Service Oriented Architecture - *13MCA545*
Internal Assessment Test II – November 2016 – Answer Key

1. Explain in detail about Choreography in SOA (10)

Choreography

The Web Services Choreography Description Language (WS-CDL) is one of several specifications that attempts to organize information exchange between multiple organizations (or even multiple applications within organizations), with an emphasis on public collaboration

Figure 6.37. A choreography enables collaboration between its participants.

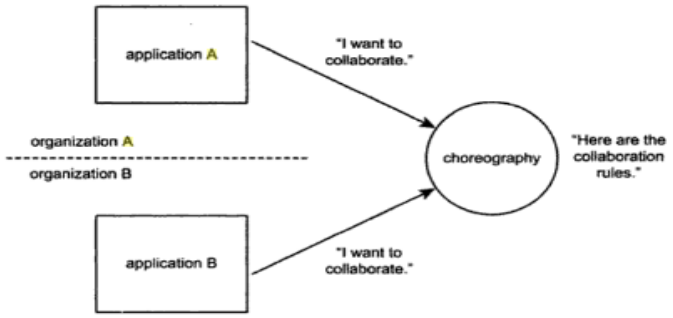


Figure 6.37
A choreography enables collaboration between its participants.

6.7.1. 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.

6.7.2. Roles and participants

- Within any given choreography, a Web service assumes one of a number of predefined *roles*.
- This establishes what the service does and what the service can do within the context of a particular business task.
- Roles can be bound to WSDL definitions, and those related are grouped accordingly, categorized as *participants* (services).

6.7.3. Relationships and channels

- Every action that is mapped out within a choreography can be broken down into a series of message exchanges between two services.
- Each potential exchange between two roles in a choreography is therefore defined individually as a *relationship*.
- Every relationship consequently consists of exactly two roles.

- *Channels* provides the means of establishing the nature of the conversation, by defining the characteristics of the message exchange between two specific roles.
- WS-CDL specification, as it fosters dynamic discovery and increases the number of potential participants within large-scale collaborative tasks.

6.7.4. Interactions and work units

- The actual logic behind a message exchange is encapsulated within an *interaction*.
- Interactions are the fundamental building blocks of choreographies because the completion of an interaction represents actual progress within a choreography.
- Related to interactions are *work units*.
- These impose rules and constraints that must be adhered to for an interaction to successfully complete

6.7.5. Reusability, composability, and modularity

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

Figure 6.38. A choreography composed of two smaller choreographies.

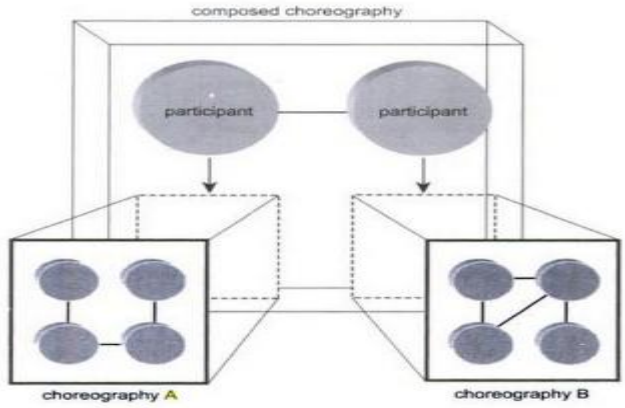


Figure 6.38
A choreography composed of two smaller choreographies.

6.7.6. Orchestrations and choreographies

- Both Orchestrations and choreographies represent complex message interchange patterns
- Both include multi-organization participants.
- An orchestration expresses organization-specific business workflow. This means that an organization owns and controls the logic behind an orchestration, even if that logic involves interaction with external business partners.
- A choreography, on the other hand, is not necessarily owned by a single entity. It acts as a community interchange pattern used for collaborative purposes by services from different provider entities

Figure 6.39. A choreography enabling collaboration between two different orchestrations

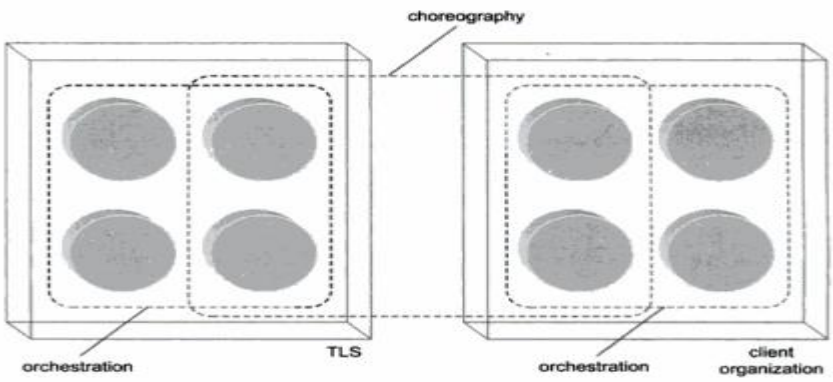


Figure 6.39
A choreography enabling collaboration between two different orchestrations.

2. a) Discuss about the anatomy of service oriented architecture (5)

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, as shown in Figure 8.5.

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).

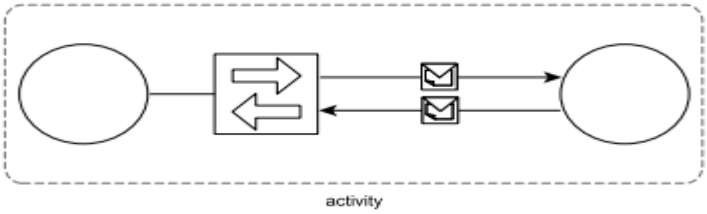


Figure 8.6
A basic communications scenario between Web services.

Logical components of automation logic

The Web services framework provides us not only with a technology base for enabling connectivity, it also establishes a modularized perspective of how automation logic, as a whole, can be comprised of independent units. To illustrate the inherent modularity of Web services, let's abstract the following fundamental parts of the framework:

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

8.2.3 Components of an SOA

Each of the previously defined components establishes a level of enterprise logic abstraction, as follows:
• A message represents the data required to complete some or all parts of a unit of work.
• An operation represents the logic required to process messages in order to complete a unit of work (Figure 8.9).

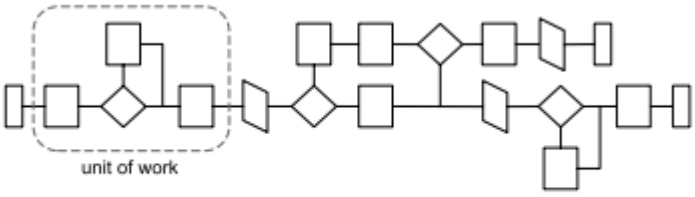


Figure 8.9
The scope of an operation within a process.

