Interstage
Business Process Manager
V11.0

Developer's Guide

J2U3-0062-02ENZ0(00)
March 2010

Publication Date	March 2010
Revision	3

Table of Contents

About this Manual

This manual describes how to use the Model API to customize and extend Interstage Business Process Manager to fit the unique needs of your organization.

Intended Audience

This manual should be read by:

- IT professionals who are interested in the development of business processes
- Independent Software Vendors who are interested in empowering their products with workflow
- Systems integrators who are interested in building workflow-enabled applications or extending existing applications with workflow

Note: Any occurance of 'administrators' in this guide refers to 'Interstage BPM Tenant Owners' and **not** 'Interstage BPM Super Users'.

This Manual Contains

Here is a list of what is in this manual:

Chapter	Title	Description
1	Introduction	Introduction to Interstage Business Process Manager.
2	Architecture Overview	Description of the interaction between the system components.
3	Concepts	An overview of the structure and operation of process definitions, process instances, work items, filters, process editing, subprocesses, etc.
4	Using the Model API	Describes the system environment for application development with the Model API.
5	Designing Process Definitions	Programming examples for simple and complex process definitions, as well as for process instances.
6	Enhancing Interstage Business Process Manager	Description of the various means of enhancing Interstage Business Process Manager, including Java Actions and Advanced Filtering & Sorting.
7	Administration	Programming examples for administrative activities.

Chapter	Title	Description
8	Retrieving History Information	Describes how to retrieve the history information of a workflow from the database.
Appendix A	Using the Interstage Business Process Manager Samples	Lists the samples provided with Interstage Business Process Manager, their structure and how to use them.
Appendix B	Customizing Forms	Describes how to customize QuickForms.
Appendix C	Supported JavaScript Functions	Lists and describes the JavaScript functions supported by Interstage Business Process Manager.
Appendix D	Troubleshooting	Describes the Interstage Business Process Manager log files and specific error situations and provides troubleshooting information.
	Glossary	Glossary of terms.

Typographical Conventions

The following conventions are used throughout this manual:

Example	Meaning
`command`	Text, which you are required to type at a command line, is identified by `Courier` font.
screen text	Text, which is visible in the user interface, is **bold**.
Reference	Reference material is in *italics*.
`Parameter`	A command parameter is identified by `Courier` font.

Other References

The following references for Interstage Business Process Manager are also available:

- *Release Notes*

  Contains an overview of Interstage Business Process Manager, installation tips, and late-breaking information that could not make it into the manuals.

- *Interstage Business Process Manager Server and Console Installation Guide*

  Describes software and hardware requirements, installation procedure for Interstage Business Process Manager Server and Console

- *Interstage Business Process Manager Server Administration Guide*

Explains how to configure and administrate Interstage Business Process Manager Server. This guide also describes the configuration parameters of the Interstage BPM Server.

- *Interstage Business Process Manager Studio User's Guide*

  Explains how to model processes using the Interstage Business Process Manager Studio.
- *Interstage Business Process Manager Tenant Management Console Online Help*

  Explains how to use the Interstage Business Process Manager Tenant Management Console user interface.
- *Interstage Business Process Manager Console Online Help*

  Explains how to use the Interstage Business Process Manager Console user interface.
- *Interstage Business Process Manager ARIS Process Performance Manager Integration Guide*

  Describes how to install and configure the PPM adapter and the PPM autoConfig tool. With both programs, process data can be transferred from Interstage Business Process Manager to ARIS Process Performance Manager.
- *API Javadoc Documentation*

  This HTML documentation provides the API and syntax of the packages, interfaces and classes for developing custom applications or embedding Interstage Business Process Manager into other products.

Abbreviations

The products described in this manual are abbreviated as follows:

- "ARIS Process Performance Manager" is abbreviated as "ARIS PPM".
- "Interstage Business Process Manager" is abbreviated as "Interstage BPM".
- "Microsoft® Windows® 2000 Server" is abbreviated as "Windows® 2000 Server".
- "Microsoft® Windows Server® 2003" is abbreviated as "Windows Server® 2003".
- "Solaris ™ Operating System" is abbreviated as "Solaris".

1 Introduction

Interstage Business Process Manager™ is a server-based workflow engine with APIs (Application Programming Interfaces) for workflow application development. It empowers developers or systems engineers to embed a workflow engine into their own products or systems which implement Interstage Business Process Manager. The target users of Interstage Business Process Manager are therefore system integrators and software product vendors.

1.1 Overview of Workflow

Workflow is a relatively new approach to automating business processes. The earliest attempts to automate business processes incorporated the logic for the flow of work in specialized applications. These specialized applications, however, were not easily updated every time the process changed. This typically resulted in a maintenance nightmare.

Workflow products and middleware were introduced to address this problem by encapsulating all the aspects of a process. This includes process-defining information, such as rules, routing paths, activities, data. More importantly, it includes the automation of the management of this information. An application that supports workflow could then implement any workflow without requiring intimate knowledge of the workflow process itself.

1.2 Overview of Interstage Business Process Manager

Interstage Business Process Manager (BPM) is a distributed Web-enabled workflow application development tool. It empowers business groups to collaboratively plan, automate, track, and improve business processes. More specifically, it allows an enterprise to design automated processes, which encapsulate the steps in the process, the responsible person(s) for each step, the order in which steps may take place, and the relevant data for the step or process.

Some key features include:

- Model API, which allows customized applications to communicate with the Interstage BPM Server or existing products to be workflow-enabled.
- Enterprise-wide, scalable infrastructure for handling processes of all types.
- Organizable and filterable universal to-do list.
- Central location for documents relevant to a process.

1.3 Advantages of Interstage Business Process Manager's Design

The philosophy behind Interstage Business Process Manager is that an automated workflow system should minimize its users' involvement with the mundane and maximize their opportunities to use their creative capabilities. For instance, it enables a customer service or sales person to focus more effectively on servicing customers and less on processing the orders, complaints, or other issues.

In engineering, business development and R&D applications, the flexibility of the Interstage Business Process Manager allows it to monitor and enhance the creative efforts of an entire team, department, division or company. By particularizing the aspects of group creativity, the abilities and energies of each individual in the team can be optimally engaged.

To best achieve these goals, it allows you to extend the functionality of the clients and classes from which the product is built. This allows your organization to incorporate context-sensitive information in your implementation as well as customizing the application to the environment in which the work takes place.

1.4 Types of Workflow

There are the following industry-accepted types of workflow products: Production, Administrative, Ad Hoc, Process Discovery, Case Handling, Collaborative, and Component. Interstage Business Process Manager supports all of these types of workflow.

Production Workflow

Production Workflow was the first workflow technology to be designed because production workflow automates the core business processes that a company relies on to produce a profit. Production workflow ensures that even at high volumes, the business process reliably follows stringent predetermined rules that never change, controlling and monitoring that does specific tasks at what time and exactly what data they have access to.

An insurance company, for instance, might use production workflow to manage the work of numerous agents upgrading and selling a large variety of policies, and constantly processing claims. Production workflow can be used wherever sales, purchasing, manufacturing, accounting, and other core business processes require extensive, enterprise-wide coordination.

The earliest workflow products were software applications that were reengineered for specific industry needs. Unfortunately, since the workflow process was hard-coded, whenever the process changed, the changes also needed to be hard-coded into the application. Programmers therefore found it difficult to keep up with the evolution of the business. The development cycle, from program design and creating new code through testing and implementation, was so long that the workflow system sometimes became obsolete as soon as it was completed.

Administrative Workflow

Administrative Workflow is a low volume subtype of production workflow, used to ensure that regular business practices stringently adhere to company policy. Examples of administrative workflow include purchase orders, travel expense reports and reimbursements, vacation requests, and similar personnel and administrative functions.

Interstage Business Process Manager is capable of managing the stringent requirements of production or administrative workflow, and at the same time allows you to make changes almost instantly when the business requirements change.

Ad Hoc Workflow

Ad Hoc Workflow is designed as an unstructured, free form workflow, which puts few, if any, constraints on the process. A key feature required for ad hoc workflow is the ability to modify or alter a running process.

Ad hoc workflow can be used as a planning tool in which the user might only instantiate a process definition once, because the process is never the same from one time to the next. A company would use workflow in this way if it's impossible to predict how the end of a project will look or if it's only possible to partially specify where it's headed. In other words, as a user, you would create a partial process definition, run the process instance, and extend it as you go along.

A pharmaceutical company might use ad hoc project management workflow while testing a new drug to satisfy conditions set by the FDA (Food and Drug Administration in the U.S.) or by a software development company while evolving a new product. These are both situations where the project's requirements are unpredictable and undeterminable. As the company acquires experience, however, it would build a knowledge base, and the process would become better defined; however, the nature of such situations may be that each process is unique and constantly mutable.

Process Discovery Workflow

Process Discovery is a type of workflow that has a goal of defining a repeatable workflow system. Process discovery uses workflow to record, organize, and refine existing business processes while they are occurring. Process discovery is appropriate in those situations where individuals have evolved their job description by the seat of their pants. It is also useful in situations where no procedure yet exists but where one is being created and refined on the fly.

Although process discovery uses a form of ad hoc workflow, the processes that are discovered and defined through the use of workflow can later become production processes. Thus, the use of workflow for process discovery enables the existing work of a company to be automated without stopping the existing flow of production.

Interstage Business Process Manager provides both subprocess and process editing capabilities, which enables users to use it for both ad hoc workflow and process discovery. Both of these types of workflow can lead to highly organized systems of dealing with unanticipated business events, evaluating the effectiveness of improvised responses and creating effective procedures from what works.

These capabilities also make it a good platform for "growing" complex case handling workflow, described below. Case handling workflow tends to involve a multifaceted set of variables, which must be handled both in a modular manner and as a single, unified case.

Case Handling Workflow

Case Handling Workflow is a subtype of production workflow. It is used to assist in complex decision-making situations where each case might be completely unique, and yet the entire case needs to be handled in the correct order and with due consideration of all dependencies within the system.

An example of case handling might be the workflow employed by a law office that specializes in divorce. Every divorce is different, yet there are many different common aspects, which might arise: division of assets, the house, the children, retirement funds, spousal support, and spousal protection. Such a complex set of details can easily become an unmanageable nightmare that is difficult to understand or handle properly.

By using subprocesses to break down the overall process, each component of such a process definition might be designed as a module, which is linked together through a master process definition.

Another example might be using workflow for case handling in a tax consultant business. When a corporation considers expansion, buying a smaller company, or moving to a new state, for instance, it needs to evaluate the impact of taxes fully. There are a huge number of factors and dependencies involved in such a decision making practice, including different types of taxes for real estate, tax structures around R&D, state laws, and corporate restrictions. Managing this amount of complexity would involve a carefully sequenced hierarchy of subprocesses so that all of the required information is available at every step of the process.

Collaborative Workflow

Collaborative Workflow has two meanings. The most common meaning is a system in which the users must continuously see proactively updated information in real time. In addition, fat clients are typically used to provide complete information to whoever happens to be available to take the case. An example of a department that might use this type of workflow is a customer support center.

Collaborative workflow can also refer to a form of workflow that might be used for project management of a large and complicated process, in which you might plan your team's activities for the next two months, evaluating and modifying your plans periodically along the way. Using workflow software would enable you to efficiently share data, review documents in an organized fashion, and process requests for approval by linking software components such as a word processor, spreadsheet program, and electronic mail program to enable participants to work on shared data.

Component Workflow

Component Workflow is workflow that drives software and automated processes without human intervention. Many of the nodes in this type of workflow might be Conditional Nodes, which invoke human interaction only when a business exception occurs. But so long as everything is moving along smoothly, the processes are entirely automated.

You might think of component workflow as one of your goals in your business production processes. You might design a labor-intensive business process for the moment, and whenever new technology appears that can automate steps in the process, you simply change a few nodes in your process definition to integrate the features of your new automation into the process as it already exists. Interstage Business Process Manager provides the environment that makes this form of gradual automation straightforward and orderly.

Open Architecture

Interstage Business Process Manager uses an open architecture to support all of these types of workflow. This allows a company to create an enterprise-wide system, which coordinates the work of the company end to end. Since most companies have different departments and responsibility groups with diverse modus operandi, it is a tool that can unite the different types of workflow needed throughout.

Interstage Business Process Manager supports various operating system platforms, includes a Model API for applications to speak to the Interstage BPM Server with, supports RDBMS databases, and allows routing of any kind of work (forms, images, and executables). In addition, its flexible architecture is designed to facilitate changes to a workflow dynamically and to support integration with an existing infrastructure.

1.5 Creating Your Own Application

The purpose of this guide is to enable you to embed Interstage BPM into your own products or systems and extend delivered Interstage BPM EJBs. To do this most effectively and practically, we recommend that you first consult the examples provided in the `client/samples` subdirectory of your Interstage BPM Server installation directory and consider what is built into the sample classes, and that you create a strategy for extending those capabilities in the application that you design. You should start with the specific capabilities and orientation of your end users and tailor the application to them.

Here are the basic possibilities that you can work with:

- You can create your own Java GUI using the Model API. Documentation for this API can be found in the `javadocs` subdirectory of your Interstage BPM Server installation directory.
- You can extend the functioning of the system by using Java Actions that can automatically manipulate the data and make decisions. Thereafter, less user interaction is needed to maintain the flow of information through your organization.

2 Architecture Overview

Interstage Business Process Manager (Interstage BPM) is a server-based workflow engine with APIs (Application Programming Interfaces) for workflow application development. It empowers developers or systems engineers to embed a workflow engine into their own products or systems which implement Interstage BPM.

Some key features of Interstage BPM include:

- API, which allows customized applications to communicate with the workflow engine or existing products to be workflow-enabled.
- Enterprise-wide, scalable infrastructure for handling processes of all types
- Organizable and filterable universal to-do list
- Central location for documents relevant to a process

Interstage BPM can run on multiple application servers providing load balancing and failover capabilities for non-stop operation with 100% reliability. Therefore, Interstage BPM is ideally suited for large mission critical applications deployed on the leading J2EE-compliant application servers. Refer to section *System Architecture* on page 19 for more information.

Interstage BPM can be used together with the following integration components:

- **Interstage BPM Analytics**: The Interstage BPM server can provide business process monitoring data to the Interstage Analytics engine. Using Interstage Analytics, this data can be evaluated and processed. Refer to section *Interstage Analytics Integration* on page 22 for more information.
- **ARIS Process Performance Manager Integration**: For analyzing the process data from the Interstage BPM Server, ARIS Process Performance Manager (ARIS PPM) can be used. With the PPM adapter and the PPM autoConfig tool the process data from Interstage BPM can be transferred to ARIS PPM. Refer to section *ARIS Process Performance Manager Integration* on page 23 more information.
- **CentraSite Integration**: CentraSite provides storage infrastructure for Web Services registries, Meta model repositories, as well as data necessary for auditing, access security and versioning. Refer to section *CentraSite Integration* on page 25 for more information. In fact, Interstage BPM can be used together with any UDDI- and WebDAV-compliant repository.

2.1 Configuration Overview

Interstage BPM can be run in the following configuration:

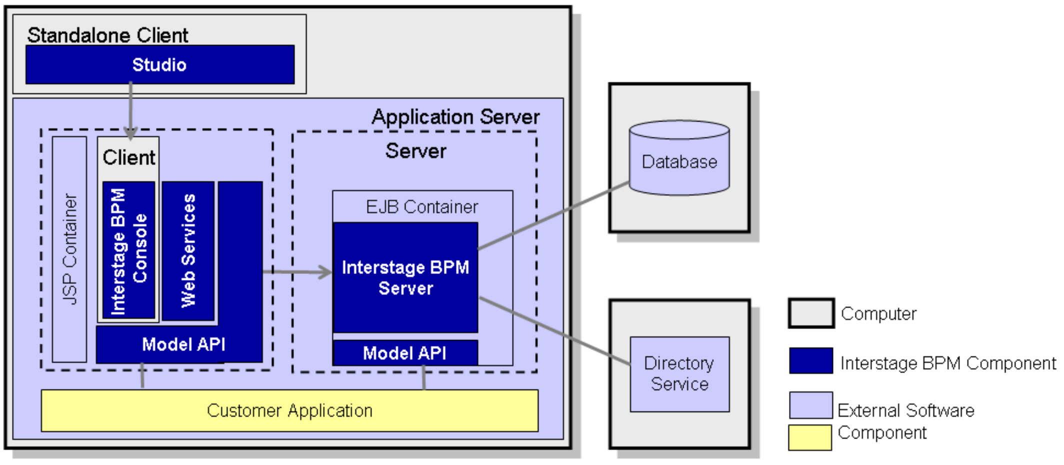


Figure 1: Configuration Overview

The Interstage BPM Server operates with a Database and optionally a Directory Service. The Interstage BPM Clients are used to access the Interstage BPM Server.

You can install the components that make up a complete Interstage BPM installation in various configurations:

- All systems are installed one and the same computer
- One or several of the following is installed on separate computers:
  - Interstage BPM Server
  - Database
  - Directory Service
  - Interstage BPM Console
  - Studio

Refer to the *Interstage Business Process Manager Server and Console Installation Guide* for details.

2.2 System Architecture

Interstage BPM basically consists of a Server and a Model API. Several connectivity options allow for the integration of third party tools and other systems. This section provides an overview of the Interstage BPM components and their interaction.

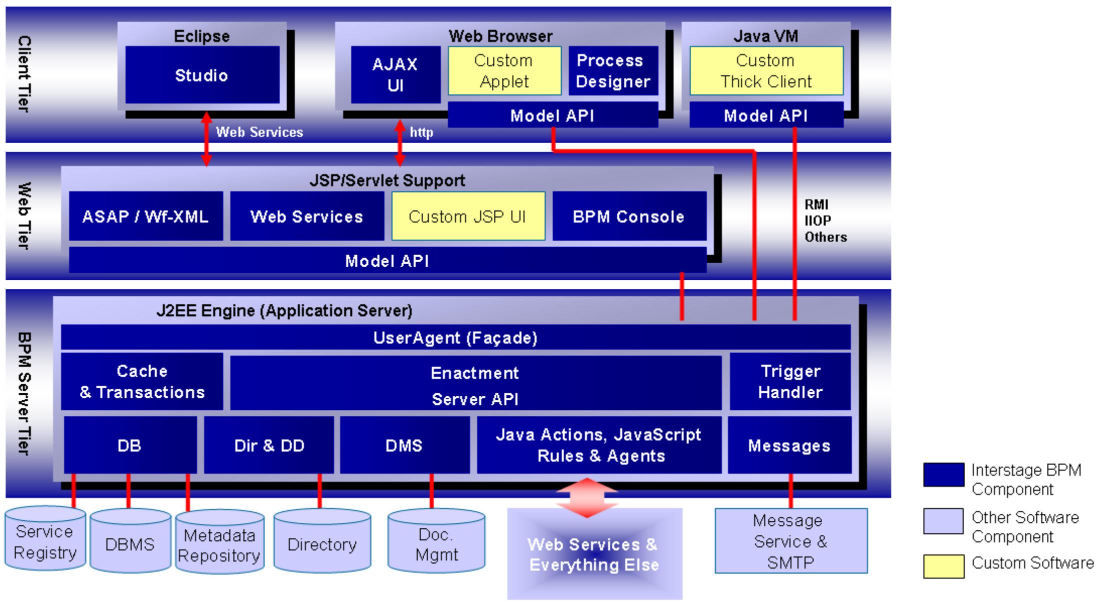


Figure 2: Architecture Overview

2.2.1 Interstage BPM Server Tier

The Interstage BPM Server is running inside an application server providing an Enterprise Java Bean (EJB) interface. The server negotiates interaction between users and other components, enacts processes started by users, and notifies users of changes in status within a process. Interstage BPM can be configured using the standard capabilities of the application server. The only way to the server is through a client developed using the Model API.

Interstage BPM is deployed into a standard application server that isolates the application from operating system and database differences. Interstage BPM utilizes the facilities of the application server to provide, for example, for clustering, load balancing and failover capabilities.

The server is composed as a collection of EJBs that run in an application server, and make use of application server functionality. The Interstage BPM EJBs participate in container transactions so that the server and any client application can participate in the same transactions. Container-based transactions ensure a consistent state of the server.

The subsequent sections describe the EJBs in more detail.

User Agent (Façade)

The User Agent EJB (UA bean) enables the client to “log in” to the Interstage BPM system and validates the client for further interaction with Interstage BPM. First, the client requests a UA bean to be created; next the server creates the UA bean and returns a handle to it to the client. Then the client provides user name, password and server name for login to the server; this information is validated either through Interstage BPM's local user management capabilities or using the functionality of a connected Directory Service.

A UserAgent instance represents the login session to the server. It holds information for that particular login session. As the client's agent, the UserAgent makes bean requests and method calls to the other Interstage BPM components on behalf of its client, i.e. it acts as a gateway for the model to access the process definition, process instance, work item, directory, and other objects. Therefore, the UA bean is also referred to as **Façade**, representing the interface between the server and the model.

Another function of the UA bean is to interpret the various filters on process definition, process instance and work item objects. Upon a client's option to log out, the UserAgent will do all the necessary cleanup of resources held on behalf of the associated client. In addition, the UA bean implements session synchronization.

The server requests a unique instance of the UA bean for each client that logs into Interstage BPM. It requests these instances from the EJB container that is part of the application server. In a manner of speaking, the Server bean is a factory that "produces" instances of UA beans.

Enactment Engine - Process Definition Interpreter

The Process Definition Interpreter is the heart of the Interstage BPM Server. It is responsible for enacting a process defined with Interstage BPM. The server communicates with the Database adapter to maintain process state data, process instance and activity-related data, and process history information. The server controls database request queues.

There are two types of entity beans representing data objects holding the information about process definitions and instances. Upon enactment of a process, a Process Definition EJB is created, which in turn creates Process Instance beans. Both beans implement the application server functionality of Bean-Managed-Persistence and Container Transactions.

They are not exposed to the model. All requests to the model pass through the UA bean(s).

Process data, e.g. information about the current state, is stored in and, on request, retrieved from the database. The server communicates with the database to maintain process state data, process and activity-relevant data.

Messaging

A combination of message-driven beans (MDBs) and a Java class library implement the Interstage BPM's type system (Meta model). Process enactment events are encapsulated in JMS messages that the MDBs process. Interstage BPM makes use of the default application server functionality.

In Interstage BPM, message-driven beans (MDBs) realize the flow of information between the server components by means of asynchronous messages.

For example, there are the following MDBs:

- Enactment Message bean: When a process instance is created, this bean generates a message so that the client is informed about this.
- Email Dispatcher bean: Handles email messages to the client
- Action Agent bean: handles Action Agents

Custom EJBs

Any application can implement EJBs that run on the same application server installation as the Interstage BPM Server. Custom EJBs use the Model API, and Interstage BPM EJBs can call custom EJBs using Java Actions.

User and Group Management

Every user that is to work with Interstage BPM needs a user account and must be assigned to one or more groups. Groups are used to determine who is responsible for carrying out a task in a process.

Interstage BPM comes with its own user and group management capabilities. Interstage BPM also allows for connecting to a Directory Service. Depending on your choices when deploying the server, users are managed either in Interstage BPM's local user store or in a Directory Service. Groups can be managed in Interstage BPM's local group store, in a Directory Service or in both systems.

Connectivity

The Interstage BPM architecture allows for the integration with third-party products. The server can communicate with the other components via "adapter classes". An adapter behaves as a converter that allows the server to speak to a common interface. Interstage BPM allows for connecting to the following:

- **Database (DB)** adapter using the JDBC standard. The server provides the communication mechanism between the server and a database server. The database persistently stores and maintains all process information. The DB Adapter is responsible for the translation of server internal objects into various persistent formats. Included with Interstage BPM is an adapter that persists the structures in a relational database using JDBC and some stored procedures. Initialization scripts are available to configure Oracle, Microsoft SQL Server for use.
- **Directory Service (Dir & DD)** adapter which implements an Interstage BPM specific interface to expand a user group into a list of individuals. The enactment engine uses this at runtime to determine who to give work items to. The Directory Adapter uses the LDAP standard. Currently, Microsoft® Active Directory and Sun Java System Directory Server is supported.

  The DD Framework Adapter is used by the User Agent at login time to authenticate users.
- **DMS** adapter which is used to interface the Interstage BPM system to external file systems using standard copy and transfer protocols. Forms, attachments, process definitions, etc. can be stored in a file system. A locator for such documents is stored in the attachments attributes of a process instance. Interstage BPM includes a DMS adapter that accessed documents stored on a file system or on a file store accessible via the WebDAV protocol. Access to other document management systems requires a custom DMS Adapter.
- **Messages** using the SMTP standard. Email can be sent from the server to the SMTP mail server as a response to Interstage BPM events.
- **External Systems**:
  - **Java Actions and JavaScript**: You can implement Java Actions for connecting to any external system, such as CRM or ERP systems. Java Actions are extensions to the workflow engine.

    Java Actions are data structures in the process definition that tell the Interstage BPM Server how to call a particular Java method during execution. These method calls customize process enactment and allow execution of Java business methods outside the scope of the Interstage BPM Server. Java Actions make application integration easier and calls to external applications and adapters faster.
  - **Agents**: Agents in Interstage BPM are set up to run automatically and act asynchronously on your behalf. You can use Agents to access external systems such as legacy systems or Web Services, both inside and outside of company firewalls. Using Agents, you can incorporate these external services into your Interstage BPM process instances. This mode of integrating Java is particulary convenient when multiple retries may be required.

- **Rules**: The Rules Engine Bridge is a Java Action that is included in Interstage BPM in order to invoke rules engines like the iLog JRules Engine. Rules have all the same capabilities that are available from JavaScript, so you can think of the rules as a kind of scripting engine.

2.2.2 Interstage BPM Web and Client Tier

The **Model API** is an abstraction over the server and provides a single unified API to the server. The Model API runs in the client process, and handles all the communications to the server.

Interstage BPM comes with several client applications, for example, the Studio and the BPM Console.

Except for the Studio, the clients run in a servlet engine and are accessed using a Web browser. They are comprised of a combination of Java User Interface classes. Such client components are structured in two layers: a model layer (using the Model API) and a user interface layer (using Java User Interface classes). The model layer encapsulates the state of the client objects and interacts with the server. With using the Model API, you can develop your own clients and user applications.

The Studio is a standalone process design tool that can be installed separately. The Studio is independent of application server functionality. It interacts with the Interstage BPM Server through the Web tier.

Web Service capabilities is included in the web tier. This functionality can receive a number of defined SOAP (Simple Object Access Protocol) requests, and respond with XML-formatted results. Most of the common Interstage BPM operations are accessible through this port.

Interstage BPM supports another kind of Web Services Interface that is known as an Asynchronous Web Services Interface. This is an implementation of a standard way to access process instances and other long running programs. The standard is known as the Asynchronous Service Access Protocol (ASAP).

Refer to the *Interstage Business Process Manager Server and Console Installation Guide* for details on the Interstage BPM Web Services.

2.3 Interstage Analytics Integration

Interstage Analytics can be used for analyzing and evaluating the process data from the Interstage BPM Server. The Interstage BPM Deployment Tool allows for configuring whether Interstage BPM generates events to Interstage BPM Database that can then be evaluated by Interstage Analytics.

The following figure illustrates the interaction of the applications involved.

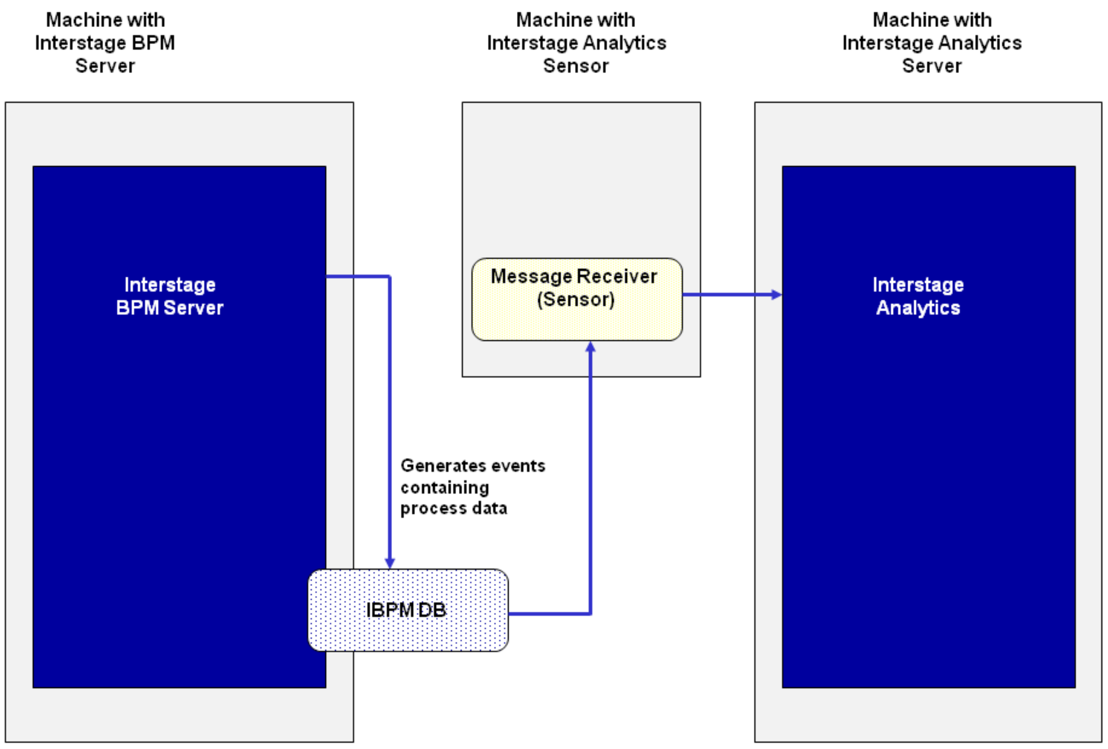

Machine with Interstage BPM Server
Machine with Interstage Analytics Sensor
Machine with Interstage Analytics Server
Interstage BPM Server
Message Receiver (Sensor)
Interstage Analytics
Generates events containing process data
IBPM DB

Figure 3: Interstage Analytics Integration

The Interstage BPM Server, the Interstage Analytics Sensor and the Interstage Analytics Server can be installed on the same physical machine, or on different machines on the same network.

Refer to the *Interstage Business Process Manager Server and Console Installation Guide* for information on how to configure the sending of events during deployment.

Refer to the *Interstage Analytics documentation* for information on how to setup the Sensor and use Interstage Analytics.

2.4 ARIS Process Performance Manager Integration

For analyzing the process data from the Interstage BPM Server, ARIS Process Performance Manager (ARIS PPM) can be used. With the PPM adapter and the PPM autoConfig tool the process data from Interstage BPM can be transferred to ARIS PPM.

The following figure illustrates the interaction of the applications involved.

