the name of the product.

The following XML document could be constructed from the previous two XML documents to indicate that a customer has placed an order for a particular product:

```
<Customer>
<Name>John Smith</Name>
<Order>
<Product>
<Name>Memory cards</Name>
</Product>
</Order>
</Customer>
```

The **user can easily distinguish the differences between the two <Name> elements**. The first <Name> element, which appears as a child of the <Customer> element, contains the customer's name. The second <Name> element, on the other hand, contains the product's name. But the **parser can't able to find the difference** we have to explicitly specify so that the parser can find the difference between these two names.

Therefore, modifying the preceding XML document to specify the appropriate namespaces turns it into this:

```
<Customer>
<cust:Name xmlns:cust="customer-namespace-URI">John Smith</cust:Name>
<Order>
```

<Product>

<prod:Name xmlns:prod="product-namespace-URI">Memory cards</prod:Name>

</Product>

</Order>

</Customer>

Now, the XML parsers can easily tell the difference between any validation rules between the customer's <Name> element and the product's <Name> element.

Default declaration of Namespace:

A **default namespace declaration specifies a namespace to use for all child elements of the current element** that do not have a namespace prefix associated with them.

<Customer xmlns="http://www.eps-software.com/po">

<Name>Travis Vandersypen</Name>

<Order>

<Product>

<Name>Mobile Phones</Name>

</Product>

</Order>

</Customer

For this XML document, **all child elements of the <Customer> element** are specified as **belonging to the http://www.eps-software.com/po namespace.**

Explicit declaration of Namespace:

It may be necessary and more readable to **explicitly declare an element's namespace**. This is accomplished much the same way in which a default namespace is declared, **except a prefix is associated with the xmlns attribute**.

<**po:**Customer xmlns:po="http://www.eps-software.com/**po**">

<**po**:Name>Travis Vandersypen</**po**:Name>

<**po**:Order>

<**po**:Product>

<**po**:Name>Hot Dog Buns</**po**:Name>

</**po**:Product>

</**po**:Order>

</**po**:Customer>

We have used the **prefix of po** with the elements in this document to explicitly declare the namespace.

The explicitly declaring namespaces becomes useful when we utilize elements from different namespaces, :

<cust:Customer xmlns:cust="http://www.eps-software.com/customer"

➥xmlns:ord="http://www.eps-software.com/order">

<cust:Name>Travis Vandersypen**</cust:Name>**

<ord:Order>

<ord:Product>

<**ord:Name xmlns:prod="product-namespace-URI">**Mobile Phones**</ord:Name>**

</ord:Product>

</ord:Order>

</cust:Customer>

We can see that two different namespaces are referenced: one for customers and one for orders. This allows a different set of rules to be applied for customer names versus product names.

Identifying the Scope of Namespaces:

By default, all child elements within a parent element , appear within the parent's namespace. This allows all child elements to "inherit" their parent element's namespace. However, this "inherited" namespace can be overwritten by specifying a new namespace on a particular child element

<Customer xmlns="http://www.eps-software.com/customer">

<Name>Travis Vandersypen</Name>

<Order xmlns="http://www.eps-software.com/order">

<Product>

<Name>Mobile Phones </Name>

</Product>

</Order>

</Customer>

The <Customer> element declares a default namespace located at http://www.eps-software.com/customer. All elements contained within the <Customer> element "inherit", the namespace declared by the <Customer> element. However, the <Order> element also declares a default namespace it is overwritten so the child elements inherit the namespace declared by the <Order> element.

6. Explain in detail about the Document type Definition. With examples explain internal and external DTD(Nov/Dec2016)

Document Type Definition:

DTD stands for Document Type Definition. A Document Type Definition allows the XML author to **define a set of rules for an XML document** to **make it valid**. An **XML document is considered "well formed"** if that **document is syntactically correct** according to the **syntax rules of XML 1.0.**

Features of DTD (or) Importance of DTD :

- The **DTD** will **define the elements required by an XML** document, such as the **elements** that are optional, the **number of times an element should** (could) **occur**, and the **order** in which **elements should be nested**.
- DTD markup also defines **the type of data that will occur in an XML** element and the attributes that may be associated with those elements.
- **A document**, even if well formed, is **not considered valid if it does not follow the rules defined in the DTD.**

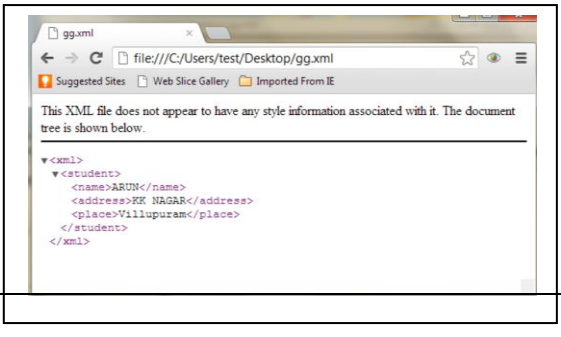
Types of DTD:

- **Internal DTD - residing within the body of a single XML document**
- **External DTD - referenced by the XML document**

Example : Internal DTD

Internal DTD file is within the DTD elements in XML file:

```
<?xml version="1.0"encoding="UTF=8"?>
<!DOCTYPE student[
<!ElEMENT student(name,address,place)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT address(#PCDATA)>
<!ELEMENT place(#PCDATA)>
]>
```



<student>
<name>ARUN</name>
<address>KK NAGAR</address>
<place>Villupuram</place>
</student>

The Document Type Declaration will appear between the XML declaration and the start of the document itself.

Example : External DTD

DTD File: (student.dtd)

<!ElEMENT student(name,address,place)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT address(#PCDATA)
<!ELEMENT place(#PCDATA)>

XML File:(externaldtd.xml)

<?xml version="1.0" encoding="UTF=8"?>
<!DOCTYPE student SYSTEM "student.dtd">
<student>
<name>ARUN</name>
<address>KK NAGAR</address>
<place>Villupuram</place>
</student>

External DTD file is created and its name must be specified in the corresponding XML file.

Structure of a Document Type Definition:

The structure of a DTD consists of :

- **Document type declaration**
- **Elements**
- **Attributes**
- **Entities**
- **Other keywords**

6.1 Document Type Declaration

There may be one Document Type Declaration per XML document.

Syntax:

```
<!DOCTYPE rootelement SYSTEM | PUBLIC DTDlocation [ internalDTDelements ] >
```

- **Exclamation mark (!)-** signify the **beginning of the declaration**.

- **DOCTYPE -** keyword used to **denote this as a Document Type Definition**.
- **root element** - **name of the root element** or document element of the XML document.
- **SYSTEM and PUBLIC** - keywords used to designate that the **DTD is contained in an external document.** SYSTEM keyword is used in cyclic with a URL to locate the DTD. **PUBLIC** - **specifies some public location** that will usually be some **application-specific resource reference**.
- **internalDTDelements** - **internal DTD declarations**. These declarations will always be placed within opening ([) and closing (]) brackets.

6.2 DTD Elements: All elements in a valid XML document are defined with an element declaration ,it defines the name and all allowed contents of an element.

Rules for declaring the DTD Elements:

- Element names must start with a letter or an underscore and may contain any combination of letters.
- Element names must never start with the string "xml".
- Colons should not be used in element names because they are normally used to reference namespaces.

Syntax:

<!ELEMENT elementname rule >

• **ELEMENT** - **tag name** that **specifies** that this is an **element definition**.

• **elementname** - **name** of the **element.**

• **rule** - **definition** to which the **element's data content must conform.**

Example : contactlist.dtd

```
<!ELEMENT contactlist (fullname, address, phone, email) >
<!ELEMENT fullname (#PCDATA)>
<!ELEMENT address (addressline1, addressline2)>
<!ELEMENT addressline1 (#PCDATA)>
<!ELEMENT addressline2 (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

- The first element in the DTD, contactlist, is the document element which is the parent element and contains the fullname, address, phone, and email as child elements.

- The rule for all child element is each contains parsed character data (#PCDATA), which means that the elements will contain marked-up character data that the XML parser will interpret.
- The address element has two child elements: addressline1 and addressline2.
- This DTD defines an XML structure that is nested two levels deep. The root element, contactlist, has four child elements. The address element is, in turn, parent to two more elements.

Example: contactlist.xml

```
<?xml version="1.0"?>
<!DOCTYPE contactlist SYSTEM "contactlist.dtd">
<contactlist>
<fullname> Bobby </fullname>
<address>
<addressline1>101 South Street</addressline1>
<addressline2>Apartment #2</addressline2>
</address>
<phone>(405) 555-1234</phone>
<email>bs@mail.com</email>
</contactlist>
```

This is a valid XML document because it is well formed and complies with the structural definition laid out in the DTD.

DTD Element Rules:

Content Rules - The content rules for .elements deal with the actual data that defined elements may contain. These rules include the

- **ANY rule**
- **EMPTY rule**
- **#PCDATA rule.**

The ANY Rule

An element may be defined using the ANY rule. The **element may contain other elements and/or normal character data**. An element using the ANY rule would appear as follows:

<!ELEMENT elementname ANY>

Example: <elementname> This is valid content </elementname>

The EMPTY Rule This rule is the **exact opposite of the ANY rule**. An element that is defined with this rule will **contain no data**. However, an element with the EMPTY

rule could still contain attributes .The following element is an example of the EMPTY rule:

<!ELEMENT elementname EMPTY>

The #PCDATA Rule

The #PCDATA rule **indicates that parsed character data** will be contained in the element. **Parsed character data** is data that **may contain normal markup** and will be **interpreted and parsed by any XML parser** accessing the document. The following element demonstrates the #PCDATA rule:

<!ELEMENT elementname (#PCDATA)>

It is possible in an element using the #PCDATA rule to use the **CDATA keyword to prevent the character data from being parsed.**

CDATA

```
<sample>
<data>
<![CDATA[<tag>This will not be parsed</tag>]]>
</data>
</sample>
```

All the data between <![CDATA[and]]> will be ignored by the parser and treated as normal characters (markup ignored).

Structure Rules- This structure rules **deal** with **how that data may be organized**. There are two types of structure rules

- **Element only rule.**
- **Mixed rule.**

The "Element Only" Rule

It specifies that **only elements may appear as children of the current element**. The following element definition demonstrates the "element only" rule:

<!ELEMENT elementname (element1, element2, element3)>

If there are options for which elements will appear, the listed elements should be separated by the pipe symbol (|).The element defined here will have a single child element: either element1 or element2.

<!ELEMENT elementname (element1 | element2)>

The "Mixed" Rule

It defines **elements** that **may have both character data (#PCDATA) and child elements** in the data they contain. A list of options or a sequential list will be enclosed by parentheses. Options will be separated by the pipe symbol (|), whereas sequential lists will be separated by commas. The following element is an example of the "mixed" rule:

<!ELEMENT elementname (#PCDATA | childelement1 | childelement2)*>

Example: <!ELEMENT Son (#PCDATA | Name | Age)*>

This definition defines an element, Son, for which there may be character data, elements, or both. A man might have a son, but he might not. If there is no son, then normal character data (such as "N/A") could be used to describe this condition.

DTD Element Symbols: The element symbols can be used to control the occurrence of data.

Asterisk (*) - The data will appear zero or more times (0, 1, 2, …).

Eg: <!ELEMENT children (name*)>

Comma (,) - Provides separation of elements in a sequence**.**

Eg: <!ELEMENT address (street, city, state, zip)>

Parentheses [( )] - The parentheses are used to contain the rule for an element.

Eg: <!ELEMENT address (street, city, (state |province), zip)>

Pipe (|) - Separates choices in a set of options.

Eg: <!ELEMENT dessert (cake | pie)>

The element dessert will have one child element: either cake or pie.

Plus sign (+) - Signifies that the data must appear one or more times (1, 2,3, …).

Eg: <!ELEMENT appliances (refrigerator+)>

Question mark (?) - Data will appear either zero times or one time in the element.

Eg: <!ELEMENT employment (company?)>

No symbol- When no symbol is used, the data must appear once in the XML file.

Eg: <!ELEMENT contact (name)>

6.3 DTD Attributes

XML attributes are **name/value pairs** that are **used as metadata to describe XML elements.** Attributes are also defined in DTDs. **Attribute definitions** are **declared using** the **ATTLIST** declaration. An **ATTLIST** declaration will **define one or more attributes** for the element that it is referencing.

Syntax:

<!ATTLIST elementname attributename type defaultbehavior defaultvalue>

- **ATTLIST -** tag name that specifies that **this definition will be for an attribute list**.
- **elementname -** name of the element that the **attribute will be attached** to.
- **attributename - actual name** of the attribute.
- **type -** indicates **which of the 10 valid kinds of attributes** this attribute definition.
- **defaultbehavior –**specifies whether the **attribute will be required, optional, or fixed** in value.
- **defaultvalue - value** of the attribute if **no value is explicitly set.**

ATTLIST Declaration

<!ATTLIST name
sex CDATA #REQUIRED
age CDATA #IMPLIED >

An XML element using the attribute list declared here would appear as follows:
<name sex="male" age="30" >Michael </name>
The name element contains the value "Michael". It also has two attributes of Michael: sex, age.

#REQUIRED- Indicates that the value of the attribute must be specified

#IMPLIED-Indicates that the value of the attribute is optional

#FIXED- Indicates that the attribute is optional, but if it is present, it must have a specified set value that cannot be changed.

Default - This is not an actual default behavior type. The value of the default is supplied in the DTD. **Eg: <!ATTLIST children number CDATA "0">**

DTD Entities

Entities in DTDs are **storage unit**. Entities are **special markups** that contain **content for insertion into the XML document**. An entity's content could be well-formed XML, normal text, binary data, a database record, and so on.

Syntax:

<!ENTITY entityname [SYSTEM | PUBLIC] entitycontent>

• **ENTITY -** tag name that specifies that this definition will be for an entity.

• **entityname -**name by which the entity will be referred in the XML document.

• **entitycontent** -actual contents of the entity

• **SYSTEM and PUBLIC** - optional keywords.

Entities may either **point to internal data or external data.**

- **Internal entities** represent **data** that is **contained completely within the DTD**.

- **External entities point to content in another location via a URL**. External data could be anything from normal parsed text in another file, to a graphics or audio file, to an Excel spreadsheet.

Example for Using Internal Entities

book.dtd

```
<?xml version="1.0"?>
<!DOCTYPE library [
<!ENTITY cpy "Copyright 2000">
<!ELEMENT library (book+)>
<!ELEMENT book (title copyright)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT copyright (#PCDATA)>
]>
```

bookdetails.xml

```
<library>
<book>
<title>XML Complete Reference</title>
<author>Ron</author>
<copyright>&cpy;</copyright>
</book>
<book>
<title>Java Complete Reference</title>
<author>Deitel</author>
<copyright>&cpy;</copyright>
</book>
</library>
```

In the copyright element of the XML document , this entity is referenced by using &cpy;. When this document is parsed, &cpy; will be replaced with "Copyright 2000"

Types of Entities:

- **Predefined Entities**
- **External Entities**
- **Parameter Entities**

Predefined Entities:

There are five predefined entities. These entities do not have to be declared in the DTD

TABLE 3.4 Predefined Entities

Entity	*Content*
&amp;	&
&lt;	<
&gt;	>
&quot;	"
&apos;	'

Examples for using Predefined Entities

```
<icecream>
<flavor>Cherry Garcia</flavor>
<vendor>Ben &amp Jerry's</vendor>
</icecream>
```

In this example, the ampersand in "Ben & Jerry's" is replaced with the predefined entity for an ampersand (&amp).

External Entities

External entities are used to **reference external content**. The external entities get their content **by referencing it via a URL** placed in the entitycontent portion of the entity declaration. Either the SYSTEM keyword or the PUBLIC keyword is used here to let the XML parser know that the content is external.

Examples for Using External Entities

```
<?xml version="1.0"?>
<!DOCTYPE employees [
<!ENTITY bob SYSTEM "http://srvr/emps/bob.xml">
<!ENTITY nancy SYSTEM "http://srvr/emps/nancy.xml">
<!ELEMENT employees (clerk)>
<!ELEMENT clerk (#PCDATA)>
]>
<employees>
<clerk>&bob;</clerk>
<clerk>&nancy;</clerk>
</employees>
```

Non-Text External Entities and Notations

Some external entities will contain non-text data, such as an image file. We do not want the XML parser to attempt to parse these types of files. In order to stop the XML parser, we use the NDATA keyword.

```
<!ENTITY myimage SYSTEM "myimage.gif" NDATA gif>
```

A notation is a special declaration that identifies the format of non-text external data so that the XML application will know how handle the data. Notations are declared in the body of the DTD and have the following syntax:

<!NOTATION notationname [SYSTEM | PUBLIC] dataformat>

• **notationname -** name by which the notation will be referred in the XML document.

• **dataformat -** reference to a MIME type, ISO standard, or some other location that can provide a definition of the data being referenced.

Example using External Non-Text Entities

<!NOTATION gif SYSTEM “image/gif” >
<!ENTITY employeephoto SYSTEM “images/employees/MichaelQ.gif” NDATA gif >
<!ELEMENT employee (name, sex, title, years) >
<!ATTLIST employee pic ENTITY #IMPLIED >
<employee pic=”employeephoto”>
</employee>

Parameter Entities

The main difference between an internal entity and a parameter entity is that a parameter entity may only be referenced inside the DTD. Parameter entities are in effect entities specifically for DTDs.Parameter entities can be useful when you have to use a lot of repetitive or lengthy text in a DTD.

Syntax:

<!ENTITY % entityname entitycontent>

In the syntax, after the declaration, there is a space, a percent sign, and another space before entityname. This alerts the XML parser that this is a parameter entity and will be used only in the DTD. These types of entities, when referenced, should begin with % and end with ;.

Example using Parameter Entities

<!ENTITY % pc “(#PCDATA)”>
<!ELEMENT name %pc;>
<!ELEMENT age %pc;>
<!ELEMENT weight %pc;>

DTD Drawbacks

- DTDs are composed of non-XML syntax
- DTDS are not object oriented. There is no inheritance in DTDs.
- DTDs do not support namespaces very well. For a namespace to be used, the entire namespace must be defined within the DTD.
- DTDs have weak data typing and no support for the XML DOM.

- DTDs can be overridden there is possible for security issues.

7.Explain in detail about XML Schemas and XML files(Nov/Dec2016)

XML Schema

- A more **powerful way of defining the structure and constraining the contents of XML documents.** An XML Schema definition is itself an XML document
- An XML Schema describes the structure of an XML document, just like a DTD.
- Typically stored as a standalone .xsd file.
- XML (data) documents refer to external .xsd files

Use of XML Schema

- With XML Schema, XML files can carry a description of its own format.
- With XML Schema, independent groups of people can agree on a standard for interchanging data.
- With XML Schema, we can verify data. It is easier to convert data between different data types

XML Schema Definition

The XML Schema language is also referred to as **XML Schema Definition (XSD).**

XML Schema Structure

- Elements
- XSD Simple Elements
- XSD Complex Elements
- Attributes
- XSD Restrictions

7.1 Elements

Elements are the main building block of any XML document; they contain the data and determine the structure of the document. An element can be defined within an XML Schema (XSD) as follows:

<xs:element name="x" type="y"/>

name -name that will appear in the XML document.

type - provides the description of what can be contained within the element when it appears in the XML document.

XSD - The <schema> Element :

The <schema> element is the root element of every XML Schema.

```
<?xml version="1.0"?>
<xs:schema>
...
...
</xs:schema>
```

The <schema> element may contain some attributes such as namespace and other attributes.

Example Schema declaration:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns=http://www.w3schools.com
elementFormDefault="qualified">
</xs:schema>
```

- Specifies that the elements and data types that come from the "http://www.w3.org/2001/XMLSchema" namespace should be **prefixed with xs:**
- Indicates that the elements defined by this schema (note, to, from, heading, body.) come from the "http://www.w3schools.com" namespace.
- Indicates that the default namespace is "http://www.w3schools.com".
- Indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.

7.2 XSD Simple Elements

A simple element is an XML element that contains only text. It cannot contain any other elements or attributes. The text can be of many different types. It can be one of the types included in the XML Schema definition (boolean, string, date, etc.), or it can be a custom type

Defining a Simple Element

Syntax:

<xs:element name="xxx" type="yyy"/>

where xxx is the name of the element and yyy is the data type of the element.

XML Schema built-in data types.

The most common types are:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

Example: details.xml

```
<lastname>John</lastname>
<age>36</age>
<dateborn>1970-03-27</dateborn>
```

The corresponding simple element definitions: details.xsd

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>
```

Default and Fixed Values for Simple Elements

A default value is automatically assigned to the element when no other value is specified. In the following example the default value is "red":

<xs:element name="color" type="xs:string" default="red"/>

A fixed value is also automatically assigned to the element, and you cannot specify another value. In the following example the fixed value is "red":

<xs:element name="color" type="xs:string" fixed="red"/>

XSD Attributes

Simple elements cannot have attributes. If an element has attributes, it is considered to be of a complex type. But the attribute itself is always declared as a simple type.

Defining an Attribute

Syntax:

<xs:attribute name="xxx" type="yyy"/>

where xxx is the name of the attribute and yyy specifies the data type of the attribute.

Example: Here is an XML element with an attribute:

<lastname lang="EN">Smith</lastname>

The corresponding attribute definition:

<xs:attribute name="lang" type="xs:string"/>

Default and Fixed Values for Attributes

Attributes may have a default value OR a fixed value specified.

In the following example the default value is "EN":

<xs:attribute name="lang" type="xs:string" default="EN"/>

In the following example the fixed value is "EN":

<xs:attribute name="lang" type="xs:string" fixed="EN"/>

Optional and Required Attributes

Attributes are optional by default. To specify that the attribute is required, use the "use" attribute:

<xs:attribute name="lang" type="xs:string" use="required"/>

7.3 XSD Restrictions/Facets:

Restrictions are used to **define acceptable values for XML elements or attributes**. Restrictions on XML elements **are called facets.**

Restrictions on Values

The following example defines an element called "age" with a restriction. The value of age cannot be lower than 0 or greater than 120:

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on a Set of Values

To limit the content of an XML element to a set of acceptable values, we would use the enumeration constraint. The example below defines an element called "car" with a restriction. The only acceptable values are: Audi, Golf, BMW:

```
<xs:element name="car">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Audi"/>
      <xs:enumeration value="Golf"/>
      <xs:enumeration value="BMW"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on a Series of Values

To limit the content of an XML element to define a series of numbers or letters that can be used, we would use the pattern constraint.

The example below defines an element called "letter" with a restriction. The only acceptable value is ONE of the LOWERCASE letters from a to z:

```
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on Whitespace Characters

To specify how whitespace characters should be handled, we would use the whiteSpace constraint. The whiteSpace constraint is set to "preserve", which means that the XML processor WILL NOT remove any white space characters:

```
<xs:element name="address">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
    </xs:restriction>
```

```
  </xs:simpleType>
</xs:element>
```

Restrictions on Length

To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints.

The value must be minimum five characters and maximum eight characters:

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="5"/>
      <xs:maxLength value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

7.4 XSD Complex Elements

A complex element contains other elements and/or attribute.

There are four kinds of complex elements:

- Empty elements
- Elements that contain only other elements
- Elements that contain only text
- Elements that contain both other elements and text

Note: Each of these elements may contain attributes as well!

Defining a Complex Element

The complex XML element, "employee", which contains only other elements:

```
<employee>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</employee>
```

We can define a complex element in an XML Schema two different ways:

1. The "employee" element can be declared directly by naming the element, like this:

```
<xs:element name="employee">
```

```
<xs:complexType>
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

Here only the "employee" element can use the specified complex type. Note that the child elements, "firstname" and "lastname", are surrounded by the <sequence> indicator. This means that the child elements must appear in the same order as they are declared.

2. The "employee" element can have a type attribute that refers to the name of the complex type to use:

```
<xs:element name="employee" type="personinfo"/>
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Here not only the employee element all child elements can refer to the same complex type

XSD Empty Elements

An empty complex element cannot have contents, only attributes.

An empty XML element:

```
<product prodid="1345" />
```

The XSD for the empty XML element product

```
<xs:element name="product" type="prodtype"/>
<xs:complexType name="prodtype">
  <xs:attribute name="prodid" type="xs:positiveInteger"/>
</xs:complexType>
```

XSD Elements Only

An "elements-only" complex type contains an element that contains only other elements. An XML element, "person", that contains only other elements:

```
<person>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</person>
```

We can define the "person" element in a schema, like this:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

XSD Text-Only Elements

A complex text-only element can contain text and attributes.

This type contains only simple content (text and attributes), therefore we add a simpleContent element around the content. When using simple content, we must define an extension OR a restriction within the simpleContent

Example of an XML element, "shoesize", that contains text-only:

```
<shoesize country="france">35</shoesize>
```

The following example declares a complexType, "shoesize".

```
<xs:element name="shoesize">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="country" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Complex Types with Mixed Content

An XML element, "letter", that contains both text and other elements:

```
<letter>
```

Dear Mr.<name>John Smith</name>.

Your order <orderid>1032</orderid>

will be shipped on <shipdate>2001-07-13</shipdate>.

</letter>

The following schema declares the "letter" element:

```
<xs:element name="letter">
 <xs:complexType mixed="true">
  <xs:sequence>
   <xs:element name="name" type="xs:string"/>
   <xs:element name="orderid" type="xs:positiveInteger"/>
   <xs:element name="shipdate" type="xs:date"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

XSD Indicators

We can control HOW elements are to be used in documents with indicators.There are seven indicators:

Order indicators: used to define the order of the elements.

- All
- Choice
- Sequence

Occurrence indicators: used to define how often an element can occur.

- maxOccurs
- minOccurs

Group indicators: used to define related sets of elements.

- Group name
- attributeGroup name

Example for creating a Document type Definition , XML Schema Definition and XML for the Employee Details

DTD for Employee Information (employee.dtd)

```
<!ELEMENT Employee_Info (Employee)*>
<!ELEMENT Employee (Name, Department, Telephone, Email)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Department (#PCDATA)>
<!ELEMENT Telephone (#PCDATA)>
```

```
<!ELEMENT Email (#PCDATA)>
<!ATTLIST Employee Employee_Number CDATA #REQUIRED>
```

XML Schema definition for Employee Information（employee.xsd）

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
<xs:element name="Employee_Info" type="EmployeeInfoType" />
<xs:complexType name="EmployeeInfoType">
<xs:sequence>
<xs:element ref="Employee" minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>
<xs:element name="Employee" type="EmployeeType" />
<xs:complexType name="EmployeeType">
<xs:sequence >
<xs:element ref="Name" />
<xs:element ref="Department" />
<xs:element ref="Telephone" />
<xs:element ref="Email" />
</xs:sequence>
<xs:attribute name="Employee_Number" type="xs:int" use="required"/>
</xs:complexType>
<xs:element name="Name" type="xs:string" />
<xs:element name="Department" type="xs:string" />
<xs:element name="Telephone" type="xs:string" />
<xs:element name="Email" type="xs:string" />
</xs:schema>
```

Valid XML Document for XML Schema (employee.xml)

```
<?xml version="1.0"?>
<Employee_Info
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="employee.xs">
  <Employee  Employee_Number="105">
    <Name>John</Name>
    <Department>HR Department</Department>
    <Telephone>03-1452-4567</Telephone>
    <Email>john@xmltr.co.jp</Email>
  </Employee>
```

```
  <Employee  Employee_Number="109">
    <Name>Alex</Name>
    <Department>Sales Department</Department>
    <Telephone>03-6459-98764</Telephone>
    <Email>alex@xmltr.co.jp</Email>
  </Employee>
</Employee_Info>
```

8. Explain in detail about the XPath(or) How can we find the elements from the XML file. (Nov/Dec2016)

XPath

The XML Path Language (XPath) is a **standard for creating expressions that can be used to find specific pieces of information within an XML document**. XPath expressions are used by both XSLT (for which XPath provides the core functionality) and XPointer to locate a set of nodes.

XPath expressions have the ability to locate nodes based on the nodes' type, name, or value or by the relationship of the nodes to other nodes within the XML document. XPath expression can also return any of the following:

- A node set
- A Boolean value
- A string value
- A numeric value

XPath Processing

The **XML Path Language** is, used to **select the desired set of nodes from XML documents.** Each pattern describes a set of matching nodes to select from a hierarchical XML document. The XML Path Language navigates a hierarchical tree of nodes within an XML document to select the set of nodes.

XPath Terminology Nodes:

In XPath, there are **seven kinds of nodes**:

- **element**
- **attribute**
- **text**
- **namespace**
- **processing-instruction**
- **comment,**
- **document nodes.**

XML documents are **treated as trees of nodes.** The topmost element of the tree is called the root element.

Example :bookdetails.xml	**Xpath details of the XML document**
<?xml version="1.0" encoding="UTF-8"?> <bookstore> <book> <title lang="en">Harry Potter</title> <author>J K. Rowling</author> <year>2005</year> <price>29.99</price> </book> </bookstore>	**Nodes in the XML document** <bookstore> (root element node) <author>J K. Rowling</author> (element node) lang="en" (attribute node) **Atomic values** Atomic values are nodes with no children or parent. **J K. Rowling** "**en**" **Items** Items are atomic values or nodes.
Relationship of Nodes **Parent** Each element and attribute has one parent.From above example; the book element is the parent of the title, author, year, and price. **Children** Element nodes may have zero, one or more children. From above example; the title, author, year, and price elements are all children of the book element:	**Siblings** Nodes that have the same parent. From above example; the title, author, year, and price elements are all siblings: **Ancestors** A node's parent, parent's parent, etc.From above example;; the ancestors of the title element are the book element and the bookstore element: **Descendants** A node's children, children's children, etc.From above example; descendants of the bookstore element are the book, title, author, year, and price elements:

Selecting Nodes

XPath uses path expressions to **select nodes in an XML document**. The node is selected by following a path or steps. The most useful path expressions are listed below:

Expression	**Description**
nodename	Selects all nodes with the name "*nodename*"
/	Selects from the root node

//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

Predicates

Predicates are used to find a specific node or a node that contains a specific value.

Predicates are always embedded in square brackets.

Example XML document (bookdetails.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book>
  <title lang="en">Harry Potter</title>
  <price>29.99</price>
</book>
<book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
</book>
</bookstore>
```

Example for Path Expressions

Path Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element.
//title[@lang='en']	Selects all the title elements that have a "lang" attribute with a value of "en"
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00

XPath Axes - An axis defines a node-set relative to the current node.

AxisName	Result
ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node
ancestor-or-self	Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself

attribute	Selects all attributes of the current node
child	Selects all children of the current node
descendant	Selects all descendants (children, grandchildren, etc.) of the current node
descendant-or-self	Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself
following	Selects everything in the document after the closing tag of the current node
following-sibling	Selects all siblings after the current node
namespace	Selects all namespace nodes of the current node
parent	Selects the parent of the current node
preceding	Selects all nodes that appear before the current node in the document, except ancestors, attribute nodes and namespace nodes
preceding-sibling	Selects all siblings before the current node
self	Selects the current node

Location Path Expression A location path can be absolute or relative.

An **absolute location path**: **/step/step/...**

A **relative location path:** **step/step/...**

Each step is evaluated against the nodes in the current node-set. A step consists of:

- **An Axis** (defines the tree-relationship between the selected nodes and the current node)
- **A Node-Test** (identifies a node within an axis)
- **Zero Or More Predicates** (to further refine the selected node-set)

Syntax for a location step

axisname::nodetest[predicate]

Examples for Axes

Example	**Result**
child::book	Selects all book nodes that are children of the current node
attribute::lang	Selects the lang attribute of the current node

child::*	Selects all element children of the current node
attribute::*	Selects all attributes of the current node

To receive the result using XML path, a "location path" is needed to locate the result nodes. These location paths select the resulting node set relative to the current context. A location path has many location steps. Each step is further comprised of three pieces:

- **An axis-** The axis portion of the location step identifies the hierarchical relationship for the desired nodes from the current context.
- **A node test-** The node test portion of a location step indicates the type of node desired for the results. Every axis has a principal node type: If an axis is an element, the principal node type is element; otherwise, it is the type of node the axis can contain. Here's a list of these node tests:
  - comment()
  - node()
  - processing-instruction()
  - text()
- **A predicate-** The predicate portion of a location step filters a node set on the specified axis to create a new node set. A forward axis predicate contains the current context node and nodes that follow the context node. A reverse axis predicate contains the current context node and nodes that precede the context node.

XPath Operators

Below is a list of the operators that can be used in XPath expressions:

Operator	Description	Example
|	Computes two node-sets	//book | //cd
+	Addition	6 + 4
-	Subtraction	6 - 4
*	Multiplication	6 * 4
div	Division	8 div 4
=	Equal	price=9.80

or	or	price=9.80 or price=9.70
and	and	price>9.00 and price<9.90
mod	Modulus (division remainder)	5 mod 2

XPath Functions

XPath functions are used to evaluate XPath expressions and can be divided into one of four main groups:

- Boolean
- Node set
- Number
- String

9. Explain how XPointer and Xlink are used with XPath to find the individual elements and to link it with other XML files in the XML

Xpointers **address the individual parts of an XML document**. The node tests for an XPointer are, the most part, the same as for an XPath node test. However, in addition to the node tests already listed for XPath expressions, XPointer provides two more important node tests:

- **point()**
- **range()**

Location set in XPath

For an **XPath expression**, the **result from a location step** is known as a **node set;** for an **XPointer expression**, the **result is known as a location set**. Four of the functions that return location sets, id(), root(), here(), and origin().

Function	**Description**
id()	Selects all nodes with the specified ID
root()	Selects the root element as the only location in a location set
here()	Selects the current element location in a location set
origin()	Selects the current element location for a node using an out-of-line link

Points

XPointer points to allow a context node to be specified and an index position indicating how far from the context node the desired point is. Two different types of points can be represented using XPointer points:

- **Node points**
- **Character points**

Node points are **location points in an XML document** that are nodes that **contain child nodes.** If 0 is specified for the index, the point is considered to be immediately before any child nodes. A node point could be considered to be the gap between the child nodes of a container node.

When the **origin node is a text node**, the index **position indicates the number of characters.** These location points are **referred to as character points.**

Ranges

An XPointer range defines just that—a **range consisting of a start point and an endpoint.** A range will contain the XML between the start point and endpoint but does not necessarily have to consist of neat subtrees of an XML document.

XPointer Range Functions

Function	Description
end-point()	Selects a location set consisting of the endpoints of the desired location steps
range-inside()	Selects the range(s) covering each location in the location-set argument
range-to()	Selects a range that completely covers the locations within the location-set argument
start-point()	Selects a location set consisting of the start points of the desired location steps

Example:

```
<novel copyright="public domain">
    <title>The Wonderful Wizard of Oz</title>
    <author>L. Frank Baum</author>
    <year>1990</year>
</novel>
```

Syntax for using the Xpointer

xpointer(//title[position( ) =1]/text ( )/point( ) [position ( )=3])

Initially finds the document's first title element, then it takes its text node child. Within this text node, it selects the point between the third and fourth character. Apply it to this example, it would point to the space after **The**.

XLink

The XML Linking Language, allows a **link to another document to be specified on any element within an XML document.** The XML Linking Language creates a **link to another resource through** the, not through the actual elements **use of attributes specified on elements** themselves.

XLink Syntax

Example of how to use XLink to create links in an XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<homepages xmlns:xlink="http://www.w3.org/1999/xlink">
 <homepage xlink:type="simple" xlink:href="http://www.w3schools.com">Visit W3Schools</homepage>
 <homepage xlink:type="simple" xlink:href="http://www.w3.org">Visit W3C</homepage>
</homepages>
```

To get **access to the XLink features** we must **declare the XLink namespace**. The XLink namespace is: "**http://www.w3.org/1999/xlink".**

The xlink:type and the xlink:href attributes in the <homepage> elements come from the XLink namespace.

The **xlink:type="simple"** creates a simple "**HTML-like" link** (means "click here to go there").

The **xlink:href** attribute specifies the **URL to link to**.