The scope of an operation within a process.
• A service represents a logically grouped set of operations capable of performing related units of work.
• A process contains the business rules that determine which service operations are used to complete a unit of automation. In other words, a process represents a large piece of work that requires the completion of smaller units of work (Figure 8.10).

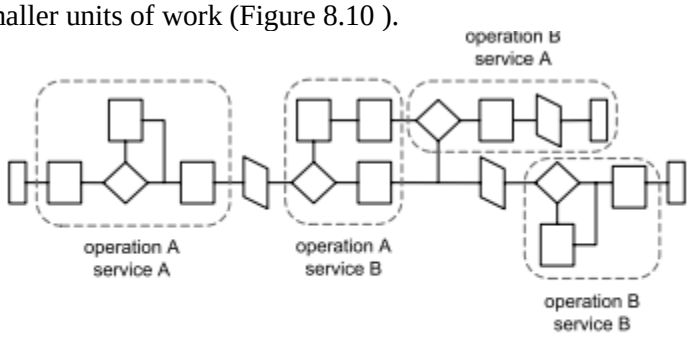


Figure 8.10
Operations belonging to different services representing various parts of process logic.

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

2. b) Explain about WS-Notification Framework (5)

7.7.3. The WS-Notification Framework

- WS-BaseNotification : Establishes the standardized interfaces used by services involved on either end of a notification exchange.
- WS-Topics : Governs the structuring and categorization of topics.
- WS-BrokeredNotification : Standardizes the broker intermediary used to send and receive messages on behalf of publishers and Subscribers

Situations, notification messages, and topics

- The notification process is tied to an event that is reported on by the publisher. This event is referred to as a *situation*. Situations can result in the generation of one or more *notification messages*.

- These messages contain information about the situation, and are categorized according to an available set of *topics*. Through this categorization, notification messages can be delivered to services that have subscribed to corresponding topics.

Notification producers and publishers

- The term *publisher* represents the part of the solution that responds to situations and is responsible for generating notification messages.
- Distribution of notification messages is the task of the *notification producer*.
- The notification producer is considered the service provider.
- This service keeps track of subscriptions and corresponds directly with subscribers. It ensures that notification messages are organized by topic and delivered accordingly.
- A publisher may or may not be a Web service, whereas the notification producer is always a Web service.
- A single Web service can assume both publisher and notification producer roles.

Notification consumers and subscribers

- A *subscriber* is the part of the application that submits the subscribe request message to the notification producer.
- This means that the subscriber is not necessarily the recipient of the notification messages transmitted by the notification producer. The recipient is the *notification consumer*, the service to which the notification messages are delivered
- The subscriber is considered the service requestor.
- A subscriber does not need to exist as a Web service, but the notification consumer is a Web service.
- Both the subscriber and notification consumer roles can be assumed by a single Web service.

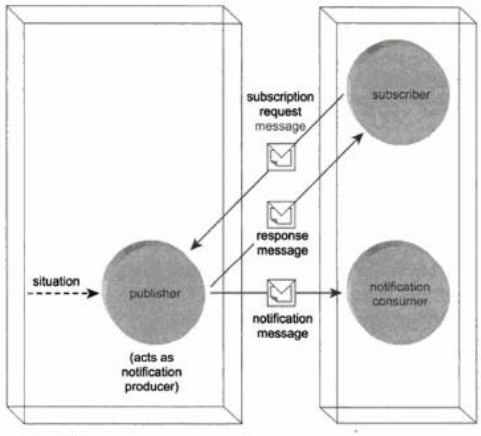


Figure 7.38
A basic notification architecture.

Notification broker, publisher registration manager, and subscription manager

- The *notification broker* A Web service that acts on behalf of the publisher to perform the role of the notification producer.
- The *publisher registration manager* A Web service that provides an interface for subscribers to search through and locate items
- available for registration.

- The *subscription manager* A Web service that allows notification producers to access and retrieve required subscriber information for a given notification message broadcast.

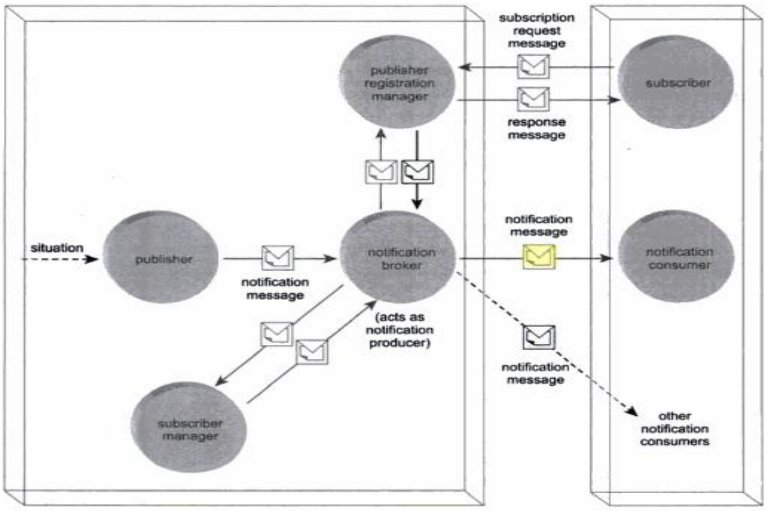


Figure 7.39
A notification architecture including a middle tier.

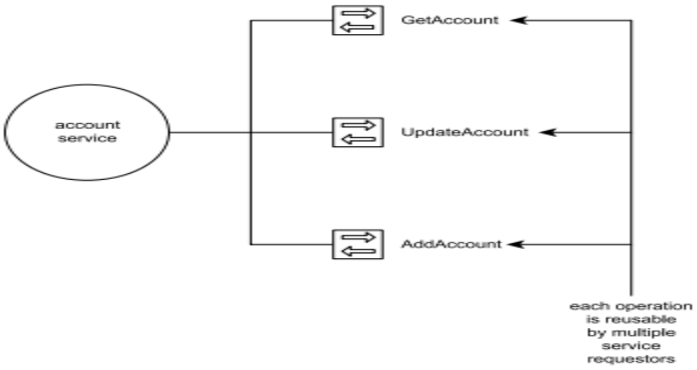
3. Discuss in detail about the common principles of service orientation (10)

