

Sample Chapter 16 from "Service-Oriented Architecture: Concepts, Technology, and Design" by Thomas Erl
For more information visit: www.soabooks.com

Service-Oriented Architecture

Concepts, Technology, and Design

Thomas Erl



PRENTICE HALL PROFESSIONAL TECHNICAL REFERENCE
UPPER SADDLE RIVER, NJ • BOSTON • INDIANAPOLIS • SAN FRANCISCO
NEW YORK • TORONTO • MONTREAL • LONDON • MUNICH • PARIS • MADRID
CAPETOWN • SYDNEY • TOKYO • SINGAPORE • MEXICO CITY

Sample Chapter 16 from "Service-Oriented Architecture: Concepts, Technology, and Design" by Thomas Erl
For more information visit: www.soabooks.com

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the U. S., please contact:

International Sales
international@pearsoned.com

Visit us on the Web: www.phptr.com

Library of Congress Number: 2005925019

Copyright © 2005 Pearson Education, Inc. Portions of this work are copyright SOA Systems Inc., and reprinted with permission from SOA Systems Inc. © 2005. Front cover and all photographs by Thomas Erl. Permission to use photographs granted by SOA Systems Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the copyright holder prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
One Lake Street
Upper Saddle River, NJ 07458

ISBN 0-13-185858-0


Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

First printing, July 2005



Chapter 16

Service-Oriented Design (Part IV: Business Process Design)

- 
- 16.1 WS-BPEL language basics
 - 16.2 WS-Coordination overview
 - 16.3 Service-oriented business process design
(a step-by-step process)

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.

In this chapter we tackle the design of an orchestration layer by using the WS-BPEL language to create a business process definition.

How case studies are used: Our focus in this chapter is the TLS environment. We provide case study examples throughout the step-by-step process description during which TLS builds a WS-BPEL process definition for the Timesheet Submission Process. This is the same process for which service candidates were modeled in Chapter 12 and for which the Employee Service interface was designed in Chapter 15.

16.1 WS-BPEL language basics

Before we can design an orchestration layer, we need to acquire a good understanding of how the operational characteristics of the process can be formally expressed. This book uses the WS-BPEL language to demonstrate how process logic can be described as part of a concrete definition (Figure 16.1) that can be implemented and executed via a compliant orchestration engine.

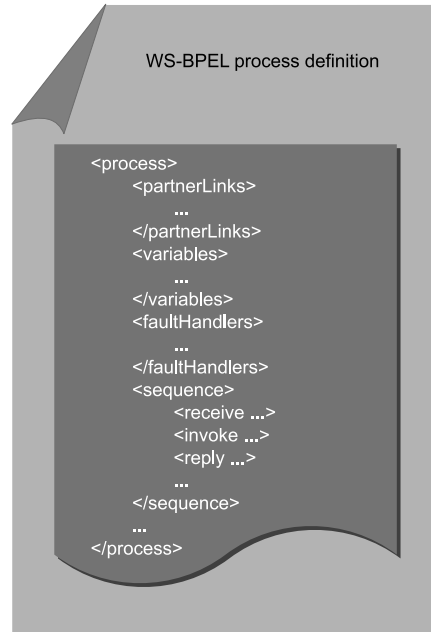


Figure 16.1

A common WS-BPEL process definition structure.

Although you likely will be using a process modeling tool and will therefore not be required to author your process definition from scratch, a knowledge of WS-BPEL elements still is useful and often required. WS-BPEL modeling tools frequently make reference to these elements and constructs, and you may be required to dig into the source code they produce to make further refinements.

NOTE

If you are already comfortable with the WS-BPEL language, feel free to skip ahead to the *Service-oriented business process design (a step-by-step process)* section.

16.1.1 A brief history of BPEL4WS and WS-BPEL

Before we get into the details of the WS-BPEL language, let's briefly discuss how this specification came to be. The Business Process Execution Language for Web Services (BPEL4WS) was first conceived in July, 2002, with the release of the BPEL4WS 1.0 specification, a joint effort by IBM, Microsoft, and BEA. This document proposed an

orchestration language inspired by previous variations, such as IBM's Web Services Flow Language (WSFL) and Microsoft's XLANG specification.

Joined by other contributors from SAP and Siebel Systems, version 1.1 of the BPEL4WS specification was released less than a year later, in May of 2003. This version received more attention and vendor support, leading to a number of commercially available BPEL4WS-compliant orchestration engines. Just prior to this release, the BPEL4WS specification was submitted to an OASIS technical committee so that the specification could be developed into an official, open standard.

The technical committee is in the process of finalizing the release of the next version of BPEL4WS. It has been announced that the language itself has been renamed to the Web Services Business Process Execution Language, or WS-BPEL (and assigned the 2.0 version number). The changes planned for WS-BPEL have been made publicly available on the OASIS Web site at www.oasis-open.org.

Notes have been added to the element descriptions in this section where appropriate to indicate changes in syntax between BPEL4WS and WS-BPEL. For simplicity's sake, we refer to the Business Process Execution Language as WS-BPEL in this book.

16.1.2 Prerequisites

It's time now to learn about the WS-BPEL language. If you haven't already done so, it is recommended that you read Chapter 6 prior to proceeding with this section. Concepts relating to orchestration, coordination, atomic transactions, and business activities are covered in Chapter 6, and are therefore not repeated here. This chapter also assumes you have read through the WSDL tutorial provided in Chapter 13.

16.1.3 The process element

Let's begin with the root element of a WS-BPEL process definition. It is assigned a name value using the `name` attribute and is used to establish the process definition-related namespaces.

```
<process name="TimesheetSubmissionProcess"
  targetNamespace="http://www.xmltc.com/tls/process/"
  xmlns=
    "http://schemas.xmlsoap.org/ws/2003/03/
      business-process/"
  xmlns:bpl="http://www.xmltc.com/tls/process/"
  xmlns:emp="http://www.xmltc.com/tls/employee/"
  xmlns:inv="http://www.xmltc.com/tls/invoice/"
```

```
xmlns:tst="http://www.xmltc.com/tls/timesheet/"
xmlns:not="http://www.xmltc.com/tls/notification/">
<partnerLinks>
    ...
</partnerLinks>
<variables>
    ...
</variables>
<sequence>
    ...
</sequence>
    ...
</process>
```

Example 16.1 A skeleton `process` definition.

The `process` construct contains a series of common child elements explained in the following sections.

16.1.4 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. Depending on the nature of the communication, the role of the process service will vary. For instance, a process service that is invoked by an external service may act in the role of “TimesheetSubmissionProcess.” However, when this same process service invokes a different service to have an invoice verified, it acts within a different role, perhaps “InvoiceClient.” The `partnerLink` element therefore contains the `myRole` and `partnerRole` attributes that establish the service provider role of the process service and the partner service respectively.

Put simply, the `myRole` attribute is used when the process service is invoked by a partner client service, because in this situation the process service acts as the service provider. The `partnerRole` attribute identifies the partner service that the process service will be invoking (making the partner service the service provider).

Note that both `myRole` and `partnerRole` attributes can be used by the same `partnerLink` element when it is expected that the process service will act as both service requestor and service provider with the same partner service. For example, during asynchronous communication between the process and partner services, the `myRole` setting indicates the process service's role during the callback of the partner service.

```
<partnerLinks>
  <partnerLink name="client "
    partnerLinkType="tns:TimesheetSubmissionType"
    myRole="TimesheetSubmissionServiceProvider"/>
  <partnerLink name="Invoice"
    partnerLinkType="inv:InvoiceType"
    partnerRole="InvoiceServiceProvider"/>
  <partnerLink name="Timesheet "
    partnerLinkType="tst:TimesheetType"
    partnerRole="TimesheetServiceProvider"/>
  <partnerLink name="Employee "
    partnerLinkType="emp:EmployeeType"
    partnerRole="EmployeeServiceProvider"/>
  <partnerLink name="Notification"
    partnerLinkType="not:NotificationType"
    partnerRole="NotificationServiceProvider"/>
</partnerLinks>
```

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.

You'll notice that in Example 16.2, each of the `partnerLink` elements also contains a `partnerLinkType` attribute. This refers to the `partnerLinkType` construct, as explained next.

16.1.5 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. Therefore, these constructs typically are embedded directly within the WSDL documents of every partner service (including the process service).