XLink Example

The following XML document contains XLink features:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore xmlns:xlink="http://www.w3.org/1999/xlink">
<book title="Harry Potter">
 <description
 xlink:type="simple"
 xlink:href="/images/HPotter.gif"
 xlink:show="new">
 As his fifth year at Hogwarts School of Witchcraft and
 Wizardry approaches, 15-year-old Harry Potter is.....
 </description>
</book>
```

```
<book title="XQuery Kick Start">
  <description
  xlink:type="simple"
  xlink:href="/images/XQuery.gif"
  xlink:show="new">
  XQuery Kick Start delivers a concise introduction
  to the XQuery standard.....
  </description>
</book>
</bookstore>
```

XLink Attribute Reference

Attribute	Value	Description
xlink:actuate	onLoad onRequest other none	Defines when the linked resource is read and shown: • onLoad - the resource should be loaded and shown when the document loads • onRequest - the resource is not read or shown before the link is clicked
xlink:href	*URL*	Specifies the URL to link to
xlink:show	embed new replace other none	Specifies where to open the link. Default is "replace"
xlink:type	simple extended locator arc resource title none	Specifies the type of link

Simple Links

A simple link consists of a xlink:type attribute with a value of simple and, optionally, an xlink:href attribute with a specified value. A simple link may have any content, and even no content. They link exactly two resources together: one local and one remote.

Extended Links

Extended links gives the ability to specify relationships between an unlimited number of resources, both local and remote. In addition,these links can involve multiple paths between the linked resources.

10. Explain how XML can be parsed using Document Object Model

Document Object Model (DOM) (NOV/DEC 2017)

The Document Object Model (DOM) provides a **way of representing an XML document in memory** so that it can be manipulated by the software. **DOM** is a standard **application programming interface** (API) that makes it easy for programmers to **access elements and delete, add, or edit content and attributes**. **DOM** is a **tree structure** that **represents elements, attributes, and content**.

Simple XML Document

```
<?xml version="1.0" encoding="UTF"?>
<purchase-order>
<customer>James Bond</customer>
<merchant>Spies R Us</merchant>
<items>
<item>Night vision camera</item>
<item>Vibrating massager</item>
</items>
</purchase-order>
```

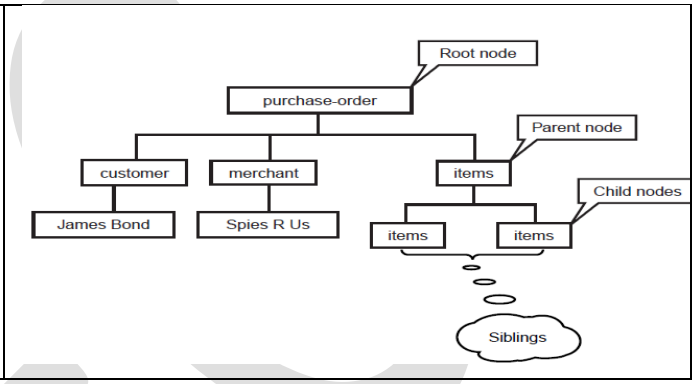


Fig .Tree structure representing the XML document

Node Parents, Children, and Siblings:

The nodes in the node tree have a hierarchical relationship to each other. The terms parent, child, and sibling are used to describe the relationships. Parent nodes have children. Children on the same level are called siblings (brothers or sisters).

- In a node tree, the top node is called the root
- Every node, except the root, has exactly one parent node
- A node can have any number of children
- A leaf is a node with no children
- Siblings are nodes with the same parent

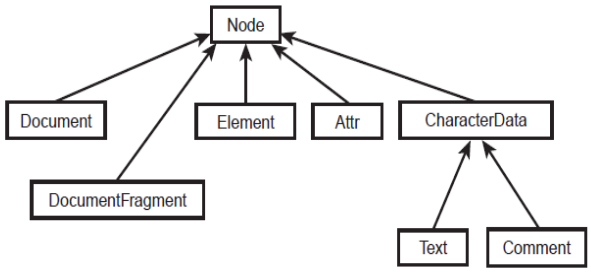
DOM Interfaces:

DOM interfaces contain methods for obtaining the parent, children, and siblings of any node. The DOM interfaces are defined in IDL so that they are language neutral.

Interface	Description
Node	The primary interface for the DOM. It can be an element, attribute, text, and so on
NodeList	An ordered collection of Nodes.

NamedNodeMap	An unordered collection of Nodes that can be accessed by name and used with attributes.
Document	A Node representing an entire document. It contains the root Node.
DocumentFragment	A Node representing a piece of a document. It's useful for extracting or inserting a fragment into a document.
Element	A Node representing an XML element.
Attr	A Node representing an XML attributes.
CharacterData	A Node representing character data
Text	A CharacterData node representing text.
Comment	A CharacterData node representing a comment.
DOMException	An exception raised upon failure of an operation.
DOMImplementation	Methods for creating documents and determining whether an implementation has certain features.

FIGURE 7.2
Interface relationships.



DOM based XML Processing

The primary goal of any XML processor is to parse the given XML document. Java has a rich source of in-built APIs for parsing the given XML document.

It is parsed in two ways:

1. **Tree based Parsing (DOM)**
2. **Event based Parsing (SAX).**

- The XML DOM contains methods (functions) to traverse XML trees, access, insert, and delete nodes.
- However, before an XML document can be accessed and manipulated, it must be loaded into an XML DOM object.
- An XML parser reads XML, and converts it into an XML DOM object that can be accessed with JavaScript.Most browsers have a built-in XML parser

Steps involved in Parsing using DOM

Following are the steps used while parsing a document using DOM Parser.

- Import XML-related packages.
- Create a DocumentBuilder
- Create a Document from a file or stream
- Extract the root element
- Examine attributes
- Examine sub-elements

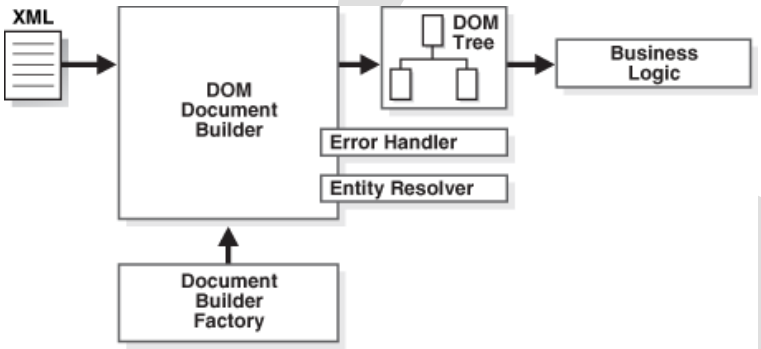


Fig. DOM Parser in Action

DOM XML Parser Example

XML File: staff.xml

```
<?xml version="1.0"?>
<company>
        <staff id="1001">
                <firstname>Raj</firstname>
                <lastname>Kumar</lastname>
                <role>Executive</ role>
                <salary>100000</salary>
        </staff>
        <staff id="2001">
                <firstname>Krishna</firstname>
                <lastname>Ram</lastname>
                < role >Manager</ role>
                <salary>200000</salary>
        </staff>
```

```
</company>
```

ReadXMLFile.java

```
import javax.xml.parsers
import org.w3c.dom;
import java.io.File;

public class ReadXMLFile
{
 public static void main(String argv[])
 {
     try
    {
          File fXmlFile = new File("staff.xml");
          //Create a DocumentBuilder
          DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
          DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
          Document doc = dBuilder.parse(fXmlFile);

          doc.getDocumentElement().normalize();
           System.out.println("Root element :" + doc. getDocumentElement(). getNodeName( ) );
          NodeList nList = doc.getElementsByTagName("staff");
          System.out.println("----------------------------");
          for (int temp = 0; temp < nList.getLength(); temp++)
         {
           Node nNode = nList.item(temp);
          System.out.println("\nCurrent Element :" + nNode.getNodeName());
          if (nNode.getNodeType() == Node.ELEMENT_NODE)
         {
    Element el = (Element) nNode;
   System.out.println("Staff id : " + el.getAttribute("id"));

System.out.println("FName:"+el.getElementsByTagName("firstname").item(0)getTextContent());
    System.out.println("LName                                    :                                    "
el.getElementsByTagName("lastname").item(0).getTextContent());
   System.out.println("Designation:                                    "                                    +
el.getElementsByTagName("role").item(0).getTextContent());
   System.out.println("Salary : " + el.getElementsByTagName("salary").item(0).getTextContent());
} } }
catch (Exception e)
{
          e.printStackTrace();
}
}
}
```

Output:

```
Root element :company
```

Current Element :staff
Staff id : 1001
First Name : Raj
Last Name : Kumar
Designation : Executive
Salary : 100000

Current Element :staff
Staff id : 2001
First Name : Krishna
Last Name : Ram
Designation : Manager
Salary : 200000

DOM Traversal:

Traversal is a convenient way to walk through a DOM tree and select specific nodes. This is useful when you want to find certain elements and perform operations on them.

Example

```
<!DOCTYPE html>
<html>
<body>
<p id="demo"></p>
<script>
var x, i ,xmlDoc;
var txt = "";
var text = "<book>" +
"<title>XML Services</title>" +
"<author> Deitel </author>" +
"<year>2005</year>" +
"</book>";
parser = new DOMParser( );
xmlDoc = parser.parseFromString(text,"text/xml");
// documentElement always represents the root node
x = xmlDoc.documentElement.childNodes;
for (i = 0; i < x.length ;i++)
{
   txt += x[i].nodeName + ": " + x[i].childNodes[0].nodeValue + "<br>";
}
document.getElementById("demo").innerHTML = txt;
</script>
</body>
</html>
```

Load the XML string into xmlDoc
Get the child nodes of the root element
For each child node, output the node name and the node value of the text node

Output:

title: XML Services
author: Deitel
year: 2005

11. Explain SAX based parsing with example.(Nov/Dec 2016 & 17)

SAX is an **API** that can be used to **parse XML documents**. A parser is a program that reads data a character at a time and returns manageable pieces of data. SAX provides a **framework for defining event listeners, or handlers**. The **handlers** are **registered with the SAX framework in order to receive events. Events** can include **start of document, start of element, end of element,** and so on.

The SAX Parser generates a stream of events ,the kinds of events are:

- The start of the document is encountered
- The end of the document is encountered
- The start tag of an element is encountered
- The end tag of an element is encountered
- Character data is encountered
- A processing instruction is encountered

Example:

Employee-Detail.xml

```
<?xml version = "1.0" ?>
<Employee-Detail>
        <Employee>
                <Emp_Id> 11032 </Emp_Id>
                <Emp_Name> Hari </Emp_Name>
                <Emp_E-mail> hari@gmail.com </Emp_E-mail>
        </Employee>
        <Employee>
                <Emp_Id> 11022 </Emp_Id>
                <Emp_Name> Ashok </Emp_Name>
                <Emp_E-mail> ashok@gmail.com </Emp_E-mail>
        </Employee>
        <Employee>
                <Emp_Id> 11011 </Emp_Id>
```

```
                    <Emp_Name> Elavarasan </Emp_Name>
                    <Emp_E-mail> ela@pec.in </Emp_E-mail>
                </Employee>
            </Employee-Detail>
```

EmployeeDetails.java

```
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.io.*;
public class EmployeeDetails
{
public static void main(String[] args) throws IOException
{
BufferedReader bf = new BufferedReader(new InputStreamReader
(System.in));
System.out.print("Enter XML file name:");
String xmlFile = bf.readLine();
EmployeeDetails detail = new EmployeeDetails(xmlFile);
}
public EmployeeDetails(String str)
{
try
{
File file = new File(str);
if (file.exists())
{
SAXParserFactory parserFact = SAXParserFactory.newInstance();
SAXParser parser = parserFact.newSAXParser();
System.out.println("XML Data: ");
DefaultHandler dHandler = new DefaultHandler()
{
boolean id;
boolean name;
boolean mail;
public void startElement(String uri, String localName, String
element_name, Attributes attributes)throws SAXException
{
if (element_name.equals("Emp_Id"))
{
id = true;
}
if (element_name.equals("Emp_Name"))
{
```

```
name = true;
}
if (element_name.equals("Emp_E-mail"))
{
mail = true;
}
}
public void characters(char[] ch, int start, int len) throws SAXException
{
String str = new String (ch, start, len);
if (id)
{
System.out.println("Emp_Id: "+str);
id = false;
}
if (name)
{
System.out.println("Name:   "+str);
name = false;
}
if (mail)
{
System.out.println("E-mail: "+str);
mail = false;
} } };
parser.parse(str, dHandler);
}
else
{
System.out.println("File not found!");
} }
catch (Exception e)
{
System.out.println("XML File hasn't any elements");
e.printStackTrace();
}  }  }
```

Output:

```
H:\WT LAB\Programs\Ex8>java EmployeeDetails
Enter XML file name:Employee-Detail.xml
XML Data:
Emp_Id:  11032
Name:    Hari
E-mail:  hari@gmail.com
```

Emp_Id: 11022
Name: Ashok
E-mail: ashok@gmail.com
Emp_Id: 11011
Name: Elavarasan
E-mail: ela@pec.in

Disadvantages of SAX Parser:

- SAX parsing is "single pass," so we can't back up to an earlier part of the document any more than you can back up from a serial data stream.
- SAX implementations are read-only parsers. They do not provide the ability to manipulate a document or its structure.
- There is no formal specification for SAX.

12. Differentiate DOM and SAX based XML Parsing (Nov/Dec 2016)

SAX	DOM
DOM is an in-memory tree structure of an XML document or document fragment.	SAX much simpler than DOM. SAX parsers tend to be smaller than DOM implementations.
DOM is a natural object model of an XML document, but it's not always practical.	There is no need to model every possible type of object that can be found in an XML document.
Large documents can take up a lot of memory. DOM might not describe the specific document efficiently.	SAX is an event-based API. Instead of loading an entire document into memory all at once, SAX parsers read documents and notify a client program when elements, text, comments, and other data of interest are found.
DOM contains many interfaces, each containing many methods.	SAX is a much lower level in API when compared with DOM. SAX is comprised of a handful of classes and interfaces.
The DOM parses XML in space	SAX parses XML in time
DOM parser hands you an entire document and allows you to traverse it any way you like. This can take a lot of memory	SAX can be significantly more efficient for large documents. SAX parsers sends events continuously

13. Explain in detail about transforming the XML documents

Transforming XML Documents

XML provides a vendor-independent, data-exchange mechanism used among

applications or companies. We would like to convert XML data to a different format.

For example, if a supplier provides a list of parts as an XML document, we might like to convert the XML document to use a different set of elements that are supported by our internal applications the **XML Stylesheet Language (XSL)** solves this problem of document conversion.

XSL Technologies

XSL has two independent languages:

• **The XSL Transformation Language (XSLT)** - used to convert an XML document to another format.

• **The XSL Formatting Object Language (XSL-FO**) - provides a way of describing the presentation of an XML document.

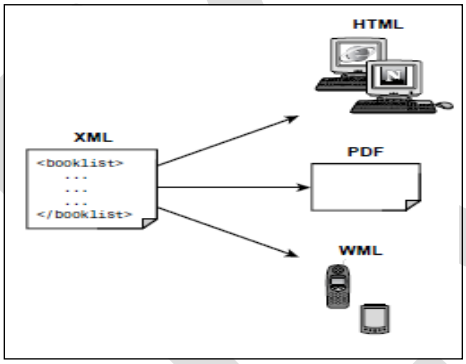


Fig. Publishing documents using XSLT

XSL Transformation Language -Using XSLT style sheets

- XSLT provides the mechanism for **converting an XML document to another format**. This is accomplished **by applying an XSLT style sheet** to the XML document.
- The **style sheet contains conversion rules** for accessing and transforming the input XML document to a different output format.
- An **XSLT processor** is responsible for **applying the rules defined in the style sheet** to **the input XML document**.

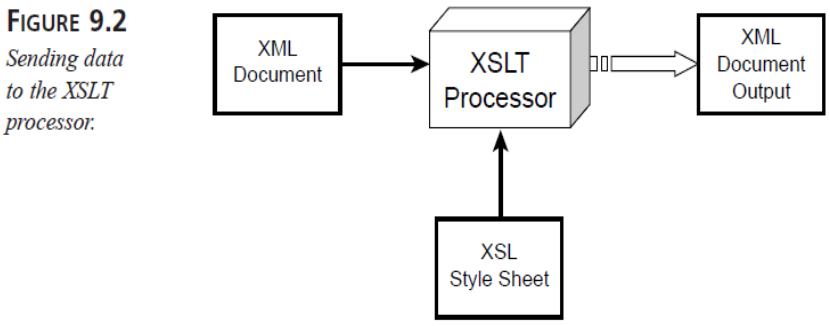


FIGURE 9.2
Sending data to the XSLT processor.

Example:

The figure shows the output of the XML document representation in the web browser.

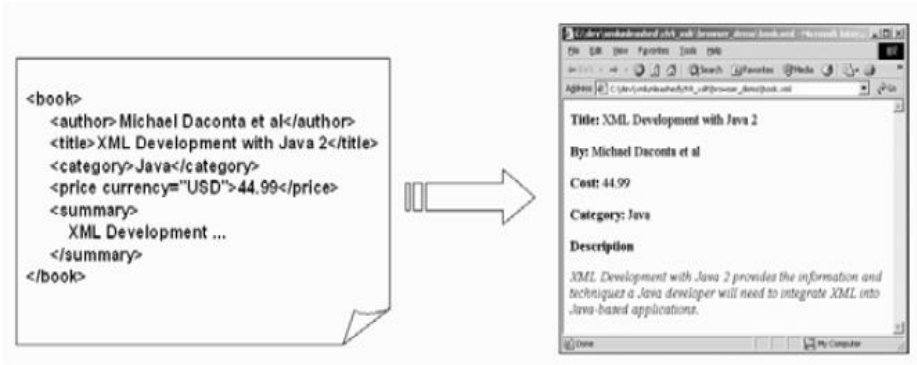


FIGURE 9.3
Converting `book.xml` *to an HTML table.*

Emp.xml

```
<?xml version = "1.0" ?>
<?xml-stylesheet href="emp.xsl" type="text/xsl"?>
<Employee-Detail>
<Employee>
<Emp_Id> 11032 </Emp_Id>
<Emp_Name> Hari </Emp_Name>
<Emp_E-mail> harideivasigamani@gmail.com </Emp_E-mail>
</Employee>
<Employee>
<Emp_Id> 11022 </Emp_Id>
<Emp_Name> Ashok Kumar</Emp_Name>
<Emp_E-mail> ashokkumar782@gmail.com </Emp_E-mail>
</Employee>
<Employee>
</Employee-Detail>
```

We will apply the style sheet in a client-side Web browser. The XML document makes a reference to a style sheet using the following code:

<?xml-stylesheet type="text/xsl" href="book_view.xsl"?>

Emp.xsl

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<title>XSLT Style Sheet</title>
<body>
<h1><p align="center">Employee Details</p></h1>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>
<xsl:template match="Employee-Detail">
<table border="2" width="50%" align="center">
<tr bgcolor="LIGHTBLUE">
<td><b>Emp_Id</b></td>
<td><b>Emp_Name</b></td>
<td><b>Emp_E-mail</b></td>
</tr>
<xsl:for-each select="Employee">
<tr>
<td><i><xsl:value-of select="Emp_Id"/></i></td>
<td><xsl:value-of select="Emp_Name"/></td>
<td><xsl:value-of select="Emp_E-mail"/></td>
</tr>
</xsl:for-each>
</table>
</xsl:template>
</xsl:stylesheet>
```

Output:

Employee Details

Emp_Id	Emp_Name	Emp_E-mail
11032	Hari	harideivasigamani@gmail.com
11022	Ashok Kumar	ashokkumar782@gmail.com
11011	Elavarasan	ela@pec.in

In the transformation process, XSLT uses XPath to define parts of the source document that should match one or more predefined templates. When a match is

found, XSLT will transform the matching part of the source document into the result document.

Implementing Server-Side XSLT Processing

A number of server-side technologies are available, including Common Gateway Interface (CGI), ColdFusion, Hypertext Processor (PHP),.The server-side processing can be done with Microsoft's Active Server Pages (ASP) and Sun Microsystems' JavaServer Pages (JSP).

Advanced Features of XSLT

Looping

The XSLT element **<xsl:for-each>** is used for **looping through a list of elements**. This is very useful when you have a collection of related items and you'd like to process them in a sequential fashion.

Syntax

```
<xsl:for-each select=node-set-expression>
<!-- content -->
</xsl:for-each>
```

Sorting

In XSLT, the **<xsl:sort>** element is used for **sorting the XML data**. It is possible to sort based on a single key or multiple keys.

Syntax

```
<xsl:sort
select = string-expression
order = { "ascending" | "descending" }
data-type = { "text" | "number" }
case-order = {"upper-first" | "lower-first" }
lang = { nmtoken } />
```

The <xsl:sort> element is used in conjunction with the <xsl:for-each> element.

For example, the following code snippet sorts the book titles in alphabetical order:

```
<!-- Sort by the book title -->
<xsl:for-each select="booklist/book" >
<xsl:sort select="title" />
<!-- insert table rows and table data -->
```

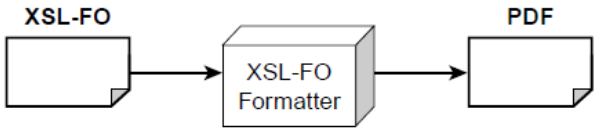
</xsl:for-each>

XSL Formatting Objects

XSL-FO was designed to **assist with the printing and displaying of XML data**. The **main importance is on the document layout and structure**. This includes the dimensions of the output document, including page headers, footers, and margins.

XSL-FO also allows the developer to define the formatting rules for the content, such as font, style, color, and positioning. XSL-FO is a sophisticated version of Cascading Style Sheets (CSS).

XSL-FO documents are well-formed XML documents. An XSL-FO formatting engine processes XSL-FO documents.



Two techniques for creating XSL-FO documents.

- Develop the XSL-FO file with the included data.
- Dynamically create the XSL-FO file using an XSLT translation.

Basic Document Structure

An XML-FO document follows the syntax rules of XML; as a result, it is well formed.XSL-FO elements use the following namespace:

http://www.w3.org/1999/XSL/Format

The following code snippet shows the **basic document setup for XSL-FO:**

```
<?xml version="1.0" encoding="utf-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<!-- layout master set -->
<!-- page masters: size and layout -->
<!-- page sequences and content -->
</fo:root>
```

The element <fo:root> is the root element for the XSL-FO document. An XSL-FO document can contain the following components:

- Page master
- Page master set

- Page sequences

Page Master: <fo:page-master> The page master describes the page size and layout.

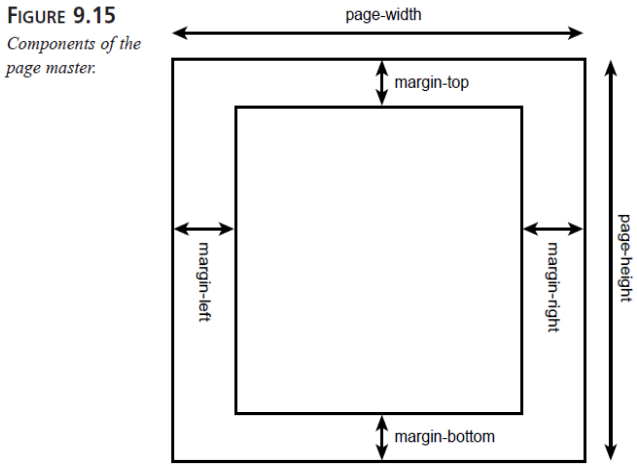
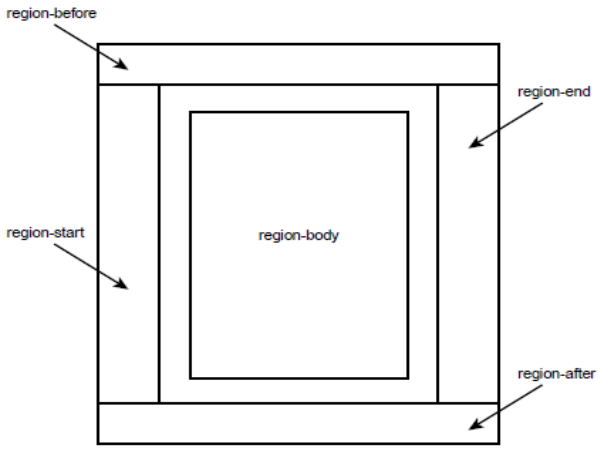


FIGURE 9.15
Components of the page master.

The <fo:simple-page-master> element can also be used to describe an A4 letter (height 210 mm and width 297 mm):

```
<fo:simple-page-master master-name=”A4-example”
page-height=”210mm”
page-width=”297mm”
margin-top=”0.5in”
margin-bottom=”0.5in”
margin-left=”0.5in”
margin-right=”0.5in”>
</fo:simple-page-master>
```

Each page is divided into five regions. Regions serve as containers for the document content.



During the definition of a page master, you specify the size of the regions using the following elements:

- <fo:region-before>
- <fo:region-after>
- <fo:region-body>
- <fo:region-start>
- <fo:region-end>

The **region-before** and **region-after** areas are commonly used for **page headers and footers**. The **region-body** area is the **center of the page** and contains the main content. The **region-start** and **region-end** sections are commonly used for **left and right sidebars,** respectively.

Page Master Set: <fo:page-master-set>

A document can be **composed of multiple pages**, each with its own dimensions. The page master set refers to the collection of page masters.

Page Sequences: <fo:page-sequence>

A page sequence **defines a series of printed pages.** Each page sequence refers to a page master for its dimensions. The page sequence contains the actual content for the document. The <fo:page-sequence> element contains

- **<fo:static-content> -** This element is **used for page headers and footers**.
  Example: We can define a header for the company name and page number, and this information will appear on every page.

- **<fo:flow> elements-** This element **contains a collection of text blocks**. It is similar to a collection of paragraphs. A body of text is defined using this.
  - The **<fo:block>** element is a **child element of <fo:flow>.** It **contains free-flowing text** that will wrap to the next line in a document if it overflows.

Example:

```
<?xml version=”1.0” encoding=”utf-8”?>
<fo:root xmlns:fo=”http://www.w3.org/1999/XSL/Format”>
<!-- layout master set -->
<fo:layout-master-set>
<!-- page masters: size and layout -->
<fo:simple-page-master master-name=”simple”
        page-height=”11in”
        page-width=”8.5in”
        margin-top=”1in”
        margin-bottom=”1in”
        margin-left=”1.25in”
        margin-right=”1.25in”>
```

```
<fo:region-body margin-top="0.5in"/>
<fo:region-before extent="3cm"/>
<fo:region-after extent="1.5cm"/>
</fo:simple-page-master>
</fo:layout-master-set>
<!-- page sequences and content -->
<fo:page-sequence master-name="simple">
<fo:flow flow-name="xsl-region-body">
<!-- this defines a level 1 heading with orange background -->
<fo:block font-size="18pt"
font-family="sans-serif"
line-height="24pt"
space-after.optimum="15pt"
background-color="orange"
color="white"
text-align="center"
padding-top="3pt">
Ez Books Online
</fo:block>
<!-- Paragraph that contains info about the company -->
<fo:block font-size="12pt"
font-family="sans-serif"
line-height="15pt"
space-after.optimum="14pt"
text-align="justify">
Welcome to Ez Books Online, the world's smallest online book store.
Our company's mission is to sell books on Java, Thrillers and Romance.
We have something for everyone...so we think. Feel free to browse our
catalog and if you find a book of interest then send us an e-mail.
Thanks for visiting!
</fo:block>
</fo:flow>
</fo:page-sequence>
```

</fo:root>

Using Graphics

XSL-FO also allows for the insertion of external graphic images. The graphic formats supported are dependent on the XSL-FO formatting engine. The Apache-FOP formatting engine supports the popular graphics formats: GIF, JPEG, and BMP.

The following code fragment inserts the image smiley.jpg:

<fo:block text-align="center">
<fo:external-graphic src="smiley.jpg" width="200px" height="200px"/>
</fo:block>

Tables

XSL-FO has rich support for structuring tabular data.

Comparing HTML Table Elements and XSL-FO Table Elements

HTML Element	**XSL-FO Element**
TABLE	fo:table-and-caption
Not applicable	fo:table
CAPTION	fo:table-caption
COL	fo:table-column
COLGROUP	Not applicable
TH	fo:table-header
TBODY	fo:table-body
TFOOT	fo:table-footer
TD	fo:table-cell
TR	fo:table-row

Code fragment to define the basic structure of the table:

<fo:table>
<!-- define column widths -->
<fo:table-column column-width="120pt"/>
<fo:table-column column-width="200pt"/>
<fo:table-column column-width="80pt"/>

```
<fo:table-header>
<fo:table-row>
<fo:table-cell>
<fo:block font-weight=”bold”>Author</fo:block>
</fo:table-cell>
<fo:table-cell>
<fo:block font-weight=”bold”>Title</fo:block>
</fo:table-cell>
<fo:table-cell>
<fo:block font-weight=”bold”>Price (USD)</fo:block>
</fo:table-cell>
</fo:table-row>
</fo:table-header>
<!-- insert table body and rows here -->
</fo:table>
```

Generating XSL-FO Tables Using XSLT

If we wanted to list 500 books. The document would be extremely large. The file, booklist.xml, contains a list of the books. We can develop an XSL style sheet that will automatically construct the XSL-FO document.

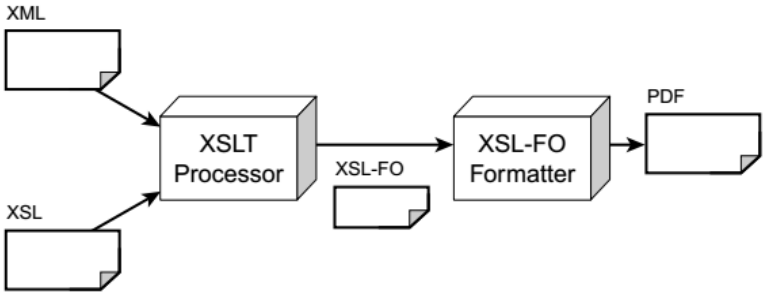


Fig. Generating XSL-FO tables with XSLT.

14. Give a brief note on modeling the database in XML

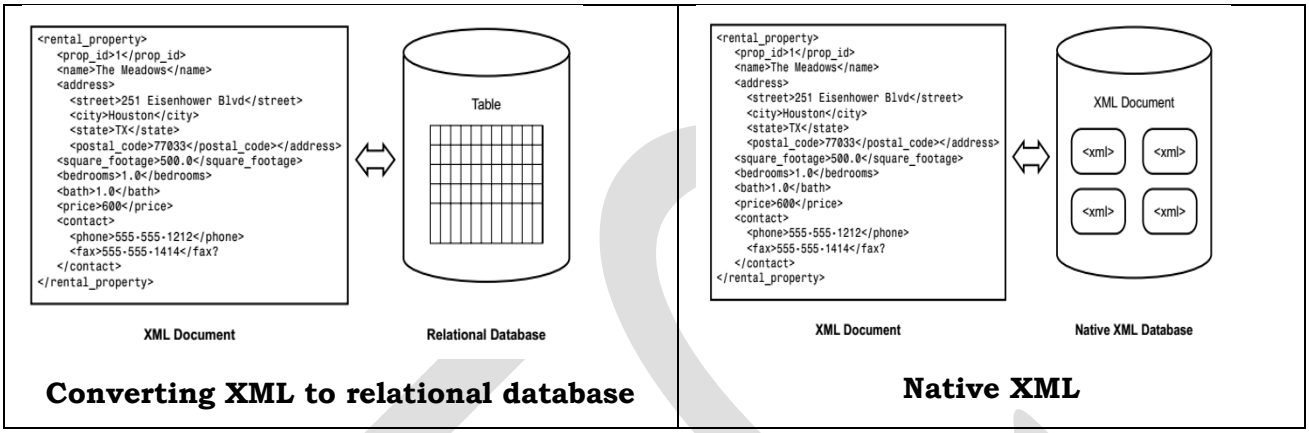
Integrating XML with relational database:

This can be performed by the following methods namely:

- **Database mapping** -provides a mapping between the XML document and

the database fields. The system dynamically converts SQL result sets to XML documents

- **Native xml support-**It actually stores the XML data in the document in its native format. Each product uses its own proprietary serialization technique to store the data.



Converting XML to relational database **Native XML**

Modeling the database in XML:

When we model a database, we provide an external representation of the database contents. We can develop a servlet that uses JDBC. The servlet will make the appropriate query to the database and use Java Database Connectivity (JDBC) API result set metadata to create the elements.

We can use the XML data binding features of Java Architecture for XML Binding (JAXB). JAXB provides a framework for representing XML documents as Java objects.

Marshalling and Unmarshalling in JAXB

In the JAXB framework, we can parse **XML documents into a suitable Java object.** This technique is referred to as **unmarshaling**. The JAXB framework also provides the capability to **generate XML documents from Java objects**, which is referred to as **marshaling.**

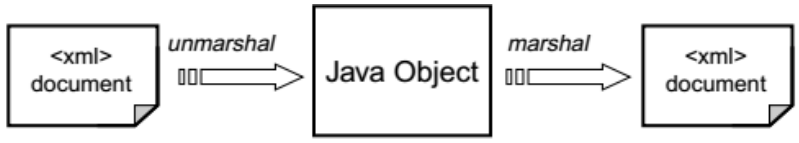


FIGURE 10.4
JAXB marshaling and unmarshaling.

Properties of JAXB

- Using JAXB, an application can parse an XML document by simply unmarshaling the data from an input stream.

- JAXB allows us to define Java objects that map to XML documents, so we can easily retrieve data.
- The JAXB framework also ensures the type safety of the data.

JAXB binding schema:

The JAXB binding schema contains instructions on how to bind the XML schema to a Java class. The steps followed are:

1. Review the database schema.
2. Construct the desired XML document.
3. Define a schema for the XML document.
4. Create the JAXB binding schema.
5. Generate the JAXB classes based on the schema.
6. Develop a Data Access Object (DAO).
7. Develop a servlet for HTTP access.

Reviewing the Database Schema:

The database schema for employee details is given below:

Field	**Type**
empid	NUMBER
name	VARCHAR2
designation	VARCHAR2
department	VARCHAR2
salary	NUMBER

Constructing the Desired XML Document

The desired output XML document is employee details. XML document provides a custom mapping of the database fields to XML element names.

Database Field	**XML Element Name**
empid	<eid>
name	<ename>
designation	<edesgn>
dept	<edept>
salary	<esalary>

An employee detail is described with a root element of <employee>, as shown in the following code:

```
<employee>
<eid>101</eid>
```

<ename>Mathew</ename>

<edesgn>Senior Executive</edesgn>

<edept>Production </edept>

<esalary>100000<esalary>

</employee>

We can also create a collection of employee details . This collection can be modeled using a <employee_list> element, as shown here:

<employee_list>

< employee _property> … </ employee _property>

< employee _property> … </ employee _property>

… …

</ employee _list>

Defining a Schema for the XML Document

Based on the desired document format, we can create a schema definition. here we define the Document Type Definition (DTD). The DTD schema format was chosen because JAXB 1.0 (early access) only supports DTDs.

employeedetails.dtd

```
<!ELEMENT employee_list (employee)*>
<!ELEMENT employee (eid,ename,edesgn,edept)>
<!ELEMENT eid (#PCDATA)>
<!ELEMENT ename (#PCDATA)>
<!ELEMENT edesgn (#PCDATA)>
<!ELEMENT edept (#PCDATA)>
<!ELEMENT esalary (#PCDATA)>
```

Creating the JAXB Binding Schema

The JAXB binding schema is an XML document that contains instructions on how to bind a DTD to a Java class. Using the JAXB binding schema, we can define the names of the generated Java classes, map element names to specific properties in the Java class, and provide the mapping rules for attributes.

employeedetails.xjs

```
<?xml version=”1.0” encoding=”ISO-8859-1” ?>
<!DOCTYPE xml-java-binding-schema SYSTEM
➥”http://java.sun.com/dtd/jaxb/1.0-ea/xjs.dtd”>
<xml-java-binding-schema version=”1.0-ea”>
<options package=”xmlunleashed.ch10.jaxb”/>
```

<element name=”employee_list” type=”class” root=”true”>
<content property=”list”/>
</element>
<element name=”salary” type=”value” convert=”double”/>
</xml-java-binding-schema>

Generating the JAXB Classes Based on Schemas:

JAXB provides a schema compiler for generating the Java source files. The schema compiler takes as input the DTD and the JAXB binding schema.

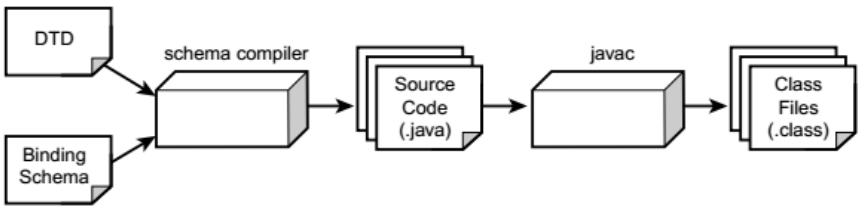


FIGURE 10.6
Generating Java classes with the JAXB compiler.

Now pass the DTD (employeedetails.dtd) and binding schema (employeedetails.xjs) to the JAXB schema compiler with the xjc command.

java com.sun.tools.xjc.Main rental_property.dtd rental_property.xjs -d source_code

This command generates source code in the source_code directory. The following files are generated:

- EmployeeList.java. This file models the <employee_list> element.
- Employee.java. This file models the <employee> element.

The Unified Modeling Language (UML) diagram are also generated for the Java classes.Using the default schema-binding definition, the JAXB schema compiler generates a property in the Java class for each XML element. In the event the XML element contains subelements, the schema compiler will create a new class.

The partial source code for Employee.java is shown below:

import javax.xml.bind.*;
import javax.xml.bind.Validator;
import javax.xml.marshal.XMLScanner;
import javax.xml.marshal.XMLWriter;
import java.io.IOException;
import java.io.InputStream;
public class Employee extends MarshallableObject implements Element

```
{
private String _empid;
private String  _ename;
private String _edesgn;
private String _edept;
private double_esalary;

public String getempid()
{
return _empid;
}
public void setempid(String _empid)
{
this._empid = _empid;
if (_empid == null)
 {
invalidate();
}
}
}
public void validateThis() throws LocalValidationException
{
… …
}
public void marshal(Marshaller m)throws IOException
{
// code to output the XML document
}
public void unmarshal(Unmarshaller u)throws UnmarshalException
{
// code to read in the XML document
}
```

Developing a Data Access Object (DAO):

A Data Access Object (DAO) provides access to the backend database. The goal of the DAO design pattern is to provide a higher level of abstraction for database access. The DAO provides access to the backend database via public methods. The DAO converts a result set to a collection of objects. The objects model the data stored in the database.

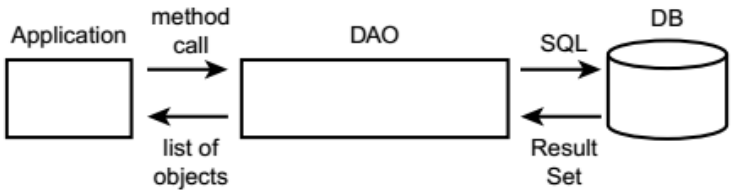


FIGURE 10.8
Data Access Object design pattern.

By using a DAO, the implementation details of the database are hidden from the application clients. The implementation details include the database schema and database vendor.