- Services are reusable— Regardless of whether immediate reuse opportunities exist, services are designed to support potential reuse.
- Services share a formal contract— For services to interact, they need not share anything but a formal contract that describes each service and defines the terms of information exchange.
- Services are loosely coupled— Services must be designed to interact without the need for tight, cross-service dependencies.
- Services abstract underling logic— The only part of a service that is visible to the outside world is what is exposed via the service contract. Underlying logic, beyond what is expressed in the descriptions that comprise the contract, is invisible and irrelevant to service requestors.
- Services are composable— Services may compose other services. This allows logic to be represented at different levels of granularity and promotes reusability and the creation of abstraction layers.
- Services are autonomous— The logic governed by a service resides within an explicit boundary. The service has control within this boundary and is not dependent on other services for it to execute its governance.
- Services are stateless— Services should not be required to manage state information, as that can impede their ability to remain loosely coupled. Services should be designed to maximize statelessness even if that means deferring state management elsewhere.
- Services are discoverable—Services should allow their descriptions to be discovered and understood by humans and service requestors that may be able to make use of their logic.

8.3.1 Services are reusable

Service-orientation encourages reuse in all services, regardless if immediate requirements for reuse exist. By applying design standards that make each service potentially reusable, the chances of being able to accommodate future requirements with less development effort are increased. Inherently reusable services also reduce the need for creating wrapper services that expose a generic interface over top of less reusable

services.



8.3.2 Services share a formal contract

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

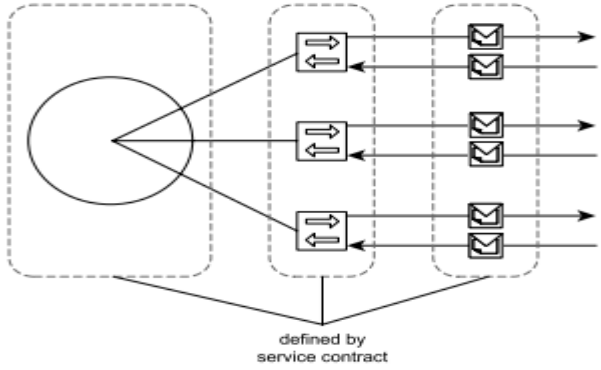


Figure 8.15
Service contracts formally define the service, operation, and message components of a service-oriented architecture.

Services are loosely coupled :

No one can predict how an IT environment will evolve. How automation solutions grow, integrate, or are replaced over time can never be accurately planned out because the requirements that drive these changes are almost always external to the IT environment. Being able to ultimately respond to unforeseen changes in an efficient manner is
a key goal of applying service-orientation

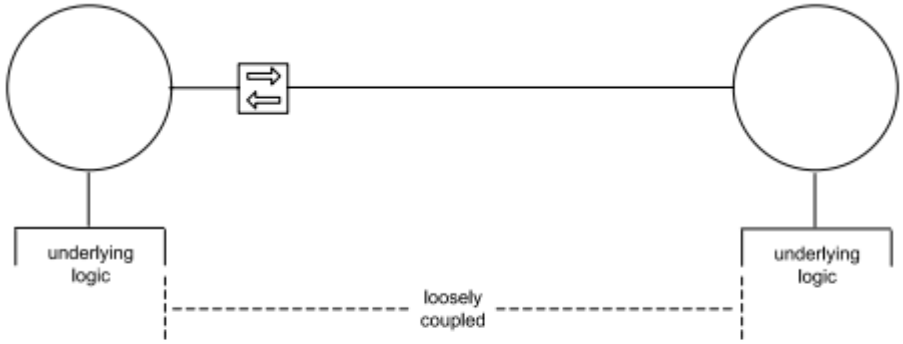


Figure 8.16
Services limit dependencies to the service contract, allowing underlying provider and requestor logic to remain loosely coupled.

Services abstract underlying logic

Also referred to as service interface-level abstraction, it is this principle that allows services to act as black boxes, hiding their details from the outside world. The scope of logic represented by a service significantly influences the design of its operations and its position within a process.

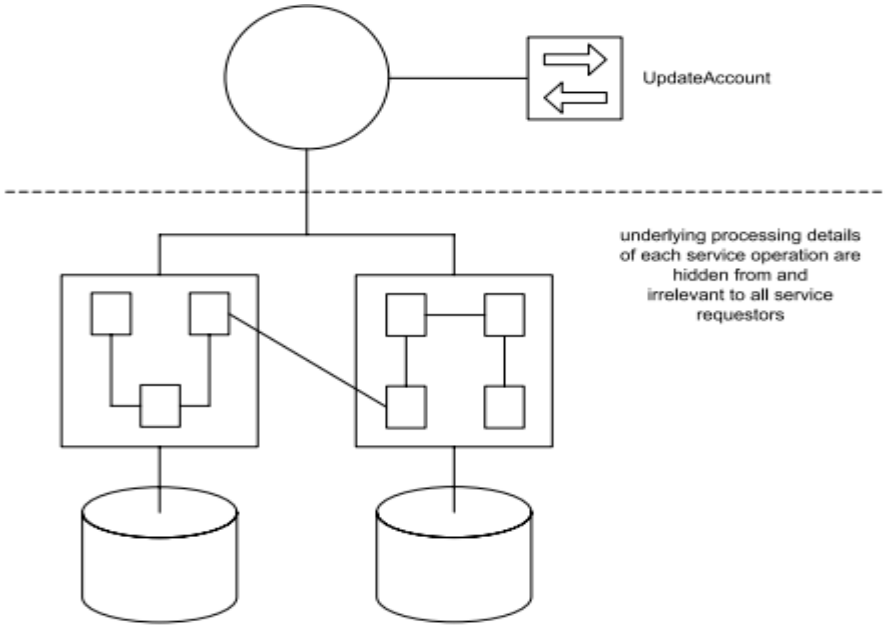


Figure 8.17
Service operations abstract the underlying details of the functionality they expose.

