**Business Process Execution Language (BPEL) Coding Practice**

Date: March 12, 2015

Table of Contents

[1 Introduction 3](#_Toc418800708)

[2 Reference 4](#_Toc418800709)

[3 Purpose 5](#_Toc418800710)

[4 Programing Standards 6](#_Toc418800711)

[4.1 Naming Conventions 6](#_Toc418800712)

[4.1.1 General Naming Standards 6](#_Toc418800713)

[4.1.2 General Namespace Naming Standards 6](#_Toc418800714)

[4.1.3 BPEL Activites 7](#_Toc418800715)

[4.1.4 Other BPEL Artifacts 13](#_Toc418800716)

[4.2 Fault Handling 13](#_Toc418800717)

[4.2.1 BPEL Standard Faults 14](#_Toc418800718)

[4.2.2 Categories of BPEL Faults 16](#_Toc418800719)

[4.2.3 BPEL activities to propagate a fault 17](#_Toc418800720)

[4.2.4 Use a fault handler within a scope 18](#_Toc418800721)

[4.2.5 Use compensation after undoing a series of operations 18](#_Toc418800722)

[4.2.6 Use the terminate activity to stop a business process instance 19](#_Toc418800723)

[4.3 Code Level Metrics 19](#_Toc418800724)

[5 Industry Best Coding Practice 19](#_Toc418800725)

[5.1 Avoid interruptible processes 20](#_Toc418800726)

[5.2 Avoid missing OTHERSWISE in SWITCH statements 20](#_Toc418800727)

[5.3 Avoid using invoke without catch 20](#_Toc418800728)

[5.4 Create scopes for each step of the flow in the process 20](#_Toc418800729)

[5.5 Use short-running flows where possible 20](#_Toc418800730)

[5.6 Use global variables when required 21](#_Toc418800731)

[5.7 Consider creating sequences instead of scopes for performance 21](#_Toc418800732)

[5.8 Use assign activities instead of transform activities 21](#_Toc418800733)

[5.9 Use XSL to construct messages instead of multiple assign statements 21](#_Toc418800734)

[5.10 Use XPath Expression constraints to check the data constraints 21](#_Toc418800735)

[5.11 Initialize a BPEL variable before copying values to internal elements 21](#_Toc418800736)

[5.12 Use <nonBlockingInvoke> in BPEL flow for performance improvements 22](#_Toc418800737)

[5.13 Keep the number of activities in BPEL as minimal as possible 24](#_Toc418800738)

[5.14 Avoid unused partner links 24](#_Toc418800739)

[5.15 Avoid repetitive names for different activities 24](#_Toc418800740)

[5.16 Don’t use BPEL for intensive time scheduled activities 24](#_Toc418800741)

[5.17 Do not include any special characters in the project name 24](#_Toc418800742)

[5.18 Don’t use the pick based initiate pattern for interdependent operations 24](#_Toc418800743)

[5.19 Don’t let a SOA composite instance grow exponentially 24](#_Toc418800744)

# Introduction

[Business Process Execution Language for Web Services](http://www-128.ibm.com/developerworks/library/ws-bpel/) (BPEL or BPEL4WS) is a language used for the definition and execution of business processes using Web services. BPEL enables the top-down realization of Service Oriented Architecture (SOA) through composition, orchestration, and coordination of Web services. BPEL provides a relatively easy and straightforward way to compose several Web services into new composite services called business processes, which is a collection of coordinated service invocations and related activities that produce a result.

BPEL builds on the foundation of XML and Web services; it uses an XML-based language that supports the Web services technology stack, including SOAP, WSDL, UDDI, WS-Reliable Messaging, WS-Addressing, WS-Coordination, and WS-Transaction. BPEL is used to standardize enterprise application integration as well as to extend the integration to the Consular Affairs (CA) legacy applications and services where an Oracle Enterprise Service Bus is a middleware solution as an intermediary between service requesters and service providers.

Using BPEL a series of discret services can be integrated into an end-to-end process flow that reduces process cost and complexity. The BPEL enables to define how to:

* Send XML messages to, and asynchronously receive XML messages from, remote services.
* Manipulate XML data structures.
* Manage events and exceptions.
* Design parallel flows of process execution.
* Undo portions of processes when exceptions occur.

# Reference

The following sources were used in creation of the original version of this standard.