Benefit of using the DAO

- Improved application maintenance.
- If the database schema changes, such as a column name modified, update only the DAO. No modifications are required to the client programs.
- If change need for database implementation from Sybase to Oracle, modifications are only required to the DAO. The clients can continue to use the DAO without any modification.
- The DAO design pattern is widely used in the industry .

The partial code for **EmployeeDAO.java.**

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class EmployeeDAO
{
```

```
protected Connection myConn;
public EmployeeDAO(String driverName, String dbUrl,String user, String pass) throws
DAOException
{
try
{
// Load the driver
log(“Loading driver: “ + driverName);
Class.forName(driverName);
// Get a connection
log(“Connecting to the database: “ + dbUrl);
log(“User id: “ + user);
myConn = DriverManager.getConnection (dbUrl, user, pass);
log(“DB connection successful at “ + new java.util.Date());
}
catch (Exception exc)
{
throw new DAOException(exc);
}
}
//Get a list of employee properties from the database
public EmployeeList getEmployeeProperties() throws DAOException {
EmployeeList  elist = new EmployeeList ();
java.util.List theList = elist.getList();
try
{
Statement myStmt = myConn.createStatement();
String rentalSql = “SELECT empid, name, designation, department,salary “FROM
employee”;
ResultSet myRs = myStmt.executeQuery(employeeSql);
EmployeeProperty temp= null;
// build a collection of JAXB EmployeeProperty objects
while (myRs.next())
 {
temp= createEmployeeProperty(myRs);
theList.add(temp);
}
// be sure to validate the new list
```

```
theList.validate();
myRs.close();
myStmt.close();
}
catch (Exception exc) {
throw new DAOException(exc);
}
return theRentalPropertyList;
}
/**
* Create a JAXB RentalProperty object based on the result set.
* This method provides the mapping between database schema and object
*/
protected EmployeeProperty createEmployeeProperty(ResultSet theRs)throws
DAOException
{
EmployeeProperty emp= new EmployeeProperty();
try {
emp.setempId(theRs.getString("empid"));
emp.setName(theRs.getString("name"));
emp.setDesignation(theRs.getString("designation"));
emp.setDepartment(theRs.getString("department"));
emp.setSalary(theRs.getString("salary"));
}
catch (SQLException exc) {
throw new DAOException(exc);
}
return theProperty;
}
protected void log(Object message) {
System.out.println("EmployeeDAO: " + message);
}
}
```

Developing a Servlet for HTTP Access

We need to provide an HTTP interface for EmployeeDAO so that a Web browser can interact with our system. Java servlets provides support for the HTTP protocol.

The servlet is responsible for creating an instance of EmployeeDAO. The servlet reads

JDBC parameters from the web.xml configuration file and constructs EmployeeDAO accordingly.

web.xml file

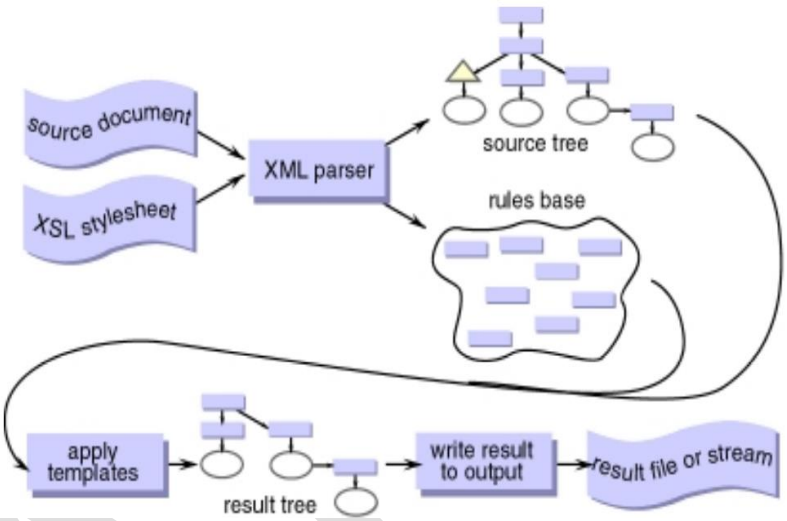
```
<servlet>
<servlet-name>EmployeeServlet</servlet-name>
<servlet-class> EmployeeXMLServlet</servlet-class>
<init-param>
<param-name>driverName</param-name>
<param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</init-param>
<init-param>
<param-name>dbUrl</param-name>
<param-value>jdbc:odbc:RentalPropertyDSN</param-value>
</init-param>
<init-param>
<param-name>user</param-name>
<param-value>test</param-value>
</init-param>
<init-param>
<param-name>pass</param-name>
<param-value>test</param-value>
</init-param>
<load-on-startup/>
</servlet>
```

The partial code for Servlet is given below:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException
{
        ServletOutputStream out = null;
        EmployeeList theList = null;
        try {
        // Set the content type to text/xml
                response.setContentType("text/xml");
        // Retrieve the servlet output stream
                out = response.getOutputStream();
```

```
        // Retrieve a list of rental properties
                theList = myEmployeeDAO.getEmployeeProperties();
        // Marshal the list as an XML document
                theList.marshal(out);
}
```

15. With an example show how XSLT transform XML to HTML



Transforming XML data using XSLT

page.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="page.xsl"?>
<PAGE>
        <HEADING>page heading</HEADING>
        <ARTICLE>
                <TITLE>article title</TITLE>
                <DESCRIPTION>article description</DESCRIPTION>
        </ARTICLE>
        <ASIDE>
                <TITLE>side widget bar</TITLE>
                <ITEM>sidebar item</ITEM>
        </ASIDE>
        <FOOTER>page footer</FOOTER>
</PAGE>
```

Page.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html><body>
      <xsl:apply-templates/>
    </body></html>
  </xsl:template>
  <xsl:template match="/PAGE/HEADING">
    <h1 align="center"> <xsl:apply-templates/> </h1>
  </xsl:template>
  <xsl:template match="/PAGE/ARTICLE">
    <div style="float:left;width:70%;"><xsl:apply-templates/> </div>
  </xsl:template>
  <xsl:template match="/PAGE/ARTICLE/TITLE">
    <h3> <xsl:apply-templates/> </h3>
  </xsl:template>
  <xsl:template match="/PAGE/ARTICLE/DESCRIPTION">
    <p> <xsl:apply-templates/> </p>
  </xsl:template>
  <xsl:template match="/PAGE/ASIDE/TITLE">
    <div style="float:left;width:30%;"><h3> <xsl:apply-templates/> </h3></div>
  </xsl:template>
  <xsl:template match="ITEM">
    <p> <xsl:apply-templates/> </p>
  </xsl:template>
  <xsl:template match="/PAGE/FOOTER">
    <div style="clear:both;"></div>
    <h1 align="center"> <xsl:apply-templates/> </h1>
  </xsl:template>
</xsl:stylesheet>
```

C# console application for accessing the XML

Transform.cs

```
using System;
using System.Xml;
using System.Xml.Xsl;
```

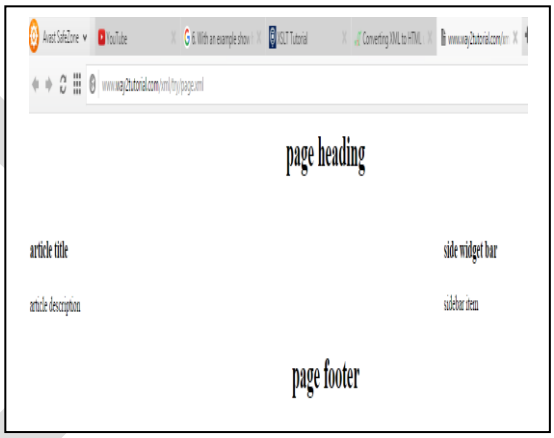
```
namespace XSLTransform{
   class myclass{
      static void Main(string[] args){
         XslTransform myXslTransform;
         myXslTransform = new XslTransform();
         myXslTransform.Load("page.xsl");
         myXslTransform.Transform("page.xml");
      }
   }
}
```

Output:

```
<html>
<body>
        <h1 align="center">page heading</h1>
        <div style="float:left;width:70%;">
                <h3>article title</h3>
                <p>article description</p>
        </div>

        <div style="float:left;width:30%;">
                <h3>side widget bar</h3>
                <p>sidebar item</p>
        </div>

        <div style="clear:both;"></div>
        <h1 align="center">page footer</h1>
</body>
</html>
```



UNIT – II SERVICE ORIENTED ARCHITECTURE (SOA) BASICS

Characteristics of SOA, Comparing SOA with Client-Server and Distributed architectures – Benefits of SOA -- Principles of Service orientation – Service layers.

PART - B

1. Write briefly on Characteristics of Contemporary SOA. (MAY/JUN 2012)

Definition:

Contemporary SOA is an **extended variation of service-oriented architecture** which builds **increasingly powerful XML and Web services** support into current technology platforms.

Common characteristics of Contemporary SOA

Contemporary SOA builds upon the primitive SOA model by leveraging industry and technology advancements with the following primary characteristics

- Contemporary SOA is at the core of the services-oriented computing platform.
- Contemporary SOA increase quality of service.
- Contemporary SOA is fundamentally autonomous.
- Contemporary SOA is based on open standards.
- Contemporary SOA supports vendor diversity.
- Contemporary SOA fosters intrinsic interoperability.
- Contemporary SOA promotes discovery.
- Contemporary SOA promotes federation.
- Contemporary SOA promotes architectural compos ability.
- Contemporary SOA fosters inherent reusability.
- Contemporary SOA emphasizes extensibility.
- Contemporary SOA supports a service-oriented business modeling paradigm.
- Contemporary SOA implements layers of abstraction.
- Contemporary SOA promotes loose coupling throughout the enterprise.

- Contemporary SOA Promotes organizational agility.
- Contemporary SOA is a building block.
- Contemporary SOA is an evolution.
- Contemporary SOA is still maturing.
- Contemporary SOA is an achievable ideal.

a. Contemporary SOA is at the core of the services-oriented computing platform.

- When a product, design, or technology is prefixed with "SOA" it is something that was created in **support of an architecture based on service-orientation** principles.
- Contemporary SOA represents an architecture that promotes service-orientation through the use of Web services

b. Contemporary SOA increase quality of service.

Contemporary SOA is striving to fill the QoS gaps of the primitive SOA model with the following requirements,

- In a secure manner
- Reliably
- With appropriate performance
- Protecting business integrity
- Executing exception logic in case of failure.

c. Contemporary SOA is fundamentally autonomous.

Autonomous principle

- Represents the ability of a **service to carry out its logic independently** of outside influences.
- **Message-level autonomy-** Messages are "intelligence-heavy" and control the way they are processed by recipients
- Autonomy concept is expanded to solution environment and the enterprise i.e. applications.

Levels of autonomy

- **Runtime autonomy-**represents the **amount of control a service** has over its execution environment at **runtime.**
- **Design-time autonomy-** represents the **amount of governance control** a service owner has over the **service design.**

Primary benefits

- Increased reliability
- Behavioral predictability

d. Contemporary SOA is based on open standards

- Based on open standards, messages travel between services via a set of protocols that is globally standardized and accepted
- Messages 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

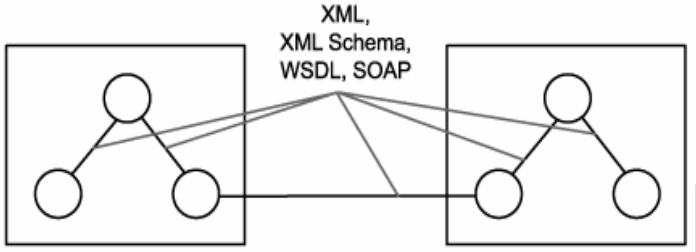


Figure 1. Standard open technologies are used within and outside of solution boundaries.

e. Contemporary SOA supports vendor diversity

- The communications framework bridges the heterogeneity within and between corporations
- Integration technologies encapsulate legacy logic through service adapters.
- Platform neutral communication such as .NET solution J2EE solution.

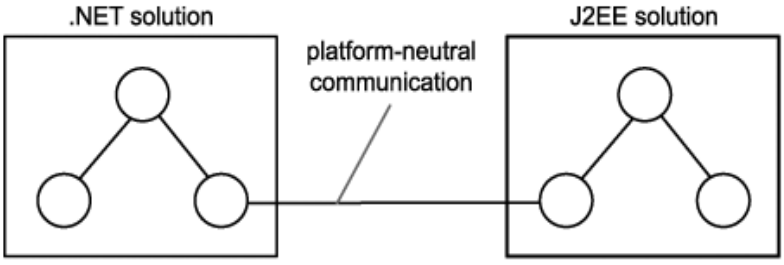


Figure 2. Disparate technology platforms do not prevent service-oriented solutions from interoperating.

f. Contemporary SOA promotes discovery

- Universal description discovers and integration **(UDDI)** provided for service registries.
- Some SOA systems used **UDDI service registry** or directory to **manage service descriptions.**
- Services are supplemented with communication meta data by which they can be effectively discovered and interpreted.
- Store Meta data in a service registry or profile documents

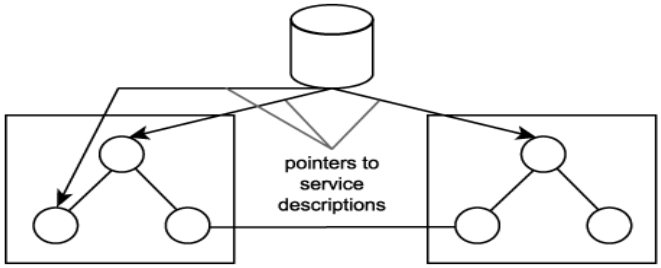


Figure 3. Registries enable a mechanism for the discovery of services.

g. Contemporary SOA fosters intrinsic interoperability

The design characteristics required to facilitate interoperability are

- Standardization
- Scalability
- Behavioral predictability
- Reliability

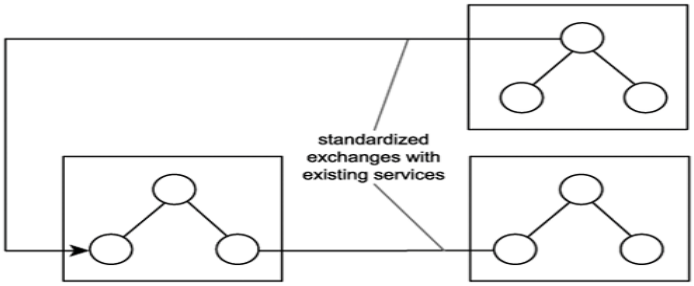


Figure 4. Intrinsically interoperable services enable unforeseen integration opportunities.

h. Contemporary SOA promotes federation

- Establishing and standardizing the **ability to encapsulate legacy and non-legacy application logic** and by exposing it via an open, common standard communications framework.

- Communication channels are all uniform and standardized.

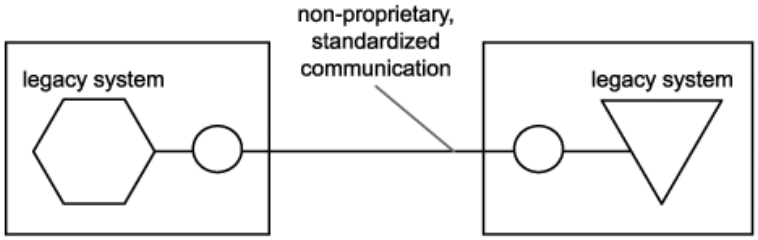


Figure 5. Services enable standardized federation of disparate legacy systems.

i. Contemporary SOA promotes architectural composability

- Supports the automation of flexible, adaptable business process by composing loosely coupled services.
- Flexible service contracts to allow different types of data exchange requirements for similar functions
- Services are effective composition participants, regardless of the size and complexity of the composition.
- Ensures services are able to participate in multiple compositions to solve multiple larger problems-related to reusability principle

Advantages

- Composite applications faster than writing a program from scratch.
- Building new services and application becomes quicker and cheaper .

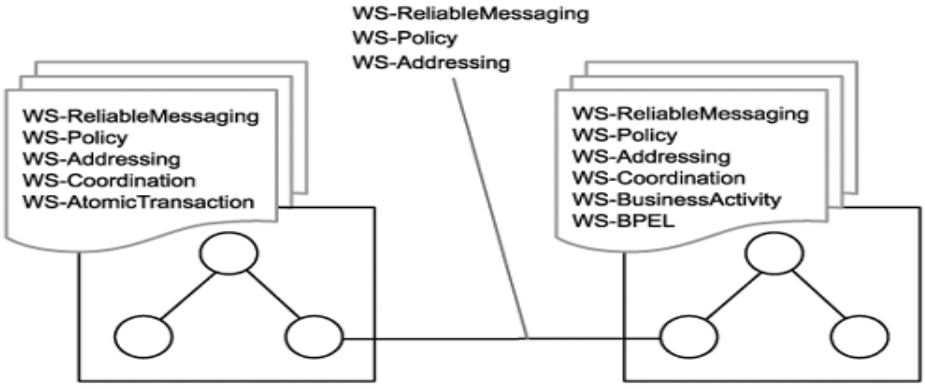


Figure 6. Different solutions can be composed of different extensions and can continue to interoperate as long as they support the common extensions required.

j. 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 contain and express logic and can be positioned as reusable enterprise resources
- **Reusable services** have the following **characteristics:**
    - Defined by an agnostic functional context
    - Logic is highly generic
    - Has a generic and extensible contract
    - Can be accessed concurrently

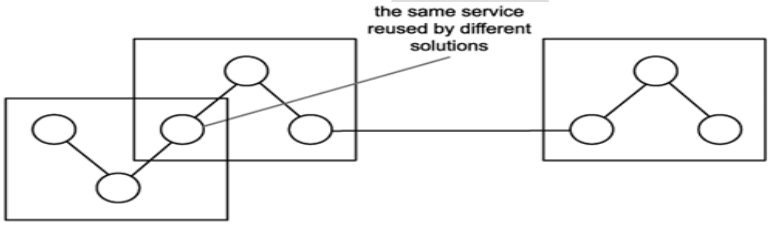


Figure 7. Inherent reuse accommodates unforeseen reuse opportunities.

k. Contemporary SOA emphasizes extensibility

- When encapsulating functionality through a service description, you are encouraged to think beyond a point-to-point solution.
- Extensibility can be achieved less significantly due to loosely coupled relationship fostered among all services.

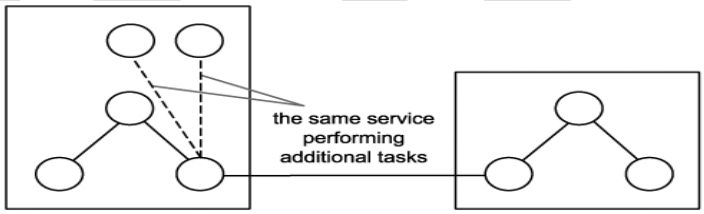


Figure 8. Extensible services can expand functionality with minimal impact.

SOA Definition after discussing these 11 Characteristics:

Contemporary SOA represents an **open, extensible, federated, compostable architecture that promotes service-orientation** and is comprised of autonomous, QOS-capable, vendor diverse, interoperable, discoverable, and potentially reusable services, implemented as web services.

l. Contemporary SOA supports a service-oriented business modeling paradigm

- Partitioning **business logic into services** that can be **composed** has significant implications as to how business processes can be modeled.

- BPM models, entity models and other forms of business intelligence can be accurately represented through coordinated composition of business-centric services.

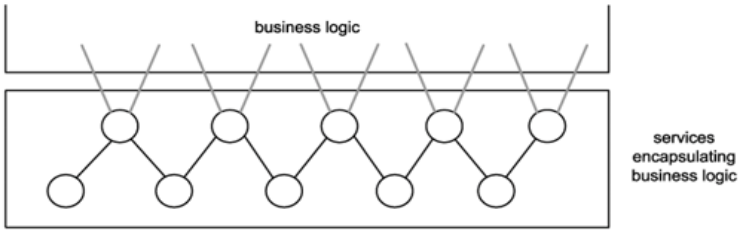


Figure 9. A collection (layer) of services encapsulating business process logic.

m. Contemporary SOA implements layers of abstraction

- SOA introduce layers of **abstraction by positioning services as the sole access points** to a Variety of resources and processing logic.
- The abstraction is targeted at business and application logic and the functionality is offered via the service interfaces.

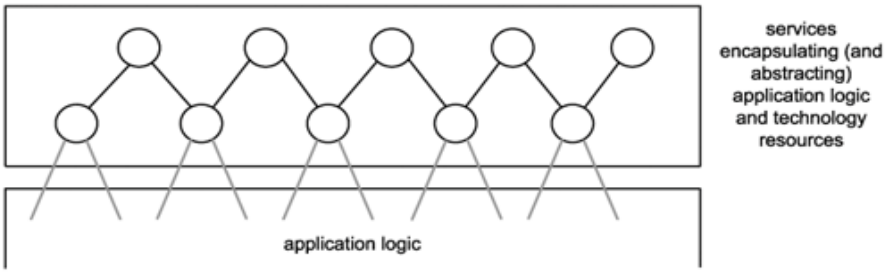


Figure 10. Application logic created with proprietary technology can be abstracted through a dedicated service layer.

n. Contemporary SOA promotes loose coupling throughout the enterprise

- The loose coupling concept is **achieved by implementing standardized service** abstraction layers when service-orientation principles are applied to both business modeling and technical design.
- Each domain are allowed to evolve more independently which result in an better accommodate business and technology-related change-quality environment known as **organizational agility.**

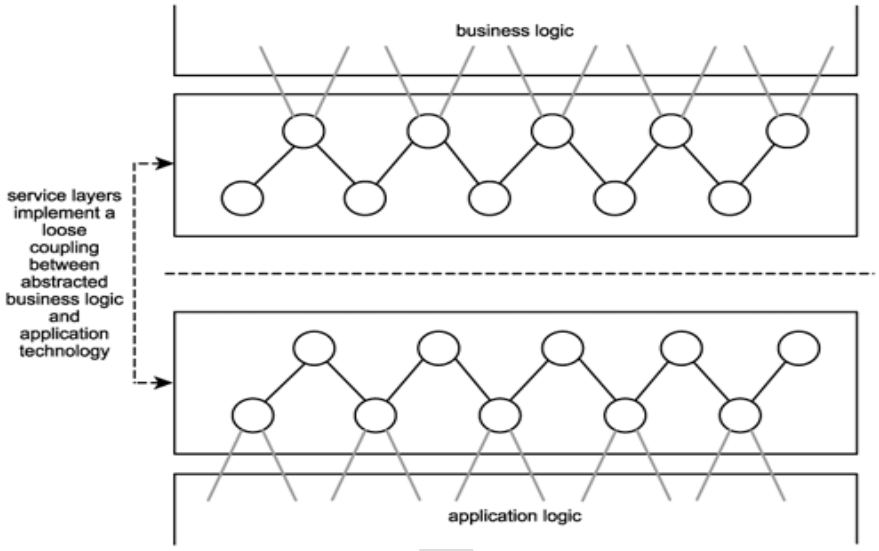


Figure 11. Through the implementation of service layers that abstract business and application logic, the loose coupling paradigm can be applied to the enterprise as a whole.

o. Contemporary SOA Promotes organizational agility

- Organizational agility refers to **efficiency with which an organization can respond to change.**
- High dependency between parts of an enterprise means that changing software is more complicated and expansive
- Leveraging service business representation, service abstraction, and loose coupling promotes agility

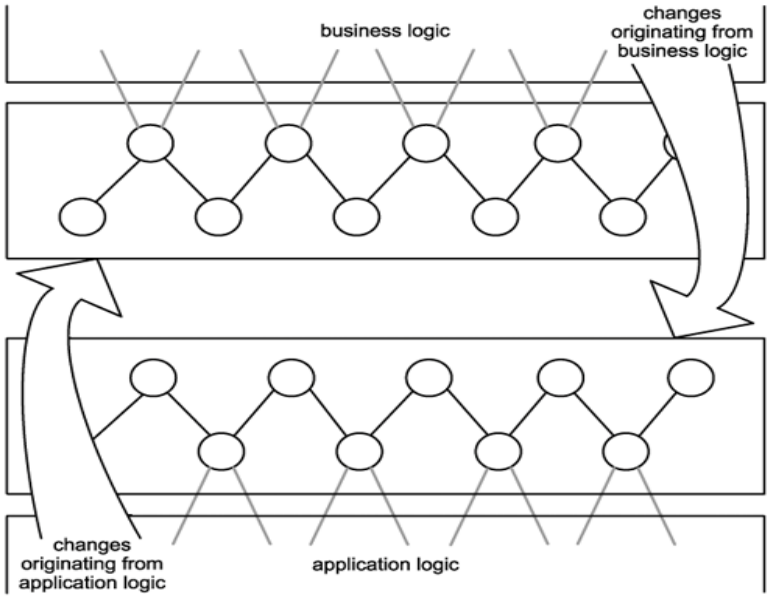


Figure 12. A loosely coupled relationship between business and application technology allows each end to more efficiently respond to changes in the other.

p. 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).
- An SOA consists of services within services, to the point that a solution based on SOA itself is one of many services within an SOE.

Definition of SOA after applying the above characteristics:

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.

q. Contemporary SOA is an evolution

- SOA is a distinct architecture from the previous.
- It is influenced by concepts in service-orientation and web services
- Promotes reuse, encapsulation, componentization, and distribution of application logic like previous technologies

r. Contemporary SOA is still maturing

- Standards organization and vendors are continuing to develop new SOA technologies.
- They are extended to support the creation of enterprise SOA solutions.

s. Contemporary SOA is an achievable ideal

- Many organizations begin with a single application and then begin leveraging service into other applications
- Changing to SOA requires cultural changes in an organization

2. Compare SOA with client-server and distributed internet architectures. (NOV/DEC 2011) (NOV/DEC 2012) (MAY/JUN 2013) (May/June 2014)(Nov/Dec 2014)

SOA vs. client-server Architecture:

Client-server architecture is architecture in which one piece of software requests or receives information from another.

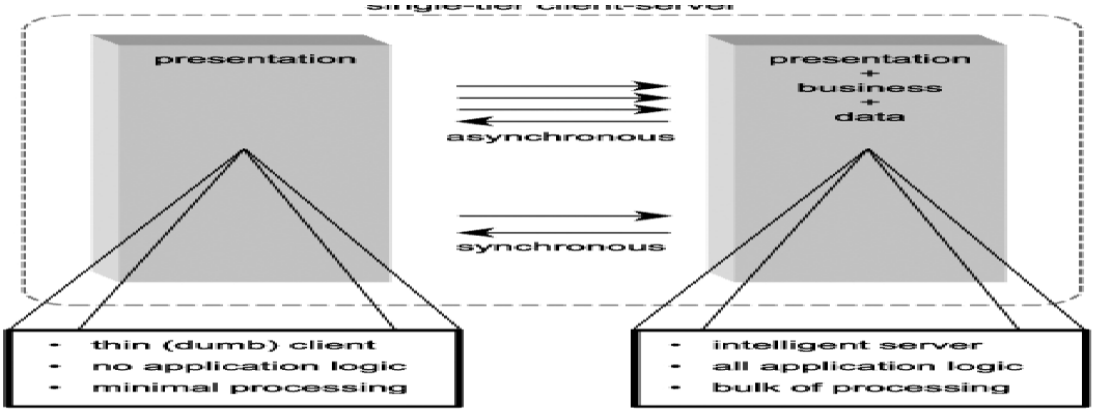
Client-server architecture:

1. Single-tier client-server architecture

Single-tier client-server architecture is an environment in which **bulky mainframe back-ends server served the thin clients.**

Thin –client model:

In a thin client model, all of the application **processing and data management** is carried out on the **server.** The **client** is simply responsible for **running the presentation software.**



Types of communication

- **Synchronous communication** - Allow the client and server to wait for each other to transfer the message. That is, the client will not continue until the server has received the message.
- **Asynchronous communication**- Allow the server to continuously receive message from the client without waiting for the server to respond.

Disadvantage

Places a heavy processing load on both the server and the network.

2.Two-tier client-server architecture

Two-tier client-server architecture consists of multiple fat clients, each with its own connection to a database on a central server.

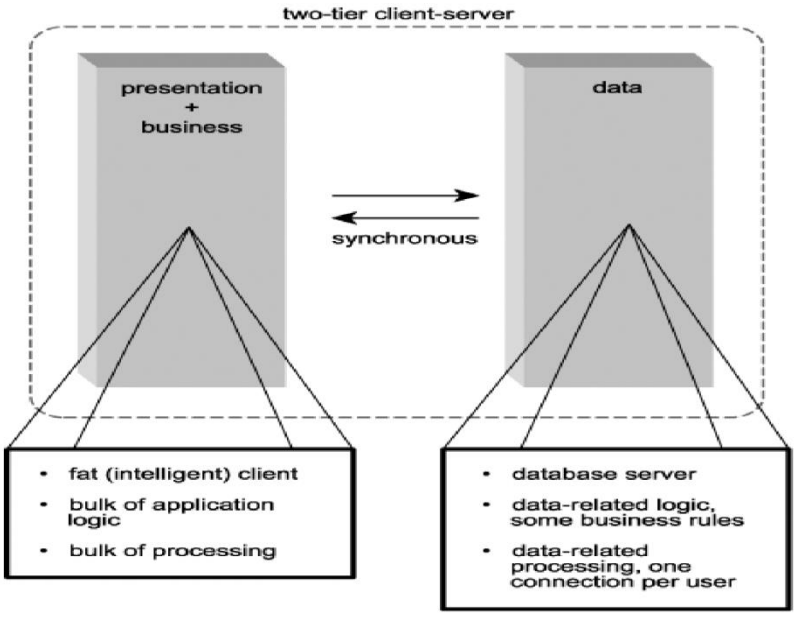
Fat-client model:

- In fat-client model, the **server** is only responsible for **data management.**
- The software on the **client implements** the **application logic** and the interactions with the system user.

Operation of Two-tier client-server architecture

- The client accepts user requests and performs the bulk of application logic that produces database requests and transmits them to the server.

- The server accepts the requests, performs the data access logic, and transmits the results to the client.
- The client accepts the results and presents them to the end user.



Characteristics of Two-tier client-server architecture

The primary characteristics of the two tier client server architecture is given below which is compared to SOA

- Application logic
- Application architecture
- Technology
- Security
- Administration

Application logic:

Client Server Architecture - The application or business logic either resides on the client or on the database server in the form of stored procedures.

SOA - The processing logic is partitioned into autonomous units which facilitate design qualities, future compos ability and reusability.

Application architecture

Client server architecture

- Client is responsible for the bulk of processing

- 80/20 ratio is used as a rule of thumb.
- Communication is predictably synchronous
- Client establish its own database connection-persistent
- Client –side executables are fully stateful
- Client is independently responsible for its actions; server doesn't track set of clients or ensure that cached data stays up to date.
- Consume a steady chunk of PC memory
- All available resources are offered to the application

SOA:

- Processing in SOA is highly distributed.
- Communication between services and requestor can be synchronous or asynchronous
- SOA provides stateless and autonomous nature or services
- Server tracks its client, takes actions to keep their cached states "current ". Client can trust its cached data. Further processing is made easier by reducing the need for runtime caching of state information

Technology

Client server architecture

Front end

- 4GL programming languages, such as Visual Basic and power builder, is used
    - Provides the ability to create aesthetically rich and more interactive user interfaces
- Traditional 3GL languages, such as c++, were also still used,
    - Required for rigid performance

Back end

- Database vendors, such as Oracle, Informix, IBM, Sybase, and Microsoft, provided robust RDBMSs that could manage multiple connections.

SOA

Front end

- Newer versions of older programming languages, such as Visual Basic, are used to create web services.

- Contemporary SOA established the XML data representation architecture, along with a SOAP messaging framework, and service architecture comprised of the ever- expanding web services platform.

Back end

- Relational database are used.

Security

Client server architecture

- Client-server security is simple
- Security is controlled within the client executable
- Corporate data is protected via a single point of authentication, establishing a single connection between client and server

SOA

- SOA security is complex with respect to degree of security measure.

Administration

Client server architecture

- Administration process is highly burden
- Maintenance costs are large associated with the distributed and maintenance of application logic across user workstations.

SOA

- Administration process is simple and flexible.
- Distributed back-end accommodate scalability for application and database servers

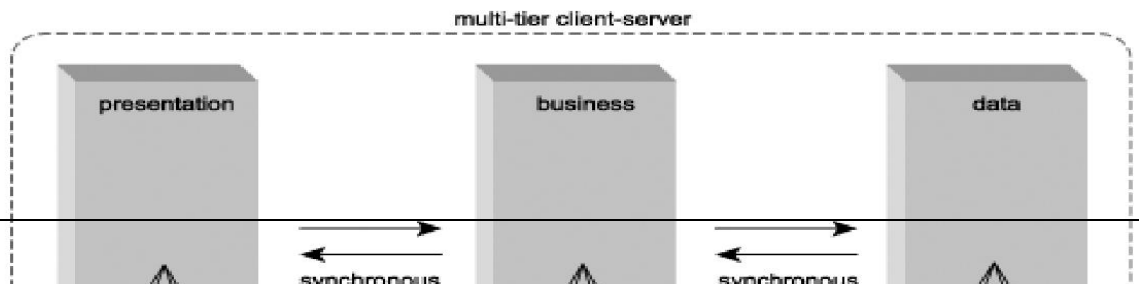
SOA vs. distributed Internet Architecture

SOA is viewed as a form of distributed Internet Architecture because the previous types of distributed Architecture are also designed as SOAs.

Distributed Internet Architecture

1. Multi-tier client-server Architecture

Multi-tier architecture (often referred to as n-tier architecture) is a client-server architecture in which the **presentation**, the **application processing**, and the **data management** are logically **separate processes.**



RPC

Client-server remote procedure call (RPC) connections are used for **remote communication** between components residing on the **client workstations and servers.**

Advantages

- **Better load balancing:**
  - More evenly distributed processing.(e.g., application logic distributed between several servers.)
- **More scalable:**
  - Only servers experiencing high demand need be upgraded.
  - Multiple concurrent requests are processed

Disadvantages

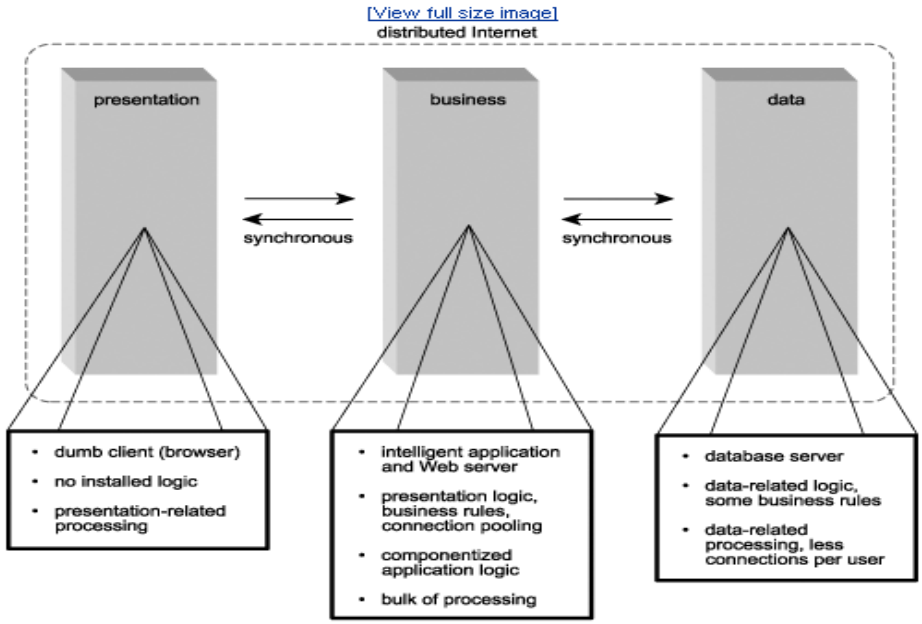
- **Heavily loaded network:**
  - More distributed processing necessities more data exchanges
  - Difficult to program and test due to increased complexity.

2. Distributed Internet architecture

- In the mid-to-late 90s, the multi-tiered client-server environment incorporates Internet technology.
- Custom **software client** component are **replaced** with the **browser.**

- A new **physical tier, web server,** is introduced.
- **Replaces RPC** protocols **by HTTP** for client and server **communication**
- In the middle 2000s, it become the computing platform for custom developed enterprise solutions

Figure 4.5. A typical distributed Internet architecture.



Issues

The issues that are raised in the client-server and the distributed Internet architecture comparisons are discussed in a comparison between multi-tier client-server and SOA.

- Application logic
- Application processing
- Technology
- Security
- Administration

Application Logic

Distributed Internet architecture

- All **application logic** is placed on the **server side** and client-side scripts are downloaded from the web server upon request.
- Components are tightly coupled
    - Little processing is wasted to locate component at runtime

- Very difficult to alter the network after implementation

- Services exchange information uses RPC-style message structures which are accompanied by a wide range of meta information, processing instructions, and policy rules.
- Reusability is strictly allowed.

SOA

- **Provider logic** resides on the **server** end where it is broken down into separate units.
- Components are **loosely coupled.**
- Support a composition model, which aggregate assemblies which are used for reuse and extensibility.
- The messaging framework used by SOA service for exchanging information is more sophisticated, bulker, and tends to result in less individual transmissions.
- **Reusability** and **cross-application interoperability** is done on a deep level by promoting the creation of solution-agnostic services.

Application Processing

Distributed Internet Architecture

Distributed Internet Architecture promotes the use of proprietary communication protocols(DCOM,CORBA)

SOA

- SOA relies on **message-based communication**
- Message use **serialization, transmission, de-serialization of SOAP message** containing XML payloads
- **RPC communication** is **faster than SOAP** and SOAP processing overhead is a significant design issue
- 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

Distributed Internet Architecture now includes **XML data representation**

SOA

XML and web services are optional for Distributed Internet Architecture but not for SOA

Security

Distributed Internet Architecture

- Traditional security architecture incorporates delegation and impersonation as well as **encryption.**

SOA

- 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

Distributed Internet Architecture

Maintaining component-based 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.