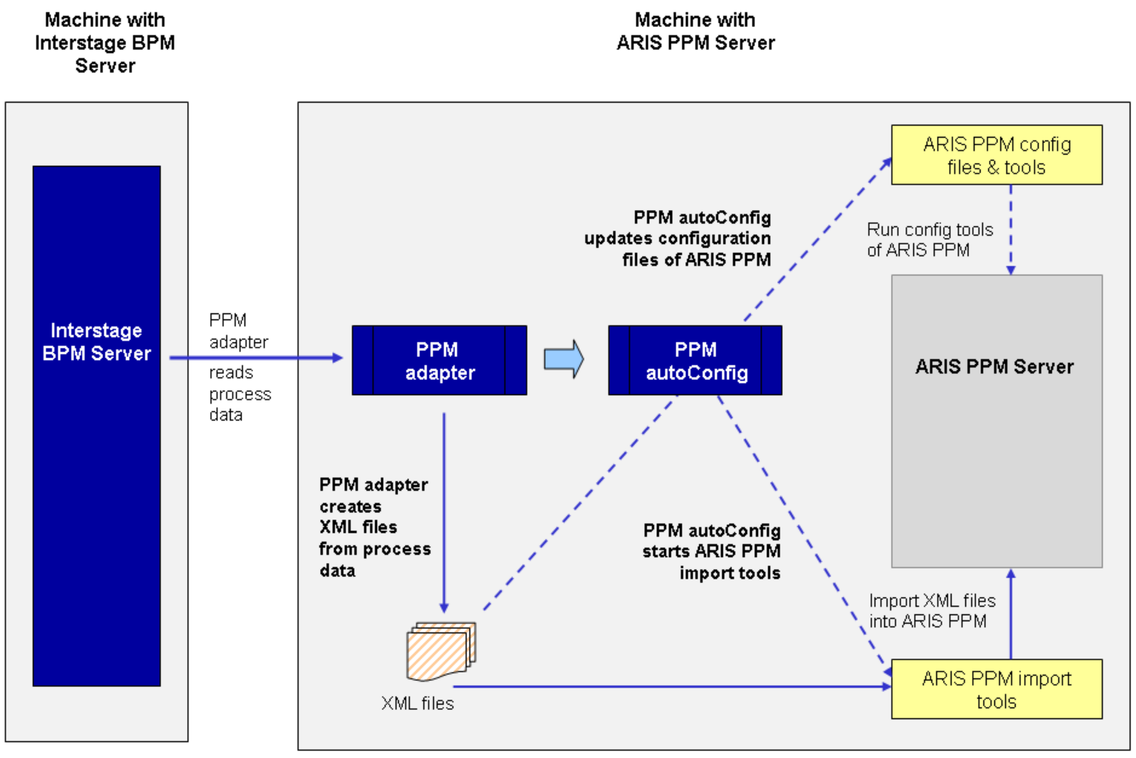

Machine with Interstage BPM Server
Machine with ARIS PPM Server
Interstage BPM Server
PPM adapter reads process data
PPM adapter
PPM autoConfig
PPM autoConfig updates configuration files of ARIS PPM
ARIS PPM config files & tools
Run config tools of ARIS PPM
ARIS PPM Server
PPM adapter creates XML files from process data
PPM autoConfig starts ARIS PPM import tools
Import XML files into ARIS PPM
XML files
ARIS PPM import tools

Figure 4: ARIS Process Performance Integration

PPM adapter

The PPM adapter reads process data from the Interstage BPM Server and exports them into an XML file in a format that meets the requirements of ARIS PPM. The process data can contain all kinds of nodes, arrows and User Defined Attributes. Process instances that are called as sub- or chained processes are also exported with all their information.

PPM autoConfig

The program PPM autoConfig is a tool for automatically extending the ARIS PPM configuration files with regard to the process instances exported by the PPM adapter. Furthermore, the PPM autoConfig tool uses ARIS PPM tools to automatically import the XML files generated by the PPM adapter into ARIS PPM.

The PPM autoConfig tool is executed after the PPM adapter successfully accomplished the export of the process instances. The execution of the PPM autoConfig tool can be done manually or from within the PPM adapter.

ARIS PPM Server

ARIS PPM is a tool for analyzing, evaluating and monitoring business process data, after this ARIS PPM generates intuitive Management Dashboards that can be used for management reporting. ARIS PPM provides a clearly and precisely display of the current performance of business processes.

Refer to the *Interstage Business Process Manager ARIS Process Performance Manager Integration Guide* for more information about the usage of the PPM adapter and the PPM autoConfig tool.

2.5 CentraSite Integration

CentraSite provides storage infrastructure for Web Services registries, Meta model repositories, as well as for data necessary for auditing, access security and versioning. In addition it provides a web-based interface to visualize reports that analyze the usage of Web Services in process instances (Interstage Business Process Manager), orchestrations (Software AG's crossvision Service Orchestrator) and information integration queries (Software AG's crossvision Information Integrator). As a result, business analysts, architects and developers can all collaborate, eliminate business risk related to change in IT assets and avoid the disruption of critical business processes.

From a functional point of view CentraSite manages metadata generated from integration software, Web Service descriptions, application specific data, and in general it serves as a central store for documents in native XML and non-XML formats.

You can use the following functionalities provided by CentraSite from Interstage BPM:

- UDDI registry
- WebDAV repository

UDDI is an industrial standard and serves the known registry functionality such as publicizing, discovering and staging consumption of Web Services. Publishing, discovering and retrieving Web Services capabilities provided by Interstage BPM is based on standard UDDI interfaces, and therefore you can use CentraSite from Interstage BPM as a UDDI registry implementation.

WebDAV is another industrial standard and can be used for storing and retrieving development artefacts, which are stored in standard formats such as XPDL. Interstage BPM provides the capability of publishing metadata into WebDAV and therefore you can use CentraSite from Interstage BPM as a WebDAV repository implementation.

2.6 Interstage BPM as a Service (SaaS)

Interstage Business Process Manager offers the option of being used in SaaS (Software as a Service) mode. If you use Interstage BPM in SaaS mode, you can create multiple tenants and lease out Interstage BPM to these tenant organizations, who will use it as a service. Note the following:

- You can decide whether or not to use Interstage BPM in SaaS mode during Interstage BPM installation.
- An organization that leases out Interstage BPM to other organizations for use as a service is called a service provider.
- An organization that uses Interstage BPM as a service from the service provider will use Interstage BPM as a 'tenant'.
- A service provider user who administrates tenants is called a Super User. Functionality of a Super User is limited to only managing tenants through the Interstage BPM Tenant Management Console, and managing the Interstage BPM Server.
    - Information about installing and accessing Interstage BPM Tenant Management Console is included in the *Interstage BPM Server and Console Installation Guide*

    - Information about using the Interstage BPM Tenant Management Console is included in the Interstage BPM Tenant Management Console Online Help
    - Information about managing the Interstage BPM Server is included in the *Interstage BPM Administration Guide*
- The Super User role cannot use or administrate Interstage BPM workflows.
- For a Service Provider to be able to use Interstage BPM as a regular user, it is required to set up a default tenant for its own use during Interstage BPM installation. Setting up a default tenant also sets up the default `System` application.
- **Even if you use Interstage BPM in the non-SaaS mode**:
    - You will still need to set up a Super User during installation. The role of this Super User will be limited to managing Interstage BPM Server.
    - You will need to set up a default tenant during installation. You will use all Interstage BPM functionality as a default tenant. You will not be allowed to create more than one tenant.
- Irrespective of whether you use Interstage BPM in SaaS mode or not, any operation on a workflow element will always be in the specific context of an application. For example, before you create a process definition or process instance, you need to choose an application. In SaaS mode, you must choose an application before operating on any element. In non-SaaS mode, selecting an application is optional.

3 Concepts

From start to finish, the developers and users of a typical workflow application may perform some or all of the following operations:

- Connect to the Interstage BPM Server
- Build a process definition
- Define and associate a form with an Activity Node
- Start a new process instance
- Modify a process instance
- Modify variables through Java Actions
- Associate an attachment with a process instance
- Execute an activity choice option
- Obtain a list of process definitions, process instances and work items
- Obtain the status of process instances and activities

The work performed by these operations can be best understood through the concepts that are discussed in this chapter.

Note: The Interstage BPM Server uses JMS messages for internal communication. JMS notification is used internally only. You cannot write external applications to receive JMS messages from the Interstage BPM Server.

3.1 Process Definitions and Process Instances

At its core, workflow is a medium of collaboration or coordination for any complex project. The basic building block of this collaboration is the process definition. Process definitions are like recipes in a cookbook: they provide you with the steps to follow, with references to the ingredients you must use, and criteria for deciding that the job is done. Like a recipe, a process definition is a static, reusable method, which guarantees a predictable result.

A process instance is the individual execution of the process definition, much like the specific act of baking a cake that you create by following a general recipe. In other words, the process instance provides a day-to-day context for coordinating the work on a project. It ensures that all of the cooks have the right ingredients at the right time, and that everyone is using those ingredients according to the agreed-upon recipe. In a business context, these ingredients generally consist of documents and data.

Since a process instance merely reflects the structure and functionality that was designed into the process definition, there is a close correspondence between the subordinate or dependent objects of process definitions and process instances. This chapter therefore focuses on process definition-related classes first to provide the background for understanding the whole system.

3.2 Process Definitions

Process definitions encapsulate all of the aspects of a process instance that can be set at design time. Aspects include predefined attributes, operations to be performed when the process instance is run, definitions for variable data that will be set at run-time, and the state of the process definition.

Process Definition Attributes

Process definition attributes include (but are not necessarily limited to) the following:

- Process definition state
- Process definition identifiers
  - ID
  - Version
  - Owner
  - Name (optional)
  - Title (optional)
  - Description (optional)
- Flow control
  - Nodes
  - Arrows
- Process definition ownership (Role)
- Variable data
  - User Defined Attributes
- Java Actions
  - Init Actions
  - Commit Actions
  - Error Actions and onSuspend, onResume, onAbort Actions
- Timers including Timer Actions

Note: These attributes are added to the process definition during its design. However, many have no specific meaning until the process definition is used to create a process instance, and that process instance is run.

3.2.1 Process Definition States

A process definition can be in one of the following states:

- Draft
- Published
- Private
- Obsolete
- Deleted

Process definitions in draft state are in the design stage. Only the owner of a draft process definition can create a process instance from it (for testing purposes). When the draft process definition is ready for general use, the process definition owner asks a administrator (tenant owner) to publish it so others can start process instances from it. If a user modifies a running process instance, a private process definition is created for just that one process instance.

Although you cannot edit a published process definition, you can make a new copy of it that is in draft state by default. This new copy of the process definition can be edited until it is published.

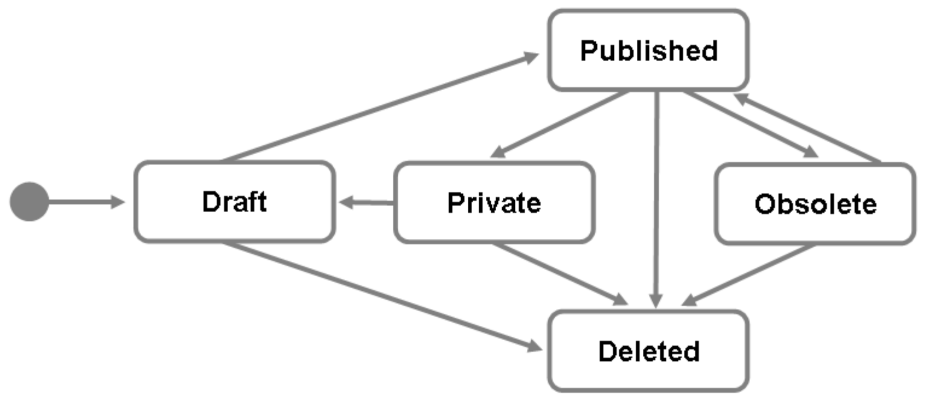


Figure 5: Process Definition States

If you want to take a published process definition out of circulation, an administrator can change the process definition state from published to obsolete. If you want to delete a process definition, an administrator can change the state to deleted. When draft and private process definitions are deleted, they are dropped from the system. When published and obsolete process definitions are deleted, they are not dropped from the system, but you can't modify their state again. You can create new versions from draft, private, published and obsolete process definitions, but not from deleted process definitions.

Note: If you want a private process definition to replace the currently used process definition going forward, simply create a new version of the private process definition. The new process definition will be in draft state with the same name and the next version number. Only the administrator (tenant owner) can change then the state of the draft process definition to published. In addition to publishing the draft process definition, this automatically changes the state of the original process definition from published to obsolete.

Each process definition family (i.e., each process definition with the same name but different version number) can only have one published process definition.

3.2.2 Process Definition Identifiers

A process definition is identified by the following attributes:

- ID
- Version
- Owner
- Name (optional)
- Title (optional)
- Description (optional)

The ID is a variable of data type Long, which is automatically assigned by the Interstage BPM Server to provide a unique handle for the process definition. The version number is automatically assigned, and the owner is the creator of the process definition.

The other three identifiers are optional. When designing an application, the `Plan` interface from the Model API gives you access to the name, title, and description if you desire.

Process Definition Versioning

New process definitions are assigned a version number of 1.0, for example `Purchase Order v1.0`. You can create a new version of a process definition without changing its name. If you copy a process definition the new version of the process definition will have the next whole number, for example `Purchase Order v2.0`.

Note: Versioning of a process is specific to the application to which it belongs. This implies you can have two processes with the same name without affecting their version number as long as the processes exist in separate applications.

3.2.3 Workflow Elements

A workflow process is a network of nodes connected by arrows. Nodes and arrows together are referred to as workflow elements.

Nodes represent the steps in a process. A step could be an activity where process participants are assigned specific tasks to be completed. A step could also be a place where a decision on the process flow is made. In this case, no user action is required.

Workflow elements have the following minimum attributes: ID, Name, Title and Description. ID is absolutely mandatory - the one attribute that all workflow elements must have. The Interstage BPM Server automatically assigns a unique ID to each element in a workflow process to identify it.

In addition to the attributes that all nodes have, specific node types have properties that are applicable to only them. In the discussion that follows, some properties are only applicable to certain individual node types - as required by their intended behavior.

Arrows are the connectors that guide the flow of the process instance from one node to another. The flow of events simply passes through the arrow on its way from one node to the next. They have little additional functionality except to change the state of their target nodes.

Arrows have a simple set of identifiers: ID, Name, and Description. The ID is a long data type that is automatically assigned by the Interstage BPM Server to provide a unique handle for the process definition.

The other two identifiers are optional. In designing an application, the `Plan` interface from the Model API gives you access to the ID, Name, and Description fields, if desired.

3.2.4 Java Actions

Process instances "live" in the Interstage BPM Server that executes them and communicate with users through whatever client is provided. Java Actions provide a mechanism through which process instances can communicate with the outside world. At its core, a Java Action is nothing more than a static Java method which Interstage BPM has been configured to know enough about to be able to call, as part of process execution.

When data is moved into a process instance, it is usually read from one or more external data sources and copied into User Defined Attributes. They are moved out of a process instance by copying the values of User Defined Attributes to one or more external data sources. Since User Defined Attributes can modify the action of certain nodes, Java Actions can do the same.

Interstage BPM offers various types of Java Actions. Java Actions are mostly defined for Nodes, some of them can be defined for entire process definitions, others for other Java Actions, for example:

- **Prologue Actions**. A Prologue Action is evaluated before the node performs its task. This Java Action can therefore be used to set up or initialize values associated with the node before it does its work.
- **Epilogue Actions**. An Epilogue Action is executed after the node finishes its task and before the process instance moves on to another node. This Java Action can therefore be used to clean up or analyze values associated with the node after the intended work is finished.
- **Role Actions**. A Role Action is evaluated after role resolution and before task assignment. This Java Action is therefore used to dynamically compute a list of assignees for a task in conjunction with the assigned user group.
- **Error Actions**. An Error Action can be used for handling specific error situations in the execution of a process. Error Actions can be defined for entire process definitions, for individual nodes and for other Java Actions, i.e. when you want to react on errors occurring during the execution of another Java Action.
- **Compensate Actions**. A Compensate Action can be defined for another Java Action that accesses a system outside of Interstage BPM, e.g. a database. Compensate Actions are useful to ensure a consistent state of all systems involved in a transaction for cleaning up and rolling back transactions, e.g. to delete a newly added row in a database.

Refer to section *Types of Java Actions* on page 91 for more information.

3.2.5 User Groups

User Groups and assignee resolution are a basic, underlying feature of the system. A User Group is a name given to a set of organizational resources. Depending on your system configuration, user groups are defined in Interstage BPM's local group store, in a Directory Service or in both systems.

Using the Directory Service functionality, a User Group can also be mapped to any other organizational resources, including machines, software programs, parameterized function calls, or even queries.

Usually, a User Group is a name given to users who are grouped according to criteria that fit the needs of the business or organization. The most common of these criteria describe the kind of work performed by one or more persons in an organization. Users can also be grouped according to their authority, responsibility, skill, or profession: The workflow engine doesn't distinguish what the organizational resources are, so long as it can be identified with a string variable.

A node can be assigned a User Group if it represents an activity that requires user involvement. In the current Interstage BPM model, only Activity Nodes and Voting Activity Nodes support this property.

3.2.6 Forms

Most business processes involve decisions and actions that are based on information. Users access this information through forms and attachments. Forms are added to process definitions at design time, and attachments are added to running process instances.

Forms can be used to display or report data that is stored in User Defined Attributes and data from external data sources. Forms can also enable users to interact with, modify, or add data to the process instance or to an external data source. Refer to section *User Defined Attributes* on page 33 for using variable data in forms.

Forms are stored on a Web server and Interstage BPM only knows the URL with which the forms can be addressed. They are associated to a node by specifying the path to the form including the form name. Forms can be associated with Start Nodes, Activity Nodes and Voting Activity Nodes.

Forms are presented to the user in starting a process instance or viewing a work item. Forms can only be accessed through work items.

3.2.7 Timers

Almost all business processes live and die according to timed criteria, including deadlines, interest costs, milestones, procrastination, project management, vacation schedules, personal organization - the list goes on forever.

The system environment allows you to create timers that trigger certain actions when they expire. These timers can be Activity-Node-Level timers that start running when the Activity Node is activated or Process-Level timers that start running whenever a new process instance is created from the process definition containing the timer.

Timers are controlled by time and action parameters, specified as User Defined Attributes, and set through Java Actions or forms. The expiration or firing time of a timer can be an absolute, specific time or relative to another event (such as the start of the activity or process instance). The timer can also be set to trigger actions periodically. Timers can also perform a variety of actions, including escalation (assigning the activity to an additional list of users) or sending emails.

The following timer types are available:

- **Relative timers**. A Relative timer calculates the time when they will expire based upon the time when the activity or process becomes active. They are only activated once.
- **Absolute timers**. An Absolute timer is set according to the absolute time when the timer will expire, measured in milliseconds since January 1, 1970, 00:00:00.
- **Periodic timers**. A Periodic timer is a Relative timer that operates repeatedly. The first occurrence of the timer is relative to the time when the activity or the process instance becomes active. Subsequent occurrences are relative to the last occurrence.
- **Business Periodic timers**. A Business Periodic timer is a Business Relative timer that operates repeatedly. The first occurrence of the timer is activated relative to the time the activity or a process instance becomes active, and subsequent occurrences are relative to the last occurrence.
- **Business Relative timers**. A Business Relative timer calculates the time that they will expire, based upon the time when the activity or a process instance becomes active, and using the 'business calendar' expression methods. They are only activated once.

The Business Periodic timers and the Business Relative timers are used within Business Calendars. For defining a timer with the Model API refer to section *Using Timers* on page 149 for details. For configuring a Business Calendar refer to the *Interstage Business Process Manager Server Administration Guide* for details.

3.2.8 Process Definition Ownership

The attributes for process ownership allow you to set permission levels in process definitions and process instances to individuals or groups. The purpose of process ownership is to identify users who are authorized to edit it structurally.

The owner of a process definition is the person who created it. When a process instance is created from the process definitions, this person is by default the owner of the process instance as well. However, process definition ownership and process instance ownership are two distinct attributes.

During process definition design, process ownership can also be directly assigned to members of a user group or programmatically assigned by using a Role Action. The Role Action interacts as described in the table below.

Set by the User Group	Set in the Role Action	The Process Owner Will Be
No	No	The process definition owner becomes the process owner.
Yes	No	The process instance owners are determined by the User Group.
No	Yes	The Role Action determines the process instance owners.
Yes	Yes	The process owners belong to the intersection between the assigned User Group and the Role Action. If this intersection is empty, the process definition owner becomes the process instance owner.

The Interstage BPM Server makes sure that users are authenticated and that their assigned Groups and their capabilities in accessing and manipulating process instances, activities and their attributes are identified. The use of Groups eliminates the need to modify every process definition when there is a personnel change. Instead, only the group definition in Interstage BPM's local group store or in the Directory Service needs to be updated.

3.2.9 User Defined Attributes

Another set of attributes assigned to the process definition consists of the User Defined Attribute definitions. These definitions describe data items that will be associated with process instances created from the process definitions. Process instance-specific values can therefore be stored in them. The values may be assigned either at the beginning or during the running of the process instance. This variable data is global to the process instance it is associated with. Any user may change it with access to it.

While designing a process definition, the designer determines what types of data will be used in the process instances and creates a strategy for assigning that data to variables.

The data can also be modified through Java Actions. Refer to section *Integrating Interstage BPM with External Applications* on page 88 for using Java Actions.

3.2.10 Node Types

Interstage BPM supports different node types. The following table gives an overview of the supported node types and their attributes. It does not contain attributes that are common to all node types.

Node Type	Assignee	Role Action	Prologue Action	Epilogue Action	onAbort, onSuspend, onResume Action	Error Action	Timers	Triggers	Forms
Start Node	-	-	-	-	-	-	-	-	Yes
Exit Node	-	-	-	-	-	-	-	-	-

Node Type	Assignee	Role Action	Prologue Action	Epilogue Action	onAbort, onSuspend, onResume Action	Error Action	Timers	Triggers	Forms
Activity Node	Yes	Yes	Yes	Yes	Yes	-	Yes	Yes	Yes
Voting Activity Node	Yes	Yes	Yes	Yes	Yes	-	Yes	-	Yes
Delay Node	-	-	Yes	Yes	Yes	-	Yes	-	-
Event Activity Node	-	-	Yes	Yes	Yes	-	Yes	Yes	-
AND Node	-	-	-	Yes	-	-	-	-	-
OR Node	-	-	-	Yes	-	-	-	-	-
Conditional Node	-	-	Yes	-	Yes	-	-	-	-
Subprocess Node	-	-	Yes	Yes	Yes	-	-	-	-
Remote Subprocess Node	-	-	Yes	Yes	Yes	Yes	-	-	-
Chained-Process Node	-	-	Yes	Yes	Yes	-	-	-	-
Compound Activity Node	Yes	Yes	Yes	Yes	Yes	-	-	-	Yes

The following sections provide a detailed discussion of the different node types.

Start Node

The Start Node identifies the beginning of a process. Every process definition has one and only one Start Node.

Not supported: This node type does not support any role assignment, timers or triggers.

Behavior:

- A process instance that is created from a process definition has to be explicitly started.
- Enactment of a process instance begins at the Start Node. When a process instance is started, an event is sent to the Start Node.
- The Start Node then immediately sends events to all the outgoing arrows.

Exit Node

An Exit Node identifies the end of a process branch.

Every process definition has at least one Exit Node. A process definition may have multiple Exit Nodes, which model different modes of completion of a process. With huge process definitions, multiple Exit Nodes can simplify the graphical representation of the process.

In a typical process definition, an Exit Node has one or more incoming arrows and no outgoing arrows.

Not supported: This node type does not support any kind of forms, role assignment, timers or triggers.

Behavior: Enactment of the process instance stops at an Exit Node. When a process instance has multiple Exit Nodes, as soon as the process instance's enactment reaches one of the Exit Nodes it is deemed to be complete.

Activity Node

An Activity Node is an interactive node that represents one or more tasks performed by humans towards completion of a business process. Upon enactment of this node, this thread of events in the process instance pauses. It only continues after human intervention.

Activity Nodes can also stand-in for activities performed by external agents (such as other devices).

A process instance may have any number of Activity Nodes. An Activity Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Forms, as many as required, can be used. The semantics of this node type support Prologue and Epilogue Actions, role assignment, timers and node-level triggers. In addition, a Java Action Set can be defined that becomes active when an Administrator aborts, suspends or resumes a process instance to which this node belongs.

Behavior:

- The list of assignees using the assigned group is computed as follows:

Group Specified	Users Specified in Java Action	Assignees
Yes	No	User group belonging to this user group or individual members of user group
Yes	Yes, list intersects with user group	Members specified in both user group and Java Action
Yes	Yes, list does not intersect with user group	Process owner
No	Yes	Members specified in Java Action

- All timers defined for this node are activated.
- One group work item or individual work items are created for the assignees.
- When one of the assignees (not all of them) completes the task associated with the Activity Node, the assignee sets the process instance rolling again by indicating that the task is complete (e.g., by clicking a button). This is referred to as committing the work item.
- Each of the outgoing arrows of this node represents a direction in which the process instance can proceed from this node. Depending on the business process, an assignee can make the process instance continue in a particular direction by choosing the arrow that represents the direction.
- Once the work item is committed, the Epilogue Action is evaluated, any process timers that are still active are cancelled, and an event is sent to the arrow that has been chosen.

Voting Activity Node

A Voting Activity Node uses voting rules to determine the choice (activity's outgoing arrow) that wins. For every Voting Activity Node the rules for voting must be specified. A voting rule specifies for one outgoing arrow the kind of controls to be used for voting, for example, TYPE_MAJORITY.

A process instance may have any number of Voting Activity Nodes. A Voting Activity Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Forms, as many as required, can be used. The semantics of this node type support Prologue and Epilogue Actions, role assignment and timers. In addition, a Java Action Set can be defined that becomes active when an Administrator aborts, suspends or resumes a process instance to which this node belongs.

Behavior:

- Upon receiving an event from any of its incoming arrows, a Voting Activity Node enters a running state. While in this state, it ignores any events that it receives from its incoming arrows. If the user group assigned to the node cannot be evaluated, the node and process instance containing it enter the error state.
- The Voting Activity Node then evaluates the Prologue Action.
- It then computes the list of assignees, using the assigned user group, as follows:

User Group Specified	Users Specified in Java Action	Assignees
Yes	No	Members of User Group
Yes	Yes, list intersects with User Group	Members specified in both User Group and Java Action
Yes	Yes, list does not intersect with User Group	Process owner
No	Yes	Members specified in Java Action

- All timers defined for this node are activated.
- Work items are created for all assignees.
- The voting rules defined for the Voting Activity Node are evaluated. A voting rule defines the kind of controls to be used for voting. The threshold of a voting rule defines the limit when the condition becomes true.
- When one of the voting rules is matched, it sets the process instance rolling again by indicating that the task is complete. This is referred to as committing the work item.
- Once the work item is committed, the Epilogue Action is evaluated, any process timers that are still active are cancelled, and an event is sent to the arrow that has been chosen.

Note: There are some custom nodes that can be used in the Interstage BPM Studio. At API level, Complex Conditional Nodes are equivalent to Conditional Nodes. Similarly, Email Nodes, DB Nodes and Web Service Nodes are equivalent to OR Nodes.

Delay Node

A Delay Node represents a step in a process in which execution of the process instance (or a particular execution thread in a process instance) is suspended for a specified amount of time.

A process definition can have any number of Delay Nodes. A Delay Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Prologue Actions, Epilogue Actions, and timers can be used as required. Although more than one timer can be specified on this node, only one is effective. In addition, a Java Action Set can be defined that becomes active when an Administrator aborts, suspends or resumes a process instance to which this node belongs. The semantics of this node type do not support forms, role assignment or triggers.

Behavior:

- When a Delay Node receives an event from an incoming arrow, it evaluates any Prologue Action, activates all the timers specified, and waits. While waiting, it ignores additional events from incoming arrows.
- As soon as one active timer expires, the node cancels all other active timers, evaluates any Epilogue Action, and sends events to all outgoing arrows.

Event Activity Node

An Event Activity Node represents a step in a process which is driven by an external event. Each Event Activity Node has a trigger defined that is supposed to fire when data, typically XML files, comes in from an external system. Once the data arrives, the trigger moves the data into User Defined Attributes (UDAs) according to the trigger's definition. The trigger then makes a choice and process execution moves forward to the next node.

This node type does not involve any human interaction. Hence, no work items are generated when a node of this type becomes active. It can only be completed by a trigger defined on the node.

A process definition can have any number of Event Activity Nodes. An Event Activity Node can have one or more incoming arrows and exactly one outgoing arrow.

Supported: An Event Activity Node must have one or more node-level triggers (make choice triggers) defined. Prologue Actions, Epilogue Actions and timers can be used as required. In addition, a Java Action Set can be defined that becomes active when an Administrator aborts, suspends or resumes a process instance to which this node belongs. Due to its semantics, this node type cannot be assigned to users, roles or agents. Also, this node type does not support Role Actions or forms.

Behavior:

- When an Event Activity Node receives an event from an incoming arrow, it evaluates any Prologue Action, activates all of its active triggers, and waits. While waiting, it ignores additional events from incoming arrows.
- As soon as one active trigger fires, the node evaluates any Epilogue Action, and sends an event to its outgoing arrow.

Note: In the Interstage BPM Studio, Event Activity Nodes are referred to as "Trigger Nodes".

AND Node

An AND Node represents a step in a process where the process instance pauses to synchronize multiple threads of execution.

A process can have any number of AND Nodes. An AND Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Epilogue Actions can be used as required. The semantics of this node do not support Prologue Actions, role assignment, forms, timers or triggers.

Behavior:

- An AND Node waits till it receives at least one event on each of its incoming arrows. Any additional events received from the same incoming arrows are ignored while waiting to receive an event from each of the arrows.
- When each of the incoming arrows has received an event, all of the execution threads represented by the incoming arrows are synchronized. Any Epilogue Action is evaluated and events are sent to all the outgoing arrows.
- Thus, this node can also result in multiple branches.

OR Node

An OR Node represents a step in a process from where a single branch of control splits into multiple parallel branches. A process can have any number of OR Nodes.

This node can have one or more incoming arrows and one or more outgoing arrows.

Supported: Epilogue Actions can be used as required. The semantics of this node type do not support Prologue Actions, role assignment, forms, timers or triggers.

Behavior: Every time this node type receives an event from one of its incoming arrows, it evaluates the Epilogue Action and sends events to all its outgoing arrows.

Conditional Node

A Conditional Node represents a step in a process where the process's execution proceeds in one of the many possible directions, depending upon the value of a certain User Defined Attribute. All of the outgoing arrows of this node represent various directions in which the process can proceed.

A process can have any number of Conditional Nodes. A Conditional Node can have one or more incoming arrows and one or more outgoing arrows.

Supported: You can use Prologue Actions as required. In addition, a Java Action Set can be defined that becomes active when an Administrator aborts, suspends or resumes a process instance to which this node belongs. The semantics of this node type do not support user assignment, forms, timers or triggers.

Behavior:

- The decision of which way to proceed depends on the value of the specified User Defined Attribute. The Conditional Node compares this value with the constant value that is associated with conditions specified by each outgoing arrow.
- When the Conditional Node receives an event from an incoming arrow, it first evaluates the Prologue Action and then the conditions on the outgoing arrows in a specified order.
- The process instance proceeds along the first single outgoing arrow whose condition is satisfied by the value of the specified User Defined Attribute. If the value of the User Defined Attribute satisfies none of the conditions the process instance proceeds along the outgoing arrow specified as the default.

Subprocess Node

A Subprocess Node represents a step in a process where a task is accomplished passing data to an external process, invoking a process there, and ultimately receiving the results back.

Subprocess Nodes facilitate the reuse of existing process definitions. A single Subprocess Node can represent an entire process definition that is already designed, accessing all of the nodes, arrows, and other characteristics of the process definition.

