BEA WebLogic® Integration

Best Practices for WLI Application Life Cycle

Version: 10.2
Document Revised: March 2008

Contents

CHAPTER 1

Introduction

This document specifies the best practices, lessons learnt, and key considerations which help WebLogic Integration users such as application architects, developers, and operation staff to develop and run high quality WebLogic Integration (WLI) applications.

Note: It is recommended that you read the following documents before you use this document:

- Introduction to WebLogic Integration
- Guide to Building Business Processes
- Using the Worklist
- Using Integration Controls

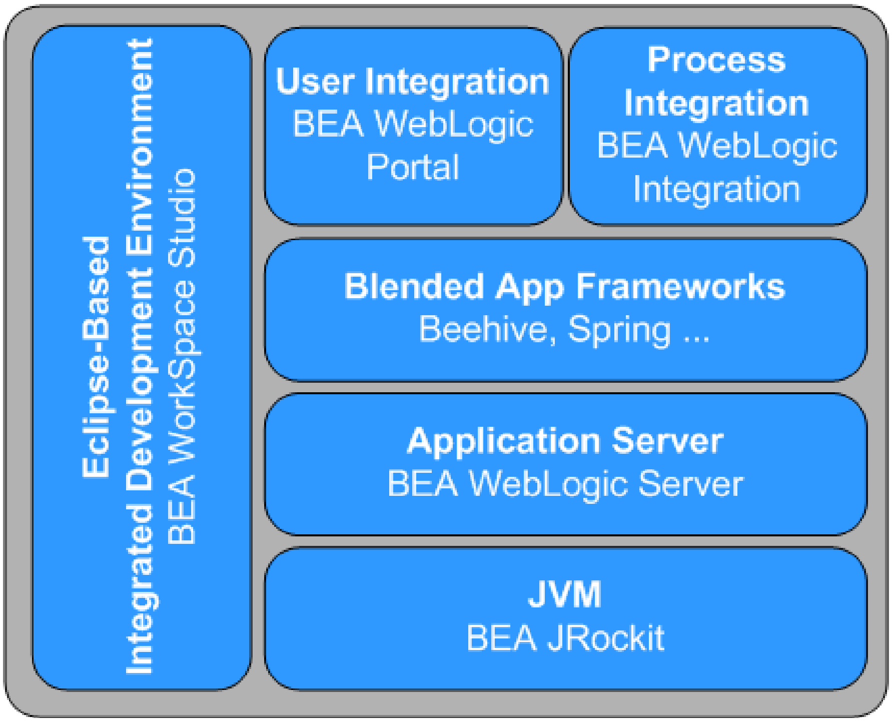
While reading this document, you may have to refer to relevant documents and information about WLI 10.2, available on the WebLogic Integration documentation page.

This document:

- Explains the basic Features of WLI
- Provides general best practices to help you with Understanding Requirements, Designing WLI Applications, and Modeling Business Processes and Services.
- Describes best practices with WLI for:
  - Composing and Developing WLI Applications
  - Deploying and Maintaining WLI Applications

WebLogic Integration is specially designed for enterprise integration and is a part of WebLogic Platform. BEA WebLogic Platform is an integrated platform for building, extending, integrating, deploying, and managing applications. Figure 1-1 shows the architecture of WebLogic platform.

Figure 1-1 Architecture of WebLogic Platform



Features of WLI

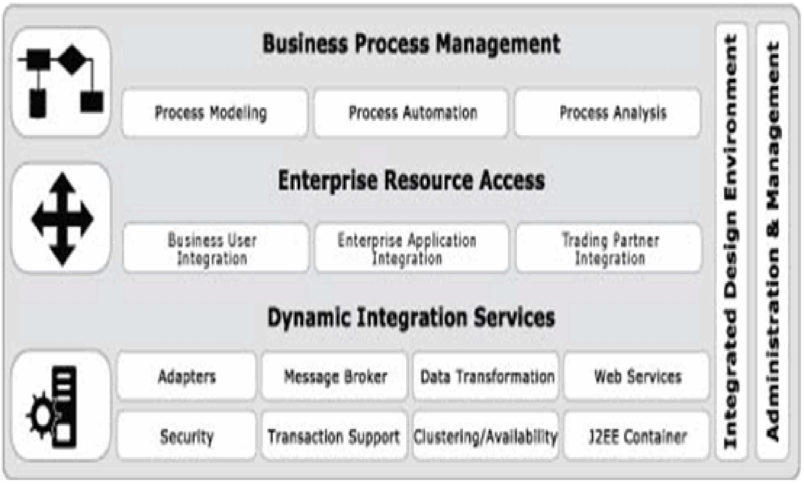
WebLogic Integration provides you with the capability to design, develop, deploy and run integration-centric applications. The main features or capabilities of WebLogic Integration are:

- Creates system (Java Process Definition) and human-centric (Task Plan) business process applications using an Eclipse-based unified IDE.
- Defines system-centric transactional processes that have fine-grained control over the process using JPD graphical editor.
- Defines human-centric, multi-step, task-oriented, and long running business processes using Task Plan graphical editor.

- Uses standard-based Beehive controls to enable easier access to enterprise resources such as EJBs, JMS and web services. In most WLI applications, JPDs, controls, and Task Plans work with each other in realizing overall business objectives. They can be easily integrated with each other and remain loosely coupled.
- Uses JPD controls and task plans that are property and annotation driven. Properties and annotations can be either static or dynamic. Dynamic properties can be changed during runtime which makes business processes that run with WLI applications agile and flexible.
- Provides extensive support of Business to Business (B2B) capabilities for inter-enterprise business processes using Trading Partner Management, Rosettanet, ebXML, and EDI.
- Creates loosely coupled, publish and subscribe architecture, style based process integration using event generators, message brokers, and event handlers.
- Provides extensive Enterprise Application Integration features such as adapters, and XQuery (2004) based data transformation. The large number of integration controls such as service and process controls make the task of integrating enterprise resources easier.
- Supports enterprise computing services support such as transaction management, clustering, security, and J2EE container services.

Figure 1-2 shows the functional overview of WLI.

Figure 1-2 Functional Overview of WLI



Business Process Management
Process Modeling
Process Automation
Process Analysis
Enterprise Resource Access
Business User Integration
Enterprise Application Integration
Trading Partner Integration
Dynamic Integration Services
Adapters
Message Broker
Data Transformation
Web Services
Security
Transaction Support
Clustering/Availability
J2EE Container
Integrated Design Environment
Administration & Management

For more information on the features and functions of WLI, see Introduction to WebLogic Integration.

CHAPTER 2

Understanding Requirements

The key to successful software development is that all stake holders develop a clear and uniform understanding of application requirements.

Software requirements can be broadly classified into two groups:

- Functional or problem domain requirements
- Non-functional or solution domain requirements

In a problem domain, the focus is on the functional or business requirements. It is recommended that you create a domain model of your functional requirements before you start thinking of the solution domain.

In a solution domain, we focus on how to deliver the solution for functional or business requirements.

Some of the important non-functional requirements of a WLI application are:

- Quality attributes such as security, high availability, scalability, performance, and reliability.
- User interface
- Integration
- Message format, transport, and protocol
- Data format and transformation
- Internationalization or i18n

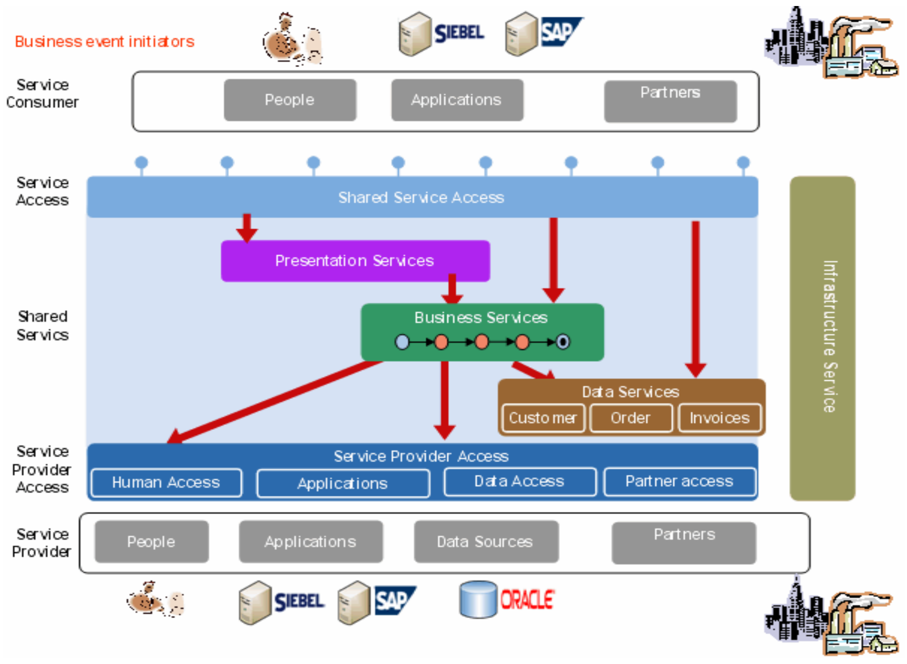
- Legal and compliance
- Runtime infrastructure
- Networking and communication
- Constraints such as the use of specific RDBMS System, protocols, and standards

For general guidelines on understanding requirements, see Appendix: Analyzing Use Cases and Requirements.

Designing WLI Applications

WLI is an enterprise class product, which can be used to implement process-driven Service Oriented Architecture (SOA) applications. Figure 3-1 shows the BEA reference architecture for service oriented applications.

Figure 3-1 BEA Reference Architecture for Service Oriented Applications



It is recommended that you use BEA reference architecture for service oriented applications for a WLI application.

The reference architecture is generic and can be implemented with WLI standalone or with a combination of several products. The main components of the reference architecture are as follows:

- **Business event initiators**: The entities that initiate business actions or events. Business initiators are either human, or based on systems and applications.
- **Service consumers or composite applications**: The applications that are developed while using WLI. They handle business actions or events initiated by business entities.
- **Shared service access layer**: Provides access to shared business services. It is based on Validate, Enrich, Transform, Route, and Operate or invoke (VETRO) patterns. This layer can be implemented using WLI or an Enterprise Service Bus (ESB) such as AquaLogic Service Bus.
- **Shared services layer**: The shared and reusable services that are used in service orchestration while creating business processes.

  The types of shared services are as follows:

  – Presentation services that present the data to the user.

  – Business services that represent core business capabilities. Business services can range from relatively simple to very complex. An example of a simple business service is credit card validation. An example of a complex business service is a cross-functional, inter-enterprise business process such as order fulfillment. Business services are task or activity-oriented. You can implement task-oriented services such as purchase order approval using task plans, and system-centric transactional services using JPDs. Most complex business services require both JPDs and task plans. JPDs and task plans can work independently and also be loosely coupled as required.

  – Data services that are entity services which provide access to enterprise data. Data services have a Validate user, Create, Retrieve, Update, and Delete (CRUD) interface and can be implemented using WLI components or special data service enabling products such as AquaLogic Data Services Platform.

  – Infrastructure services are non-functional services such as security, and audit.
- **Common service provider access layer**: Provides a common access to service providers. This layer can be implemented using WLI or an ESB. A service provider provides services and can be either human or based on systems and applications.

Modeling Business Processes and Services

The steps involved in modeling business processes and services are:

- Defining Business Processes
- Designing and Developing Services

Defining Business Processes

You can define a good business process by:

- Identifying Process Objectives and Goals
- Identifying Key Performance Indicators for the Process
- Identifying Process Actors or Participants
- Identifying Public and Private Processes
- Identifying Initiator and Participant Processes
- Keeping Processes Modular
- Enabling end-to-end, Cross-functional Processes
- Separating Production and Monitoring Processes

Identifying Process Objectives and Goals

It is important to understand the main objectives behind the decision to automate a business process. A business process is defined to:

- Improve operational efficiency and productivity
- Gain better information visibility and insight into business operations
- Improve customer service
- Improve human collaboration.

A few examples of process objectives are:

- An account receivable and payable process to improve visibility.
- A purchase order processing and fulfillment process to improve operational efficiency and productivity.

- A document approval process to improve human collaboration.
- A complaint resolution process in a call center for improving customer service and operational efficiency.

Identifying Key Performance Indicators for the Process

KPIs are the key indicators that provide the visibility for process performance. A few sample process KPIs are:

- The visibility of Days of Sales Outstanding (DSO) for an Accounts Receivable process.
- The time taken from order to delivery (efficiency), or the number of purchase orders completed in a day (productivity) for Purchase Order processing and fulfillment.

Identifying Process Actors or Participants

Identify process actors or participants before you define the process. In addition to human actors and applications, data sources and partners can also serve as actors or participants in a business process.

Identifying Public and Private Processes

Public and private processes have a special significance in the B2B domain. A public process is one that is visible to the external user. A private process is one that performs a task that is invisible to the external user within the application in the background. The private process is linked to the public process. When a public process is used as a front end for a private process, you can modify the private process without disturbing your public process clients.

It is a good practice to keep your public process modular. Public processes have higher requirements in terms of security, scalability and availability. Public processes always have special requirements for messaging, protocols, and standards, which need to be identified in advance. Figure 3-2 illustrates a public and private process pattern.

One way to integrate public and private processes is to use loosely coupled publish and subscribe architecture between these processes using the message broker component of WLI.

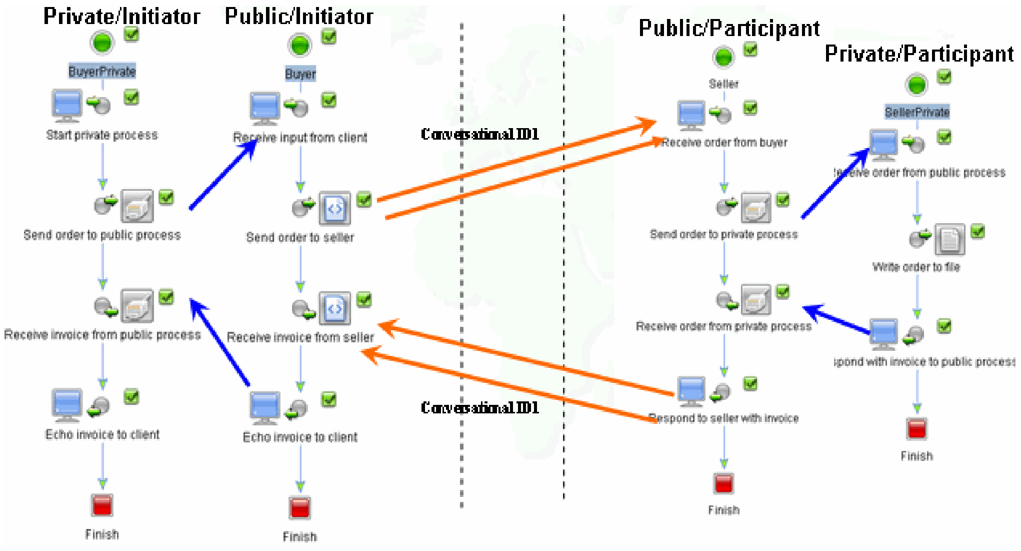
Another alternative to integrate public and private processes, is to have a process control between public and private processes, and use Java Web Services (JWS) as the front end for your public processes. Process Control (PC) is a better option if your public and private processes are in the same domain. See the Communication Between JPDs section for more details.

Identifying Initiator and Participant Processes

Many business processes, especially in the B2B domain, are conversational in nature. For example, a request quote service, where an agreement on the quoted price is achieved after several rounds of message exchange. An initiator uses an initiator process to initiate a business conversation. Participant processes respond to the request from an initiator process. An initiator process:

- Can have multiple conversations with a single participant process at the same time.
- Can also interact with multiple participant processes at the same time.
- Generates a unique conversation ID for each process. All the related participant processes, include this conversation ID in their response messages. Conversation IDs act as correlation identifiers across a conversation. Figure 3-2 shows a sample implementation of Public, Private, and Conversational patterns.