SOA

SOA requires additional runtime administration:

- Problems with messaging frameworks
- Additional administration of a private or public registry of services

3. Explain what are the benefits of using SOA?

3. 1 Improved integration (and intrinsic interoperability)

SOA can result in the creation of solutions that consist of inherently interoperable services .Utilizing solutions based on interoperable services is part of service-oriented integration (SOI) and results in a service-oriented integration architecture. The cost and effort of cross-application integration is significantly lowered when applications being integrated are SOA-compliant.

3. 2 Inherent reuse

Service-orientation promotes the design of services that are inherently reusable. Building services to be inherently reusable results in a moderately increased development effort and requires the use of design standards.

3. 3 Streamlined architectures and solutions

The concept of composition is another fundamental part of SOA. The WS-* platform is based in its entirety on the principle of composability. Benefits of streamlined solutions and architectures include the potential for reduced processing overhead and reduced skill-set requirements (because technical resources require only the knowledge of a given application, service, or service extension).

3. 4 Leveraging the legacy investment

The industry-wide acceptance of the Web services technology set has produced a large adapter market, enabling many legacy environments to participate in service-oriented integration architectures. The cost and effort of integrating legacy and contemporary solutions is lowered.

3. 5 Establishing standardized XML data representation

On its most fundamental level, SOA is built upon and driven by XML. A standardized data representation format (once fully established) can reduce the underlying complexity of all affected application environments. With contemporary SOA, establishing an XML data representation architecture becomes a necessity, providing organizations the opportunity to achieve a broad level of standardization.

3. 6 Focused investment on communications infrastructure

SOA can centralize inter-application and intra-application communication as part of standard IT infrastructure. This allows organizations to evolve enterprise-wide infrastructure by investing in a single technology set responsible for communication.

3.7 “Best-of-breed” alternatives

A key feature of service-oriented enterprise environments is the support of “best-of-breed” technology. Because SOA establishes a vendor-neutral communications framework, it frees IT departments from being chained to a single proprietary development and/or middleware platform. For any given piece of automation that can expose an adequate service interface, you now have a choice as to how you want to build the service that implements it.

3.8 Organizational agility

Agility is a quality natural in just about any aspect of the enterprise. A simple algorithm, a software component, a solution, a platform, a process—all of these parts contain a measure of agility related to how they are constructed, positioned, and leveraged.

4. Discuss in detail about the Common principles of service- orientation. (NOV/DEC 2011) (MAY/JUN 2013)(May/June 2015)

A service-oriented architecture is an environment standardized according to the principles of service-orientation in which a process that uses services (a service-oriented process) can execute.

Separation of concerns:

"**Separation of concerns**" is an established software engineering theory based on the idea of **breaking down a large problem** into a series of **individual concerns.**

- Allows the logic required to solve the problem to be decomposed into a collection of smaller, related pieces. Each piece of logic addresses a specific concern.
- Implemented in different ways with different development platforms

Principles of Service orientation:

- Services are reusable
- Services share a formal contract
- Services are loosely coupled
- Services abstract underlying logic
- Services are compos able
- Services are autonomous
- Services are stateless
- Services are discoverable

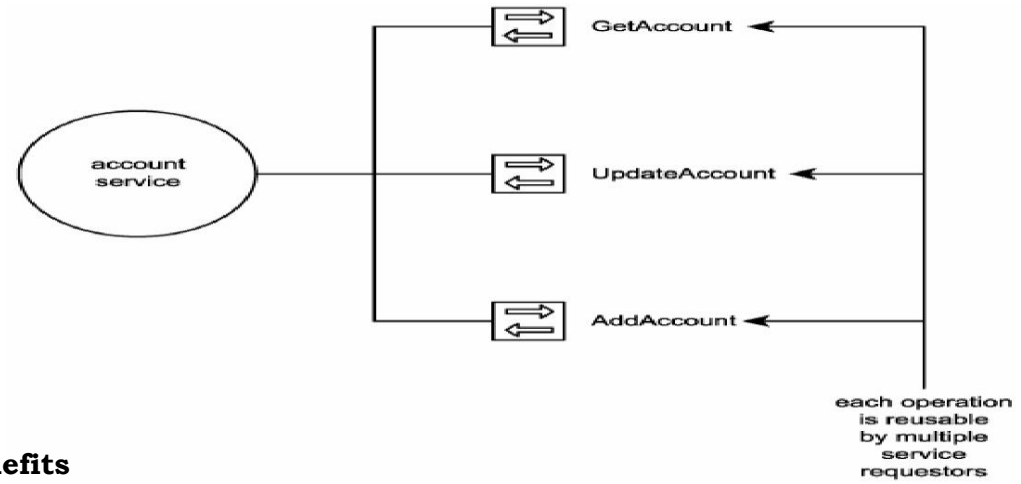
1. Services are reusable:

- **Service reusability** is a design principle that is used to **create services** (collection of related operations) that have the potential to be **reused** across the **enterprise resources.** The more generic a services operations are, the more reusable the service.
- Messaging also supports service reusability through the use of SOAP headers.

Reusability includes,

- Inter-application interoperability

- Composition
- Creation of cross-cutting or utility services



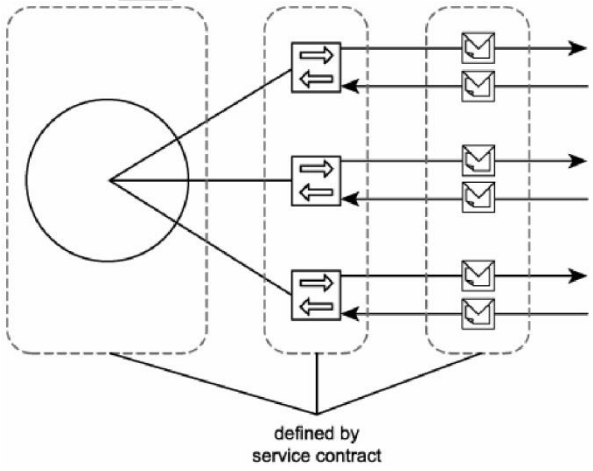
Benefits

- Accommodate future requirements with less development effort.
- Reduce the need for creating wrapper services
- Reduction of cost by not just avoiding duplication of code
- Reducing risks by reusing well-tested code and runtime environments

2. Services share a formal contract:

Service contract provide a formal definition for all of the primary parts of an SOA

- The service endpoint
- Each service operation
- Every input and output message supported by each operation
- Rules and characteristics of the service and its operations



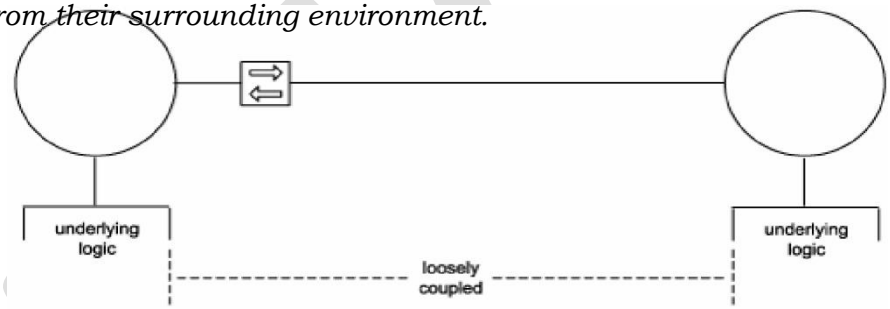
Benefits

- Provide semantic information of how a service accomplishes a particular task.
- Services express their purpose and capabilities via a service contract.
- Shared among services-need careful maintenance and versioned

3. Services are loosely coupled:

- **Loose coupling** is a condition wherein a **service acquires knowledge of another services** while still remaining independent of that services.
- It is achieved through the use of services contract that allow services to interact within predefined parameters

Note: *service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment.*

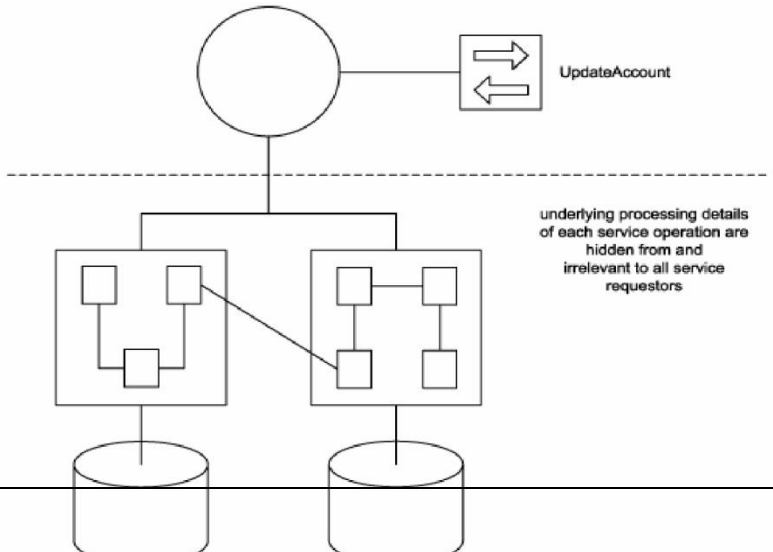


4. Services abstract underlying logic:

Service **interface-level abstraction** is this principle that allows a **service details** are **hidden** from the potential customers.

Service provides the following,

- Simple task to perform
- Gateway to an entire automation solution
- Represent limitless amount of logic
- Act as a container for the operations that abstract the logic

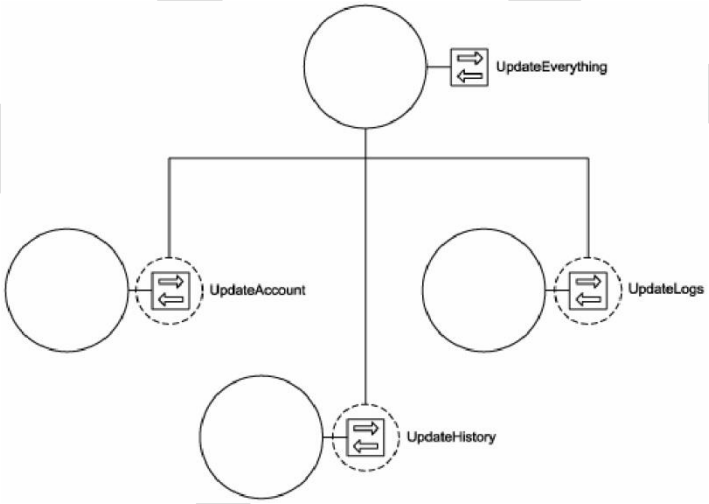


Benefits

- Directly enables and preserves the previously described loosely coupled relationship.

5. Services are composable:

- **Services are effective composition** participants, **regardless** of the **size** and **complexity** of the composition. It is controlled by a parent process service that compose process participants
- A service composition is an aggregate of services collectively composed to automate a particular task or business process.



Benefits

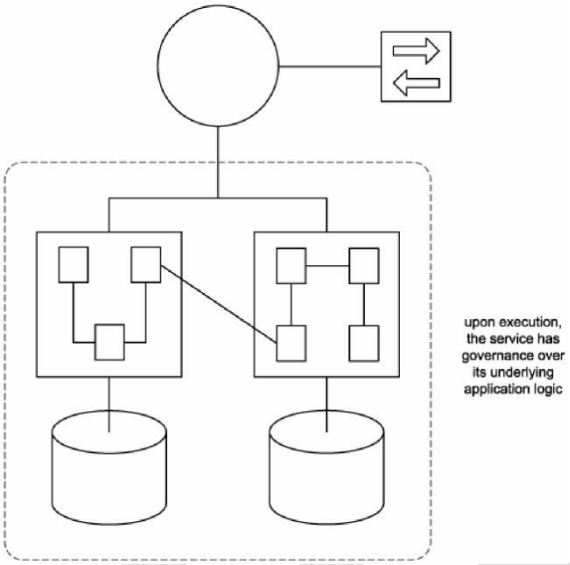
- Reusability
- Emphasis on the design of service operations

6. Services are autonomous:

- **Autonomy** allows the **service** to execute **self-governance** (self-controlling, independent, and self-contained) of all its processing (how application logic should be divided up into services and which operations should be grouped together within a service context).

- **An autonomous service** is a service whose **ability to function** is **not controlled** or inhibited **by other services.**

Note: Does not necessarily grant a service exclusive ownership of the logic encapsulates



Types of autonomy:

- **Service-level autonomy: Service boundaries** are **distinct** from each other, but the **service may share underlying resources**. It governs the legacy system but also shares resources with other legacy clients.

- **Pure autonomy :** When the underlying logic is built from the ground up in support of the service which has complete control and ownership of that logic.

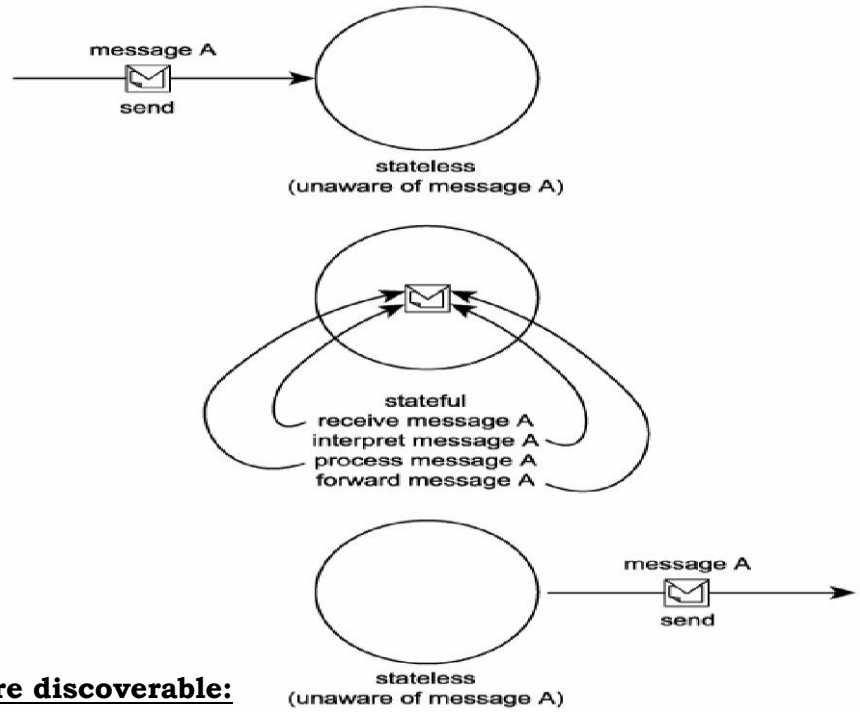
7. Services are stateless:

State refers to something's **particular condition**. They are two primary conditions can be invoked:

- **Stateful : A Stateful services** a service that is **actively engaged** in the process of **retaining** or processing **state information.**
- **Stateless :** A **stateless service** is a service whose response does **not require access** or use **of information** nor is contained in the input message.

Statelessness is the preferred condition for services.

- Stateless services do scale better
- Promotes reusability and scalability



8. Services are discoverable:

- In SOA level, **discoverability r**efers to the **architectures ability** to **provide** a **discovery mechanism**, such as a **service registry or directory.**
- On a service level, the principle of discoverability refers to the design of an individual service so that it can be as discoverable as possible.

Benefits

- Avoids the accidental creation of redundant services or services that implement redundant logic.
- One or more business processes and that SOA promotes the organization of these services into specialized layers that abstract specific parts of enterprise automation logic.
- Also by standardizing on SOA across an enterprise, a natural interoperability emerges that transcends proprietary application platforms. This allows for previously disparate environments to be composed in support of new and evolving business automation processes.

5. How do Service orientation principles inter-relate with each other?

- Services are reusable = service reusability
- Services share a formal contract = service contract

- Services are loosely coupled = service loose coupling
- Services abstract underlying logic = service abstraction
- Services are composable = service composability
- Services are autonomous = service autonomy
- Services are stateless = service statelessness
- Services are discoverable = service discoverability

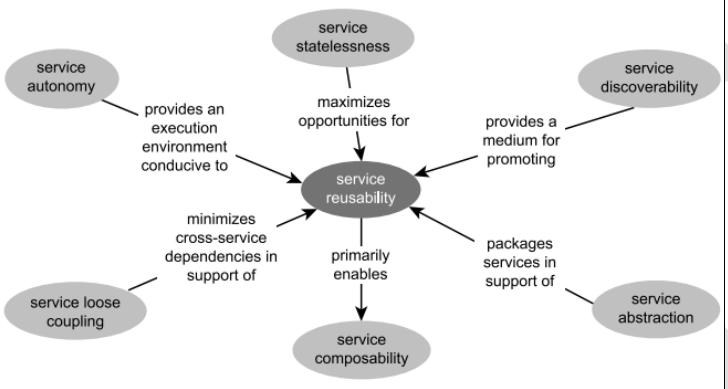


Figure 8.26
Service reusability and its relationship with other service-orientation principles.

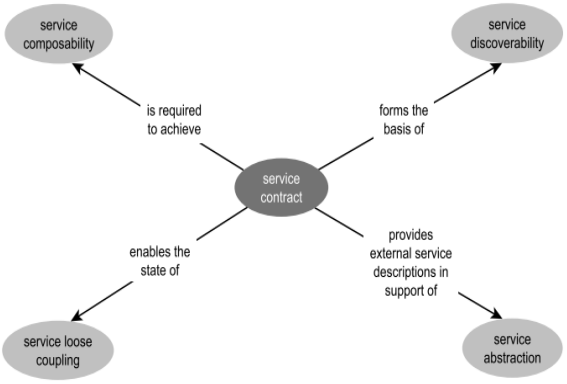


Figure 8.27
The service contract and its relationship with other service-orientation principles.

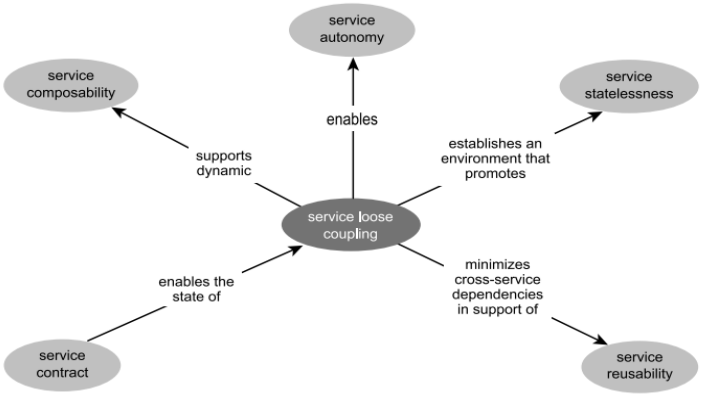


Figure 8.28
Service loose coupling and its relationship with other service-orientation principles.

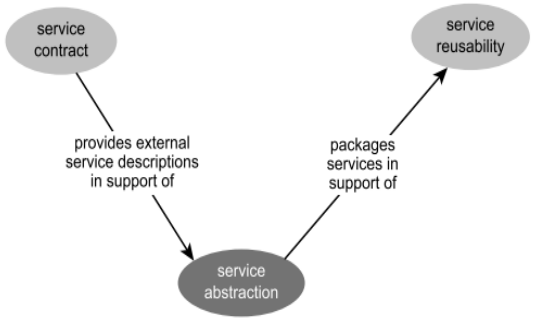


Figure 8.29
Service abstraction and its relationship with other service-orientation principles.

6.Explain in detail the Anatomy of Service oriented Architecture.

In SOA, we need to abstract the key components of the Web services framework and study their relationships more closely. Then we position them into a logical view wherein we subsequently re-examine our components within the context of SOA.

The Anatomy of Service oriented architecture includes the following Components.

6.1 Logical components of the Web services framework

Each Web service contains one or more operations. Each operation governs the processing of a specific function the Web service is capable of performing. The processing consists of sending and receiving SOAP messages,

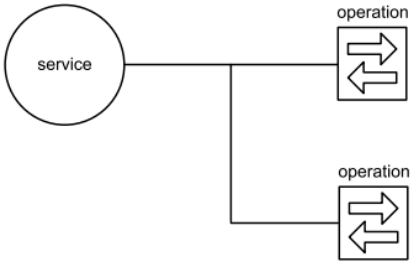


Figure 8.4
A Web service sporting two operations.

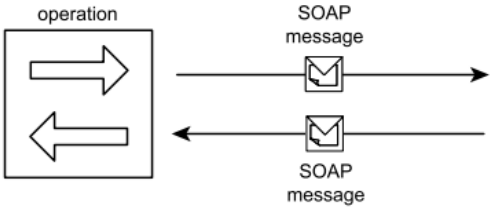


Figure 8.5
An operation processing outgoing and incoming SOAP messages.

By composing these parts, Web services form an activity through which they can collectively automate a task

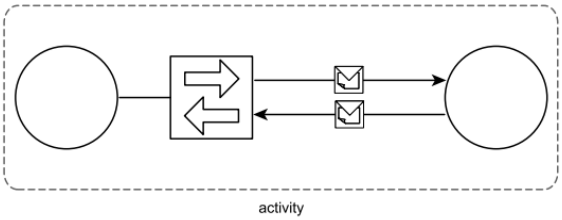


Figure 8.6
A basic communications scenario between Web services.

6.2 Logical components of automation logic

The following fundamental parts of the framework:

- SOAP messages
- Web service operations
- Web services
- activities

The latter three items represent units of logic that perform work and communicate using SOAP messages.

- 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)

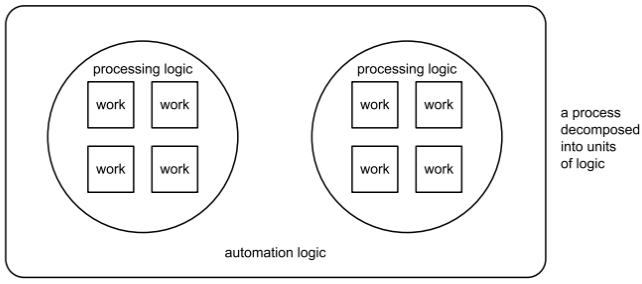


Figure 8.7
A primitive view of how SOA modularizes automation logic into units.

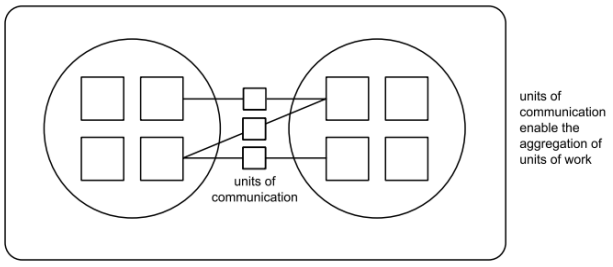


Figure 8.8
A primitive view of how units of communication enable interaction between units of logic.

6.3 Components of an SOA

The previously defined components establishes a level of enterprise logic abstraction, as follows:

- **message -** the data required to complete some or all parts of a unit of work.
- **operation-** logic required to process messages in order to complete a unit of work
- **service -** logically grouped set of operations capable of performing related units of work.
- **process -**contains the business rules that determine which service operations are used to complete a unit of automation.

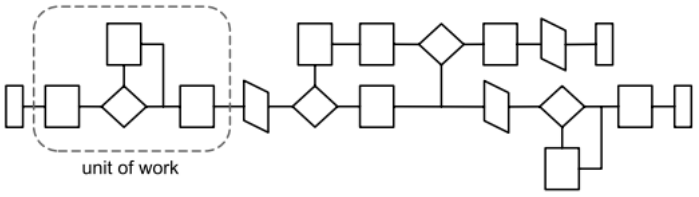
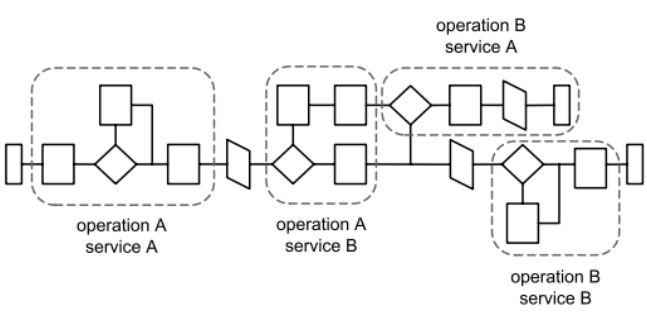


Figure 8.9
The scope of an operation within a process.



6.4 How components in an SOA inter-relate

- 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.

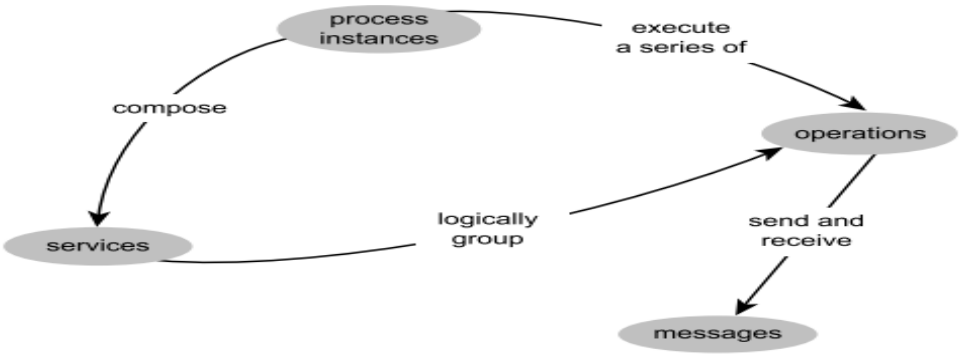


Figure 8.11
How the components of a service-oriented architecture relate.

7. Explain about Service Layer Abstraction in detail. (NOV/DEC 2011) (MAY/JUN 2012) (MAY/JUN 2013) (May/June 2015)(NOV/DEC 2017)

The service layer is between the application layer and the business process layer

Problems solved by layering services

What logic should be represented by services?

Enterprise logic can be divided into **two primary domains**

- Business logic
- 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.

How should services relate to existing application logic?

Existing legacy application logic needs to be exposed via services or whether new logic is being developed in support of services number of constrains, limitations, and environmental requirements

How can services best represent business process logic

When modeling service to represent business logic, the service representation of business logic should be alignment with existing business models.

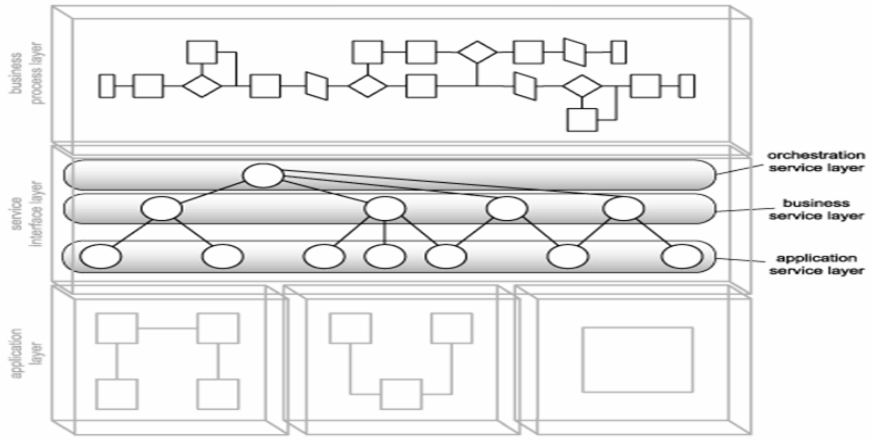
How can services be built and positioned to promote agility?

The key to building an agility SOA is in minimizing the dependencies each service has within its own processing logic.

Layers of abstraction Each layer can abstract a specific aspect of the solution, addressing to one of the issues that are identified. The three layers of abstraction identified for SOA are:

- The application service layer
- The business service layer
- The orchestration service layer

The three primary service layers



1. APPLICATION SERVICE LAYER

The application service layer consists of application services that represent technology specific logic.

Examples

Typically incarnations of application services are the

- Utility models
- Wrapper models

It consists of services that encapsulate some or all parts of a legacy environment to expose legacy functionality to service requestors.

Characteristics

- Expose functionality within a specific processing context
- Draw upon available resources within a given platform
- Solution-agnostic
- Generic and reusable
- Achieve point-to-point integration with other application services
- Inconsistent in terms of the interface granularity they expose

- Mixture of custom-developed and third-party purchased services

Hybrid application services

Services that contain both application and business logic can be referred to as hybrid application services or just hybrid services.

Application integration services

Application integration services that exist solely to enable integration between systems often are referred to as application integration services or simply integration services. It is implemented as controllers.

Proxy services

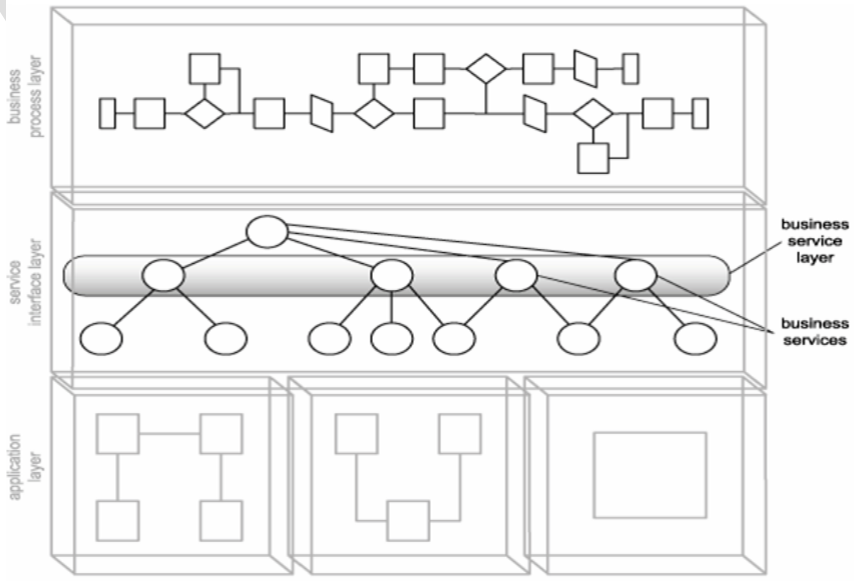
Proxy services, also known as an auto-generated WSDL, simply provide a WSDL definition that mirrors an existing component interface.

2. BUSINESS SERVICE LAYER

The business service layer is comprised of business services, a direct implemented of the business services model

Business services are ideally also controllers that compose application services to execute their business logic

The business service layer



Types

Business service layer abstraction leads to the creation of two further business service models:

1. **Task-centric business service**

A service encapsulation business logic specific to a task or business process

- Limited reusability

2. Entity centric business service

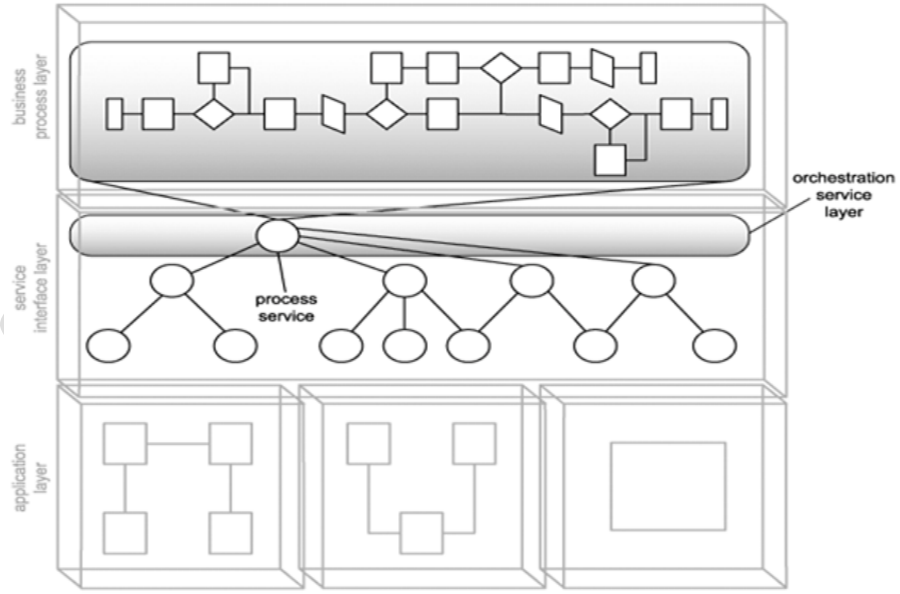
A service encapsulates a specific business entity

- Highly reusability

Even though hybrid services contain both business and application logic, they are not classified as business services.

3. ORCHESTRATION SERVICE LAYER

The orchestration service layer.



- The orchestration service layer consists of one or more process services that compose business and application services according to business rules and business logic embedded within process definitions.
- Orchestration abstracts business rules and services execution sequence logic from other services, promoting agility and reusability.

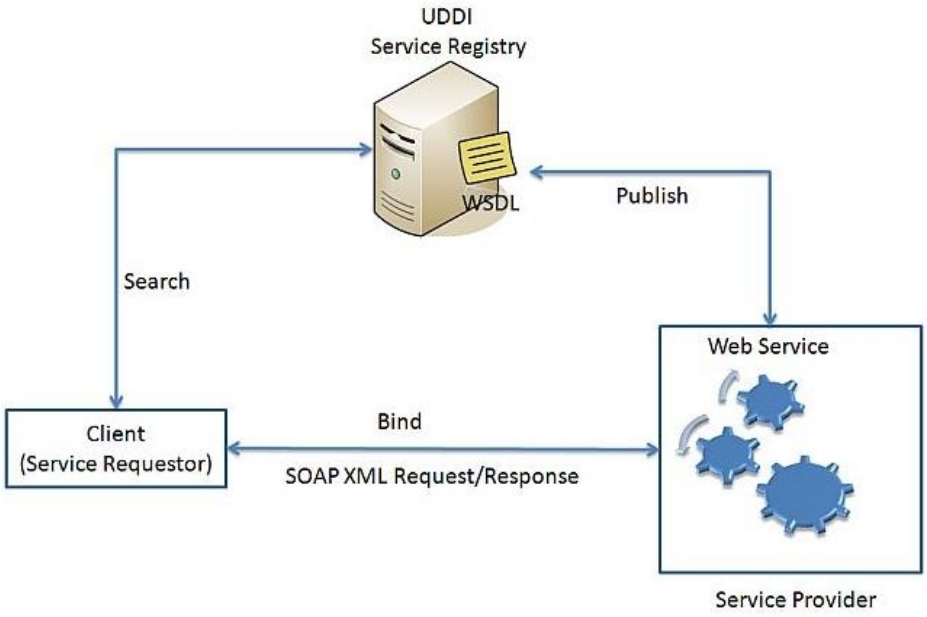
UNIT III WEB SERVICES (WS) AND STANDARDS

Web Services Platform – Service descriptions – WSDL – Messaging with SOAP – Service discovery – UDDI – Service-Level Interaction Patterns – Orchestration and Choreography

PART – B

1. Explain in detail about Web Services Platform

- ✓ XML along with HTTP forms the basis of web services. XML provides a language which can be used between different platforms and programming languages and still express complex messages and functions. The HTTP protocol is the most used Internet protocol.
- ✓ Web services platform consists of the following components:
- UDDI (Universal Description, Discovery and Integration)
- WSDL (Web Services Description Language)
- SOAP (Simple Object Access Protocol)



UDDI

UDDI (Universal Description, Discovery and Integration) is a platform-independent, XML based registry service where companies can register and search for Web services.

- UDDI is a directory for storing information about web services
- UDDI communicates via SOAP
- UDDI is a directory of web service interfaces described by WSDL

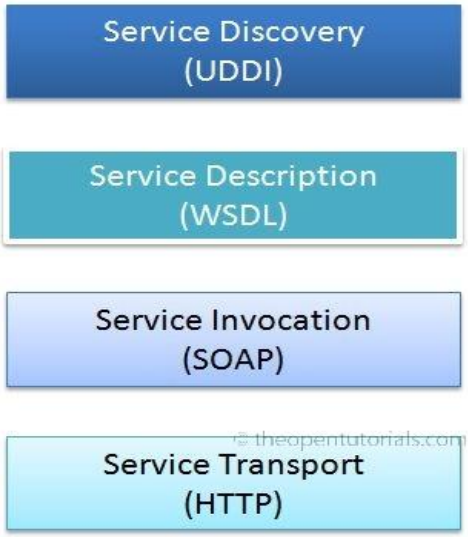
WSDL

WSDL (Web Services Description Language) is an XML-based language for locating and describing Web services. WSDL definition describes how to access a web service and what operations it will perform along with the message format and protocol details for the web service. WSDL is a W3C standard.

SOAP

SOAP (Simple Object Access Protocol) is an XML-based communication protocol for exchanging structured information between applications over HTTP, SMTP or any other protocol. In other words, SOAP is a protocol for accessing a Web Service.