With Subprocess Nodes, a user can organize a complex process into a simpler hierarchical network of subprocesses that hide the details from the top-level view. The processes and subprocesses are independent and can be designed simultaneously.

Subprocess Nodes can also be used to delegate details to other people.

A process can have any number of Subprocess Nodes. A Subprocess Node can have one or more incoming arrows and one or more outgoing arrows.

Supported:

- Prologue and Epilogue Actions can be used as required. In addition, a Java Action Set can be defined that becomes active when an Administrator aborts, suspends or resumes a process instance to which this node belongs. The semantics of this node type do not support role assignment, forms, timers or triggers.
- Attributes specific to this node type include the ID of the process definition to be used to create a subprocess instance and data-mapping references that define what data is moved between the parent process instance and subprocess instance.
- Input data mapping initializes the subprocess's User Defined Attributes with the values from the parent process's User Defined Attributes before the execution of the subprocess.
- Output data-mapping copies values from the subprocess's User Defined Attributes to the parent process's User Defined Attributes after the subprocess is completed.
- Data mapping need not be specified if no data will be copied into or out of the subprocess.

Behavior:

- On receiving an event from an incoming arrow, a Subprocess Node evaluates the Prologue Action and then creates a process instance using the process definition who's ID has been specified. Input data mapping, if specified, initializes the User Defined Attributes of the subprocess.
- The subprocess then starts, and the Subprocess Node waits for it to complete. While waiting, the node ignores any events received from incoming arrows.
- Once the subprocess completes, output data mapping, if specified, copies data from the subprocess to the parent process, evaluates the Epilogue Action, and sends events to each of its outgoing arrows. Each outgoing arrow is executed.

Remote Subprocess Node

A Remote Subprocess Node represents a step in a process where a task is accomplished passing the execution to a process running on a remote workflow server. A remote workflow server might be, for example, another Interstage BPM Server. Like Subprocess Nodes, Remote Subprocess Nodes can be used to reuse existing process definitions within new ones, simplify complex business processes, and delegate details to other people.

A process can have any number of Remote Subprocess Nodes. A Remote Subprocess Node can have one or more incoming arrows and one or more outgoing arrows. However, the names of the outgoing arrows must match with the names of the Exit Nodes in the Remote Subprocess.

Supported:

- Prologue and Epilogue Actions can be used as required. In addition, a Java Action Set can be defined that becomes active when an Administrator aborts, suspends or resumes a process instance to which this node belongs. The semantics of this node type also support the assignment of an Error Java Action that is activated when the starting of the remote subprocess fails. The semantics of this node type do not support role assignment, forms, timers or triggers.
- Attributes specific to this node type include the URI of the process definition to be used to create a remote subprocess instance and data-mapping references that define what data is moved between the parent process instance and the remote subprocess instance.

- Input data mapping initializes the remote subprocess's User Defined Attributes with the values from the parent process's User Defined Attributes before the enactment of the remote subprocess.
- Output data-mapping copies values from the remote subprocess's User Defined Attributes to the parent process's User Defined Attributes after the remote subprocess is completed.
- Data mapping need not be specified if no data will be copied into or out of the remote subprocess.

Behavior:

- On receiving an event from an incoming arrow, a Remote Subprocess Node evaluates the Prologue Action and then creates a process instance using the process definition whose URI has been specified. Input data mapping, if specified, initializes the User Defined Attributes of the remote subprocess.
- The remote subprocess then starts, and the Remote Subprocess Node waits for it to complete. While waiting, the node ignores any events received from incoming arrows.
- Once the remote subprocess completes, output data mapping, if specified, copies data from the remote subprocess to the parent process, evaluates the Epilogue Action, and sends events to each of its outgoing arrows.

Chained-Process Node

A Chained-Process Node represents a step in a process where a new process is created to accomplish a task independent of the parent process instance. This node enacts this independent process as a part of the execution of the parent process.

A process can have any number of Chained-Process Nodes. A Chained-Process Node can have one or more incoming arrows and one or more outgoing arrows.

Supported:

- Prologue and Epilogue Actions can be used as required. In addition, a Java Action Set can be defined that becomes active when an Administrator aborts, suspends or resumes a process instance to which this node belongs. The semantics of this node type do not support role assignment, forms, timers or triggers.
- Attributes specific to this node include the ID of the process definition that has to be used to create new process instance and data mapping that defines what data is to be copied to the new process instance.
- Input data mapping initializes the new process instance's User Defined Attributes with values from the User Defined Attributes of the parent process instance. A Chained-Process Node uses only input data mapping, since the flow of enactment does not return to the parent process instance. Data mapping need not be specified if no data will be copied to the new process instance.

Behavior: On receiving an event from an incoming arrow, a Chained-Process Node evaluates the Prologue Action and then creates a process instance using the process definition who's ID has been specified. Input data mapping, if specified, initializes the User Defined Attributes of the chained-process. The chained-process is independent of the parent process instance.

Compound Activity Node

A Compound Activity node encompasses a phase of a process and the milestones to be achieved at the end of the phase.

Nodes and transitions which are required to be part of particular phase can be nested inside the Compound Activity. Thus, each Compound Activity will represent a phase.

Supported:

On Compound Activity Node:

- Group/Expand Group
- Prologue, Epilogue, onAbort, onSuspend, onResume Actions
- Timers
- Due Date
- Forms
- Priority

On workitems created from Compound Activity Nodes:

- Make Choice
- Reassign
- Priority

Behavior:

When the control reaches the Compound Activity node in a process, the child Start node is activated and the compound activity node state changes to `WaitingOnSubProcess` . Once the child nodes are complete and the child Exit node is reached, the Compound Activity node is closed and the outgoing arrow from the Compound Activity node that has the same name as the child Exit node, is activated.

Milestone date can be defined on the Compound Activity by adding DueDate Timer. There might be a requirement that Phase needs to be forecfully completed. To achieve this, assignee can make choice on Workitems of Compound Node and further Process execution can continue. This way any active child nodes will be forcefully aborted .

Refer to *Using Compound Activity Nodes* on page 166 for more information on how to create and use Compound Activity Nodes.

3.2.11 Modifying Process Definitions

Process definition operations pertain to how process definitions can be modified. To modify any process definition, the following prerequisites must be fulfilled:

- The process definition must be in Draft state and in edit-mode.
- There must not be an instance of the process definition.
- You must be process definition owner or an administrator.
- You must acquire a lock on the process definition from the Interstage BPM Server.

You can only commit the changes to the process definitions atomically; i.e. either all or none can be made. If you cancel or roll back the edit, no changes are made.

3.3 Process Instances

A process definition is a general model of a business process, using workflow elements to describe it. A process instance describes and organizes one specific execution of this process definition. A running process instance mimics the flow of events in the business process from its inception to its completion.

Once a process instance is created, the Interstage BPM Server (also referred to as the workflow engine) executes the process instance according to the design of the process definition. This execution of the process instance by the Interstage BPM Server is referred to as enactment.

Process instances are enacted in the Interstage BPM Server using the event-model mechanism. Events are like messages that are sent to the workflow elements in a process instance as the

Interstage BPM Server enacts it. These events trigger the workflow elements into action. Nodes and arrows react to these events and perform tasks according to their type and definition.

Note: In the current model, arrows are used only as facilitators of the workflow. When an arrow receives an event from its source node, it merely sends an event to its target node. For simplicity, the following discussion omits references to arrow events.

Process instances typically have one Start Node and one or more Exit Nodes. Enactment of a process instance begins when the Start Node receives an event. Nodes react differently to the events, depending on their type and definition. Subsequent events are propagated from node to node in the process instance.

A process instance can have one or more branches of control flow, depending upon the number of paths that it has from its Start Node to its Exit Nodes. However, the process instance completes when one of the Exit Nodes receives an event. When this happens, all active threads are canceled.

Process Instance Attributes

The structure of a process instance is exactly the same as the structure of the process definition on which it is based. Every object and every attribute within the process definition is mirrored by a corresponding process instance or attribute. In addition, process instance have attributes referred to as "process-relevant data" which do not map directly to the process definition in any way.

A process instance can have the same attributes as the originating process definition with certain exceptions. For instance, there is no relationship between the ID of a process definition and the ID of the process instance that is created from it. In addition, a process instance can have the same name, title, and description as the process definition on which it is based, or it can have its own identifiers.

Finally, process instances have a number of additional attributes that are not shared with (or mirrored from) process definitions in any way. These include attachments, User Defined Attributes, and attributes that are meaningless prior to run-time (e.g. process priority).

A process instance attribute could be a persistent data element or a link to an external information source such as a document. Process instance attributes include (but are not necessarily limited to) the following:

- Process instance state
- Process instance identifiers
  - Process ID
  - Process instance name
  - Process instance description
  - Process instance title
- Flow control
  - Node instances
  - Arrow instances
  - Java Action Set
- Process instance ownership
  - Role (process owner)
  - Process initiator
- Variable data
  - Data item definitions

    - Creation time
    - Priority
    - Set of attachment references
- Timers including a Java Action Set

3.3.1 Process Instance States

A process instance can be in one of the following states that are stored on the Interstage BPM Server: initial, running, suspended or completed. When a process instance is created, it is in the initial state until it is started. When the process instance reaches the Exit Node, it enters the completed state.

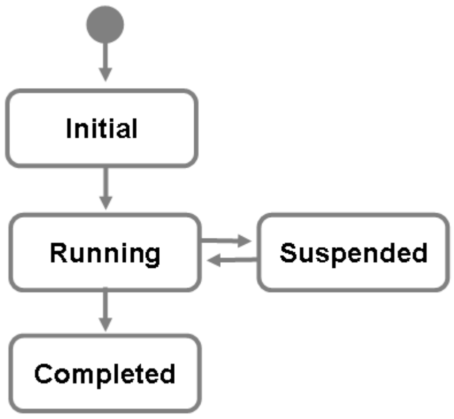


Figure 6: Process Instance States

In practice, most process instances may be created and started at the same time, though it is possible to create a process instance without starting it. However, creating and starting process instances are two separate actions.

Suspending a process instance temporarily removes it from the running state. This temporarily inactive state is called the suspended state. If this process instance has any work items or running subprocesses, they are also put in the suspended state. Also, a suspended process instance cannot be modified.

A suspended process instance is, however, not in a completed state. You cannot modify the process definition from which a suspended process instance was created, you cannot reassign a suspended process instance, and resuming a suspended process instance activates it (puts it back in running state) along with any work items or subprocesses it has.

You can suspend or resume a process instance using the Model API. You must be an administrator to use the suspend/resume methods in the API. Refer to section *Suspending Process Instances* on page 198 for using the `suspend()` and `resume()` methods.

Persistence and State

The chief criteria for whether a state is persistent (i.e. stored on the Interstage BPM Server) is whether making this state persistent will drive the process instance towards completion.

The Interstage BPM Server does not guarantee persistence and might detract from the forward movement of the process instance (such as sessions and session data). For instance, a process instance can be in a locked or unlocked state. This state is volatile and is lost if the Interstage BPM

Server goes down while the process instance is running, because in a situation where the Interstage BPM Server machine fails, a persistently locked state could interfere with recovery.

Interstage BPM supports ACID properties (atomic, consistent, isolated, and durable) for workflow process transactions, which do not include direct coordination across application participants. For mission critical applications, it is recommended that workflow processes be kept separate from application processes, and synchronized by the application based on the workflow steps. If the Interstage BPM Server is unable to complete a workflow process update and has to back it out, the application also can back out its processing to the same point. This coordination can be achieved by using the Model API. In this context, changes are visible to other transactions.

3.3.2 Node Instances and Arrow Instances

Node instances and arrow instances have a one-to-one correspondence with process definition nodes and arrows. Node instances are basically state holders for the specific nodes that they represent. Node instances and arrow instances can be in the following states: initial, running, and completed.

Refer to section *Node Types* on page 33 for information about the input behavior for the different node types.

3.3.3 Process Instance Ownership

A process instance is owned by the person who begins the process instance (Process Initiator). It is not necessary for the process owner to be involved in creating a process instance; however, it can be useful for process owners to know who created each specific process instance that belongs to them.

3.3.4 Attachments

Attachments are references to documents that are created and accessed through other software. They could be images, spreadsheets, word processor files, or any other file that might need to be accessed or modified in accomplishing a business process.

Attachments consist of only references to the document's paths and file names.

Attachments are added, removed, and accessed through a work item. However, attachments belong to the entire process instance, including Activity Nodes on other branches of the workflow.

By default, the work item allows the user to specify which programs to use to open the different file types.

3.4 Work Items

Once an Activity Node has finished a evaluating task assignment, it generates separate work items for each assigned user or user group. Work items are the users' window on the process instance; they are the mechanism by which the user can access all of the attributes of the process instance.

Work Item Attributes

- Identifiers
  - Process instance ID
  - Process instance name
  - Process instance description
  - Process instance title

  - Work item name
  - Work item title
  - Work item description
  - Set of choices
- Assignee
- Variable Data
  - Set of attachment references
  - Set of forms
  - Data items

3.4.1 Work Item Modes

When an Activity Node becomes active, work items are generated according to the associated assignee expression and possibly Role Actions.

The number of generated work items depends on the whether the activity node is set for **individual work items** or **group work items**.

Group Work Item Mode

In this mode, one work item is generated for the entire user group assigned to an Activity Node. Group work items are useful in case of frequent changes to the members of a group and you want all current members of the group to always have access to workitems even when the work item has been created some time before the last group change.

Note: This mode is not applicable for work items associated with Voting Activity Nodes, and when a Role Action is used. If you try setting this for voting activity nodes or when a Role Action is used, a group work item is not generated; instead, work items for each user in the group are generated.

Individual Work Item Mode

In this mode, one work item is generated for every user in the User Group assigned to an Activity Node. Every user works on his/her own work item. Individual work items are required if you want to keep track of users who decline the task, and for activities that can be forwarded to other individual.

Changing the Work Item Mode

The type of workitems generated for an Activity Node is controlled by the `Activity_node.markForExpandGroups` parameter. The value `true` indicates that individual workitems will be created, and `false` indicates that group level workitems will be created. By default, individual work items are generated.

Additionally, there is a `SupportGroupWorkItem` parameter of the Interstage BPM Server which effects process definitions which have no setting onthe activities. This can only happen if the process definitions was created by a very old version of the product. When a process definition that does not specify markForExpandGroups in the Activity Nodes, then that server will set the flag for each activity to the converse of this global parameter at the time that the process definition is installed into the server. Specifically, when a very old process definition is imported into the system, if SupportGroupWorkItem=true, then markForExpandGroups will be set to false, and vice versa. Refer to section *Designing a Process Definition with Start Node, Activity Node and Exit Node* on page 66 for a sample.

3.4.2 Work Item States

When an Activity Node becomes active, Interstage BPM creates one or several work items and assigns it/them to a user group with the defined Role or to one or more people who are members of a Role. By default, individual work items are generated and assigned to every member of a user group; you can change this behavior and have Interstage BPM generate one group work item for the entire group assigned to the respective Activity Node. In the first case, two or more people can avoid duplicating each other's work by accepting the task and reserving the task.

In general, work items can be in the following states: active, read, accepted, declined, inactive (individual work items only), suspended, and completed.

- **Active**: Whenever an Activity Node becomes active, Interstage BPM creates one active work item for the user group assigned to the Activity Node, or, active work items for every member of the Role, unless the selection is modified by a Role Java Action. Active work items are available for any assignee or member of the assigned Role to view, accept, suspend, reassign, or decline.
- **Read**: When a work item has been viewed by an assignee or member of the assigned Role, his or her work item is placed into a "Read" state. All assignees or group members can read one and the same work item. A work item in this state can later be accepted, suspended, reassigned, or declined, just like a work item in the "Active" state.
- **Accepted**: When a work item has been accepted by an assignee or a member of the assigned user group, his or her work item is placed into an “Accepted” state. If there are individual work items associated with that activity, they are placed into the “Inactive” state. This state prevents any replication of efforts.

  The user who has accepted a work item may later choose to decline it. When this happens, the work item becomes active again. In addition, in Regular Reassignment Mode, a person who has accepted a work item may assign it to someone else.

  Refer to section *Reassignment Modes* on page 50 for information about the possible modes.
- **Declined**: A work item in the "Declined" state is one that a user has refused. A declined work item no longer appears on a user’s list of active work items, though it still appears on the list of inactive work items. It is still available for the user to view or accept.

  If all of the assignees or all members of the Role decline their **individual work items**, an active work item is created for each of the process instance owners. However, every owner cannot decline the work item; if all process instance owners decline their work items, the work items become active for all of them again.

  If everyone declines a **group work item**, no work item is created for any process instance owner. The existing group members can work on the group work item after accepting it; any new member added to the group can work on it directly.

  A user can accept a previously declined work item, so long as someone else has not already accepted it.
- **Inactive**: A work item becomes inactive for a user when someone else has accepted it.

  An inactive work item cannot be declined or accepted, though it can become active at a later time, when it can again be accepted or declined
- **Completed**: When the user makes an appropriate choice on the work item, it enters the Completed state, and all other work items that correspond to the same activity are removed from the system. This means that they disappear from the list of work items.

3.4.3 Choices

When a user has completed the tasks specified by the work item, he or she may choose one or more commit options to complete the activity and activate the next node in the process instance.

3.4.4 Future Work Items

Future Work Items are the work items that the users may be assigned in future. As a user, Future Work Items help you to be aware of work items that may be assigned to you later so you can plan your tasks in advance.

Future Work Items generated for a user may not be accurate since task assignment can vary during the process execution.

Note: Future Work Items are generated just as normal work items, with state as `WorkItem.STATE_FUTURE`. They are displayed as a list for a particular user. User cannot actually accept them or reject them or perform any kind of actions on them since these are displayed as a read-only list.

Following are the properties of Future Work Items:

- Future Work Items are generated only on the activity nodes that have the `Activity_node.markForFutureWorkItems` flag set to `true`. By default, this flag is set to 'false'.

Note: Future Work Items can also be generated for activity nodes in Compound Node.

- Future Work Items are generated only on inactive (initial state) activity node instances of a process instance.
- Future Work Items are generated in a process instance only when at least one activity node instance is activated.
- When an activity node instance is activated, it generates work items for current node, deletes Future Work Items for current node and generates Future Work Items for other activity node instances that are in initial state.
- Future Work Items are generated using Assignee and Role Action.

Note: Any process update done in Role Actions while evaluating assignees of Future Work Items is committed. Future Work Items are generated for an activiy node whenever any other activity node instance is activated in the process instance. Hence, these Role Actions will be executed multiple times. Also, they will be executed again while generating normal work items. You need to make sure that execution of these actions multiple times, does not result in undesired behavior.

- When Role Action or Role Script is set in an Activity Node, Future Work Items are assigned to the users using the intersection of Role and assignees specified by the Role Action or Role Script. If this intersection does not contain any value, Future Work Items are not generated for process owners (normal Work Items are generated for the process owners in such a case).
- Future Work Items are generated or deleted during the process enactment. This means if an activity node instance A is in active state and waiting for the user to make choice and you set the Future Work Item flag of any other activity node instance in the initial state (say, B), to 'true' then B will not generate Future Work Items until you make choice on A.
- Future Work Items cannot be generated for the activity nodes that have Agents defined on them.
- Future Work Items cannot be generated for the activity nodes configured as Iterator Nodes.

- Future Work Items cannot be generated for Voting Activity Nodes.
- Future Work Items cannot be generated for the activity nodes having group-level work items.
- Future Work Items generated for an activity node in 'initial' state are deleted once the state of the activity node changes to 'active'.
- Any failure while generating Future Work Items is handled similar to the normal failure, that is, transaction is rolled back and exception is thrown to user or process instance is put into error state depending on inline/background enactment. Also, compensate and error actions are executed in normal fashion.

3.5 Filters

Filters in Interstage BPM operate similarly to a SQL SELECT statement: they allow you to access columns in a table in a database according to specific criteria. In other words, you can create a list from the table of process definitions, process instance, or work items; you can filter them by assignee, initiator or owner, by priority, by state, or by identifier.

Note: Advanced filtering and sorting capabilities are available through the Model API. Refer to section *Advanced Filtering & Sorting API* on page 110 for instructions in using these advanced capabilities.

3.6 The Purpose of Structural Process Editing

Process editing is the key feature required for several different types of workflow which require fewer constraints on the direction a process instance can take:

- Ad Hoc Workflow
- Process Discovery
- Collaborative Workflow
- Case Handling

Workflow users need structural process editing for the following purposes:

- **Design error**: After a process instance has been running for a period of time, users might discover that it is based on faulty assumptions or logic.
- **Maintenance**: Most long-running process instances will need to be tweaked periodically to keep them up-to-date and running smoothly.
- **Process discovery**: In the real world, many business processes are started without completely solidifying the business plan. Structural process editing allows a project team to fly by the seat of their pants, modifying the process definition from the feedback they receive, and recording their process instance for later replication.
- **Obtaining Experimental Results**: Structural process editing allows you to try out a new procedure along the way and later analyze the results of using two or more methods that are precisely recorded.
- **Adjusting to New Requirements**: Structural process editing enables you to modify your process definition whenever new government regulations or new market conditions occur.
- **Starting Over**: When a process is proceeding according to plan, but no one is happy with the results, structural process editing allows you to start it over with modifications.

3.7 The Purpose of Interstage BPM's Subprocess Capabilities

Subprocesses are an important element because of the following:

- Subprocesses enable users to break a project or process into easy to handle units of collaboration. In other words, a complex process definition might be made up of an extremely large number of nodes, making the process definition unreadable. If you reconstruct this process instance into a hierarchy of simple subprocesses, each level of the process definition can be designed as a separate task that fits easily into the whole process definition.
- Subprocesses also enable different departments with different processes to linking those processes together easily and appropriately. For instance, a purchasing department, an accounting department, and a new product development department will need completely separate process definitions to serve their individual internal needs. However, Interstage BPM's subprocesses enable them to create an interdepartmental process definition to trade information and coordinate collaborative tasks among all three seamlessly.
- Subprocesses enable an individual, a department, or a company to create a library of modular components, process definitions, or tasks that they can later paste together in a wide variety of ways.

Users can create subprocesses in two different ways: formally at design time and on the fly at run time. These two different implementation strategies involve slightly different structures in the APIs.

- **Design-time subprocess implementation**, for instance, requires the use of a Subprocess Node, a node type that is expressly constructed for this purpose. This node provides a uniform interface that users can plug into a process definition at design time, with properties that specify the subprocess definition and the method for mapping data that will be passed back and forth between the parent process instance and child process instance. This Subprocess Node becomes a member of a process definition in the same way that any other node is a member of a specific process definition.

  When the flow of an active process instance reaches a Subprocess Node, the node immediately instantiates the subprocess and then starts it running. While waiting for the subprocess to complete, the Subprocess Node enters a suspended state.

- **Run-time subprocess implementation** uses methods that are part of the Activity Node class. When a task is assigned to an individual who must delegate the task (or parts of it) to others, the person can invoke a run-time subprocess from the Activity Node. The delegating person can create a new process definition or use an existing one and can map data from the original process instance to the subprocess - just as if it had been done at design time. The original assigned work item enters a suspended state as soon as a subprocess is invoked. The node changes to a completed state, when the subprocess is complete and passes the workflow back to the work item.

The difference between run-time and design-time subprocess implementation is that run-time subprocess implementation is ephemeral and leaves the original process definition in its original state. The next time that the process definition is used to create a process, the original Activity Node is still only an Activity Node.

However, when subprocesses are used with process editing, it enables a powerful feature: **enterprise-wide process discovery workflow**. This enables an executive, for instance, to create a general statement of the work of the company, and use run-time subprocesses to get input from those below him in the hierarchy. As they develop and distribute their own subprocesses, a procedural description of the work of the entire business can evolve. Thus, enterprise-wide process discovery workflow provides a gateway into reengineering the entire business operation through the use of workflow.

3.8 Security and Reassignment Modes

The Security and Reassignment Modes address slightly different issues than the work item states. However, since all of these modes and states work together to control work item access, they must be addressed together. By default, the system is installed in Open Security mode and Regular Reassignment mode.

3.8.1 Security Modes

The Security mode can be modified by setting the `SecuritySwitch` parameter of the Interstage BPM Server. Refer to the *Interstage Business Process Manager Server Administration Guide* for more information on how to change the configuration of the Interstage BPM Server.

The major difference in work item access between Open Security Mode and Secure Mode is that in Open Security Mode, any Interstage BPM user can view a work item (read-only). In Secure Mode, only the work item assignees and the process owners can view the work item and modify process details or make a choice on the work item.

3.8.2 Reassignment Modes

The Reassignment mode can be modified by setting the `ServerReassignMode` parameter of the Interstage BPM Server. Refer to the *Interstage Business Process Manager Server Administration Guide* for more information on how to change the configuration of the Interstage BPM Server.

Reassignment allows activities to be assigned to a different set of people than those originally specified through the User Group. The access control created for reassignment operates differently than access control that is implemented by work item state. Reassignment acts on the activity, not on the work item.

Reassigning a work item causes to delete all active and inactive work items associated with the activity. Then the server creates new, active work items for the people specified in the reassignment.

Activities must be reassigned to individuals, not to Groups, because assignment is mainly appropriate for process definitions that will be reused for a period of time. Because of this, it does not verify that the people specified in the reassignment are valid users.

Interstage BPM can operate in three different reassignment modes: Regular Mode, Process-Owner-Only Mode or No-Reassignment Mode. These are configured by setting the `ServerReassignMode` parameter to regular, owner or none.

- **Regular Mode**: In this mode anyone who is a current assignee for the activity or a process owner can reassign the activity to a new set of users.
- **Process-Owner-Only Mode**: In this mode only a process owner can reassign an activity to a different set of users. The process owner doesn't have to be an assignee of a work item in order reassign the work item to someone else.
- **No-Reassignment Mode**: In this mode, reassignment is completely disabled, and no one can reassign an activity.

4 Using the Model API

This chapter describes the system environment for application development using the Model API and introduces the basic Model API architecture.

4.1 System Environment

You can develop and deploy applications using the Model API on all of the operating system platforms supported by Interstage BPM. Refer to the *Release Notes* for a list of supported operating systems.

On the system where you want to **develop** your applications using the Model API, the following is required:

- J2SE Development Kit (JDK) 5.0 Update 10 is configured correctly.
- `iFlow_api.jar`. You must add its path to the `CLASSPATH` environment variable. In a typical installation, you can find this file at the following location:

  On Windows:

  `C:\Fujitsu\InterstageBPM\client\lib\iFlow.jar`

  On UNIX or Linux:

  `/opt/FJSVibpm/client/lib/iFlow.jar`

Note: The `iFlow.jar` file contains methods that are used internally by Interstage BPM. These methods are not supported for application development and are not described in the Javadoc. For this reason, an alternative jar file is provided for development purposes: `iFlow_api.jar`, which does not contain any internally used methods. It is recommended that you compile your application with the `iFlow_api.jar` library so that you make sure that your application does not use any internal methods.

All other libraries (except `iFlow_api.jar`) contained in the `C:\Fujitsu\InterstageBPM\client\lib` or `/opt/FJSVibpm/client/lib` directory must also be added to the CLASSPATH, since they are used by the Model API implementation. Currently, there are two libraries in the `client/lib` directory that are needed at runtime:

- `js.jar`
- `log4j-1.2.15.jar`

- Make sure that you add and define the path to the required application-server specific `jar` files to the `CLASSPATH`.

4.1.1 Specifying Configuration Settings for Interstage Application Server (Local)

Prerequisite: Set up your system environment as explained in section *System Environment* on page 51.

If you want to develop applications using the Model API on the Interstage Application Server, the following configuration settings are required:

1. Add the following JAR file to the `CLASSPATH` environment variable:

   On Windows:

   ```
   <Interstage Application Server Installation
   Directory>/J2EE/var/deployment/ijserver/<Work Unit Name of the Interstage BPM
   ```

`Server>/distribute/fujitsu-ibpm-engine.ear/fujitsu-ibpm-engine-ejb.jar/fujitsu-ibpm-engine-ejb_jar_client.jar`

On Solaris/Linux:

`<Interstage Application Server Installation Directory>/FJSVj2ee/var/deployment/ijserver/<Work Unit Name of the Interstage BPM Server>/distribute/fujitsu-ibpm-engine.ear/fujitsu-ibpm-engine-ejb.jar/fujitsu-ibpm-engine-ejb_jar_client.jar`

2. Add the following JAR files to the `CLASSPATH` environment variable:

   On Windows:

   `C:\Interstage\J2EE\lib\isj2ee.jar`

   `C:\Interstage\ODWin\etc\class\ODjava4.jar`

   `C:\Interstage\jms\lib\fjmsprovider.jar`

   `C:\Interstage\EJB\lib\fjcontainer94.jar`

   `C:\Interstage\eswin\lib\esnotifyjava4.jar`

   On Solaris:

   `/opt/FJSVj2ee/lib/isj2ee.jar`

   `/opt/FSUNod/etc/class/ODjava4.jar`

   `/opt/FJSVjms/lib/fjmsprovider.jar`

   `/opt/FJSVes/lib/esnotifyjava4.jar`

   `/opt/FJSVejb/lib/fjcontainer94.jar`

   On Linux:

   `/opt/FJSVj2ee/lib/isj2ee.jar`

   `/opt/FJSVod/etc/class/ODjava4.jar`

   `/opt/FJSVejb/lib/fjcontainer94.jar`

   `/opt/FJSVes/lib/esnotifyjava4.jar`

   `/opt/FJSVjms/lib/fjmsprovider.jar`

3. Add the following directories to the `PATH` environment variable:

   On Windows:

   `C:\Interstage\bin`

   On UNIX or Linux:

   `/opt/FJSVj2ee/bin`

   `/opt/FJSVjms/bin`

4. Add the following directories to the `LD_LIBRARY_PATH` environment variable:

   On Solaris:

   `/opt/FSUNod/lib`

   `/opt/FJSVjms/lib`

   On Linux:

   `/opt/FJSVod/lib`

   `/opt/FJSVjms/lib`

5. Go to the directory `<JDK Installation Directory>/jre/lib`. Open the `orb.properties` file and add the following values:

```
org.omg.CORBA.ORBClass=com.fujitsu.ObjectDirector.CORBA.ORB
org.omg.CORBA.ORBSingletonClass=com.fujitsu.ObjectDirector.CORBA.SingletonORB
javax.rmi.CORBA.StubClass=com.fujitsu.ObjectDirector.rmi.CORBA.StubDelegateImpl
javax.rmi.CORBA.UtilClass=com.fujitsu.ObjectDirector.rmi.CORBA.UtilDelegateImpl
javax.rmi.CORBA.PortableRemoteObjectClass=
com.fujitsu.ObjectDirector.rmi.CORBA.PortableRemoteObjectDelegateImpl
```

4.1.2 Specifying Configuration Settings for Interstage Application Server (Remote)

Prerequisite: Set up your system environment as explained in section *System Environment* on page 51.

You can use the Model API on a remote computer (that is on a computer different from the one hosting the Interstage Application Server).

To use the Model API on a remote computer:

1. On your remote computer, install the Interstage Application Server Client Package.

   Refer to the *Interstage Application Server Installation Guide* for detailed information.