The `partnerLinkType` construct contains one role element for each role the service can play, as defined by the `partnerLink` `myRole` and `partnerRole` attributes. As a result, a `partnerLinkType` will have either one or two child role elements.


```
<definitions name="Employee"
  targetNamespace="http://www.xmltc.com/tls/employee/wsd1/"
  xmlns="http://schemas.xmlsoap.org/wsd1/"
  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.

NOTE

In version 2.0 of the WS-BPEL specification, it is being proposed that the portType element be changed so that it exists as an attribute of the role element.

16.1.6 The variables element

WS-BPEL process services commonly use the variables construct to store state information related to the immediate workflow logic. Entire messages and data sets formatted as XSD schema types can be placed into a variable and retrieved later during the course of the process. The type of data that can be assigned to a variable element needs to be predefined using one of the following three attributes: messageType, element, or type.

The messageType attribute allows for the variable to contain an entire WSDL-defined message, whereas the element attribute simply refers to an XSD element construct. The type attribute can be used to just represent an XSD simpleType, such as string or integer.

```
<variables>
  <variable name="ClientSubmission"
    messageType="bpl:receiveSubmitMessage"/>
  <variable name="EmployeeHoursRequest"
    messageType="emp:getWeeklyHoursRequestMessage"/>
  <variable name="EmployeeHoursResponse"
    messageType="emp:getWeeklyHoursResponseMessage"/>
  <variable name="EmployeeHistoryRequest"
    messageType="emp:updateHistoryRequestMessage"/>
  <variable name="EmployeeHistoryResponse"
    messageType="emp:updateHistoryResponseMessage"/>
  ...
</variables>
```

Example 16.4 The `variables` construct hosting only some of the child `variable` elements used later by the Timesheet Submission Process.

Typically, a variable with the `messageType` attribute is defined for each input and output message processed by the process definition. The value of this attribute is the message name from the partner process definition.

16.1.7 The `getVariableProperty` and `getVariableData` functions

WS-BPEL provides built-in functions that allow information stored in or associated with variables to be processed during the execution of a business process.

getVariableProperty(variable name, property name)

This function allows global property values to be retrieved from variables. It simply accepts the variable and property names as input and returns the requested value.

getVariableData(variable name, part name, location path)

Because variables commonly are used to manage state information, 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 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.

16.1.8 The sequence element

The `sequence` construct allows you to organize a series of activities so that they are executed in a predefined, sequential order. WS-BPEL provides numerous activities that can be used to express the workflow logic within the process definition. The remaining element descriptions in this section explain the fundamental set of activities used as part of our upcoming case study examples.

```
<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.

Note that `sequence` elements can be nested, allowing you to define sequences within sequences.

16.1.9 The `invoke` element

This element identifies the operation of a partner service that the process definition intends to invoke during the course of its execution. The `invoke` element is equipped with five common attributes, which further specify the details of the invocation (Table 16.1).

Table 16.1 `invoke` element attributes

Attribute	Description
<code>partnerLink</code>	This element names the partner service via its corresponding <code>partnerLink</code> .
<code>portType</code>	The element used to identify the <code>portType</code> element of the partner service.
<code>operation</code>	The partner service operation to which the process service will need to send its request.
<code>inputVariable</code>	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 <code>variable</code> element with a <code>messageType</code> attribute.
<code>outputVariable</code>	This element is used when communication is based on the request-response MEP. The return value is stored in a separate <code>variable</code> element.

```
<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.

16.1.10 The receive element

The `receive` element allows us to establish the information a process service expects upon receiving a request from an external client partner service. In this case, the process service is viewed as a service provider waiting to be invoked.

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

Table 16.2 `receive` element attributes

Attribute	Description
<code>partnerLink</code>	The client partner service identified in the corresponding <code>partnerLink</code> construct.
<code>portType</code>	The process service <code>portType</code> that will be waiting to receive the request message from the partner service.
<code>operation</code>	The process service operation that will be receiving the request.
<code>variable</code>	The process definition <code>variable</code> construct in which the incoming request message will be stored.
<code>createInstance</code>	When this attribute is set to "yes," the receipt of this particular request may be responsible for creating a new instance of the process.

Note that this element also can 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.

16.1.11 The `reply` element

Where there's a `receive` element, there's a `reply` element when a synchronous exchange is being mapped out. The `reply` element is responsible for establishing the details of returning a response message to the requesting client partner service. Because this element is associated with the same `partnerLink` element as its corresponding `receive` element, it repeats a number of the same attributes (Table 16.3).

Table 16.3 `reply` element attributes

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

```
<reply partnerLink="client"
  portType="tns:TimesheetSubmissionInterface"
  operation="Submit"
  variable="TimesheetSubmissionResponse"/>
```

Example 16.9 A potential companion `reply` element to the previously displayed `receive` element.

16.1.12 The switch, case, and otherwise elements

These three structured activity elements allow us to add conditional logic to our process definition, similar to the familiar select case/case else constructs used in traditional programming languages. The `switch` element establishes the scope of the conditional logic, wherein multiple `case` constructs can be nested to check for various conditions using a `condition` attribute. When a `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.

16.1.13 The assign, copy, from, and to elements

This set of elements simply gives us the ability to copy values between process variables, which allows us to pass around data throughout a process as information is received and modified during the process execution.

```
<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 (for example, only a part of a message can be extracted and copied into a variable). `from` and `to` elements also can contain optional `part` and `query` attributes that allow for specific parts or values of the variable to be referenced.

16.1.14 `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.

16.1.15 Other WS-BPEL elements

The following table provides brief descriptions of other relevant parts of the WS-BPEL language.

Table 16.4 Quick reference table providing short descriptions for additional WS-BPEL elements (listed in alphabetical order).

Element	Description
<code>compensationHandler</code>	A WS-BPEL process definition can define a compensation process that kicks in a series of activities when certain conditions occur to justify a compensation. These activities are kept in the <code>compensationHandler</code> construct. (For more information about compensations, see the <i>Business activities</i> section in Chapter 6.)
<code>correlationSets</code>	WS-BPEL uses this element to implement correlation, primarily to associate messages with process instances. A message can belong to multiple <code>correlationSets</code> . Further, message properties can be defined within WSDL documents.
<code>empty</code>	This simple element allows you to state that no activity should occur for a particular condition.
<code>eventHandlers</code>	The <code>eventHandlers</code> element enables a process to respond to events during the execution of process logic. This construct can contain <code>onMessage</code> and <code>onAlarm</code> child elements that trigger process activity upon the arrival of specific types of messages (after a predefined period of time, or at a specific date and time, respectively).
<code>exit</code>	See the <code>terminate</code> element description that follows.
<code>flow</code>	A <code>flow</code> construct allows you to define a series of activities that can occur concurrently and are required to complete after all have finished executing. Dependencies between activities within a <code>flow</code> construct are defined using the child <code>link</code> element.
<code>pick</code>	Similar to the <code>eventHandlers</code> element, this construct also can contain child <code>onMessage</code> and <code>onAlarm</code> elements but is used more to respond to external events for which process execution is suspended.

Table 16.4 Quick reference table providing short descriptions for additional WS-BPEL elements (listed in alphabetical order) (*Continued*).

Element	Description
scope	Portions of logic within a process definition can be sub-divided into scopes using this construct. This allows you to define <code>variables</code> , <code>faultHandlers</code> , <code>correlationSets</code> , <code>compensationHandler</code> , and <code>eventHandlers</code> elements local to the scope.
terminate	This element effectively destroys the process instance. The WS-BPEL 2.0 specification proposes that this element be renamed <code>exit</code> .
throw	WS-BPEL supports numerous fault conditions. Using the <code>throw</code> element allows you to explicitly trigger a fault state in response to a specific condition.
wait	The <code>wait</code> element can be set to introduce an intentional delay within the process. Its value can be a set time or a predefined date.
while	This useful element allows you to define a loop. As with the <code>case</code> element, it contains a <code>condition</code> attribute that, as long as it continues resolving to "true," will continue to execute the activities within the <code>while</code> construct.

SUMMARY OF KEY POINTS

- 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.

16.2 WS-Coordination overview

NOTE

Only element descriptions are provided in this section. Concepts relating to these elements are covered in Chapter 6. If you are not interested in learning about WS-Coordination syntax at this point, then feel free to skip ahead to the *Service-oriented business process design* process description. This section is not a prerequisite to continue with the remainder of the book.