* + Oracle Infrastructure Components and Utilities User's Guide for Oracle Application Integration Architecture Foundation Pack, <https://docs.oracle.com/cd/E28280_01/doc.1111/e17366/toc.htm>
  + [Fusion Middleware Developer's Guide for Oracle Application Integration Architecture Foundation Pack, 11g Release 1 (11.1.1.7)](https://docs.oracle.com/cd/E28280_01/doc.1111/e17364/toc.htm), <https://docs.oracle.com/cd/E28280_01/doc.1111/e17364/toc.htm>
  + Fusion Middleware Developer's Guide for Oracle SOA Suite, <https://docs.oracle.com/cd/E28280_01/dev.1111/e10224/bp_appx_ref.htm#SOASE2070>
  + OASIS WS-BPEL v2.0 specification, <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf>.

# Purpose

The purpose of this document is to collect industry best practices and recommendations for BPEL coding to improve the overall quality, performance, security, maintainability, and robustness of Oracle BPEL based solution implementations. Conformance to the practices will promote measurable improvements in the performance of codes and a reduction in sustainment costs.

Coding guidelines in the forms of code metrics and coding rules are provided. This document is intended for the CST Systems Engineering and Integration (SEI) team and development teams. Upon review of this document, the best practices will be converted to CAST rules.

# Programing Standards

## Naming Conventions

### General Naming Standards

Follow these general naming standards:

* Lower-Camel-case must be used for naming attributes.

Example: <xsd:attribute name="unitCode"/>

* Upper-Camel-case must be used for naming elements and types.

Example: <xsd:element name="UnitOfMeasure"/> <xsd:complexType name="InvoiceEBOType"/>

* Names must be singular unless the concept itself is plural.

For example repeating elements must have a singular name.

* Names must not contain special characters such as: space, '-', '\_', '.', '$', '%', '#', ....
* Avoid having numeric characters in the name.

There are cases were using a numeric character is required to convey some significance.

* Complex type names should end with the 'Type' suffix to help recognize types from elements.

Example: <xsd:complexType name="InvoiceEBOType"/>

* The name of a simple type definition should be the name of the root element with the 'ContentType' suffix.

Example: <xsd:simpleType name="PhoneNumberContentType">

### General Namespace Naming Standards

Follow these general namespace naming standards:

* All namespaces must start with <http://xmlns.oracle.com/>.
* Namespaces used by Enterprise Business Objects(EBOs) and Enterprise Business Messages (EBMs) start with http://xmlns.oracle.com/EnterpriseObjects/.

Example: <http://xmlns.oracle.com/EnterpriseObjects/Core/EBO/Invoice/V1>

* Namespaces used for externally facing services must start with http://xmlns.oracle.com/EnterpriseServices/.

Examples: <http://xmlns.oracle.com/EnterpriseServices/Core/Invoice/V1>

* Namespaces for versioned artifacts must have the major version number as a suffix with 'V' as an abbreviation for 'version'.

Example: http://xmlns.oracle.com/ EnterpriseObjects/Core/EBO/Invoice/V1

* The namespace structure should closely map to the taxonomy of the types it encapsulates.

Example: Horizontal: http://xmlns.oracle.com/EnterpriseObjects/Core/EBO/Invoice/V1.

* Namespaces for artifacts generated within ABCSs must start with: <http://xmlns.oracle.com/ABCS/>
* When importing or including schema in a schema file, the schema location must always use relative path.

Example: <xsd:importnamespace="http://xmlns.oracle.com/EnterpriseObjects/Core/EBO/Invoice/V1" schemaLocation="../../../../Core/EBO/Invoice/InvoiceEBO .xsd"/>

* Namespace prefixes must be a minimum of six (6) lowercase characters abbreviation of the namespace. The abbreviation must be descriptive and unambiguous within the context where it is being used.
* Namespace prefixes for EBOs and EBMs must adhere to the following standard wherever used regardless of the applications or technology used. Auto-generated prefixes such as ns1, ns2 must not be used. Auto-generated prefixes for standard namespaces such as xsd, xsi are acceptable.

### BPEL Activites

#### BPEL Process Name and Namespace

The BPEL process JDeveloper project name should match the BPEL process name (use default project setting).

Name standards

* The name should follow the general standard naming standards depending on whether it is being used for EBS, Application Business Connector Services (ABCS), or Adapter Service.
* The name should clearly describe the process and action/verb being performed.

Namespace standards:

* The namespace should follow the general namespace standards depending on whether it is being used for EBS, ABCS or Adapter Service.
* The namespace must reflect the taxonomy of the process.
* The namespace must include the major version number where appropriate.
* BPEL composite's reference component name should follow the general naming standards based on the type of AIA artifacts it is calling.