2. Copy the following JAR file to your remote computer and add the JAR file to the `CLASSPATH` environment variable:

   On Windows:

   ```
   <Interstage Application Server Installation
   Directory>/J2EE/var/deployment/ijserver/<Work Unit Name of the Interstage BPM
   Server>/distribute/fujitsu-ibpm-engine.ear/fujitsu-ibpm-engine-ejb.jar/fujitsu-
   ibpm-engine-ejb_jar_client.jar
   ```

   On Solaris/Linux

   ```
   <Interstage Application Server Installation
   Directory>/FJSVj2ee/var/deployment/ijserver/<Work Unit Name of the Interstage
   BPM Server>/distribute/fujitsu-ibpm-engine.ear/fujitsu-ibpm-engine-
   ejb.jar/fujitsu-ibpm-engine-ejb_jar_client.jar
   ```

3. Add the following JAR files to the `CLASSPATH` environment variable:

   On Windows:

   ```
   C:\Interstage\J2EE\lib\isj2ee.jar
   C:\Interstage\ODWIN\etc\class\ODjava4.jar
   C:\Interstage\jms\lib\fjmsprovider.jar
   C:\Interstage\ODWIN\etc\class\esnotifyjava4.jar
   C:\Interstage\EJBCL\lib\fjcontainer94.jar
   ```

   On Solaris:

   ```
   /opt/FJSVj2ee/lib/isj2ee.jar
   /opt/FSUNod/etc/class/ODjava4.jar
   /opt/FJSVjms/lib/fjmsprovider.jar
   /opt/FJSVes/lib/esnotifyjava4.jar
   /opt/FJSVejb/lib/fjcontainer94.jar
   ```

   On Linux:

```
/opt/FJSVj2ee/lib/isj2ee.jar
/opt/FJSVod/etc/class/ODjava4.jar
/opt/FJSVejb/lib/fjcontainer94.jar
/opt/FJSVes/lib/esnotifyjava4.jar
/opt/FJSVjms/lib/fjmsprovider.jar
```

4. Add the following directories to the `PATH` environment variable:

   On Windows:

   ```
   C:\Interstage\bin
   ```

   On UNIX or Linux:

   ```
   /opt/FJSVj2ee/bin
   /opt/FJSVjms/bin
   ```

5. Add the following directories to the `LD_LIBRARY_PATH` environment variable:

   On Solaris:

   ```
   /opt/FSUNod/lib
   /opt/FJSVjms/lib
   ```

   On Linux:

   ```
   /opt/FJSVod/lib
   /opt/FJSVjms/lib
   ```

6. Go to the directory `<JDK Installation Directory>/jre/lib`. Open the `orb.properties` file and add the following values:

   ```
   org.omg.CORBA.ORBClass=com.fujitsu.ObjectDirector.CORBA.ORB
   org.omg.CORBA.ORBSingletonClass=com.fujitsu.ObjectDirector.CORBA.SingletonORB
   javax.rmi.CORBA.StubClass=com.fujitsu.ObjectDirector.rmi.CORBA.StubDelegateImpl
   javax.rmi.CORBA.UtilClass=com.fujitsu.ObjectDirector.rmi.CORBA.UtilDelegateImpl
   javax.rmi.CORBA.PortableRemoteObjectClass=
   com.fujitsu.ObjectDirector.rmi.CORBA.PortableRemoteObjectDelegateImpl
   ```

7. Execute the following command to configure the server host for the CORBA services that work on the Interstage Application Server:

   ```
   odsethost -a -h MyServer -p 8002
   ```

Note: When several server hosts are registered with this command, Model API tries to connect to the server that was registered first. If you want to connect to another server, execute the `odsethost` command with the `-a` or `-d` option to change the order of the server hosts.

8. Execute the following command to register the definition of the JMS connection factory:

   ```
   jmsmkfact -t -i IflowClient TopicConnectionFactory
   ```

Note: You can display the list of connection factory definitions by executing the `jmsinfofact` command.

9. Execute the following commands to register the definitions of the JMS destinations:

   ```
   jmsmkdst -t -g IflowECG2 -c IflowECNotify NotificationTopic
   jmsmkdst -t -g IflowECG2 -c IflowSQNotify SQNotificationTopic
   ```

Note: You can display the list of JMS destination definitions by executing the `jmsinfodst` command.

10. If you wish to use the JMS interface of Interstage BPM, execute the following commands to register the definitions of the JMS destinations:

    ```
    jmsmkdst -t -g IflowECG2 -c IflowECCommand CommandTopic
    ```

    ```
    jmsmkdst -t -g IflowECG2 -c IflowECResponse ResponseTopic
    ```

    For more information about the JMS interface, refer to section *Samples Related to the JMS Interface* on page 215.

4.1.3 Specifying Configuration Settings for WebSphere Application Server for developing Standalone Applications (Local)

Prerequisite: Set up your system environment as explained in section *System Environment* on page 51 of the *Interstage BPM Developer's Guide*.

To develop standalone applications using the Model API on WebSphere Application Server, add the following JAR files to the CLASSPATH environment variable:

Note: `<WAS AppServer home>` in the JAR files below, stands for `<WAS installation dir>\IBM\WebSphere\AppServer`

- `<WAS AppServer home>\plugins\com.ibm.ws.runtime_6.1.0.jar`
- `<WAS AppServer home>\plugins\com.ibm.ws.sib.server_2.0.0.jar`
- `<WAS AppServer home>\plugins\com.ibm.ws.sib.utils_2.0.0.jar`
- `<<WAS AppServer home>\runtimes\com.ibm.ws.webservices.thinclient_6.1.0.jar`
- `<WAS AppServer home>\deploytool\itp\plugins\com.ibm.websphere.v61_6.1.200\ws_runtime.jar`
- Depending on your operating system:
  - On Windows:
    `C:\Fujitsu\InterstageBPM\server\deployment\InterstageBPMServer.ear\fujitsu-ibpm-engine-ejb.jar`
  - On Solaris or Linux:
    `/opt/FJSVibpm/server/deployment/InterstageBPMServer.ear/fujitsu-ibpm-engine-ejb.jar`

4.1.4 Specifying Configuration Settings for WebSphere Application Server for Deploying Client J2EE Application (Local)

Prerequisite: Make sure that the WebSphere Application Server of the version stated in the System Requirements is properly installed and running on your machine.

You can use Model API in your J2EE application on the same machine where the Interstage BPM Server is installed.

To deploy Client J2EE Application for WAS:

1. Create a `J2EE application (EAR)` including your web application and `fujitsu-ibpm-engine-model-ejb.jar` as follows:
   a) Create a new folder, for example, **My Application** on your computer, and copy the following files/folders to it:
      - your application WAR file (for example `myapp.war`)
      - `fujitsu-ibpm-engine-model-ejb.jar` from IBPM Server installation directory, which is `<IBPMServer_Installation_Directory>\client\lib\`
      - the META-INF folder which contains the deployment descriptor for your J2EE application

   b) Specify `fujitsu-ibpm-engine-model-ejb.jar` as a module in your application deployment descriptor (`application.xml`). For example,

```
<module id="EjbModule_1187608318108">
   <ejb>fujitsu-ibpm-engine-model-ejb.jar</ejb>
</module>
```

Note: `"EjbModule_1187608318108"` can be customized.

   Refer to sample deployment descriptor from Interstage BPM CD-Image under `console\ibpmconsole-ear\`

   c) From **My Application**, run the following command: `<java_home>/jar.exe cvfm myapp.ear META-INF\MANIFEST.MF .`

2. Deploy the `J2EE Application(EAR)`, you created in the previous step, on WebSphere Application Server with single class loader option.
   a) In WAS Console, select **Applications**>**Enterprise Applications**.
      A list of Enterprise Applications appears.
   b) Click the link for your application.
      The configuration page for the application appears.
   c) Select the **Class loading and update detection** link.
      The class loader details page appears.
   d) In the **WAR class loader policy** section, select the **Single class loader for application** option, and click **Apply**. Save your changes.
3. Copy `fujitsu-ibpm-engine-ejb.jar` from `<IBPM_installation_directory>\server\deployment\InterstageBPMServer.ear\` to `<J2EE_APP_HOME> \WEB-INF\lib` location.
4. Copy `iFlow.jar` and `js.jar` from `<IBPM_installation_directory>\client\lib\` to `<J2EE_APP_HOME>\WEB-INF\lib`.
5. Start J2EE application.

Note: `<J2EE_APP_HOME>` stands for `<WAS_APPSERVER_HOME>\profiles\AppSrv01\installedApps\<cell Name>\<application_ name>.ear\<web_application_name>.war\`.

4.1.5 Specifying Configuration Settings for WebSphere Application Server for developing Standalone Applications (Remote)

Prerequisite: Set up your system environment as explained in section *System Environment* on page 51 of the *Interstage BPM Developer's Guide*.

To use the Model API for developing standalone applications on a remote computer (that is on a computer different from the one hosting the WebSphere Application Server):

1. Install WebSphere Application Client on the remote machine. During installation, ensure you:
   a) Select **Custom J2EE and thin client** as the setup type.
   b) Select the **Web Services Thin Client** feature.
2. Copy `iFlow.jar` and `fujitsu-ibpm-engine-ejb.jar` to the remote machine and add these files to the `CLASSPATH` environment variable.
3. Add the following WAS Client JAR files to the `CLASSPATH` environment variable:

Note: `<WAS AppClient home>` in the JAR files below, stands for `<WAS Client installation dir>\IBM\WebSphere\AppClient`.

- `<WAS AppClient home>\plugins\com.ibm.ws.runtime_6.1.0.jar`
- `<WAS AppClient home>\plugins\com.ibm.ws.emf_2.1.0.jar`
- `<WAS AppClient home>\plugins\com.ibm.ws.wccm_6.1.0.jar`
- `<WAS AppClient home>\plugins\com.ibm.ws.sib.client_2.0.0.jar`
- `<WAS AppClient home>\plugins\com.ibm.ws.sib.utils_2.0.0.jar`
- `<WAS AppClient home>\runtimes\com.ibm.ws.webservices.thinclient_6.1.0.jar`

4. Execute the client program using the JDK of WebSphere Application Client.

4.1.6 Specifying Configuration Settings for WebSphere Application Server for Deploying Client J2EE Application (Remote)

Prerequisite: Ensure that there are two machines, Machine A and Machine B, for instance, and WebSphere Deployment manager as well as IBPM Server is installed in Machine A and WebSphere Application server is installed in Machine B. Make sure that the WebSphere Application Server of both machines is of the version stated in the System Requirements .

To deploy Client J2EE Application for WAS:

1. On Machine B, create a `J2EE application(EAR)` including your web application and `fujitsu-ibpm-engine-model-ejb.jar` as follows:
   a) Create a new folder, for example, **My Application** on your computer, and copy the following files/folders to it:
      - your application WAR file (for example `myapp.war`)
      - `fujitsu-ibpm-engine-model-ejb.jar` from IBPM Server installation directory, which is `<IBPMServer_Installation_Directory>\client\lib\`
      - the META-INF folder which contains the deployment descriptor for your J2EE application

b) Specify `fujitsu-ibpm-engine-model-ejb.jar` as a module in your application deployment descriptor (`application.xml`). For example,

```
<module id="EjbModule_1187608318108">
   <ejb>fujitsu-ibpm-engine-model-ejb.jar</ejb>
</module>
```

Note: `"EjbModule_1187608318108"` can be customized.

Refer to sample deployment descriptor from Interstage BPM CD-Image under `console\ibpmconsole-ear\`

c) From **My Application**, run the following command: `<java_home>/jar.exe cvfm myapp.ear META-INF\MANIFEST.MF .`

2. Deploy the `J2EE Application(EAR)`, you created in the previous step, on Machine B's WebSphere Application Server with single class loader option.
   a) In WAS Console, select **Applications**>**Enterprise Applications**.
      A list of Enterprise Applications appears.
   b) Click the link for your application.
      The configuration page for the application appears.
   c) Select the **Class loading and update detection** link.
      The class loader details page appears.
   d) In the **WAR class loader policy** section, select the **Single class loader for application** option, and click **Apply**. Save your changes.
3. Configure the JMS related settings on Deployment Manager of Machine A as follows:
   a) Add managed nodes of Machine A and Machine B to the Deployment Manager. For details refer the *Adding Managed Nodes* section of the *Interstage BPM Server and Console Installation Guide (WebSphere Application Server)*.
   b) A SI Bus `machineANode01IBPMBuss` will be created during IBPM server installation on Machine A. Add Server B as Member of this SI Bus, as follows:

Note: Server A already exists on the SI Bus on Machine A.

1. On the **Buses** page, click the name of the `machineANode01IBPMBuss` bus.
2. On the **Configuration** tab, click **Bus members**.
3. Click **Add**.
4. Select `machineANode01:server1` from the **Server** drop-down list and click **Next**.
5. Make sure that **File Store** is selected as the message store and click **Next**.
6. Use the default message store properties and click **Next**.
7. Click **Finish**.
8. In the message displayed at the top of the page, click **Review** . Select **Synchronize changes with Nodes** and click **Save** to save the configuration.

c) Add `SQNotificationTopic` and `NotificationTopic` to your `J2EE Application(EAR)` as follows:

1. Go to **Resources > JMS** tab and select **Topics**.
2. Select scope as **Node** = `machineBNode01`, **Server** = `server1`.

3. Click on **New**, leave the default setting as it is and then select **ok**.
4. Specify the field details for `SQNotificationTopic` as follows:
   - **Name** as `iFLowSQNotificationTopic`
   - **JNDI name** as `iFlow/jms/sq/SQNotificationTopic`
   - **Topic name** as `SQNotification_MDB_TOPIC`
   - Select `machineANode01IBPMBuss` as **Bus name**
   - Select `SQNotification.Topic.Space` as **Topic space** from the list
   - Select `Nonpersistent` for **JMS delivery mode**

   leave all other values as it is.
5. Select **Apply**, then review and save the configuration.
6. Repeat steps 1 to 5 for `NotificationTopic`. Specify the field details as follows:
   - **Name** as `iFLowNotificationTopic`
   - **JNDI name** as `iFlow/jms/NotificationTopic`
   - **Topic name** as `JmsNotification_MDB_TOPIC`
   - Select `machineANode01IBPMBuss` as **Bus name**
   - Select `JmsNotification.Topic.Space` as **Topic space** from the list
   - Select `Nonpersistent` for **JMS delivery mode**

   leave all other values as it is.

d) Add **Activation Specification** for `SQNotificationTopic` and `NotificationTopic` as follows:

1. Go to **Resources > JMS** tab and select **Activation Specification**.
2. Select scope as **Node** = `machineBNode01`, **Server** = `server1`.
3. Click on **New**, leave the default setting as it is and then select **ok**.
4. Specify the field details for `SQNotificationTopic` as follows:
   - **Name** as `SQNotificationTopic`
   - **JNDI name** as `eis/SQNotificationTopic`
   - **Destination type** as `Topic`
   - **Destination JNDI name** as `iFlow/jms/sq/SQNotificationTopic`
   - Select `machineANode01IBPMBuss` as **Bus name**

   leave all other values as it is.
5. Select **Apply**, then review and save the configuration.
6. Repeat steps 1 to 5 for `NotificationTopic`. Specify the field details as follows:
   - **Name** as `NotificationTopic`
   - **JNDI name** as `eis/NotificationTopic`
   - **Destination type** as `Topic`
   - **Destination JNDI name** as `iFlow/jms/NotificationTopic`
   - Select `machineANode01IBPMBuss` as **Bus name**

   leave all other values as it is.

4. Copy `fujitsu-ibpm-engine-ejb.jar` from `<IBPM_installation_directory>\server\deployment\InterstageBPMServer.ear\` to `<J2EE_APP_HOME> \WEB-INF\lib` location.

5. Copy `iFlow.jar` and `js.jar` from `<IBPM_installation_directory>\client\lib\` to `<J2EE_APP_HOME>\WEB-INF\lib`.
6. Restart both Server A and Server B respectively.

Note: `<J2EE_APP_HOME>` stands for `<WAS_APPSERVER_HOME>\profiles\AppSrv01\installedApps\<cell Name>\<application_ name>.ear\<web_application_name>.war\`.

4.1.7 Specifying Configuration Settings for WebLogic

Prerequisite: Set up your system environment as explained in section *System Environment* on page 51.

- If you want to develop applications using the Model API on the WebLogic Application Server, add the following JAR files to the `CLASSPATH` environment variable:

  `<BEA WebLogic Installation Directory>/weblogic92/server/lib/wlclient.jar`

  `<BEA WebLogic Installation Directory>/weblogic92/server/lib/wljmsclient.jar`

4.1.8 Specifying Configuration Settings for JBoss

Prerequisite: Set up your system environment as explained in section *System Environment* on page 51.

- If you want to develop applications using the Model API on the JBoss Application Server, add the following directory containing the required JAR files to the `CLASSPATH` environment variable:

  `<JBoss Installation Directory>/client`, for example, `/opt/jboss-4.0.5.GA/client/jbossall-client.jar`.

4.2 Executing a Model API Application

Prerequisites:

- RMI access is possible to the computer where Interstage BPM is installed.
- The Interstage BPM Server has been started.

To execute your Model API application:

- In your Interstage BPM Server installation directory, you find a script for starting the programming samples provided with Interstage BPM:

  On Windows `C:\Fujitsu\InterstageBPM\client\samples\examples\bin\StartSamples.bat`

  On UNIX or Linux `/opt/FJSVibpm/client/samples/examples/bin/StartSamples.sh`

- You need to ALWAYS execute the command specified in the "Start the Sample" section of this script BEFORE you can execute a client application. This script requires you to enter the Java class name of your application.

  For details on how to use the samples, refer to Appendix *Using the Interstage Business Process Manager Samples* on page 213.

4.3 Location of Properties Files

You can access properties files from within your Model API application. The following code fragment shows an example of how to load the `iFlowClient.properties` file:

```
Properties iflowProps = new Properties();
try {
   FileInputStream fin = new FileInputStream("iFlowClient.properties");
   iflowProps.load(fin);
}
finally {
   fin.close();
}
```

As a default, properties files are loaded from the current runtime directory. You need to store your properties files there. The sample `iFlowClient.properties` file is located in the following directory:

`<Interstage BPM Server Installation Directory>/client`

If you want to store your properties file in a different directory, you need to specify the file's absolute path or the path relative to the current runtime directory. The following is an example:

```
Properties iflowProps = new Properties();
try {
   FileInputStream fin = new
FileInputStream("C:\\Fujitsu\\InterstageBPM\\iFlowClient.properties");
   iflowProps.load(fin);
}
finally {
   fin.close();
}
```

Note: For Interstage Application Server, the following properties available in the iflow.properties file are **NOT** used:

`NamingProviderURL` and `JMSNamingProviderURL`.

4.4 Model API Architecture

The Model API allows you to access and manipulate Interstage BPM, including its administration function. With the Model API you can build your own applications which you can integrate in your own graphical user interface.

The Model API contains, among others, the following packages. For detailed information, refer to the API Javadoc.

- **`com.fujitsu.iflow.model.event`**: Contains interfaces and classes that listen for changes in the Interstage BPM objects to provide proactive notification to the user.
- **`com.fujitsu.iflow.model.util`**: Contains low level utility classes that are commonly used by other classes and interfaces. It also contains Exceptions thrown by the Model classes themselves.
- **`com.fujitsu.iflow.model.wfadapter`**: Contains interfaces that manage the Document Management System (DMS). This includes retrieving and updating information about folders and attachments in the DMS, checking objects in or out from the DMS.
- **`com.fujitsu.iflow.model.workflow`**: Contains interfaces that manage information required by the workflow process definitions and process instances. This includes objects that represent

nodes, arrows, forms, attachments, work items and permission levels. The programming examples explained in this manual, above all, use the following interfaces:

- **`Arrow`** interface: Used for creating and manipulating arrows.
- **`ArrowInstance`** interface: Used to access arrow attributes within a process instance.
- **`AttachmentRef`** interface: Used to access attachments to a process instance in the DMS. There is no limit to the number of attachments that can be associated with a process instance, and there are no restrictions on the types of attachments that can be added.
- **`DataItemRef`** interface: Used to hold the name, type, and initial value of User Defined Attributes (UDAs) defined in a process definition.
- **`JavaActionSet`** interface: Provides methods for using Java Actions within process definitions.
- **`Node`** interface: Defines all the functions that can be used to obtain information about any node type within a process definition.
- **`Plan`** interface: Used for creating and manipulating process definitions.
- **`ProcessInstance`** interface: Provides operations for creating and initiating a process instance.
- **`WFAdminSession`** interface: Provides methods used by an administrator. Extends the `WFSession` interface.
- **`WFDetailsList`** interface: Allows for retrieving information about multiple objects at a time.
- **`WFObjectList`** interface: Allows for advanced sorting and filtering of process definitions, process instances or work items.
- **`WFSession`** interface: Provides methods for establishing and maintaining access to the Interstage BPM Server for the entire time a user is logged into it.
- **`WorkItem`** interface: Provides access to all aspects of a work item, that is, an activity that is assigned to a particular user.

- **`com.fujitsu.iflow.server.intf`**: Contains an interface that provides access to workflow data. This interface is called ServerEnactmentContext. This interface also contains classes for implementing action agents.

4.5 Exception Handling

There are two levels of exceptions thrown by the Model API. The `ModelException` class is the super class. Use this class for your exception handling.

The `ModelException` class contains some sub classes. For reasons concerning the future compatibility, use the `ModelException` class only.

5 Designing Process Definitions

This chapter provides programming examples, using the Model API, for designing process definitions and starting process instances.

The examples include:

- Logging in/logging out a normal user
- Creating a process definition with a Start Node, Activity Node and Exit Node
- Starting a process instance
- Executing work items

5.1 Designing a Simple Process Definition

To get started you can build a simple process definition with a Start Node, an Activity Node and an Exit Node, using the Model API.

You can find the complete programming code of the examples presented in this section in the `SimplePlan.java` sample file.

Before creating a process definition, a user must be logged in, and a workflow appliction selectd. Afterwards he/she can create a simple process definition, whose basic structure consists of one and only one Start Node, an Activity Node and at least one Exit Node. Logging in a user means to create a session, i.e. a `WFSession` object. This session is finished when a user is logged out again.

The following figure shows the necessary steps for designing a process definition using the Model API:

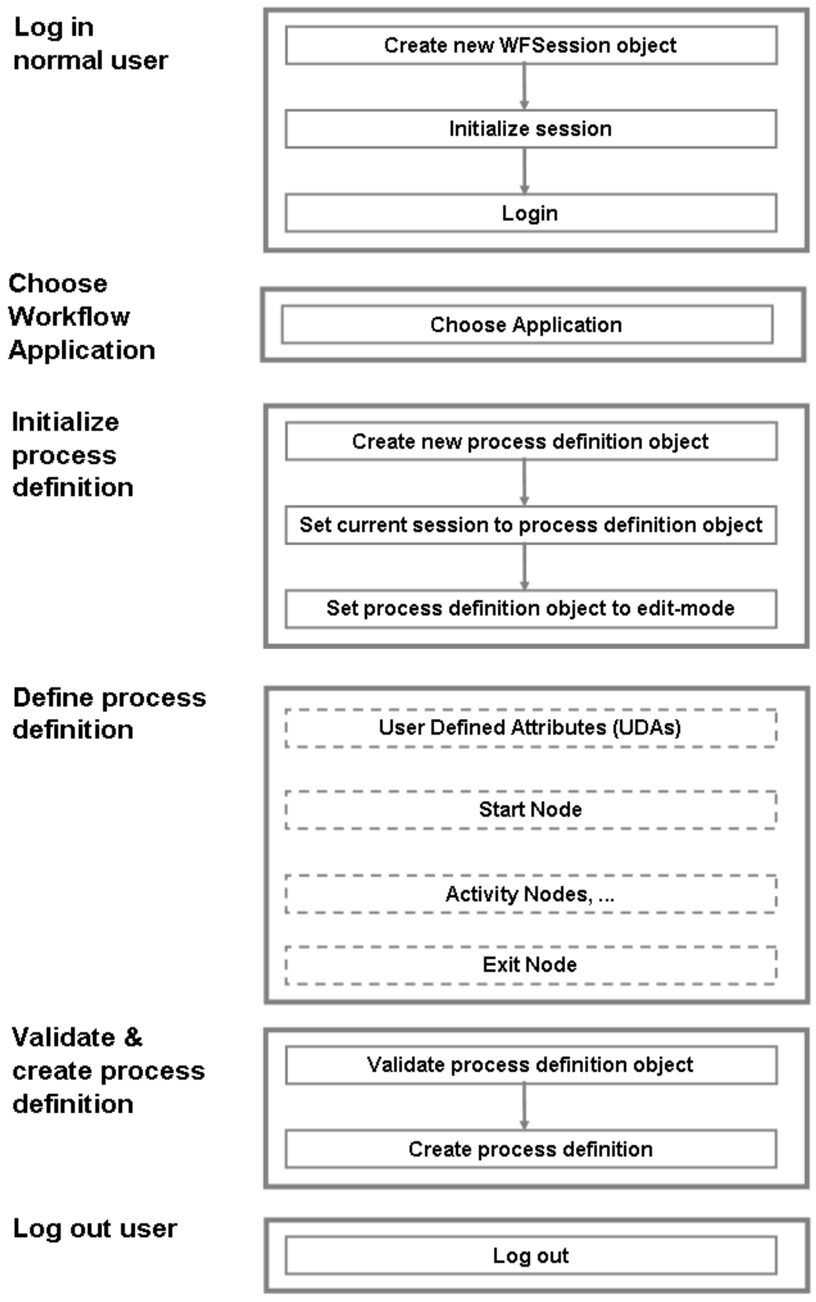

Log in normal user
Create new WFSession object
Initialize session
Login
Choose Workflow Application
Choose Application
Initialize process definition
Create new process definition object
Set current session to process definition object
Set process definition object to edit-mode
Define process definition
User Defined Attributes (UDAs)
Start Node
Activity Nodes, ...
Exit Node
Validate & create process definition
Validate process definition object
Create process definition
Log out user
Log out

Figure 7: Designing a Process Definition Using the Model API

5.1.1 Logging in/Logging out a Normal User

Before working with process definitions, a tenant user must be logged in to the Interstage BPM Server. When working with process definitions is finished, he/she must log out again.

To log in/log out a tenant user:

1. Create a new `WFSession` object.

```
WFSession session;
session = WFObjectFactory.getWFSession();
```

   Use the `WFObjectFactory` class for accessing workflow objects. `getWFSession()` then creates a `WFSession` object.

2. Initialize the session with the appropriate configuration file.

   You can use the default `iFlowClient.properties` file located in `<Interstage BPM Server Installation Directory>/client`.

   Either use this file or write the configuration parameters into a new file, which must be located in the current runtime directory. For the location of the current runtime directory, refer to section *Location of Properties Files* on page 61.

Note: Ensure you specify the tenant name logged in for the property file used. The tenant name is specified for value of `WFObjectFactory.TENANT_NAME(TenantName)`. `TenantName=Default` is specified for login to `Default` tenant.

Note: In the `iFlowClient.properties` file, any backslashes "\" or colons ":" are escaped by backslashes. For example, a server address is specified like this:

`ibpmhost\:49950`

When loading the `iFlowClient.properties` file using the `java.util.Properties.load()` method, escape characters will automatically be taken into account. If you use another way to load the properties, make sure that you handle any escape characters correctly. For details about escape sequences that may occur in the `iFlowClient.properties` file, refer to the Java documentation of the `java.util.Properties.store()` method.

   Load the configuration file `iFlowClient.properties` for the session:

```
Properties sessionProps = new Properties();
     sessionProps.load(new
FileInputStream("../classes/iFlowClient.properties"));
```

   Initialize the session:

```
session.initForApplication(null, sessionProps);
```

3. Log in a user.

```
String server = sessionProps.getProperty("HostName") + "Flow";
   session.logIn(server, userName, password);
```

   For implementation reasons, always attach the constant `"Flow"` to the given host name.

4. After work with the process definition is finished, the user has to log out from the `WFSession`. To log out a user:

```
if (session != null ) {
   session.logOut();
   }
```

5.1.2 Choosing a Workflow Application

After logging in, choose a workflow application within which you want to operate, using `WFSession.chooseApplication()`

```
session.chooseApplication(myApplicationID);
```

Note: If the `ApplicationModeSecurity` server parameter is set to `Relax`, choosing an application is optional. To know about detailed behavior when an application is not chosen, refer the `ApplicationModeSecurity` parameter in the *Interstage BPM Server Administration Guide.*

5.1.3 Designing a Process Definition with Start Node, Activity Node and Exit Node

After a user is logged in, the user can design a new process definition. A simple process definition consists of only one Start Node, of one or more Activity Nodes and of one or more Exit Nodes.

Note: Before creating a process definition, ensure you select an application.

A **Start Node** identifies the start of a process. Every process definition must have one and only one Start Node. An **Activity Node** represents a step in the process where human intervention is planned. A process definition may have any number of Activity Nodes. An **Exit Node** identifies the end of a process. Every process definition must have at least one Exit Node.

Each node can be connected to each other with arrows. The fundamental purpose of an arrow is to control the flow within a process. When a node completes, the process instance moves from the tail of an arrow to the head of another arrow. There is no limit to the number of arrows that can enter or leave a node (except for Start Nodes, which can only have an unlimited number of arrows leaving them, or Exit Nodes, which can only have an unlimited number of arrows entering them).

To design a process definition:

1. Create an empty process definition object with `WFObjectFactory.getPlan()`. Set the current session to the process definition object with `Plan.setWFSession()`.

```
Plan plan = null;
plan = WFObjectFactory.getPlan();
plan.setWFSession(session);
```

2. Before adding nodes and arrows to a process definition, change the current mode of the process definition to edit-mode with `startEdit()` from the `Plan` interface.

```
plan.startEdit();
```

3. You can add some general information to the process definition with `setName()`, `setTitle()`, and `setDesc()` from the `Plan` interface.

```
plan.setName("My Plan");
plan.setTitle("My first process definition");
plan.setDesc("Creation of a simple process definition");
```

4. Add the Start Node, Activity Node, and Exit Node to the process definition object.

   For adding nodes use `addNode(name, nodeType)` from the `Plan` interface. The constant `nodeType` defines the node type you want to add. You find the possible values for this constant in the `Node` interface.

   `setPosition()` from the `Node` interface defines the position of a node. Each node has an X and Y coordinate to show where it resides on the canvas if it is shown graphically. This positioning information is implementation-specific, so it will depend on how the node is represented. When setting this parameter, choose a coordinate system that is appropriate for your implementation.

   a) **To add a Start Node**:

   ```
   Node startNode = plan.addNode("Start", Node.TYPE_START);
   startNode.setPosition(new Point(100, 150));
   ```

   b) For Activity Nodes you can assign a User Group with `setRole()` from the `Node` interface. In addition, you can define whether the Activity Node will generate a group work item when it becomes active, or individual work items for every member of the Group assigned to the Activity node. This setting also depends on the value of the `SupportGroupWorkItem` parameter of the Interstage BPM Server. By default, this parameter is set to `false`. Refer to the *Interstage Business Process Manager Server Administration Guide* for details on the `SupportGroupWorkItem` parameter.