Provided in this section is a brief look at WS-Coordination, which can be used to realize some of the underlying mechanics for WS-BPEL orchestrations. Specifically, we describe some of the elements from the WS-Coordination specification and look at how they are used to implement the supplementary specifications that provide coordination protocols (WS-BusinessActivity and WS-AtomicTransaction).

Note that a syntactical knowledge of these languages is generally not necessary to create WS-BPEL process definitions. We discuss these languages at this stage only to provide an insight as to how WS-Coordination can be positioned within a WS-BPEL orchestration model, and to get a glimpse at some of the syntax behind the specifications we first introduced only on a conceptual level in Chapter 6.

When we explained WS-Coordination earlier, we described the overall coordination mechanism that consists of the activation service, the registration service, a coordinator, and participants that implement specific protocols. It is likely that these underlying context management services will be automatically governed by the orchestration engine platform for which you are creating a WS-BPEL process definition.

In terms of the WS-Coordination language and its two protocol documents, what may be of interest to you is the actual `CoordinationContext` header that is inserted into SOAP messages. You may encounter this header if you are monitoring messages or if you need to perform custom development associated with the coordination context.

Also while this section briefly discusses the WS-Coordination specification within the context of the orchestration service layer, it is important to note that this specification is a standalone SOA extension in its own right. Its use is in no way dependent on WS-BPEL or an orchestration service layer.

16.2.1 The CoordinationContext element

This parent construct contains a series of child elements that each house a specific part of the context information being relayed by the header.

```
<Envelope
  xmlns="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsc=
    "http://schemas.xmlsoap.org/ws/2002/08/wscoor"
  xmlns:wsu=
    "http://schemas.xmlsoap.org/ws/2002/07/utility">
  <Header>
    <wsc:CoordinationContext>
      <wsu:Identifier>
        ...
      </wsu:Identifier>
      <wsu:Expires>
        ...
      </wsu:Expires>
      <wsc:CoordinationType>
        ...
      </wsc:CoordinationType>
      <wsc:RegistrationService>
        ...
      </wsc:RegistrationService>
    </wsc:CoordinationContext>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

Example 16.13 A skeleton CoordinationContext construct.

The activation service returns this CoordinationContext header upon the creation of a new activity. As described later, it is within the CoordinationType child construct that the activity protocol (WS-BusinessActivity, WS-AtomicTransaction) is carried. Vendor-specific implementations of WS-Coordination can insert additional elements within the CoordinationContext construct that represent values related to the execution environment.

16.2.2 The Identifier and Expires elements

These two elements originate from a utility schema used to provide reusable elements. WS-Coordination uses the `Identifier` element to associate a unique ID value with the current activity. The `Expires` element sets an expiry date that establishes the extent of the activity's possible lifespan.

```
<Envelope
  ...
  xmlns:wsu=
    "http://schemas.xmlsoap.org/ws/2002/07/utility">
  ...
  <wsu:Identifier>
    http://www.xmltc.com/ids/process/33342
  </wsu:Identifier>
  <wsu:Expires>
    2008-07-30T24:00:00.000
  </wsu:Expires>
  ...
</Envelope>
```

Example 16.14 Identifier and Expires elements containing values relating to the header.

16.2.3 The CoordinationType element

This element is described shortly in the *WS-BusinessActivity and WS-AtomicTransaction coordination types* section.

16.2.4 The RegistrationService element

The `RegistrationService` construct simply hosts the endpoint address of the registration service. It uses the `Address` element also provided by the utility schema.

```
<wsc:RegistrationService>
  <wsu:Address>
    http://www.xmltc.com/bpel/reg
  </wsu:Address>
</wsc:RegistrationService>
```

Example 16.15 The `RegistrationService` element containing a URL pointing to the location of the registration service.

16.2.5 Designating the WS-BusinessActivity coordination type

The specific protocol(s) that establishes the rules and constraints of the activity are identified within the `CoordinationType` element. The URI values that are placed here are predefined within the WS-BusinessActivity and WS-AtomicTransaction specifications.

This first example shows the `CoordinationType` element containing the WS-Business-Activity coordination type identifier. This would indicate that the activity for which the header is carrying context information is a potentially long-running activity.

```
<wsc:CoordinationType>  
  http://schemas.xmlsoap.org/ws/2004/01/wsba  
</wsc:CoordinationType>
```

Example 16.16 The `CoordinationType` element representing the WS-BusinessActivity protocol.

16.2.6 Designating the WS-AtomicTransaction coordination type

In the next example, the `CoordinationType` element is assigned the WS-AtomicTransaction coordination type identifier, which communicates the fact that the header's context information is part of a short running transaction.