Services are composable

A service can represent any range of logic from any types of sources, including other services. The main reason to implement this principle is to ensure that services are designed so that they can participate as effective members of other service compositions if ever required. This requirement is irrespective of whether the service itself composes others to accomplish its work

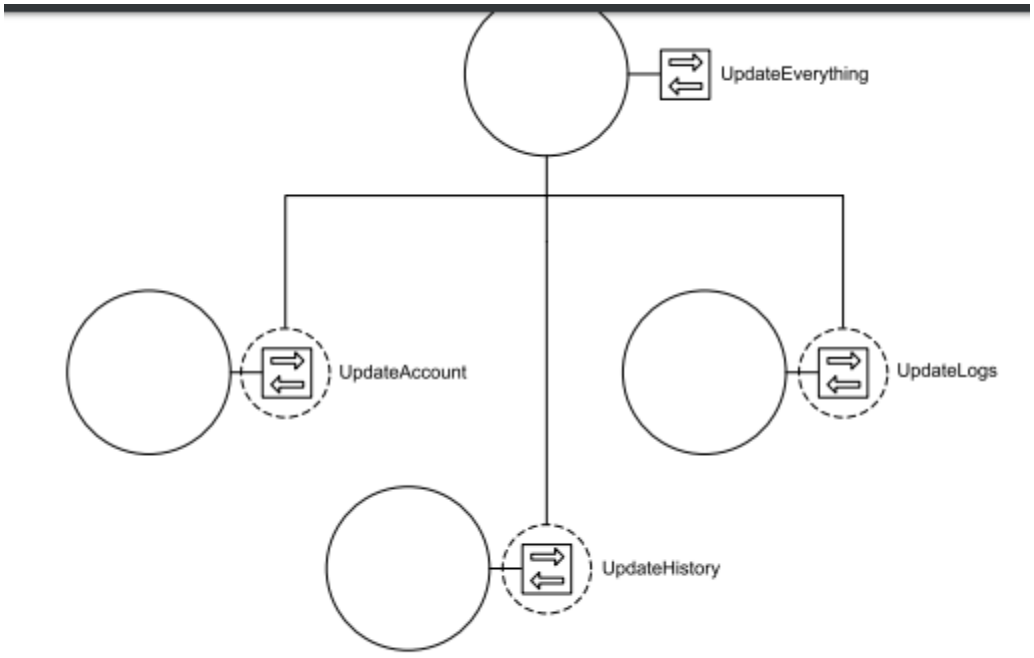


Figure 8.19
The UpdateEverything operation encapsulating a service composition.

Services are autonomous

Autonomy requires that the range of logic exposed by a service exist within an explicit boundary. This allows the service to execute self-governance of all its processing. It also eliminates dependencies on other services, which frees a service from ties that could inhibit its deployment and evolution (Figure 8.22). Service autonomy is a primary consideration when deciding how application logic should be divided up into services and which operations should be grouped together within a service context.

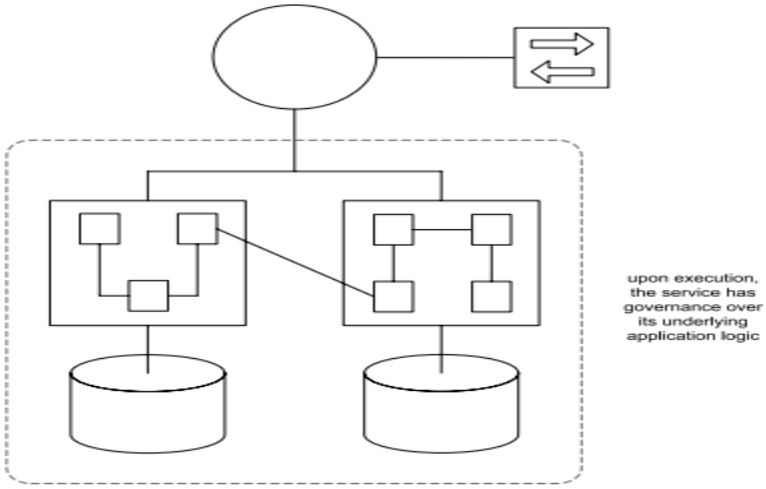


Figure 8.22
Autonomous services have control over underlying resources.

Services are stateless

Services should minimize the amount of state information they manage and the duration for which they hold it. State information is data-specific to a current activity. While a service is processing a message, for

example, it is temporarily stateful (Figure 8.24). If a service is responsible for retaining state for longer periods of time, its ability to remain available to other requestors will be impeded

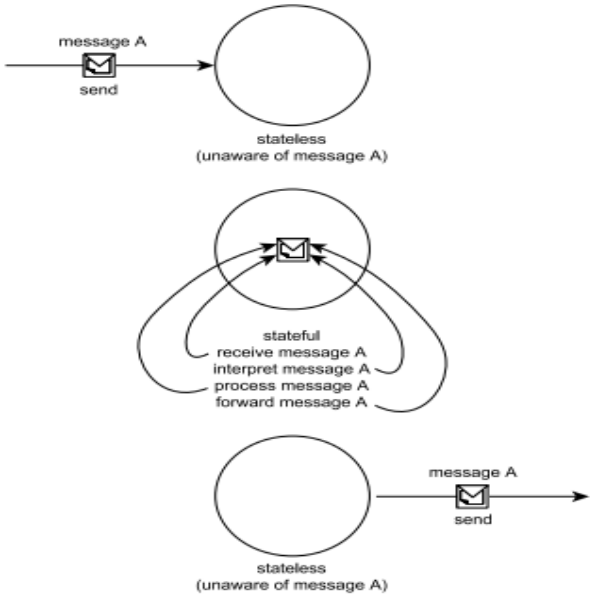


Figure 8.24
Stateless and stateful stages a service passes through while processing a message.