   To add an Activity Node :

   ```
   Node activityNode = plan.addNode("Activity", Node.TYPE_ACTIVITY);
   activityNode.setPosition(new Point(200, 250));
   activityNode.setRole("SampleGroup");
   activityNode.markForExpandGroups(false);
   ```

   Only Activity Nodes and Voting Activity Nodes support assigning Groups.

   c) **To add an Exit Node**:

   ```
   Node exitNode = plan.addNode("Exit", Node.TYPE_EXIT);
   exitNode.setPosition(new Point(300, 350));
   ```

5. Connect nodes with arrows using `addArrow()` from the `Node` interface. When adding an arrow, specify the name, source and target of the arrow.

```
Arrow goArrow = plan.addArrow("go", startNode, activityNode);
Arrow stopArrow = plan.addArrow("stop", activityNode, exitNode);
```

6. After adding nodes and arrows to a process definition object, you need to validate it with `validatePlan()` from the `Plan` interface.

```
plan.validatePlan();
```

7. Create a process definition with `createProcessDef()` from the `Plan` interface.

```
plan = plan.createProcessDef();
```

Note: A process definition will not be created on the Interstage BPM Server until you use `createProcessDef()`. At this point, when the process definition object is committed to the Interstage BPM Server, the Interstage BPM Server assigns a process definition ID to it.

Note: If you are in SaaS mode, you must choose an application before working with a process definition; if you are in non-SaaS mode, it is not mandatory to choose an application before working with a process definition (except while creating it).

The figure below illustrates the resulting process definition.

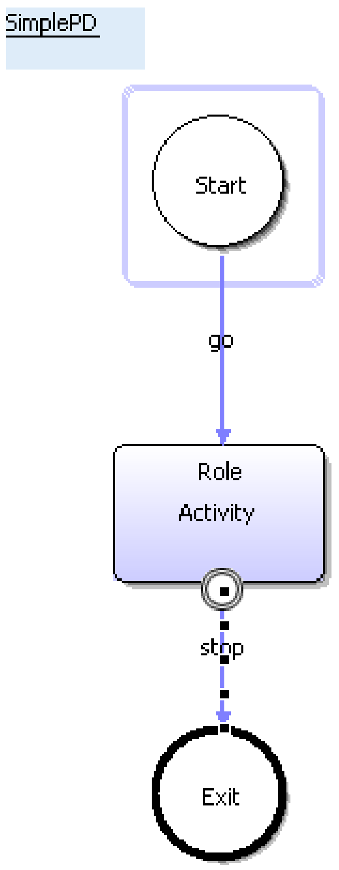


Figure 8: Simple Process Definition with a Start Node, Activity Node and Exit Node

5.2 Designing a Complex Process Definition

This section provides programming instructions for creating a complex process definition with nearly all possible components. You can find the complete programming code of the examples presented in this section in the `ComplexPlan.java` sample file. The following figure illustrates how the workflow elements defined in the sample file interact.

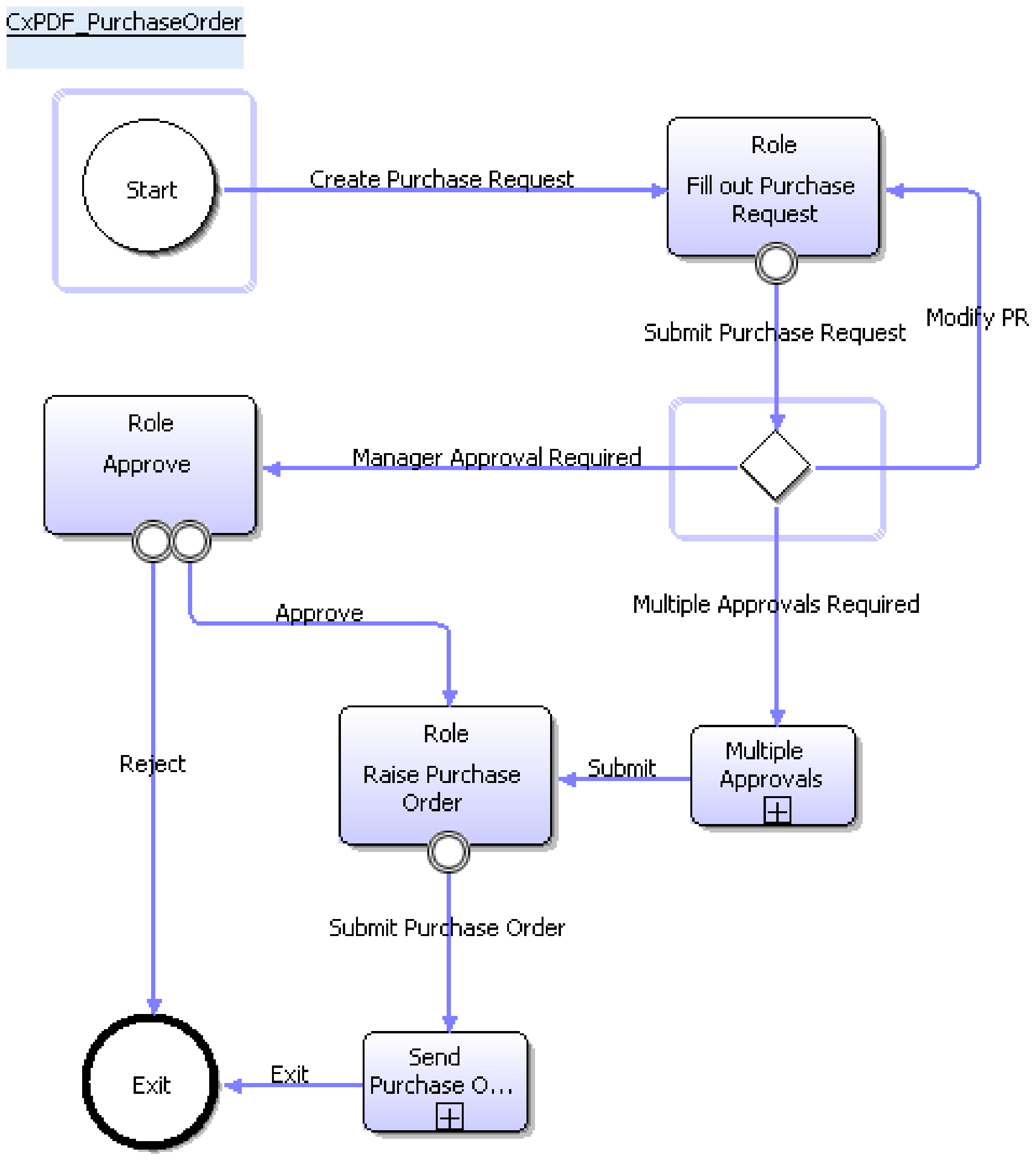


Figure 9: A Complex Process Definition with Different Node Types

In the previous section, you learned how to add basic workflow elements like Start Node, Exit Nodes, and Activity Nodes. The following sections explain how to add User Defined Attributes and how to add additional node types like Voting Activity Nodes, AND Nodes, OR Nodes, and so on.

To design a complex process definition:

1. Log in a user.

Refer to section *Logging in/Logging out a Normal User* on page 65 for information about to how to log in and log out a user.

2. Use `startEdit()` from the `Plan` interface to set the edit-mode for a process definition.

   Once in edit-mode, the Interstage BPM Server locks the process definition so that no other user can make changes to it while an edit session is running. Only draft or private process definitions can be edited. Draft process definitions can be edited only if there are no current running process instances associated with them.

3. Add a Start Node, at least one Exit Node and the required Activity Nodes.

   Refer to section *Designing a Process Definition with Start Node, Activity Node and Exit Node* on page 66 for information about how to add these node types.

4. Add User-Defined Attributes to the process definition. Add the other node types like Voting Activity Nodes, AND Nodes, and so on.

   For details, refer to the following sections.

5. Connect the nodes with arrows.

   For details, refer to section *Designing a Process Definition with Start Node, Activity Node and Exit Node* on page 66.

6. Validate the process definition and create it.

   For details, refer to section *Designing a Process Definition with Start Node, Activity Node and Exit Node* on page 66.

7. Check if you want to make use of the extended features for enhancing your business processes, such as Java Actions, Extended attributes, agents, timers, error handling, etc.

   For details, refer to chapter *Enhancing Interstage Business Process Manager* on page 88.

5.2.1 Adding User Defined Attributes

The `DataItemRef` interface is used to hold the identifier, the name, type, and the initial values of User Defined Attributes (UDAs) defined in the process definitions.

UDAs are variables that are global to the process instance, so that all nodes within the process instance can access all of the UDAs. UDAs specify the behaviour of nodes and store data for process execution.

UDAs have one of the following data types: BIGDECIMAL, BOOLEAN, DATE, FLOAT, INTEGER, LONG, STRING, XML.

Note: A UDA has a name and an identifier.

The UDA name

- is user-defined
- may contain any character (maximum: 64 characters)
- must not start with two underscores (__), because two underscores are used as prefix of UDAs created and maintained by the system.

The UDA identifier

- is either user-defined or generated by the system. If you do not specify an identifier when creating a UDA, the identifier is automatically generated as follows: The name is taken as input; all special characters are removed. If this "sanitized name" (identifier) is longer than 32 characters, empty or not unique, the identifier will be composed of the prefix 'uda<number>', e.g. 'uda1'. The number is increased by one each time a new UDA is added that fulfills the above conditions.
- may consist of a maximum of 32 characters
- may not contain any special characters (all characters except 'a' - 'z', 'A' - 'Z', '0' - '9')
- must be unique throughout the entire process definition

For every user interaction, the UDA name is to be used. Whenever necessary, the name is automatically mapped to the identifier. Once created, the UDA identifier cannot be changed anymore; the UDA name can, however, be changed any time.

To add UDAs to a process definition:

1. Make sure that you have changed the current mode of the process definition to edit-mode with `startEdit()` from the `Plan` interface.
2. To add UDAs, use `addDataItemRef()` or `addDataItemRefWithId()` from the `Plan` interface.

   Using the first method will generate an identifier for the UDA; using the second method, you can define your own identifier.

Example

```
protected final static String WLUDA_PRICE = "Price";
protected final static String WLUDA_QTY = "Qty";
protected final static String WLUDA_TOTAL = "Total";

DataItemRef udaPrice = plan.addDataItemRef(WLUDA_PRICE,
             DataItemRef.TYPE_FLOAT, "0.0");
DataItemRef udaQty = plan.addDataItemRef(WLUDA_QTY,
             DataItemRef.TYPE_INTEGER, "0");
DataItemRef udaTotal = plan.addDataItemRef(WLUDA_TOTAL,
             DataItemRef.TYPE_FLOAT, "0.0");
DataItemRef udaJavaActionTest = plan.addDataItemRefWithId(
           "JavaActionTest", "My JA Test", DataItemRef.TYPE_STRING, "0");
DataItemRef udaMapping = plan.addDataItemRefWithId(
            ("MapUDAParent", "Mapping UDA Parent",
             DataItemRef.TYPE_STRING,
             "This value is from parent process");
DataItemRef udaCondition = plan.addDataItemRef("Condition",
             DataItemRef.TYPE_STRING, "");
```

```
DataItemRef udaSec = plan.addDataItemRef("SEC",
             DataItemRef.TYPE_STRING, "sec");
```

Note: Use the `isIdentifierUnique(String ID)` method from the `Plan` interface for checking whether a user-defined identifier is unique within the entire process definition.

5.2.2 Using Voting Activity Nodes

A Voting Activity Node uses voting rules to determine the choice (activity's outgoing arrow) that wins. For every Voting Activity Node you specify the rules for voting.

To add a Voting Activity Node:

1. Create a node using `addNode()`. Set the constant `nodeType` to `TYPE_VOTING_ACTIVITY`.

```
Node directorApproveNode = plan.addNode("Approve",
   Node.TYPE_VOTING_ACTIVITY);
directorApproveNode.setRole(userGroup);
directorApproveNode.setPosition(new Point(290, 350));
```

2. Define the rules and the threshold for the Voting Activity Node using `setVotingRule()` from the `Node` interface.

   Choose between the following types of rules:

   - `NUMBER`
   - `PERCENTAGE`
   - `MAJORITY`

   The threshold defines the value when a vote is performed.

```
directorApproveNode.setVotingRule("Reject",
   VotingRule.TYPE_NUMBER, 1);
directorApproveNode.setVotingRule("Approve",
   VotingRule.TYPE_NUMBER, 1);
```

   The sample code specifies a voting rule of type `NUMBER` with the threshold 1 for both outgoing arrows. This means, if one user makes a choice, the action associated with the arrow is performed.

3. Specify the time when the votes are checked using `setEvaluateRulesMode()`.

   You can choose between the following:

   - `Node.ON_EVERY_VOTE`: Conditions will be checked after every vote.
   - `Node.WHEN_ALL_VOTES_ARE_CAST`: Conditions will be checked after all users of the assigned Group have voted.

```
directorApproveNode.setEvaluateRulesMode(Node.ON_EVERY_VOTE);
```

4. Define the default choice for the voting rule with `setDefaultChoice()`.

```
directorApproveNode.setDefaultChoice("Reject");
```

5.2.3 Adding AND Nodes and OR Nodes

An AND Node represents a step in a process where the process instance pauses to synchronize multiple threads of execution. A process definition can have any number of AND Nodes.

An OR Node represents a step in a process from where a single branch of control splits into multiple parallel branches. A process definition can have any number of OR Nodes.

- **To add an AND Node:**

  Use `addNode()` and set the constant `nodeType` to `TYPE_AND`.

```
Node andNode = plan.addNode("And Node", Node.TYPE_AND);
andNode.setPosition(new Point(440, 460));
```

- **To add an OR Node:**

  Use `addNode()` and set the constant `nodeType` to `TYPE_OR`.

```
Node orNode = plan.addNode("Or Node", Node.TYPE_OR);
orNode.setPosition(new Point(430, 230));
```

5.2.4 Using Conditional Nodes

A Conditional Node is an Activity Node with a Prologue Action. After adding the Conditional Node to the process definition you first have to define a Prologue Action and then you have to specify the condition for the node.

To add a conditional node:

1. Use `addNode()` and set the constant `nodeType` to `TYPE_CONDITION`.

```
Node CondNode = plan.addNode("Conditional Node",
                   Node.TYPE_CONDITION);
CondNode.setPosition(new Point(430, 140));
```

2. Define the Prologue Actions. Refer to chapter *Enhancing Interstage Business Process Manager* on page 88 for details.
3. Define the conditions for the Conditional Node using `getConditionSpec()` and `setCondBranchSpecInfo()` from the `Node` interface.

Example

In the following example, three conditions for the outgoing arrows of a Conditional Node are defined. The values of the User Defined Attribute (UDA) `Total` represent the borders for a given budget:

- Modify PR: Total <= 0.0 (default)
- Manager Approval Required: Total > 0.0 and Total < 1000.0
- Multiple Approvals Required: Total > 1000.0

It is not possible to use the UDA `Total` to decide which branch is used, because it contains two conditions, but only one condition can be defined per branch. Therefore a further User Defined Attribute `Condition` is introduced. Depending on the UDA `Total` the UDA `Condition` is set to one of three unique values. These values are required to control which arrow is used. A Java Action, named `SampleJavaActions` in this example, is used to wrap the current value of the UDA `Total` to the UDA `Condition`, which is then checked.

The conditions for the wrapping are:

```
if (Total <= 0.0) {
   Condition = Modify PR;
} else if (Total > 0.0 && Total <= 1000.0) {
   Condition = "Manager Approval Required";
```

```
} else if (Total > 1000.0) {
   Condition = "Multiple Approvals Required";
} else {
   Condition = "Modify PR";
```

The `getConditionSpec()` from the `Node` interface which is defined for this Conditional Node uses this User Defined Attribute (UDA) to retrieve information about the arrow it has to choose. This is necessary because only one `ConditionSpec` object can be defined containing only one UDA. With `setCondBranchSpecInfo()` from the `ConditionSpec` interface you can add a new branch to the `ConditionSpec` object.

```
ConditionSpec conditionSpec = condNode.getConditionSpec();
conditionSpec.setConditionAttribute("Condition");

conditionSpec.setCondBranchSpecInfo("Modify PR",
   BranchSpec.EQUAL_OP, "Modify PR", true );
conditionSpec.setCondBranchSpecInfo("Multiple Approvals Required",
   BranchSpec.EQUAL_OP,
   "Multiple Approvals Required", false );
conditionSpec.setCondBranchSpecInfo("Manager Approval Required",
   BranchSpec.EQUAL_OP, "Manager Approval Required", false );

condNode.setConditionSpec(conditionSpec);
```

The Interstage BPM Model API also provides a method for accessing values of XML substructures: `setConditionAttribute(String udaName, String xPath)` as well as for retrieving the value of an xPath by calling the `getConditionAttributeXPath()` method. Refer to the API Javadoc for detailed information.

5.2.5 Using Subprocess Nodes

A Subprocess Node represents a step in a process where a task is accomplished by invoking another process definition, and waiting for a result to come back. This node can be used to serve multiple purposes, especially for reusing existing process definitions within new ones.

A single Subprocess Node can represent an entire process definition that is already designed, accessing all of the nodes, arrows, and other characteristics of that process definition. The Subprocess Node can then call an already existing process definition via its name.

In the example file `ComplexPlan.java` a Subprocess Node calls a process definition with the name `CxPD_Approval`. The following figure illustrates this process definition:

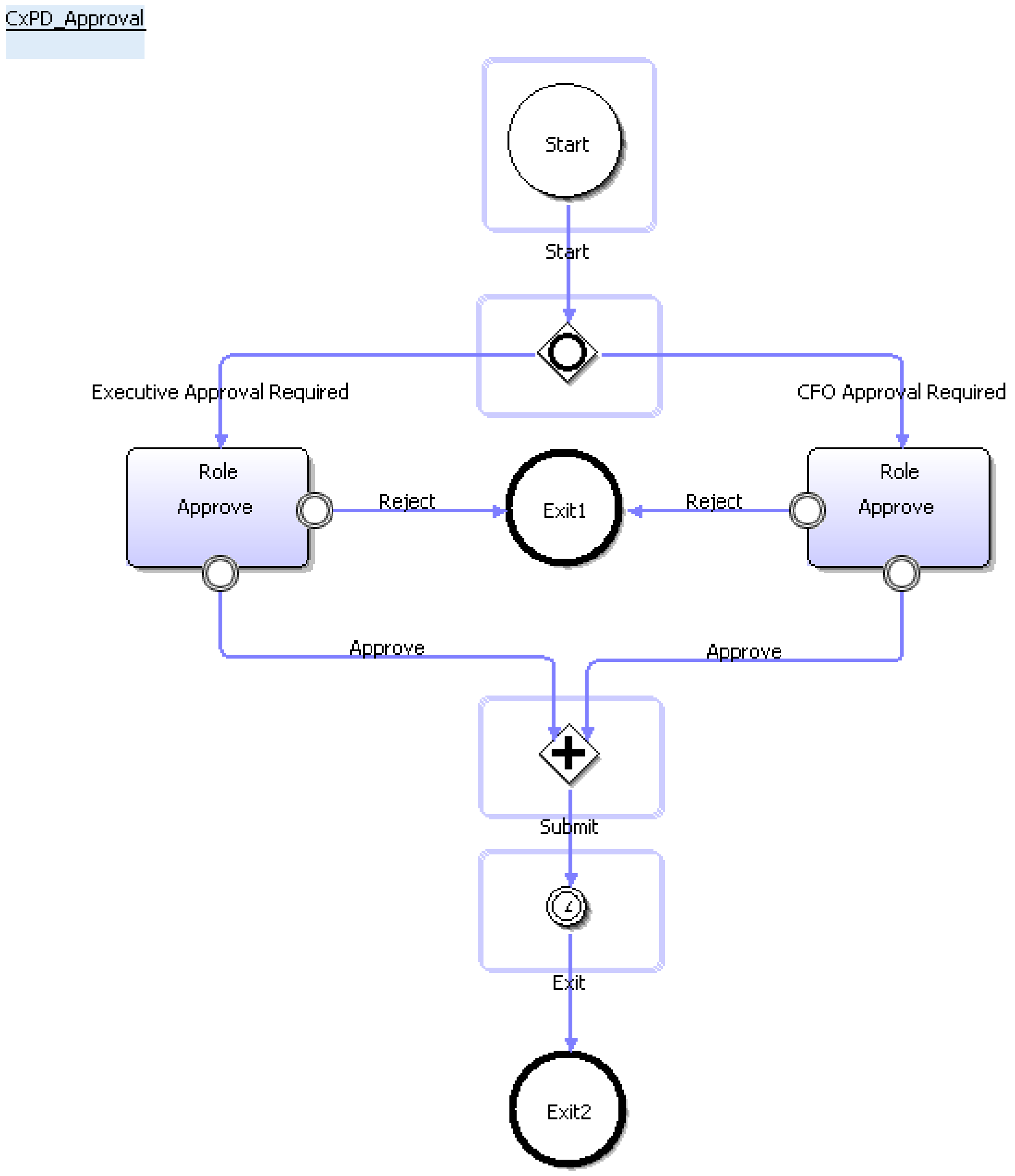


Figure 10: A Process Definition with Different Node Types called from a Subprocess Node

To use a subprocess node:

1. Add a Subprocess Node to the process definition. To do so, use `addNode()` and set the constant `nodeType` to `TYPE_SUB_PROCESS`.

```
Node subProcessNode = plan.addNode("Multiple Approvals",
    Node.TYPE_SUB_PROCESS);
subProcessNode.setPosition( new Point(406, 237));
```

2. Specify the name of an existing process definition for which a process instance is to be generated when the Subprocess Node is called. To do so, use `setSubPlanName()` from the `Node` interface.

```
subProcessNode.setSubPlanName(subPlan.getName());
```

A new process instance of the **latest version** of this subprocess definition will automatically be generated.

Note: If you need to use a subprocess definition of the **same version** as that of the parent process definition, use the `setSameSubPlanVersion(boolean)` method of the `com.fujitsu.iflow.model.workflow.Plan` interface to do this. For example, for a parent process definition 'A' of version 2, you can use the method mentioned above to use only version 2 (and not the latest version) of all its subprocesses. Sample code to set such functionality for a parent process definition is as follows:

```
plan.startEdit();
plan.setSameSubPlanVersion(true);
plan.commitEdit();
```

If you set this feature for a parent process, ensure that for a version of a parent process definition, subprocess definitions having that same version exist. If the parent process instance tries to call a subprocess definition with a version number that does not exist, an error is thrown.

While updating an application using the `WFSession.updateApplication()` method, if you want to use the same versions of the subprocess definitions, you need to update the parent process definition as well as the child process definitions.

3. Map the flow of information between a User Defined Attribute (UDA) in the parent process definition and a UDA in the subprocess definition using `addDataMappingElement()`.

A `DataItemMappingElement` that contains a UDA in a parent process definition mapped to a UDA in a subprocess definition is called "input data mapping element".

```
subProcessNode.addDataMappingElement("MappingUDAParent",
   "MappingUDAChild", DataItemMappingElement.INOUT);
```

Note: Be careful when designing process definitions that have recursive subprocesses. Check all process definitions involved and make sure that there are no infinite recursions.

5.2.6 Using Delay Nodes

A Delay Node represents a step in a process where process execution is suspended for a certain amount of time.

To use a Delay Node:

1. Add a Delay Node to the process definition. To do so, use `addNode()` and set the constant `nodeType` to `TYPE_DELAY`.

```
Node delayNode = plan.addNode("Delay Node", Node.TYPE_DELAY);
delayNode.setPosition(new Point(434, 500));
```

2. Add a timer to the Delay Node to specify how long the process execution will be suspended.

   Refer to section *Defining a Timer* on page 150 for an example.

5.2.7 Using Chained-Process Nodes

A Chained-Process Node represents a step in a workflow process where a new process instance is created to accomplish a task independent of the parent process instance. This node enacts this independent process instance as a part of the execution of the parent process instance. The Chained-Process Node can only call an already existing process definition via its name.

In the example file `ComplexPlan.java` a Chained-Process Node calls a process definition with the name `CxPD_Send`. The following figure illustrates this process definition.

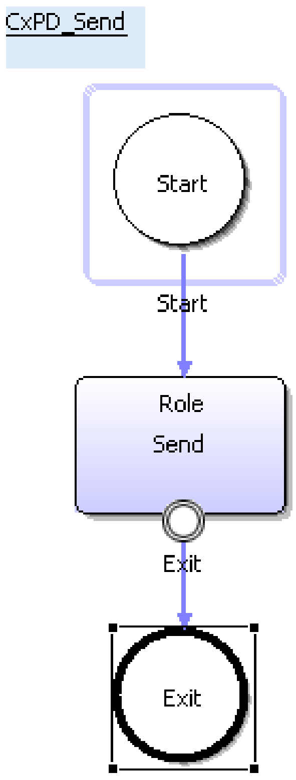


Figure 11: A Process Definition Called from a Chained-Process Node

To use a Chained-Process Node:

1. Add a Chained-Process Node to the process definition. To do so, use `addNode()` and set the constant `nodeType` to `TYPE_CHAINED_PROCESS`.

```
Node chainedProcessNode = plan.addNode("Send Purchase Order",
Node.TYPE_CHAINED_PROCESS);
chainedProcessNode.setPosition( new Point(212, 391));
```

2. Specify the name of an existing process definition for which a process instance is to be generated when the Chained-Process Node is called. To do so, use `setChainedPlanName()` from the `Node` interface.

```
chainedProcessNode.setChainedPlanName(chainedPlan.getName());
```

A new process instance of the **latest** version of this chained-process definition will automatically be generated.

Note: If you need to use a chained-process definition of the **same version** as that of the parent process definition, use the `setSameSubPlanVersion(boolean)` method of the `com.fujitsu.iflow.model.workflow.Plan` interface to do this. For example, for a parent process definition 'A' of version 2, you can use the method mentioned above to use only version 2 (and not the latest version) of all its chained-processes. Sample code to set such functionality for a parent process definition is as follows:

```
plan.startEdit();
plan.setSameSubPlanVersion(true);
plan.commitEdit();
```

If you set this feature for a parent process, ensure that for a version of a parent process definition, chained-process definitions having that same version exist. If a parent process instance tries to call a chained-process with a version number that does not exist, an error is thrown.

While updating an application using the `WFSession.updateApplication()` method, if you want to use the same versions of the subprocess definitions, you need to update the parent process definition as well as the child process definitions.

3. Map the flow of information between a User Defined Attribute (UDA) in the parent process definition and a UDA in the chained-process definition using `addDataMappingElement()`.

   A `DataItemMappingElement` that contains a UDA in a parent process definition mapped to a UDA in a chained-process definition is called "input data mapping element".

```
chainedProcessNode.addDataMappingElement("MappingUDAParent",
   "MappingUDAChild", DataItemMappingElement.INOUT);
```

5.3 Working with Process Instances

The following sections provide instructions for retrieving a process definition and starting a process instance from it. In addition, the execution of work items, which are generated within a process instance, is shown. You can find the complete programming code of the examples presented in these sections in the sample file `ProcessExecution.java`.

Before doing any work on process instances and work items, the user must be logged in. Refer to section *Logging in/Logging out a Normal User* on page 65 for information about how to log in and log out a user.

If you are in SaaS mode, it also necessary to choose an application before working with process instances.

The following figure shows the necessary steps when working with process instances:

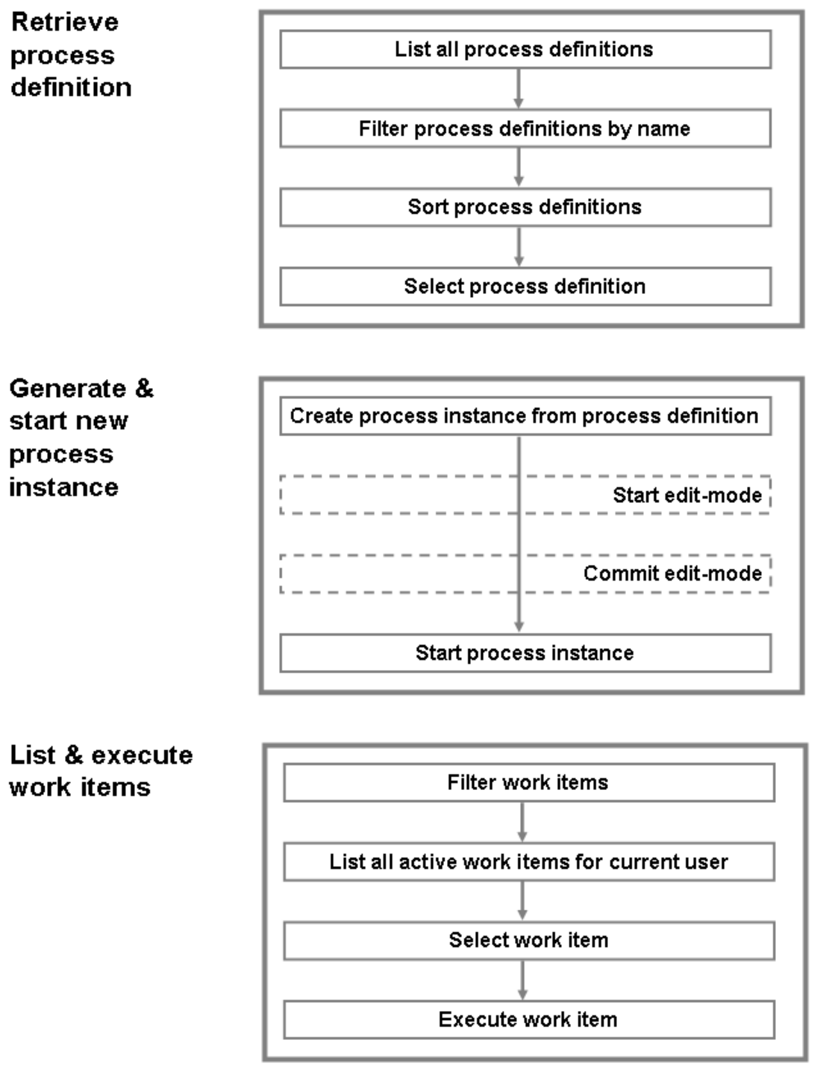

Retrieve process definition
List all process definitions
Filter process definitions by name
Sort process definitions
Select process definition
Generate & start new process instance
Create process instance from process definition
Start edit-mode
Commit edit-mode
Start process instance
List & execute work items
Filter work items
List all active work items for current user
Select work item
Execute work item

Figure 12: Working with Process Instances Using the Model API

5.3.1 Retrieving the Latest Version of a Process Definition

To retrieve the latest version of a process definition:

1. List all process definitions that are available on the Interstage BPM Server. To do so, use the `WFObjectList` interface from the package `com.fujitsu.iflow.model.workflow` to get a list of the existing `WFObjects` from the Interstage BPM Server.

   The `WFObjectFactory` class provides you with consistent, non-proprietary means of accessing the workflow objects.

```
Plan plan = null;
WFObjectList wfObjectList =
   WFObjectFactory.getWFObjectList(session);
Object[] planList = null;
```

2. Filter the process definitions according to their names. To do so, use `addFilter()` from the `WFObjectsList` interface to add filtering criteria to the list of the process definitions to be retrieved.

```
wfObjectList.addFilter(WFObjectList.LISTFIELD_PLAN_NAME,
WFObjectList.SQLOP_EQUALTO, "\'" + planName + "\'");
```

Note: Include the process definition name in double quotes `''`, because it is directly included in the generated database query.

3. Sort the process definitions according to a desired criteria. Use `addSortOrder()` to add a field-based sort order to the list of the process definitions to be retrieved.

```
wfObjectList.addSortOrder(WFObjectList.LISTFIELD_PLAN_ID,
   WFObjectList.SORTORDER_DESCENDING);
wfObjectList.openBatchedList(Filter.AllPlans);

planList = wfObjectList.getNextBatch(1);
```

   In the code sample, process definitions are sorted by their ID. `openBatchedList()` returns a list of process definitions from the Interstage BPM Server matching the filter criteria. `getNextBatch(1)` retrieves the next entry only, because we only search for the latest version of the process definition.