#### Assign

Follow these guidelines:

* The name should follow the general standard naming standards.
* Starts with the **Assign** prefix.
* Followed by a name describing what is being assigned. If what is assigned is a message, then use the message name.
* In case there are multiple assignments, provide a name that describes the group of assignments if possible.

Pattern: **Assign<Name of what is being assigned>**

Example: *AssignPaymentEBM, AssignOrderInitialValues*

#### Compensate

Follow these guidelines:

* The name should follow the general standard naming standards.
* Start with the **Compensate** prefix.
* Followed by the scope encapsulating the tasks to be compensated.

Pattern: **Compensate<scope name>**

Example: *CompensateProcessCreditCheckMilestone, Compensate TranseferFundsScope*

#### Flow

Follow these guidelines:

* The name should follow the general standard naming standards.
* Starts by a name describing the tasks being run concurrently.
* Ends with the **Flow** suffix.

Pattern: **<Name describing concurrent tasks>Flow**

Example: *CallManufacturersFlow, GetQuotesFlow*

#### FlowN

Follow these guidelines:

* The name should follow the general standard naming standards.
* Starts by a name describing the dynamic tasks being run concurrently.
* Ends with the **FlowN** suffix.
* The index variable name should be the flow name with **Index** as suffix.

Pattern: **name = <Name describing concurrent tasks>FlowN, index variable = <Name describing concurrent tasks>FlowNIndex**

Example: *ActivateUsersFlowN (ActivateUsersFlowNIndex), CheckSuppliersFlowN (CheckSuppliersFlowNIndex)*

#### Invoke

Follow these guidelines:

* The name should follow the general standard naming standards.
* Starts with the **Invoke** prefix.
* Followed by the partner link to be invoked.
* Followed by **Call** if synchronous invocation or **Start** if asynchronous invocation.
* Followed by the operation name within the partner link.

Pattern: **Invoke<Partner Link Name>{Call/Start}<Operation>**

Example: *InvokeCustomerServiceCallGetCustomer, InvokeNotificationServiceStartNotifyByEmail*

#### Java Embedding

Follow these guidelines:

* The name should follow the general standard naming standards.
* The name should be similar to a Java method Name with lower-camel-case.

Pattern: **<A name describing the functionality>**

Example: *getDiscountPrice*

#### Pick

Follow these guidelines:

* The name should follow the general standard naming standards.
* Starts with the **Pick** prefix.
* Followed by a name describing as accurate as possible all branches (onMessage and onAlarm) within the pick activity.

Pattern: **Pick<Name describing the branches to pick from>**

Example: *PickOrderAckOrTimeout, PickFirstQuote*

#### Receive

Follow these guidelines:

* The name should follow the general standard naming standards.
* Starts with the **Receive** prefix.
* Contains the name of the message it is receiving.

Pattern: **Receive<Message Name>**

Example: *ReceiveUpdateInvoiceEBM*

#### Scope

Follow these guidelines:

* The name should follow the general standard naming standards.
* Including brief information about transaction type may be appropriate.
* Use **Milestone** as the suffix if the scope is a candidate for end-user monitor.
* If it is not intended for the end-user monitor, use **Scope** as the suffix.

Pattern: **<Name describing the Scoped Tasks>{ Scope |Milestone}**

Examples: *GetCreditRatingScope, GetLoanOfferScope, ProcessCreditCheckMilestone*

#### Sequence

Follow these guidelines:

* The name should follow the general standard naming standards.
* The sequence name should describe the steps performed in the sequence.
* The sequence name should end with **Sequence** suffix.

Pattern: **<Name describing the Sequenced Tasks>Sequence**

Example: *GetCustomerInfoSequence*

#### Switch

Follow these guidelines:

* The name should follow the general standard naming standards.
* Start with the **Switch** prefix.
* Followed by what is being evaluated

Pattern: **Switch<Name of what is being evaluated>**

Example: *SwitchCreditRating*

#### Case

Follow these guidelines:

* The name should follow the general standard naming standards.
* Start with the **Case** prefix.
* Followed by the evaluated value.

Pattern: **Case<Name evaluated value>**

Example: *CaseBadCredit, CaseApprovalRequired*

#### Terminate

Follow these guidelines:

* The name should follow the general standard naming standards.
* Starts with the **Terminate** prefix.
* Followed by a name describing the termination reason.

Pattern: **Terminate<reason of termination>**

Example: *TerminateTimeout, TerminateEndOfProcess*

#### Throw

Follow these guidelines:

* The name should follow the general standard naming standards.
* Starts with the **Throw** prefix.
* Followed by the fault name.
* The fault variable name is typically named the same as the fault name.