Figure 3-2 Public, Private, and Conversational Patterns



Keeping Processes Modular

It is a good practice to define small and modular processes. A modular process meets a specific business need. Modular process definition results in the reuse of business processes. It is easier to maintain many modular processes than a few long processes.

Enabling end-to-end, Cross-functional Processes

You can get the best results from Business Process Management (BPM) when you enable end-to-end cross-functional processes, to realize a specific business objective.

A few examples of end-to-end, cross-functional processes are as follows:

- Order to cash
- Request to quote
- Complaint to resolution

The ideal way to enable end-to-end processes is using loosely coupled integration between a number of modular business processes.

Separating Production and Monitoring Processes

It is always a good practice to separate your core production processes from your monitoring processes. Production processes have higher scalability, performance, and availability requirements than monitoring processes. An example of a core production process is the order fulfillment process. An example of monitoring processes are audit, compliance, and KPI calculation processes.

You can have loosely coupled integration between production and monitoring processes using publish and subscribe architecture with Message Broker. For more information, see Design Loosely Coupled Services.

Note: It is a good practice to capture monitoring data from a production process and publish it to specific channels, which the monitoring process can then subscribe to. Subsequently, a monitoring process can process the monitoring data as per business needs.

Designing and Developing Services

The steps that you need to follow when you design and develop services are:

- Service Classification
- Identify Services
- Define Service Contract
- Define Input and Output Service Messages
- Define Pre and Post Conditions for Services

- Decide the Service Calling Paradigm
- Decide the Service Granularity
- Define Quality of Service Requirements
- Design Service With a Service Proxy
- Design Reusable Services
- Design Loosely Coupled Services

Service Classification

Table 3-1 lists the services that can be broadly classified into categories.

Table 3-1 Service Classification

Category	Service Category as per Reference Architecture	Characteristic	Example
Task-oriented	Business services	Long running and human-centric	Document-oriented such as Purchase Order approval
Activity	Business services	System-centric	• Purchase Order fulfillment • Credit card verification service
Entity service	Data services	CRUD service for business entities	Retrieve customer address

Identify Services

The first step in service design is to identify the candidate services. You can identify business services based upon business events. Table 3-2 contains a few examples of business events and related services.

Table 3-2 Sample Business Events and Related Services

Business Events	Services
Purchase Order is received	Purchase Order processing service
Purchase Order is validated	Purchase Order fulfillment service

Table 3-2 Sample Business Events and Related Services

Invoice is received	Invoice processing service
Purchase Order amendment request is received	Purchase Order amendment service

If you identify a service with its related business event, you have event-driven services, which can be easily orchestrated into long running business processes.

Define Service Contract

Once you have identified a service, the next step is to define a service contract. The service contract specifies the functions that a service provides in a format that the service consumer easily understands. A service contract defines how a service can be used and specifies the Quality of Service (QoS) parameters for a service. However, a service contract does not contain any implementation level details.

Define Input and Output Service Messages

After defining the service contract, the next most important step is to define service messages. A service receives an input message from service consumer and may or may not return a message to the consumer. Define an appropriate service message schema for each service message.

Define Pre and Post Conditions for Services

The next step is to define pre and post conditions for each service. Pre-condition represents the condition which should be satisfied, before a service is invoked. The service consumer has to ensure that the service is invoked, only when the pre-condition is satisfied. The service provider has to ensure that post conditions are satisfied after the service invocation is completed. If you develop your services using this method, you can perform better exception management for services.

Decide the Service Calling Paradigm

You need to decide whether the service is invoked in a synchronous or asynchronous mode. Small and data oriented services such as obtaining a customer address, can be designed in a synchronous mode. Long running services that require state management, should be designed as asynchronous services. Asynchronous services are more loosely coupled and scalable compared to synchronous services.

Decide the Service Granularity

Services should be more coarse-grained in comparison to API calls. You should decide on the actual granularity based upon your specific situation. A service call involves a round trip on the network. For performance reasons, you should try to minimize round trips on the network. In case you have too many fine-grained services, it is good practice to create a coarse-grained service by performing a light weight orchestration of the fine-grained services. Subsequently, you can expose the coarse-grained services to the user.

Define Quality of Service Requirements

Services are required to meet minimum quality requirements in terms of availability and performance. Service providers are required to have a Service Level Agreement (SLA) with service consumers. You should understand and specify QoS requirements well in advance.

One way to improve your QoS is to design your service as idempotent. Idempotent services do not change the state of the system, even if the service is invoked by the same input message several times. With idempotent services, you can easily use standards such as web service reliable messaging, which can re-transmit a message in case of failure. All read only services are idempotent.

However, even write services can be designed as idempotent. This design ensures that your services are more reliable and improves QoS. For example, when a service receives an invoice from a supplier, the invoice amount should be added to the total accounts receivable amount. In such a scenario, you can first assign a unique number to the invoice in the database and then increase the account payable amount. If the same message is sent again, the uniqueness constraint of the invoice number throws a database exception and discards the duplicate message.

Design Service With a Service Proxy

If you design your service with a proxy service as the front end, you can obtain several advantages using a proxy service as your main service. A proxy service can be used for authentication, enrichment, transformation, and versioning. The use of proxy services establishes a loose coupling between service providers and consumers.

Design Reusable Services

You should identify business functions that are reusable across different domains or departments. Such functions should be prime candidates for being exposed as standard based services.

Design Loosely Coupled Services

Services should be designed as loosely coupled. There should be minimum coupling between service consumers and service providers. The best practices to design loosely coupled services are as follows:

- Message format and protocol coupling: Use a proxy as a front end for your service to reduce the message format and protocol coupling.
- Time coupling: Design asynchronous services and reduce time coupling. You can also reduce time coupling by making your services highly available (7/24/365).
- Type coupling: Design document-oriented services to reduce type coupling. Document-oriented services are more loosely coupled than Remote Procedure Call (RPC) styled services. In an RPC styled service, the client makes a method call to a service. The client has to know the method name and data type of the method input as well as the output parameters. This leads to tight coupling between the service consumer and provider.

  In the case of document-oriented services, the service consumer interacts with the service by sending documents that are meant to be processed as complete entities. These documents are written in XML, and defined in a commonly agreed upon schema between the service provider and consumer. For example, a document-oriented service is a purchase order processing service which receives a purchase order (in XML form), processes it, and returns a purchase order confirmation (XML document) to the client. JPDs are designed to be document-oriented. A JPD uses a document and processes it as an entity.
- Location coupling: You can reduce location coupling by making your service available. You can publish your service WSDL and meta data to a registry or repository. The service consumer can then discover the service from the registry and then invoke the service as required.
- Version Coupling: You can reduce version coupling by making your service backward compatible. Existing service consumers should not be affected if there is a new version of the service. If you use a proxy service as a front end, it receives the initial request and then directs it to the appropriate version of the service.

Composing and Developing WLI Applications

There are several best practices for composing and developing WLI applications. They are described in the following sections:

- Naming Standard for WLI Application Artifacts and Variables
- Modular JPD Design
- Modular XML Document
- Parallel Node
- JPD Exceptions
- Event Handlers for Process Events
- JPD Transactions and Compensation Management
- JPD State Management
- Singleton JPD
- Race Condition With Dynamic Subscription
- Dead Letter Channel Subscription
- High Quality of Service JPD
- SLA Threshold for JPDs
- Monitor JPDs

- Security Policy for JPDs
- Interoperable JPDs
- Communication Between JPDs
- Using Controls
- Data Transformation
- Canonical Data Model
- Runtime Selection of a Transformation
- Developing a Task Plan

For more information on this section, see Guide to Building Business Processes and Using WebLogic Integration Controls.

Naming Standard for WLI Application Artifacts and Variables

When you develop WLI applications, you should follow a uniform and consistent naming standard for various WLI artifacts such as JPDs, task plans, controls, projects, and files. All controls, processes, methods, variables, and other WLI object names should follow standard object oriented and Java naming standards.

In WLI version 9.2, JPD, control, and data-transformation files have common .java extensions. You should add a suffix to each artifact so that they can be easily identified. Table 4-1 contains an example of naming controls, the related suffix for each artifact, and their locations.

Table 4-1 Example of Naming Controls/Artifacts

Type of Artifact	Suffix	Example	Folder/Package
JPD projects	EAR, JPD	<*Name*>earjpd	
	Web, JPD	<*Name*>webjpd	
	Util, JPD	<*Name*>utiljpd	
Task plan projects	EAR, JPD	<*Name*>eartp	EarContent/<folder>
	Web, JPD	<*Name*>webtp	
	Util, JPD	<*Name*>utiltp	

Table 4-1 Example of Naming Controls/Artifacts (Continued)

Type of Artifact	Suffix	Example	Folder/Package
JPD	process	<*name*>process.java	src/processes package
Data transformation	DTF	<*name*>dtf.java	src/processes package
Custom control	None	None	
Database control	DBC	<*name*>DBC.java	src/controls package
Web service	WSC	<*name*>WSC.java	src/processes package
EJB control	EJBC	<*name*>EJBC.java	src/controls package
JMS	JMSC	<*name*>JMSC.java	
E-mail	EmailC	<*name*>EmailC.java	
File	FileC	<*name*>FileC.java	
HTTP	HTTPC	<*name*>HTTPC.java	
MQ Series control	MQC	<*name*>MQC.java	src/controls package
Process control	PControl	<*name*>PControl.java	src/controls package
Task control	TASKC	<*name*>TASKC.java	src/controls package
Channels/Message Broker	Channel	<*name*>Channel.java	src/processes package
Event generators	EG	<*name*>EG	
XML Schema	XSD	<*name*>.xsd	src/schemas package
WSDL	WSDL	<*name*>.wsdl	src/schemas package
XQ	XQ	<*name*>.xq	src package

Modular JPD Design

If you have to define a large process, it is always a good practice to make it modular. A modular JPD meets a specific business objective. A few examples of modular JPDs are:

- Process purchase order

- Fulfill purchase order

You can divide a large JPD into number of smaller JPDs or sub-processes. Sub-processes can be invoked using a central or main JPD. You can use process control to communicate between a JPD and a sub-process (another JPD). You can also loosely couple two JPDs using Message Broker or publish and subscribe architecture. See Communication Between JPDs and Core Implementation Patterns for a JPD for more details.

Modular XML Document

Every JPD has a start node and uses a XML document to start. Ensure that the XML document is modular and small for better performance. It is a good practice to have separate XSDs for each type of document. For example, you should have a separate XSDs for a Purchase Order and Invoice, instead of one single large schema for both the Purchase Order and the Invoice.

The modularity of a JPD and its associated document is a relative concept. If a JPD is too modular, it may be counter productive. You should take a balanced decision on modularity depending upon the specific scenario.

Parallel Node

The JPD provides a parallel node. Before you use this node, it is recommended that you understand how this node works. Parallel nodes can be used for managing multiple tasks which are related but not dependent. Parallel nodes are meant for business level parallelism. The actual execution does not take place in parallel.

At any given time, execution can start in one branch without any execution in other branches. For example, the branch executing a task may request for a quote from a partner and wait for a response. While this branch is waiting for a response, execution may start in another branch thus achieving business level parallelism.

There are two forms of parallel nodes – AND and OR mode. In the AND mode, all parallel paths should complete their execution, before the business process can proceed further. In the OR mode, the path that completes execution first is the winner. The processing of other path is stopped and the business process proceeds further.

The branches of a parallel node are isolated by transaction (default behavior). You can override this behavior to improve the performance of parallel nodes. Set the **`continueTransaction`** property to **`true`** in the source for each parallel element as follows:

```
<parallel continueTransaction="true">
```

JPD Exceptions

You can define exceptions and exception handlers in JPDs at nodes, groups, or process-levels. Exception handlers are executed in the following order:

1. Nodes
2. Groups
3. Processes

It is a good practice at the process-level to have a **`catch all`** process-level exception handler.

Exceptions that are not handled can cause process failure. You can set a **`freezeOnfailure`** property to avoid this scenario. If this property is set, any exceptions that are not handled cause the process to freeze. The process then rolls back to the last committed point and the administrator can restart the process from the last committed state.

Table 4-2 contains a list of high-level design guidelines which you can follow while working on a JPD exception.

Table 4-2 High-Level Design Guidelines

High-Level Design Guideline	Description
Asynchronous two-way processes should establish a way to pass exceptions or errors back to the caller process, with a separate client response or publication node in the exception handler.	When an error or an exception occurs, it is possible to catch an exception within the process. However, asynchronous processes, by nature cannot immediately respond to the exception. However, the caller should be informed that the process did not complete successfully. Your callback process should have a success or failure path and the result in both cases should be communicated to the caller.
All synchronous stateless processes must throw an exception back to the caller.	If the caller is blocking, it must be notified of the failure, as it is the only component that understands what should be done next.
Process designers should define a process-level exception handler at the start of every process definition. This handler acts as a global exception handler and catches any undefined exceptions.	This is a catch-all process-level exception handler. Exceptions that are not handled cause the process to fail and no recovery is possible from this state. When you define a global exception handler, you can ensure that the process never goes to an aborted state.

Table 4-2 High-Level Design Guidelines (Continued)

High-Level Design Guideline	Description
Set the **`freezeOnfailure`** property to true	If you set the **`freezeOnfailure`** property on the process, the process is rolled back to last commit state and persisted. The administrator can then fix the problem and re-activate the process.
Process designers should be especially careful about handling exceptions when processes call sub processes synchronously through the process control.	Exceptions that are not handled in the called sub process can set transactions to roll back. In this case, both the sub-process and caller process can be rolled back. You can define an appropriate exception handler in the sub process to avoid the roll back of the caller process.

Event Handlers for Process Events

An Event Choice node group in a business process represents a point at which the business process waits to receive one or more events. The first node on each branch of an event choice node group handles the receipt of an event. You can design the other nodes within an event choice node group to handle the incoming events. Each branch works as event handler for the event or message.

You must create an event choice node at a point in a business process at which it waits to receive multiple events. If you use an Event Choice node to start a business process, it can contain Client Request, Client Request with Return, and Subscription nodes. An Event Choice node which is located at a point other than the start node in a business process can contain Client Request nodes and Control Receive nodes. You can also add a timer branch to your Event Choice node to start that branch after a specified time out.

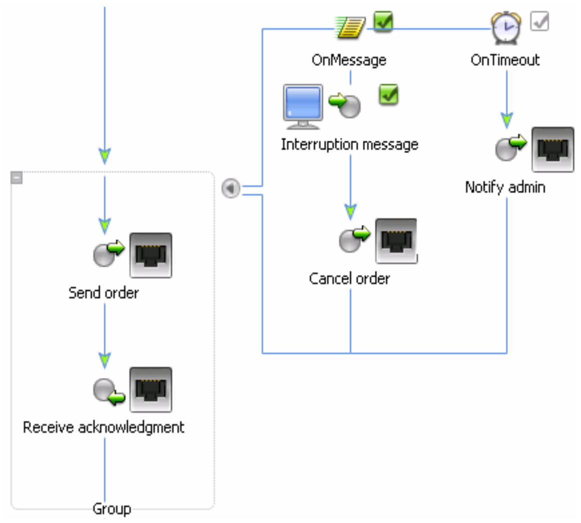
The events can include:

- Receive messages from clients and invokes **`OnMessage`** event handler.
- Receive messages from resources, such as a database, or a JMS queue, or an EJB (A business process interacts with resources using controls.) and invokes **`OnMessage`** event handler
- A Timer event. The timer starts when the execution of the business process reaches the Event Choice node and pauses to wait for an event .

`OnMessage` and `OnTimeout` events handlers allow interruption of process by outside events. Events handlers allow outside events to interrupt the process using `OnMessage` and `OnTimeout` event handlers. `OnMessage` event handlers can accept client requests.