4. Use the process definition with the latest version.

```
if (planList != null && planList.length > 0) {
   plan = (Plan) planList[0];
}
return plan;
```

   The first entry of the returned list is the searched process definition version.

5.3.2 Creating and Starting a New Process Instance

After you have retrieved a process definition, you can create a process instance from it and start the process instance.

To create and start a new process instance:

1. Create a process instance using `createProcessInstance()` from the `Plan` interface.

```
processInst =
WFObjectFactory.createProcessInstance(plan.getId(),session);
```

   `createProcessInstance()` copies the process definition's structure and attributes to a running process instance, making it appear as if it had been instantiated.

   When a process definition is copied to a new process instance, the components of the process definition are also copied as components of the new process instance. For example, a `StartNode` object in the process definition is copied to a `StartNodeInstance` object in the new process instance.

2. You can edit the parameters of the process instance by setting it to edit-mode with `startEdit()`.
3. When the editing is finished, stop edit-mode with `commitEdit()`.

   Otherwise the process instance is locked for further operations.

4. Start the process instance.

```
processInst.start();
```

5.3.3 Listing Work Items

For listing work items, use the `WFObjectsList` interface from the `com.fujitsu.iflow.model.workflow` package. This interface retrieves a list of existing `WFObjects` from the Interstage BPM Server.

To list work items:

1. Build a filter using the `Filter` class.

   Possible filter criteria for work items are:

   - `AllWorkItems`: Retrieves all work items. This is the main filter for retrieving group work items.
   - `MyAcceptedWorkItems`: Retrieves all work items including group work items accepted by the logged-in user.
   - `MyActiveWorkItems`: Retrieves all active work items belonging to the logged-in user.
   - `MyCompletedWorkItems`: Retrieves all work items completed by the logged-in user.
   - `MyDeclinedWorkItems`: Retrieves all work items declined by the logged-in user.
   - `MyWorkItems`: Retrieves all work items belonging to the logged-in user.
2. Use `openBatchedList()`.
3. Call `getNextBatch()`.

   `getNextBatch()` returns an array of workflow objects matching the filter criteria set with `openBatchedList()`.

Example

```
WorkItem[] workItemList = null;
if (filter == Filter.AllWorkItems
|| filter == Filter.MyAcceptedWorkItems
|| filter == Filter.MyActiveWorkItems
|| filter == Filter.MyCompletedWorkItems
|| filter == Filter.MyDeclinedWorkItems
|| filter == Filter.MyWorkItems) {
   WFObjectList wfObjectList =
   WFObjectFactory.getWFObjectList(session);
   wfObjectList.openBatchedList(filter);
   int batchSize = 50;
   Object[] elements = wfObjectList.getNextBatch(batchSize);
}
```

The filter `Filter.MyActiveWorkItems` is not supported for group work items. If the `SupportGroupWorkItem` parameter of the Interstage BPM Server is set to true, you cannot retrieve work items using this filter. To retrieve work items, use the filter `Filter.AllWorkItems` instead. For advanced filtering based on the name of the group to which the group work item belongs, use `LISTFIELD_WORKITEM_ASSIGNEE` together with SQL operators. Refer to the *Interstage Business Process Manager Server Administration Guide* for details on the `SupportGroupWorkItem` parameter.

Examples for Filtering Group Work Items

Consider UserX who is a member of two user groups: GroupA and GroupB.

Scenario 1: UserX wants to retrieve all group work items that are assigned to GroupA:

```
...
wfObjectList.addFilter (WFObjectList.LISTFIELD_WORKITEM_ASSIGNEE,
WFObjectList.SQLOP_EQUALTO, "GroupA");
wfObjectList.openBatchedList(Filter.AllWorkItems);
...
```

Scenario 2: UserX wants to retrieve all group work items that are assigned to both groups:

```
...
wfObjectList.addFilter (WFObjectList.LISTFIELD_WORKITEM_ASSIGNEE,
WFObjectList.SQLOP_IN, "GroupA,GroupB");
wfObjectList.openBatchedList(Filter.AllWorkItems);
...
```

Scenario 3: UserX wants to retrieve all group work items that are assigned to both groups as well as all individual work items assigned to himself/herself:

```
...
wfObjectList.addFilter (WFObjectList.LISTFIELD_WORKITEM_ASSIGNEE,
WFObjectList.SQLOP_EQUALTO, "GroupA,GroupB,UserX");
wfObjectList.openBatchedList(Filter.AllWorkItems);
...
```

5.3.4 Executing Work Items

Only active work items can be executed.

To execute a work item:

1. Generate a list with all active work items, for example:

```
wfObjectList.addFilter(WFObjectList.LISTFIELD_WORKITEM_ASSIGNEE,
   WFObjectList.SQLOP_IN, "User,Group");
WorkItem[] workItemList = listWorkItems(Filter.AllWorkItems);
```

2. Retrieve the possible choices allowed for a work item using `getChoices()` from the `WorkItem` interface.

```
choices = workItem.getChoices();
```

   The choices are the names of each of the outgoing arrows attached to the Activity Node that the work item represents.

3. Accept the work item by using `accept()` from the `WorkItem` interface.

```
if (choices != null ) {
   workItem.accept();
   int choiceIdx = 0;
   ...
}
```

   `accept()` changes the state of the work item to `STATE_ACCEPTED`, and all other active work items associated with this activity change to `STATE_DEACTIVE`.

4. Execute the work item, i.e. make a choice for it using `makeChoice()` from the `WorkItem` interface.

```
workItem.makeChoice(choices[choiceIdx]);
```

`makeChoice()` captures the specified choice for this work item through a parameter. `makeChoice()` completes the work item.

5.3.5 Recalling Work Items

Only completed work items can be recalled.

The activity that is the target of recall is called the **Recall Target**. The activity that was active prior to the **Recall Target** is called the **Recall Source**.

The status of activities in the recall process undergoes the following changes:

- Recalling a work item results in the deactivation of the target activity and it goes back to the closed state, the only exception is the AND node which continues to remain active.
- The Recall Source then goes into the active state.

Note: Only single level recall of work item is possible. Multiple level recall of work items cannot be done.

The figures below illustrate a successfully recalled work item, and the change in the status of activities before and after recall:

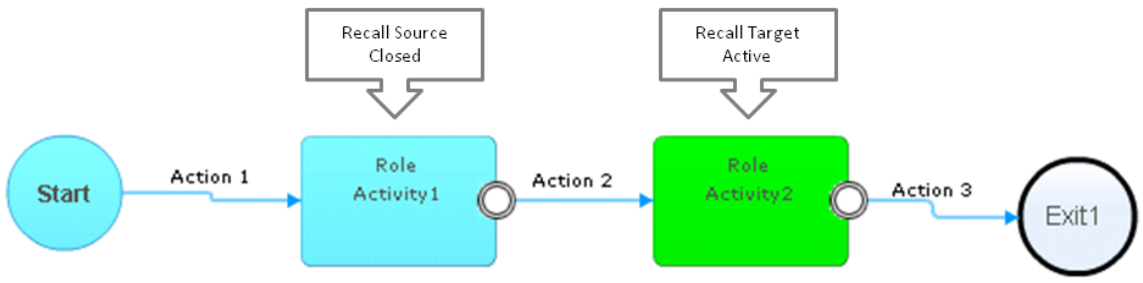


Figure 13: Status of Activities before Recall

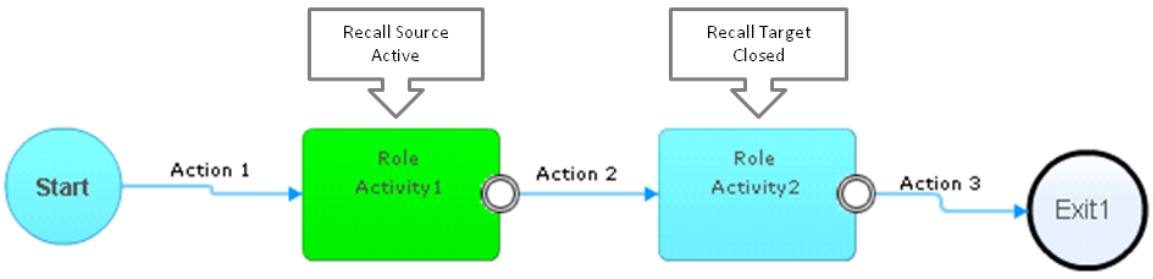


Figure 14: Status of Activities after Recall

In case of activities connected through parallel branches to the AND or OR nodes or leading to a common activity, both the parallelly connected activities cannot be recalled simultaneously. Only one activity can be recalled at a time. The work item for the recalled activity has to be committed and only then can the second activity be recalled.

For instance in the figures below, Activity 2 has been recalled. Activity 1 cannot be recalled until work item for Activity 2, that is, action 4 is committed.

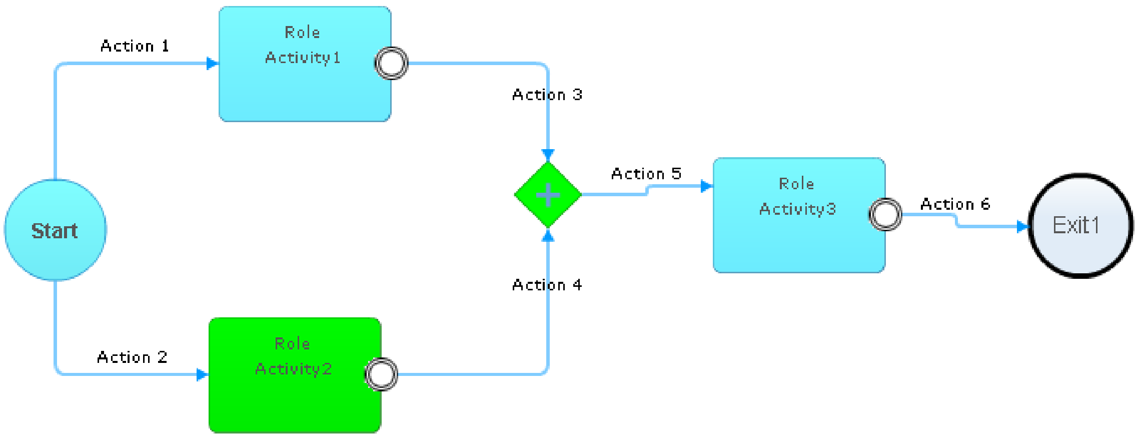


Figure 15: Activities connected parallely to the AND Node

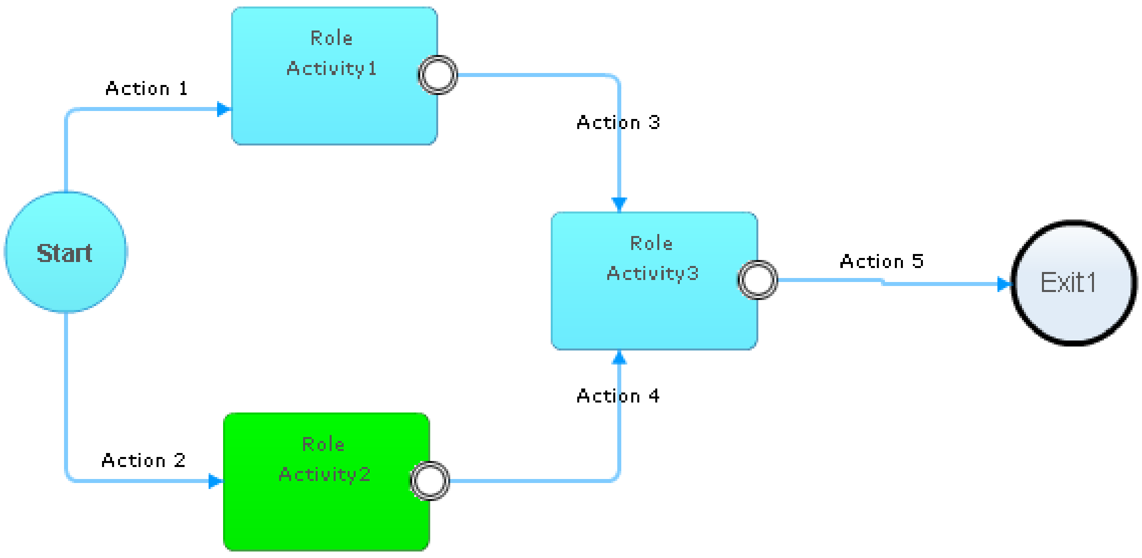


Figure 16: Activities connected parallely leading to a common activity

When a work item is recalled, compensate java actions are executed to roll back operations done by java actions executed before recall.

Conditions for Recalling Work Item

To recall a work item, its imperative that the node types are supported by the recall functionality. However this support is provided only if the relevant conditions for the node types are met.

The following table gives an overview of the node types supported and not supported by the recall functionality and the conditions to be met by different node types in order to be supported as recall targets.

Node Type	Recall Source	Recall Targets	Conditions
Activity Node	Supported	Supported	• It is not an Agent • It has not been accepted • It is in the active state and not in the error or closed state • The process instance is not in the suspended or aborted state
Voting Node	Not supported	Supported	• No voting has been done • The process instance is not in the suspended or aborted state
Route Nodes (AND, OR, Conditional, Complex Conditional, Email, DB, WebService and Customized Nodes)	Not supported	Not supported	-
Subprocess, Chained-process, and RemoteSubprocess Nodes	Not supported	Not supported	-
Delay Node	Not supported	Supported	• It is in the active state and not in the error or closed state • The process instance is not in the suspended or aborted state
Trigger Node	Not supported	Supported	• It is in the active state and not in the error or closed state • The process instance is not in the suspended or aborted state
Exit Node	Not supported	Not supported	-
Start Node	Not supported	Not supported	-

Node Type	Recall Source	Recall Targets	Conditions
Iterator Node	Supported	Supported	• Only Activity Nodes are included in the Iterator Node, that is, no Subprocess or Agent Nodes are included in it • All the Activity Nodes included in the Iterator Node are active • Activity Nodes included in the Iterator Node have not been accepted • The process instance is not in the suspended or aborted state

Note: Route Nodes include the AND, OR, Conditional, Complex Conditional, Email, DB, WebService and Customized Nodes. They cannot become recall targets, however, when they exist between recall source and recall target, recall is possible.

The recall flag is used to disable or enable the recall functionality for an activity. If the recall flag is set to `false`, then recall is enabled, and if it is set to `true` then recall is disabled. To check the status of recall flag, use `isRecallDisabled()`. By default, recall flag is set to `false`. To set the recall flag to `true` or `false` use `setRecallDisabled(boolean)`.

To recall a work item:

1. Generate a list of Completed Work Items.

```
wfObjList.openBatchedList(Filter.MyCompletedWorkItems);
```

2. Select the work Item you want to recall.

```
Object[] elements = wfObjectList.getNextBatch(100);
workItem =(WorkItem)elements[n];
```

3. Recall the source work item.

```
 workItem.recall();
```

   This results in the recall of the selected work item.

5.3.6 Handling Attachments

In the context of a process instance, the `AttachmentRef` interface allows you to access attachments in a specified Document Management System (DMS).

Note: The DMS must be one of the DMS directories specified in the Interstage BPM configuration.

An attachment is referenced by two attributes: name and path. The name attribute is a short descriptive value used for identification. The path attribute describes the fully qualified path to the attachment.

There is no limit to the number of attachments that can be associated with a process instance, and there are no restrictions on the types of attachments that can be added.

Attachments are global to the process instance; any activity in a process instance can access the attachment.

- **To add an attachment:**

  Make sure that the process instance is in enactment edit-mode or in structural edit-mode. Then, construct a new `AttachmentRef` object using `ProcessInstance.addAttachment`.

```
procInst.startEdit();
procInst.addAttachment(ATTACHMENT_NAMES[attachIdx],
   ATTACHMENT_FILES[attachIdx]);
procInst.commitEdit();
```

- **To retrieve all attachments:**

  Use `ProcessInstance.getProcessAttachments` .

```
currentAttachments = procInst.getProcessAttachments();
```

  `getProcessAttachments()` returns the references to all the attachments associated with a process instance. Attachments are added to a process instance while it is running.

- **To retrieve a particular attachment:**

  Use `ProcessInstance.getAttachment`.

```
newAttachment =
procInst.getAttachment(ATTACHMENT_NAMES[attachIdx]);
```

  `getAttachment` returns the `AttachmentRef` object of the attachment associated with the process instance.

The `ProcessExecution.java` sample contains examples for handling attachments. Refer to the provided sample source code and the API Javadoc for details.

6 Enhancing Interstage Business Process Manager

This chapter describes the following means of enhancing and extending Interstage BPM so that you can exploit its full power. Refer to the following sections for details:

6.1 Integrating Interstage BPM with External Applications

Interstage BPM is designed to be integrated with other applications. You can use Java Actions, Agents, or JavaScripts to call methods on classes external to the Interstage BPM Server. There are, however, some simple integration steps that you must take to do this.

Note: Actions, Agents, and JavaScripts must call the `ServerEnactmentContext` interface. The interfaces in the Model API are not accessible to Actions, Agents, or JavaScripts .

To integrate Interstage BPM with external applications:

1. To include external libraries, so they can be used in Interstage BPM, copy your external classes or JARs to the following locations:
    - To be able to use the files across tenants
        - If you use classes separately, copy the class files to `<Interstage BPM Server Installation Directory>/server/instance/default/classes`
        - If you use a custom library, copy the JAR files to the Interstage BPM library extensions directory `<Interstage BPM Server Installation Directory>/server/instance/default/lib/ext`
    - To be able to use the files so that they are specific to a tenant
        - If you use classes separately, copy the class files to `<Interstage BPM Server Installation Directory>/server/instance/default/tenants/<tenant name>/classes`
        - If you use a custom library, copy the JAR files to `<Interstage BPM Server Installation Directory>/server/instance/default/tenants/<tenant name>/lib/ext`
    - To be able to use the files so that they are specific to an application
        - If you use classes separately, copy the class files to `<DMSRoot>/apps/<application ID>/engine_classes`

- If you use a custom library, copy the JAR files to `<DMSRoot>/apps/<application ID>/engine_lib`
- If you use JavaScript files, copy them to `<DMSRoot>/apps/<application ID>/engine_js`

2. To use your own custom classes in a JavaScript, add them to a JAR file and copy the JAR file into the appropriate directory as mentioned in the step above. Then, instantiate the class with the "Packages." notation.

Example of Using a Custom Class in a JavaScript

You want to use `test.class`, a special purpose custom Java class. You add it to a JAR file called `test.jar`, and copy the JAR file into `<Interstage BPM Server Installation Directory>/server/instance/default/lib/ext`. Then, you instantiate `test.class` as follows in your JavaScript:

```
var test = new Packages.test();
```

The class `SalaryCommission` must be located in either of the following paths:

- In the global classpath (`<ServerRoot>/classes` or for libraries `<ServerRoot>/lib/ext`)
- In the tenant classpath (`<ServerSharedRoot>/tenants/<tenant name>/classes` or for libraries `<ServerSharedRoot>/tenants/<tenant name>/lib/ext`)
- In the application classpath (`<DMSRoot>/apps/<application ID>/engine_classes` or for libraries `<DMSRoot>/apps/<application ID>/engine_lib`)

But the javascript uses the classloader of the `ServerEnactmentContext` instance where it is executed in. This instance is created without the knowledge of the action classpath. JavaScriptUtil then uses this classloader and additionally adds the global and if needed the application classpath. Please correct the test and add SalaryCommission to one of the loaded classpaths.

6.2 Using Application Variables

Application Variables allow all users of a particular application to share data between processes in that application. With this ability, more dynamic behavior and various actions associated with it are possible in processes. For example, the Web Service JavaAction can use an Application Variable to designate the Web Service location. This allows end users to change the Web Service location dynamically without changing the JavaAction specification in the process definition.

These variables are defined as part of an application during application development in the Interstage BPM Studio and are available throughout the life cycle of the application. All of the Application Variables for an application are represented in the code as an XML file named Appvariable.xml. It can be found directly under the application. Appvariable.xml contains the variable names and values. The following is an example:

```
<properties>
    <entry key="VariableName1">Value1</entry>
    <entry key="VariableName2">Value2</entry>
    …
</properties>
```

Variables can be created only in the Interstage BPM Studio, but their values can be changed in the Interstage BPM Console. However, the value of an Application Variable can be updated only by the Application Administrator.

Application Variables are especially helpful to developers because they might need a particular setting or a variable to have a common value across all of the processes in that application. These variables are used in much the same way as user-defined attributes (UDAs). The only difference between them is that UDAs are limited to sharing data in a particular running process.

Model API

The following methods in the Model API WFSession package allow manipulation of Application Variables in runtime:

```
public java.util.Properties getApplicationVariables  (String appId) throws
ModelException;
```

This will return a Property object of Application Variables name, value pair.

```
public void setApplicationVariables(String appId, java.util.Properties properties)
 throws ModelException;
```

This method updates the Application Variables.

NOTE: You must be an Application Owner or System Administrator to set Application Variables.

6.3 Using Java Actions

Java Actions are extensions to the workflow engine. At its core, a Java Action is nothing more than a static Java method which Interstage BPM has been configured to know enough about to be able to call, as part of process execution.

Java Actions can be used to extend the workflow engine to do anything that is possible to do in Java. Typically, these Java Actions are used to connect to and interact with external programs and services. Data can be moved from external programs and services to Interstage BPM and vice versa. Java Actions are also used to make use of standard protocols to again access external programs and services.

In addition, you can use Java Actions for dealing with specific error situations or for handling situations where, e.g. a process instance is aborted or suspended by an Interstage BPM Administrator.

The `JavaActionSet` interface of `com.fujitsu.iflow.model.workflow` is a container for `JavaActions`. You add Java Actions, i.e. a set of one or more Java methods, to a process definition at design time.

You can add Java Actions to the process definition itself, to nodes, to timers, and you can add Error or Compensate Java Actions to other Java Actions.

Java Actions can be invoked at the following points of process execution:

- at process initialization (Init Actions and Process Owner Actions)
- at process completion (Commit Actions)
- before an activity starts (Prologue Actions and Role Actions)
- at activity completion (Epilogue Actions)
- when a timer expires (Timer Action)
- when a process instance is aborted, suspended, or resumed (onAbort, onSuspend, or onResume Action)
- when an error occurs during process execution (Error Actions)
- when an error occurs during the execution of a Java Action (Error Actions and Compensate Actions)
- when the starting of a remote subprocess fails (Error Action for a Remote Subprocess Node)

Refer to section *Types of Java Actions* on page 91 for details.

Java Actions run within the context of a process instance. This means that they can read and update User Defined Attributes (UDAs) and other attributes of the particular process instance they are embedded within. When data is moved to a process instance, it is usually read from one or more external data sources and copied into UDAs. When data is moved to an external program or service, the values of UDAs are copied to one or more external data sources.

Note: If Java Actions are supposed to manipulate the process flow, they need access to the process instance context through the Server Enactment Context API. The interfaces in the Model API are not accessible to them.

You can, for example:

- Manipulate process flow through external programs or services. Refer to section *Accessing Workflow Data Using the Server Enactment Context Interface* on page 92 for an example.
- Enhance Interstage BPM with functions that just read context data, e.g. to compare values. Refer to sections *Assigning Prologue Actions to an Activity Node* on page 93 and *Assigning Epilogue Actions to an Activity Node* on page 95 for examples.
- Manipulate process flow through existing Interstage BPM functionality (built-in Java Actions). Refer to section *Using Built-In Java Actions* on page 96 for more information.
- Define specific error handling when an error occurs during the execution of a Java Action. Refer to section *Dealing with Errors in Java Actions* on page 102 for more information.
- Define specific error actions in case a remote subprocess cannot be started. Refer to section *Using Error Java Actions* on page 101 for more information.
- Handle the case when a process instance is aborted, suspended or resumed. Refer to section *Using onAbort, onSuspend, onResume Java Actions* on page 109 for an example.

6.3.1 Types of Java Actions

The `JavaAction` interface allows the inclusion of Java methods in a process definition. Those methods will be executed during the process enactment. Java Actions can be added to the process definition at design time. They can be added at either the process definition level as Init, Process Owner, Commit, onAbort, onResume, onSuspend, or Error Actions, or at Node level as Role, Prologue, Epilogue, onAbort, onResume, onSuspend, Error or Timer Actions. In addition, Compensate and Error Actions can be added at Java Action level.

Interstage BPM offers the following types of Java Actions:

- **Init Actions** and **Process Owner Actions** are executed upon process initialization. These Java Actions initialize User Defined Attribute data before the first activity is activated.
- **Commit Actions** are executed upon process completion. They can be used to clean up or analyze the data of an entire process instance.
- **Prologue Actions**. A Prologue Action is evaluated before the node performs its task. This Java Action can therefore be used to set up or initialize values associated with the node before it does its work.
- **Epilogue Actions**. An Epilogue Action is executed after the node finishes its task and before the process instance moves on to another node. This Java Action can therefore be used to clean up or analyze values associated with the node after the intended work is finished.
- **Role Actions**. A Role Action is evaluated after assignee resolution and before task assignment. This Java Action is therefore used to dynamically compute a list of assignees for a task in conjunction with a User Group.
- **Timer Actions** are executed when a timer expires.

- **Error Actions**. An Error Action can be used for handling specific error situations in the execution of a process. Error Actions can be defined for entire process definitions, for individual nodes and for other Java Actions, i.e. when you want to react on errors occurring during the execution of another Java Action.
- **Compensate Actions**. A Compensate Action can be defined for another Java Action that accesses a system outside of Interstage BPM, e.g. a database. Compensate Actions are useful to ensure a consistent state of all systems involved in a transaction for cleaning up and rolling back transactions, e.g. to delete a newly added row in an external database.
- A special type of action can be activated as soon as an Administrator issues the command to abort, suspend or resume the processing of a process instance. Such actions are stored in either of the following Action Sets:
  - **onAbort Action**
  - **onResume Action**
  - **onSuspend Action**

  on* Java Actions are to be performed before the state of a process instance is changed. They can be defined for individual activities, i.e. for individual nodes in a process definition, or for an entire process definition.

Note: When programming Java Actions, you specify its type only when you set it for a process definition, a node or another Java Action. The methods available from the `JavaAction` interface can be used for any type of Java Action, however, you need to be aware that the type of Java Action determines which methods are executed at runtime. Refer to the API Javadoc for more information.

6.3.2 Accessing Workflow Data Using the Server Enactment Context Interface

You can use Java Actions to manipulate process flow. In this case, Java Actions need access to the process instance context through the Server Enactment Context interface.

The Server Enactment Context interface provides access to workflow data and is implemented as a server interface (`com.fujitsu.iflow.server.intf.ServerEnactmentContext`). It provides, for example, the following methods:

- `addAttachment()`
- `getAttachment()`
- `getProcessOwners()`
- `setProcessDescription()`

Refer to the API Javadoc for details on the available classes and methods.

To access workflow data using the Server Enactment Context:

1. When creating your Java class:
   a) Import the ServerEnactmentContext interface.
   b) Make sure that one parameter of the method that you want to call from a Java Action is of type `ServerEnactmentContext`.
2. When defining the Java Action, you specify the values to pass to your method. Use the `sec` identifier as value for the `ServerEnactmentContext` parameter.

   At run time, Interstage BPM will pass an object with this interface to your method.

Example

Here is an example of a Java method you might write that needs to access workflow data. The bold text shows how to import and use the Server Enactment Context interface:

```
import com.fujitsu.iflow.server.intf.ServerEnactmentContext;
public class MyClass {
   public void reassignIfTooExpensive(ServerEnactmentContext sec,
      int amount) {
      String[] employee = {"Employee1", "Employee2"};
      String[] managers = {"Manager1", "Manager2"};
      if (amount <= 5000)
         sec.setActivityAssignees(employee);
      else
         sec.setActivityAssignees(managers);
   }
}
```

This is how you might define the Java Action. The bold text shows how to pass the context data to your method:

```
JavaAction[] myAction = MyActionSet.createJavaActions(1);
myAction[0].setActionDescription("Decide on purchase requisition");
myAction[0].setActionName("RoutePurchaseRequisition");
myAction[0].setClassName("MyPackage.MyClass");
myAction[0].setMethodName("reassignIfTooExpensive(ServerEnactmentContext,
 int)");
String[] args = new String[2];
args[0] = "sec";
args[1] = "uda.amount";
String params = Utils.combineParametersToXML(args);
myAction[0].setArgumentsUDANames(params);
MyActionSet.setJavaActions(myAction);
```

6.3.3 Assigning Prologue Actions to an Activity Node

This section gives you an example of how to assign Prologue Actions to an Activity Node. The Java Actions used are part of the example class `ComplexPlan`. Use `setClassName()` from the `JavaAction` interface to refer to the example class `ComplexPlan`.

In the example, Prologue Actions are defined that return the initial values for the User Defined Attributes (UDAs) `Qty` and `Price`. This means that you implement your own Java class and add its methods as Java Actions without using the Server Enactment Context API.

Note: When programming Java Actions, you specify its type only when you set it for a process definition, a node or another Java Action. The methods available from the `JavaAction` interface can be used for any type of Java Action, however, you need to be aware that the type of Java Action determines which methods are executed at runtime. Refer to the API Javadoc for more information.

To assign Prologue Actions to an Activity Node:

1. Add an Activity Node. The name of the first activity in the process definition is protected final static String NODE_FILL_OUT_PR = "Fill out Purchase Request";

```
protected final static String NODE_FILL_OUT_PR =
   "Fill out Purchase Request";
```

```
Node fillOutNode = plan.addNode("NODE_FILL_OUT_PR",
   Node.TYPE_ACTIVITY);
fillOutNode.setRole("SampleGroup");
fillOutNode.setPosition(new Point(450, 40));
```

2. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet foPJavaActionSet =
   WFObjectFactory.getJavaActionSet();
```

3. Generate the required number of Java Actions for `JavaActionSet`.

   In the following example, three Java Actions are generated.

```
JavaAction[] foPJavaActions =
   foPJavaActionSet.createJavaActions(3);
```

4. Define the Java Actions.

   These are the Java Actions that set the initial quantity and price:

```
foPJavaActions[0].setActionDescription("Sets initial value for Qty");
foPJavaActions[0].setActionName("load initial qty");
foPJavaActions[0].setMethodName("getInitialQty");
foPJavaActions[0].setClassName(CLASS_NAME_JAVA_ACTION);
foPJavaActions[0].setReturnValueUDAName("Qty");
...
foPJavaActions[1].setActionDescription("Sets initial value for Price");
foPJavaActions[1].setActionName("load initial price");
foPJavaActions[1].setMethodName("getInitialPrice");
foPJavaActions[1].setClassName(CLASS_NAME_JAVA_ACTION);
foPJavaActions[1].setReturnValueUDAName("Price");
...

foPJavaActionSet.setActionSetDescription("Operations to set"
   + " initial values of UDAs Qty and Price");
foPJavaActionSet.setActionSetName("Qty and Price Setter");
```

   In the example, another Prologue Action is defined for creating an attachment. To handle errors that might occur, Error Actions are defined for that Java Action. For details refer to section *Dealing with Errors in Java Actions* on page 102.

5. Assign `JavaActionSet` to the prologue part of the Activity Node.