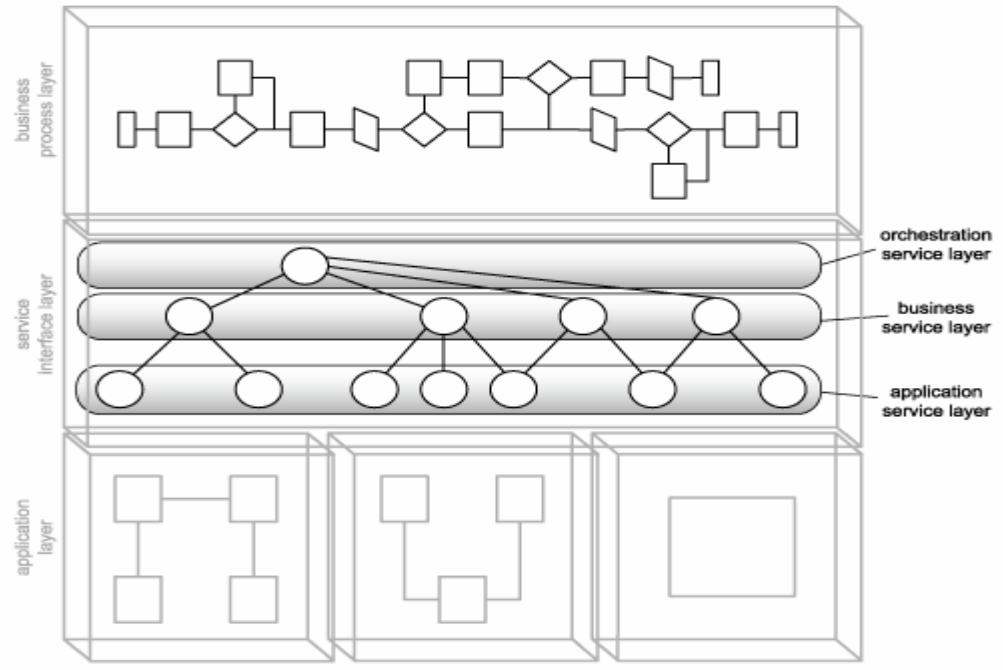
Services are discoverable

Discovery helps avoid the accidental creation of redundant services or services that implement redundant logic. Because each operation provides a potentially reusable piece of processing logic, metadata attached to a service needs to sufficiently describe not only the service's overall purpose, but also the functionality offered by its operations.

4. Write a short not about the following
   a) Service layer abstraction (5)

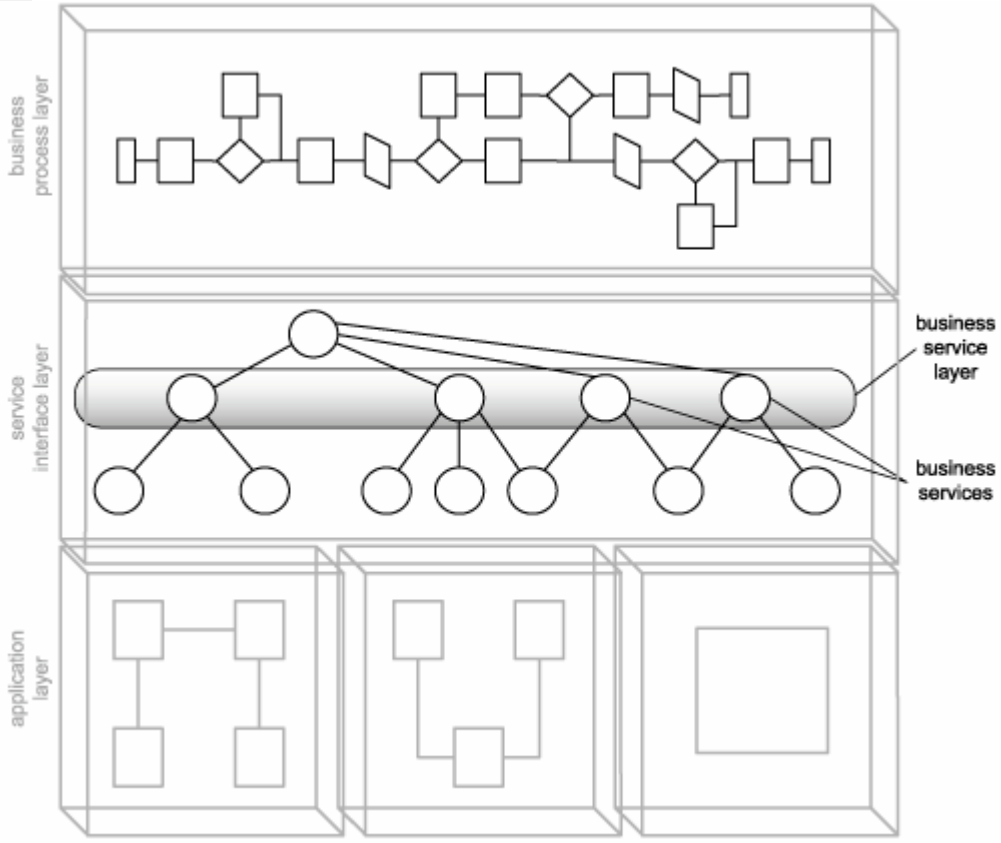
The three layers of abstraction we identified for SOA are:

- the application service layer
- the business service layer
- the orchestration service layer



b) Business Layer abstraction (5)

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



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

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

5. Compare the service orientation principles and object orientation principles (10)

Table 8.1. An overview of how service-orientation principles relate to object-orientation principles.

Service-Orientation Principle	Related Object-Orientation Principles
service reusability	Much of object-orientation is geared toward the creation of reusable classes. The object-orientation principle of modularity standardized decomposition as a means of application design. Related principles, such as abstraction and encapsulation, further support reuse by requiring a distinct separation of interface and implementation logic. Service reusability is therefore a continuation of this goal.
service contract	The requirement for a service contract is very comparable to the use of interfaces when building object-oriented applications. Much like WSDL definitions, interfaces provide a means of abstracting the description of a class. And, much like the "WSDL first" approach encouraged within SOA, the "interface first" approach also is considered an object-orientation best practice.
service loose coupling	Although the creation of interfaces somewhat decouples a class from its consumers, coupling in general is one of the primary qualities of service-orientation that deviates from object-orientation. The use of inheritance and other object-orientation principles encourages a much more tightly coupled relationship between units of processing logic when compared to the service-oriented design approach.
service abstraction	The object-orientation principle of abstraction requires that a class provide an interface to the external world and that it be accessible via this interface. Encapsulation supports this by establishing the concept of information hiding, where any logic within the class outside of what is exposed via the interface is not accessible to the external world. Service abstraction accomplishes much of the same as object abstraction and encapsulation. Its purpose is to hide the underlying details of the service so that only the service contract is available and of concern to service requestors.

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

6. Explain the basics of WS-BPEL Language (10)

16.1.1 The process element

- BPEL processes are exposed as WSDL services †
  - Message exchanges map to WSDL operations †
  - WSDL can be derived from partner definitions and the role played by the process in interaction with partners †
  - BPEL processes interact with WSDL services exposed by business partners

```
<process name="TimesheetSubmissionProcess"
    targetNamespace="http://www.xmltc.com/tls/process/"
    xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:bpl="http://www.xmltc.com/tls/process/"
      <partnerLinks>
            ...
      </partnerLinks>
      <variables>
            ...
      </variables>
      <sequence>
            ...
      </sequence>
      ...
</process>
```

Example 16-1 ***A skeleton process definition.***

16.1.2 The partnerLinks and partnerLink elements

A partnerLink element establishes the port type of the service (partner) that will be participating during the execution of the business process. Partner services can act as a client to the process, responsible for invoking the process service. Alternatively, partner services can be invoked by the process service itself. The contents of a partnerLink element represent the communication exchange between two partners – the process service being one partner and another service being the other.

```
<partnerLinks>
        <partnerLink name="client"
                partnerLinkType="tns:TimesheetSubmissionType"
                myRole="TimesheetSubmissionServiceProvider"/>
</partnerLinks>
```