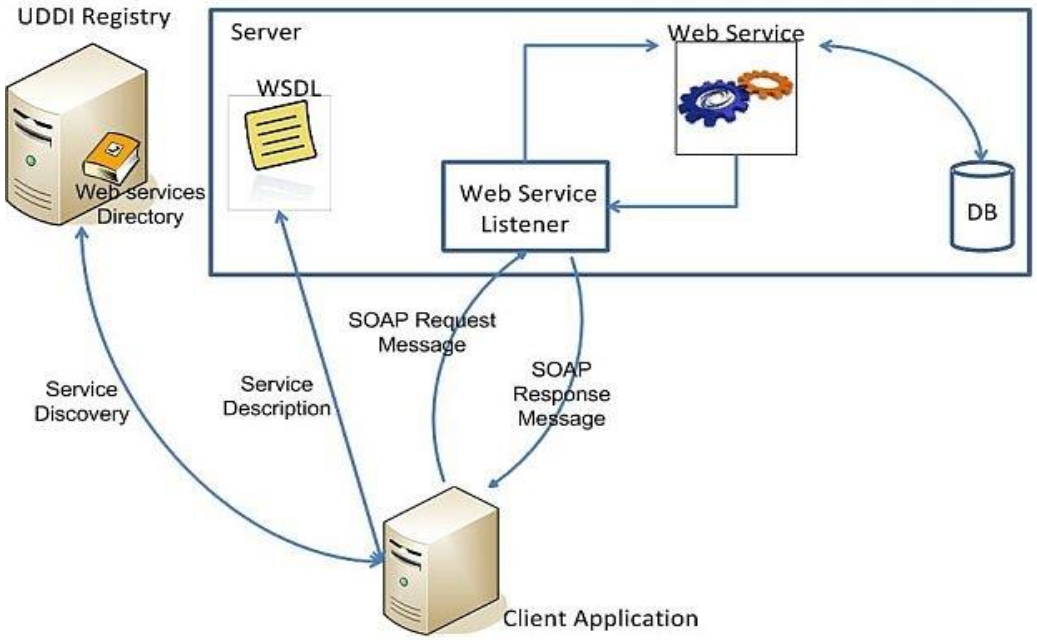
Web Services Architecture



- **Service Discovery:** This part of the architecture is responsible for centralizing services into a common registry and providing easy publish/search functionality. UDDI handles service discovery.
- **Service Description:** One of the most interesting features of Web Services is that they are self-describing. This means that, once a Web Service is located, it will let us know what operations it supports and how to invoke it. This is handled by the Web Services Description Language (WSDL).
- **Service Invocation:** Invoking a Web Service involves passing messages between the client and the server. SOAP (Simple Object Access Protocol) specifies how we should format request messages to the server, and how the server should format its response messages.

- **Transport:** Finally, all these messages must be transmitted somehow between the server and the client. The protocol of choice for this part of the architecture is HTTP (HyperText Transfer Protocol) – the protocol used to access conventional web pages on the Internet. We could also use other protocols, but HTTP is currently the most used one.

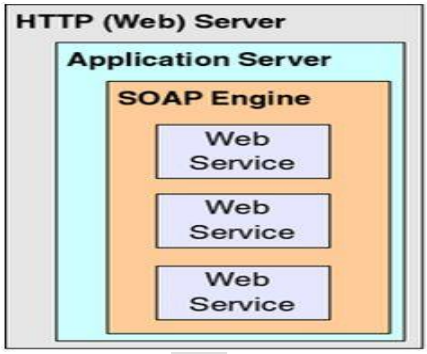
Web Services working



1. The Service Provider generates the WSDL describing the application or service and registers the WDSL in UDDI directory or Service Registry.
2. The Service Requestor or client application which is in need of web service contacts the UDDI and discovers the web service.
3. The client based on the web service description specified in the WSDL sends a request for a particular service to the web service application listener in SOAP message format.
4. The web service parses the SOAP message request and invokes a particular operation on the application to process that particular request. The result is packed in an appropriate SOAP response message format and sent to the client.

5. The client parses the SOAP response message and retrieves the result or error messages if any.

Server-side Components of Web Services Application



- **Web service:** This is the software or component that exposes a set of operations. For example, if we are implementing our Web service in Java, our service will be a Java class (and the operations will be implemented as Java methods). Clients will invoke these operations by sending SOAP messages.
- **SOAP Engine:** Web service implementation does not know anything about interpreting SOAP requests and creating SOAP responses. To do this, we need a SOAP engine. This is a piece of software that handles SOAP requests and responses. Apache Axis is an example of SOAP engine. The functionality of the SOAP engine is usually limited to manipulating SOAP.
- **Application Server:** To actually function as a server that can receive requests from different clients, the SOAP engine usually runs within an Application Server. This is a piece of software that provides a 'living space' for applications that must be accessed by different clients. The SOAP engine runs as an application inside the application server. A good example is the Apache Tomcat server – a Java Servlet and JSP container.
- **HTTP Server:** Many application servers already include some HTTP functionality, so we can have Web services up and running by installing a SOAP engine and an application server. However, when an application server lacks HTTP functionality, we also need an HTTP Server. This is more commonly called a 'Web server'. It is a piece of software that knows how to handle HTTP messages. A good example is the Apache HTTP Server, one of the most popular web servers in the Internet.

2. Explain in detail about Service descriptions (with WSDL)?

There exists a 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**.

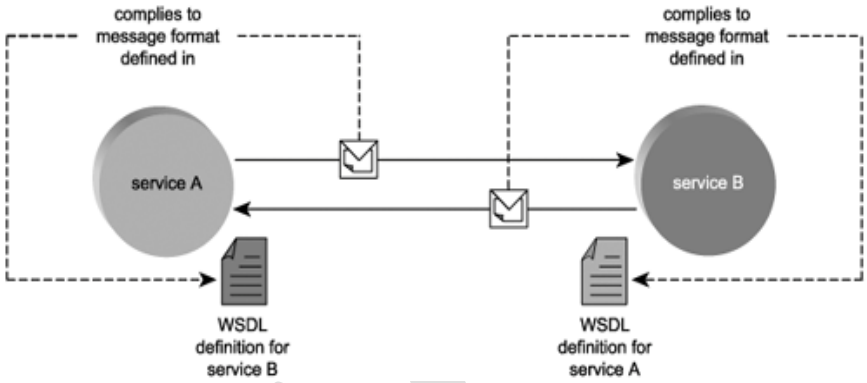


Figure. WSDL definitions enable loose coupling between services.

Service endpoints and service descriptions

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.

A **WSDL service description** (also known as WSDL service definition or just WSDL definition) can be separated into **two categories:**

- **Abstract description**
- **Concrete description**

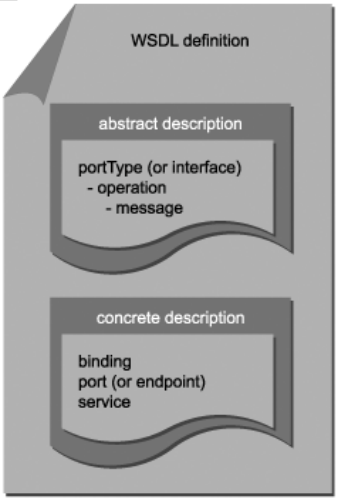


Figure .WSDL document consisting of abstract and concrete parts that collectively describe a service endpoint.

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**
- **Message**

- The **parent portType** section of an abstract description provides a **high-level view of the service interface** by sorting the messages a service can process into groups of functions known as operations.
- Each **operation** represents a **specific action performed by the service**. A service operation is comparable to a public method used by components in traditional distributed applications. Much like component methods, operations also have input and output parameters.
- Because 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.

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.

This connection is defined in the **concrete description** portion of the WSDL file, which consists of **three related parts:**

- **binding**
- **port**
- **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.
- 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 **port,** which represents the **physical address** at which a **service can be accessed** with a specific protocol. This piece of physical implementation data exists separately to allow location information to be maintained independently from other aspects of the concrete description.
- The term **service** is used to **refer to a group of related endpoints**.

Metadata and service contracts

Policies can **provide rules, preferences, and processing** details above and beyond what is expressed through the WSDL and XSD schema documents.

So now we have up to **three separate documents** that each describes an **aspect of a service:**

- **WSDL definition**
- **XSD schema**
- **policy**

Service description documents can be collectively viewed as **establishing a service contract** a **set of conditions that must be met** and accepted by a potential service requestor to enable successful communication.

A **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

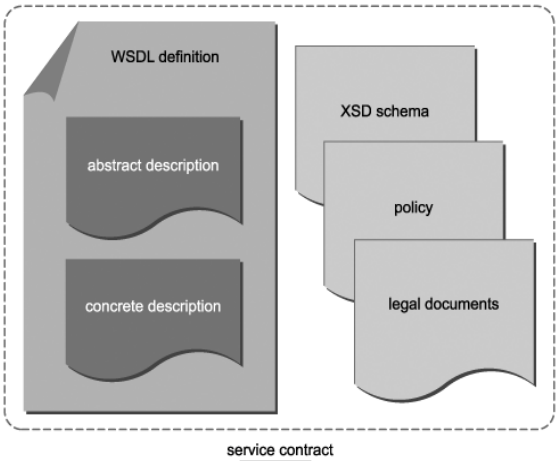


Figure. A service contract comprised of a collection of service descriptions and possibly additional documents.

Semantic descriptions

The most challenging part of providing a complete description of a Web service is in communicating its semantic qualities.

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?

- Most of the time service semantics are assessed by humans, either verbally by discussing the qualities of a service with its owner, or by reading supplementary documentation published alongside service descriptions.
- The ultimate goal is to provide sufficient semantic information in a structured manner so that, in some cases, service requestors can go as far as to evaluate and choose suitable service providers independently.

Service description advertisement and discovery

The sole requirement for one service to contact another is access to the other service's description. 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, UDDI formed part of the first generation of Web services standards. Though not yet commonly implemented, **UDDI** provides us with a registry model.

Private and public registries:

UDDI specifies a relatively accepted standard for structuring registries that keep track of service descriptions . These registries can be searched manually and accessed programmatically via a standardized API.

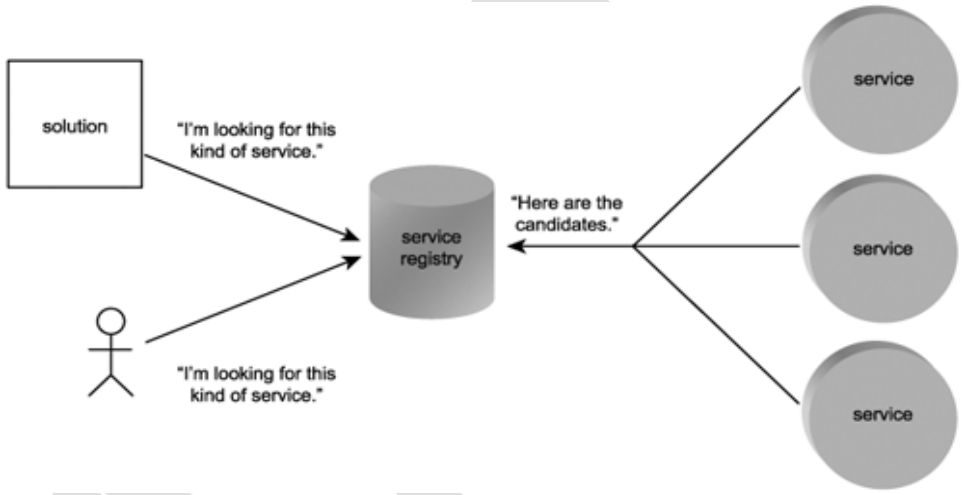


Figure. Service description locations centralized in a registry.

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.

Descriptions of the primary parts that comprise UDDI registry records:

Business entities and business services

- Each public registry record consists of a business entity containing basic profile information about the organization (or service provider entity).
- Included in this record are one or more business service areas, each of which provides a description of the services offered by the business entity. Business services may or may not be related to the use of Web services.

Binding templates and tModels

- The WSDL definitions stored implementation information separately from the actual interface design. This resulted in an interface definition that existed independently from the transport protocols to which it was eventually bound.
- Registry records follow the **same logic** in that they **store binding information** in a separate area, called the **binding template.**
- Each business service can reference one or more binding templates. The information contained in a binding template may or may not relate to an actual service.

For example, a binding template may simply point to the address of a Web site. However, if a Web service is being represented, then the binding template references a tModel. The tModel section of a UDDI record provides pointers to actual service descriptions (Figure).

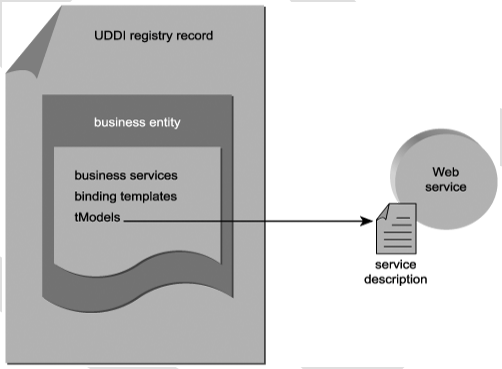
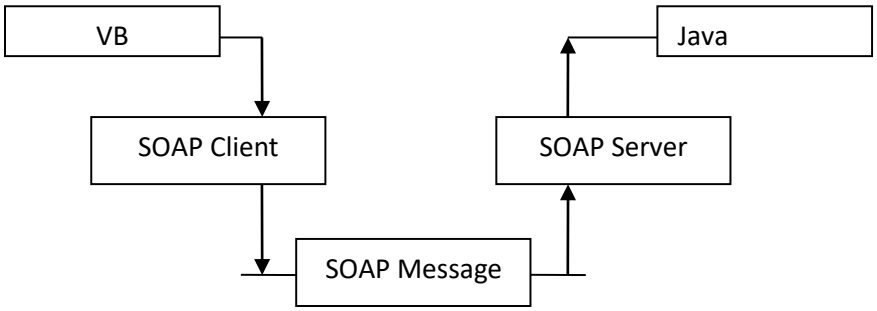


Figure . The basic structure of a UDDI business entity record.

3. Write in detail about SOAP with examples. (NOV/DEC 2011) (NOV/DEC 2012) (NOV/DEC 2013) (May/June 2015)

The simple object access protocol (SOAP) is used to **define a standard message format** which is used for communication between services running on different operating systems.



Characteristics

SOAP messaging framework has the following three characteristics that is

- Extensible
- Interoperable
- Independent

SOAP Message Format

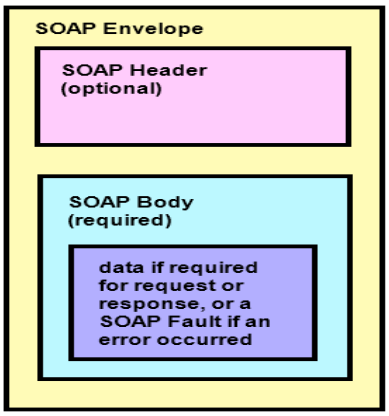
Basic structure

SOAP message consists of four parts:

- SOAP envelope
- SOAP header
- SOAP body
- SOAP fault

SOAP envelope

- The **SOAP <envelope>** is the **root element** in every SOAP message, and contains two child elements, an optional <Header> and a mandatory<Body>.
- Every **SOAP message** is **packaged** into a container known as an **envelope.** It defines an overall framework for expressing what is in a message and who should deal with it.



SOAP Header (optional)

Header determines how a **recipient of a SOAP message** should **process** the message.

SOAP Body

Body contains the **actual message content** which consists of XML formatted data. The contents of a message body are often referred to as the message payload.

Header Blocks

The **immediate child elements** of the <header> element are called **blocks.** A header block is an application-defined XML element, and represents a logical grouping of data which can be targeted at SOAP nodes that might be encountered in the message path from a sender to an ultimate receiver.

Header Processing

The attributes on the header blocks indicate how the header blocks are to be processed by the SOAP nodes.

- SOAP messages are allowed to pass through many intermediaries before reaching their destination.
    - Intermediary=some unspecified routing application
    - The final destination processes the body of the message
- Headers are allowed to be processed independently of the body
    - Processed by intermediaries.

Message styles

The SOAP offers two messaging styles:

- **RPC(Remote Procedure Call)-style**

  Creating tightly coupled, inter-object style interfaces for web services components

  It is also known as section 5 encoding
- **Document-style**

  Developing loosely coupled, application-to-application and system-to-system interfaces.

  It is also known as message-style or document-literal encoding

Attachments

SOAP attachments are used to send large quantities binary data with the SOAP message which may not fit well into a XML SOAP element.

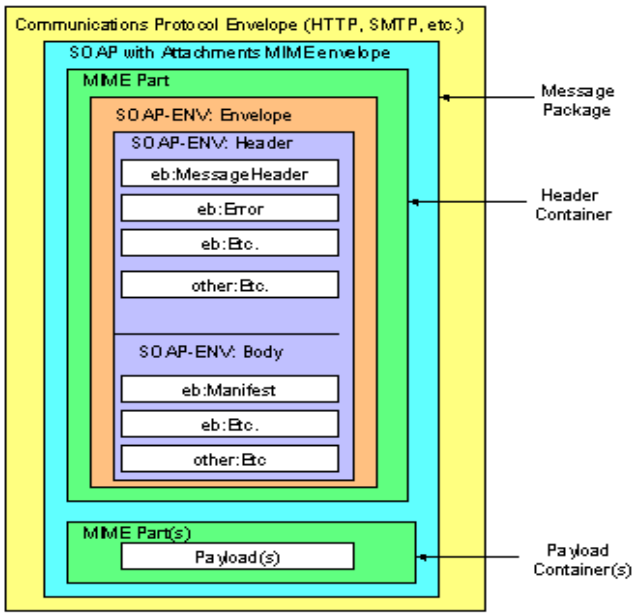
Faults

The SOAP fault element holds errors and status information for a SOAP message. It appear as an immediate child of the body element.<faultcode> and <faultstring> are required.

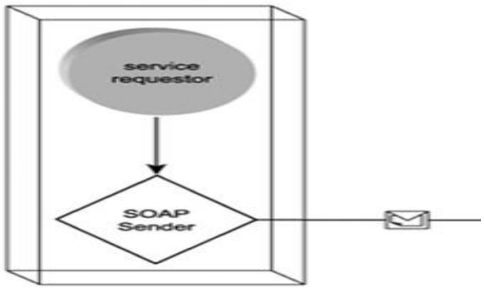
Anatomy of a SOAP message

```
<? xml version="1.0"?>
<soap:Envelope
        xmlns:soap=http://www.w3.org/2001/12/soap-envelope
    soap:encoding style=http://www.w3.org/2001/12/soap-encoding>
    <soap:header>
    ……
    </soap:header>

    <soap:body>
    ….
        <soap:fault>
        …….
        </soap:fault>
    </soap:body>
</soap:envelope>
```



- Originators, recipients, and receivers of SOAP messages are all called SOAP nodes.



Node type

SOAP nodes types or labels are assigned depending on what form of processing they are involved with in a given message processing scenario.

SOAP Concepts

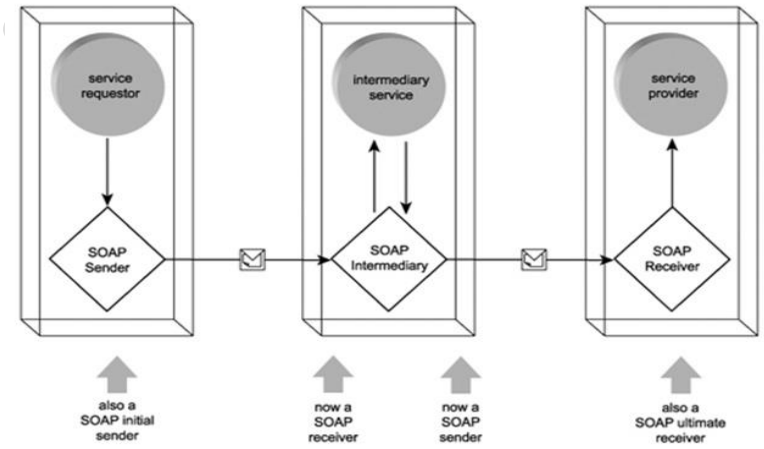
- **SOAP sender -**SOAP node that generates and transmits a SOAP message
- **SOAP receiver-**SOAP node that receives and processes the SOAP message that was generated by a SOAP sender
- **SOAP intermediary-**SOAP node that is considered a SOAP receiver as well as a SOAP sender.
- **Initial SOAP sender-**It is the SOAP sender that generated the original SOAP message
- **Ultimate SOAP receiver-**It is a SOAP receiver that is the final destination of a SOAP message

SOAP Intermediaries

SOAP intermediaries are nodes that can process parts of a SOAP message as it travels from origin to destination

SOAP message header blocks are intended to be processed in general by intermediary nodes

Different types of SOAP nodes involved with processing a message



Types of intermediaries

SOAP intermediaries nodes are classified as

- Forwarding
- Active

Forwarding intermediaries

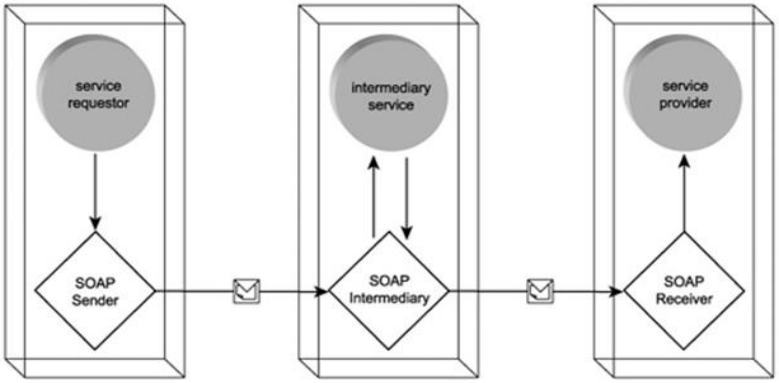
- Forwarding intermediaries are used to route message to other SOAP nodes, based on header information.
- May do additional processing as described in a SOAP header

Active intermediaries

Active intermediaries do additional processing to a message that is NOT described in any of the message headers.

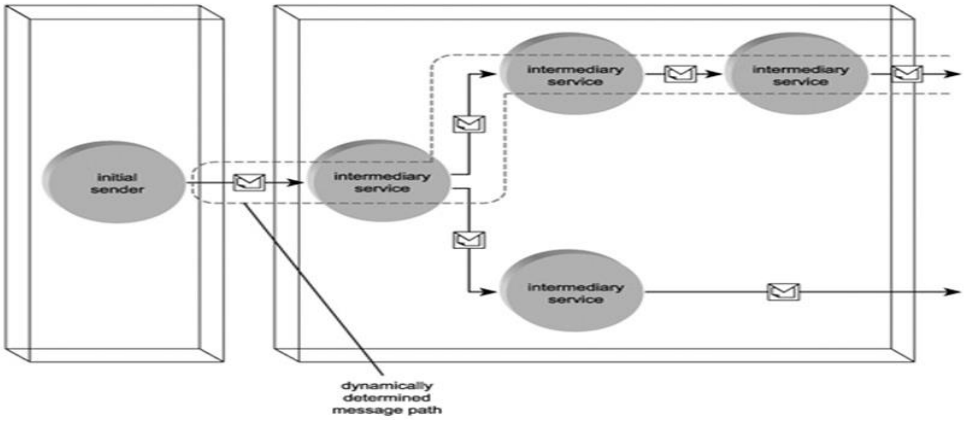
Message paths

The **route taken by the message** is called the **SOAP message path**. The set of SOAP nodes through which the SOAP message passes, including the initial sender, the ultimate receiver and one or more intermediaries, are called the SOAP message path



A **message path is not predetermined**. It is **determined dynamically** using header clocks by **intermediaries.**

A message path determined at runtime.



A SOAP message path is comprised of a series of SOAP nodes, beginning with the initial SOAP sender and ending with the ultimate SOAP receiver. Every node refers to a physical installation of SOAP software's, each with its own physical address.

4. Explain in detail about Service Discovery and how it is performed using the UDDI?

Service Discovery:

- The primary mechanism involved in performing Service Discovery is a service registry, which contains relevant metadata about available and upcoming services as well as pointers to the corresponding service contract documents that can include SLAs. The communications quality of the metadata and service contract documents play a significant role in how successful this process can be carried out.
- After the discovery process is complete, the service developer or client application should know the exact location of a Web service (URI), its capabilities, and how to interface with it.

Service Registry:

An SOA registry supports the UDDI (Universal Description, Discovery and Integration) specification, an XML– (Extensible Markup Language) based registry that was developed for the purpose of making systems interoperable.

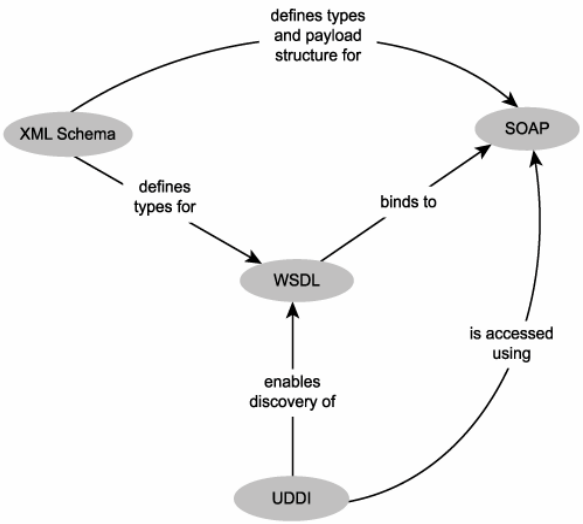
Publication of a service requires proper description of a Web service in terms of business, service, and technical information. Once, the services are published the registry will create the dependencies, associations and versions of these services and metadata.

UDDI:

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.

Therefore, depending on the scope (application-level, enterprise-wide, etc.) of the service-oriented architecture being designed, UDDI may become one of the technologies established as part of the overall service-oriented environment.

While UDDI enables the discovery of service descriptions and is also one of the core specifications identified by the WS-I Basic Profile, some organizations are resorting to traditional directory-based approaches (such as LDAP) to keep track of their service descriptions. Regardless, our service design processes take potential discovery into account by promoting the creation of intuitive service interface designs and the documentation of supplementary metadata



Benefits of SOA Service Discovery

- Discovery of the service, its status, and its owner will be critical to achieve the benefits of SOA reusability.
- Dynamic service registration and discovery becomes much more important in these scenarios in order to avoid service interruption.
- Handling Fail over of service instances
- Load balancing across multiple instances of a Service

5. Explain in detail about Service-Level Interaction Patterns.

Definition:

Message exchange pattern defines the **way that SOAP message are exchanged** between the web services requester and web service providers. It represents a set of templates

Example

Request - response pattern.

Primitive MEPs

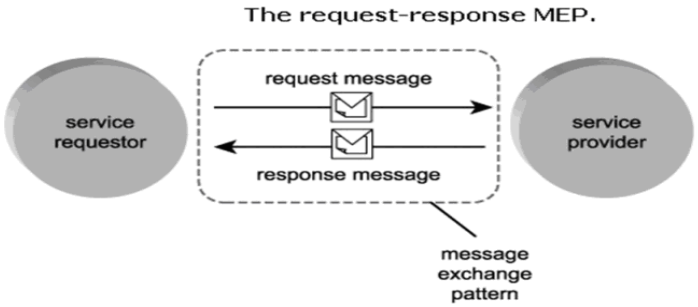
A common set of primitive MEPs have been in existence are listed below

- **Request-response**

- **Fire-and-forgot**
- **Complex MEPs**

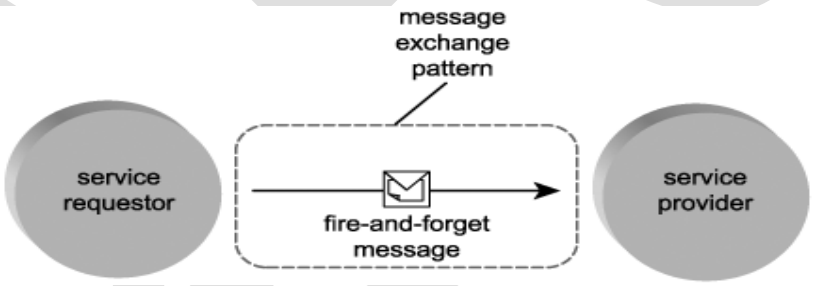
i)Request-response

In a request-response message exchange the **SOAP client** sends a **SOAP request** message to the service. The **service** responds with a **SOAP response** message.



ii)Fire-and-forgot

Fire-and-forgot is a simple **asynchronous message exchange p**attern is based on the **unidirectional transmission** of messages from a source to one or more destinations. Response to a transmitted message is not permitted.



Types

- **Single-destination pattern-** A source sends a message to one destination only
- **Multi-cast pattern-** A source sends message to a predefined set of destination
- **Broadcast pattern -**Similar to the multi-cast pattern, except that the message is sent out to a broader range of messaging models.

iii) Complex MEPs

Complex MEPs is a message pattern in which primitive MEPs are assembled to create different types of messaging models

Example

- Publish-and-subscribe pattern

Publish-and-subscribe pattern

Publish-and-subscribe pattern is a asynchronous MEP in which publishers sends messages to all interested subscribers.

The steps involved are generally similar to the following

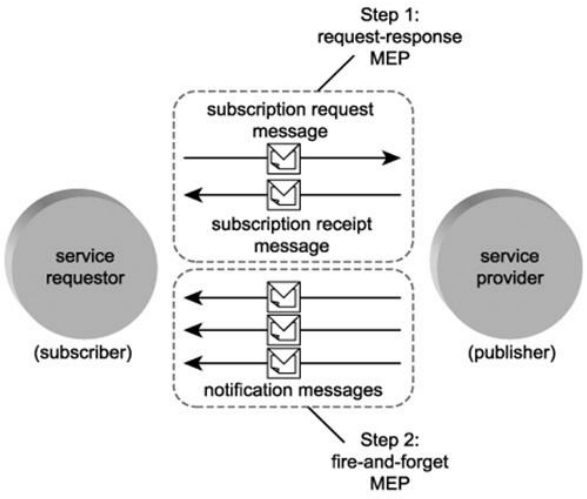
Step 1:

The subscriber sends a message to notify the publisher that it wants to receive messages on a particular topic.

Step 2:

Upon the availability of the requested information, the publisher broadcast messages on the particular topic to all of the topics subscribers.

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



MEPs and SOAP:

A **MEPs specification MUST** conform to the **requirements for SOAP** features specifications are

- Provides a one-way message transfer
- Generate countless messages implements through SOAP header blocks
- Identify the MEP associated with a message with an optional parameter

MEPs and WSDL:

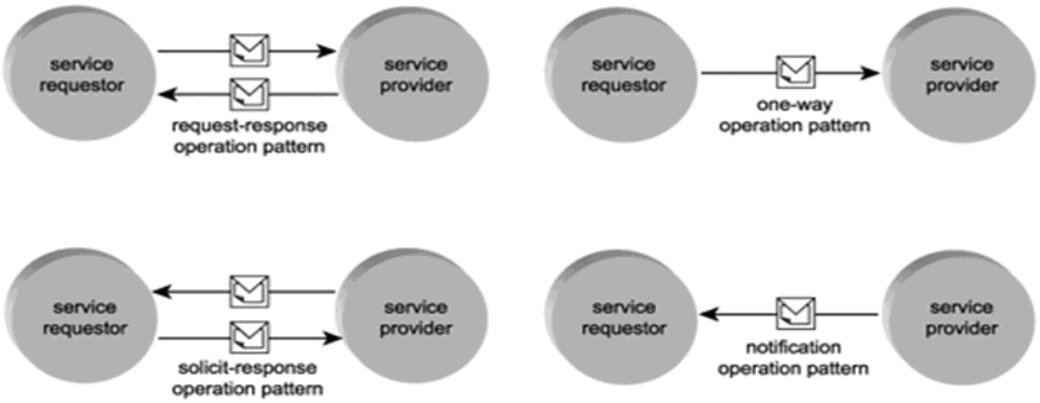
MEPs and WSDL service description coordinate the input and output messages associated with an operation. The association of MEPs to WSDL operations embeds expected conversational behavior into the interface definition

Basic patterns supported by WSDL 1.1

WSDL specification provides support for four message exchange patterns are

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

The four basic patterns supported by WSDL 1.1.



Patterns supported by WSDL 2.0

WSDL 2.0 specification extends MEP support to eight patters as follows

1. The **in-out pattern**, comparable to the request-response MEP
2. The **out-in pattern**, which is the reverse of the previous pattern where the service provider initiate the exchange by transmitting the request
3. The **in-only pattern**, which essentially supports the standard fire-and-forgot MEP
4. The **out-only pattern**, which is the reverse of the in-only pattern. it is used primarily in support of event notification

5. 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
6. The **robust out-only 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.
7. 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.
8. 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.

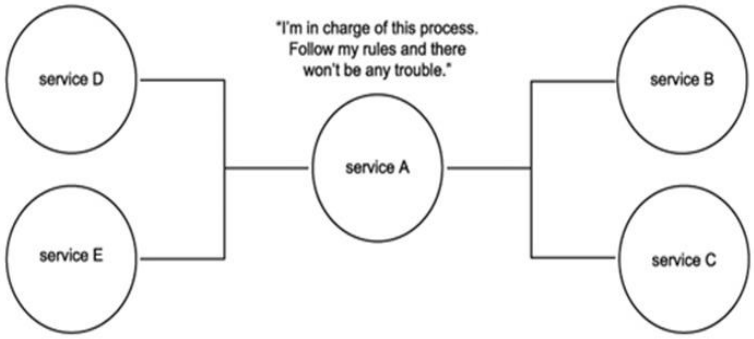
6. Discuss in detail about Orchestration and Choreography. (NOV/DEC 2012) (MAY/JUN 2013) (NOV/DEC 2013) (May/June 2015)

1. ORCHESTRATION

Orchestration, also called as a heart of SOA, which facilitates **connecting of different processes/services without** having to **redevelop the solution** that originally automated the processes individually. It introduces workflow logic, consist of numerous business rules, conditions, and events, which is abstracted and more easily maintained.

Web services business process execution language or WS-BPEL is the industry specification that standardizes orchestration.

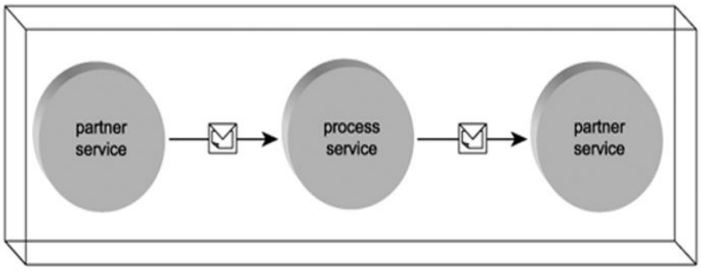
An orchestration controls almost every facet of a complex activity



Business protocols and process definition

- **Business protocols** Defines how participants can interoperate to achieve the completion of a business task.
- **Process definition -**Encapsulates and expresses the details of the workflow logic**.**
- **Process services** A process service is the process itself represented as a service. It coordinates and exposes functionality from the partner services.
- **Partner services**
    - Partner services or partner links is the services allowed to interact with the process service.
    - The process service, after first being invoked by a partner service, then invokes another partner service.

The process service, after first being invoked by a partner service, then invokes another partner service



Basic activities and structured activities

- WS-BPEL breaks down workflow logic into a series of predefined primitive activities

Basic activities represent fundamental workflow

- Invoke
- Receive
- Reply
- Throw
- Wait

Structured activities are created by assembling other activities using logics

- Sequence
- Switch
- While
- Flow
- Pick

Sequences, flows, and links

- **Sequences** -Sequences align a group of related activities in a list that determines sequential execution order.
- **Flows**- Flows also contains group of related activities but activities can be executed concurrently. The flow does not finish till the time all the activities are completed.
- **Links**-Links are used to establish formal dependencies between activities that are part of a flow. Links are also knows as synchronization dependencies.

Orchestration and activities

An activity is a generic term that can be applied to any logical unit of work completed by a service-oriented solution. The scope of a single orchestration, therefore, can be classified as a complex, and most likely, long-running activity.

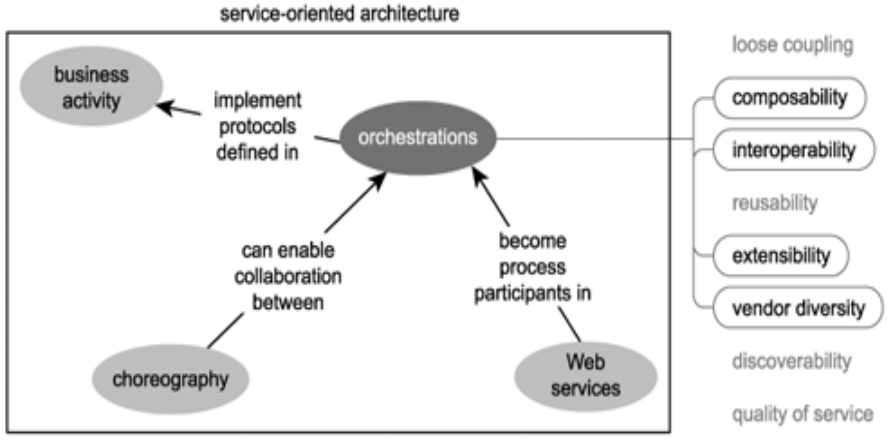
Orchestration and coordination

Orchestration, as represented by WS-BPEL, can fully utilize the WS-Coordination context management framework by incorporating the WS-BusinessActivity coordination type. This specification defines coordination protocols designed to support complex, long-running activities.

Orchestration and SOA

Through the use of orchestrations, service-oriented solution environments achieves the following characteristics

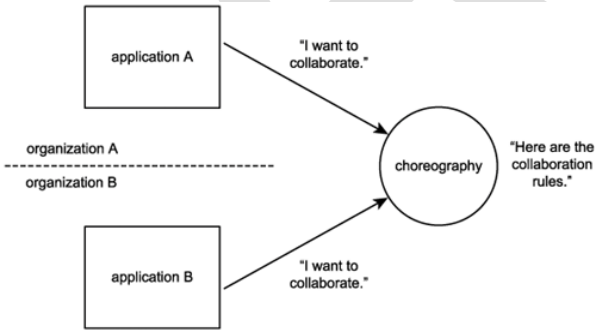
- Composability
- Interoperability
- Extensibility
- Vendor diversity



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.

2. CHOREOGRAPHY (MAY/JUN 2013)

Choreography is a complex activity comprised as a service composition and a series of MEPs. The web services choreography description language (WS-CDL) is the specification that represents the Choreography.



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

Roles and participants

Roles

Roles establish what the service does and what the service can do within the context of a particular business task.

Participants

Participants are the roles that bounded to WSDL definitions, and those related are grouped.