```
foPJavaActionSet.setJavaActions(foPJavaActions);
fillOutNode.setJavaActionSet(foPJavaActionSet,JavaActionSet.NODE_PROLOGUE);
```

An internal copy of the Java Action Set is made and saved with the Activity Node. Further changes of the Java Action Set will have no effect on the Java Actions that are assigned to the Activity Node.

If you want to change the Activity Node, you must modify the Java Action Set, and then assign it again to the Activity Node.

6.3.4 Assigning Epilogue Actions to an Activity Node

Prerequisite: You have defined an Activity Node as explained in section *Assigning Prologue Actions to an Activity Node* on page 93.

In this example, `ComplexPlan` defines an Epilogue Action which returns the calculated price `Total` for the User Defined Attributes (UDAs) `Qty` and `Price`. The Epilogue Action is added to an Activity Node.

Note: When programming Java Actions, you specify its type only when you set it for a process definition, a node or another Java Action. The methods available from the `JavaAction` interface can be used for any type of Java Action, however, you need to be aware that the type of Java Action determines which methods are executed at runtime. Refer to the API Javadoc for more information.

To add an Epilogue Action to an Activity Node:

1. Use `getJavaActionset()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet foEJavaActionSet =
   WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`.

   In the following example, one Java Action is generated.

```
JavaAction[] foEJavaActions =
   foEJavaActionSet.createJavaActions(1);
```

3. Define the Java Action.

```
foEJavaActions[0].
   setActionDescription("Sets calculated total amount to UDA 'Total'");
foEJavaActions[0].setActionName("calculate Total");
foEJavaActions[0].setClassName(CLASS_NAME_JAVA_ACTION);
foEJavaActions[0].setClassPath(CLASS_PATH_JAVA_ACTION);
foEJavaActions[0].setMethodName("getTotal(float,int)");
foEJavaActions[0].
   setArgumentsUDANames("<E>uda.Price</E><E>uda.Qty</E>");
foEJavaActions[0].setReturnValueUDAName("Total");
foEJavaActionSet.
   setActionSetDescription("Operation to calculate total");
foEJavaActionSet.setActionSetName("Total Setter");
```

4. Assign `JavaActionSet` to the epilogue part of the Activity Node.

```
foEJavaActionSet.setJavaActions(foEJavaActions);
fillOutNode.setJavaActionSet(foEJavaActionSet,JavaActionSet.NODE_EPILOGUE);
```

6.3.5 Using Built-In Java Actions

Interstage BPM provides various Java methods for manipulating process flow. You can call these methods directly from a Java Action without having to implement them. These methods are referred to as built-in Java Actions.

There are, for example, the following built-in Java Actions:

- Java Action that can evaluate JavaScript. Refer to section *JavaScript Java Actions* on page 96 for an example.
- Java Actions that can invoke evaluations of business rules in compatible business rules engines. Refer to section *Rules JavaActions* on page 97 for more information.
- Java Actions to retrieve and update data in external databases.
- Java Actions to invoke and fetch information from web services.

Built-in Java Actions are available through the `IflowActions` class (`com.fujitsu.iflow.actions.IflowActions`). This class provides, for example, the following methods:

- `escalateActivity()`
- `makeChoiceAction()`
- `sendEmail()`

Refer to the API Javadoc for details on the available methods.

You can use the `IflowActions` methods in a Java Action by specifying the appropriate class name. The class path is not required.

Example

The following example shows how to specify a method of the `IflowActions` class:

```
JavaAction[] myAction = MyActionSet.createJavaActions(1);
myAction[0].setActionDescription("Decide on purchase requisition");
myAction[0].setActionName("MakeChoice");
myAction[0].setClassName("com.fujitsu.iflow.actions.IflowActions");
myAction[0].setMethodName("makeChoiceAction(String, long,
ServerEnactmentContext)");
String[] args = new String[3];
...
```

For more information on defining Java Actions, refer to section *Accessing Workflow Data Using the Server Enactment Context Interface* on page 92.

6.3.6 JavaScript Java Actions

Java Actions are flexible in that they allow you to do anything that Java can do, but they have the drawback that you must compile your Java into a class that is accessible to the process engine. For this reason, a special JavaScript Java Action that can evaluate JavaScript is available. In JavaScript, you can do everything that you could do in Java; only the JavaScript is not compiled until you need to execute it. Because the JavaScript is kept in source form inside the process definition, there is no associated class file that must be carried around externally.

JavaScript uses a syntax very much like Java, but there are a few differences, such as dynamically typed variables that make sense in a scripting environment. JavaScript is actually an ECMA standard described in the document ECMA-262.PDF that is provided as a printable file with the software. The same Server Enactment Context API is available to the JavaScripts.

In addition, you might want to create your own custom JavaScript extensions to effect script behavior that you will use over and over in your specific situation.

Note: The size that a method used in a JavaScript may have is limited by the Java Virtual Machine (JVM). Currently, the method byte code size is limited to 65535 bytes (64 KBytes). If you use a method with a larger size, the JVM will throw an error and you have to reduce the size of the method before executing the JavaScript again.

Calling Java Methods in JavaScript

Standard JavaScript has the ability to call any Java method you wish. It can even construct Java objects, and call methods on them, passing other Java objects as parameters if you wish.

Example of a call to a static method in ScriptPlugIn.class:

```
Packages.ScriptPlugIn.createFile(“c:\\temp\\test.txt”);
```

Example of a call to a member method in JDBCPlugIn.class:

```
x = new Packages.JDBCPlugIn();
var customer = uda.Customer;
var finish = x.findCustomer(customer);
```

6.3.7 Rules JavaActions

A Rules JavaAction is a special type of JavaAction that acts as an interface to a Rules Engine. This interface allows you to use advanced process logic in your processes at every point in the process where you could attach a JavaAction Set.

You can apply these rules using the following Interstage BPM methods along with any of your own custom methods:

- `getCurrentProcessId`
- `getCurrentActivityId`
- `getCurrentActivityName`
- `getCurrentActionType`
- `getProcessDefinitionName`
- `getProcessDefinitionIdentifier`
- `getOwners`
- `getInitiator`
- `getMembers`
- `getActivityAssignees`
- `getActivityIteratorIndex`
- `getProcessAttribute`*
- `getProcessPriority`
- `getActivityPriority`
- `getProcessTitle`
- `getProcessOwners`
- `getProcessInitiator`
- `getProcessDescription`

- `getProcessName`
- `setProcessName`
- `getGroupMembers`
- `getActivityActor`
- `setProcessOwners`
- `setActivityAssignees`
- `setProcessAttribute`*
- `setProcessPriority`
- `setActivityPriority`
- `setOwners`
- `setProcessTitle`
- `setProcessDescription`

*For the `getProcessAttribute` and `setProcessAttribute` methods, the user-defined process attributes that you are getting or setting must be defined on the process definition on which the Rules JavaAction is attached.

ILOG Configuration

Note: To use the ILOG Rules Engine, you must have the ILOG application installed on your system and copy its `rulesall.jar` file to the location as specified in *Integrating Interstage BPM with External Applications* on page 88.

This section discusses the development of a business rules (`*.ilr`) file in the ILOG Rules Engine. It provides instructions for integrating Interstage BPM methods in ILOG but not for using ILOG. For instructions in using ILOG, please refer to your ILOG documentation.

With a business rules file you can create a Rules JavaAction to implement the advanced process logic of ILOG.

You can write your business rules file in ILOG by importing the Rules Engine Interface class into ILOG. This class is called `RulesEngineIntf.class`, and it can be found in the `com.fujitsu.iflow.rules` package. Once you import this class, it will appear in your ILOG editor, and you will see all the Interstage BPM methods (mentioned above) under it.

If you want to use ILOG Rules in Java Actions, the following configuration step is required:

On the Interstage BPM Server

- Specify a rule file name by using the first parameter of InputUDAList.
- If in SaaS mode, or if only the file name was specified, copy the file to `<DMSRoot>/apps/<application ID>/dms/Attachments`.
- If in non-SaaS mode the complete file path was specified, copy the file to that location.

Blaze Advisor Configuration

If you want to use Blaze Advisor rules in Java Actions, the following configuration steps are required.

1. If it is the first time that you use Blaze Advisor with Interstage BPM, copy the following JAR files with the license directory from the Blaze Advisor `lib` directory into the `<Interstage BPM Domain>/lib` directory, e.g. `C:/bea/user_projects/domains/base_domain/lib`:
   - `AdvCommon.jar`

- `Advisor.jar`
- `AdvisorSvr.jar`
- `collections.jar`
- `InnovatorRT.jar`
- `jaxen.jar`
- `Ndkjc-2.1A-bin.jar`
- `OROMatcher.zip`
- `saxpath.jar`
- `iFlow.jar`. This file is located in the client/lib folder of your Interstage BPM installation, e.g. `C:/Fujitsu/InterstageBPM/client/lib`.

2. On the Interstage BPM Server
   - Specify a server configuration file name by using the first parameter of InputUDAList.
   - If in SaaS mode, or if only the file name was specified, copy the file to `<DMSRoot>/apps/<application ID>/dms/Attachments`.
   - If in non-SaaS mode the complete file path was specified, copy the file to that location.
3. Include the Blaze license file in your CLASSPATH.

   If you are using Interstage BPM for WebLogic, update the script `setDomainEnv.cmd` or `setDomainEnv.sh` located in the `<Interstage BPM Domain>/bin` directory, e.g. `C:/bea/user_projects/domains/base_domain/bin`:

   On **Windows**:

   Delete the line:

```
set
CLASSPATH=%PRE_CLASSPATH%;%WEBLOGIC_CLASSPATH%;
%POST_CLASSPATH%;%WLP_POST_CLASSPATH%
```

   Add the following lines:

```
set BLAZE_LICENSEPATH=<path to your license file>
set
CLASSPATH=%PRE_CLASSPATH%;%WEBLOGIC_CLASSPATH%;
%POST_CLASSPATH%;%WLP_POST_CLASSPATH%;%BLAZE_LICENSEPATH%
```

   On **UNIX** or **Linux**:

   Delete the line:

```
CLASSPATH="{PRE_CLASSPATH}${CLASSPATHSEP}
${WEBLOGIC_CLASSPATH}${CLASSPATHSEP}
${POST_CLASSPATH}${CLASSPATHSEP}
${WLP_POST_CLASSPATH}"
```

   Add the following lines:

```
BLAZE_LICENSEPATH=<path to your license file>
export $BLAZE_LICENSEPATH
CLASSPATH="{PRE_CLASSPATH}${CLASSPATHSEP}
${WEBLOGIC_CLASSPATH}${CLASSPATHSEP}
${POST_CLASSPATH}${CLASSPATHSEP}
${WLP_POST_CLASSPATH}${CLASSPATHSEP}
${BLAZE_LICENSEPATH}"
```

4. Stop and start the Interstage BPM Server.

   For instructions, refer to the *Interstage Business Process Manager Server Administration Guide*.

Decision Tables JavaAction

Decision Tables is a feature built into Interstage BPM that allows designing of advanced rules for decision making without programming. For details of how to create rules using Decision Tables, refer section *Decision Tables* on page 171.

To use a decision table in a process definition, a predefined JavaAction to evaluate Decision Tables (called a Decision Table JavaAction, a type of Rules JavaAction) is provided. For further details refer section *Using a Decision Table within a Process Definition* on page 173, the *Interstage BPM Studio User's Guide* and/or the *Interstage BPM Console Online Help*.

6.3.8 Activity Actors and Relationships

Activity Actors

After an activity is assigned to a role, any person in that role can perform that activity. For instance, if an activity is assigned to the 'managers' role, any person who is a member of the 'managers' group can perform that activity. But only one particular manager will actually perform the activity. That manager can be referred to as the activity actor for that activity.

After an activity has been performed, certain workflows may demand determining who was the activity actor, and then take further action based on that information. For instance, consider the activity 'Record customer issue' assigned to the 'customer support executive' role. It will be needed to ascertain exactly which customer executive performed a particular 'Record customer issue' activity, so that the same actor can be assigned the next activity 'Escalate issue' in that process instance.

Determining Activity Actors

The `ServerEnactmentContext` interface of the `com.fujitsu.iflow.server.intf` package contains the `getActivityActor()` method to determine activity actors.

The `getActivityActor()` method gets the actor of a completed Activity node using the name of the Activity node as a parameter. Voting activity node is not supported by this method.

Note: It is recommended that all activity nodes within a process instance have unique names. This method throws an exception if a node name passed to it exists in duplicate.

The following sample gets the actor of 'Activity A' and assigns the current activity to that actor.

```
public void assignActivityActor(ServerEnactmentContext sec){
 String[] actor = new String[1];
 actor[0] = sec.getActivityActor("Activity A");
 sec.setActivityAssignees(actor);
}
```

Note: You need to either call this method in a JavaAction defined by you (refer *Accessing Workflow Data Using the Server Enactment Context Interface* on page 92) or as part of a JavaScript.

Refer to the API Javadoc for details of this method.

Relationships

Certain workflows may demand finding out the hierarchical relationships for an activity actor or a UDA value. For example, the UDA 'Company Name' may have a value 'Fujitsu', and it may be required to ascertain who is the 'account executive' for the value 'Fujitsu', so only that account executive (and not any other) may be assigned a particular activity. Or, an activity may have an actor called 'Jim', and it may be required to find out who is the 'manager' for Jim, so that, that manager can be assigned a particular activity.

Determining Relationships

The `ServerEnactmentContext` interface of the `com.fujitsu.iflow.server.intf` package contains the `resolveRelationship()` method to determine relationships.

The `resolveRelationship()` method returns a target value using a source value and a relationship as parameters. For this method to work, mappings of source values, relationships, and target values need to be pre-defined, so that this method can refer to the mappings and return appropriate values.

SourceValue	Relationship	TargetValue
`Jim`	`manager`	`Robert`
`Jim`	`assistant`	`Arthur`
`Fujitsu`	`executive`	`Bob`

For example, `resolveRelationship("assistant", "Jim")` returns the value `Arthur`.

SourceValue-Relationship-TargetValue mappings should be stored as user profiles in the Directory Service or local user store, where the source value is a user ID, relationship is a user attribute name, and target value is a user attribute value. To create a user profile, refer the `DirectoryServices` interface of the `com.fujitsu.iflow.model.workflow` package in the API Javadoc.

Note that SourceValues can be both human as well as non-human (such as company name or group name). User profiles of such non-human objects also need to be added to the directory server or local user store, taking care that such user profiles cannot be used for the purpose of logging in.

The following sample gets the actor of 'Activity A', gets the manager of that actor, and assigns the current activity to this manager.

```
public void assignManagerOfActivityActor(ServerEnactmentContext sec){
 String[] managers =
sec.resolveRelationship("Manager",sec.getActivityActor("Activity A"));
 sec.setActivityAssignees(managers);
}
```

Note: You need to either call this method in a JavaAction defined by you (refer *Accessing Workflow Data Using the Server Enactment Context Interface* on page 92) or as part of a JavaScript.

Refer to the API Javadoc for details of this method.

6.3.9 Using Error Java Actions

Error Java Actions can be used to handle specific errors and to determine the behavior of a process instance in case an error occurs. If you do not define any error handling, a process instance will go into error state as soon as an exception is thrown, irrespective of where the exception is thrown: e.g. if the starting of a Remote Subprocess fails, if an email cannot be sent, etc.

Error Java Actions can be defined on different levels:

- On **Process Definition level**: These Error Java Action Sets will be executed in case of any error, independent of the activity in which an error occurs and independent of the severity of an error. Error Java Action sets defined on process definition level will be executed immediately before the process instance will go into error state. Note that such Error Actions cannot influence the behavior of the process instance. Error Actions on process definition level may, for example, be used for sending a notification email or for writing additional information into a log file.
- On **Node level** (for Remote Subprocess Nodes only): An Error Java Action Set on this level will become active if the remote subprocess fails to start. Refer to section *Error Handling for Remote Subprocesses* on page 166 for an example.
- On **Java Action level**: An Error Java Action Set on this level will be executed when an exception is thrown in the related "regular" Java Action. Error Java Actions can be defined for all types of Java Actions (e.g. Prologue, Epilogue, Timer, onAbort Java Action) except for an Error or Compensate Action itself. Assigning Error Actions to Java Actions is described in section *Dealing with Errors in Java Actions* on page 102.

Below you find an example of how to assign an Error Java Action to a process definition. The Java Actions used are part of the example class `ComplexPlan`.

To define an Error Java Action Set on process definition level:

1. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet plErrorJavaActionSet =
   WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`.

   In the following example, one Java Action is generated.

```
JavaAction[] plErrorJavaAction =
   plErrorJavaActionSet.createJavaActions(1);
```

3. Define the Java Action Set.

```
plErrorJavaAction[0]
    .setActionName("WriteLogEntryWithException");
plErrorJavaAction[0]
    .setActionDescription("Writes a log entry with exception");
plErrorJavaAction[0]
    .setMethodName("writeLogEntryWithException(String,ServerEnactmentContext)");
plErrorJavaAction[0]
    .setClassName(CLASS_NAME_JAVA_ACTION);
plErrorJavaAction[0]
    .setArgumentsUDANames("<E>uda.ProcessLevelErrorLogEntry</E><E>sec</E>");
plErrorJavaActionSet.setJavaActions(plErrorJavaAction);
```

4. Assign the `JavaActionSet` to the process definition.

```
plan.setJavaActionSet(plErrorJavaActionSet,
    JavaActionSet.PLAN_ERROR);
```

6.3.10 Dealing with Errors in Java Actions

When an error occurs during the execution of a Java Action, an exception is thrown. Interstage BPM allows you to define your own error handling for erroneous Java Actions and Remote Subprocess calls. In this way, you can prevent that a process instance goes into error state when an exception is thrown.

In addition, you can define actions for Java Actions which perform a "cleanup" before a transaction is rolled back and a process instance is set to the error state. This includes a rollback of all Java Actions in a Java Action Set, and allows you to perform general actions (e.g. sending notification emails in any error case), or executing some specific actions before setting the process instance to error state.

Interstage BPM provides the following options to handle errors in Java Actions:

- **Compensate Actions**: Compensate Actions are used to clean up the system and to ensure a consistent state of all systems involved in a transaction, e.g. external databases or mail servers. Compensate Actions are particularly useful whenever external systems are involved.

  If you do not define any error handling for a Java Action, the following happens: When an exception is thrown in this Java Action, the transaction will be rolled back. A rollback, however, is only possible for changes in the Interstage BPM Application Server context. Any transactions in external systems cannot be rolled back, for example, if a row has been added to an external database. Therefore, it is sometimes necessary to manually clean up external systems to ensure a consistent state of all systems used in the transaction. Optionally, you can use Compensate Action Sets.

  You can specify a Compensate Action for every Java Action in an Action Set. You can use a Compensate Actions e.g. for removing a newly added row in a database or for sending out an additional email. If an exception occurs in a regular Java Action Set, all Compensate Actions defined for all Java Actions that have been successfully executed before the exception was thrown, are invoked in reverse order.

Note: You cannot embed a Compensate Action in another Compensate Action. If a Compensate Action throws an exception, the process instance will immediately go to error state, and the execution of remaining Compensate Actions will be aborted.

  Refer to the sample below for more information.

- **Error Actions**: On Java Action level, Error Action sets will become active when an exception is thrown in the related regular Java Action. Error Actions are bound together in Action Sets, similar to all other Action Sets. Error Actions can be specified for any action inside an Action Set. Note that you cannot define an Error Action that handles exceptions occurring in an Error Action.

  Refer to section *Using Error Java Actions* on page 101 and the sample below for more information.

  The Continue Setting

  Error Actions allow for determining whether the error will be caught and process execution will continue or not. This behavior is defined by the Continue Setting "**catchException**":

  If set to `true`, the exception gets caught and the process instance continues its execution after the defined Error Actions have been executed.

  If set to `false`, any Compensate Action defined will be executed and afterwards the process instance will go into error state.

  In case several Error Actions are defined that have different "catchException" settings, `false` overrides `true` and the process instance will go into error state as soon as one Error Action has set the "catchException" to `false`.

  The Exception Class

  In addition to the Continue Setting, Error Actions on Activity Node level and on Java Action level allow for determining on which exceptions the Error Action reacts: You can specify a concrete exception class (default: `java.lang.Exception`. In this case, an Error Action will only be executed when the exception thrown is an instance or a subclass of this specified exception class. For example:

You specify the `java.io.FileNotFoundException`) class so that the Error Action will only be considered in case the occurred exception is an instance or a subclass of this class.

Note: In case an Error Java Action throws an exception, the transaction will be rolled back instantly and the process instance will go into error state. No Error Action can be defined for such a case.

Any Java Action Set can contain several Java Actions. For each Java Action, you can define Error Actions, as well as Compensate Actions. Refer to section *Java Action Structure and Execution Plan in Case of Errors* on page 106 for additional information.

Defining and Assigning Compensate Java Actions to a Java Action

The sample below defines a Compensate Java Action that is to be executed in case the execution of another Java Action fails: The administrator is to be notified by email in case the retrieval of the initial quantity fails. This retrieval is defined in another Java Action that is executed as Prologue Action for the FillOut Activity Node. You can find the whole code of this sample in the `ComplexPlan.java` sample. The methods for sending an email and for retrieving the initial quantity are defined in the `SampleJavaActions` sample.

To define and assign a Compensate Java Action Set:

1. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet compensateJavaActionSet =
    WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`. In the following example, one Java Action is generated.

```
JavaAction[] compensateJavaAction =
    compensateJavaActionSet.createJavaActions(1);
```

3. Define the Java Action. Note that the `sendEmail()` method is defined in the `SampleJavaActions` sample.

```
compensateJavaAction[0].setActionName("Send a notification mail");
compensateJavaAction[0]
    .setActionDescription("Send notification mail to administrator");
compensateJavaAction[0].setMethodName
    ("sendEmail(String, String, String, String, String, String,
      ServerEnactmentContext)");
compensateJavaAction[0].setClassName(CLASS_NAME_JAVA_ACTION);
compensateJavaAction[0].setArgumentsUDANames
     ("<E>uda.MailTo</E><E>uda.MailFrom</E>
       <E>uda.MailCc</E><E>uda.MailBcc</E><E>uda.MailSubject</E>
       <E>uda.MailBody</E><E>sec</E>");
```

4. Add the `compensateJavaAction` to the Java Action that retrieves the initial quantity. This Java Action is part of the Prologue Java Action Set of the FillOut Activity Node. You can find the entire definition in the `ComplexPlan.java` sample.

```
....
JavaActionSet foPJavaActionSet = WFObjectFactory.getJavaActionSet();
JavaAction[] foPJavaActions = foPJavaActionSet.createJavaActions(3);
foPJavaActions[0]
        .setActionDescription("Sets initial value for Qty");
foPJavaActions[0].setActionName("load initial qty");
foPJavaActions[0].setMethodName("getInitialQty");
foPJavaActions[0].setClassName(CLASS_NAME_JAVA_ACTION);
foPJavaActions[0].setReturnValueUDAName("Qty");
foPJavaActions[0].setJavaActionSet(compensateJavaActionSet,
        JavaActionSet.ACTION_COMPENSATE);
....
```

Defining and Assigning Error Actions to a Java Action

The sample below defines an Error Action Set that is to be executed in case the execution of a related Java Action fails: An entry is to be written to the log file and the processing of the process instance is to be continued in case an attachment cannot be added. The method for adding an attachment is defined in the in the `SampleJavaActions` sample. This Java Action is executed as Prologue Action for the FillOut Activity Node. You can find the whole code of this sample in the `ComplexPlan.java` sample.

To define and assign an Error Java Action Set:

1. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet errorJavaActionSet =
    WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`. In the following example, two Java Actions are generated.

```
JavaAction[] errorJavaActions =
    errorJavaActionSet.createJavaActions(2);
```

3. Define the Java Actions. Note that the `addAttachment()` method is defined in the `SampleJavaActions` sample.

```
errorJavaActions[0]
    .setActionDescription("Writes a log entry with exception");
errorJavaActions[0]
    .setMethodName("writeLogEntryWithException(String,ServerEnactmentContext)");
errorJavaActions[0].setClassName(CLASS_NAME_JAVA_ACTION);
errorJavaActions[0]
    .setArgumentsUDANames("<E>uda.ProcessLevelErrorLogEntry</E><E>sec</E>");
errorJavaActions[0].setCatchException(true);

errorJavaActions[1].setActionName("Continue process");
errorJavaActions[1].setActionDescription("Catches FileNotFoundExceptions
    and continues the execution of the process");
errorJavaActions[1].setMethodName("noop()");
errorJavaActions[1]
    .setClassName("com.fujitsu.iflow.actions.IflowActions");
errorJavaActions[1]
     .setEditorClassName("com.fujitsu.iflow.actions.IflowActions");
```

```
errorJavaActions[1]
      .setCatchException(true);
```

4. Add the above Error Java Action Set to the Java Action that is to add the attachment. This Java Action is part of the Prologue Java Action Set of the FillOut Activity Node. You can find the entire definition in the `ComplexPlan.java` sample.

```
....
foPJavaActions[2].setJavaActionSet(errorJavaActionSet,
    JavaActionSet.ACTION_ERROR);
...
```

6.3.11 Java Action Structure and Execution Plan in Case of Errors

This section describes the possible structure of Java Action Sets including Compensate and Error Action Sets. In addition, for any error that occurs in a Java Action, it shows which actions will be executed in which order.

Any "regular" Java Action Set can consist of several Java Actions. In addition, you can define an Error Java Action set for handling execptions thrown by a "regular" Java Action in the Java Action Set (e.g. a Prologue Java Action Set), and you can define a Compensate Action Set for the "regular"

Java Action Set that will become active in case the execution of the "regular" Java Action Set throws an exception and a rollback of an external system is required:

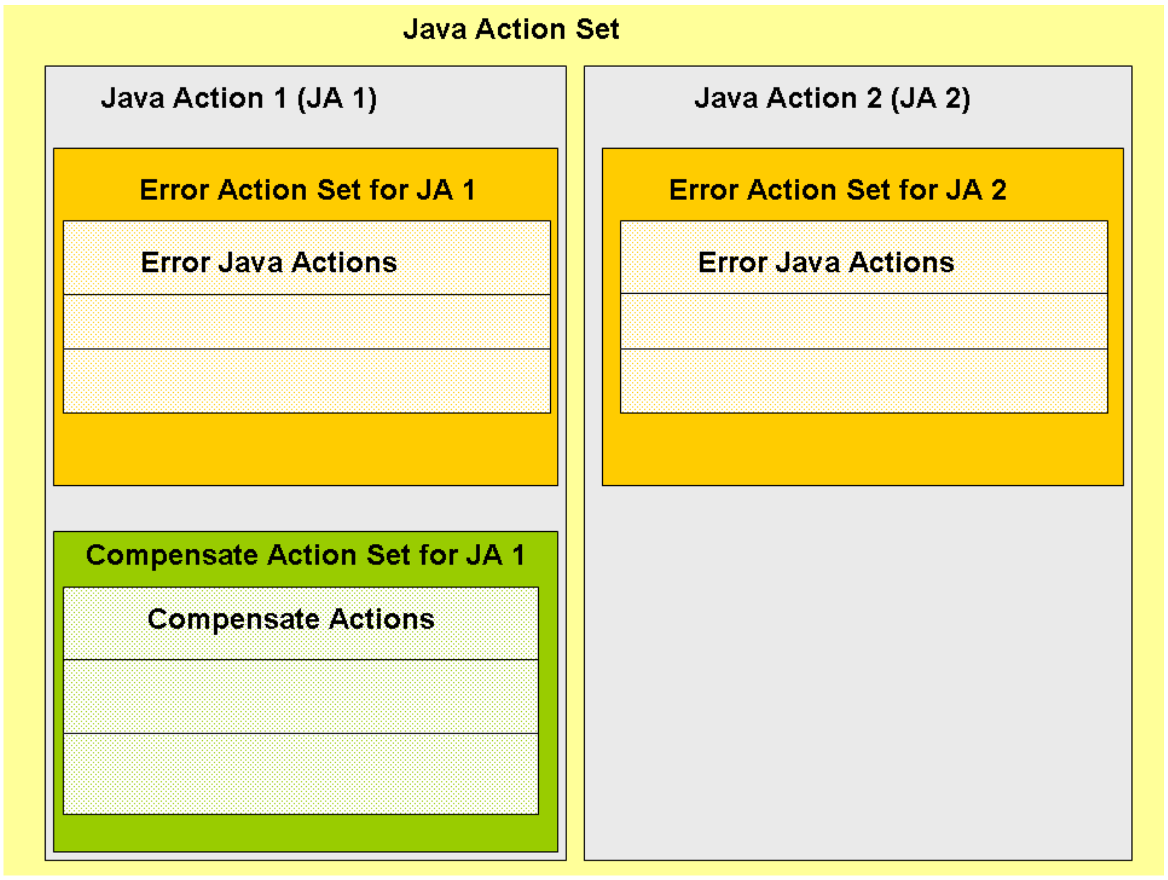


Figure 17: Java Action Set Structure

Example of a Regular Java Action Set

Assume you have defined a Prologue Java Action Set comprising two Java Actions (JA_1 and JA_2).

- **JA_1**: An Error Java Action Set with one Error Java Action (EJA_JA_1) and a Compensate Java Action Set with two Compensate Java Actions (CJA_1_JA_1 and CJA_2_JA_2) have been defined.

- **JA_2**: An Error Java Action Set with two Error Java Actions (EJA_1_JA_2 and EJA_2_JA_2) has been defined, but no Compensate Java Action Set is available:

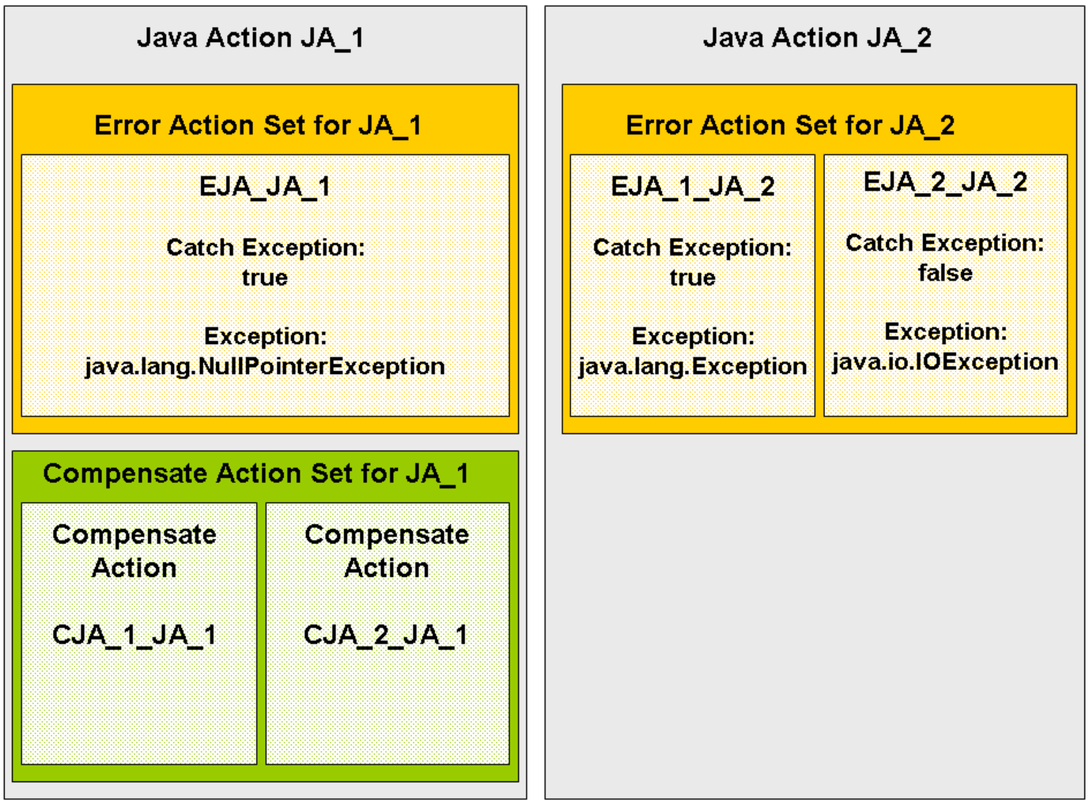

Java Action JA_1
Error Action Set for JA_1
EJA_JA_1
Catch Exception: true
Exception: java.lang.NullPointerException
Compensate Action Set for JA_1
Compensate Action
CJA_1_JA_1
Compensate Action
CJA_2_JA_1
Java Action JA_2
Error Action Set for JA_2
EJA_1_JA_2
Catch Exception: true
Exception: java.lang.Exception
EJA_2_JA_2
Catch Exception: false
Exception: java.io.IOException

Figure 18: Regular Java Action Set

Execution Plan in Case of Errors

The table below lists several error situations and explains which Java Actions will be executed in case of an error:

Situation	Result
JA_1 throws a `FileNotFoundException`.	Since no Error Action is defined for this situation, the process instance will go into error state.
JA_1 throws a `NullPointerException`.	The exception gets caught and EJA_JA_1 will be executed. The process instance continues, i.e. JA_2 will be executed and afterwards the transaction will be committed.
JA_1 executes successfully. JA_2 throws a `NullPointerException`.	EJA_1_JA_2 will be executed, because `NullPointerException` is a subclass of `Exception`. The exception will be caught, the transaction will be committed and the process instance continues.

Situation	**Result**
JA_1 executes successfully. JA_2 throws a `FileNotFoundException`.	EJA_1_JA_2 and EJA_2_JA_2 will be executed, because `FileNotFoundException` is a subclass of `Exception` and of `IOException`. However, the exception will not be caught due to the `Catch Exception` being set to `false` in EJA_2_JA_2. This setting overrides the EJA_1_JA_2 setting. Therefore, after execution of the two Error Actions, the Compensate Actions CJA_1_JA_1 and CJA_2_JA_1 will be executed before the process instance goes into error state.
JA_1 and JA_2 are executed successfully.	No Error or Compensate Actions are executed. The process instance remains in its regular state.

Refer to sections *Dealing with Errors in Java Actions* on page 102 and *Using Error Java Actions* on page 101 for additional information.

6.3.12 Using onAbort, onSuspend, onResume Java Actions

The onSuspend, onResume, and onAbort Java Actions can be used to execute specific actions before the state of a process instance is changed because an Interstage BPM Administrator has called the suspend, resume, or abort command.

Similar to other Java Action Sets, the onAbort, onResume, and onSuspend Action Sets can contain several Java Actions. Each of these sets can be defined on node level as well as on process definition level. Node level Action Sets will always be executed first, but only if the respective activity is active when the abort, suspend, resume command has been issued. The process definition level Action Sets will be executed independent of the activities that are active when the command is being issued.

Below you find an example of how to assign an onAbort Java Action to a process definition. The Java Actions used are part of the example class `ComplexPlan`. Use `setClassName()` from the `JavaAction` interface to refer to the example class `ComplexPlan`.

To design and assign an onAbort Java Action Set on process definition level:

1. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
onAbortJavaActionSet = WFObjectFactory.getJavaActionSet()
```