Pattern: **Throw<fault name>**

Example: *ThrowExceededMaxAmount*, which uses ExceededMaxAmount variable.

**Note:**

When defining a Catch in the Scope activity, the displayed catch name is the fault name.

#### Transform

Follow these guidelines:

The name should follow the general standard naming standards.

* Starts with the **Xform** prefix.
* Followed by the source name.
* Followed by **To**.
* Followed by the destination name.

Pattern: **Xform<source>To<destination>**

Example: *XformBillToPortal80Bill*

#### Wait

Follow these guidelines:

The name should follow the general standard naming standards.

* Starts with the **Wait** prefix.
* Followed by a name describing the reason for waiting.

Pattern: **Wait<Name describing the waiting reason>**

Example: *WaitOrderAcknowledgeTimeout, WaitWarmUpTime*

#### While

Follow these guidelines:

* The name should follow the general standard naming standards.
* Starts with the **While** prefix.
* Followed by a name describing the loop condition.

Pattern: **While<Name describing the loop condition>**

Example: *WhileAllMsgsSent*

### Other BPEL Artifacts

#### Variables

Follow these guidelines:

* The name should follow the general standard naming standards.
* Use lower-camel-case for variable names.
* The data type must not be part of the variable name.

Example: *accountBalance, invoiceAmount*.

#### Properties

Property names follow the general BPEL variables naming standards.

#### Correlation Sets

Follow these guidelines:

* The name should follow the general standard naming standards.
* Starts with a name describing the correlation set.
* Ends with **CorSet** suffix.

Pattern: **<Name describing the correlation>CorSet**

Example: *PurchaseOrderCorSet*

#### Correlation Set Properties

The correlation set property names follows the general BPEL variables naming standards.

## Fault Handling

The term *fault* refers to any exceptional condition that can alter the normal processing of a business process. Within the context of a business process, a fault need not result in a process ending condition; instead, a fault should lead to an actionable event. If a fault is not handled, it will lead to unexpected conditions, outcomes or even unpredictable failures of the business process. A well-designed business process should handle faults so that failures lead to predictable outcomes. Within BPEL, fault handlers can catch faults and attach business relevant execution logic to deal with the exceptional situation.

Explicit fault handlers, if used, attached to a scope provide a way to define a set of custom fault-handling activities, defined by <catch> and <catchAll> constructs. Each <catch> construct is defined to intercept a specific kind of fault, defined by a fault QName. An optional variable can be provided to hold the data associated with the fault. If the fault name is missing, then the catch will intercept all faults with the same type of fault data. The fault variable is specified using the faultVariable attribute in a <catch> fault handler. The variable is deemed to be implicitly declared by virtue of being used as the value of this attribute and is local to the fault handler. It is not visible or usable outside the fault handler in which it is declared. A <catchAll> clause can be added to catch any fault not caught by a more specific fault handler.

### BPEL Standard Faults

#### BPEL 2.0 Standard Faults

The following list specifies the standard faults defined within the WS-BPEL specification. All standard fault names are qualified with the standard WS-BPEL namespace.