Relationships and channels

Relationship

Relationship defines potential exchange between two roles in choreography. Every relationship consequently consists of exactly two roles.

Channels

Channels define the characteristics of the message exchange between two specific roles.

Further, to facilitate more complex exchanges involving multiple participants, channel information can actually be passed around in a message. This allows one service to send another the information required for it to be communicated with by other services. This is a significant feature of the WS-CDL specification, as it fosters dynamic discovery and increases the number of potential participants within large-scale collaborative tasks.

Interaction and work units

Interaction

Interaction encapsulates the actual logic behind a message exchange.

Work units

Work units are the rules and constrains that must be adhered to for an interaction to successfully completed

Reusability, composability, and modularity

Reusability

Designed choreography can be applied to different business tasks comprised of the same fundamental actions.

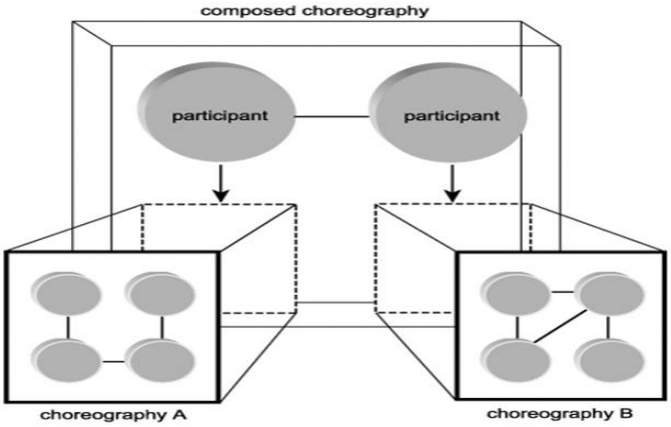
Modularity

Choreography can be assembled from independent modules that can represent distinct sub-task.

Composability

Choreography composes a set of non-specific services to accomplish a task.

A choreography composed of two smaller choreographies.

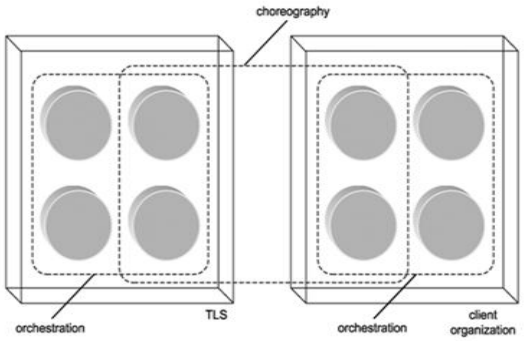


Orchestrations and choreographies:

Although **orchestrations and choreographies** look very similar to each other there are **significant differences** among them.

- An orchestration expresses organization specific business workflow
- An organization controls the logic behind an orchestration even if it involves external businesses
- Choreography is not owned by a single entity, it acts as a community interchange pattern used for collaborative purpose by services from different provider entities.

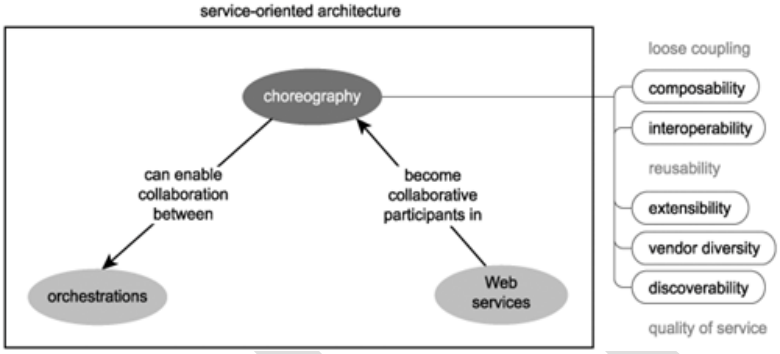
A choreography enabling collaboration between two different orchestrations.



Choreography and SOA

Choreography can assist in the realization of SOA across organization boundaries. It support the following SOA characteristics

- Compos ability
- Reusability
- Extensibility
- Increase organizational agility
- Discoverability



- Choreography is a complex activity comprised of a service composition and a series of MEPs.
- Choreographies consist of multiple participants that can assume different roles and that have different relationships.
- Choreographies are reusable, composable, and can be modularized.
- The concept of choreography extends the SOA vision to standardize cross-organization collaboration.

UNIT IV WEB SERVICES EXTENSIONS

WS-Addressing - WS-Reliable Messaging - WS-Policy – WS-Coordination – WS - Transactions - WS-Security – Examples

PART - B

1. Explain in detail about WS-Addressing.

The WS-Addressing specification implements these addressing features by providing two types of SOAP headers (explained shortly). Though relatively simple in nature, these addressing extensions are integral to SOA's underlying messaging mechanics. Many other WS-* specifications implicitly rely on the use of WS-Addressing.

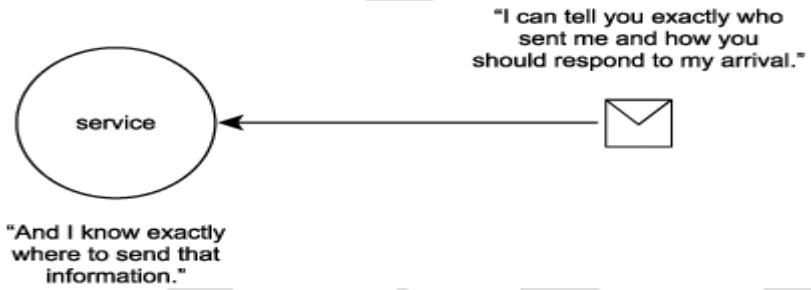


Figure. Addressing turns messages into autonomous units of communication.

Endpoint references

- ✓ Traditional Web applications had different ways of managing and communicating session identifiers. The most common approach was to append the identifier as a query string parameter to the end of a URL. While easy to develop, this technique resulted in application designs that lacked security and were non-standardized.
- ✓ 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.

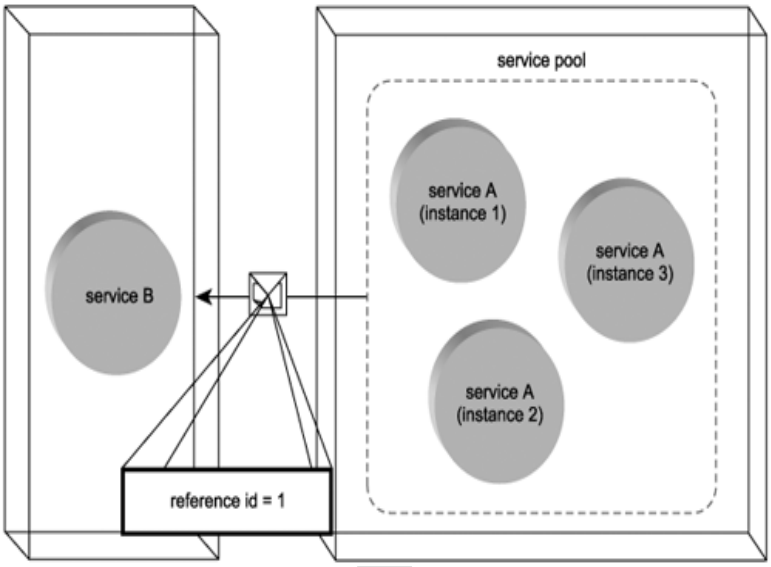


Figure A SOAP message containing a reference to the instance of the service that sent it.

An endpoint reference consists of the following parts:

- address The URL of the Web service.
- reference properties A set of property values associated with the Web service instance. (In our previous In Plain English example, the "attention" line used in the first scenario is representative of the reference ID property.)
- reference parameters A set of parameter values that can be used to further interact with a specific service instance.
- 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.
- policy A WS-Policy compliant policy that provides rules and behavior information relevant to the current service interaction (policies are explained later in this chapter).

✓ Additional parts exist, which mostly identify corresponding WSDL information. With the exception of the address, all parts are optional.

Message information headers

In sophisticated service-oriented solutions, services often require the flexibility to break a fixed pattern. For example, they may want to dynamically determine the nature of a message exchange. The extensions provided by WS-Addressing were broadened to include new SOAP headers that establish message exchange-related

characteristics within the messages themselves. This collection of standardized headers is known as the message information (or MI) headers

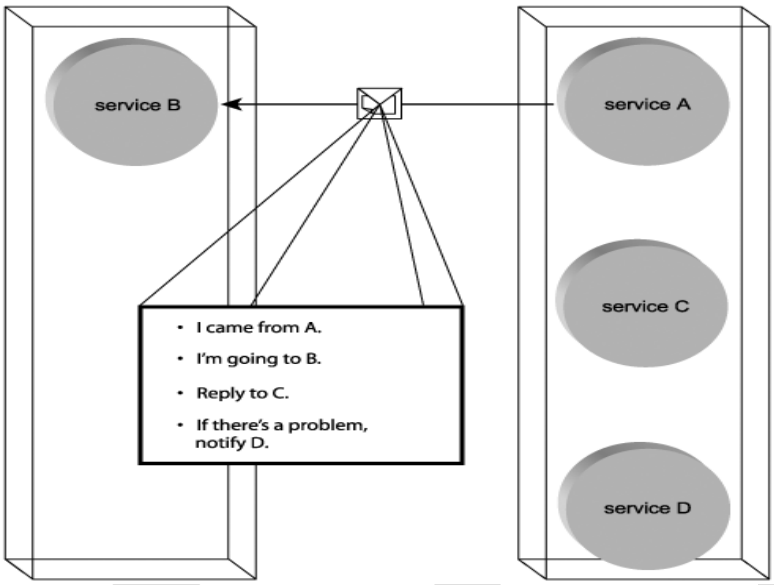


Figure. A SOAP message with message information headers specifying exactly how the recipient service should respond to its arrival.

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 (this header is required when using the reply endpoint header).
- relationship Most commonly used in request-response scenarios, this header contains the message id of the related message to which a message is replying (this header also is required within the reply message).
- action A URI value that indicates the message's overall purpose (the equivalent of the standard SOAP HTTP action value).

✓ Outfitting a SOAP message with these headers further increases its position as an independent unit of communication. Using MI headers, SOAP messages now can contain detailed information that defines the messaging interaction

behavior of the service in receipt of the message. The net result is standardized support for the use of unpredictable and highly flexible message exchanges, dynamically creatable and therefore adaptive and responsive to runtime conditions.

- ✓ Historically, many of the details pertaining to how a unit of communication arrives at point B after it is transmitted from point A was left up to the individual protocols that controlled the transportation layer. While this level of technology-based abstraction is convenient for developers, it also leads to restrictions as to how communication between two units of processing logic can be achieved.
- ✓ The standardized SOAP headers introduced by WS-Addressing remove much of this protocol-level dependence. These headers put the SOAP message itself in charge of its own destiny by further increasing its ability to act as a standalone unit of communication.

Addressing and SOA

- ✓ Addressing achieves an important low-level, transport standardization within SOA, further promoting open standards that establish a level of transport technology independence. The use of endpoint references and MI headers deepens the intelligence embedded into SOAP messages, increasing message-level autonomy.

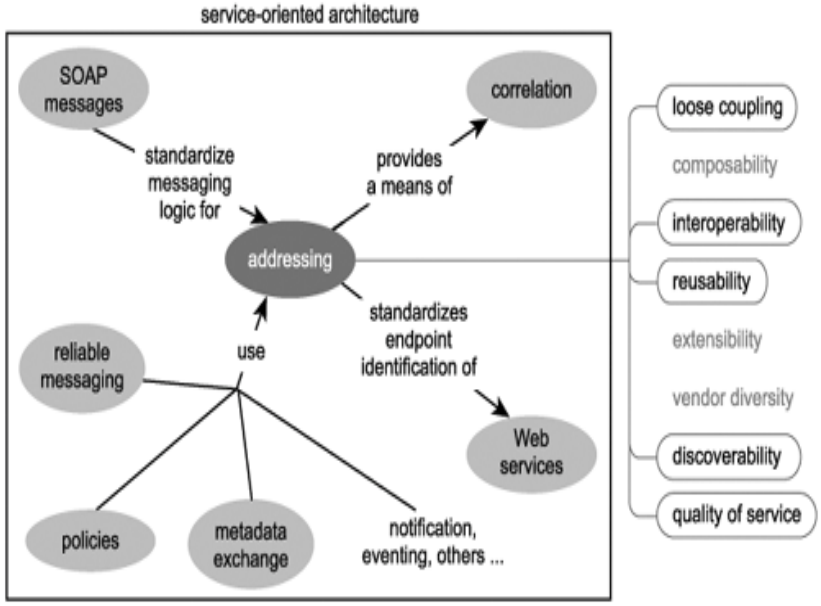


Figure 7.5. Addressing relating to other parts of SOA.

- ✓ Empowering a message with the ability to self-direct its payload, as well as the ability to dictate how services receiving the message should behave, significantly increases the potential for Web services to be intrinsically interoperable. It places the task-specific logic into the message and promotes a highly reusable and generic service design standard that also facilitates the discovery of additional service metadata.
- ✓ Further, the use of MI headers increases the range of interaction logic within complex activities and even encourages this logic to be dynamically determined. This, however, can be a double-edged sword. Even though MI headers can further increase the sophistication of service-oriented applications, their misuse (or overuse) can lead to some wildly creative and complex service activities.

2. Explain Web Service Reliable messaging in detail with necessary diagrams.

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

Reliable messaging addresses these concerns by establishing a measure of quality assurance that can be applied to other activity management frameworks

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

WS-ReliableMessaging provides a framework capable of guaranteeing:

• that service providers will be notified of the success or failure of message transmissions

• that messages sent with specific sequence-related rules will arrive as intended (or generate a failure condition)

Although the extensions introduced by reliable messaging govern aspects of service activities. Reliable messaging does not employ a coordinator service to keep track of the state of an activity. All reliability rules are implemented as SOAP headers within the messages themselves.

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

WS-ReliableMessaging makes a distinction between the parts of a solution that are responsible for initiating and perform a message transmission. It "send," "transmit," "receive," and "deliver," the messages. These are used for reliable messaging framework from the overall SOA.

Application source/Service/Application logic

It sends the message to the RM source, the physical processor or node that performs the actual wire transmission.

RM destination/ target processor/node

It receives the message and subsequently delivers it to the application destination.

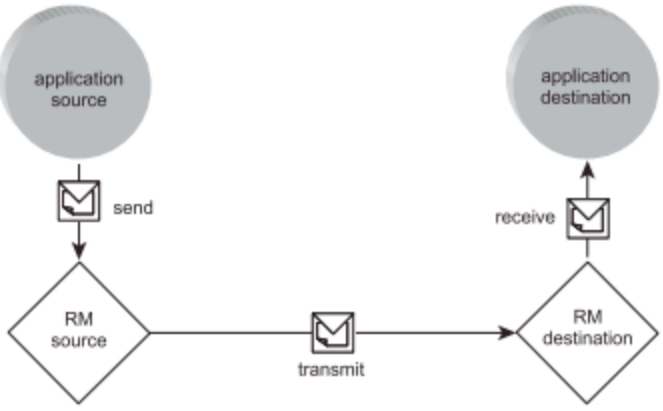


Figure 7.8
An application source, RM source, RM destination, and application destination.

7.2.2 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.

7.2.3 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 (Figure 7.9). 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.

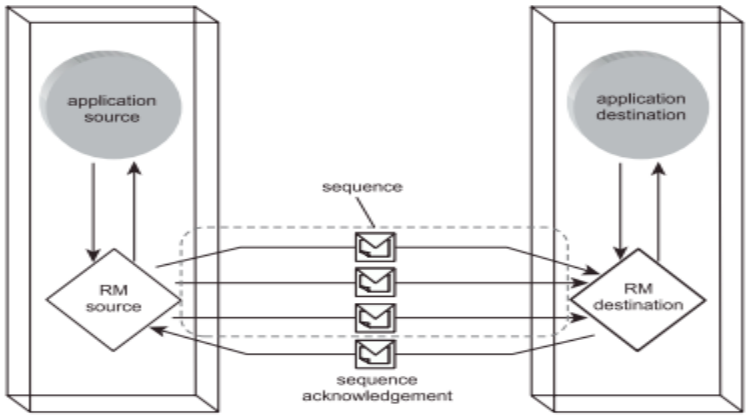


Figure 7.9
A sequence acknowledgement sent by the RM destination after the successful delivery of a sequence of messages.

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). 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).

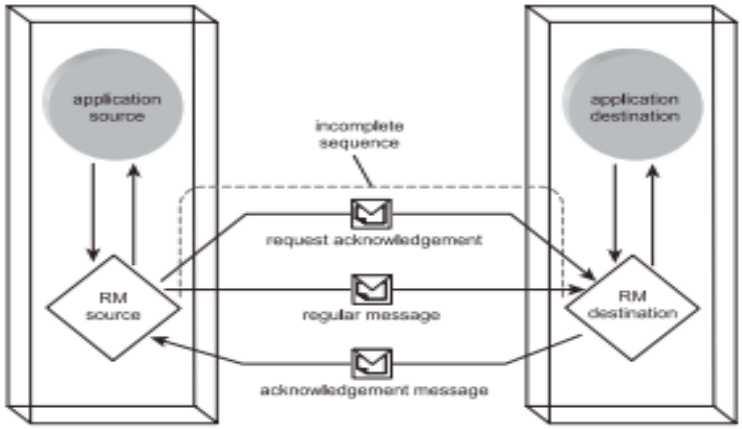


Figure 7.10
A request acknowledgement sent by the RM source to the RM destination, indicating that the RM source would like to receive an acknowledgement message before the sequence completes.

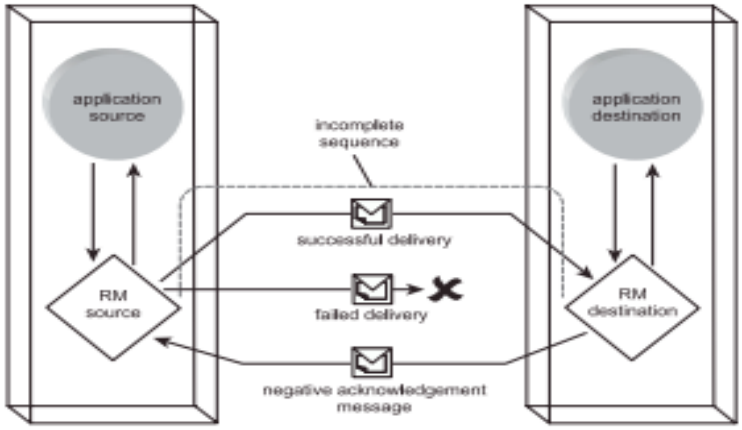


Figure 7.11
A negative acknowledgement sent by the RM destination to the RM source, indicating a failed delivery prior to the completion of the sequence.

7.2.4 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 (Figure 7.12).

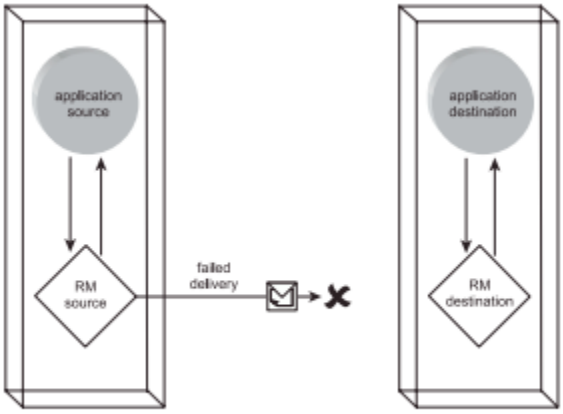


Figure 7.12
The AtMostOnce delivery assurance.

The AtLeastOnce delivery assurance allows a message to be delivered once or several times. The delivery of zero messages creates an error condition (Figure 7.13).

The ExactlyOnce delivery assurance guarantees that a message only will be delivered once. An error is raised if zero or duplicate messages are delivered (Figure 7.14).

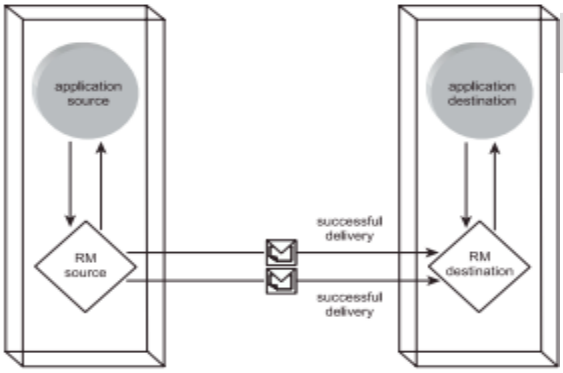
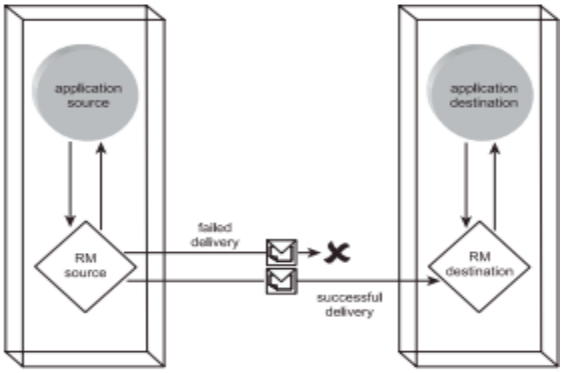


Figure 7.13
The AtLeastOnce delivery assurance.



The InOrder delivery assurance is used to ensure that messages are delivered in a specific sequence (Figure 7.15). 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.

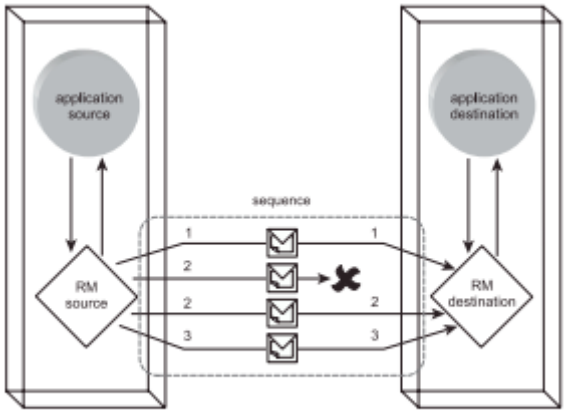


Figure 7.15
The InOrder delivery assurance.

7.2.5 Reliable messaging and addressing

WS-Addressing is closely tied to the WS-ReliableMessaging framework. In fact, it's interesting to note that the rules around the use of the WS-Addressing message id header were altered specifically to accommodate the WS-ReliableMessaging specification. Originally, message id values always had to be unique, regardless of the circumstance. However, the delivery assurances supported by WS-ReliableMessaging required the ability for services to retransmit identical messages in response to communication errors. The subsequent release of WS-Addressing, therefore, allowed retransmissions to use the same message ID.

7.2.6 Reliable messaging and SOA

Reliable messaging brings to service-oriented solutions a tangible quality of service (Figure 7.16). 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.

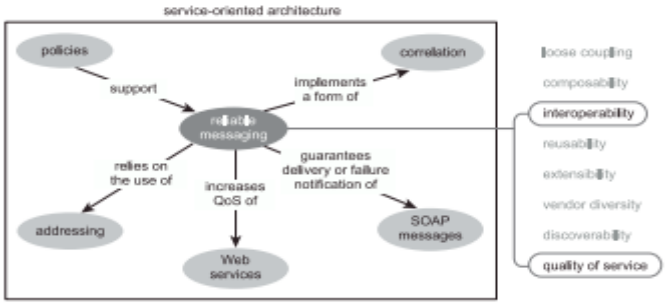


Figure 7.16
Reliable messaging relating to other parts of SOA.

3. Explain about WS-Policy with code example. (MAY/JUN 2012) (Nov/Dec 2014)

WS-policy defines a **framework** for **allowing web services** to e**xpress** their **constraints and requirements** in relation to security, processing, or message content.

Goal:

It provides the mechanisms needed to enable web services applications to specify polices.

Specification

The WS-policy framework is comprised of the following three specifications:

- WS-policy
- WS-policyassertions
- WS-policyAttachements

The policy element

The policy element establishes the root construct used to contain the various policy assertions that comprise the policy.

Policy assertions

- A policy assertion represents an individual preference, requirement, capability, or other characteristics
- It is basic building block of a policy expression
- An XML element with a well-known name and meaning.

Standard policy assertions

WS-policy assertions define four general policy assertions for any subject.

Policy assertions	**description**

Wsp:textencoding	Specifies a character encoding
Wsp:language	Specifies a natural language
Wsp:specversion	Specifies a version of a particular specification
Wsp:messagepredicate	It can be tested against the message

The usage attribute

The usage attribute to indicate whether a given policy assertion is required. It values form part of the overall policy rules.

Value	***meaning***
Wsp:required	the assertion must be applied, otherwise an error results
Wsp:rejected	the assertion is not supported and, if present,will failure
Wsp:optional	the assertion may be made of subject, but not required

The preference attributes

The preference attribute is

- Used to specify the services preference as an integer value
- Larger integer=>higher preference
- Omitted preference attribute is interpreted as a 0

Policy operators

A policy operator is used to combine multiple assertions in different ways.

Policy operator	***description***
Wsp:all	requires that all the child elements be satisfied
Wsp:exactlyone	requires that exactly one child element be satisfied
Wsp:oneormore	at least one of its child element be satisfied

The exactlyone element

Policy assertions combined using the exactlyone operator requires exactly one of the behaviours represented by the assertions.

The all element

Policy assertions combined using the all operator requires all the behaviours represented by the assertions.

Policy attachments

Policy attachments are a mechanism for associating policy expressions with subjects. It specifically defines mechanisms for:

- XML elements
- WSDL definitions

- UDDI entries

Attributes

- Wsp:policyURLs-list of URLs
- Wsp:policypres-list of Qnames

The policyattachment element

The policy attachment element binds an endpoint to a policy expression.

- AppliesTo construct is positioned as the parent of the subject elements.
- PolicyReference element then follows the appliesto construct to identify the policy assertions that will be used.

Example

```
<wsp:policyAttachment>
<wsp:AppliesTo>
<wsa:EndpointReference xmlns="...">
<was:serviceName>s:someservice</wsa:servicename>
</wsa:Endpointreferenc>
</wsp:appliesTo>
<wsse:security>
<ds:signature>....</ds:signature>
</wsse:security>
</wsp:policyAttachment>
```

The policyReference element

The policyReference element is used to link an lement with one or more policies. Each policyReference element contains a URI attribute that points to one policy document or a specific policy assertion within the document.

The policyURIs attribute

The policyURIs attribute is used to link an element with one or more policies. Each policyURIs element contains multiple URI attribute that points to one policy document or a specific policy assertion within the document.

4. Explain about WS-Coordination with code example. (NOV/DEC 2012)(Nov/Dec 2014)

WS-coordination is a framework for coordinating distributed activities

- **Coordinator**

- Activation service for creating coordination instance.
- Registration service for registering participating applications.
- Additional protocol specific services.

- **Set of coordination protocols**

The coordination context element

The coordination context is used to carry information about active coordination to participants,

- Information inside context is coordination protocol specific
- Context format is not mandated by the standard
- Typically passed in SOAP headers

Structure

<Envelope
Xmlns=http://schemas.xmlsoap.org/soap/envelope/
Xmlns:wsc=http://schemas.xmlsoap.org/ws/2002/08/wscoor"
Xmlns:wsc=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>

The identifier and expires elements

- **Identifier-** The identifier elements is used to associate a unique ID value with the current activity

- **Expires -** The expires element sets an expiry date that establishes the extent of the activity's possible lifespan.

The coordination type element

- The specific protocols that establish the rules and constraints of the activity are identified within the coordination type element.
- The coordination type element is used to represent the WS-business activity and WS-atomic transaction coordination types section.

Designating the WS-business activity coordination type

The coordination type element is assigned the WS-atomic transaction coordination type identifier, which communicates the fact that the headers context information is part of a short running transaction.

Structure

<wsc:coordinationType>
http://schemas.xmlsoap.org/ws/2003/09/wsat
</ wsc:coordinationType>

The registration service element

The registration service construct simply hosts the endpoint address of the registration service. It uses the address element also provide by the utility schema.

Structure

The value inside address tag containing a URL pointing to the location of the registration service.

<wsc:registrationservice>
<wsu:address> http://www.xmltc.com/bpel/reg </wsu:address>
</wsc:registrationservice>

5. Explain in detail about Atomic Transaction Process with suitable diagrams. (NOV/DEC 2011) (MAY/JUN 2013)

Atomic transaction:
An atomic transaction is a **series of database operation** either **all occur,** or **nothing occur.** It implements commit and rollback features to enable cross-service transaction support.

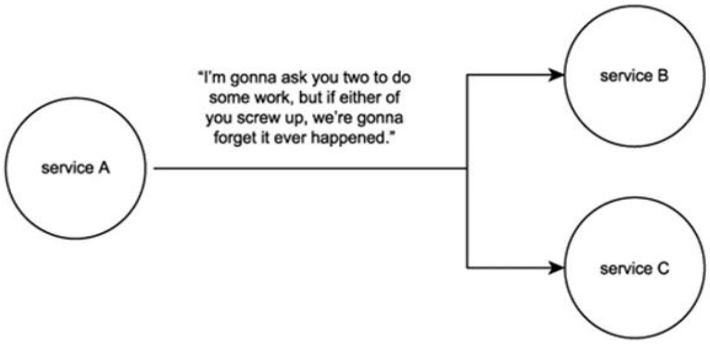
ACID transactions
The acronym ACID refers to the four key properties of a transaction:

1. **Atomicity**
   It follows an all or nothing rule. Either all changes or no changes succeed.
2. **Consistency**
   System restored to a constant state after completion.
3. **Isolation**
   Multiple transaction don't interface
4. **Durability**
   Changes made as part of transaction survive subsequent failures.

Transaction protocols
The atomic transaction defines the following protocols

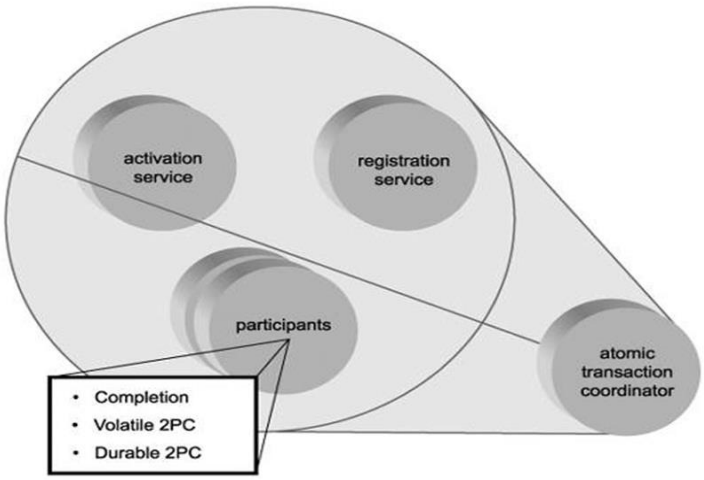
1. **Completion protocol-**Used to initiate the commit or abort states of the transaction.
2. **Durable 2PC protocol-**Used by services for representing permanent data repositories' should register.
3. **Volatile 2PC protocol-**Used by services for managing non-persistent data.

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



The atomic transaction coordinator
When **ES-atomic transaction protocols** are used, the coordinator controller service can be referred to as an **atomic transaction coordinator** used to manage the participants of the transaction process and in deciding the transactions ultimate outcome.

The atomic transaction coordinator service model



The atomic transaction process

The atomic transaction coordinator is responsible for deciding the outcome of a transaction based on feedback it receives from all of the transaction participants.

Feedback collection

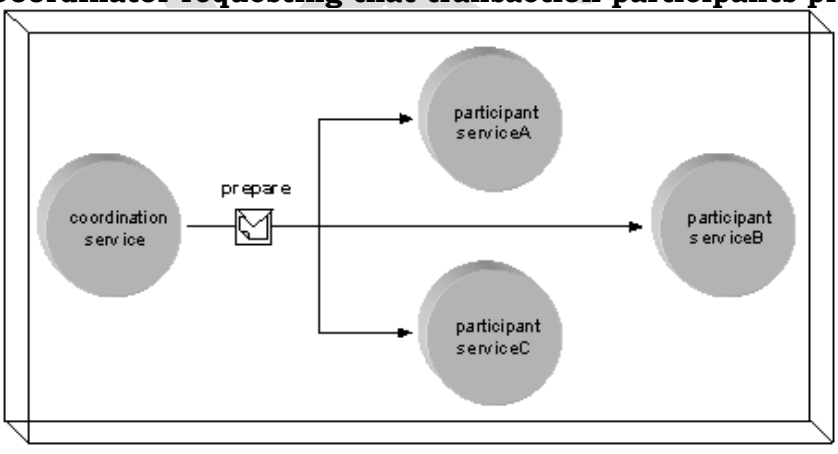
The feedback collection is divided into two phases

1. Prepare phase
2. Commit phase

Prepare phase

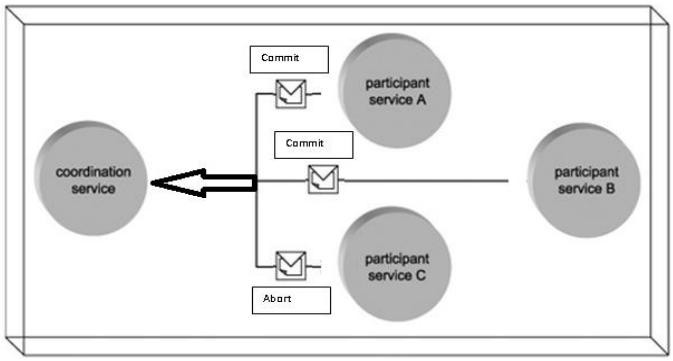
- All participants are notified by the coordinator to prepare and then issue a vote

The Coordinator requesting that transaction participants prepare to vote



- Vote consists of either a “commit” or “abort” request.

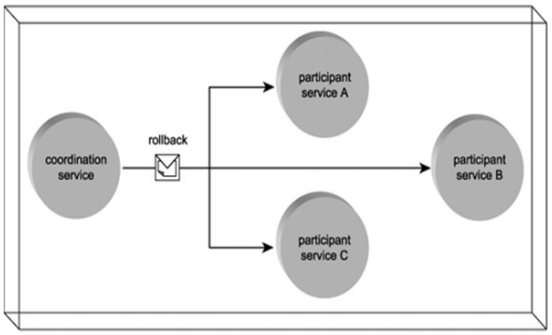
The transaction participants voting on the outcome of the atomic transaction participants prepare to vote



Commit phase

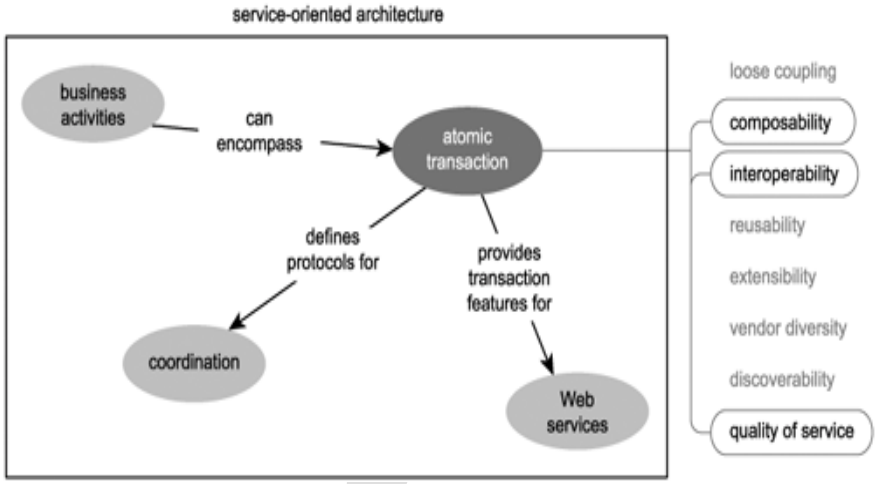
- Based on voting of the participants, the coordinator decides whether to commit or abort the transaction, and notifies the result to all the participants.

The coordinator aborting the transaction and notifying participants to rollback all changes



Atomic transactions and SOA

The atomic transactional capabilities lead to a robust execution environment for SOA activities, they promote interoperability when extended into integrated environments. This allows the scope of an activity to span different solutions built with different vendor platforms, while still being assured a guaranteed all-or-nothing outcome. The WS-AtomicTransaction is supported by the affected applications, this option broadens the application of the two-phase commit protocol beyond traditional application boundaries (thus, supporting service interoperability).



6. Explain about WS-Security with code example. (NOV/DEC 2011 & 17)

WS-security, also known as web services security or WSS, is a flexible and extensible framework to SOAP to apply security to web services.

Why WS-security?

The WS-security is used to implement

- **Message-level security measures**
  - Protect message contents during transport and during processing by service intermediaries.

- **Authentication and authorization control**
  - Protect service provides from malicious requestors.

WS-security specification

The WS-security framework is comprised of numerous specification, some are listed below:

- WS-security
- XML-encryption
- XML-signature

WS-security elements

The security element (WS-security)

The security header block contains the security-related information for the message targeted at a specific receiver.

It consists of the following child elements

- XML-encryption and XML-signature constructs
- Token elements provided by the WS-security specification itself

Syntax

```
<envelope>
<header>
....
<wsse:security actor="..."mustunderstand="...">
....
</wsse:security>
.....
</header>
<body>
....
</body>
</envelope>
```

The usernameToken, username, and password elements (WS-security)

The <usernameToken> is a way of providing a username and optional password information for authentication and authorization purpose.