Figure 4-1 shows an example of how an `OnMessage` and `OnTimeout` event handler is used.

Figure 4-1 Example of OnMessage and OnTimeout Event Handler



JPD Transactions and Compensation Management

The following sections describe the best practices for various JPD transactions and compensation management:

- Transaction Boundaries
- Transactions for Synchronous and Asynchronous Processes
- Transactions and Controls
- JPD Versioning
- Using Dynamic Properties and Annotation for Controls
- Buffering Service Control Methods During Asynchronous Calls
- Using Control Factory to Manage Multiple Instances of a Control

Transaction Boundaries

Processes in WLI are transactional in nature. Every step of a process is executed within the context of a Java Transaction API (JTA) transaction. When you are building a process, implicit transaction boundaries are formed based upon the location of blocking elements such as a Control Receive or a Client Send. As you add process nodes, the transaction boundaries within a process keep changing.

You can also create explicit transaction boundaries. To do this, select contiguous nodes and declare them in a separate transaction to distinguish between them and the implicit nodes that the application creates. The transaction may also contain resources accessed by a process, depending on the nature of the resource and the control that provides the access.

Transactions for Synchronous and Asynchronous Processes

A stateless process is executed either in a client transaction or when a new transaction is started. Using JPD proxy, a Java client can invoke a JPD over RMI. In this scenario, the client transactions are propagated to the JPD. The caller transaction is not propagated when a JPD is invoked as a web service. The caller transaction is not propagated to a JPD for asynchronous processes as well.

Transactions and Controls

Transaction controls are of three types:

- Transactional and XA Compliant
- Transactional and not XA Compliant Controls
- Non-transactional Controls

Transactional and XA Compliant

If all controls used within a process are transactional and XA compliant, then the transaction of the process can be used to commit or terminate the underlying transaction branches. The process needs exception handlers to catch any issues and make the necessary transaction decisions. The developer needs to be aware where the transaction was started as any abort or rollback takes the control back to the starting point and this may not be within the process where the exception occurred.

Transactional and not XA Compliant Controls

XA is a protocol used to manage distributed transactions. WLI extends XA to allow non-XA resources to participate in distributed transactions, with the limitation that in a given transaction, only one transactional resource can be non-XA compliant. Therefore, if more than one transactional non-XA resource needs to be accessed in a process, then the access to these resources should be encapsulated in separate JPD sub-processes, which should be called asynchronously by the original process. Asynchronous invocation is necessary because synchronously called subprocesses run in the same transaction as the calling process. See Using Integration Controls for more details.

Non-transactional Controls

Non-transactional controls such as email controls do not support transactions. In the case of non-transactional controls, you need to have a strategy for exception handling. Use automatic rollback where controls are transactional. Use a caught exception for non-transactional controls to handle the error based on the business problem.

In addition to providing support for Atomicity Consistency Isolation and Durability (ACID) and XA transaction, JPDs also provide support for compensation. You should ensure that the transaction block for which you want to provide compensation, is not marked for **`rollback only`**. Define an exception handler path for the transaction block and enable the **`execute on rollback`** exception handler property. In such a situation, when a transaction fails, the exception handler path is executed first. You can define your undo or compensation logic in such a path. Figure 4-2 shows an example of non-transactional controls.

Figure 4-2 Non-transactional Controls

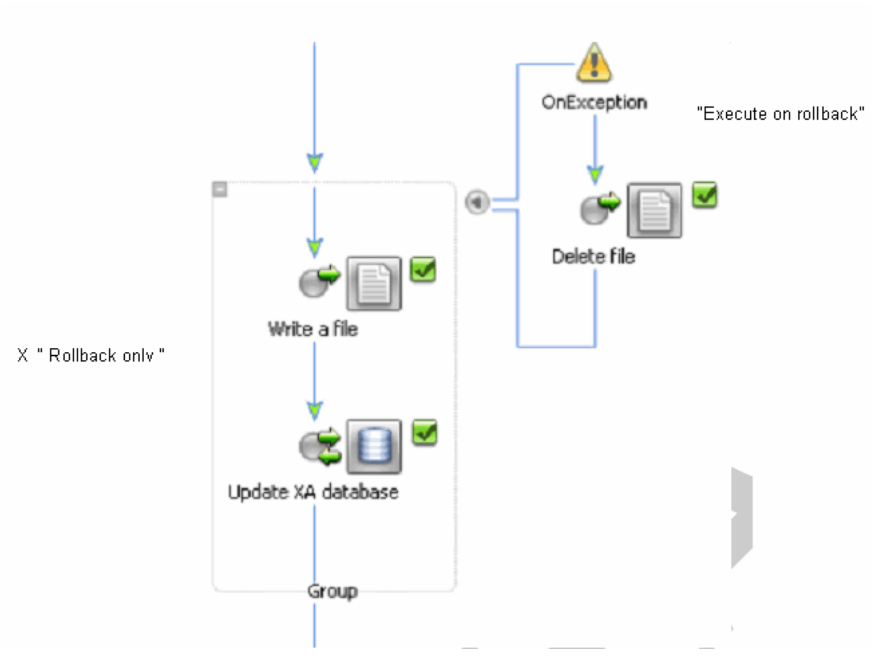


Table 4-3 contains a list of high-level guidelines to decide the transaction characteristic for your JPD.

Table 4-3 High-Level Design Guidelines for JPD Transaction Characteristics

High-Level Design Guidelines	Description
Implement the automatic execution of compensating transactions only with extreme caution.	At a design level it seems very simple to implement the reverse. The problems arise when the compensation fails, the question being the recovery from the failure. The solution to this situation is often unclear. The default design rule should be to raise an alert and manually decide on a solution.

Table 4-3 High-Level Design Guidelines for JPD Transaction Characteristics (Continued)

High-Level Design Guidelines	Description
A process needs to have exception handlers in place to find any issues and make the necessary transaction decisions.	You must not leave anything to default, always identify the exceptions and decide on a solution. This is applicable to transactional and non-transactional resources.
When more than one transactional non-XA resource has to be accessed in a process, the access to these resources should be encapsulated in separate JPD sub-processes.	The original process should call the sub-processes asynchronously. In this way, the sub-process runs in a separate transaction and is able to access the non-transactional resource.

JPD State Management

A stateless JPD is a process executed in memory only. Its state does not persist. All stateless processes are compiled into a stateless session bean. Stateless processes are intended to support business scenarios that involve short-running logic and have high performance requirements.

As the JPD does not persist its state to a database, it is optimized for lower-latency and higher-performance execution.

Table 4-4 contains a list of high-level guidelines that you should follow, when working on stateless processes.

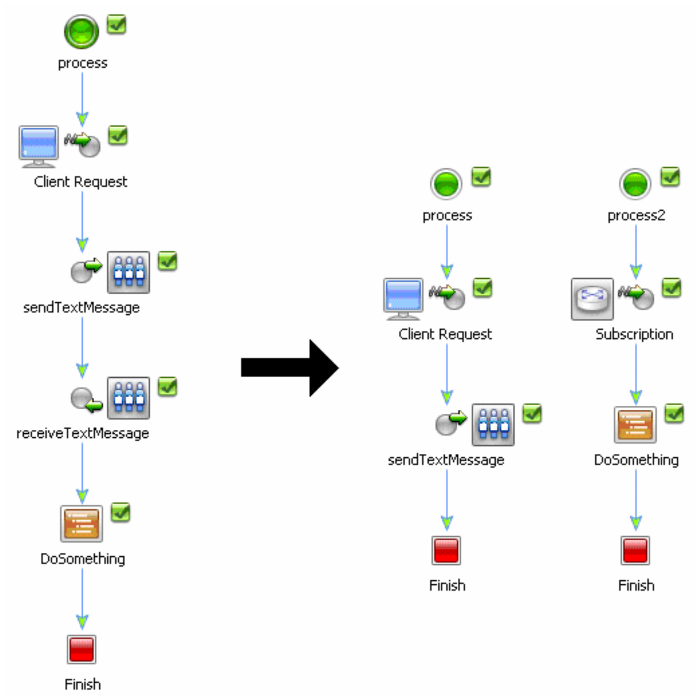
Table 4-4 High-Level Guidelines for Stateless Processes

High-Level Design Guidelines	Description
Do not use default values in global variables, unless the variable is either declared static or final. Initialize all global variables before use.	Stateless processes are implemented as stateless session beans. After a process is complete, subsequent process instances reuse the same stateless session bean instances, and therefore inherit the last known value of the global variables.
Set the **on sync failure** property on the process to **re-throw** for synchronously called processes where the requestor needs to handle transaction demarcation.	This property only applies to your process if it is configured to be a synchronous sub-process; it is ignored for any other business processes. If a synchronous sub-process fails, the default behavior is to mark it as **`rollback`**, which causes both the sub-process and the parent process to rollback. However, if the **`on sync failure`** property is set to **`re-throw`**, only the sub-process is rolled back.

A stateful process is a process that runs within the scope of more than one transaction. The process persists its state in the database. The state of the JPD survives even if the server crashes. The stateful JPD process is compiled into an entity bean. Stateful processes are intended to support business scenarios that involve complex, and long-running logic.

Stateful processes, in general, are slower than stateless processes. Use stateless processes, especially in scenarios where a state does not need to persist. In certain situations, you can split a stateful process into several stateless processes. Figure 4-3 shows how you can split a stateful process into a stateless process.

Figure 4-3 Splitting a Stateful Process Into a Stateless Processes



JPD Versioning

WLI has a version feature that helps you change your business process without interrupting any instances of the process that are currently running. When you create a version of a business process, you are actually creating a child version of a business process that shares the same public interface as the parent business process. At runtime, the version of the process that is marked active, is the process that external clients access using the public URI. Through the regular development cycle, new process versions are deployed with new versions of the application.

When the new version of a JPD is deployed, the existing instances run to completion on the same version of the JPD that they started with. You can version business processes, but not the individual controls associated with that process, or other business process related components such as schemas and transformations. When you version a business process, you must also version the sub-processes of that process, as they are not assigned a version automatically along with their parent process.

Table 4-5 contains a list of high-level guidelines that you can follow to set versions for JPDs.

Table 4-5 High-Level Guidelines to Version JPDs

High-Level Design Guidelines	Description
If you have to deploy a new version of a business process, you must set a version for your existing process before deploying the new version.	If you do not follow that process, you must let non-versioned instances run to completion before deploying the new versioned process.
Set the process version **strategy** according to the parent-child relationship of the business process.	This describes how you can invoke sub-processes when different versions of the parent process exist. From the strategy drop-down menu: • Select **loosely-coupled** if you want the sub-process version to be set when the sub-process is invoked. • Select **tightly-coupled** if you want the sub-process version to be set at the time the parent process is invoked.

Singleton JPD

A JPD can subscribe to a message broker channel in two ways: Static or Dynamic.

If a process subscribes to a channel at the start node, this is called a static subscription. The subscription is known at the time the application is compiled and remains through the life time of the application.

When a process subscribes to a message broker channel during its flow using the message broker control, this is known as a dynamic subscription. The subscription starts and ends at runtime. A static subscription to a message broker channel can be specified as suppressible if you set the suppressible attribute for the subscription in the business process. The accepted values for the suppressible attribute are true and false (false is the default value).

A singleton JPD has only one instance of the JPD running at any time. The singleton JPD can be re-run only after the previous instance is complete. The messages in the queue that are received when one instance of the singleton JPD is running are rejected.

To create a singleton JPD, first define a JPD with a static subscription and set the suppressible attribute to True. In this situation, the first message on the channel invokes the JPD and creates an instance of the JPD. Subsequent messages on the static channel do not lead to the creation of a new JPD instance. Singleton JPDs created in this way can continue to receive messages using dynamic subscription.

Race Condition With Dynamic Subscription

A race condition is possible when you use the message broker with a dynamic subscription. A message is lost if it is sent before the subscription is complete. This problem is evident when processes start waiting indefinitely for a response, coupled with messages appearing in the dead-letter channel.

If the request message is non-transactional, for example, in the case of a web service call, the message is sent immediately, whereas the subscription to the response occurs only when the transaction is committed. If the subscription appears before the message is sent in the process flow, it might occur later. In this case, ensure that the transaction is committed (for example, add an empty explicit transaction) after the subscription node and before the request message is sent.

Dead Letter Channel Subscription

If a message is sent to a channel without a subscriber after the filtering process is complete, the message goes to the dead-letter channel. Subscribe to the dead-letter channel to check if messages are published there, because in most cases, this is not a desired behavior.

High Quality of Service JPD

You need to take special steps to implement **`at-least-once`** or **`once-only`** Quality of Service. If these steps are not taken, the default is the **`at-least-once`** Quality of Service. The steps for the different process types are as follows:

- Invoking a Synchronous JPD: The quality of service is the responsibility of the caller.
- Invoking an Asynchronous JPD with JMS: When you use JMS as a calling mechanism to a process, the steps to achieve high Quality of Service are:
  – Configure re-delivery limit and re-delivery delay override for the associated JMS queue

  - JMS queues persist
  - JMS connection factories are transactional
  - JMS queue has an error queue configured
  - Any exception that was not handled must be resent to the caller using the call back process. This step ensures that the message is delivered to the process. If an exception is not handled, the message is stored in the error queue and can be recovered.

Note: You cannot ensure the successful completion of every computer program, but a high Quality of Service means that the error is always recoverable.

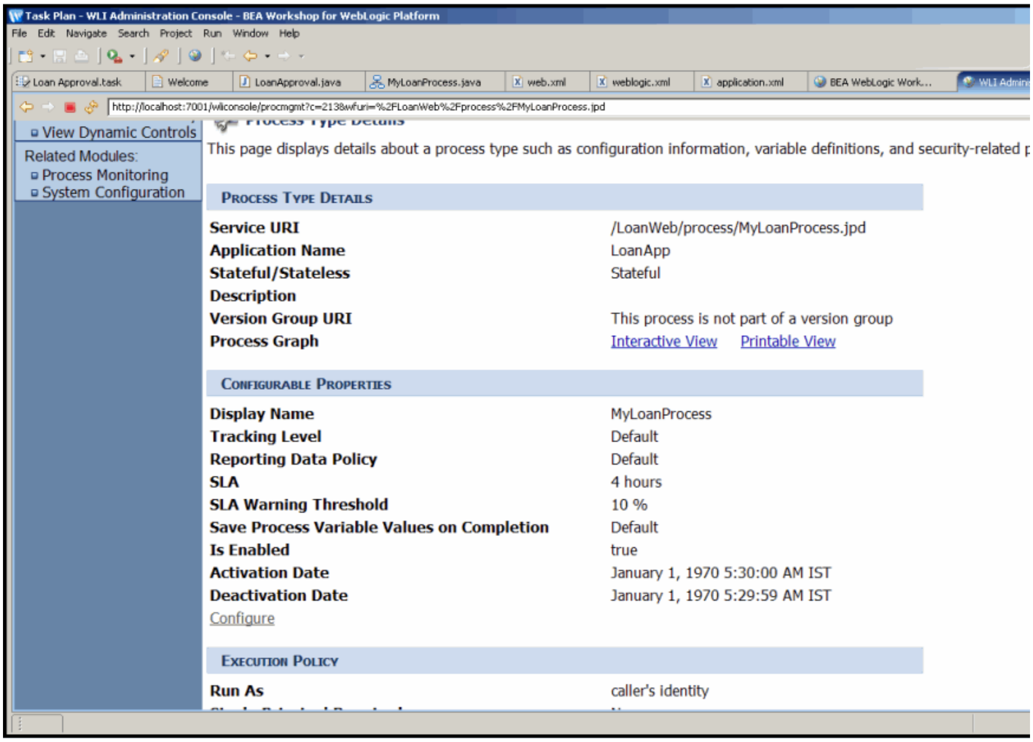
- The **`once-only`** Quality of Service cannot be implemented for event generator synchronous services such as files and e-mail that have source events that are not transactional.