| **Fault name** | **Description** |
| --- | --- |
| ambiguousReceive | Thrown when a business process instance simultaneously enables two or more IMAs for the same partnerLink, portType, operation but different correlationSets, and the correlations of multiple of these activities match an incoming request message. |
| completionConditionFailure | Thrown if upon completion of a directly enclosed <scope> activity within <forEach> activity it can be determined that the completion condition can never be true. |
| conflictingReceive | Thrown when more than one inbound message activity is enabled simultaneously for the same partner link, port type, operation and correlation set(s). |
| conflictingRequest | Thrown when more than one inbound message activity is open for the same partner link, operation and message exchange. |
| correlationViolation | Thrown when the contents of the messages that are processed in an <invoke>, <receive>, <reply>, <onMessage>, or <onEvent> do not match specified correlation information. |
| invalidBranchCondition | Thrown if the integer value used in the <branches> completion condition of <forEach> is larger than the number of directly enclosed <scope> activities. |
| invalidExpressionValue | Thrown when an expression used within a WS-BPEL construct (except <assign>) returns an invalid value with respect to the expected XML Schema type. |
| invalidVariables | Thrown when an XML Schema validation (implicit or explicit) of a variable value fails. |
| joinFailure | Thrown when the join condition of an activity evaluates to false and the value of the suppressJoinFailure attribute is yes. |
| mismatchedAssignmentFailure | Thrown when incompatible types or incompatible XML infoset structure are encountered in an <assign> activity. |
| missingReply | Thrown when an inbound message activity has been executed, and the process instance or scope instance reaches the end of its execution without a corresponding <reply> activity having been executed. |
| missingRequest | Thrown when a <reply> activity cannot be associated with an open inbound message activity by matching the partner link, operation and message exchange tuple. |
| scopeInitializationFailure | Thrown if there is any problem creating any of the objects defined as part of scope initialization. This fault is always caught by the parent scope of the faulted scope. |
| selectionFailure | Thrown when a selection operation performed either in a function such as bpel:getVariableProperty, or in an assignment, encounters an error. |
| subLanguageExecutionFault | Thrown when the execution of an expression results in an unhandled fault in an expression language or query language. |
| uninitializedPartnerRole | Thrown when an <invoke> or <assign> activity references a partner link whose partnerRole endpoint reference is not initialized. |
| uninitializedVariable | Thrown when there is an attempt to access the value of an uninitialized variable or in the case of a message type variable one of its uninitialized parts. |
| unsupportedReference | Thrown when a WS-BPEL implementation fails to interpret the combination of the reference-scheme attribute and the content element OR just the content element alone. |
| xsltInvalidSource | Thrown when the transformation source provided in a bpel:doXslTransform function call was not legal (i.e., not an EII). |
| xsltStylesheetNotFound | Thrown when the named style sheet in a bpel:doXslTransform function call was not found. |

##### Fault handling order of precedence in BPEL 2.0

In BPEL 2.0, the order of precedence for catching faults thrown without associated data is as follows:

* If there is a catch activity with a matching faultName value that does not specify a faultVariable attribute, the fault is sent to the identified catch activity.
* Otherwise, if there is a catchAll activity, the fault is sent to the catchAll fault handler.
* Otherwise, the fault is processed by the default fault handler.

In BPEL 2.0, the order of precedence for catching faults thrown with associated data is as follows:

* If there is a catch activity with a matching faultName value that does not specify a faultVariable attribute, the fault is sent to the identified catch activity.
* If the fault data is a WSDL message type in which the following exists:
  + The message contains a single part defined by an element.
  + There exists a catch activity with a matching faultName value that has a faultVariable whose associated faultElement QName matches the QName of the runtime element data of the single WSDL message part.

Then, the fault is sent to the identified catch activity with the faultVariable initialized to the value in the single part's element.

* Otherwise, if there is a catch activity with a matching faultName value that does not specify a faultVariable attribute, the fault is sent to the identified catch activity. In this case, the fault value is not available from within the fault handler, but is available to the rethrow activity.
* Otherwise, if there is a catch construct without a faultName attribute that has a faultVariable whose type matches the type of the runtime fault data, then the fault is sent to the identified catch activity.
* Otherwise, if the fault data is a WSDL message type in which the message contains a single part defined by an element and there exists a catch activity without a faultName attribute that has a faultVariable whose associated faultElement's QName matches the QName of the runtime element data of the single WSDL message part, the fault is sent to the identified catch activity with the faultVariable initialized to the value in the single part's element.
* Otherwise, if there is a catchAll activity, the fault is sent to the catchAll fault handler.
* Otherwise, the fault is handled by the default fault handler.

### Categories of BPEL Faults

A BPEL fault has a fault name called a Qname (name qualified with a namespace) and a possible messageType. There are two categories of BPEL faults:

* Business faults
* Runtime faults

#### Business Faults

Business faults are application-specific faults that are generated when there is a problem with the information being processed (for example, when a social security number is not found in the database). A business fault occurs when an application executes a throw activity or when an invoke activity receives a fault as a response. The fault name of a business fault is specified by the BPEL process service component. The messageType, if applicable, is defined in the WSDL. A business fault can be caught with a faultHandler using the faultName and a faultVariable.

<catch faultName="ns1:faultName" faultVariable="varName">

#### Runtime Faults

Runtime faults are the result of problems within the running of the BPEL process service component or web service (for example, data cannot be copied properly because the variable name is incorrect). These faults are not user-defined, and are thrown by the system. They are generated if the process tries to use a value incorrectly, a logic error occurs (such as an endless loop), a Simple Object Access Protocol (SOAP) fault occurs in a SOAP call, an exception is thrown by the server, and so on.