Example

```
<wsse:UsernameToken>
<wsse:username>
Joe
</wsse:username>
<wsse:password>
ILoveJava
</wsse:password>
</wsse:usernameToken>
```

The binarySecurityToken element (WS-security)

The binarySecurityToken element defines a security token that is binary encoded. It has two attributes that are used to interpret it

- valueType - identifies the type of the security token
- encodingType - indicates how the security token is encoded.

Syntax

<BinarySecurityToken Id=....
EncodingType=....
ValueType=.../>

The securityTokenRefrence element (WS-security)

The securityTokenReference element provides a pointer to a token that exist outside of the SOAP message document.

Syntax

<securityTokenReference Id="...">
<reference URI="..."/>
</securityTokenReference>

XML-Encryption Elements

The XML-Encryption elements are used to define how to encrypt the contents of an XML element.

The EncryptionData element (XML-Encryption)

The EncryptionData element hosts the encryption portion of an XML document. If located at the root of an XML document, the entire document contents are encrypted.

Type attribute

Type attribute indicates what is included in the encrypted content

The cipherData, CipherValue, and CipherReference elements (XML-Encryption)

The cipherdata is a mandatory element that provides the encrypted data. It consists of the,

- **cipherValue-** hosting the character representing the encrypted text.
- **cipherReference-** provides a pointer to the encrypted values.

XML-signature elements

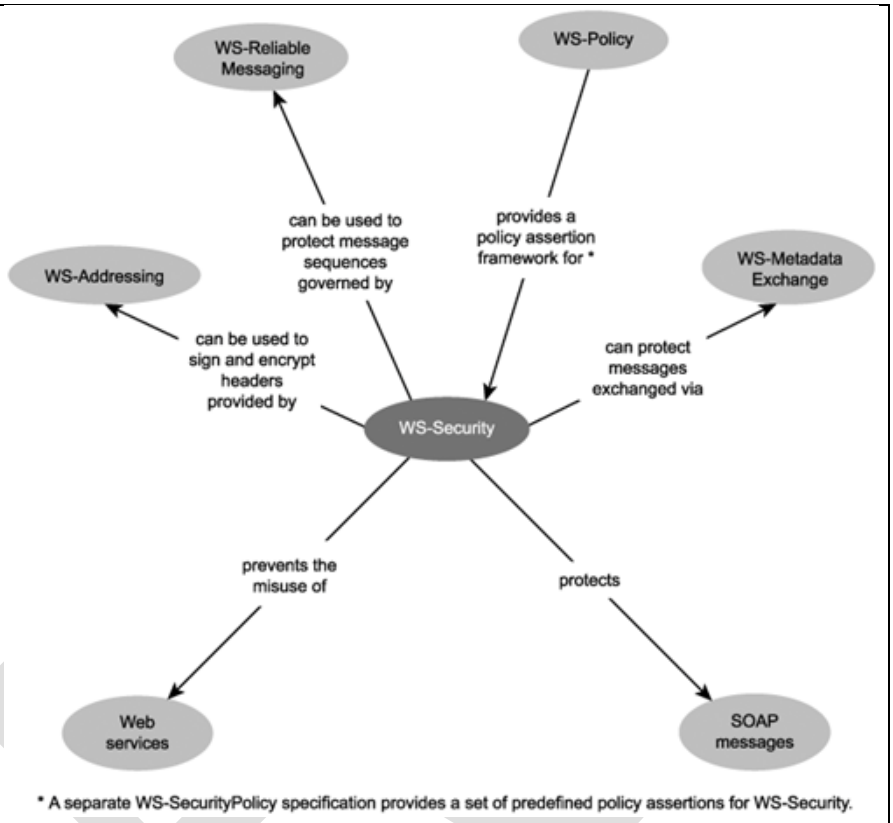
The XML-signature elements provides message integrity and authentication information about the originator of the message.

Element	**description**
CanonicalizationMethod	This element identifies the type of "canonicalization algorithm" used to detect and represent subtle variables in the document content.
DigestMethod	Identifies the algorithm used to create the signature.
DigestValue	It represents the document being signed
KeyInfo	It contains public key information of message sender.
Signature	The root element, housing all information for the digital signature.
SignatureMethod	The algorithm used to produce the digital signature. The digest and canonicalization algorithms are taken into account when creating 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 e reference construct that hosts digest and optional transformation details.

Basic structure

The basic structure of the XML signature is as follows:

<signature>
<signedinfo>
<canonicalizationmethod/
>
<signaturemethod/>
<reference>
<transforms>
<digestmethod>
<digestValue>
</reference>
<reference/>
</signedinfo>
<signaturevalue/>
<keyinfo/>
<object/>
</signature>



UNIT V SERVICE ORIENTED ANALYSIS AND DESIGN

SOA delivery strategies – Service oriented analysis – Service Modelling – Service oriented design – Standards and composition guidelines -- Service design – Business process design – Case Study

PART – B

1. Explain in detail about SOA Delivery Strategies

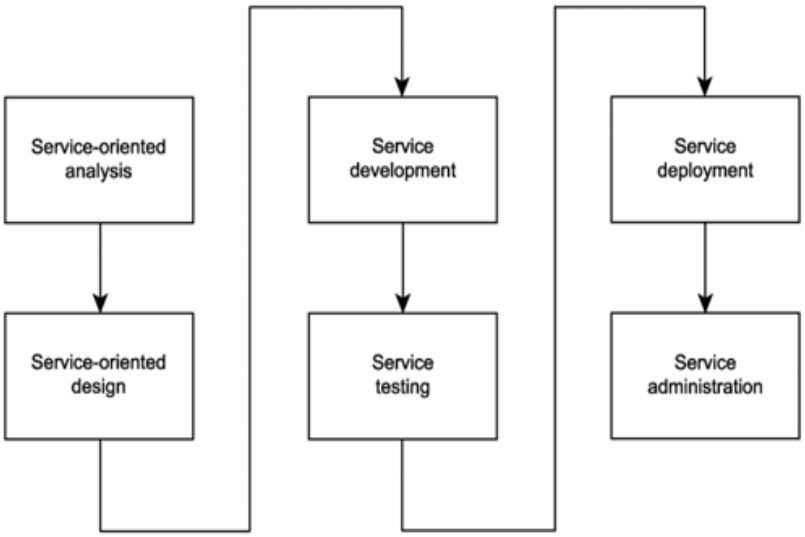
- Delivery strategies are arranging the stages into different sequences based on organizational priorities.

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.

10.1.1. 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.
- When we dig a bit deeper under the layers of service-orientation, though, we'll find that to properly construct and position services as part of SOA, traditional project cycles require some adjustments.
- The main reason we make this distinction is because it is during the analysis and design stages that the SOA characteristics and service-orientation principles are incorporated into the solution being built. So much so, that they warrant unique analysis and design processes that is distinctly "service-oriented."
- The service phases are primarily concerned with the delivery of services that implement the results of service-oriented analysis and design efforts.



Common phases of an SOA delivery lifecycle.

10.1.2. 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.
- A formal step-by-step service modeling process is dedicated to the service-oriented analysis phase.

10.1.3. 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 dedicated to the service-oriented design phase.

10.1.4. 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.

10.1.5. 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.

10.1.6. 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.

10.1.7. Service administration

- After services are deployed, standard application management issues come to the fore front.
- 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).

10.1.8. SOA delivery strategies

- The lifecycle stages identified in the previous sections represent a simple, sequential path to building individual services.

To organize stages into a process

- 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
- 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.
- 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)

- These paths differ in priorities and practical considerations. The following three sections provide process descriptions and explore the pros and cons of each approach.
- How you approach the creation of a service-oriented environment ultimately determines what you will end up with. The strategies discussed here, therefore, will confront you with some important decision-making. Choosing the right approach will determine the extent to which your service-oriented modeling and design efforts can realize the full potential of SOA.

2. Explain in detail about the top-down strategy of SOA Delivery Strategies

- 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.

10.2.1. Process

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

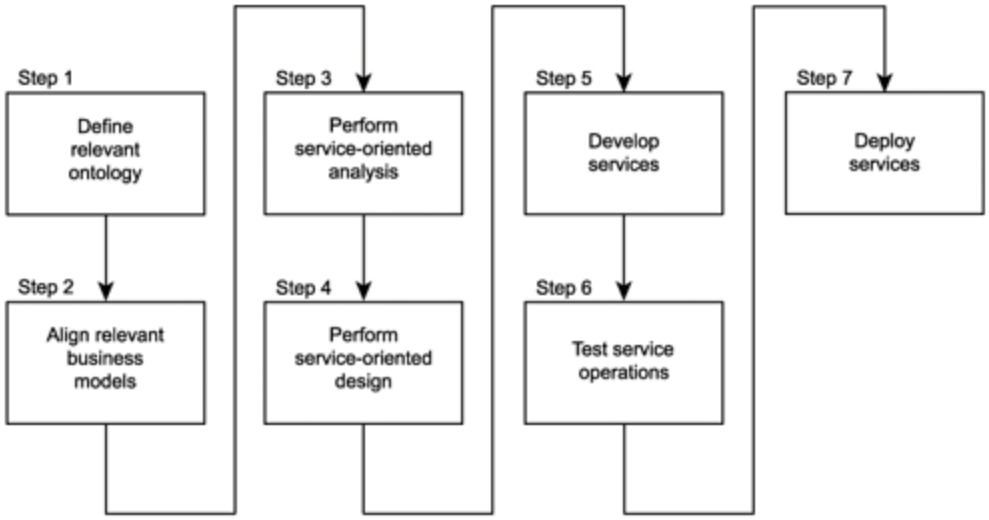


Figure. Common top-down strategy process steps.

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.
- ✓ 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, such as the one described in Chapters 11 and 12, is completed.

Step 4: Perform service-oriented design

- ✓ The service layers are formally defined as part of a service-oriented design process, such as the one described in Chapters 13 through 16.

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.

10.2.2. 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 (proportional to the size of the organization and the immediate solution), without showing any immediate results.

3. Explain in detail about the bottom-up strategy of SOA Delivery Strategies

- 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.

10.3.1. Process

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

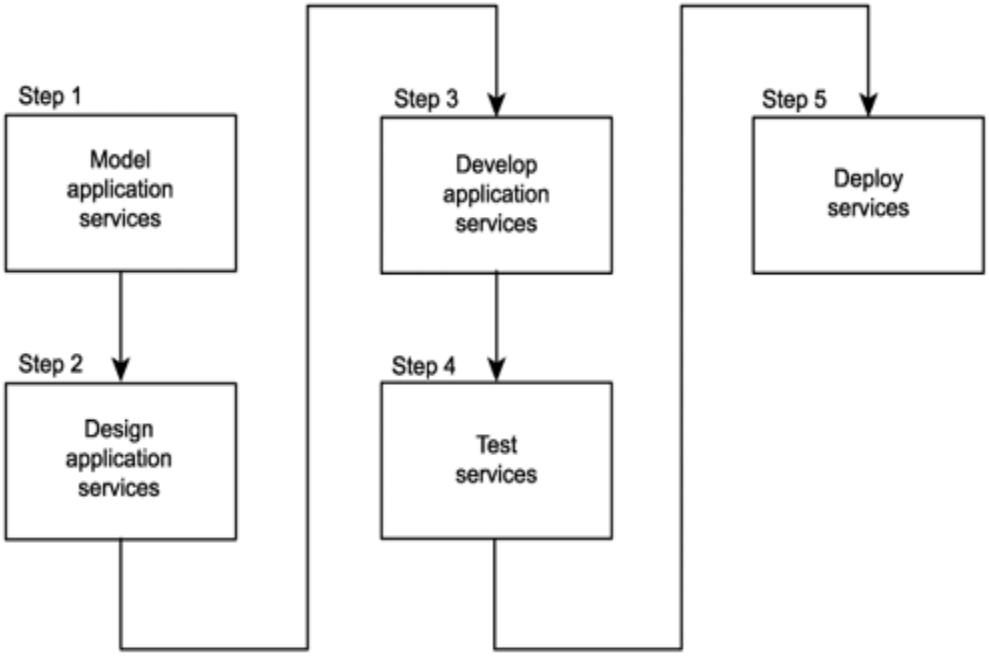


Figure. Common bottom-up strategy process steps.

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. Chapter 15 provides a design process specifically for reusable application services.

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.

10.3.2. 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.

4. Explain in detail about the agile strategy of SOA Delivery Strategies

- 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.

10.4.1. Process

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

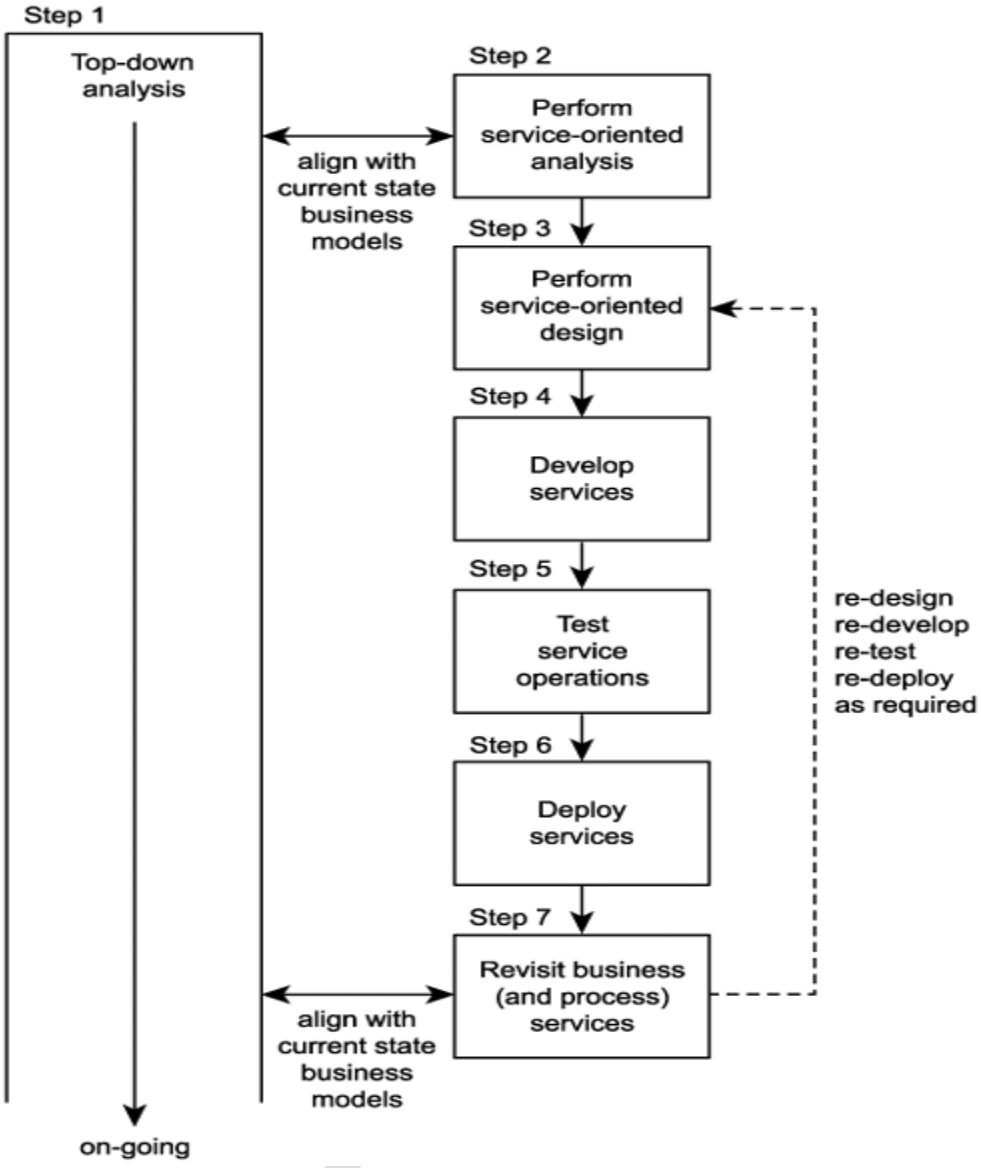


Figure. A sample agile strategy process.

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. The further along it progresses, the more service designs will benefit.
- ✓ 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. It may require an educated judgment call to determine whether the on-going top-down analysis is sufficiently mature to proceed with the creation of business service models. This consideration must then be weighed against the importance and urgency of pending project requirements.

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.

10.4.2. 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.

5. Explain in detail about the 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.

11.1.1. 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. Many of the issues we discussed in the past two chapters can be part of this stage. Specifically, the determination of which service layers to build and how to approach their delivery are critical decision points that will end up forming the structure of the entire service-oriented environment.
- 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.

11.1.2. 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.
- The process described in this section is not intended to supplant existing procedures. Instead, it proposes a sequence of supplementary steps, specific to the delivery of a service-oriented solution.
- Service-oriented analysis can be applied at different levels, depending on which of the SOA delivery strategies are used to produce services. As explained in the previous chapter, the chosen 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. 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 are common tasks associated with this phase and are described further in the following sections.

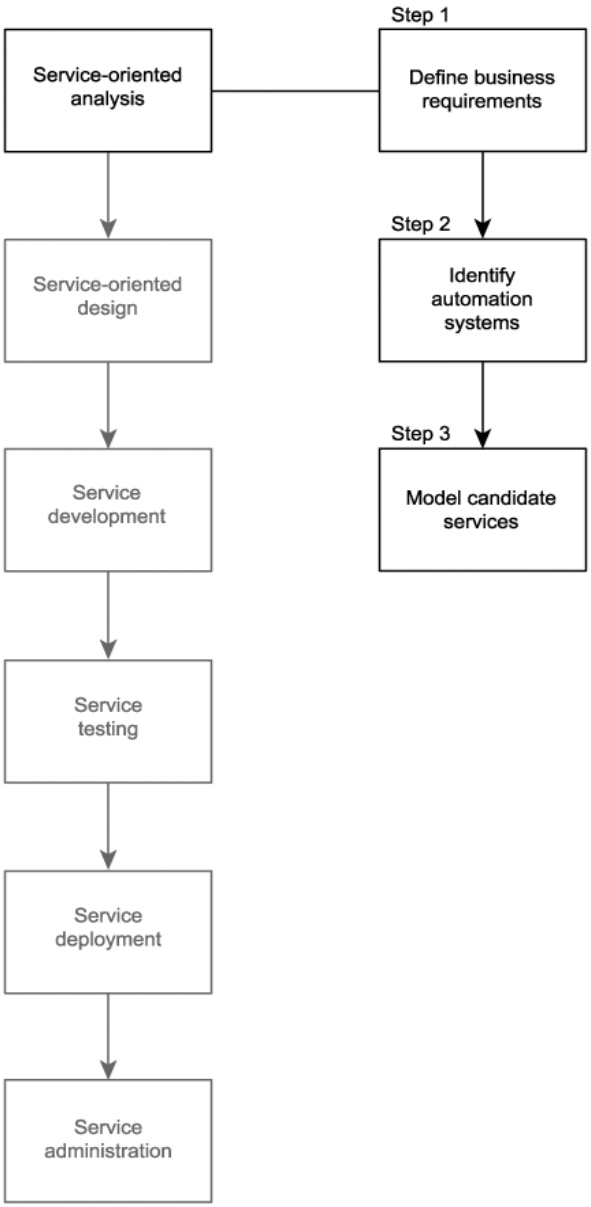


Figure. A high-level service-oriented analysis process.

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. 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.
- The details of how Web services relate to existing systems are ironed out in the service-oriented design phase. For now, this information will be used to help identify application service candidates during the service modeling process described in Step 3.

Step 3: Model candidate services

- A service-oriented analysis introduces the concept of service modelinga 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.

6. Explain in detail about the Service Modeling of Service-Oriented Analysis

- A service modeling process is essentially an exercise in organizing the information we gathered in Steps 1 and 2 of the parent service-oriented analysis process.
- Sources of the information required can be diverse, ranging from various existing business model documents to verbally interviewing key personnel that may have the required knowledge of a relevant business area. As such, this process can be structured in many different ways.

- The process described in this section is best considered a starting point from which you can design your own to fit within your organization's existing business analysis platforms and procedures.

12.1.1. "Services" versus "Service Candidates"

- 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.
- The reason this distinction is so relevant is because once our candidates are submitted to the design process, they are subjected to the realities of the technical architecture in which they are expected to reside. Once constraints, requirements, and limitations specific to the implementation environment are factored in, the end design of a service may be a significant departure from the corresponding original candidate.
- So, at this 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 (Figure). Specifically, this particular process provides steps for the modeling of an SOA consisting of application, business, and orchestration service layers.

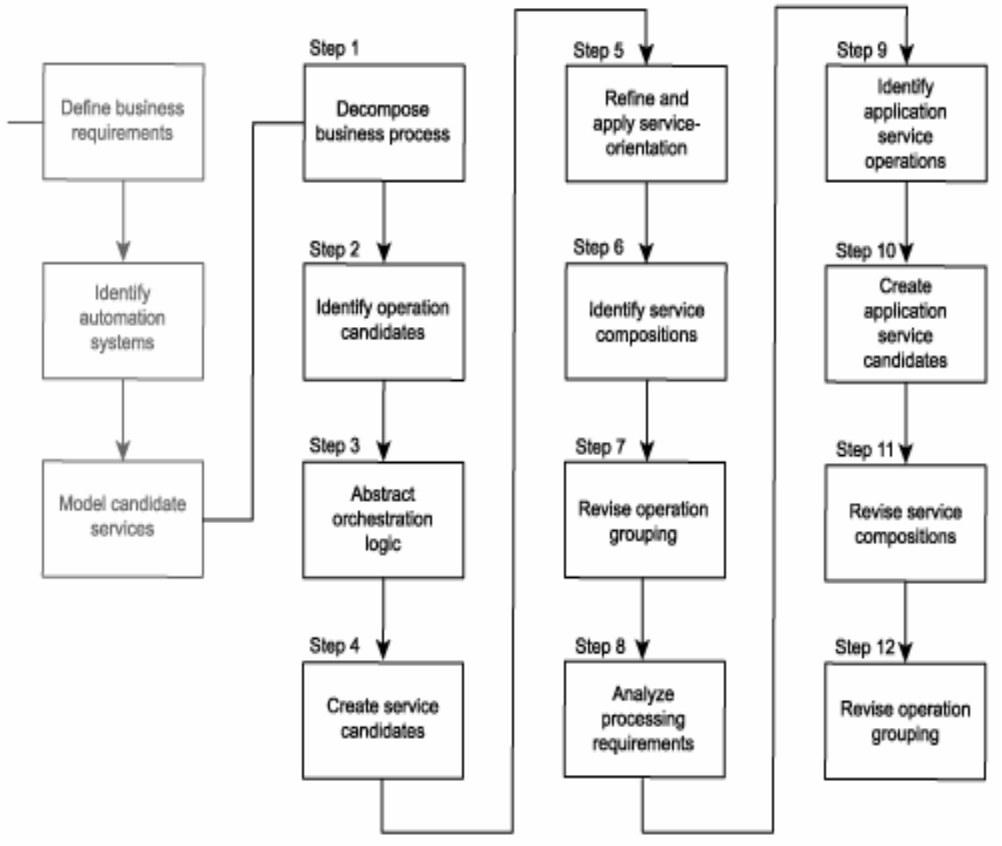


Figure. A sample service modeling process.

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.

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:
  - o Manual process steps that cannot or should not be automated.
  - o 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:
  - o business rules
  - o conditional logic
  - o exception logic
  - o sequence logic
- ✓ For example, some processing step descriptions consist of a condition and an action (if condition x occurs, then perform action y). In this case, only remove the condition and leave the action.
- ✓ Also note that some of the identified workflow logic likely will be dropped eventually. This is because not all processing steps necessarily become service operations. Remember that at this point, we are only creating candidates. When we enter the service-oriented design phase, practical considerations come into play. This may result in the need to remove some of the candidate operations, which would also require that corresponding workflow logic be removed from the orchestration layer as well.

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.
- ✓ For example, task-centric business services will require a context specific to the process, while entity-centric business services will introduce the need to group processing steps according to their relation to previously defined entities.
- ✓ 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.

Step 5: Refine and apply principles of service-orientation

- ✓ 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.
- ✓ This step gives us a chance to make adjustments and apply key service-orientation principles. This is where the study we performed in the Native Web service support for service-orientation principles section of Chapter 8 becomes useful. As you may recall, we identified the following four key principles as those not intrinsically provided through the use of Web services:
  - o reusability
  - o autonomy
  - o statelessness
  - o discoverability
- ✓ Of these four, only the first two are important to us at the service modeling stage. The latter two on this list are addressed in the service-oriented design process. Therefore, our focus in this step is also to ensure that each service operation candidate we identify is potentially reusable and as autonomous as possible.

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:
  - o It gives you a good idea as to how appropriate the grouping of your process steps is.
  - o It demonstrates the potential relationship between orchestration and business service layers.
  - o It identifies potential service compositions.

- It highlights any missing workflow logic or processing steps.

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.

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. It requires that you more closely study the underlying processing requirements of all service candidates to abstract any further technology-centric service candidates from this view that will complete a preliminary application services layer. To accomplish this, each processing step identified so far is required to undergo a mini-analysis.
- ✓ Specifically, what you need to determine is:
  - What underlying application logic needs to be executed to process the action described by the operation candidate.
  - Whether the required application logic already exists or whether it needs to be newly developed.
  - Whether the required application logic spans application boundaries. In other words, is more than one system required to complete this action?

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:

- association with a specific legacy system
- association with one or more solution components
- logical grouping according to type of function

✓ Various other issues are factored in once service candidates are subjected to the service-oriented design process. For now, this grouping establishes a preliminary application service layer.

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.

7. Explain in detail about the 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.

13.1.1. 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.

13.1.2. "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.

13.1.3. 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 (Figure 13.1).

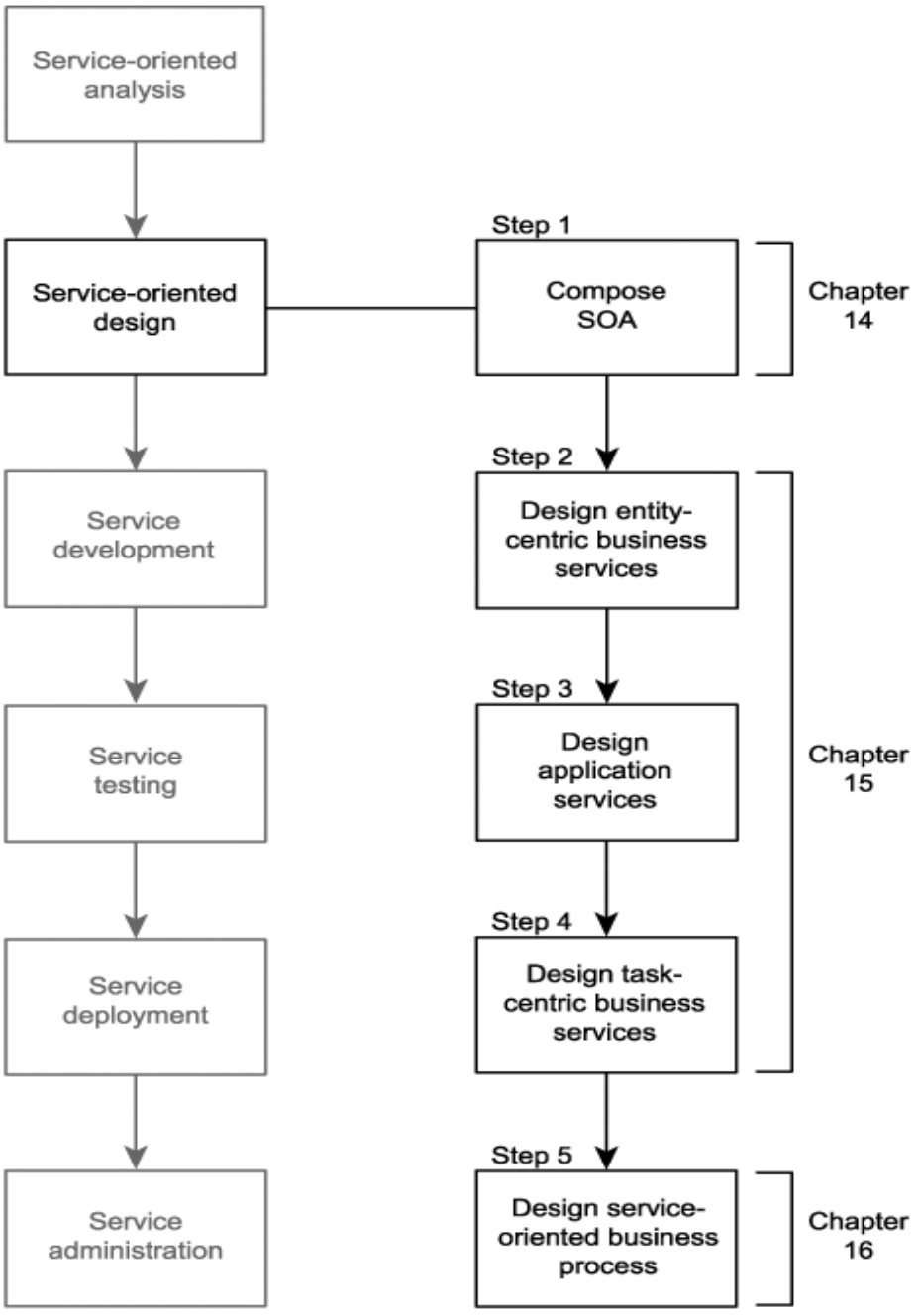


Figure. A high-level service-oriented design process.

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

Steps 2 to 4: Design services

- ✓ 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.
- ✓ Our primary input for each of these service design processes is the corresponding service candidates we produced in the service modeling process during the service-oriented analysis.

Step 5: Design service-oriented business process

- ✓ Upon establishing an inventory of service designs, we proceed to create our orchestration layerthe 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

13.1.4. Prerequisites

- Before we get into the details of the service-oriented design process, we should make sure that we have a sufficient understanding of key parts of the languages required to design services.

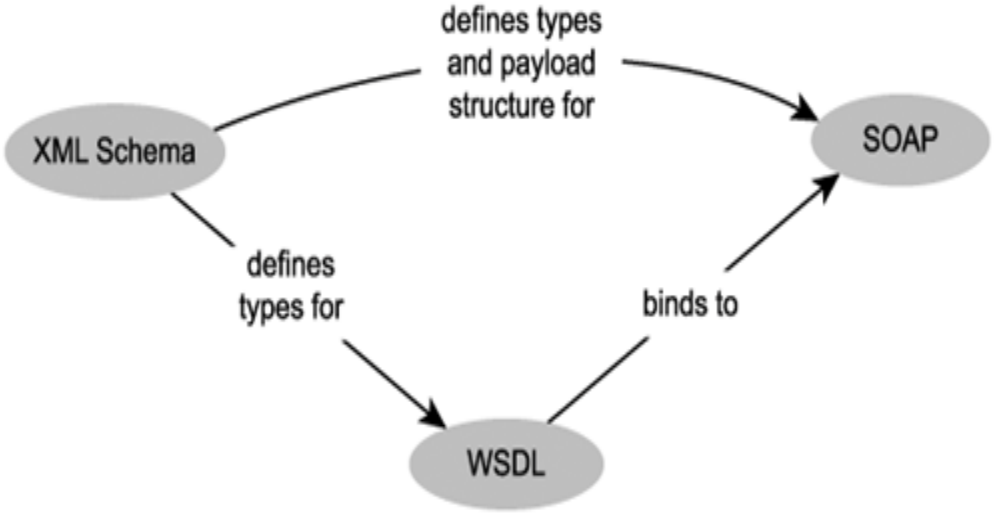


Figure. Three core specifications associated with service design.

8. Explain in detail about the SOA Standards and Composition Guidelines.

- Defining architecture is a common step within the delivery lifecycle of any form of automation solution project. It establishes application boundaries, the supporting technology set, and target deployment environments.
- However, SOA brings with it unique characteristics that differ from traditional architectural models. This also leads to a unique approach to defining the architecture itself.

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 (Figure). 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

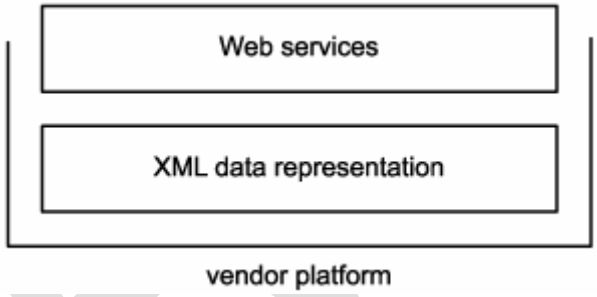


Figure. The most fundamental components of an SOA.

- 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.
- Provided in Figure and further described in the following sections is an informal set of steps for composing a service-oriented architecture. Depending on your goals and the nature of your technical environment, additional considerations likely will be needed.

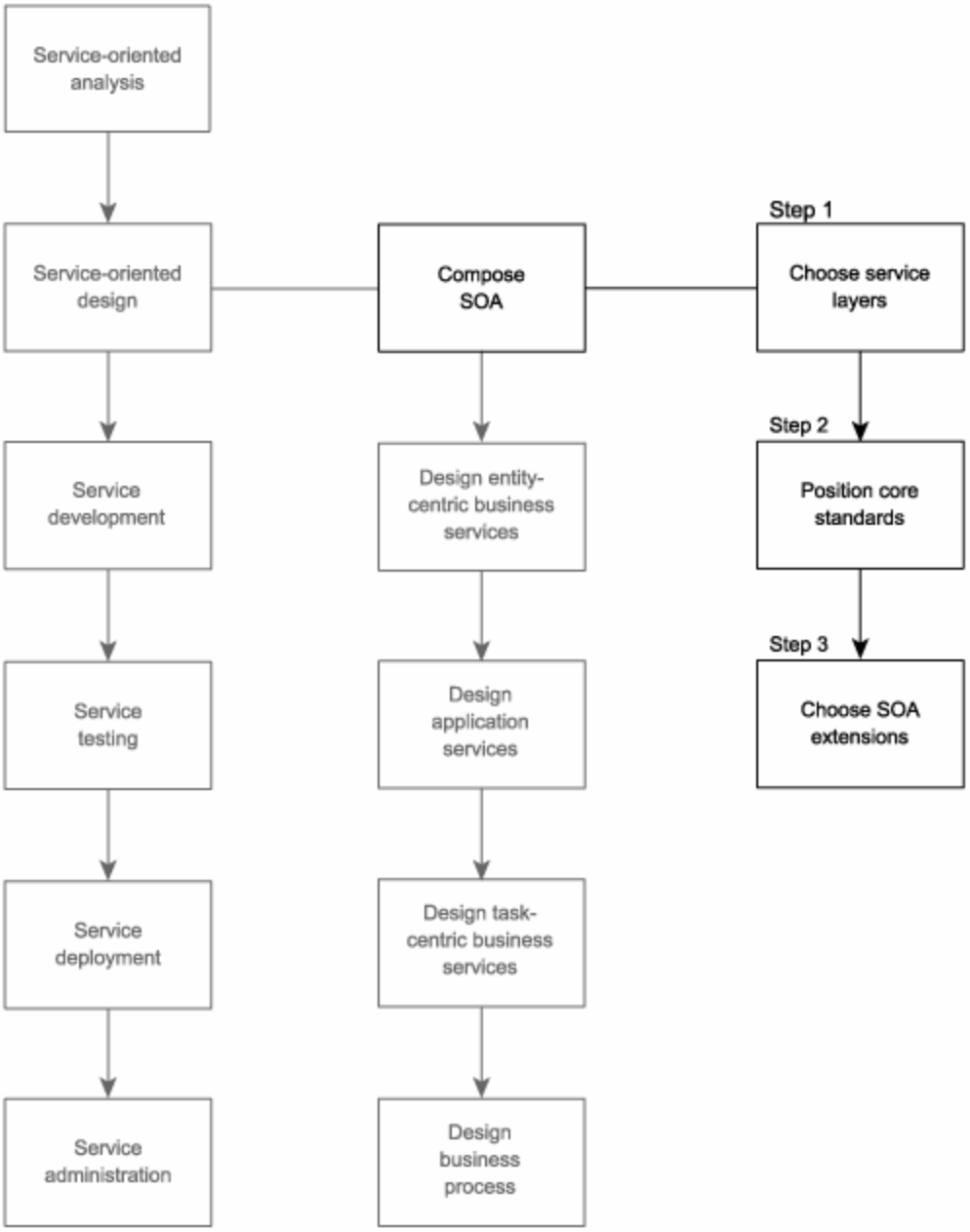


Figure 14.2. Suggested steps for composing a preliminary SOA.

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. Some guidelines are provided in the Considerations for choosing service layers section.

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.

9. Explain in detail about the SOA Service design.

- 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:
    - encapsulate the required amount of logic
    - conform to service-orientation principles
    - meet business requirements
- Given the four main types of service layers we identified previously, following is a suggested design sequence:
    1. Entity-centric business services
    2. Application services

3. Task-centric business services
4. Process services

- 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.
- This results in you having to revisit the application service designs to determine if you should add operations or entirely new services.

15.1.1. Design standards

- It is important to note that a firm set of design standards is critical to achieving a successful SOA. Because the design we are defining as part of this phase is concrete and permanent, every service produced needs to be as consistent as possible.
- Otherwise, many key SOA benefits, such as reusability, composability, and especially agility, will be jeopardized. It is therefore assumed that prior to starting this phase, design standards are already in place.
- In previous service-oriented analysis process, design standards were not as heavily emphasized. This is primarily because service candidates can continue to be modified and refined after corresponding services have been developed and implemented, without significant impact. Standards are still relevant to service-oriented analysis, but not as much as they are integral to service-oriented design.