SLA Threshold for JPDs

A Service Level Agreement (SLA) specifies the performance target for a JPD. An internal or external commitment shows that a JPD is executed within a specified period of time. To help you achieve the SLA for a process, the WLI Administration Console allows you to set the following thresholds (Figure 4-4):

- SLA threshold, that represents the commitment applicable to the process type. For example, number of seconds, minutes, hours, or days.
- SLA warning threshold, which is a percentage of the total SLA.

Figure 4-4 SLA Threshold Details


Task Plan - WLI Administration Console - BEA Workshop for WebLogic Platform
File Edit Navigate Search Project Run Window Help
Loan Approval.task
Welcome
LoanApproval.java
MyLoanProcess.java
web.xml
weblogic.xml
application.xml
BEA WebLogic Work...
WLI Admin
http://localhost:7001/wliconsole/procmgmt?c=2138wfuri=%2FLoanWeb%2Fprocess%2FMyLoanProcess.jpd
View Dynamic Controls
Related Modules:
Process Monitoring
System Configuration

This page displays details about a process type such as configuration information, variable definitions, and security-related

Process Type Details

Service URI	/LoanWeb/process/MyLoanProcess.jpd
Application Name	LoanApp
Stateful/Stateless	Stateful
Description	
Version Group URI	This process is not part of a version group
Process Graph	Interactive View Printable View

Configurable Properties

Display Name	MyLoanProcess
Tracking Level	Default
Reporting Data Policy	Default
SLA	4 hours
SLA Warning Threshold	10 %
Save Process Variable Values on Completion	Default
Is Enabled	true
Activation Date	January 1, 1970 5:30:00 AM IST
Deactivation Date	January 1, 1970 5:29:59 AM IST

Configure

Execution Policy

Run As	caller's identity

The process status that is relative to these thresholds is tracked for each process instance as follows:

When the elapsed time for a process instance reaches the warning threshold, a warning is displayed on the Process Instance Summary and Detail pages of the WLI Administration Console. The amount of time remaining until the SLA threshold is reached is also displayed. When the elapsed time exceeds the SLA, a red flag is displayed on the WLI Administration Console. The time limit by which the SLA threshold has been exceeded is also displayed. However, there is no event fired and you do not receive a separate notification for the SLA exceeding the threshold other than the warning on the WLI Administration Console.

This ability to set SLA thresholds allows you to easily identify processes that do not execute within the target time frame. You can then make the required changes to meet agreements between suppliers and customers, or to achieve your own performance goals.

Monitor JPDs

You can track a process at various levels. The system contains a default tracking level and then each process can override this. The WLI Administration Console and underlying MBeans provide a monitoring interface to running instances and their variable values. Although variable values cannot be changed, it is possible to query the process using the instance ID or process label. You can set the process label in the instance by calling the JPD context setProcessLabel. Table 4-6 contains a list of high-level design guidelines to monitor JPDs.

Table 4-6 High-Level Design Guidelines for Monitoring JPDs

High-Level Design Guidelines	Description
Use tracking data only for operational support, and not for business or audit logging.	You need to purge tracking data regularly. Tracking can be set on and off dynamically for each process at runtime using the WLI Administration Console.
Disable process tracking for processes that require high performance.	Process tracking involves information that is written into the database several times. For performance reasons, tracking must be eliminated.
WLI system data should only be used for support and not business level audit.	The WLI system data should not be used for business level audit. A designed audit or logging framework must be implemented outside the WLI system data capture area.
Log every InstanceID.	You must record the InstanceID for any error, log, and audit message.
Schedule regular archiving of WLI data.	Archiver is a process that runs a Select query on the database, so it must be scheduled to run regularly on small amounts of data. Avoid scheduling the archiving process at peak hours.
Set the process label to relevant query values.	The process label is a preferred query string to design the values, rather than depending on the person who implements the values. For example, a support person finds an order number easily if it is represented on the process label.

Security Policy for JPDs

You can define the security policy for a JPD. The security policy controls the identity that the JPD uses to access external or backend systems. It allows the administrator to configure whether a JPD accesses an external system as the invoking application, or as an application that calls into the process later. For example, if a process subscribes to a channel and then waits for a client request, the administrator can set the execution policy and use the identity from the client request while accessing backend resources.

The JPD security policy has four main components:

- Process Execution Policy: The execution policy specifies whether the operations in the process are run with start user or the caller’s identity.
  - If the administrator specifies start user, each operation assumes the identity of the user that started the process.
  - If the administrator specifies caller’s ID, the operation after the call assumes the identity of the calling component.

  The policy also ascertains if a single principal is required or not. If a single principal is required, all incoming client requests must come from the same user.
- Process authorization policy: The administrator can configure the roles authorized to invoke the process method or client request. All methods in the process inherit the roles specified in the process authorization policy. If the process authorization policy is not defined, everyone is authorized.
- Method authorization policy: The administrator can configure the roles authorized to invoke the process methods or client requests. All methods inherit the roles specified in the process authorization policy. You can also add roles to the authorization policy for a method.
- Callback authorization policy: The administrator can configure the roles authorized to invoke the process callback. If the callback authorization policy is not defined, everyone is authorized to make callbacks.

Interoperable JPDs

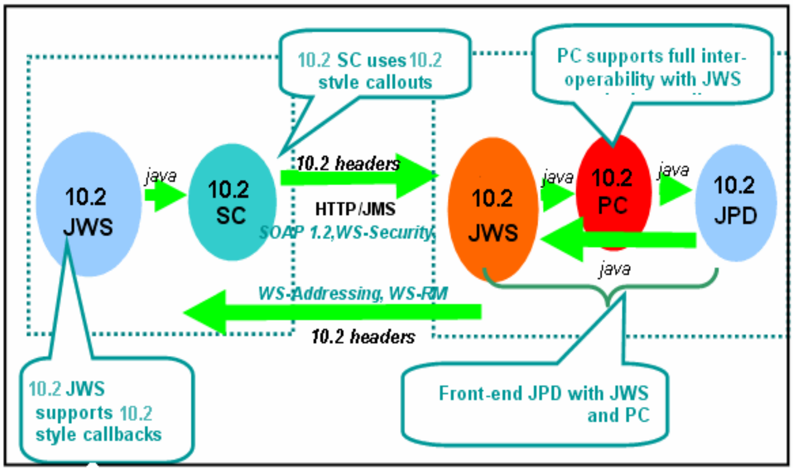
You can expose a JPD as a Java Web Service (JWS). There are limitations to developing interoperable JPDs, which include:

- JWS supports web service standards such as WS-Security and WS-Addressing.

- WebLogic Server 9.2 JWS does not support WLI 8.1 JPD style callbacks.
- WLI 9.2 JPDs support only WLI 8.1 conversations.
- WLS 9.2 Service Control (SC) can invoke WLI 8.1 JPDs.
- Process Control (PC) supports full interoperability with JWS via Java calls over RMI.

Figure 4-5 shows the recommended architecture, keeping the interoperability limitations in mind.

Figure 4-5 Front-end JWS with SC and JPD with a JWS and Process Control


10.2 SC uses 10.2 style callouts
PC supports full inter-operability with JWS
10.2 JWS
java
10.2 SC
10.2 headers
HTTP/JMS
SOAP 1.2,WS-Security
10.2 JWS
java
10.2 PC
java
10.2 JPD
java
WS-Addressing, WS-RM
10.2 headers
10.2 JWS supports 10.2 style callbacks
Front-end JPD with JWS and PC

A business process can be exposed as a web service (JWS) as follows:

1. Select a business process (JPD). Right click on the JPD. Select **Generate > Process Control**.
2. Generate a fronting JWS using the process control. Right click on the process control. Select **Generate Fronting JWS**.

Communication Between JPDs

A WLI application contains several JPDs that communicate with each other. A JPD that is invoked by other JPDs is called a sub-process. Table 4-7 lists the high-level guidelines that you can follow for JPD to JPD communication.

Table 4-7 High-Level Design Guidelines for JPD Communication

High-Level Design Guideline	Description
For asynchronous communications between processes in different domains, the use of JMS and the messaging bridge is recommended, together with the JMS event generator or web services over JMS.	The messaging bridge supports all of the QoS options for asynchronous messaging between domains. The store-and-forward capabilities of the messaging bridge insulate local processes from problems and provide access to remote providers.
For asynchronous two-way communications between processes in the same domain, it is recommended that you use process control rather than the message broker.	You get approximately the same performance when you start a process using process control or message broker, but it is significantly faster to receive a process control callback than a message broker subscription. The message broker subscription filter mechanism uses a database to map the filter values to process instances. Process control callbacks are routed directly to process instances.
Use raw JMS where the message size is large.	JMS is more efficient than marshalling large messages in web services.
Use the process control, not service or service broker control for synchronous communications between processes in the same domain, if a transaction must be propagated between processes, or performance is an issue.	Process control is transactional and avoids SOAP marshalling. Service and service broker controls are not transactional.
For synchronous communications between processes in different domains, use service and service broker controls or the JPD proxy.	Process control can only be used within a domain. The JPD proxy provides transactional propagation at the cost of tight coupling. Service and service broker control do not provide transaction propagation but enable loose coupling.
Use the service broker control rather than the service or process control, if the data-dependent routing facilities are required.	The service broker control allows for configurable routing of service calls to different service implementations depending on data, but this facility is not transactional.

Using Controls

Controls are an integral part of a JPD. WLI 9.2 controls are based upon open Apache Beehive standards. They support annotations based on JSR-175 standards. Controls are reusable and provide easy access to enterprise resources. Controls can be used within a process definition to make calls to a backend system. For example, the database control allows a process to send SQL to an RDBMS using a JDBC connection pool. Table 4-8 lists the high-level guidelines that you can follow for using controls.

Table 4-8 High-Level Guidelines for Using Controls

High-Level Design Guideline	Description
Control reuse	All controls should be written so that they can be reused across process definitions.
You should develop custom controls keeping in mind native WLI 9.2 controls.	Use the controls available in WLI 9.2 as much as possible. Write custom controls only when absolutely necessary.
Use custom controls instead of custom Java code if reuse of the code is a consideration.	A control that is created for custom Java code means that the code can be reused across different process definitions.
Controls should be versioned in source code control.	Controls should be treated as Java code and versioned appropriately.
Each application project in BEA Workshop for WebLogic should have a separate control project.	Import controls from the component library into this project so that they become available to all the applications.

Using Dynamic Properties and Annotation for Controls

In many cases, control attributes are statically defined using annotations. However, some controls provide a Java API to dynamically change attributes. Dynamic controls, such as service broker and process controls provide the means to dynamically set control attributes. A dynamic or late binding process is used wherein attributes are determined at runtime using a combination of lookup rules and values. Controls that support dynamic binding are called dynamic controls.

Look-up rules are defined during design time. Look-up values can be defined by changing the DynamicProperties.XML file containing the look-up rules and values during runtime via the WLI Administration Console. It contains mappings between values from the message payload (the look-up key) and the corresponding control properties. This file is a domain-wide file shared by all WLI applications in the domain and managed using the WLI Administration Console. This feature allows the complete de-coupling of control attributes from the application. This file allows you to administer dynamic properties while an application is running, and you do not need to redeploy the application for the changes to take effect. The file is located in a subdirectory of the domain root called wliconfig.

To achieve the dynamic binding of properties, use:

- Selectors
- The `setProperties()` API. Use data transformation to set the properties of a control dynamically.
- Setter methods for individual properties, such as `setEndPoint()`.

Use the `getProperties()` method to retrieve the current property settings. You can also use the XML Meta Data Cache control to improve the performance of dynamic data look-up at run time.

Buffering Service Control Methods During Asynchronous Calls

When you call web service controls asynchronously from business processes, it is recommended that you buffer the asynchronous call to ensure that the message sent from the business process to the web service is enqueued. An asynchronous call to a resource marks the boundary of a transaction in your business process. A call to a resource is not enqueued until the transaction is committed.

By buffering the call to the resource, you ensure that the transaction is committed before there is any response from the resource. If you do not buffer the call, your business process must wait for the HTTP acknowledgement before the transaction is committed. In this situation, the resource may attempt to respond to the business process before the HTTP acknowledgement.

Using Control Factory to Manage Multiple Instances of a Control

The control factory feature of a control enables a JPD to interact with a multiple instances of the same JPD. You can implement File, e-mail, WLI JMS, Trading Partner Management, Service, and Worklist controls as control factories. For example, if a JPD is required to send a document, such as loan application, to multiple service providers, it can use a control factory to create multiple instances of the service control, and dispatch requests to the service control instances in parallel. If the control uses callbacks, a single parameterized callback handler in the calling JPD, can manage the callbacks received from all the control instances.

Data Transformation

Data transformation and manipulation is an integrated part of a business process. Service consumers and providers require varied data format and types. WLI provides the following tools for data transformation:

- XQuery(2004)
- XSLT
- Format Builder, Message Format Language (MFL)
- XML beans
- Data model

Note: WLI 9.2 supports the runtime execution of XQuery 2002 for the purpose of backward compatibility with WLI 8.1.

Canonical Data Model

It is recommended that you create a canonical data model for your application. A canonical data model maps the data to an agreed standard form. If correctly implemented, such a data model provides the services with an Enterprise Information System (EIS) neutral interface. It specifies how you can map the reference, static, and identifier data from an EIS data format to a standard format, de-coupling the data model of the host and the data model of the recipient. You should first create standards for the service interface, that in turn, enforces a common representation of data types and entities within the enterprise. You must maintain a consistent method of representing dates, numbers, post codes, and addresses.

Runtime Selection of a Transformation

A dynamic transformation control enables a business process to dynamically select and execute a transformation during runtime. It allows you to choose the XQuery, XSLT, or MFL file that is invoked at runtime. For example, if you have an integration hub that receives documents from various regional offices, you can use the dynamic transformation control to perform different transformations based on the area code of each regional office. Table 4-9 lists the high-level design guidelines for data transformation.

Table 4-9 High-Level Design Guidelines for Data Transformations

High-Level Design Guideline	Description
Create new data transformations using XQuery rather than XSLT.	XQuery has several advantages over XSLT, including additional functionality, and better performance.
Use a data model (Enterprise Data Model (EDM)) only where there are clear benefits that offset the additional cost and complexity.	An enterprise requires a large amount of resources to use a data model. A data model should be used only when an organization is willing to commit the required resources.
Implementing a data model involves high cost and social challenges. You can implement standards at the interface to ensure a quick return on investment.	This is to ensure that data is represented consistently across all service interfaces. For example, dates, post codes, and addresses are always represented in a specific style. Ensure that you represent information such as message headers in a standard format.
The origin of the message should not be disclosed to the recipient**.**	If a data model or standard service interface is appropriately implemented, the recipient of a message is not aware of the data format of the message despatching system.
Wrap any re-usable transformation as a stateless process.	If a re-usable transformation is wrapped as a stateless process, other components can call it a service. This includes direct calls from a front-end system.

Developing a Task Plan

WLI 9.2 worklist has several enhanced features as follows:

- You can model multi-step tasks such as loan approval, and purchase order approval, using a simple drag-and-drop task plan editor. You do not need any prior knowledge of Java or J2EE to be able to use task plan editors.
- A task can be assigned to multiple human actors (one at a time).
- An enhanced and system generated web user interface allows you to work with and test a task plan. You can write your custom user interface using the APIs that are exposed by the task plan.

- The enhanced task assignment and load balancing facilities allow the efficient utilization of a human actor's time.
- The task plan can work without using a JPD. However, you can use a task plan control to interact with the JPD. JPDs can also subscribe to task plan events and process them as per business needs.

Task Plan for Exception Management

You can use the task plan to manage exceptions in a JPD. Users can work on a task plan that is invoked when an exception occurs.

Integrating Custom Logic With a Task Plan

Use the task plan event service to integrate your custom logic with the task plan.