```
<wsc:CoordinationType>  
  http://schemas.xmlsoap.org/ws/2003/09/wsat  
</wsc:CoordinationType>
```

Example 16.17 The `CoordinationType` element representing the WS-AtomicTransaction protocol.

SUMMARY OF KEY POINTS

- WS-Coordination provides a sophisticated context management system that may be leveraged by WS-BPEL.
 - WS-BusinessActivity and WS-AtomicTransaction define specific protocols for use with WS-Coordination.
-

16.3 Service-oriented business process design (a step-by-step process)

Designing the process of a service-oriented solution really just comes down to properly interpreting the business process requirements you have collected and then implementing them accurately. The trick, though, is to also account for all possible variations of process activity. This means understanding not just *what* can go wrong, but *how* the process will respond to unexpected or abnormal conditions.

Historically, business processes were designed by analysts using modeling tools that produced diagrams handed over to architects and developers for implementation. The workflow diagram and its accompanying documentation were the sole means of communicating how this logic should be realized within an automated solution. While diligent analysis and documentation, coupled with open minded and business-aware technical expertise, can lead to a successful collaboration of business and technical team members, this approach does leave significant room for error.

This gap is being addressed by operational business modeling languages, such as WS-BPEL. Modeling tools exist, allowing technical analysts and architects to graphically create business process diagrams that represent their workflow logic requirements, all the while auto-generating WS-BPEL syntax in the background.

These tools typically require that the user possess significant knowledge of the WS-BPEL language. However, more sophisticated tools, geared directly at business analysts, already are emerging, removing the prerequisite of having to understand WS-BPEL to create WS-BPEL process definitions.

The result is a diagram on the front-end that expresses the analysts' vision of the process and a computer executable process definition on the back-end that can be handed over to architects and developers for immediate (and not-open-to-interpretation) implementation (Figure 16.2).

When operational, the WS-BPEL process is appropriately represented and expressed through a process service within the service interface layer. This process service effectively establishes the orchestration service sub-layer, responsible for governing and composing business and application layers.

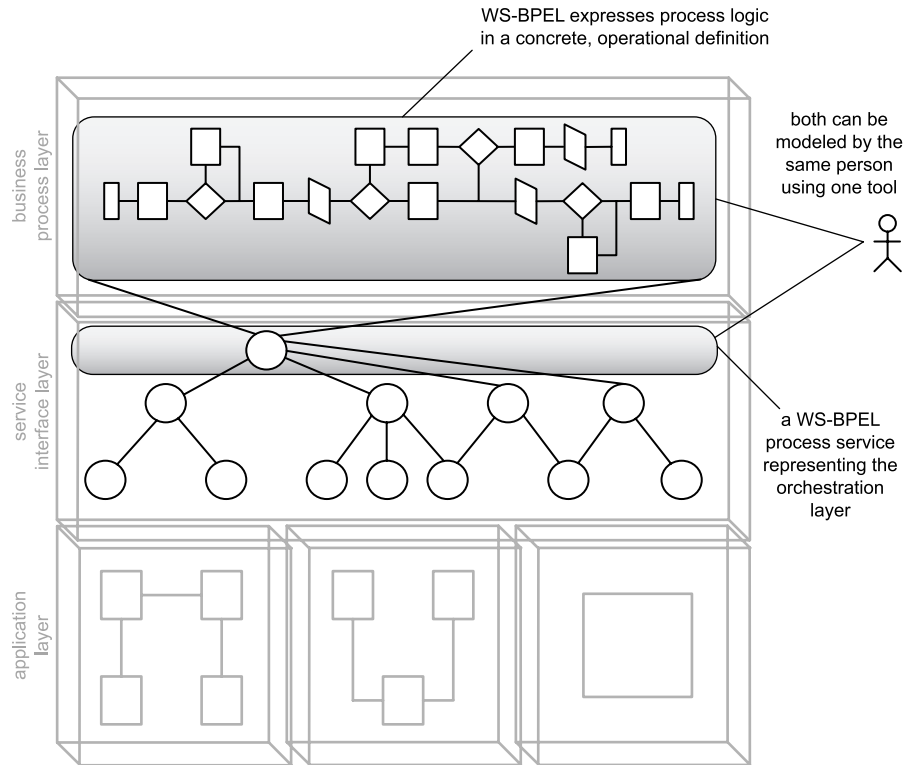


Figure 16.2

A concrete definition of a process service designed using a process modeling tool.

16.3.1 Process description

The following step-by-step design process (Figure 16.3) provides some high-level guidance for how to approach the creation of a WS-BPEL process definition. The steps are similar to those used by the *Task-centric business service design* process described in Chapter 15, except for one important detail.

When we designed a task-centric service, we simply produced a service interface capable of handling anticipated message exchanges. The details of the workflow logic were deferred to the design and development of the underlying application logic. When designing a WS-BPEL process, this workflow logic is abstracted into a separate process definition. Therefore, the design of workflow details is addressed at this stage, along with the definition of the process service interface.

Service-oriented business process design (a step-by-step process)

587

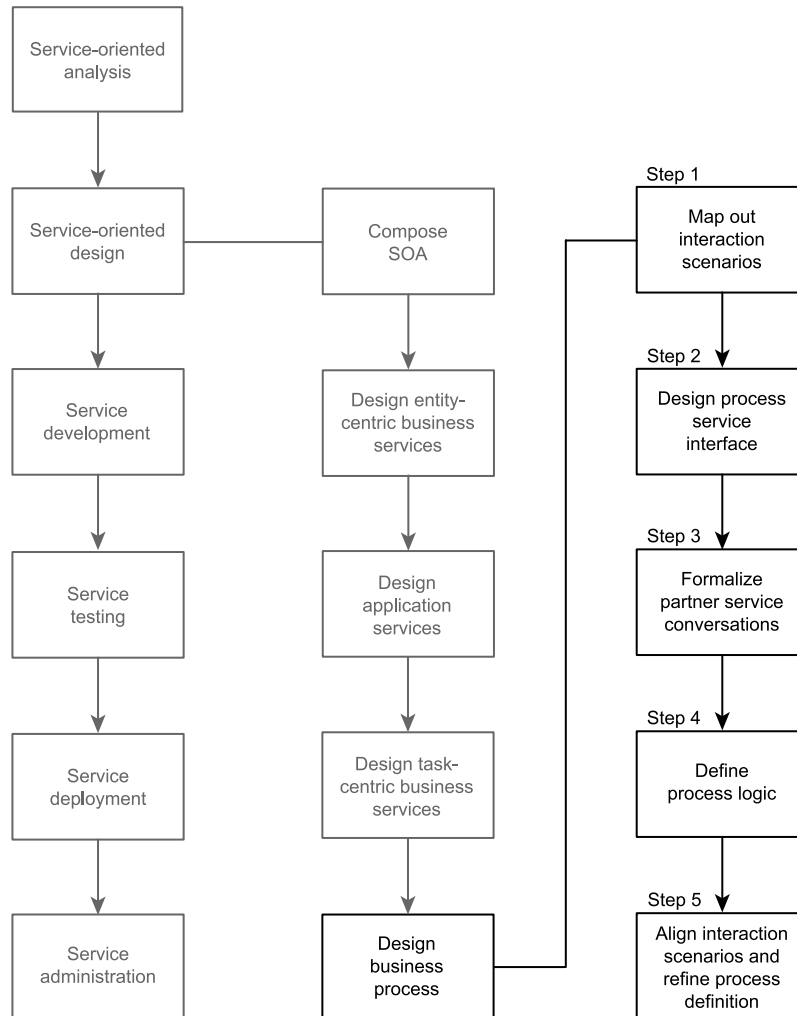


Figure 16.3