Example 16-2 ***The partnerLinks construct containing one partnerLink element in which the process service is invoked by an external client partner, and four partnerLink elements that identify partner services invoked by the process service.***

16.1.3 The partnerLinkType element

For each partner service involved in a process, partnerLinkType elements identify the WSDL portType elements referenced by the partnerLink elements within the process definition. The partnerLinkType construct contains one role element for each role the service can play Therefore, a partnerLinkType will have either one or two child role elements.

```
<definitions name="Employee"
        targetNamespace="http://www.xmltc.com/tls/employee/wsdl/"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
        ...
>
        ...
        <plnk:partnerLinkType name="EmployeeServiceType"
                xmlns="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
                <plnk:role name="EmployeeServiceProvider">
                        <portType name="emp:EmployeeInterface"/>
                </plnk:role>
        </plnk:partnerLinkType>
  ...
</definitions>
```

Example 16-3 ***A WSDL definitions construct containing a partnerLinkType construct.***

Note that multiple partnerLink elements can reference the same partnerLinkType. This is useful for when a process service has the same relationship with multiple partner services. All of the partner services can therefore use the same process service portType elements.

16.1.4 The variables element

Variables are used to define data containers „

- WSDL messages received from or sent to partners „
- Messages that are persisted by the process „
- XML data defining the process state
- messageType, element, or type.
- The messageType attribute allows for the variable to contain an entire WSDL-defined message,
- Element attribute simply refers to an XSD element construct.
- The type attribute can be used to just represent an XSD simpleType, such as string or integer.

```
<variables>
    <variable name="ClientSubmission"
                messageType="bpl:receiveSubmitMessage"/>
</variables>
```

Example 16-4 ***The variables construct hosting only some of the child variable elements used later by the Timesheet Submission Process.***

16.1.5 The getVariableProperty and getVariableData functions

getVariableProperty(variable name, property name)

- accepts the variable and property names as input and returns the requested value.

getVariableData(variable name, part name, location path)

This function is required to provide other parts of the process logic access to this data.

The getVariableData function has a mandatory variable name parameter, and two optional arguments that can be used to specify a specific part of the variable data.

In our examples we use the getVariableData function a number of times to retrieve message data from variables.

getVariableData(‘InvoiceHoursResponse’,‘ResponseParameter’)

getVariableData(‘input’,’payload’,‘/tns:TimesheetType/Hours/...’)

Example 16-5 ***Two getVariableData functions being used to retrieve specific pieces of data from different variables.***

16.1.6 The sequence element

The sequence construct allows you to organize a series of activities so that they are executed in a predefined, sequential order. WS-BPEL provides numerous activities that can be used to express the workflow logic within the process definition.

```
<sequence>
    <receive>
                ...
    </receive>
    <assign>
                ...
    </assign>
    <invoke>
                ...
    </invoke>
    <reply>
                ...
    </reply>
</sequence>
```

Example 16-6 ***A skeleton sequence construct containing only some of the many activity elements provided by WS-BPEL.***

16.1.7 The invoke element

The invoke element is equipped with five common attributes which further specify the details of the invocation (Table 16.1).

Attribute	Description
partnerLink	This element names the partner service via its corresponding partnerLink.
portType	The element used to identify the portType element of the partner service.
operation	The partner service operation to which the process service will need to send its request.
inputVariable	The input message that will be used to communicate with the partner service operation. Note that it is referred to as a variable because it is referencing a WS-BPEL variable element with 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.

Table 16-1 ***invoke element attributes.***

```
<invoke name="ValidateWeeklyHours"
        partnerLink="Employee"
        portType="emp:EmployeeInterface"
        operation="GetWeeklyHoursLimit"
        inputVariable="EmployeeHoursRequest"
        outputVariable="EmployeeHoursResponse"/>
```

Example 16-7 ***The invoke element identifying the target partner service details.***

16.1.8 The receive element

The receive element allows us to establish the information a process service expects upon receiving a request from an external client partner service.

The receive element contains a set of attributes, each of which is assigned a value relating to the expected incoming communication (Table 16.2).

Attribute	Description
partnerLink	The client partner service identified in the corresponding partnerLink construct.
portType	The partner service's portType involved in the message transfer.
operation	The partner service's operation that will be issuing the request to the process service.
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.

Table 16-2 ***receive element attributes.***

Note that this element can also be used to receive callback messages during an asynchronous message exchange.

```
<receive name="receiveInput"
        partnerLink="client"
        portType="tns:TimesheetSubmissionInterface"
        operation="Submit"
```

```
        variable="ClientSubmission"
        createInstance="yes"/>
```

Example 16-8 ***The receive element used in the Timesheet Submission Process definition to indicate the client partner service responsible for launching the process with the submission of a timesheet document.***

16.1.9 The reply element

The reply element is responsible for establishing the details of returning a response message to the requesting client partner service.

Attribute	Description
partnerLink	The same partnerLink element established in the receive element.
portType	The same portType element displayed in the receive element.
operation	The same operation element from the receive element.
variable	The process service variable element that holds the message that is returned to the partner service.
messageExchange	It is being proposed that this optional attribute be added by the WS-BPEL 2.0 specification. It allows for the reply element to be explicitly associated with a message activity capable of receiving a message (such as the receive element).

Table 16-3 ***reply element attributes.***

```
<reply partnerLink="client"
        portType="TimeSubmissionProcessInterface"
    operation="SubmitTimesheet"
    variable="TimesheetSubmissionResponse"/>
```

Example 16-9 ***A potential companion reply element to the previously displayed receive element.***

The switch, case, and otherwise elements

The switch element establishes the scope of the conditional logic
multiple case constructs can be nested to check for various conditions using a condition attribute.
condition attribute resolves to “true,” the activities defined within the corresponding case construct are executed.
The otherwise element can be added as a catch all at the end of the switch construct.
Should all preceding case conditions fail, the activities within the otherwise construct are executed.

```
<switch>
        <case condition="getVariableData('EmployeeResponseMessage','ResponseParameter')=0">
                ...
        </case>
        <otherwise>
                ...
        </otherwise>
</switch>
```

Example 16-10 ***A skeleton case element wherein the condition attribute uses the getVariableData function to compare the content of the EmployeeResponseMessage variable to a zero value.***

Note: It has been proposed that the switch, case, and otherwise elements be replaced with if, elseif, and else elements in WS-BPEL 2.0.