To subscribe to task plan events, write your own custom event listeners and register them with the task plan. You can write custom code in your listener class. This custom code is executed when task plan events invoke the respective event listeners associated with the task.

You can use a custom assignment handler in the custom logic to change the default task assignment at runtime.

See Using the Worklist for detailed information.

CHAPTER 5

Deploying and Maintaining WLI Applications

There are several best practices for deploying, running, and maintaining WLI applications, as explained in the following sections:

- Deploying WLI Application During Runtime
- Deploying WLI Application in a Cluster

Deploying WLI Application During Runtime

When you work in the development mode, you can use the WLI IDE to build and deploy your application. The IDE provides a feature that helps you generate an Ant script to create a build for production purposes. You can also execute these build scripts outside the IDE via the command prompt and generate a single EAR file. You can deploy this EAR file via the command prompt or using the WebLogic Server Console. For more information, see Deploying WebLogic Integration Solutions.

Deploying WLI Application in a Cluster

A WebLogic Server cluster domain contains only one administration server, and one or more managed servers. The managed servers in a WLI domain can be grouped in a cluster. When you configure WLI resources that can be clustered, you target the resources to a named cluster. If you specify a cluster as the target for resource deployment, you can dynamically increase the capacity by adding managed servers to your cluster. The best practices that you can apply to a cluster are as follows:

- Configuring Trading Partner Integration Resources

- Changing Cluster Configurations and Deployment Requests
- Load Balancing in a WLI Cluster

Configuring Trading Partner Integration Resources

You must deploy Trading Partner Integration components homogeneously to a cluster. To avoid a single point of failure, ensure that Trading Partner Integration resources are deployed identically on every managed server.

The guidelines you can follow to configure Trading Partner Integration in a cluster are as follows:

- Specify the host and port numbers of the hardware or software routers as the HTTP end points in the binding of trading partners. This step protects the identity of your managed servers, which are behind a firewall, and allows managed servers to change operational status without impacting the external customer.
- Update Trading Partner Management configuration using the WLI Administration Console. A JMS broadcast mechanism propagates these changes to the managed servers. These changes are quick, but not instantaneous. There is a brief period during which the managed servers contain a mix of the old and new configuration information. To minimize the impact, it is recommended that you update configurations when the resources requiring updates are inactive.

Changing Cluster Configurations and Deployment Requests

You can change configurations for a cluster. For example, you can add new nodes to the cluster or modify Trading Partner Integration configuration only while the administration server of the cluster is active.

Requests to deploy or disable a cluster are interrupted if the administration server is inactive, but the managed servers continue to serve requests. If you can ensure that the required configuration files such as msi-config.xml, SerializedSystemIni.dat, and optionally boot.properties exist in each managed server's root directory, you can boot or reboot managed servers using an existing configuration.

Managed servers that work without an administrative server, operate in a Managed Server Independence (MSI) mode. For more information about MSI mode, see "*Understanding Managed Server Independence Mode*" sub-section in Avoiding and Recovering from Server Failure of *Managing WebLogic Server Start up and Shutdown*.

Load Balancing in a WLI Cluster

One of the goals of clustering in your WLI application is to achieve scalability. For a cluster to be scalable, each server must be fully utilized. Load balancing distributes the work load proportionally among all the servers in a cluster, so that each server can run at full capacity. Load balancing is required for various functional areas in a WLI cluster. The functions are:

- HTTP Functions in a Cluster
- JMS Functions in a Cluster
- Synchronous Clients and Asynchronous Business Processes
- RDBMS Event Generators
- Application Integration Functions in a Cluster

HTTP Functions in a Cluster

Web services (SOAP or XML over HTTP) and WebLogic Trading Partner Integration protocols can use HTTP load balancing. You can use the WebLogic **`HttpClusterServlet`**, a web server plug-in, or a hardware router for external load balancing.

JMS Functions in a Cluster

WLI or WLI applications most often utilize JMS queues that are configured as distributed destinations. The exception to this rule is that a JMS queue is targeted to a single managed server.

Synchronous Clients and Asynchronous Business Processes

If your WLI solution includes communication between a synchronous client and an asynchronous business process, you can enable server affinity for the **`weblogic.jws.jms.QueueConnectionFactory`**. This is the default setting.

WARNING: If you disable server affinity for a solution that includes communication between a synchronous client and an asynchronous business process in an attempt to tune JMS load balancing, the resulting load balancing behavior is unpredictable.

RDBMS Event Generators

The RDBMS event generator has a dedicated JMS connection factory called **`wli.internal.egrdbms.XAQueueConnectionFactory`**. Load balancing is enabled for this connection factory by default. You must disable load balancing and enable server affinity for

`wli.internal.egrdbms.XAQueueConnectionFactory` to disable load balancing for RDBMS events.

Application Integration Functions in a Cluster

Application Integration allows load balancing of synchronous and asynchronous services and events within a cluster. The usage of synchronous and asynchronous services are explained in detail as follows:

- Synchronous Services
- Asynchronous Services

Synchronous Services

Synchronous Services are implemented as method calls on a session EJB. They are load balanced within the cluster according to EJB load balancing rules. These EJBs are published at design time and each application view is represented as two session EJBs: one stateless, and one stateful.

In a standard operation, stateless session EJBs invoke the services, and load balancing occurs on a per-service basis. Every time you invoke a service on an application view, you may be routed to a different EJB on a different WebLogic managed server instance.

When you use the local transaction facilities of the application view during a local transaction, the stateful session EJB invokes the services. The stateful session EJB keeps the connection to the EIS open, so that the local transaction state can persist between service invocations. In this mode, service invocations are pinned to a single EJB instance on a single managed server within the cluster. Once the transaction is complete, either through a commit or rollback, the standard per-service load balancing is applicable.

Asynchronous Services

Asynchronous services are always invoked as method calls on a stateless session EJB. You cannot use the local transaction facility of the application view for asynchronous service invocations.

A single asynchronous service invocation translates to two method invocations on two different stateless session EJB instances. The load balancing for asynchronous service in this case occurs on two occasions, the first upon receipt of the request, and the second in the execution of the request and delivery of the response.

In addition, both the asynchronous service request and response are posted to a distributed JMS queue. JMS load balancing as a result, applies to both the request and the response. In this case, the **`invokeServiceAsync`** method of the application view may be serviced on one managed server, the request delivered to a second managed server where the request is processed and the response generated, and the response delivered to a third server for retrieval by the client.

WLI Performance Tips and Application Recovery

Performance Tips

The following sections provides tips about designing and tuning high-performance WebLogic Integration applications. It contains the following topics:

- Design-Time Performance Tips
- Run-Time Tuning Tips

Design-Time Performance Tips

This section provides the following design time performance tips:

- Processes, Controls, and Callback Methods
- Best Performance From Parallel Nodes in Business Processes
- Pending Messages in JMS Queues
- Removing Pending JMS Timer Messages
- Increasing the Transaction Timeout Period

Processes, Controls, and Callback Methods

- Stateless processes are significantly faster than stateful processes.
- Process controls are faster than service controls.

- You get approximately the same performance when starting a process using the process control or message broker, but it is significantly faster to receive a process control callback than a message broker subscription. The message broker subscription filter mechanism uses a database to map the filter values to process instances. Process control callbacks are routed directly to the process instances.

Best Performance From Parallel Nodes in Business Processes

By default, the branches of a parallel node are transactionally isolated. To improve performance, you can override this behavior by setting the `continueTransaction` property in the source for each parallel element as follows:

```
<parallel continueTransaction="true">
```

Pending Messages in JMS Queues

Timer messages (messages set with a delivery time in the future) show up as "pending messages" in the WebLogic Server console.

Removing Pending JMS Timer Messages

For stateful processes, turn off maximum conversation lifetime if you are not using it:

```
weblogic.jws.Conversational
```

For more information on the conversation lifetime feature, see "@jws:conversation-lifetime Annotation" in Java Web Service Annotations in the "WebLogic Workshop Reference" in WebLogic Workshop Help.

Increasing the Transaction Timeout Period

The default timeout period for a transaction is 300 seconds (5 minutes). You can increase the amount of elapsed time before a transaction times out by using one of the following approaches:

- Increase the value of the `transaction-timeout` element of `wlw-config.xml.` This will change the timeout period for all transactions in your application. For more information, see wlw-config.xml Configuration File in the "Configuration File Reference" in "WebLogic Workshop Reference" in the WebLogic Workshop Help.

Note: The `wlw-config.xml` file is same for both the 8.1 and 9.2 releases.

- Increase the `trans-timeout-seconds` value for the AsyncDispatcher MDB in `weblogic-ejb-jar.xml`. This will change the timeout period for transactions processed by the AsyncDispatcher MDB only. For more information, see "trans-timeout-seconds" in

"2.0 weblogic-ejb-jar.xml in WebLogic Server 9.2" in the weblogic-ejb-jar.xml Deployment Descriptor Reference in *Programming WebLogic Enterprise JavaBeans*.

For recovery considerations regarding transaction timeout periods, see "Configuring WebLogic Integration Application for Recovery" on page 6-9.

Run-Time Tuning Tips

This section provides run-time tuning tips:

- Using Flags to Start WebLogic Server and Maximize Performance
- Tuning WebLogic Integration Applications
- Configuring max-beans-in-free-pool for the JMS Event Generator
- Versioning for a Stateful Business Process
- Setting Process Tracking Levels to Optimize Performance
- Running the Archiver When Process Tracking Levels are set to None
- Disabling Information Web Services Messages
- Using the Document Store With the WebLogic Integration Application
- Choosing the Best Persistence Model for JMS
- Monitoring Parameters for WebLogic Integration Application Running Under Load
- Reducing the Number of Transactions Timing out
- Causes for Memory Leaks in Application

Using Flags to Start WebLogic Server and Maximize Performance

For optimum performance, use the following flags when starting WebLogic Server:

```
production noiterativedev nodebug notestconsole
```

Note: These flags should be set by default when using a generated production domain, but not when using a development domain.

Tuning WebLogic Integration Applications

WebLogic Integration projects map onto J2EE resources. The primary EJB pools you can tune within a WebLogic Integration project are the project beans: the `SyncDispatcher` Stateless

Session Bean (SLSB) pool and the `AsyncDispatcher` Message-Driven Bean (MDB) pool. These pools exist for every WebLogic Integration project in an application.

All synchronous requests are routed through the `SyncDispatcher`; all asynchronous (buffered) communication is routed through the `AsyncDispatcher`.

There are two ways to configure the pool size of these beans:

- The preferred method to configure MDB pool size is to configure the EJBs in the project beans directory to use a dedicated thread pool by updating the `weblogic-ejb-jar.xml` file to contain a `dispatch-policy` element with the name of the thread pool to use.

  For more information, see "dispatch-policy" in "2.0 weblogic-ejb-jar.xml Elements" in the weblogic-ejb-jar.xml Deployment Descriptor Reference in *Programming WebLogic Enterprise JavaBeans*.

- You can use deployment plans to perform the same task. For more information, see Configuring Applications for Production Deployment in *Deploying Applications to WebLogic Server.*

- Another method to configure MDB pool size is to set `max-beans-in-free-pool` for the project beans.

  For more information about configuring MDB pool size, see "Setting Performance-Related weblogic-ejb-jar.xml Parameters" in Tuning WebLogic Server EJBs in *WebLogic Server Performance and Tuning*.

For background information on tuning in general, see *WebLogic Server Performance and Tuning.*

Configuring max-beans-in-free-pool for the JMS Event Generator

Yes. The JMS event generator by default has `max-beans-in-free-pool` set to `5`. You can often set this value higher and improve performance.

For more information, see "Setting Performance-Related weblogic-ejb-jar.xml Parameters" in Tuning WebLogic Server EJBs in *WebLogic Server Performance and Tuning*.

Versioning for a Stateful Business Process

If you ever plan to use versioning with a long-running business process, version your process from the beginning before deploying your application in production mode. Otherwise, you must let non-versioned instances run to completion before deploying the new versioned process.

For more information, see "Managing Process Versions" in Process Configuration in *Managing WebLogic Integration Solutions*.

Setting Process Tracking Levels to Optimize Performance

Minimize process tracking levels as much as possible to optimize performance. Set tracking to `none` as the default, and track selected JPDs as needed.

For information on how to configure tracking levels, see "Managing Process Tracking Data" in "About Process Configuration" in Process Configuration in *Managing WebLogic Integration Solutions*.

Running the Archiver When Process Tracking Levels are set to None

Yes. You should run the archiver process during non-peak hours to clean the process monitor table for stateful processes.

For information on how to configure the archiver process, see "Managing Process Tracking Data" in Process Configuration in *Managing WebLogic Integration Solutions*.

Disabling Information Web Services Messages

Edit `<domain-home>/apacheLog4jCfg.xml` and change all occurrences of `info` to `warn`.

Using the Document Store With the WebLogic Integration Application

Each application will have a different threshold for the size of documents it is more efficient to pass by reference (using the document store) rather than inline. In general, the more trips a document makes, the better it is to use the document store. You configure the size of files passed by reference by setting the value of the `weblogic.wli.DocumentMaxInlineSize` parameter in the WebLogic Integration application domain's `wli-config.properties` file.

Note: When using the document store, a two-phase commit related race condition in some cases may cause the following message to appear in the log: `SQLException in retrievData()`. In this situation, you can set the value of `weblogic.wli.DocumentMaxInlineSize` to a large number so that all documents will be passed inline.

Choosing the Best Persistence Model for JMS

If you have a fast disk subsystem or a battery backed caching controller, file-based persistence can be quite a bit faster than JDBC.

Monitoring Parameters for WebLogic Integration Application Running Under Load

The following table shows parameters of key resources used by your WebLogic Integration application that you should monitor in the WebLogic Server Administration Console, and actions you should take if a problem condition arises.

Table 6-1 Parameters to Monitor in WebLogic Integration Applications

Parameter	Where to Monitor	Problem Condition	Actions Required
JDBC connection pool usage	Services→JDBC Data Sources→each data-source→Monitoring.	Connections in use approaching maximum available.	Increase the `MaxCapacity` attribute of `JDBCConnectionPool`. For more information, see “How JDBC Connection Pools Enhance Performance” in Tuning WebLogic Server in *WebLogic Server Performance and Tuning*.
JMS async queues for project beans	Services→Messaging→JMS Modules→conversational-jms.	Many messages in the queue.	Increase the pool size for the AsyncDispatcher. For more information, see Tuning WebLogic Integration Applications.

Table 6-1 (Continued) Parameters to Monitor in WebLogic Integration Applications

Parameter	Where to Monitor	Problem Condition	Actions Required
`wli.internal.tracking.buffer`	Services→Messaging→JMS Modules→conversational-jms.	Many messages in the queue	Create a thread execute queue named `wli.internal.ProcessTracking`, and increase the number of threads in that pool. For more information, see Tuning WebLogic Server in *WebLogic Server Performance and Tuning*.
`wli.internal.instance.info.buffer`	Services→Messaging→JMS Modules→conversational-jms.	Many messages in the queue	Create a thread execute queue named `wli.internal.ProcessTracking`, and increase the number of threads in that pool. For more information, see Tuning WebLogic Server in *WebLogic Server Performance and Tuning*.

Reducing the Number of Transactions Timing out

If you have deployed your application as an exploded EAR, you can increase the amount of time that elapses before a transaction times out by increasing the value of `trans-timeout-seconds` in the WebLogic Server Administration Console. You should also change this value in the source for your WebLogic Integration application so that the value of `trans-timeout-seconds` is not reset to its default value when you redeploy the application.

If you have not deployed your application as an exploded EAR, `trans-timeout-seconds` is not accessible through the WebLogic Server Administration Console. You must increase the value of `trans-timeout-seconds` in your application, and then redeploy the application with the increased timeout value.

For information on how to change the value of `trans-timeout-seconds`, see “Edit Deployment Descriptors” in Implementing Enterprise Java Beans in *Programming WebLogic Enterprise JavaBeans*.