A high-level process for designing business processes.

The examples used to demonstrate each step are intentionally simple so that the basic WS-BPEL element descriptions we just covered in the previous section can be easily understood. When designing more complex workflow logic, a more detailed and elaborate design process is required.

Business process design is the last step in our overall service-oriented design process. This means that, for the most part, the application and business services required to carry out the process logic will have already been modeled and designed as we begin.

CASE STUDY

The original workflow logic for the TLS Timesheet Submission Process (Figure 16.4) that was created during the modeling exercise in Chapter 12 is revisited as TLS analysts and architects embark on designing a corresponding WS-BPEL process definition.

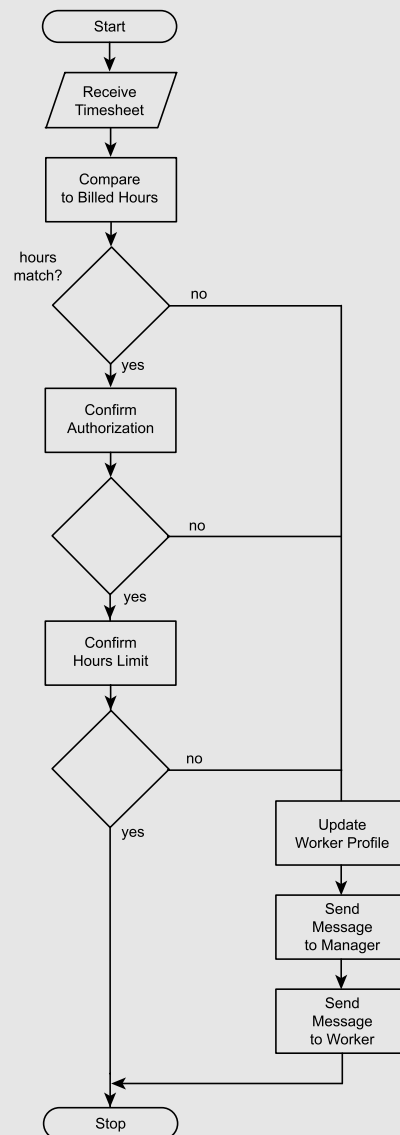


Figure 16.4

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. In our previous case study examples, we only stepped through the creation of the Employee Service. The other service designs are provided here to help demonstrate the WS-BPEL partner links we define later on.

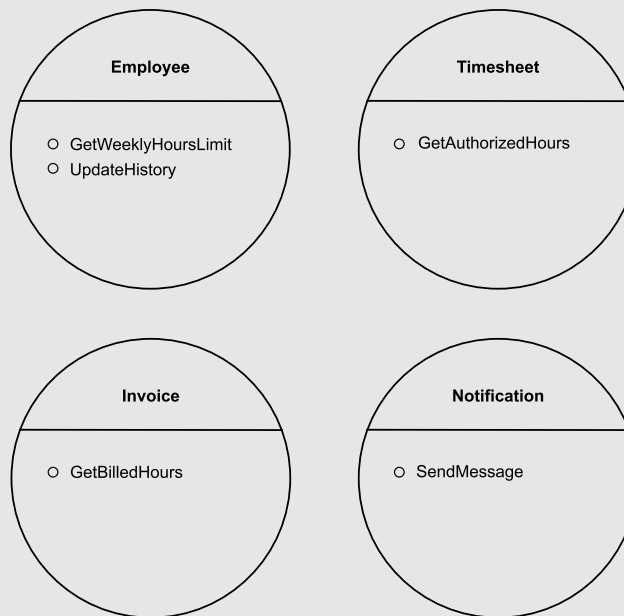


Figure 16.5
Service designs created so far.

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.

Finally, TLS architects revive the original service candidate created for the Timesheet Submission Process Service (Figure 16.7).

With all of this information in hand, TLS proceeds with the business process design.

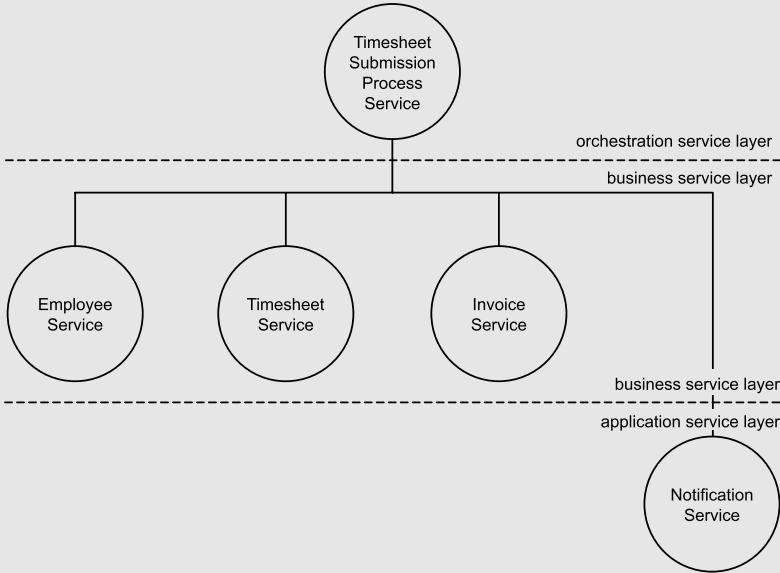


Figure 16.6
The original service composition defined during the service modeling stage.

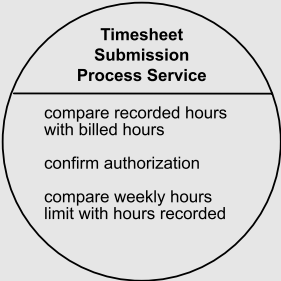


Figure 16.7
The Timesheet Submission Process Service candidate.

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 now is 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.

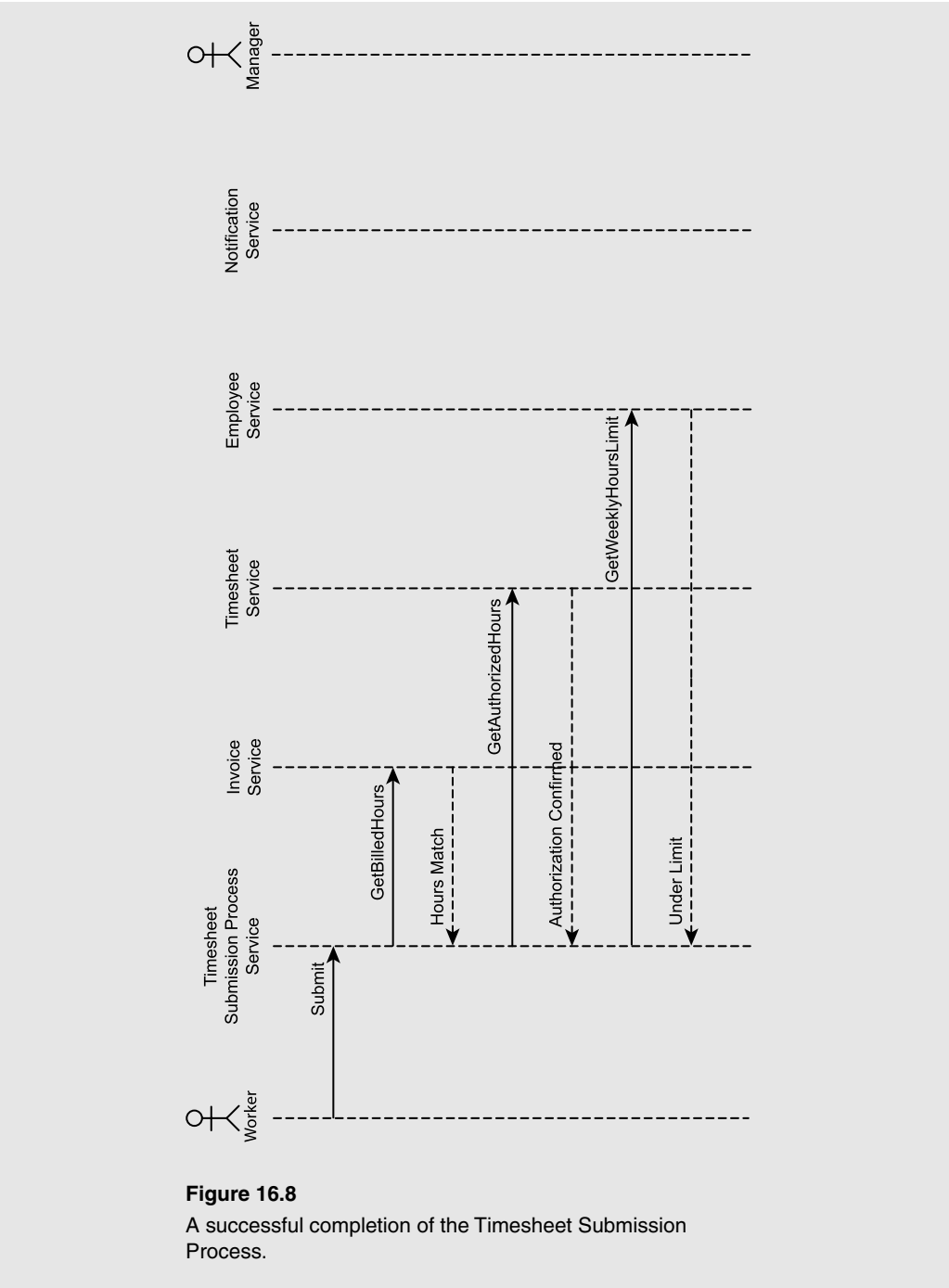
CASE STUDY

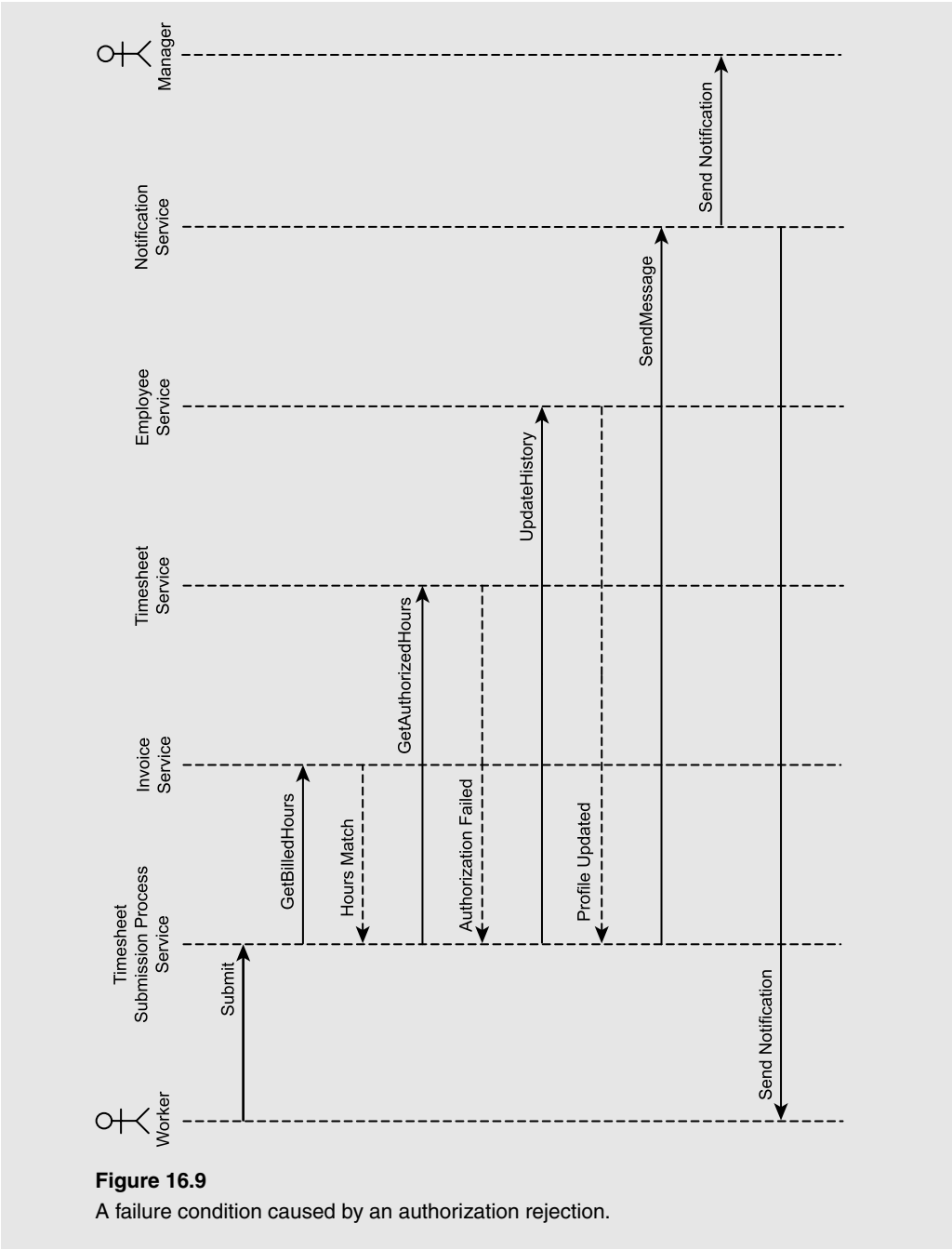
TLS maps out a series of different service interaction scenarios using activity diagrams. Following are examples of two scenarios.

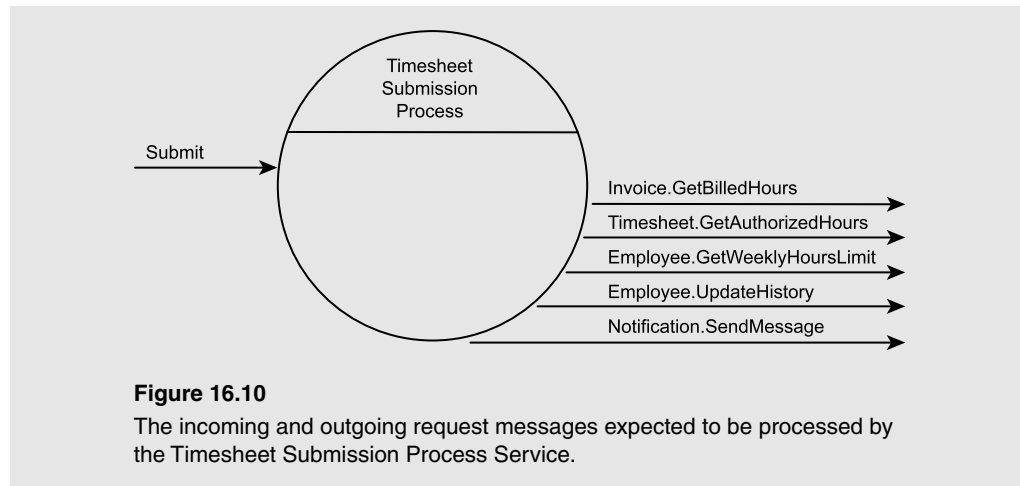
Figure 16.8 illustrates the interaction between services required to successfully complete the Timesheet Submission Process with a valid timesheet submission. Note that in this scenario, the Notification Service is not used.

Figure 16.9 demonstrates a scenario in which the timesheet document is rejected by the Timesheet Service. This occurs because the timesheet failed to receive proper authorization.

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

Now that we understand the message exchange requirements, we can proceed to define a service definition for the process service. When working with process modeling tools, the process service WSDL definition will typically be auto-generated for you. However, you also should be able to edit the source markup code or even import your own WSDL definition.

Either way, it is best to review the WSDL definition being used and revise it as necessary. Here are some suggestions:

- 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 that 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 additional design standards within the confines of the modeling tool.

There is less opportunity to incorporate the other steps from the service design processes described in Chapter 15. For example, applying the service-orientation principle of statelessness is difficult, given that process services maintain state so that other services don't have to.

CASE STUDY

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).

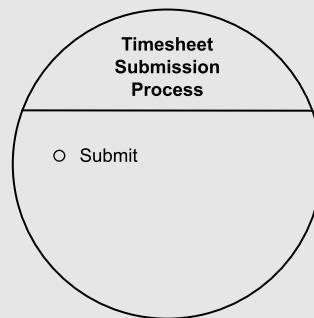


Figure 16.11
Timesheet Submission Process
Service design.

Following is the corresponding WSDL definition:

```
<definitions name="TimesheetSubmission"
  targetNamespace="http://www.xmltc.com/tls/process/wsd1/"
  xmlns="http://schemas.xmlsoap.org/wsd1/"
  xmlns:ts="http://www.xmltc.com/tls/timesheet/schema/"
  xmlns:tsd=
    "http://www.xmltc.com/tls/timesheetservice/schema/"
  xmlns:soap="http://schemas.xmlsoap.org/wsd1/soap/"
  xmlns:tns="http://www.xmltc.com/tls/timesheet/wsd1/"
  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 16.18 The abstract service definition for the Timesheet Submission Process Service.

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

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 `partnerLinkType` 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.

CASE STUDY

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 (as designed in Chapter 15) is amended to incorporate the WS-BPEL `partnerLinkType` construct and its corresponding namespace.

```
<definitions
  name="Employee"
  targetNamespace="http://www.xmltc.com/tls/employee/wsd1/"
  xmlns="http://schemas.xmlsoap.org/wsd1/"
  xmlns:act=
    "http://www.xmltc.com/tls/employee/schema/accounting/"
  xmlns:hr="http://www.xmltc.com/tls/employee/schema/hr/"
  xmlns:soap="http://schemas.xmlsoap.org/wsd1/soap/"
  xmlns:tns="http://www.xmltc.com/tls/employee/wsd1/"
  xmlns:plnk=
    "http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
  ...
  <plnk:partnerLinkType name="EmployeeType">
    <plnk:role name="EmployeeService">
      <plnk:portType name="tns:EmployeeInterface"/>
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

```
</plnk:role>
</plnk:partnerLinkType>
</definitions>
```

Example 16.19 The revised Employee service definitions construct.

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.

```
<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>
```

Example 16.20 The `partnerLinks` construct containing `partnerLink` elements for each of the process partner services.

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.

```
<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>
```

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.

If you check back to the Employee Service definition TLS designed in Chapter 15, you'll notice that the name values of the message elements correspond to the values assigned to the `messageType` attributes in the previously displayed `variable` elements.

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.

CASE STUDY

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. (Note that this diagram illustrates the process flow that corresponds to the success condition expressed by the first of the two activity diagrams created during Step 1.)

NOTE

The complete process definition is several pages long and therefore is not displayed here. Instead, we highlight relevant parts of the process, such as activities and fault handling. The entire process definition is available for download at www.serviceoriented.ws.

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.

```
<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 16.22 The `receive` element providing an entry point by which the process can be initiated.

By tracing the `receive` element's `operation` value back to the original Timesheet Submission Service WSDL, you can find out that the expected format of the input data will be a complete timesheet document, defined in a separate XSD schema document. When a document is received, it is stored in the `ClientSubmission` process variable.