* bindingFault: a bindingFault is thrown inside an activity if the preparation of the invocation fails. For example, the WSDL of the process fails to load. A bindingFault is not retryable. This type of fault usually must be fixed by human intervention.
* remoteFault: a remoteFault is also thrown inside an activity. It is thrown because the invocation fails. For example, a SOAP fault is returned by the remote service.
* replayFault: a replayFault replays the activity inside a scope. At any point inside a scope, this fault is migrated up to the scope. The server then re-executes the scope from the beginning.

### Typical fault handling flow

When a fault occurs in a process, the current operational flow moves to the fault handler within the immediate scope. If the current fault handler does not have the appropriate Catch element defined to trap and handle this fault, the business process container checks from inner scope to outer scope until it finds an appropriate fault handler that can trap this fault.

The Catch mechanism handles a specific application or standard fault. In cases when the type of a fault is unknown, the Catch All element can be used. It is a best practice to use a Catch All element in the global process’s fault handler.

When designing fault handlers, consider the following options:

* Catch a fault and try to correct the problem, allowing the business process to continue to normal completion.
* Catch a fault and find that it is not resolvable at this scope. Additional options can be used:
* Throw a new fault.
* Re-throw the original fault to allow another scope to handle it.
* Reply with a fault to the process initiator.
* Invoke a human task to correct the issue.
* If the fault handler cannot resolve the issue, you might need to rollback and compensate.

### BPEL activities to propagate a fault

BPEL allows fault propagation using throw, rethrow and reply within a fault handler.

#### Throw

A throw activity indicates a problem that a business process flow cannot handle. It is used to throw an exception corresponding to an internal error condition. Use a throw activity within the flow of a business process or within a fault handler, to allow an outer fault handler to handle the fault. A throw activity can throw one of the standard BPEL faults or a custom fault.

#### Rethrow

When the current fault handler cannot handle the fault and wants to propagate it to an outer-scoped fault handler, use a rethrow activity. In the absence of a rethrow activity, a fault propagated to a higher level using a throw activity would be a new fault. When a rethrow activity is invoked, the fault is the same instance. The rethrow activity is available only within a fault handler because only an existing fault can be rethrown.

#### Reply with a fault

This construct allows the propagation of the fault to the client that initiated the process. A reply with fault activity can only return a fault defined on the interface the process is implementing. This is useful when the business process cannot properly respond to the caught fault, and the process initiator may be better equipped to respond; for example if the client passes an account number that is not found by the business process, the process should reply to the service call with an AccountNotFound Fault.

### Use a fault handler within a scope

If a fault is not handled, it creates a faulted state that migrates up through the application and can throw the entire process into a faulted state. To prevent this, contain the parts of the process that have the potential to receive faults within a scope. The catch activity works within a scope to catch faults and exceptions before they can throw the entire process into a faulted state.

You can use specific fault names in the catch activity to respond in a specific way to an

individual fault. To catch any faults that are not already handled by name-specific

catch activities, use the catchAll activity.

### Use compensation after undoing a series of operations

Compensation occurs when the BPEL process cannot complete a series of operations after some of them have already completed, and the BPEL process must backtrack and undo the previously completed transactions. For example, if a BPEL process is designed to book a rental car, a hotel, and a flight, it may book the car and the hotel and then be unable to book a flight for the right day. In this case, the BPEL flow performs compensation by going back and unbooking the car and the hotel.

You can invoke a compensation handler by using the compensate activity, which names the scope for which the compensation is to be performed (that is, the scope whose compensation handler is to be invoked). A compensation handler for a scope is available for invocation only when the scope completes normally. Invoking a compensation handler that has not been installed is equivalent to using the empty activity. This ensures that fault handlers do not have to rely on state to determine which nested scopes have completed successfully. The semantics of a process in which an installed compensation handler is invoked more than once are undefined.

If an invoke activity has a compensation handler defined inline, then the name of the activity is the name of the scope to be used in the compensate activity. The syntax is as follows:

<compensate scope="ncname"? standard-attributes>

standard-elements

</compensate>

### Use the terminate activity to stop a business process instance

The terminate activity immediately terminates the behavior of a business process instance within which the terminate activity is performed. All currently running activities must be terminated as soon as possible without any fault handling or compensation behavior. The terminate activity does not send any notifications of the status of a BPEL process.