Causes for Memory Leaks in Application

The top causes of memory leaks are:

- Operating a WebLogic Integration application with the test console running. This is not actually a leak, but running with the test console on uses a lot of memory.

  For more information, see Using Flags to Start WebLogic Server and Maximize Performance.

- Lots of JMS timer messages. This is also not actually a memory leak, but this situation does cause an increasing amount of memory to be used. The memory is recovered when the messages are delivered.

  To reduce JMS timer messages, see Increasing the Transaction Timeout Period.

WLI Application Recovery

The following section provides answers to questions about configuring WebLogic Integration applications for recovery. It contains the following topics:

- Configuring WebLogic Integration Application for Recovery
- Starting the Recovery Process

Configuring WebLogic Integration Application for Recovery

You should check to make sure your WebLogic Integration application is configured in the following manner:

- cgDataSource is configured in `cgPool`.Best Practices for WebLogic® Integration Application Life Cycle
- All JMS servers are targeted to a migratable target. (Migratable targets have the word *migratable* in the target name.)

  Note: If you need to change the targeting of one or more JMS servers in an active cluster, you must restart the cluster after you make changes to the targeting configuration.

- The `WLI Admin` component of `WLI System EJBs` is targeted to the cluster. You can verify the targeting of `WLI Admin` by inspecting its configuration in the `config.xml` file for the `WLI System EJBs` application.

  Note: `WLI Admin` can be targeted to the cluster only when JMS servers are targeted to migratable targets.

- Retries and retry intervals are appropriate. Retries multiplied by retry interval should exceed the time it takes for JTA recovery to run.

  If you need to tune your retries and retry intervals, you have the following choices:

  – Carefully set `retry-count` and `retry-interval` in your JPDs.

  – Set the `retry-count` and `retry-interval` of the async and error queues for each JPD project (WebApp).

    Note: This will break explicit retry settings on the JPD, but it is the easiest and recommended approach.

  For more details, see JMSQueue in the *WebLogic Server Configuration Reference*.

- `JTA` attribute `TimeoutSeconds` in the `config.xml` file is set to a value less than the value of the `JDBCConnectionPool.XA TransactionTimeout` attribute for XA connection pools.

  For more information about `TimeoutSeconds`, see JTA in the *WebLogic Server Configuration Reference*.

- `JTA` attribute `MaxTransactions` in the `config.xml` file is set to a value large enough to accommodate the number of simultaneous transactions that could occur on a server during recovery. For transaction intensive applications, you should increase the value of this attribute from the default setting.

  For more information about `MaxTransactions`, see Domain-->Configuration-->JTA in the *WebLogic Server Administration Console Online Help*.

- `JDBCConnectionPool` attribute `MaxCapacity` in the `config.xml` file is set to a value greater than the value of the execute queue `ThreadCount` attribute.

  For more information about `MaxCapacity`, see JDBCConnectionPool in the *WebLogic Server Configuration Reference*.

  For information about `Server ThreadCount`, see "Tuning the Default Execute Queue Threads" in Tuning WebLogic Server in *WebLogic Server Performance and Tuning*.

- When using Oracle, the repository tables `dba_2pc_pending`, `dba_2pc_neighbors`, and `dba_pending_transactions` must have the right permissions for recovery to be called. If they do not, you will get a database error and recovery will fail.

Starting the Recovery Process

It can take several minutes for in-doubt transactions to show up in Oracle. There may be a race if recovery has started prior to Oracle detecting the loss of the TM. Wait several minutes before starting recovery—either by restarting the server or by doing a JTA migration.

The Recovery process will fail if after initiating recovery, you check the Oracle `dba_2pc_pending` table and see a record associated with the failed server that has a timestamp prior to initiating recovery. Restart the server.

CHAPTER 7

Core Implementation Patterns for WLI Applications

There are several core implementation patterns for WLI applications, as explained in the following sections:

- Core Implementation Patterns for a JPD
- Coarse-Grained Process Front-end for a Fine-Grained Process
- Loosely Coupled Process With a Common Message Interface
- Dynamic Property Driven Processes

Core Implementation Patterns for a JPD

JPDs are of several types and are designed with the following intrinsic characteristics:

- Basic complexity: A basic process makes only one call out to another service or an external system. A basic process does not contain complex logic but may contain any number of steps for transformation and exception handling.
- Composite complexity: A composite process contains two or more calls out to external systems. A process that makes two calls out to a single back end service is defined as composite. Composite processes may contain complex logic such as parallel calls out, and client requests after the start node.

Note: Composite processes have more complex exception management because of the multiple calls.

- Synchronous or asynchronous calling paradigm

- Stateful or stateless JPD state
- One or two-way exchange paradigm

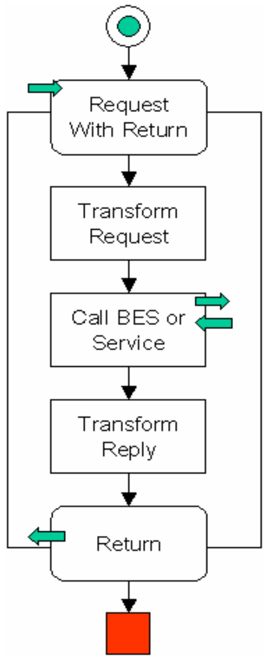
Several patterns of JPDs have resulted from combinations of these characteristics. The following sections describe these patterns in brief:

- Pattern 1: Basic Synchronous Stateless two-way Service
- Pattern 2: Basic Asynchronous Stateless two-way Service
- Pattern 3: Basic Asynchronous Stateless one-way Service
- Pattern 4: Basic Asynchronous Stateful two-way Service
- Pattern 5: Basic Asynchronous Stateful one-way Service
- Pattern 6: Composite Synchronous Stateless two-way Service
- Pattern 7: Composite Synchronous Stateful two-way Service
- Pattern 8: Composite Asynchronous Stateless two-way Service
- Pattern 9: Composite Asynchronous Stateless one-way Service
- Pattern 10: Composite Asynchronous Stateful two-way Service
- Pattern 11: Composite Asynchronous Stateful one-way Service
- Other Patterns

Pattern 1: Basic Synchronous Stateless two-way Service

Use this pattern illustrated in Figure 7-1 to implement some of the fastest processes. The standard usage is for simple access to a backend system or to implement helper processes.

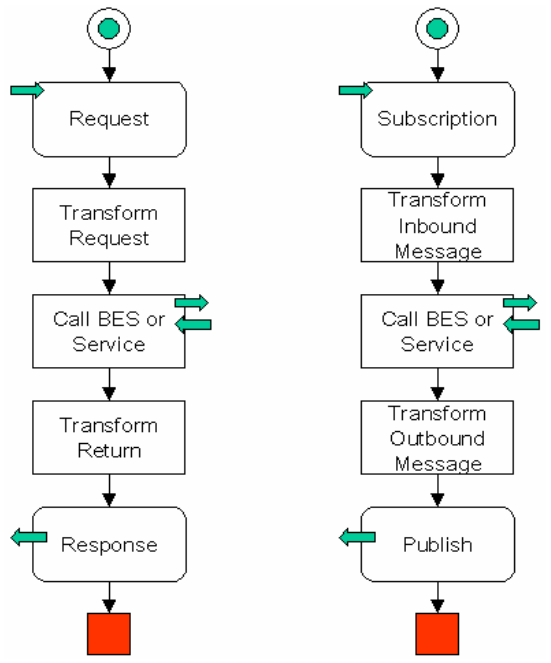
Figure 7-1 Pattern 1



Pattern 2: Basic Asynchronous Stateless two-way Service

Use this pattern illustrated in Figure 7-2 to implement some of the fastest processes. The standard usage is for simple access to a backend system, when de-coupling with the process client is required.

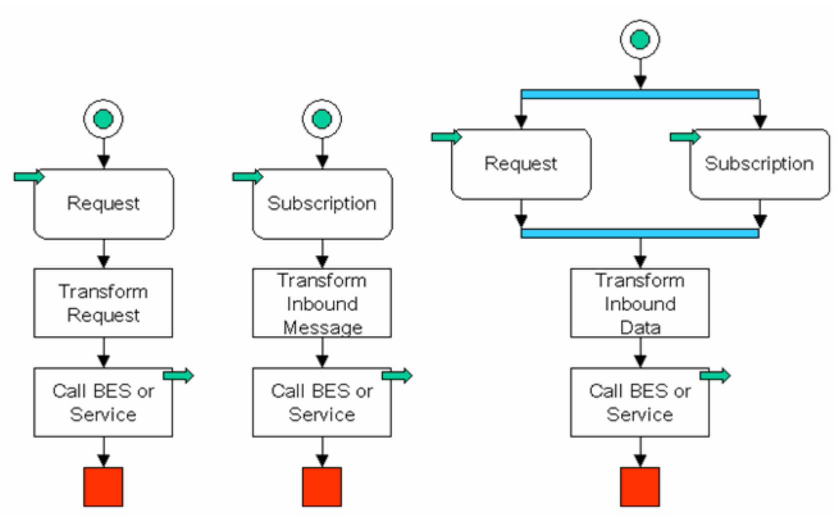
Figure 7-2 Pattern 2



Pattern 3: Basic Asynchronous Stateless one-way Service

Use this pattern illustrated in Figure 7-3 to implement some of the fastest processes. The standard usage is to access backend systems in event-driven situations.

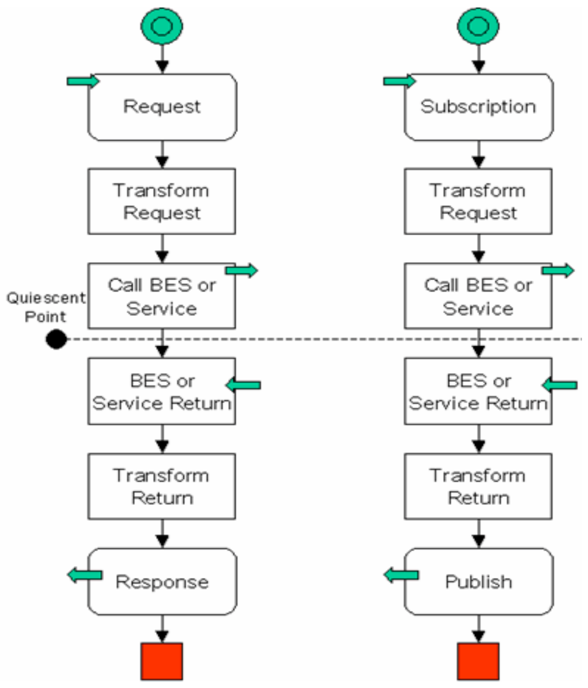
Figure 7-3 Pattern 3



Pattern 4: Basic Asynchronous Stateful two-way Service

This pattern is not as fast as its stateless equivalent. Use this pattern illustrated in Figure 7-4 to access backend systems that provide an asynchronous interface.

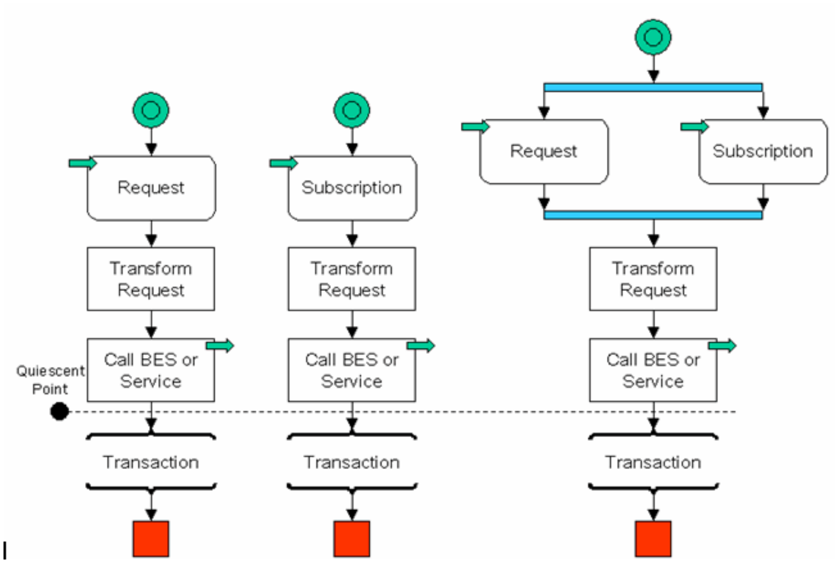
Figure 7-4 Pattern 4



Pattern 5: Basic Asynchronous Stateful one-way Service

This pattern is very rare because most of the one-way services are stateless and so there is no need to wait for an answer. Use this pattern illustrated in Figure 7-5 to quickly access a backend system in event-driven situation where a call that is waiting is required.

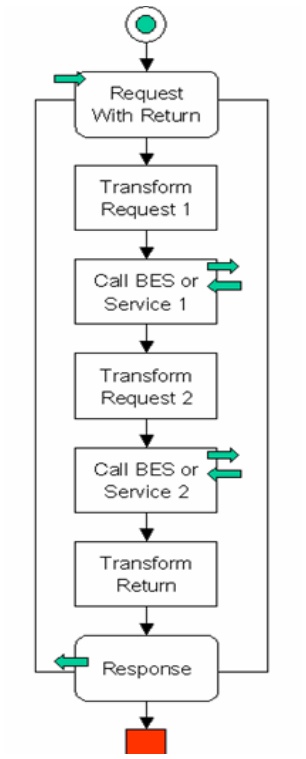
Figure 7-5 Pattern 5



Pattern 6: Composite Synchronous Stateless two-way Service

Use this pattern illustrated in Figure 7-6 to implement integration logic that requires good performance and coupling with the client.

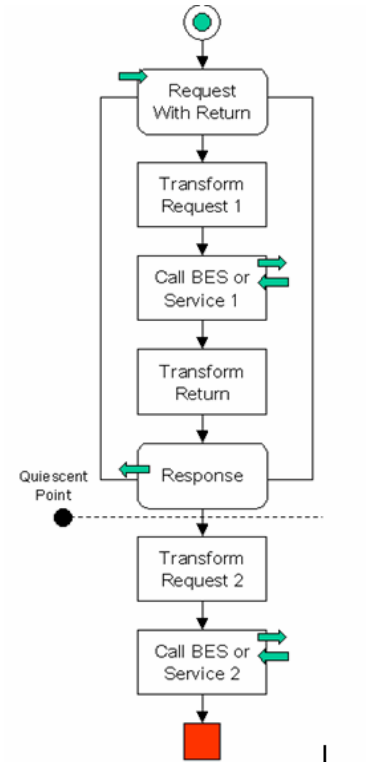
Figure 7-6 Pattern 6



Pattern 7: Composite Synchronous Stateful two-way Service

This pattern represents an unusual case, where the logic is implemented after the client has received a response. Use this pattern illustrated in Figure 7-7 to enable a stateful process that runs for a period of time, and is started by a synchronous request or response. Once the synchronous reply is sent, the process moves to a traditional asynchronous model.

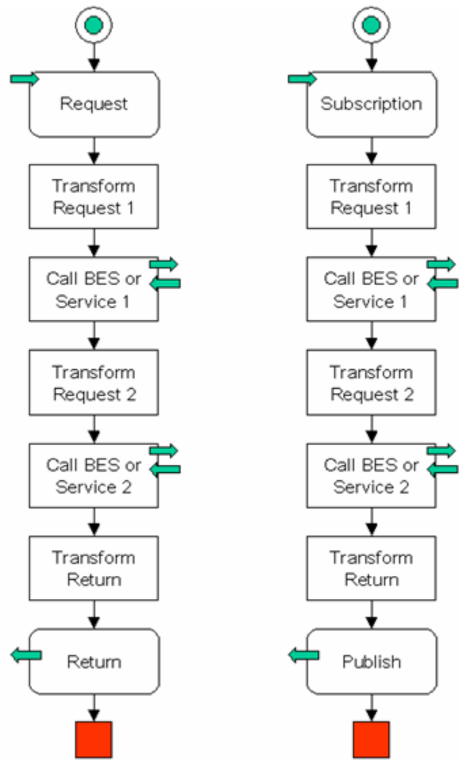
Figure 7-7 Pattern 7



Pattern 8: Composite Asynchronous Stateless two-way Service

This pattern illustrated in Figure 7-8 is a standard method to implement integration logic that requires good performance while maintaining de-coupling from its client. It can also be used to access a backend system when de-coupling from the caller is required. It is not always possible to implement this pattern as it requires all the resources used to be stateless.