16.1.10 The assign, copy, from, and to elements

This set of elements simply gives us the ability to copy values between process variables

```
<assign>
```

```
        <copy>
                <from variable="TimesheetSubmissionFailedMessage"/>
        <to variable="EmployeeNotificationMessage"/>
        </copy>
        <copy>
                <from variable="TimesheetSubmissionFailedMessage"/>
        <to variable="ManagerNotificationMessage"/>
        </copy>
</assign>
```

Example 16-11 ***Within this assign construct, the contents of the TimesheetSubmissionFailedMessage variable are copied to two different message variables.***

Note that the copy construct can process a variety of data transfer functions
from and to elements can contain optional part and query attributes that allow for specific parts or values of the variable to be referenced.

16.1.11 faultHandlers, catch, and catchAll elements

This construct can contain multiple catch elements, each of which provides activities that perform exception handling for a specific type of error condition.
Faults can be generated by the receipt of a WSDL-defined fault message, or they can be explicitly triggered through the use of the throw element.
The faultHandlers construct can consist of (or end with) a catchAll element to house default error handling activities.

```
<faultHandlers>
        <catch faultName="SomethingBadHappened"
                faultVariable="TimesheetFault">
                ...
        </catch>
        <catchAll>
                ...
        </catchAll>
</faultHandlers>
```

Example 16-12 ***The faultHandlers construct hosting catch and catchAll child constructs.***

7. Explain how to design service oriented business process (10)

Step 1: Map out interaction scenarios.
Step 2: Design the process service interface.
Step 3: Formalize partner service conversations.
Step 4: Define process logic.
Step 5: Align interaction scenarios and refine process. (Optional)

Step 1: Map out interaction scenarios.

By using the following information gathered so far, we can define the message exchange requirements of our process service:

- Available workflow logic produced during the service modeling process in Chapter 12.
- The process service candidate created in Chapter 12.
- The existing service designs created in Chapter 15.

This information is now used to form the basis of an analysis during which all possible interaction scenarios between process and partner services are mapped out. The result is a series of processing requirements that will form the basis of the process service design we proceed to in Step 2.

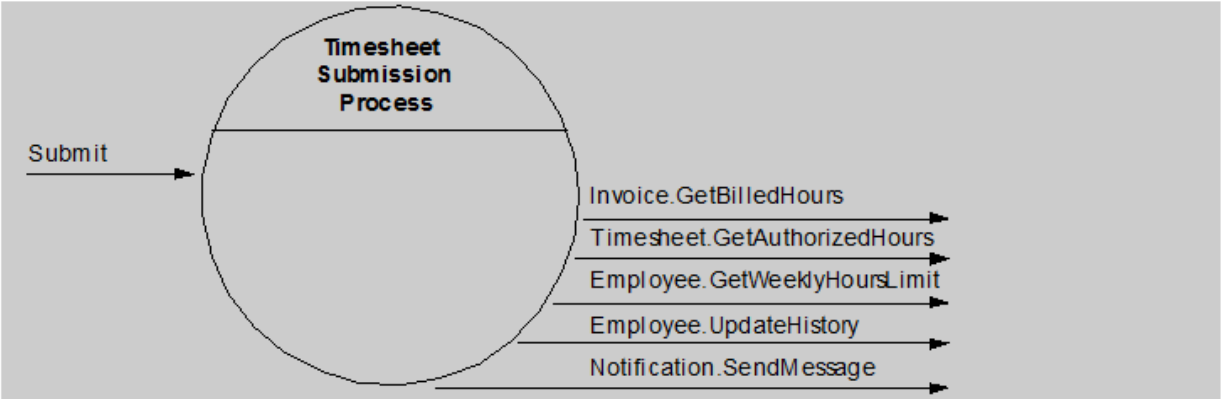


Figure 16-10 ***The incoming and outgoing request messages expected to be processed by the Timesheet Submission Process Service.***

16.1.12 Step 2: Design the process service interface.

- Document the input and output values required for the processing of each operation, and populate the `types` section with XSD schema types required to process the operations. Move the XSD schema information to a separate file, if required.
- Build the WSDL definition by creating the `portType` (or `interface`) area, inserting the identified `operation` constructs. Then, add the necessary `message` constructs containing the `part` elements which reference the appropriate schema types. Add naming conventions that are in alignment with those used by your other WSDL definitions.
- Add meta information via the `documentation` element.
- Apply other design standards within the confines of the modeling tool.

Below is the corresponding WSDL definition.

```
<definitions name="TimesheetSubmission"
      targetNamespace="http://www.xmltc.com/tls/process/wsdl/"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:ts="http://www.xmltc.com/tls/timesheet/schema/"
      xmlns:tsd="http://www.xmltc.com/tls/timesheetservice/schema/"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:tns="http://www.xmltc.com/tls/timesheet/wsdl/"
      xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
<types>
      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.xmltc.com/tls/timesheetsubmissionservice/schema/">
            <xsd:import
namespace="http://www.xmltc.com/tls/timesheet/schema/"
schemaLocation="Timesheet.xsd"/>
            <xsd:element name="Submit">
```

```
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element name="ContextID"
type="xsd:integer"/>
                        <xsd:element name="TimesheetDocument"
type="ts:TimesheetType"/>
                  </xsd:sequence>
            </xsd:complexType>
            </xsd:element>
      </xsd:schema>
</types>
<message name="receiveSubmitMessage">
      <part name="Payload" element="tsd:TimesheetType"/>
</message>
<portType name="TimesheetSubmissionInterface">
      <documentation>
            Initiates the Timesheet Submission process. </documentation>
      <operation name="Submit">
            <input message="tns:receiveSubmitMessage"/>
      </operation>
</portType>
<plnk:partnerLinkType name="TimesheetSubmissionType">
      <plnk:role name="TimesheetSubmissionService">
            <plnk:portType name="tns:TimesheetSubmissionInterface"/>
      </plnk:role>
</plnk:partnerLinkType>
</definitions>
```

Example 16-13 ***The abstract service definition for the Timesheet Submission Process Service.***

Note the bolded `plnk:parnterLinkType` construct at the end of this WSDL definition. This is added to every partner service.

16.1.13 Step 3: Formalize partner service conversations.

We now begin our WS-BPEL process definition by establishing details about the services with which our process service will be interacting.

The following steps are suggested:

1. Define the partner services that will be participating in the process and assign each the role it will be playing within a given message exchange.
2. Add `parterLinkType` constructs to the end of the WSDL definitions of each partner service.
3. Create `partnerLink` elements for each partner service within the process definition.
4. Define `variable` elements to represent incoming and outgoing messages exchanged with partner services.