15.1.2. About the process descriptions

- The sample processes in this section consist of generic sets of steps that highlight the primary considerations for creating service designs. This is our last chance to ensure a service properly expresses its purpose and capabilities.
- As part of each abstract service description we create, the following parts are formally defined:
  - definition of all service operations
  - definition of each operation's input and output messages

- definition of associated XSD schema types used to represent message payloads

15.1.3. Prerequisites

- Service design processes approach the creation of the service interface from a hand coding perspective. This means that many references to the WSDL and XSD schema markup languages are made throughout the process descriptions.
- Further, to support our processes, numerous interspersed case study examples provide actual WSDL and XSD schema markup samples. Reading through the WSDL and XSD tutorials provided in Chapter 13 therefore is recommended to best understand the process descriptions and associated examples.

The Service-Oriented Design Process

Step 1: Compose SOA

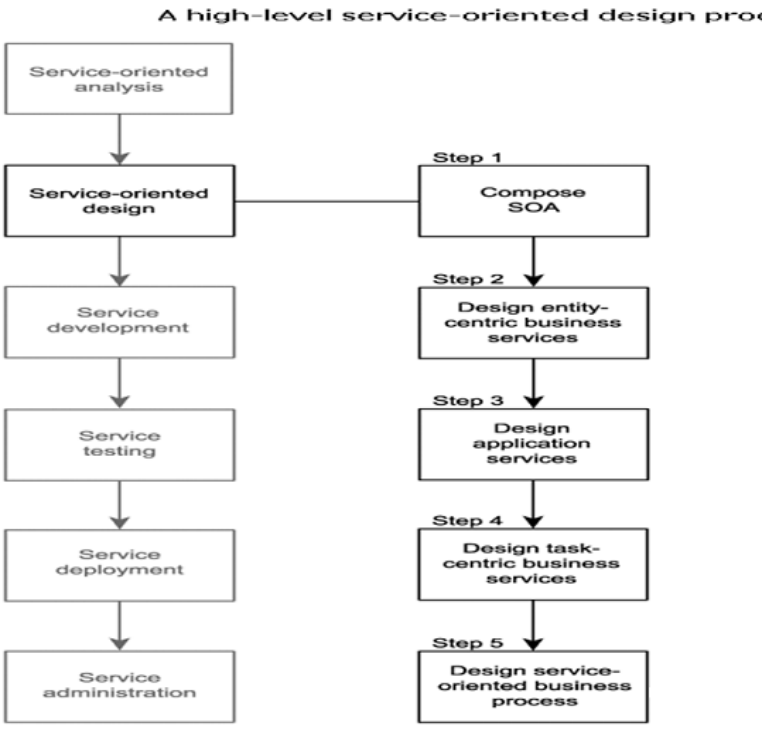
Each instance of a service-oriented architecture is uniquely composable which is used to implement a common set of shared technologies based on key XML and first-generation web services specification

The step consists of the following **three steps**

1. **Choose service layers**
   - Existing services, SOA applications, ect.
   - Enterprise standards
   - Composition performance consideration
2. **Position core SOA standards**
   - Industry standards
   - Will you use UDDI?
3. **Choose SOA extensions**
   - Which extensions will you use in which compositions?

Step 2: Entity-centric business service design process

- Entity services are also known as entity-centric business services
- Establish the business service layer
  - Based on its functional boundary and context on one or more related business entities



Step 3: Application (utility) services design process

- Utility services are also known as application services
- Establish the utility services layer.

Step 4: Task-centric business service design process

- Task services are also known as task-centric business services.
- Complete the business service layer(s)
- The primary input is the corresponding service candidates we produced in the service modeling process during the service-oriented analysis.

Step 5: Design service-oriented business process

The formal, executable definitions of workflow logic are translated into the creation of a WS-BPEL process definition.

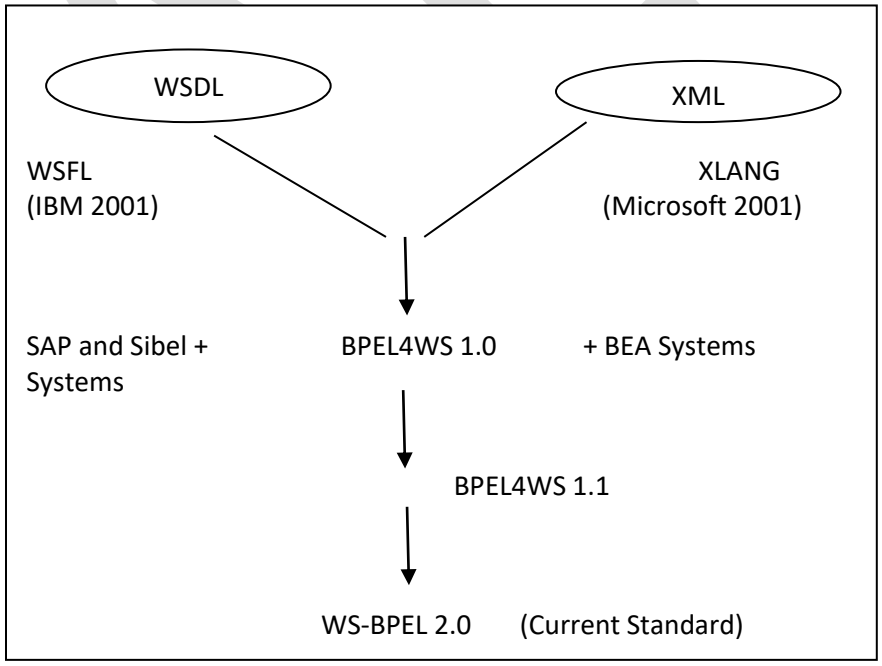
10. Explain in detail about the Business Process Design.

- 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.
- **WS-BPEL** stands for **web services business process Execution Language**. WS-BPEL is an **XML based language.** Enabling user to describe business process activities as web services and define how they can be connected to accomplish specific tasks.
- Composition is based on pre-modeled workflow.
- In WS-BPEL everything is a service.

WS-BPEL family tree

WS-BPEL 2.0 is a revision of the original acronym BPEL4WS 1.0 and 1.1.



WS-BPEL terminology

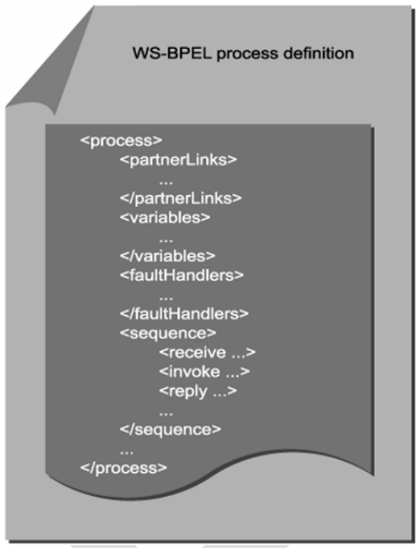
- **Activities**

- Message exchange or intermediate result transformation

- **Process-the composition result**
  - A process consists of a set of activities.

The structure of WS-BPEL process

A common WS-BPEL process definition structure.



- A WS-BPEL process definition is represented at runtime by the process service.
- Services that participate in WS-BPEL defined processes are considered partner services and are established as part of the process definition.
- Numerous activity elements are provided by WS-BPEL to implement various types of process logic.

  The common WS-BPEL process definition structures are given below:

The process element

An <process> element is the root element and must have a name attribute for assigning the name value. It is used to establish the process definition-related namespaces.

The child elements of <process> are listed in the following section.

Process definition skeleton

<process name="TimesheetSubmissiomProcess"
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>
……..
</variable>
<sequence>
……..
</sequence>
…….
</process>

The partnerLinks and partnerLinks elements

The partnerLinks define the services that are orchestrated by the process. It contains a set of <partnerLink> element each represent the communication exchange between two partners-the process service being one partner and another service being the other.

The partnerLink element

The partnerLink element therefore contains the following attributes

- ***myRole***
  - Used when the process service is invoked by a partner client service.
  - Process service acts as the service provider.
- ***partnerRole***
  - Identifies the partner service that the process service will be invoking
  - Partner service acts as the service provider.

Example

<partnerLink>
<partnerLink name=”ClientStartUplink”
partnerLinkType=”wsdl:ClientStartUpPLT”myRole=”Client”/>
</partnerLink>

The partnerLinkType element

The partnerLinkType elements identify the WSDL portType elements refrenced by the partnerLink elements within the process definition.

It contains one role element for each role the service can play, so it will have either one or two child role elements.

The variables element

Variables hold the data that constitute the state of a BPEL business process during runtime.

Attributes

- ***MessageType***

  Allow for the variable to contain an entire WSDL-defined message.

- ***Element***

  Refer to an XSD element construct.

- ***Type***

  Used to just represent an XSD simpleType, such as string or integer.

Example

```
<variables>
<variable name=”myVar1”messageType=”myNS:myWSDLMessageDataType”/>
<variable name=”myVar1”element=”myNS:myXMLElement”/>
<variable name=”myVar2”type=”xsd:string”/>
<variable name=”myVar2”type=”myNS:myComplexType”/>
</variables>
```

WS-BPEL Functions

a) The getVariableProperty function

The getVariableProperty function allows global property values to be retrived from variables. It simply accepts the variable and property names as input and returns the requested value.

Syntax

getVariableProperty (variable name, property name)

Example

getVariableProperty (TicketApproval’,’Class’)

b) The getVariableData function

The getVariableData function has a mandatory variable name parameter and two optional arguments that can be used to specify a part of the variable data.

Syntax

getVariableData (variable name, part name, location path)

Example

getVariableData ('input','payload','/tns:TimesheetType/Hours/...')

Basic Activities

a) The invoke element

The <invoke> activity is used to invoke the web service operations provided by partners.

Attributes

Attributes specify the details of the invocation. There are five common attributes equipped with invoke element

Attribute	Description
partnerLink	this element names the partner service via its corresponding partner Link
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 b 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 a messageType 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.

Example

The invoke element identifying the target partner service details.

<invoke name="ValidateWeeklyHours"
partnerLink="Employee"
portType="emp:EmployeeInterface"
operation="GetWeeklyHoursLimit"
inputVariable="EmployeeHoursRequest"
outputVariable="EmployeeHoursResponse"/>

b) The receive element

A <receive> activity is used to receiverequests in a BPEL business process to provide services to its partners. The process block until the message is received.

Attributes

The attributes value relates the expected incoming communication.

Attribute	Description
partnerLink	the client partner identified in the corresponding partnerLink 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.

Example

```
<receive name="receiveInput"
partnerLink="client"
portType='tns:TimesheetSubmissionInterface"
operation="ClientSubmission"
createInstance="yes"/>
```

c) The reply element

A <reply> activity is used to send a response to a request previously accepted through a <receive> activity. Responses are used for synchronous request/reply interactions.

Example

The following examples show the reply message for the above receive element.

```
<reply partnerLink="client"
portType="tns:TimesheetSubmissionInterface"
operation="Submit"
variable="TimesheetSubmissionResponse"/>
```

Structured Activity

a) The sequence element

The sequence construct is to organize a series of activities so that they are executed in a predefined, sequential order. Nesting of sequence is allowed.

Structure

A skeleton sequence constructs containing only some of the many activity elements provided by WS-BPEL

```
<sequence>
<receive>....</receive>
<assign>....</assign>
<invoke>....</invoke>
<reply>....</reply>
```

b) The switch, case, and otherwise elements

The switch statement is a control statement that handles multiple selections by passing:

- Control to one of the case statements within its body.
- Control is transferred to the corresponding case statement whose condition attribute is true. If all the preceding case conditions fail, then the otherwise construct is executed.

Structure

```
<switch>
<case
condition="getVariableData('EmployeeResponseMessage',ResponseParameter')=
0">
....
</case>
<otherwise>
....
</otherwise>
</switch>
```

Flow

The <flow> activity provides concurrent execution of enclosed activities and their synchronization.

While

The while construct executes the contained activity as long/until a given conditions becomes true.

Pick

The pick construct will block and wait for one event from a given set of events(an event is an incoming message or a time-out alarm)

If /else

The If/Else construct selects exactly one branch from a set of choice.

Scope

Scopes define behavior contexts for activities. They provide local partner links, message exchanges, variables, correlation sets, fault handlers, compensation handlers, termination handlers, and event handlers for activities.

The assign, copy, from, and to elements

The <assign> activity is used to:

- Copy data from on variable to another
- Construct and insert new data using expressions and literal values
- Copy partner link endpoint references

Example

Within this assign construct, the contents of the TimesheetSubmissionFailedMessage variable are copied to two different message variables.

```
<assign>
<copy>
<from vvariable="TimsheetSubmissionFailedMessage"/>
<to variable="EmployeeNotificationMessage"/>
</copy>
</assign>
```

The faultHandlers, catch, and catchall element

Fault handlers are used to react to fault that occur while the business process activities are executing. The *faultHandlers* constructs contain multiple catch element and a catchall child constructs.

Catch

The <catch> activity is used to specify faults that are to be caught and handled. At least one <catch> activity needs to be specified.

CatchAll

The <catchAll> activity is used to catch all faults. It is optional.

Syntax

```
<faultHandlers>
<catch faultName="somethingBadHappened"
faultVariable="TimesheetFault">
....
</catch>
<catchAll>
Activity
</catchAll>
```

</faultHandlers>

The compensationHandler element

The compensationHandler element is used to define compensation activities. Compensation handlers gather all activities that have to be carried out to compensate another activity.

Syntax

<compensationHandler>

Activity

</compensationHandler>

The <coorrelationSets> element

A correlation set is a set of properties share by messages and used for correlation. It is used to associate a message with a business process instance. Each correlation set has a name attribute.

Syntax

<coorrelationSets>?

<correlationSet name=”NCName” properties=’QName-list”/>+

</coorrelationSets>

The <empty> element

An activity that does nothing is defined by the <empty> tag.

Syntax

<empty standard-attributes>

Standard-elements

</empty>

Example

<empty/>

The <wait> element

A <wait> activity is used to specify a delay for a certain period of time or until a certain deadline is reached.

11. Discuss the TLS analysts and architects embark on designing a corresponding WS-BPEL process definition

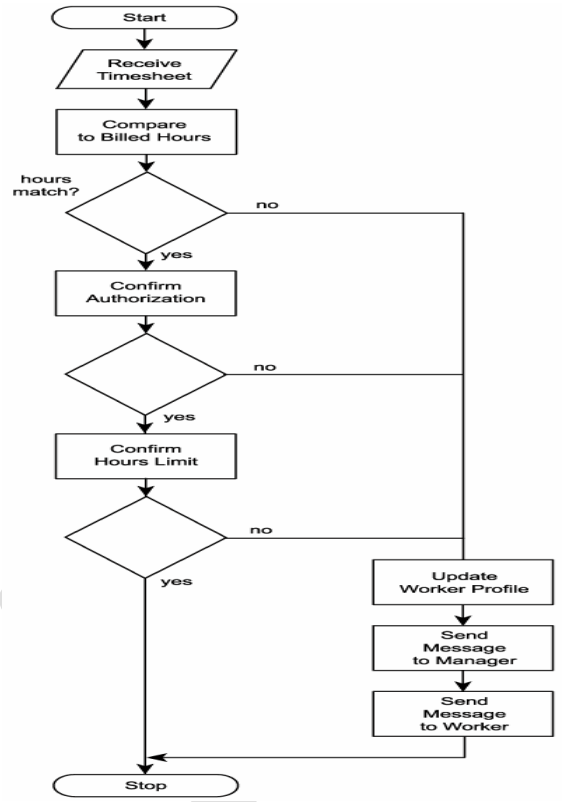


Figure 16.4. The original TLS Timesheet Submission Process.

- As part of completing the previous service design processes, TLS now has the inventory of service designs displayed in Figure 16.5.
- The other service designs are provided here to help demonstrate the WS-BPEL partner links we define later on.

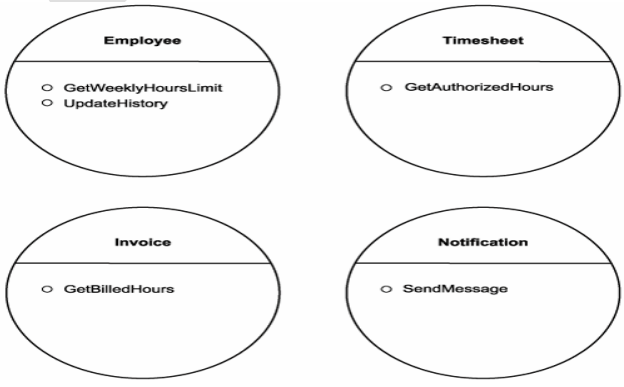


Figure 16.5. Service designs created so far.

- TLS also digs out the original composition diagram (Figure 16.6) that shows how these four services form a hierarchical composition, spearheaded by the Timesheet Submission Process Service TLS plans to build.

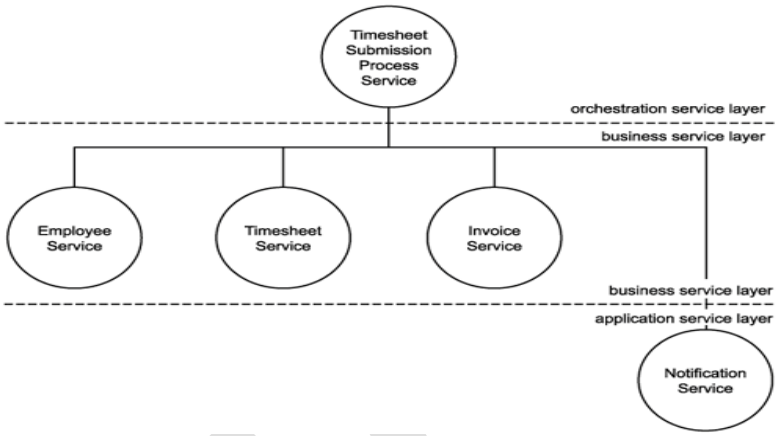


Figure 16.6. The original service composition defined during the service modeling stage.

- Finally, TLS architects revive the original service candidate created for the Timesheet Submission Process Service (Figure 16.7).

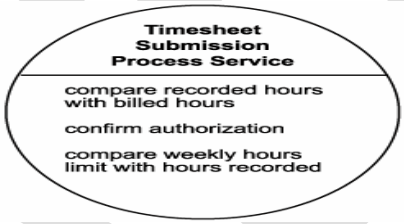


Figure 16.7. The Timesheet Submission Process Service candidate.

- With all of this information in hand, TLS proceeds with the business process design.

Step 1: Map out interaction scenarios

- TLS maps out a series of different service interaction scenarios using activity diagrams. Following are examples of two scenarios.
- Figure 16.8 illustrates the interaction between services required to successfully complete the Timesheet Submission Process with a valid timesheet submission. Note that in this scenario, the Notification Service is not used.

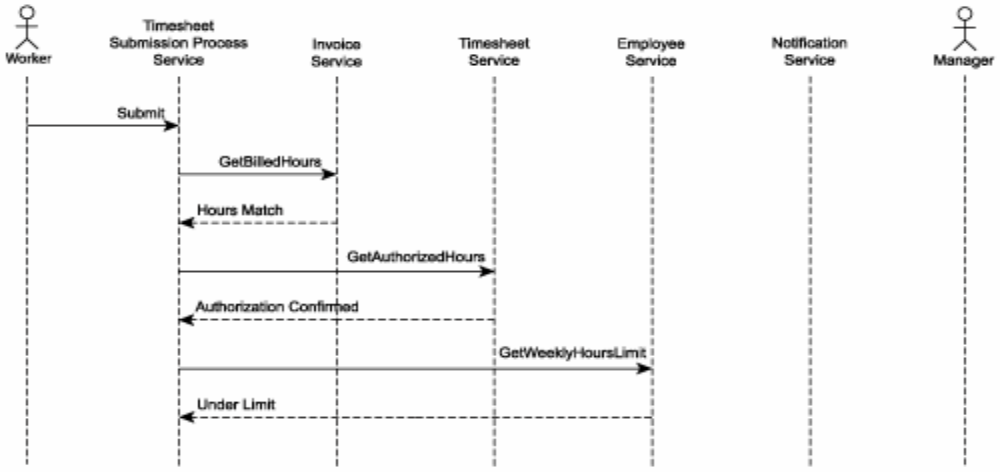


Figure 16.8. A successful completion of the Timesheet Submission Process.

- Figure 16.9 demonstrates a scenario in which the timesheet document is rejected by the Timesheet Service. This occurs because the timesheet failed to receive proper authorization.

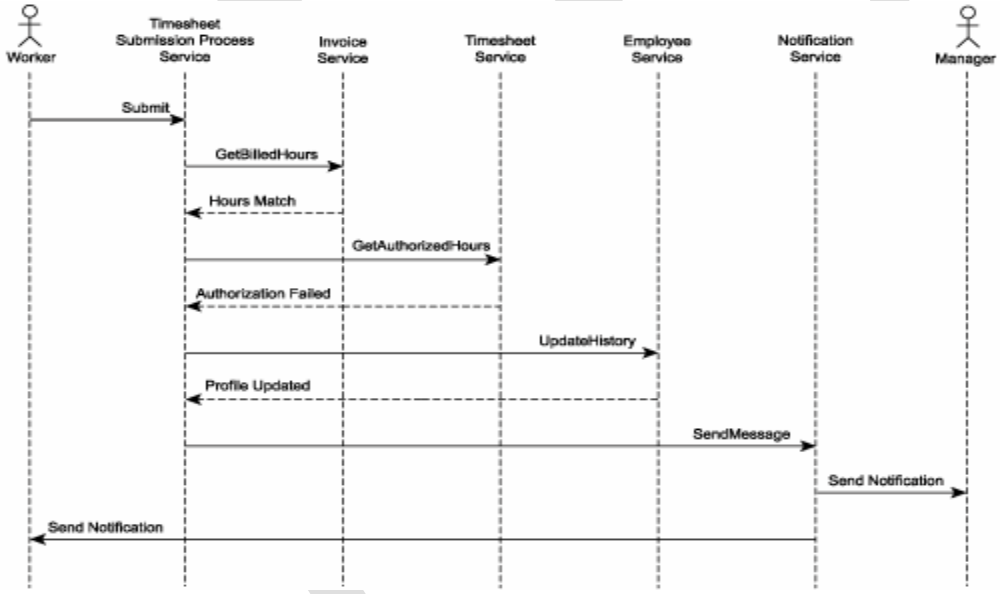
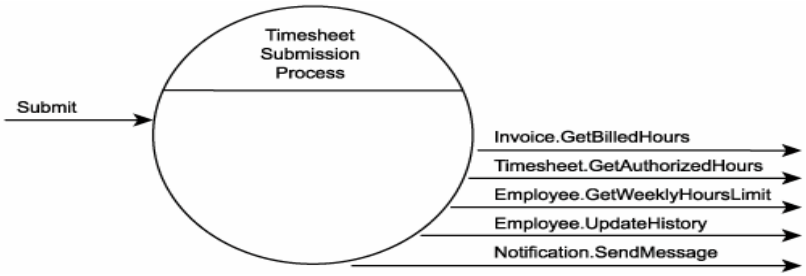


Figure 16.9. A failure condition caused by an authorization rejection.

- The result of mapping out interaction scenarios establishes that the process service has one potential client partner service and four potential partner services from which it may need to invoke up to five operations (Figure 16.10).



Step 2: Design the process service interface

- It looks like the Timesheet Submission Process Service interface will be pretty straightforward. It only requires one operation used by a client to initiate the process instance (Figure 16.11).

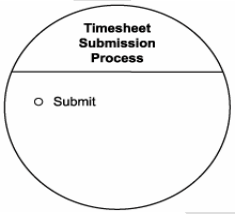


Figure 16.11. Timesheet Submission Process Service design.

- Example 16.18. The abstract service definition for the Timesheet Submission Process Service.

<xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">

Initiates the Timesheet Submission Process.

<name="tns:TimesheetSubmissionInterface"/>

Step 3: Formalize partner service conversations

- Now that the Timesheet Submission Process Service has an interface, TLS can begin to work on the corresponding process definition. It begins by looking at the information it gathered in Step 1. As you may recall, TLS determined the process service as having one potential client partner service and four potential partner services from which it may need to invoke up to five operations.
- Roles are assigned to each of these services, labeled according to how they relate to the process service. These roles are then formally defined by appending existing service WSDL definitions with a partnerLinkType construct.
- Example 16.19 shows how the Employee Service definition is amended to incorporate the WS-BPEL partnerLinkType construct and its corresponding namespace.

Example 16.19. The revised Employee service definitions construct.

```
<xmlns:plnk= "http://schemas.xmlsoap.org/ws/2003/05/partner-
link/">
...
```

- This is formalized within the process definition through the creation of partnerLink elements that reside within the partnerLinks construct. TLS analysts and architects work with a process modeling tool to drag and drop partnerLink objects, resulting in the following code being generated.
- Example 16.20. The partnerLinks construct containing partnerLink elements for each of the process partner services.

```
<partnerLinks>
 <partnerLink name="client"
 partnerLinkType="bpl:TimesheetSubmissionProcessType"
 myRole="TimesheetSubmissionProcessServiceProvider"/>
 <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>
```

- Next the input and output messages of each partner service are assigned to individual variable elements, as part of the variables construct. A variable element also is added to represent the Timesheet Submission Process Service Submit operation that is called by the HR client application to kick off the process.
- Example 16.21. The variables construct containing individual variable elements representing input and output messages from all partner services and for the process service itself.

```
<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"/>
<variable name="InvoiceHoursRequest"
messageType="inv:getBilledHoursRequestMessage"/>
<variable name="InvoiceHoursResponse"
messageType="inv:getBilledHoursResponseMessage"/>
<variable name="TimesheetAuthorizationRequest"
messageType="tst:getAuthorizedHoursRequestMessage"/>
<variable name="TimesheetAuthorizationResponse"
messageType="tst:getAuthorizedHoursResponseMessage"/>
<variable name="NotificationRequest"
messageType="not:sendMessage"/>
variables>
```

Step 4: Define process logic

- The TLS team now creates a process definition that expresses the original workflow logic and processing requirements, while accounting for the two service interaction scenarios identified earlier. The remainder of this example explores the details of this process definition.
- A visual representation of the process logic about to be defined in WS-BPEL syntax is displayed in Figure 16.12.

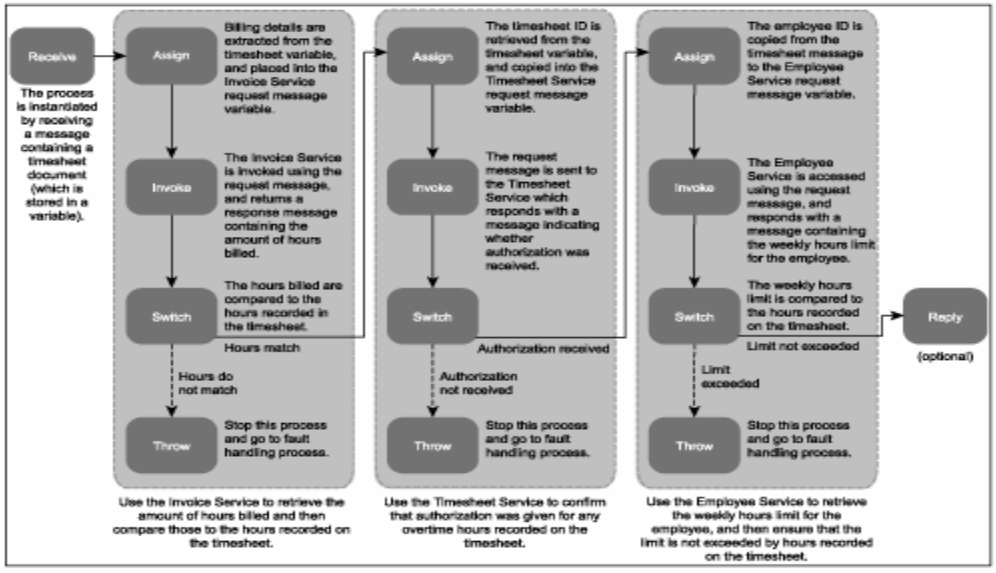


Figure 16.12. A descriptive, diagrammatic view of the process definition logic.

- Established first is a receive element that offers the Submit operation of the Timesheet Submission Process Service to an external HR client as the means by which the process is instantiated.
- Example 16.22. The receive element providing an entry point by which the process can be initiated.

```
<receive xmlns=
 "http://schemas.xmlsoap.org/ws/2003/03/business-process/"
 name="receiveInput"
 partnerLink="client"
 portType="tns:TimesheetSubmissionInterface"
 operation="Submit"
 variable="ClientSubmission"
 createInstance="yes"/>
```

- Example 16.23. The assign and copy constructs hosting a from element that retrieves customer billing information from the message stored in the ClientSubmission variable and a to element that is used to assign these values to the InvoiceHoursRequest variable.

```
<assign name="GetInvoiceID">
 <copy>
 <from variable="ClientSubmission" part="payload"
 query="/TimesheetType/BillingInfo"/>
 <to variable="InvoiceHoursRequest"
 part="RequestParameter"/>
 copy>
assign>
```

- Example 16.24. The invoke element containing a series of attributes that provide all of the information necessary for the orchestration engine to locate and instantiate the Invoice Service.

```
<invoke name="ValidateInvoiceHours"
 partnerLink="Invoice"
 operation="GetBilledHours"
 inputVariable="InvoiceHoursRequest"
 outputVariable="InvoiceHoursResponse"
 portType="inv:InvoiceInterface"/>
```

- Example 16.25. The switch construct hosting a case element that uses the getVariableData function within its condition attribute to compare hours billed against hours recorded.

```
<switch name="BilledHoursMatch">
 <case condition=
 "getVariableData('InvoiceHoursResponse',
 'ResponseParameter') !=
 getVariableData('input','payload',
 '/tns:TimesheetType/Hours/...')">

 case>
switch>
```

- If the condition (billed hours is not equal to invoiced hours) is not met, then the hours recorded on the submitted timesheet document are considered valid, and the process moves to the next step.
- If the condition is met, a fault is thrown using the throw element. This circumstance sends the overall business activity to the faultHandlers construct, which resides outside of the main process flow. This is the scenario portrayed in the second of the two activity diagrams assembled by TLS in Step 1 and is explained later in this example.
- What TLS has just defined is a pattern consisting of the following steps:

1. Use the assign, copy, from, and to elements to retrieve data from the ClientSubmission variable and assign it to a variable containing an outbound message.
2. Use the invoke element to interact with a partner service by sending it the outbound message and receiving its response message.
3. Use the switch and case elements to retrieve and validate a value from the response message.
4. Use the throw element to trigger a fault, if validation fails.

- Example 16.26. The faultHandlers construct used in this process.

```
<faultHandlers>
 ...
faultHandlers>
```

- Although individual catch elements could be used to trap specific faults, TLS simply employs a catchAll construct, as all three thrown faults require the same exception handling logic.
- The tasks performed by the fault handler routine are:

1. Update employee profile history.
2. Send notification to manager.
3. Send notification to employee.

- To implement these three tasks, the same familiar assign and invoke elements are used. Figure 16.13 shows an overview of the fault handling process logic.

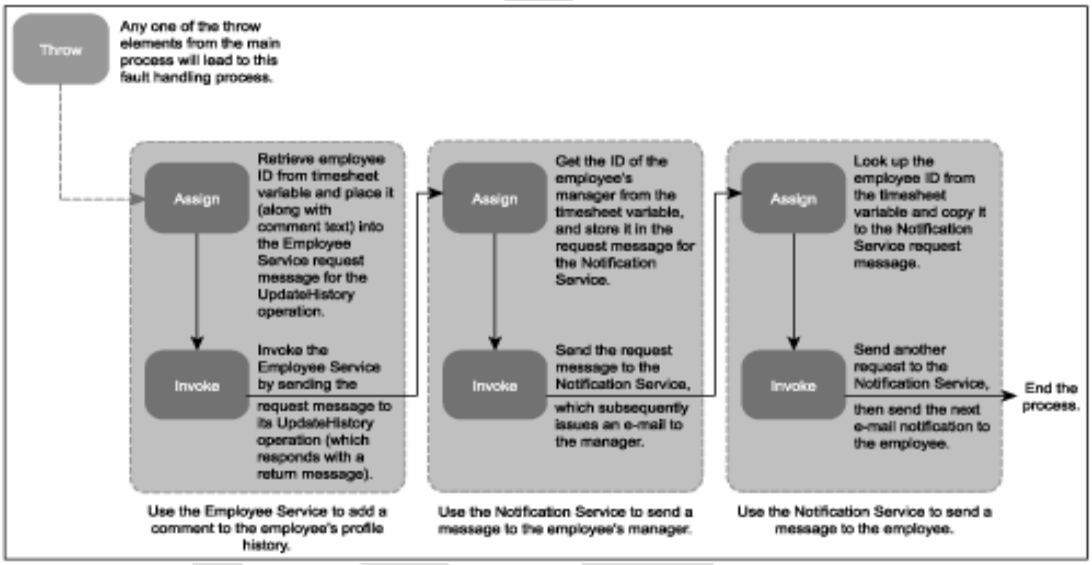


Figure 16.13. A visual representation of the process logic within the faultHandlers construct.

- Next, the second assign construct retrieves the EmployeeID value from the same ClientSubmission variable, which the Notification Service ends up using to send a message to the employee.
- The very last element in the construct, terminate, halts all further processing.

Step 5: Align interaction scenarios and refine process (optional)

- TLS analysts and architects revise their original activity diagrams so that they accurately reflect the manner in which process logic was modeled using WS-BPEL. However, in reviewing the interaction scenarios and their current process model, they recognize a key refinement that could significantly optimize the process definition they just created.
- The three primary tasks performed by this process:

1. Validate recorded timesheet hours with hours billed on invoice.
2. Confirm authorization of timesheet.
3. Ensure that hours submitted are equal to or less than the weekly hours limit.

- As shown in Figure 16.14, the process has been designed so that these three tasks execute sequentially (one begins only after the former ends). Although this approach is useful when dependencies between tasks exist, it is determined that there are no such dependencies between these three tasks.

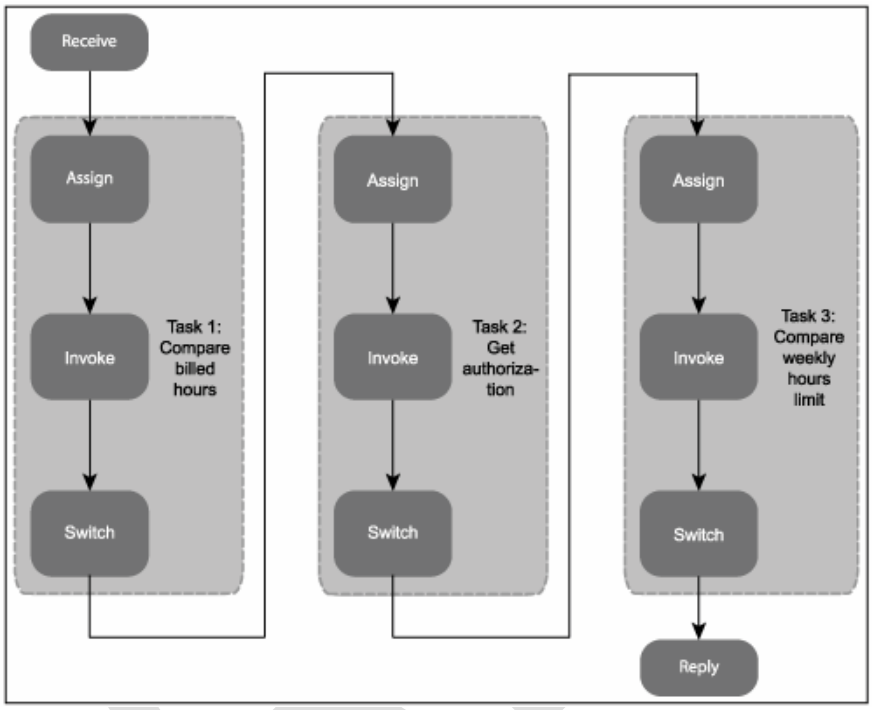


Figure 16.14. Sequential, synchronous execution of process activities.

- Therefore, they all can be executed at the same time, the only condition being that the process cannot continue beyond these tasks until all have completed. This establishes a parallel processing model.
- By utilizing the WS-BPEL flow construct, TLS can model the three activities to execute concurrently (Figure 16.15), resulting in significant performance gains. It is further determined that the same form of optimization can be applied to the process logic within the fault handling routine, as neither of those activities have inter-dependencies either.

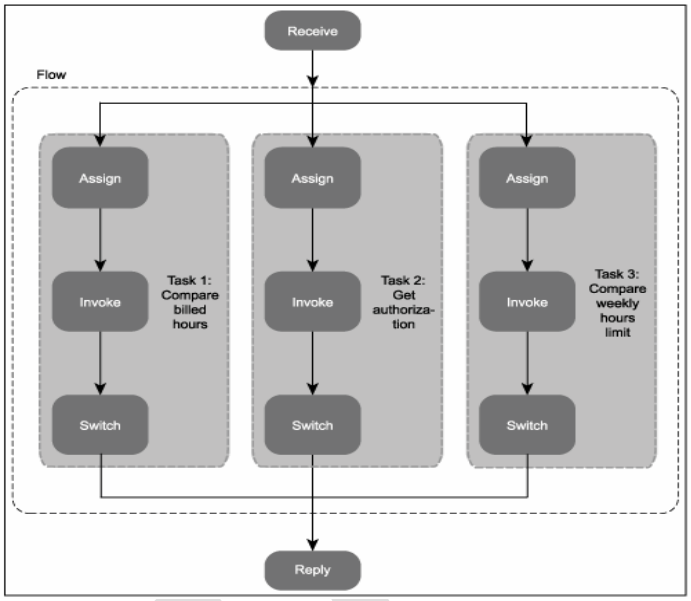


Figure 16.15. Concurrent execution of process activities using the flow construct.

- Finally, while reviewing the structure of the fault handling routine, a further refinement is suggested. Because the last two activities invoke the same Notification Service, they can be collapsed into a while construct that loops twice through the invoke element.