The syntax for the terminate activity is as follows:

<terminate standard-attributes>

standard-elements

</terminate>

## Code Level Metrics

| Name | Description and How to Calculate | Recommended Threshold |
| --- | --- | --- |
| Minimize the Number of BPEL Variables | Use as few variables as possible, and minimize the size and the number of business objects used. | Less than x |
| Avoid BPEL Processes with High Depth of Code | Depth of Code is measured as the maximum number of nested control statements in an artifact. For example, an artifact that contains an IF statement which contains a While loop which itself contains another IF statement will have a Depth of Code of 3 (at least). | Less than 9 |
| Avoid Large Java Snippets | BPEL is an orchestration language. The services that are being orchestrated should be implemented outside of BPEL. Java should be used only for small interactions only. Long Java Snippets make BPEL more complex. Java Snippets should not have more than X lines of code. | Less than x |
| Avoid BPEL artifacts with low comments/code ratio | Maintainability of the code is facilitated if there is documentation in the code. This rule will ensure there are comments within the Artifact. | More than 5% |

# Industry Best Coding Practice

SOA and BPEL bring a range of new challenges as composite applications introduce greate rcomplexity and more dependencies.

## Avoid interruptible processes

Use interruptible processes, also known as macroflows or long-running processes, only when required. Whenever possible, use synchronous interactions for non-interruptible processes. A non-interruptible process is much more efficient than an interruptible process. If interruptible processes are required for some capabilities, separate the processes such that the most frequent scenarios can be executed in non-interruptible processes and exceptional cases are handled in interruptible processes.

## Avoid missing OTHERSWISE in SWITCH statements

 The OTHERWISE CASE is executed when none of the conditions being tested for in the CASE statement are met or executed. Having no OTHERWISE means that there is a strong assumption about the value of data. The cases that are "impossible" today are the ones most likely to be the causes of untraceable bugs in the future, when the impossible changes to the standard. Add the proper exception handling code in the OTHERWISE clause.

## Avoid using invoke without catch

The BPEL process has to perform the task efficiently and reliably enough that network interruptions or the unavailability of a web service or other resource doesn't break it down.

## Create scopes for each step of the flow in the process

Create scopes for each step of the flow in the process so as to make it modular.  This will help in creating local variables within that scope. Scopes provide a context for the execution and/or documentation of enclosed activities, and they can have variables that are visible and usable at and within the scope level. Scopes can have both default and defined fault and event handling logic, and they can be undone, if necessary. Undoing the work of a scope involves the concept of compensation. When designing BPEL processes, they should be organized into logical units of work that can be undone. Scopes can be used as a context to create variables, fault/compensation/event handlers and for organizational purposes. They are the basic building blocks used to assemble a BPEL process.

## Use short-running flows where possible

A business transaction that is represented through a long-running flow has a lifetime that can spanminutes, hours, days or even months, and is typically divided into several technical transactions. The state of such a process instance is persistedin a database between two transactions so that operating system resourcesare occupied only during an in-flight transaction.

The short-running flow is used when the corresponding businesstransaction is fully automated, completes within a short timeframe, and has no asynchronousrequest/response operations. The entire set of of flow activities runs within one singletechnical transaction, navigation is all done in memory, and the intermediate state is not saved toa database. Such short-running flows can run between five and fifty times faster than comparablelong-running flows and are recommended, where possible.

## **Use global variables when required**

This will help in maintainability. However, declaring many global variables needs to be avoided in a BPEL process, instead use scope or local variables. The variables defined at the process root are global variables, which have a global visibility throughout the entire process. The variables defined within a particular scope are visible only inside that scope and all nested scopes. These variables are called local variables. A variable defined for an inner scope element can hide an upper defined variable of the same name.

## Consider creating sequences instead of scopes for performance

Do not add too many scopes since it may affect the performance. Instead, consider creating sequences. A sequence is a structured activity which can contain other activities, all of which will be executed in a specifically defined order. The purpose of a sequence, therefore, is to define the execution order for a group of activities. BPEL sequences can contain other sequences and can be nested as deeply as you want. Sequences have all the standard attributes and elements and they must contain at least one or more activities.

## Use assign activities instead of transform activities

Use assign activities instead of transform activities wherever possible since it take less memory.

Assign activity provides a method for data manipulation, such as copying the contents of one variable to another. This activity can contain any number of elementary assignments. Transform activity enables to create a transformation that maps source elements to target elements (for example, incoming purchase order data into outgoing purchase order acknowledgment data).

## Use XSL to construct messages instead of multiple assign statements

Don’t use multiple assign statement to construct a message; use XSL to construct messages.

## Use XPath Expression constraints to check the data constraints

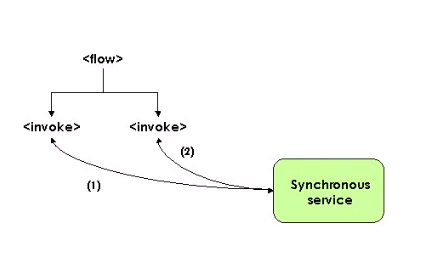
Don’t loop through the data to check data constraints; use XPath Expression constraints to check the data constraints. Use Fully Qualified XPath expressions.