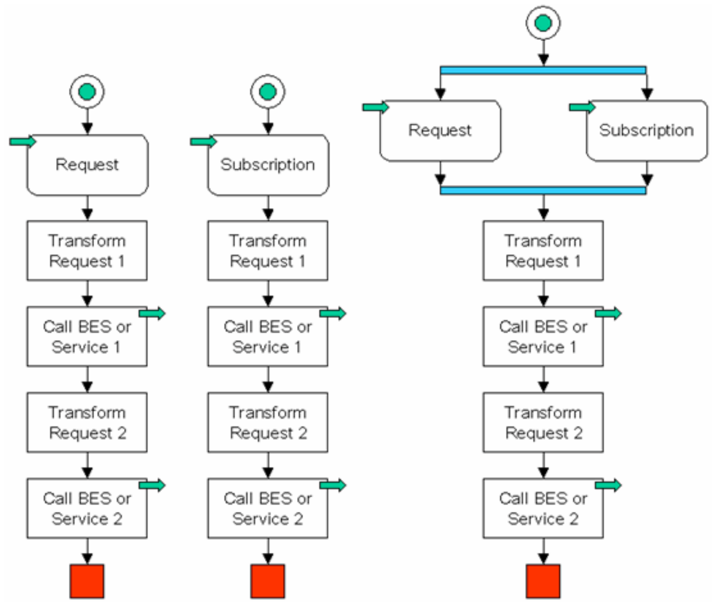
Figure 7-8 Pattern 8



Pattern 9: Composite Asynchronous Stateless one-way Service

Use this pattern illustrated in Figure 7-9 to implement integration logic that requires good performance in event-driven situations.

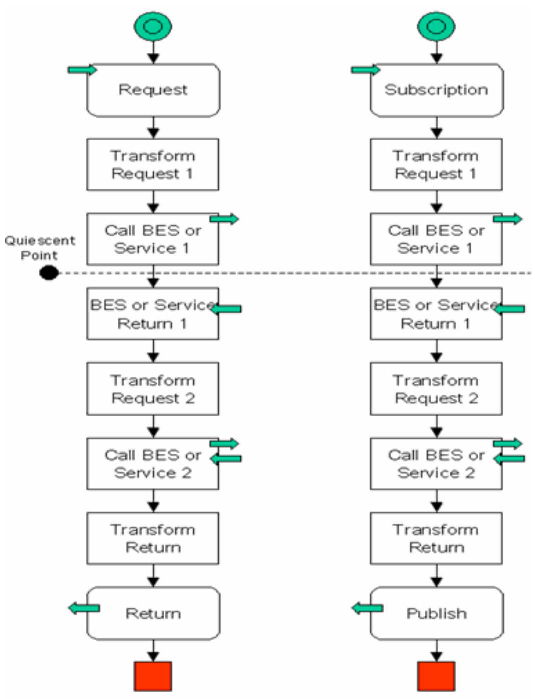
Figure 7-9 Pattern 9



Pattern 10: Composite Asynchronous Stateful two-way Service

Use this pattern illustrated in Figure 7-10 when the stateless version of the same pattern is not possible, as at least one of the resources contains an asynchronous interface. It is also used as a standard pattern for long-running processes.

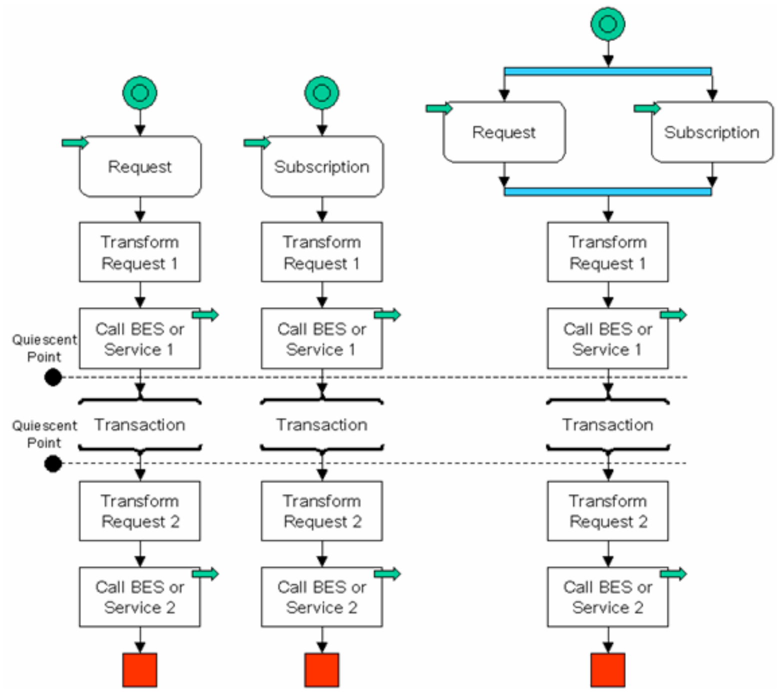
Figure 7-10 Pattern 10



Pattern 11: Composite Asynchronous Stateful one-way Service

This pattern is not common as most of the one-way services are usually stateless and there is no need to wait for an answer. Use this pattern illustrated in Figure 7-11 to implement integration logic in event-driven scenarios, and where a call that is waiting is also required.

Figure 7-11 Pattern 11



Other Patterns

This section re-groups useful patterns, which can be used in combination with one of the core patterns.

SyncAsync Pattern

You can create a business process containing a client request node with a sync or async callback name attribute property, to enable synchronous clients to interact with business processes that have asynchronous interactions with resources. The client request node property holds the name of the callback method that the associated client response node uses. The client request and client response nodes delineate the activities (including asynchronous activities) that occur while the client is blocking the process. After setting this property, generate the sync-to-async WSDL. The synchronous WSDL generation process replaces the SOAP address of the service with a modified SOAP address. The modified address causes the synchronous servlet to process the client request and subsequent return action. A sample of the generated service entry is as follows:

Normal WSDL

```
<service name="syncAsync">

<port name="syncAsyncSoap" binding="s0: syncAsyncSoap">

<soap: address

location="http://localhost:7001/SyncAsyncWeb/processes/syncAsync.jpd"/>

</port>
```

Synchronous WSDL

```
<service name="syncAsync">

<port name="syncAsyncSoap" binding="s0: syncAsyncSoap">

<soap:address

location="http://localhost:7001/sync2AsyncIM/SyncAsyncWeb/processes/syncAs
ync.sync2JPD"/>
```

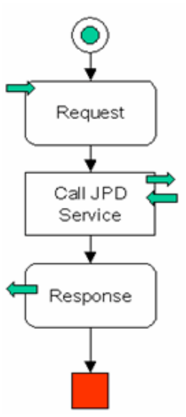
De-Synchronizer Pattern

The de-synchronizer pattern allows you to asynchronously call a synchronous process. This is recommended if:

- The process needs to support both synchronous and asynchronous clients.
- De-coupling is required, in particular if a sub-process needs to run in its own transaction.
- The client is unable to call services synchronously.

Implement the process in Figure 7-12 to provide an asynchronous interface to a synchronous process.

Figure 7-12 De-Synchronizer Pattern



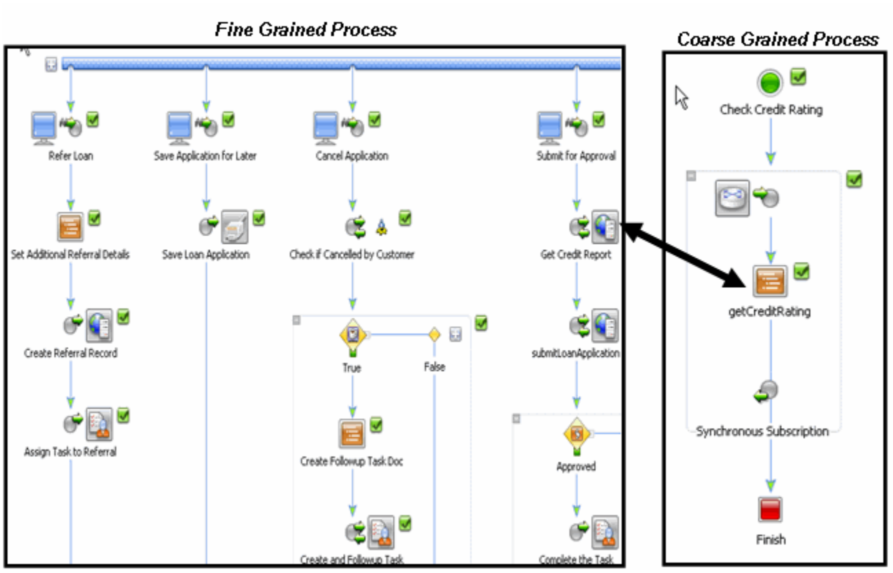
De-Synchronizer Service

This is a simple proxy process. The signature of the request and the response should remain the same as that of the original service. The de-synchronizer passes its input values to the sub-process and sends the return value to the caller of the sub-process.

Coarse-Grained Process Front-end for a Fine-Grained Process

Figure 7-13 illustrates an example of how you can use a combination of coarse-grained and fine grained processes. A coarse-grained process assigns specific work to a number of fine-grained processes.

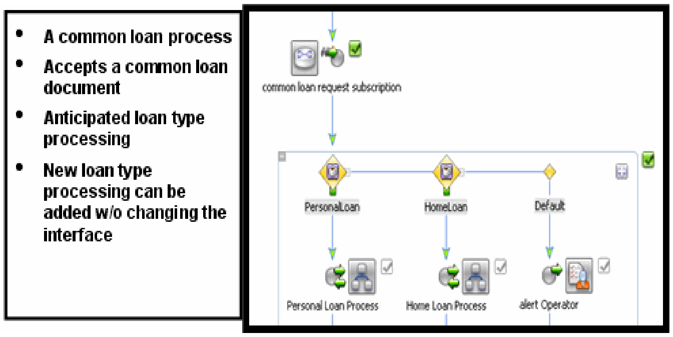
Figure 7-13 Front-end a Fine-Grained Process With a Coarse-Grained Process



Loosely Coupled Process With a Common Message Interface

This pattern illustrated in Figure 7-14 shows how you can create a loosely coupled process using a common messaging interface through message brokers, based upon a Publish and Subscribe architecture.

Figure 7-14 Loosely Coupled Process Using Message Interface


- A common loan process
- Accepts a common loan document
- Anticipated loan type processing
- New loan type processing can be added w/o changing the interface

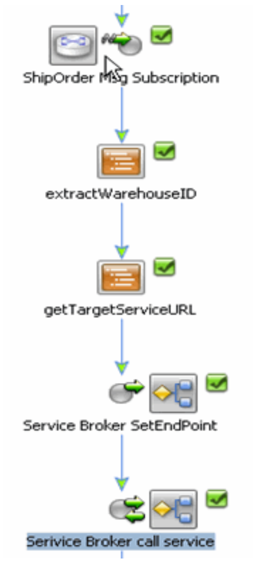
Dynamic Property Driven Processes

You can use the following controls to define a dynamic, property driven process:

- Dynamic controls:
  - Dynamic Transformation control
  - XML Meta data control
- Dynamic Binding controls:
  - Service Broker control
  - Process control
- Dynamic subscription:
  - Message Broker Subscription control
- Runtime controls - Most controls have **setProperties**(<*type*>**ControlPropertiesDocument xml**)

Figure 7-15 illustrates an example of an agile and dynamic process with a Register and Table look-up.

Figure 7-15 Agile, Dynamic Process with a Register and Table Look-up


ShipOrder Msg Subscription
extractWarehouseID
getTargetServiceURL
Service Broker SetEndPoint
Serivice Broker call service

The features of this process are as follows:

- The target URL is determined at runtime and contains a:
  - Registry Look-up
  - Table Look-up
- The Service Broker control accepts the end point as the runtime property
- The Service Broker invokes the specified service at a location determined at runtime.

IDE Recommendations

There are a few, significant differences in developing applications between the Workshop-based WebLogic Integration (WLI) 8.x IDE and the Eclipse-based WLI 10.2 IDE. This document provides tips and tricks to users familiar with the WLI 8.x environment, which help them transition to developing applications in the WLI 10.2 IDE.

This document includes the following sections:

Differences between WLI 8.x IDE and WLI 10.2 IDE

The Integrated Development Environment (IDE) for WLI 10.2 is based on BEA Workshop for WebLogic Platform 10.2, which uses Eclipse 3.2.2. It is different from the WLI 8.x IDE as described in the following sections:

Project Structure

WLI 8.x applications included artifacts and other sub-projects such as schema, web, and EJB projects and they were hierarchical in nature. In 10.2, WLI applications have a flat organization and consist of an EAR project, one or many Web projects and one or many Utility projects.

Each project maintains references to other related projects in WLI 10.2. When a WLI 10.2 application is created through the Process Application Wizard, these references are already established. If projects are imported in WLI 10.2, then such references should be manually updated (see Managing Project Dependencies).

In a WLI 10.2 application, an EAR project is the central point of the application. It is used to create EAR (Enterprise Archive) files and includes:

- JAR files that are shared by the projects in the enterprise application
- Links to all of the projects in the application used by Workshop

Note: Libraries that need to be available to any project in the application should be stored in `<EAR project>/EarContent/APP-INF/lib`.

In a WLI 10.2 application, a Web project includes processes, controls, XQ files, message broker, and transformation files. A Utility project includes schemas, XML, and WSDLs.

When a project is created, you are required to do some or all of the following:

- Specify the type of project
- Add standard libraries
- Set compiler options
- Control publishing tasks
- Set build path or add an annotation processor.

The above options are specified by choosing facets during project creation. Basically each project type has its own facets, which assigns the corresponding builders, validators. Facets can also be added and deleted from a project after its initial creation. To edit a project's facets, select **Project > Properties > Project Facets**.

Note: A process can be created only in a WEB project which has Weblogic Integration Process facet (process-enabled) added to it. Similarly a Worklist task plan can be created in an EAR project which has Worklist Integration Worklist Application Module facet (worklist-enabled) added to it.

Table 4-1 lists various artifacts and their locations.

Eclipse-based Projects

WLI 10.2 Applications and their projects depend on Eclipse. Unlike WLI 8.x Applications, you cannot move projects from one system to another by merely copying projects. In 10.2, the projects have to be imported into the workspace using the Import wizard and selecting the **Existing Projects into Workspace** option.

File Extensions

In WLI 8.x, the various types of WLI artifacts had their own file extensions like .jpd, .dtf, .jcx. In 10.2, unique file extensions for specialized java files like .jpd, .dtf, .jcx no longer exist. All file extensions end with .java (for example, process.java). Therefore, while creating WLI 10.2 artifacts it is a good practice to name Process files, controls, and data transformations in such a way that each can be identified easily.

Note: XQuery files have .xq extensions in 9.2.

No Wizard for Some Controls

In WLI 10.2, some of the Workshop controls (for example, JDBC or Timer) no longer have wizards. You must use the **Source** view or **Annotation** view to configure them.