This information essentially documents the possible conversation flows that can occur within the course of the process execution. Depending on the process modeling tool used, completing these steps may simply require interaction with the user-interface provided by the modeling tool.

16.1.14 Step 4: Define process logic.

Finally, everything is in place for us to complete the process definition. This step is a process in itself, as it requires that all existing workflow intelligence be transposed and implemented via a WS-BPEL process definition.

16.1.15 Step 5: Align interaction scenarios and refine process. (Optional)

This final, optional step encourages you to perform two specific tasks: revisit the original interaction scenarios created in Step 1 and review the WS-BPEL process definition to look for optimization opportunities.
Let's start with the first task. Bringing the interaction scenarios in alignment with the process logic expressed in the WS-BPEL process definition provides a number of benefits, including:

- The service interaction maps (as activity diagrams or in whatever format you created them) are an important part of the solution documentation, and will be useful for future maintenance and knowledge transfer requirements.
- The service interaction maps make for great test cases, and can spare testers from having to perform speculative analysis.
- The implementation of the original workflow logic as a series of WS-BPEL activities may have introduced new or augmented process logic. Once compared to the existing interaction scenarios, the need for additional service interactions may arise, leading to the discovery of new fault or exception conditions that can then be addressed back in the WS-BPEL process definition.

Secondly, spending some extra time to review your WS-BPEL process definition is well worth the effort. WS-BPEL is a multi-feature language that provides different approaches for accomplishing and structuring the same overall activities. By refining your process definition, you may be able to:

- Consolidate or restructure activities to achieve performance improvements.
- Streamline the markup code to make maintenance easier.
- Discover features that were previously not considered.

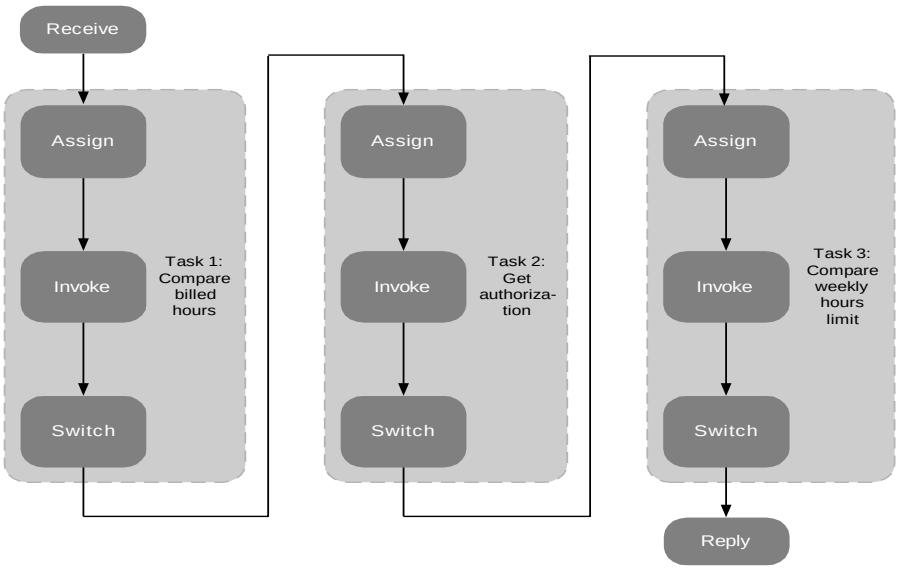


Figure 16-1 ***Sequential, synchronous execution of process activities.***

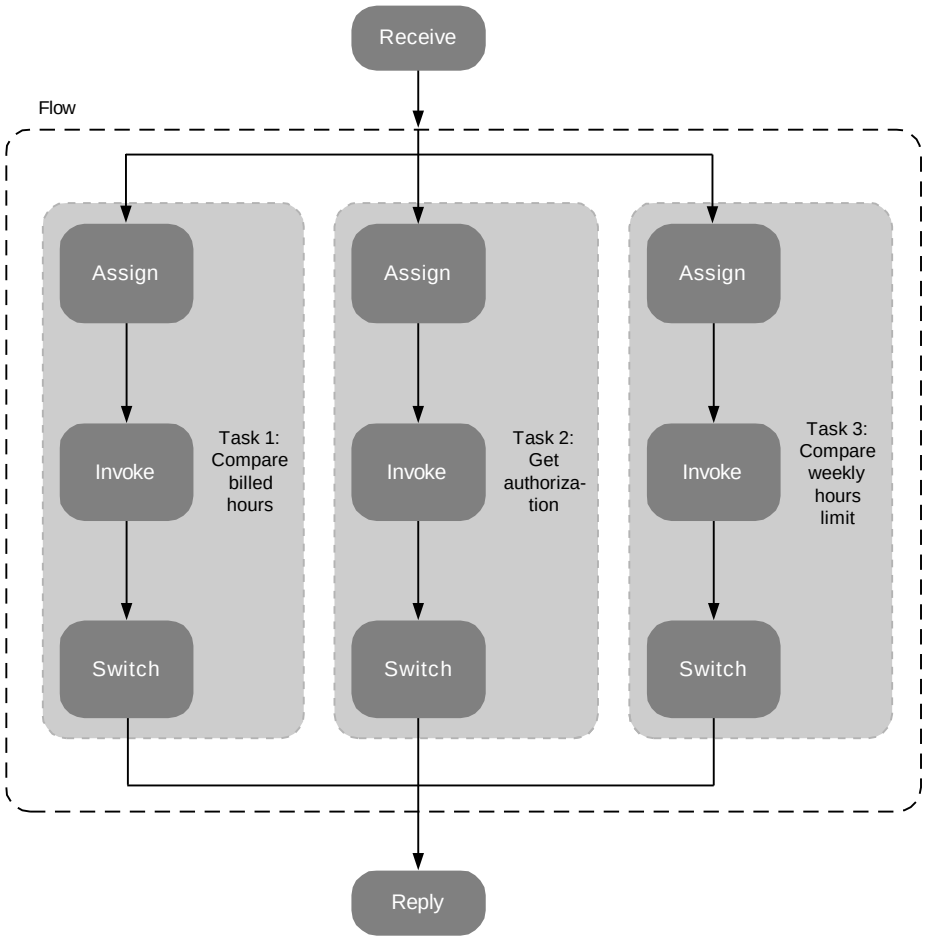


Figure 16-2 ***Concurrent execution of process activities using the*** `Flow` ***construct.***