## Initialize a BPEL variable before copying values to internal elements

Variables offer the possibility to store messages that hold the state of the process. Initialize a variable before copying values to internal elements to avoid “SelectionFailurefaults or "variable counter isn't properly initialized" on execution.

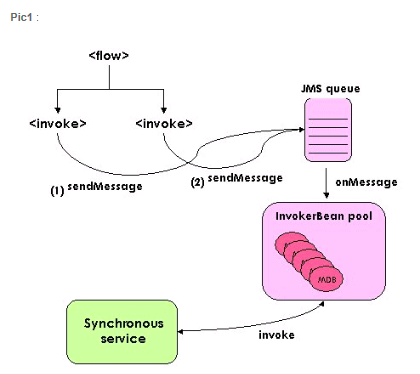
## Use <nonBlockingInvoke> in BPEL flow for performance improvements

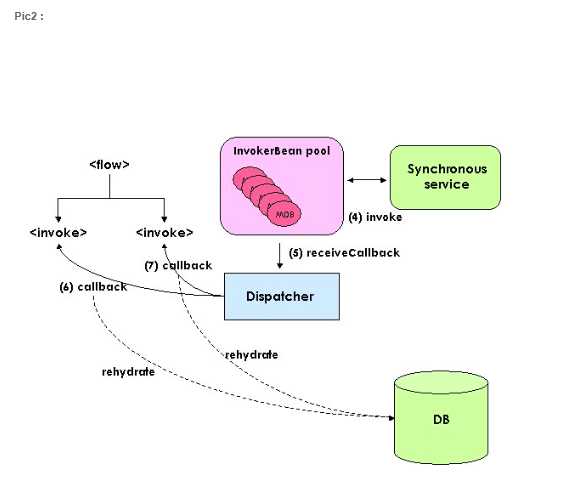
Normally when executing a synchronous (two-way) invoke activity, Oracle BPEL  
server waits for the response from the endpoint before executing the subsequent  
activity. This behavior can present problems when synchronous invoke activities are  
placed inside a flow activity, because Oracle BPEL Server executes the flow using  
pseudo-parallelism.



The second invoke activity is not executed until the response from the first invoke activity is received. If the number of branches in the flow is large, the delay before the final invoke activity is executed is the sum of all the preceding synchronous invoke activities.  
  
**Solution**

With a non-blocking invoke depicted in Pic 1 and Pic 2, the execution of the synchronous invoke activity is scheduled to be performed by a separate thread in the background. With this change, the initial execution time of the invoke activities is reduced.





The sequence of events in Pic 1 and Pic 2 for a non-blocking invoke call is as follows:

1. The first synchronous invoke activity sends a message to a JMS queue. This activity now waits for the asynchronous response and relinquishes control to the next activity.
2. The second synchronous invoke activity sends a message to a JMS queue.
3. Since there are no additional activities to execute, the instance now dehydrates.
4. The invoke activities cannot complete until they receive their callbacks.
5. Two InvokerBean instances pick up the messages in the JMS queue and execute  
   the synchronous invoke against the partner endpoint.
6. The response from the endpoint is cached and a callback message is scheduled  
   with the dispatcher module.
7. The first invoke activity receives its callback. The instance is rehydrated during this step.
8. The second invoke activity receives its callback. The instance is rehydrated during this step.

A performance test showed a 30 % improvement in comparison with not using nonBlockingInvoke.

## Keep the number of activities in BPEL as minimal as possible

Keep the number of activities in BPEL as minimal as possible; increasing the number of activities will decrease the performance of BPEL engine.

## Avoid unused partner links

## Avoid repetitive names for different activities

## Don’t use BPEL for intensive time scheduled activities

Extensive use of activities such as alarm and wait can lower system performance.

## Do not include any special characters in the project name

Special characters (such as periods) in the project name cause errors during compilation.

## Don’t use the pick based initiate pattern for interdependent operations

The pick activity waits for the occurrence of exactly one event from a set of events, then executes the activity associated with that event. After an event has been selected, the other events are no longer accepted by that pick. If a race condition occurs between multiple events, the choice of the event is implementation dependent.

## Don’t let a SOA composite instance grow exponentially

Define rules to keep the house clean, purge the instances at regular interval to obtain better performance from the BPEL engine and Enterprise Manager.