Using Enterprise JavaBeans (EJB) Controls

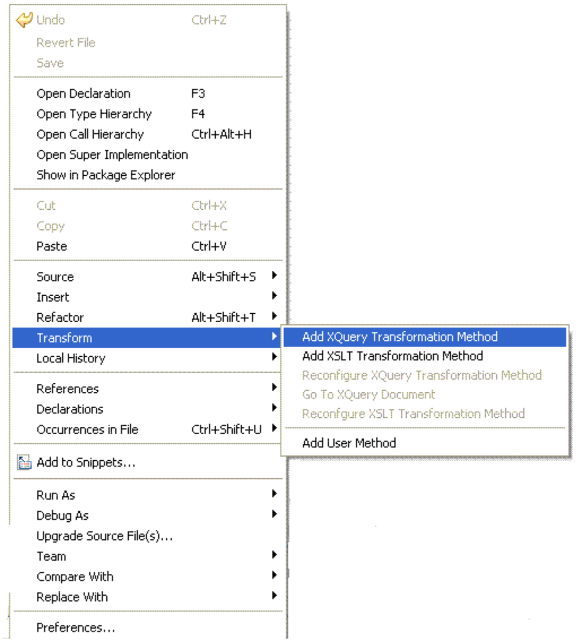
In WLI 10.2, to create a new EJB control, you need to import the ejb jar file and the source code of the application.

DTF Design View not Available

In WLI 10.2, there is no DTF design view. Only a **Source** view is available.

To configure a Transformation method, right-click **Source** view, then select **Transform**, and then select one of the options (see Figure 8-1).

Figure 8-1 Configure Transformation Method



To reconfigure a Transformation method, open the transformation file and in **Source** view, select the transformation method and then right-click, select **Transform** > **Reconfigure XQuery Transformation Method**.

XQuery

WLI 8.x supports XQuery 2002. WLI 10.2 supports XQuery 2004, and XQuery 2002 support is extended for backward compatibility. Upgrade of an Xquery file from 8.x to 10.2 may not be successful due to the incompatibility between the XQuery specifications. In such cases, you are expected to manually correct the upgraded xquery file (see Updating XQuery Use to Support XQuery Implementation).

Note: An XQuery file which has not been configured in WLI 8.x can be upgraded successfully in WLI 10.2, but will lead to a compilation failure, as WLI 10.2 enforces strict configuration for XQuery method.

Iterative Development

In WLI 10.2, if the interface of a Process file is modified, then you need to recreate all the other dependant artifacts (such as process controls that are generated out of this Process file, WSDLs generated from this Process file). In addition, iterative development does not automatically redeploy the application as in WLI 8.x.

In the case of Task Controls, it is very necessary to regenerate the task control each time you make a change to the Task Plan and reuse it in the process.

Standards

WLI 8.x uses the Javadoc-comment style annotations. In WLI 10.2, Java 5 annotations are used; therefore, when upgrading a WLI 8.x application, the annotations in the application are converted to 10.2 annotations. Some of the 8.x annotations are deprecated, because of which the conversion to WLI 10.2 annotations may cause error. You have to manually correct them (see Upgrading Annotations).

Upgrading Sample Application System Schemas

The WLI 8.x Schema Builder allows you to a build sample application even though the namespace is not specified. The WLI 10.2 Schema Builder does not support empty namespaces. If you import the `envelope.xsd` from WLI 8.x, it will lead to some errors.

To fix these errors do one of the following:

- Import the schema from the WLI 10.2 Sample Application
- After importing the 8.x schema, change it from:
  ```
  xmlns:eb="http://www.oasis-open.org/committees/ebxml-msg/schema/msg-hea
  der-2_0.xsd">
  ```
  to:
  ```
  xmlns:eb="http://www.oasis-open.org/committees/ebxml-msg/schema/msg-hea
  der-2_0.xsd">
  schemaLocation="http://www.oasis-open.org/committees/ebxml-msg/schema/m
  sg-header-2_0.xsd"/>
  ```

  Note: Import namespace statements should be added.

Mapper Test View

The XQuery Mapper test view in WLI 8.x is not a standalone tester and always requires a running WLI server. Where as, the XQuery Mapper test view in WLI 10.2 is a standalone tester and does not runs inside a process file container. (It does not matter if there is a WLI 10.2 server running, the XQuery Mapper tester does not use it). This is by design, because the XQuery Mapper is used in other BEA products outside of WLI.

Performance Setting for a Complex Application

If you are developing a complex application with high consumption of memory, change your JVM setting by changing the memory parameters in the file **`workSpaceStudio.ini`** available at your *BEA_HOME* directory (for example, `C:\bea\workSpaceStudio_1.1\workSpaceStudio\workSpaceStudio.ini`) as follows:

- Xms384m: Do not change any setting
- Xmx768m: You can change the setting to 1024m.

Note: The above settings are only a recommendation; you can calculate settings best suited for your machine (see Configuration Requirements).

Guidelines

Guidelines While Creating WLI Application

1. In case you have to work on a large process, it is a good practice to divide it into a number of subprocesses (Each subprocess is also a process in itself). Individual developers can work on each sub-process. Later, all sub-processes can be called through a central process.
2. It is good practice to, **Select Add WebLogic Integration System and Control Schemas** in Utility Project check box this adds the system schemas to the Utility Project/schemas folder.
3. WLI 10.2 provides a project wizard which helps you to create projects required for process and Worklist applications. In many applications, your business process may interact with a task plan. In such scenarios, it is a good practice to create a project and then add a Worklist facet to your project as follows:
   – Right-click on your Ear Project, go to **Properties** > **Project Facets** > **Add and Remove Project Facets**, and select **WebLogic Integration Worklist Application Module** and then click **Finish**.
   – Right-click on your Web Project, go to **Properties** > **Project Facets** > **Add and Remove Project Facets**, and select **WebLogic Integration Worklist Application Module** and **Beehive NetUI 1.0** and then click **Finish**.
   – Create a folder under your **Ear Project** > **EarContent**, and create the task plans inside the EarContent folder.

Once you add a worklist facet to a process project, you can create a Process file and a task plan in the same application.

4. Whenever you move to a specific artifact (Process file, XQ, or Worklist), ensure that you are in the appropriate perspective (Process, XQuery Transformation, and Task Plan respectively). For example, navigating to an XQuery file from a Process file does not automatically change the perspective to the XQuery Transformation perspective.

5. If you are working with CVS, ensure the following:

   Do not check in the following directories to source control. (Not all of these directories and files will exist in every project.)

   – .metadata (workspace-level)
   – build (project-level folder; In every web and util project, the build folder should be excluded)
   – templib (project-level; in every web and util project, the temp lib folder, if present, should be excluded)
   – .apt_src (project-level; in every web and util project, the .apt_src folder should be excluded)
   – .xbean_src and .xbean_bin (project-level; only for projects with XMLBeans Builder enabled)

   The following directories and files should be included in source control. (Not all of these directories and files will exist in every project.):

   – .settings/* (project-level)
   – .classpath (project-level)
   – .factorypath (project-level)
   – .project (project-level)
   – .datasync-project (project-level)

   When you try to use this project from another machine, perform the following steps,

   – Open an empty workspace.
   – Import the Application by right clicking and selecting **Import > Existing projects into workspace**.
   – Select the workspace (which is the top folder), all project folders would be listed in the dialog-box.

   - Select the required folders and click **OK**, all folders will be imported, and then start building.

     Note: Occasionally you might see some errors such as the library module reference: `beehive-controls-1.0` is on the classpath of a dependent project, but it is not included in this EAR project. In this case, clean and build the project again. If it does not work, switch to the same workspace and build to resolve the errors.

6. During the development process, it is quite possible that you may have to transfer resources from one developer's system to another developer's system. For such transfers, you make a portable ZIP file. If you are making a Portable Workspace ZIP file (contains workspace and project) manually or through an Ant task, make sure to exclude these directories:

   - .metadata (workspace-level)
   - build (project-level folder; in every web and util project, the build folder should be excluded)
   - templib (project-level; in every web and util project, the temp lib folder, if present, should be excluded)
   - .apt_src (project-level: in every web and util project, the .apt_src folder should be excluded)
   - .xbean_src and .xbean_bin (project-level; only for projects with XMLBeans Builder enabled)

   If you are making a portable ZIP file with Workshop for WebLogic, select **File > Export > Archive file > Next**. In the left-hand pane, select the projects you want to include, but unselect the following directories within each project:

   - build
   - templib
   - .apt_src
   - .xbean_src and .xbean_bin (only for projects with XMLBeans Builder enabled)

   To retain the original directory structure when your workspace has multiple projects, make sure to place a checkmark next to **Create directory structure for files**.

   To uncompress the ZIP file and use the workspace, select **File > Import > Existing Projects into Workspace**, then select **Archive file** option and provide the ZIP file.

Guidelines During Application Development Stage

1. Occasionally, the XML Validator and WSDL validator associated with WTP might show errors on the XMLs/WSDLs available in the projects. Clear the XML Validator and WSDL validator check boxes on WLI-enabled utility project and Web project to disable them.

2. Clicking on the **Transformation > Create Transformation** option in node editors generates a transformation file with a default name (for example, RequestQuoteTransformation.java) and a method with a default name (for example, availProcessorGetAvail). Such transformation files are difficult to share as they have system generated names. If you, want to edit the system generated names, ensure that all references to them are modified accordingly.

   In a node editor, each time you edit the transformation, by modifying its inputs or outputs, a new Transformation method with a new signature and a new XQ file is generated. If you intend to reuse these transformations across processes, it is good practice to develop a separate transformation file containing all node editor-related transformations, then, you can reuse the transformation methods through the **Advanced Options** dialog in the node editor.

3. File control is based on DynamicProperties.xsd schema (It is a part of the system schemas), when you use file control and APIs, these schema will exist in the Work Space. In some scenarios, while creating your business process application, you have not selected the Select Add WebLogic Integration System and Control Schemas in Utility Project check box, the DynamicProperties.xsd schema will not be added to your application, and when you create a transformation in the File Control Node, using File Control and accessing FileControl Properties APIs. it leads to the IDE. To fix the problem, copy the DynamicProperties.xsd schema from <Home Directory>:\bea\wlserver_10.0\eclipse\plugins\com.bea.wli.ide.jpd_1.0.0\templates\wli_newprocess\default_schemas_project\system location and place it to the Utility Project/schemas folder.

4. If some annotation of a particular node is not visible in the Annotation View even after selecting that node in the Design View, try selecting the source code of that node in the source view. This should reveal the expected annotation in the annotation table.

5. To generate Dynamic Selector XQuery on the Service Broker control, right click on the Process file in the Package Explorer pane, select Generate > Service Broker Control, then click the Query Builder Button. If you try to create a Service Broker Control in other ways such as generating it from a WSDL, configuring dynamic selector will not work.

6. Try not to open more than one instance of BEA Workshop for WebLogic Platform.

7. Try using XQuery transformations instead of relying on the XMLBeans API for simple transformations. For example if you want to retrieve EmpID from an EMP XmlBean, instead of using `getEmpID()` get method on that XML Bean object, create a simple XQuery transformation, using the node editor, for extracting EmpID from EMP.
8. If your application is transformation centric, processing large (ie. ~1 MB) schemas files, the XMLBeans Builder can result in unacceptably long build times due to the performance of both the XMLBeans compiler and the Eclipse Java builder. XMLBeans compiler performance can be improved by disabling assertions for the XMLBeans code. Assertions can be disabled by adding the line "`-da:org.apache.xmlbeans`" to the file workSpaceStudio_1.1\workSpaceStudio\workSpaceStudio.ini.
9. There is a known issue when you drag and drop custom methods from the Task Control into the Process Canvas. The suggestion is to go to **Source** view and change `eventSet = tmp.TaskC.Callback`.class to `eventSet = tmp.TaskC.CustomCallback`.class in the `EventHandler` annotation.

Guidelines While Building an Application

1. In the WLI IDE 9.2, the Build process has two modes – Auto and Manual. In Auto Build mode, resources (Projects, folders, files) are built as they are changed and saved. The default option is **Build automatically**. To improve performance you can switch off this option. To disable **Build automatically**, select **Project** tab on your menu bar, and from the drop-down list clear **Build automatically**.
2. It is a good practice to perform a clean build operation once you have completed your development work or if you have build-related problems.
3. Be sure to save your project before you build an application, changes are not saved by default.
4. Closing projects which are not of immediate relevance may improve build performance.

Recommended Reading

For more information, refer to the following documents:

- Introducing BEA WebLogic Integration
- Guide to Building Business Process
- Using Integration Controls
- Guide to Data Transformation

- Using Worklist

See edocs, for more WLI related information.

Appendix: Analyzing Use Cases and Requirements

You can use the following SMART criteria to evaluate functional and non-functional requirements:

- Specific - Is the requirement unambiguous, with consistent terminology, simple, and at the appropriate level of detail?
- Measurable - Is it possible to verify that this requirement has been met? What tests must be performed, or what criteria must be satisfied to verify that the requirement is met?
- Attainable - What is your professional judgment of the technical feasibility of the requirement?
- Realistic - Is the requirement realistic, given the resources? Do you have adequate staff? Do you have the skills? Do you have access to the requisite development infrastructure? Do you have access to the required runtime infrastructure? Do you have enough time?
- Traceable - Is the requirement linked from its conception through its specification to its subsequent design, implementation, and test?

When in a solution domain, use robustness analysis to categorize your use cases.

You can divide a use case into four kinds of objects using robustness analysis:

- Actors - Objects external to use cases. Actors could be human or other external objects such as systems, applications, or devices. Actors interact with the use case by sending or receiving messages.

- Boundary objects - The public face of the use case. Actors interact with use cases using boundary objects. The user interface element or the public API of the use case are examples of boundary objects.
- Entity objects - The objects with long lives in the use case. Examples of entity objects are purchase order, invoice, and customer.
- Controller objects - The glue between boundary elements and entity objects. They contain methods for specific functions in a use case such as Validate user, Create, Retrieve, Update, and Delete (CRUD) functions.

Figure A-1 shows how the use case is divided into objects.

Figure A-1 Use Case Realization

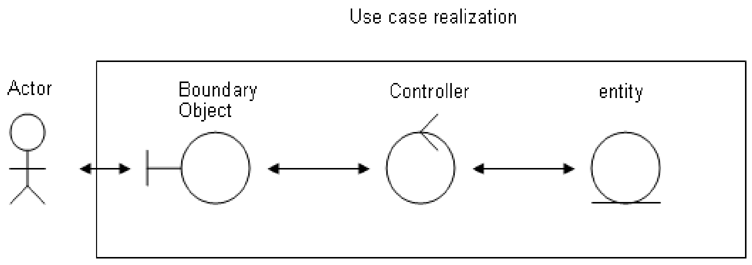
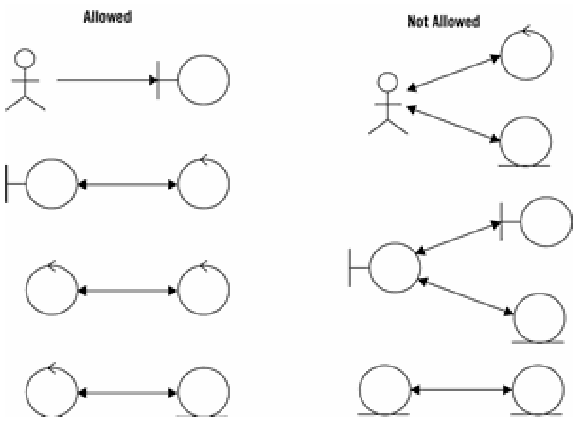


Figure A-2 illustrates the rules of robustness analysis.

Figure A-2 Rules of Robustness Analysis



The rules are as follows:

- You can have one or more actors, boundary objects, controllers, and entity objects in a use case.
- Actors can only talk to boundary objects.
- Boundary objects can only talk to controllers and actors.
- Entity objects can only talk to controllers.
- Controllers can talk to boundary and entity objects, other controllers, but not to actors.

Note: Dr. Ivar Jacobson developed the use case and robustness analysis technique. He is well known as the father of use cases and was also one of the authors of the original UML specifications.