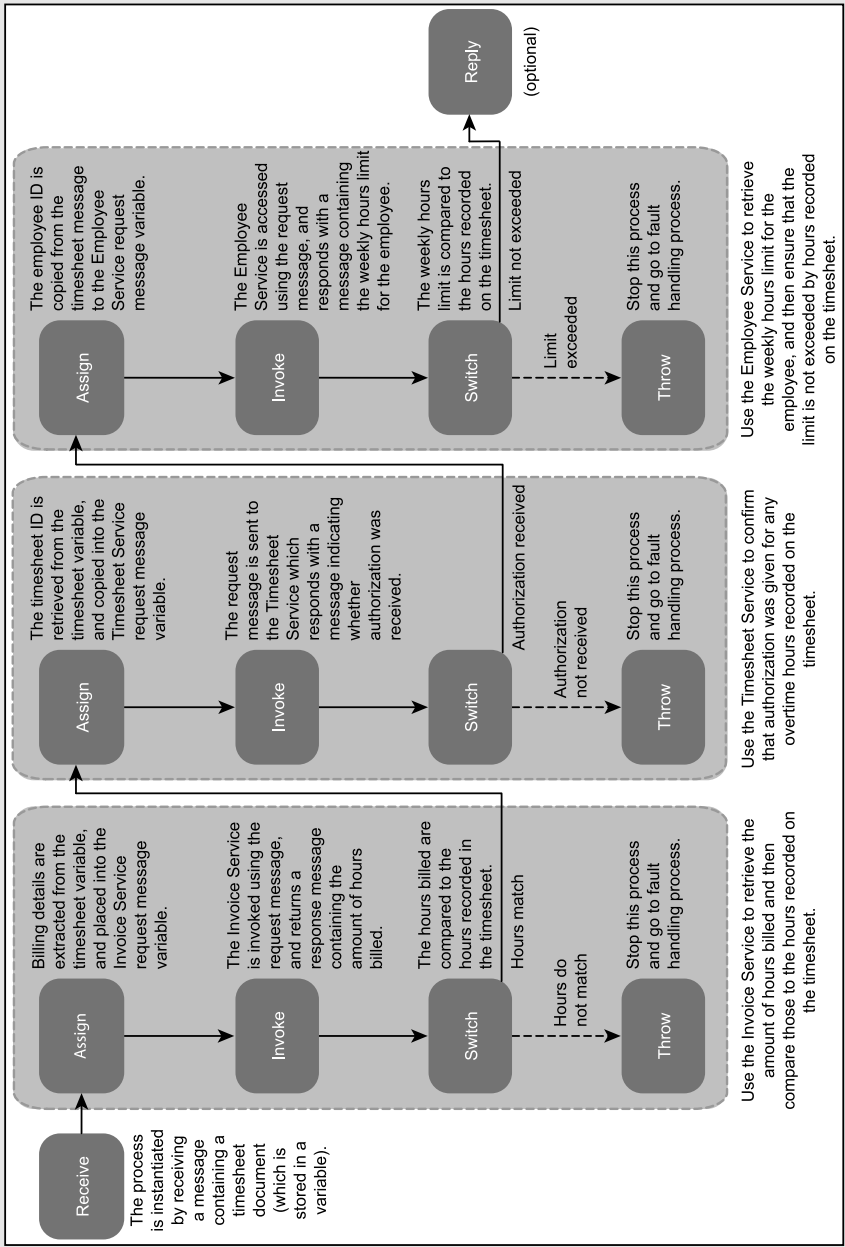


Figure 16.12
A descriptive, diagrammatic view of the process definition logic.

As per the interaction logic defined in the initial activity diagram (Step 1), the first activity the process is required to perform is to interact with the Invoice Service to compare the hours submitted on the timesheet with the hours actually billed out to the client. The Invoice Service will not perform the comparison for the process; instead, it will simply provide the amount of hours for a given invoice identified by an invoice number.

Before invoking the Invoice Service, the `assign` construct first needs to be used to extract values from the original timesheet document, which now is stored in the `ClientSubmission` variable. Specifically, the customer ID and date values (encapsulated in the `BillingInfo` element) are required as input for the Invoice Service's `GetBilledHours` operation.

```
<assign name="GetInvoiceID">
  <copy>
    <from variable="ClientSubmission" part="payload"
      query="/TimesheetType/BillingInfo"/>
    <to variable="InvoiceHoursRequest"
      part="RequestParameter"/>
  </copy>
</assign>
```

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.

Now that the `InvoiceHoursRequest` variable contains the required input values for the Invoice Service's `GetBilledHours` operation, the `invoke` element is added.

```
<invoke name="ValidateInvoiceHours"
  partnerLink="Invoice"
  operation="GetBilledHours"
  inputVariable="InvoiceHoursRequest"
  outputVariable="InvoiceHoursResponse"
  portType="inv:InvoiceInterface"/>
```

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.

Upon invoking the Invoice Service, a response message is received from the Get-BilledHours operation. As defined in the `invoke` element's `outputVariable` attribute, this message is stored in the `InvoiceHoursResponse` variable.

If the value in this variable matches the value in the timesheet document, then the hours have been validated. To determine this, the `switch` construct is used. A child `case` construct is added, which contains a `condition` attribute in which the conditional logic is defined.

```
<switch name="BilledHoursMatch">
  <case condition=
    "getVariableData('InvoiceHoursResponse',
                     'ResponseParameter') !=
    getVariableData('input','payload',
                    '/tns:TimesheetType/Hours/...')">
    <throw name="ValidationFailed"
          faultName="ValidateInvoiceHoursFailed"/>
  </case>
</switch>
```

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.

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.

A good part of the remaining process logic repeats this pattern, as illustrated in the original process overview displayed back in Figure 16.12. For brevity, this part of the process is summarized here:

- Use the `assign` construct to copy the `TimesheetID` value from the `ClientSubmission` variable to the `TimesheetAuthorizationRequest` variable that is used via the `invoke` element as the input message for the `GetAuthorizedHours` operation of the `Timesheet` service. The authorization result is extracted from the response message within the `switch` construct, and if positive, the process proceeds to the next step. If authorization fails, a fault is raised using the `throw` element.
- Using the `assign` element, the `EmployeeID` value is retrieved from the `ClientSubmission` variable and placed in the `EmployeeHoursRequest` variable. This variable becomes the request message used by the `invoke` element to communicate with the `Employee Service's GetWeeklyHoursLimit` operation. The response message from that operation is submitted to the `condition` attribute of the `case` element within the `switch` construct. The result is a determination as to whether the employee exceeded the allowed maximum hours per week. If the value was exceeded, the process jumps to the `faultHandlers` construct.

That pretty much sums up the primary processing logic of the `TLS Timesheet Submission Process`. Although the initial requirements do not call for it, the process flow could end with a `reply` element that responds to the initial client that instantiated the process.

Now it's time to turn our attention to the second scenario (portraying a failure condition) mapped out in the other activity diagram from Step 1. To accommodate this situation, `TLS architects` choose to implement a `faultHandlers` construct, as shown here:

```
<faultHandlers>
  <catchAll>
    <sequence>
```

```
...  
    </sequence>  
  </catchAll>  
</faultHandlers>
```

Example 16.26 The `faultHandlers` construct used in this process.

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.

Note that the following, abbreviated markup code samples reside within the sequence child construct of the parent `faultHandlers` construct established in the previous example.

First up is the markup code for the "Update employee profile history" task.

```
<assign name="SetEmployeeMessage">  
  <copy>  
    <from variable="ClientSubmission" .../>  
    <to variable="EmployeeHistoryRequest" .../>  
  </copy>  
  <copy>  
    <from expression="...">  
    <to variable="EmployeeHistoryRequest" .../>  
  </copy>  
</assign>  
<invoke name="UpdateHistory"  
  partnerLink="Employee"  
  portType="emp:EmployeeInterface"  
  operation="UpdateHistory"  
  inputVariable="EmployeeHistoryRequest"  
  outputVariable="EmployeeHistoryResponse"/>
```

Example 16.27 Two `copy` elements used to populate the `EmployeeHistoryRequest` message.

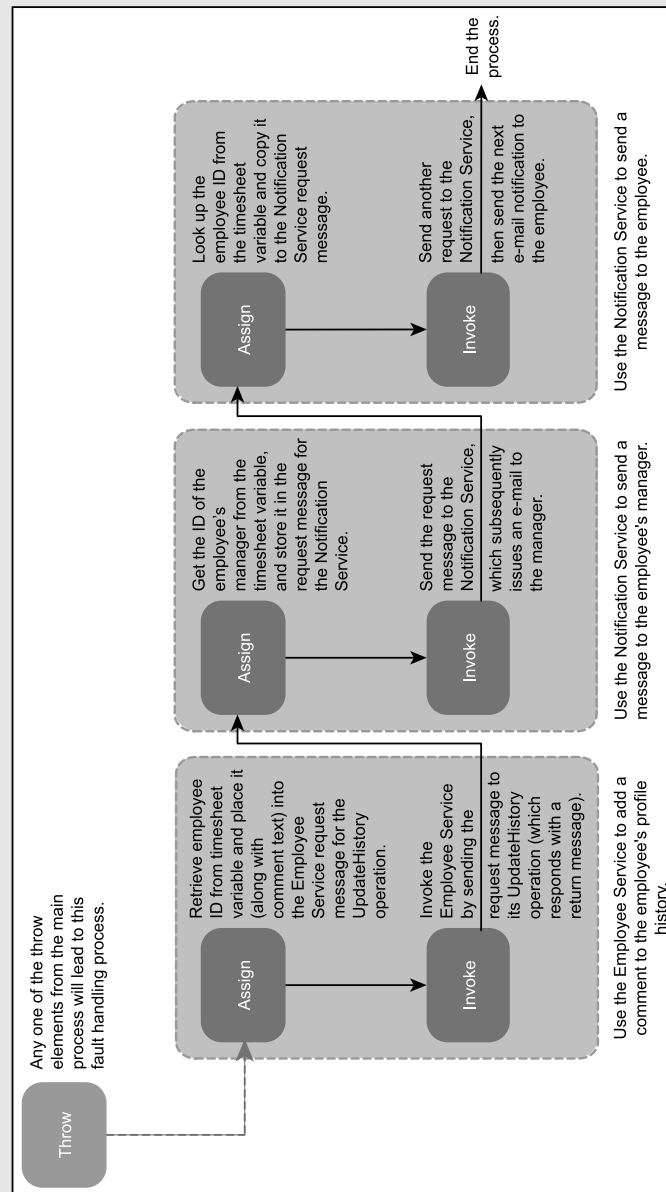


Figure 16.13

A visual representation of the process logic within the `fault-handlers` construct.

To perform the first task of updating the employee history, the fault handler routine uses an `assign` construct with two `copy` constructs. The first retrieves the `EmployeeID` value from the `ClientSubmission` variable, while the latter adds a static employee profile history comment.

The `invoke` element then launches the Employee Service (used previously for its `GetWeeklyHoursLimit` operation) and submits the `EmployeeHistoryRequest` message to its `UpdateHistory` operation to log the profile history comment.

The next block of markup code takes care of both the remaining “Send notification” tasks.

```
<assign name="GetManagerID">
  <copy>
    <from expression="getVariableData(...)" />
    <to variable="NotificationRequest" ... />
  </copy>
</assign>
<invoke name="SendNotification"
  partnerLink="Notification"
  portType="not:NotificationInterface"
  operation="SendMessage"
  inputVariable="NotificationRequest" />
<assign name="GetEmployeeID">
  <copy>
    <from expression="getVariableData(...)" />
    <to variable="NotificationRequest" ... />
  </copy>
</assign>
<invoke name="SendNotification"
  partnerLink="Notification"
  portType="not:NotificationInterface"
  operation="SendMessage"
  inputVariable="NotificationRequest" />
<terminate name="EndTimesheetSubmissionProcess" />
```

Example 16.28 The last activities in the process.

The `faultHandlers` construct contains two more `assign` + `invoke` element pairs. Both use the Notification Service’s `SendMessage` operation, but in different ways. The first `assign` construct extracts the `ManagerID` value from the `ClientSubmission` variable, which is then passed to the Notification Service. It is

the sole parameter that the service subsequently uses to look up the corresponding e-mail address for the notification message.

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)

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 then can 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.

CASE STUDY

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.

Here's a recap of 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.

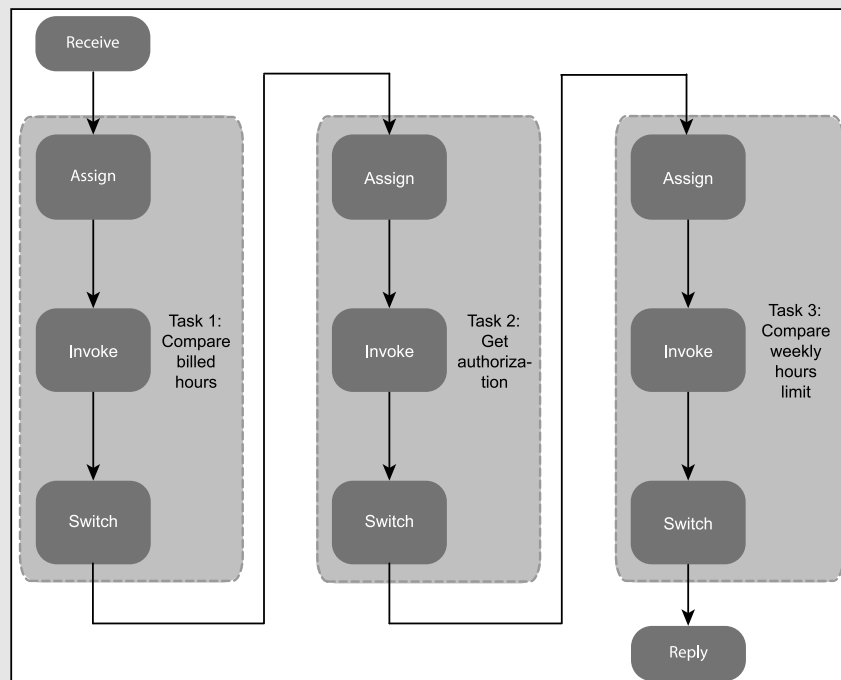


Figure 16.14 Sequential, synchronous execution of process activities.

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.

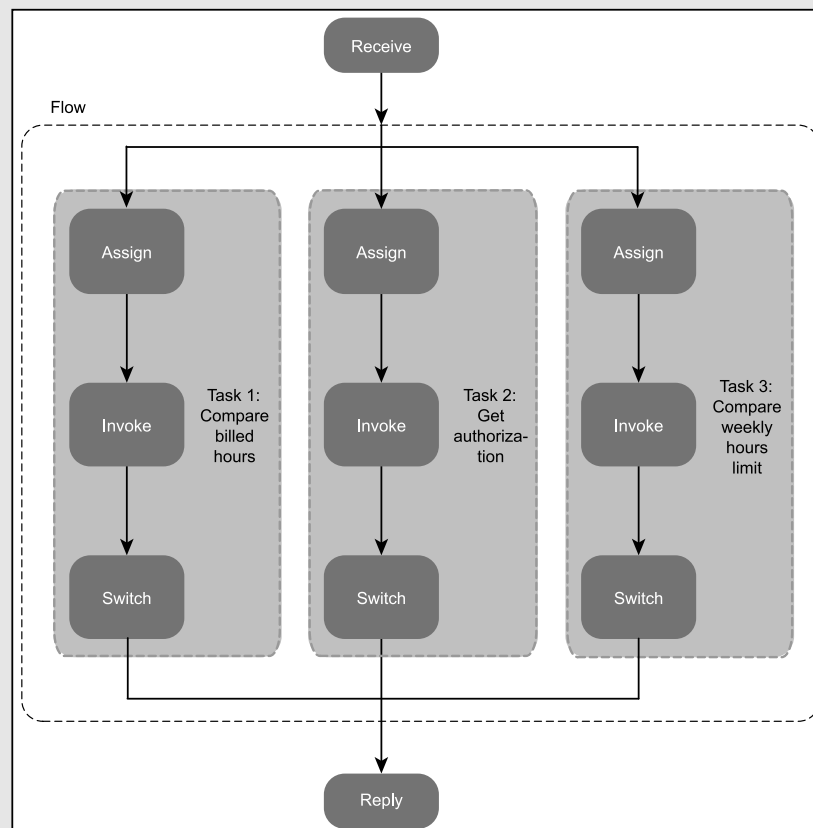


Figure 16.15 Concurrent execution of process activities using the `flow` construct.

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.

SUMMARY OF KEY POINTS

- Designing a process service requires the design of the service interface and the design of the process definition.
 - Process definition is typically accomplished using a graphical modeling tool, but familiarity with WS-BPEL language basics is often still required.
 - There are numerous ways in which WS-BPEL process definitions can be streamlined and optimized.
-

Sample Chapter 16 from "Service-Oriented Architecture: Concepts, Technology, and Design" by Thomas Erl
For more information visit: www.soabooks.com



About the Author

Thomas Erl is an independent consultant with XMLTC Consulting in Vancouver, Canada. His previous book, *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, became the top-selling book of 2004 in both Web Services and SOA categories. This guide addresses numerous integration issues and provides strategies and best practices for transitioning toward SOA.

Thomas is a member of OASIS and is active in related research efforts, such as the XML & Web Services Integration Framework (XWIF). He is a speaker and instructor for private and public events and conferences, and has published numerous papers, including articles for the *Web Services Journal*, *WLDJ*, and *Application Development Trends*.

For more information, visit <http://www.thomaserl.com/technology/>.

Sample Chapter 16 from "Service-Oriented Architecture: Concepts, Technology, and Design" by Thomas Erl
For more information visit: www.soabooks.com

About SOA Systems

SOA Systems Inc. is a consulting firm actively involved in the research and development of service-oriented architecture, service-orientation, XML, and Web services standards and technology. Through its research and enterprise solution projects SOA Systems has developed a recognized methodology for integrating and realizing service-oriented concepts, technology, and architecture.

For more information, visit www.soasystems.com.

One of the consulting services provided by SOA Systems is comprehensive SOA transition planning and the objective assessment of vendor technology products.

For more information, visit www.soaplanning.com.

The content in this book is the basis for a series of SOA seminars and workshops developed and offered by SOA Systems.

For more information, visit www.soatraining.com.