2. Generate the required number of Java Actions for `JavaActionSet`.

   In the following example, one Java Action is generated.

```
JavaAction[] onAbortJavaAction =
     onAbortJavaActionSet.createJavaActions(1);
```

3. Define the Java Action Set.

```
onAbortJavaAction[0].setActionName("WriteLogEntry");
onAbortJavaAction[0]
    .setActionDescription("Writes a log entry if process is aborted");
onAbortJavaAction[0].setMethodName("writeLogEntry(String)");
onAbortJavaAction[0].setClassName(CLASS_NAME_JAVA_ACTION);
```

```
onAbortJavaAction[0]
    .setArgumentsUDANames("<E>uda.OnAbortLogEntry</E>");
onAbortJavaActionSet.setJavaActions(onAbortJavaAction);
```

4. Assign the `JavaActionSet` to the process definition.

```
plan.setJavaActionSet(onAbortJavaActionSet,
                    JavaActionSet.PLAN_ABORT);
```

6.4 Advanced Filtering & Sorting API

You can filter lists of process definitions, process instances and work items on a range of fields like the process definition name, process initiator, work item name, or any other. With this feature, you can retrieve lists that match certain filtering criteria like: get all process instances initiated by "userX", or get all work items originating from process instances with title "Purchase Order". You can also filter on multiple fields. A table of the fields and the lists with which they are valid is provided in the next section.

6.4.1 Interface WFObjectList (Package com.Fujitsu.iflow.model.workflow)

This interface offers constants that identify the fields that can be used for filtering/sorting lists along with the filters for which they are valid, for example:

Filter	**listField**

`Filter.AllPlans`
`Filter.MyPlans`
	`LISTFIELD_APPLICATION_IDENTIFIER`
	`LISTFIELD_PLAN_ID`
	`LISTFIELD_PLAN_NAME`
	`LISTFIELD_PLAN_OWNER`
	`LISTFIELD_PLAN_STATE`
	`LISTFIELD_PLAN_IDENTIFIER`

Filter	listField

`Filter.AllProcesses`
`Filter.AllActiveProcesses`
`Filter.MyProcesses`
`Filter.MyActiveProcesses`
`Filter.MyInactiveProcesses`
`Filter.AllInactiveProcesses`
`Filter.AllProcessesInErrorState`
	`LISTFIELD_APPLICATION_IDENTIFIER`
	`LISTFIELD_PLAN_ID`
	`LISTFIELD_PLAN_NAME`
	`LISTFIELD_PLAN_IDENTIFIER`
	`LISTFIELD_PROCESSINSTANCE_ID`
	`LISTFIELD_PROCESSINSTANCE_STATE`
	`LISTFIELD_PROCESSINSTANCE_INITIATOR`
	`LISTFIELD_PROCESSINSTANCE_PRIORITY`
	`LISTFIELD_PROCESSINSTANCE_NAME`
	`LISTFIELD_PROCESSINSTANCE_TITLE`
	`LISTFIELD_PROCESSINSTANCE_PARENTID`
	`LISTFIELD_PROCESSINSTANCE_CREATEDTIME`
	`LISTFIELD_PROCESSINSTANCE_CLOSEDTIME`
	`LISTFIELD_PROCESSINSTANCE_OWNER`
	`LISTFIELD_PROCESSINSTANCE_DUEDATE`
	`LISTFIELD_PROCESSINSTANCE_TYPE`

`Filter.AllWorkItems`
`Filter.AllInactiveWorkItems`
`Filter.MyWorkItems`
`Filter.MyAcceptedWorkItems`
`Filter.MyActiveWorkItems`
`Filter.MyDeclinedWorkItems`
	`LISTFIELD_APPLICATION_IDENTIFIER`
	`LISTFIELD_PLAN_ID`
	`LISTFIELD_PLAN_NAME`
	`LISTFIELD_PLAN_IDENTIFIER`
	`LISTFIELD_PROCESSINSTANCE_ID`
	`LISTFIELD_PROCESSINSTANCE_STATE`
	`LISTFIELD_PROCESSINSTANCE_NAME`
	`LISTFIELD_PROCESSINSTANCE_TITLE`
	`LISTFIELD_PROCESSINSTANCE_INITIATOR`
	`LISTFIELD_PROCESSINSTANCE_PRIORITY`
	`LISTFIELD_PROCESSINSTANCE_CREATEDTIME`
	`LISTFIELD_WORKITEM_ID`
	`LISTFIELD_WORKITEM_ASSIGNEE`
	`LISTFIELD_WORKITEM_STATE`
	`LISTFIELD_WORKITEM_CREATEDTIME`
	`LISTFIELD_WORKITEM_DUEDATE`
	`LISTFIELD_WORKITEM_PRIORITY`
	`LISTFIELD_WORKITEM_NAME`
	`LISTFIELD_WORKITEM_ACTIVITYINSTANCEID`

Filter	**listField**
`Filter.AllArchivedPlans`	`LISTFIELD_PLAN_ID` `LISTFIELD_PLAN_NAME`

`Filter.AllArchivedProcesses`	`LISTFIELD_PLAN_ID`
	`LISTFIELD_PROCESSINSTANCE_ID`
	`LISTFIELD_PROCESSINSTANCE_NAME`

`Filter.MyFutureWorkItems`
`Filter.AllFutureWorkItems`
	`LISTFIELD_APPLICATION_IDENTIFIER`
	`LISTFIELD_PLAN_ID`
	`LISTFIELD_PLAN_NAME`
	`LISTFIELD_PLAN_IDENTIFIER`
	`LISTFIELD_PROCESSINSTANCE_ID`
	`LISTFIELD_PROCESSINSTANCE_STATE`
	`LISTFIELD_PROCESSINSTANCE_NAME`
	`LISTFIELD_PROCESSINSTANCE_TITLE`
	`LISTFIELD_PROCESSINSTANCE_INITIATOR`
	`LISTFIELD_PROCESSINSTANCE_PRIORITY`
	`LISTFIELD_PROCESSINSTANCE_CREATEDTIME`
	`LISTFIELD_WORKITEM_ID`
	`LISTFIELD_WORKITEM_ASSIGNEE`
	`LISTFIELD_WORKITEM_PRIORITY`
	`LISTFIELD_WORKITEM_NAME`
	`LISTFIELD_WORKITEM_ACTIVITYINSTANCEID`

The following constants denote sorting order:

- `SORTORDER_ASCENDING`
- `SORTORDER_DESCENDING`

The following constants denote SQL relational operands:

- `SQLOP_EQUALTO`
- `SQLOP_GREATERTHAN`
- `SQLOP_LESSTHAN`
- `SQLOP_GREATERTHANOREQUALTO`
- `SQLOP_LESSTHANOREQUALTO`
- `SQLOP_NOTEQUALTO`
- `SQLOP_LIKE`
- `SQLOP_NOTLIKE`
- `SQLOP_IN`

You can find examples using filters in the following sections: *Listing Work Items* on page 81, *Listing Process Definitions* on page 192, and *Listing Process Instances* on page 196.

6.4.2 Methods for Filtering and Sorting

There are the following methods for filtering and sorting:

- `addFilter()`: Adds a field-based filtering criteria for the list of objects to be retrieved. The values passed in the parameters conceptually translate to "where <actual-column-name-of-the-listField> <actual-value-of-the-SQL-operator> <value>".
- `addFilter()`: Adds a UDA-based filtering criteria for the list of objects to be retrieved. The values passed in the parameters conceptually translate to "where <udaName> <actual-value-of-the-SQL-operator> <value>".
- `addSortOrder()`: Adds a field-based sort order for the list of objects to be retrieved.
- `addSortOrder()`: Adds a UDA-based sort order for the list of objects to be retrieved.
- `getNextBatch[]`: Returns an array of list objects that are next in sequence. The returned array size is equal to `batchSize`. It may be less than `batchSize` for the last batch. The call returns `null` if there are no more elements in the list to return.

For details, refer to the API Javadoc.

6.4.3 Interface Semantics

The following sample represents the proper order in which the methods of this interface are to be invoked:

```
wfol = WFObjectFactory.getWfObjectList()
wfol.addFilter(…)
wfol.addSortOrder(…)
Object[] listElements = null;
int batchSize = 10;
wfol.openBatchedList(…)
while ((listElements = wfol.getNextBatch(batchSize)) != null) {
   // process elements in listElements array
}
```

Make sure that the `openBatchedList()` call follows `addFilter()` and/or `addSortOrder()` call(s) and precedes `getNextBatch()` calls. If the methods are invoked in any other order, the behavior of the `WFObjectList` is undefined.

6.4.4 Identifying UDAs to be Used With Lists

You can filter lists of process definitions, process instances and work items on values of User Defined Attributes (UDAs). Users can retrieve lists which match filtering criteria based on values of UDAs, for example "get all work items where transaction_amount > 5000.*". To do this, make sure, that the `WorkListUDA` flag is set on the UDA. This can be accomplished when you design the process definition by using `markAsWorkListUDA()` from the `DataItemRef` interface. Refer to section *Worklist UDAs* on page 120 for more information.

If UDAs that are not identified as described above are used to filter and sort lists, the behavior of the list will be undefined.

6.4.5 Sorting and Filtering Combinations of Fields and UDAs

A list can be sorted and filtered on combinations of fields, but with the following restriction:

Field-based sort order can’t be used together with UDA-based sort order.

The following table identifies the valid combinations of field/UDA filters and sort orders.

	1 UDA Filter	1 UDA Sort Order	1 or more Field Filter	1 or more Field Sort Order
1 UDA Filter	-			
1 UDA Sort Order	Same*	-		
1 or more Field Filter	Valid	Valid	-	
1 or more Field Sort Order	Valid	Not-Valid	Valid	-

* Both the sort and filtering has to be done on the same UDA.

6.4.6 Note on Batching

You can retrieve lists in batches. If there were a large number of records in the database, opening a list would take a long time even if you needed only a few items of the list. You can use `openBatchedList` and `getNextBatch(int howMany)` to retrieve lists in batches. This batching works in conjunction with the filtering and sorting described here; i.e., Interstage BPM will retrieve a batch of the filtered and sorted list. However, the instructions provided in *Interface Semantics* on page 113 must be followed to obtain the desired result.

Note: If filtering or sorting is used, the relationship between the speed of retrieving a batch and the speed of retrieving the full list depends on the distribution of the values of the field/UDA used for filtering/sorting. The better the distribution of the values, the more advantage is derived from using batching.

For example if a list of process instances is sorted by process name, and if all the process instances have the same name, retrieving a batch internally degenerates to retrieving the full list. On the other hand if there are process instances with most process instances having different names, then the full advantage of using batching is derived.

6.4.7 Notification

Notification will be supported only in simple-list cases, i.e. without batch, filters or sort orders. If the list is opened with notification the following methods will throw a ModelException when invoked.

- `addFilter()`
- `addSortOrder()`
- `getNextBatch()`

6.5 Retrieving Information about Multiple Objects at a Time

The `WFDetailsList` interface allows you to efficiently retrieve information about many objects at a time. This capability is also referred to as bulk processing.

Currently, the interface provides methods to do the following:

- Retrieve User Defined Attributes (UDAs) of multiple process instances at a time
- Retrieve outgoing arrows of multiple work items at a time

This section explains the typical steps that you follow to use this interface. A programming sample is presented, which retrieves UDAs of multiple process instances. You can find the complete programming code in the `BatchDetailsRetrieval.java` sample file. The sample file also contains an example of how to retrieve outgoing arrows of multiple work items.

To retrieve UDAs of multiple process instances at a time:

1. Make sure that you have a list of process instance IDs for which you want to retrieve UDAs later. To do so, you typically use one of the process instance filters provided by the `WFObjectList` interface.

   In the programming sample, to prepare for the bulk processing functionality, the active process instances of the logged-in user are retrieved:

   ```
   WFObjectList list = WFObjectFactory.getWFObjectList(session);
   list.openBatchedList(Filter.MyActiveProcesses);
   Object[] batch = list.getNextBatch(10);
   ```

   For more information about filters, refer to section *Advanced Filtering & Sorting API* on page 110.

2. If process instances are found, assign their IDs to an array.

   ```
   if (batch == null) {
      < ... >
      return;
   }
   long[] idValues = new long[batch.length];
   for (int i = 0; i < batch.length; i++) {
      idValues[i] = ((ProcessInstance) batch[i]).getId();
   }
   ```

3. As the list of process instances is not needed for further processing, you are recommended to close it.

   ```
   list.closeList();
   ```

4. Retrieve the UDAs of all of the process instances that you previously retrieved. To do so, create a `WFDetailsList` instance and call `getUDAsForProcessInstances()`.

   ```
   WFDetailsList detailsList = WFObjectFactory.getWFDetailsList(session);
   ProcessInstancesUDASet udaInfo = detailsList
                   .getUDAsForProcessInstances(idValues);
   ```

   The `udaInfo` object stores the UDAs of the process instances in a Java Map. The process instance ID is used as key, and another Java Map representing the UDAs of that process instance is used as value.

   Next, you will retrieve the UDAs of a particular process instance.

5. To retrieve the UDAs of a particular process instance, use `getUDAsForProcessInstance()` from the `ProcessInstancesUDASet` class:

   ```
   for (int i = 0; i < idValues.length; i++) {
      Map udaData = udaInfo.getUDAsForProcessInstance(idValues[i]);
      < ... >
   }
   ```

   In the sample, the UDAs of a process instance are assigned to a Java Map named `udaData`. This Map uses the UDA's name as key and an object of type `UDAData` as value. The `UDAData` object stores the name, data type and value of an individual UDA.

6. To process the individual UDAs of a process instance, use a Java Iterator to iterate through the UDAs of that process instance. You can retrieve the name, data type and value of a UDA, using `getUdaName()`, `getUdaType()` and `getUdaValue()` from the `UDAData` class.

   The following sample creates a Java Iterator, which is required to iterate through the elements of the `udaData` Map. Each object returned by `iterator.next()` is casted to a `UDAData` object. Finally, the UDA's name, data type and value are retrieved for further processing.

```
Iterator iterator = udaData.keySet().iterator();
while (iterator.hasNext()) {
   UDAData data = (UDAData) udaData.get(iterator.next());
   System.out.println("\tUDA name: '" + data.getUdaName() + "'");
   System.out.println("\tUDA type: '" + data.getUdaType() + "'");
   System.out.println("\tUDA value: '" + data.getUdaValue()
                       + "'\n");
}
```

6.6 Using Process Comments

To facilitate flexible communication among users, Interstage BPM allows users to add comments to the node or process instance.

You can add, retrieve, and delete comments on `ProcessInstance` and `NodeInstance` interfaces in the `com.fujitsu.iflow.model.workflow` package using the following APIs.

Note: The following APIs can be used for both process instance and node instance.

- To fetch a comment - `getComments()`. This API returns an array of comments of a process instance or node instance. Comments can be fetched by all users.

  For example, to fetch comments from

  - the node instance object `activity` of an Activity Node, use `activity.getComments();`
  - the process instance object `pi`, use `pi.getComments();`

  You can retrieve comments' details, using `Comment` interface in the `com.fujitsu.iflow.model.workflow` package. For example, the following APIs can be used.

  - getMessage(): Returns message for this Comment
  - getId(): Returns comment ID for this Comment
  - getUserId(): Returns the user ID who has added this Comment
  - getTimeStamp(): Returns the timestamp in milliseconds, for when this comment is added.

  Refer to the API Javadoc for details on the available classes and methods.

  The following sample code gives an example of retrieving individual comment information from a process instance object `pi`.

```
Comment [] comments = pi.getComments();
for(int count=0;count<comments.length;count++){
     System.out.println("CommentId: "+comments[count].getId());
     Date date = new Date(comment[count].getTimeStamp());
     System.out.println("Timestamp: "+ date);
     System.out.println("UserId: "+comments[count].getUserId());
     System.out.println("CommentMsg: "+comments[count].getMessage());
     System.out.println("Deleted flag: "+comments[count].isDeleted());
  }
```

Users can also fetch all comments of a process instances and comments from associated node instances using `ProcessInstance.getAllComments()` API.

The following sample code gives an example of fetching all comments from the process instance object `pi` and comments from all node instances of process instance object `pi`.

```
 Comment [] comments = pi.getAllComments();
```

Note: If no comments are present on the process instance or node instance, an empty array of comments is returned.

- To add a comment -`addComment()`. This API adds the user-provided comment to the process instance or node instance. Administrators, process instance owners, process instance initiator, and assignees of each work item can add comments on that process instance. Administrators, process instance owners, process instance initiator, and assignees of each work item on that node can add comments on that node instance.

Note: For a node instances, comments can be added to the Activity nodes, Voting nodes, Compund Activity nodes and Dynamic Activity nodes.

Note: Comments can be added on a node instance and process instance, if the process instance is not locked by another user.

The following sample code gives an example of adding a comment to the node instance object `activity` of an Activity Node.

```
String comment = "This is a comment on the node instance";
activity.addComment(comment);
```

The following sample code gives an example of adding a comment to the process instance object `pi`.

```
String comment = "This is a comment on the process instance";
pi.addComment(comment);
```

- To delete a comment - `deleteComment()`. This API marks the given comment as deleted. Comments can be deleted by the user who has added the comment and administrators.

Note: A comment can be deleted from a process instance, only if the process instance is not locked by another user.

The following sample code gives example of deleting a comment from the node instance object `activity` of an Activity Node.

```
Comment comments = activity.getComments();
long commentId = comments[0].getId();
activity.deleteComment(commentId);
```

Note: Make sure that the CommentID for the comment that you want to delete, is specific to the node instance or process instance from which you want to delete the comment. If you try to delete a comment from the node instance using a CommentID of a comment belong to the process instance, you will get an error. Same holds true for process instance.

The following sample code gives an example of deleting a comment from the process instance object `pi`.

```
Comment comments = pi.getComments();
long commentId = comments[0].getId();
pi.deleteComment(commentId);
```

6.7 Special User Defined Attribute Properties

6.7.1 Working with User Defined Attributes of Type XML

When modeling the data that will be used in a process, you might need to handle structured data in XML format. For example, an external system might pass an XML structure to Interstage BPM which needs to be modified during process execution and then passed back.

Interstage BPM provides a special data type for XML data. User Defined Attributes (UDAs) of type XML allow you to access and manipulate the entire XML structure as well as substructures, single elements and attributes. XPath expressions are used to select components from the XML data for further processing.

Before storing data in a UDA of type XML, the Interstage BPM Server checks whether the data is well-formed. Only well-formed XML data can be stored. Optionally, you can specify an XML schema for each UDA and validate the XML data against that schema.

UDAs of type XML can be created, updated, read and deleted like any other UDA. For UDAs of type XML, there are several additional methods provided for the following:

- setting and retrieving an XSD schema
- setting and retrieving the UDA value as file
- setting and retrieving the UDA value using an XPath expression, i.e. you can set and retrieve a value for the entire XML structure, for a substructure and for a single element.
- validating the XML content of a UDA

When mapping UDAs of type XML, the `DataItemMappingElement` class of the `WorkItem` interface allows for

- mapping a full XML structure to a full XML structure
- mapping a single element to another UDA of type XML
- mapping an XML substructure to another XML substructure
- mapping an XML substructure to a full XML structure

Below you find some examples for working with UDAs of type XML. Refer to the `ComplexPlan.java` sample file for the entire sample code.

To work with UDAs of type XML:

1. When adding UDAs, specify `TYPE_XML` as the data type:

```
DataItemRef dataRefOrder = procDef.addDataItemRef(XMLUDA_BOOKSTORE,
             DataItemRef.TYPE_XML, XMLVAL_BOOKSTORE);
```

   The XMLUDA_BOOKSTORE constant defines the name of the UDA:

```
private final static String XMLUDA_BOOKSTORE = "XMLBookstore";
```

   The XMLVAL_BOOKSTORE constant defines the value for the UDA in the following XML structure:

```
private final static String XMLVAL_BOOKSTORE =
     "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>"
     + "<bookstore>"
     + "<book category=\"COOKING\">"
     + "<title lang=\"en\">Everyday Italian</title>"
     + "<author>Giada De Laurentiis</author>"
     + "<year>2005</year>"
     + "<price>30.00</price>"
     + "</book>"
     + "<book category=\"CHILDREN\">"
     + "<title lang=\"en\">Harry Potter</title>"
     + "<author>J K. Rowling</author>"
     + "<year>2005</year>"
     + "<price>29.99</price>"
     + "</book>"
     + "<book category=\"WEB\">"
     + "<title lang=\"en\">XQuery Kick Start</title>"
     + "<author>James McGovern</author>"
     + "<author>Per Bothner</author>"
     + "<author>Kurt Cagle</author>"
     + "<author>James Linn</author>"
     + "<author>Vaidyanathan Nagarajan</author>"
     + "<year>2003</year>"
     + "<price>49.99</price>"
     + "</book>"
     + "<book category=\"WEB\">"
     + "<title lang=\"en\">Learning XML</title>"
     + "<author>Erik T. Ray</author>"
     + "<year>2003</year>"
     + "<price>39.95</price>" + "</book>" + "</bookstore>";
```

2. To read a single element of the XML structure, in this case the price for the first book, specify the element using an XPath expression:

```
DataItem di = pi.getDataItem(XMLUDA_BOOKSTORE);
// retrieve the price for the first book
String price = di.getElementValue("/bookstore/book[1]/price/text()");
            logger.log(Logger.DEBUG, "Price is: " + price);
```

3. To read the name of an XML node in the substructure, in this case the name of the XML node defining the second book:

```
org.w3c.dom.Node subNode = di.getSubTreeValue("/bookstore/book[2]");
            logger.log(Logger.DEBUG, "Node name is: " +
subNode.getNodeName());
```

   Note that you must use the `org.w3c.dom.Node` interface for addressing the XML node, because Interstage BPM also offers a `Node` interface.

4. To compare the value of a single XML element:

```
String xPath = "/bookstore/book[3]/author/text()";
     String expAuthor = "Erik T. Ray";
     logger.log(Logger.DEBUG, "Author: " + di.getElementValue(xPath));
     if (di.getElementValue(xPath).equals(expAuthor)) {
     logger.log(Logger.DEBUG, "Expected author found");
```

6.7.2 Worklist UDAs

Many applications require that the work list display certain User Defined Attribute (UDA) values that are used to help determine which work item will be worked on next. Normal access to UDA values from the worklist can be slow - Worklist UDAs provide a faster way of access. In addition, marking a UDA as Worklist UDA allows you to sort and filter a list of workflow elements by this UDA.

Worklist UDAs are returned as an object within the worklist object. The same set of UDA values will be included regardless of the activity from which the worklist object is generated. These values will be included with the work item object, and can be immediately read without having to load the entire process instance into the model.

There are two methods that can be used to obtain UDA information on a particular work item using the Model API. You start by obtaining the work item object. Then, you can either obtain all of the UDAs (DataItems) of this work item with `getDataItems()` or a subset that consists of the work item `DataItems` with the `getWorklistDataItems()` method.

The prerequisite for using the `getWorklistDataItems()` method from the `WorkItem` interface is that you mark certain `DataItems` as Worklist UDAs. This can be accomplished when you design the process definition by using `markAsWorkListUDA()` from the `DataItemRef` interface.

There are the following worklist UDA-specific methods:

- `markAsWorkListUDA()`: Marks a UDA as a worklist UDA.
- `boolean isWorkListUDA()`: Checks, whether a UDA is a worklist UDA.

The following sample code demonstrates how you can define UDAs and mark them as Worklist UDAs when you create a process definition. Refer to the `ComplexPlan.java` sample file for the complete process definition.

```
//*** Definition of UDAs ***
DataItemRef udaPrice = plan.addDataItemRefWithId("Price",
      "WLUDA_PRICE", DataItemRef.TYPE_FLOAT, "0.0");
DataItemRef udaQty = plan.addDataItemRefWithId("Quantity",
      "WLUDA_QTY", DataItemRef.TYPE_INTEGER, "0");
DataItemRef udaTotal = plan.addDataItemRefWithId("Total",
      "WLUDA_TOTAL", DataItemRef.TYPE_FLOAT, "0.0");

//Mark the UDAs as Worklist UDAs.
udaPrice.markAsWorkListUDA(true);
```

```
udaQty.markAsWorkListUDA(true);
udaTotal.markAsWorkListUDA(true);
```

You can use these Worklist UDAs, for example, to retrieve process instances that match filtering criteria based on values of UDAs. For a programming sample, refer to the `listProcsByUDA()` method in the `ComplexPlan.java` sample file.

Note: UDAs of type XML cannot be marked as worklist UDAs.

6.8 Using Extended Attributes

Interstage BPM allows you to define extended attributes. Extended attributes store extra information that is not native to Interstage BPM. You can assign extended attributes to process definitions, nodes, and arrows.

Extended attributes are specified as an XML document. Values of extended attributes are always of type String. The following XML schema defines the format of extended attributes:

```
<xsd:element name="ExtendedAttributes">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="xpdl:ExtendedAttribute" minOccurs="0"
            maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
<xsd:element name="ExtendedAttribute">
   <xsd:complexType mixed="true">
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
         <xsd:any minOccurs="0" maxOccurs="unbounded"/>
      </xsd:choice>
      <xsd:attribute name="Name" type="xsd:NMTOKEN" use="required"/>
      <xsd:attribute name="Value" type="xsd:string"/>
   </xsd:complexType>
</xsd:element>
```

When using extended attributes, make sure that your XML documents comply with this XML schema. The following sample is a valid instance:

```
<ExtendedAttributes>
   <ExtendedAttribute Name="SomeAttributeName" Value="Some value"/>
   <ExtendedAttribute Name="AnotherAttributeName" Value="200"/>
</ExtendedAttributes>
```

The `Plan`, `Node`, and `Arrow` interfaces provide the following methods related to extended attributes:

- `setExtendedAttributes (org.w3c.dom.Document attrs)`: Assigns extended attributes to the process definition, node or arrow. `attrs` is an XML document that contains the extended attributes to be assigned.
- `getExtendedAttributes()`: Returns the XML document that contains all extended attributes of the process definition, node or arrow.

`ComplexPlan.java` contains a sample that shows how to work with extended attributes. A sample process definition `CxPD_PurchaseOrder` is created that is supposed to be used by divisions in different countries. Therefore, certain strings like the process definition name and node names need to be available in different languages. Extended attributes are used to store the language-specific names.

The following sections explain how to assign and retrieve extended attributes.

6.8.1 Assigning Extended Attributes

To assign extended attributes to a process definition, node, or arrow:

1. Create an XML document containing the extended attributes that you want to assign to an element.

   The sample program `ComplexPlan.java` creates two XML documents: One for the language-specific names of the `CxPD_PurchaseOrder` process definition and another for the language-specific names of an Activity Node. The following sample code shows the XML document that is created for the language-specific names of the process definition:

```
StringBuffer sb_xmlProcDef = new StringBuffer();
sb_xmlProcDef.append("<ExtendedAttributes>");
sb_xmlProcDef.append("<ExtendedAttribute
   Name=\"DE\" Value=\"Kaufauftrag\"/>");
sb_xmlProcDef.append("<ExtendedAttribute
   Name=\"EN\" Value=\"Purchase Order\"/>");
sb_xmlProcDef.append("<ExtendedAttribute
   Name=\"FR\" Value=\"Contrat d'achat\"/>");
sb_xmlProcDef.append("</ExtendedAttributes>");
InputStream in_xmlProcDef = new
ByteArrayInputStream(sb_xmlProcDef.toString().getBytes("UTF-8"));
```

   This is the XML document that is created for the language-specific names of an Activity Node:

```
StringBuffer sb_xmlProcDef = new StringBuffer();
sb_xmlProcDef.append("<ExtendedAttributes>");
sb_xmlProcDef.append("<ExtendedAttribute Name=\"DE\"
   Value=\"Bestellformular ausfuellen\"/>");
sb_xmlProcDef.append("<ExtendedAttribute Name=\"EN\"
   Value=\"Fill out Purchase Requisition\"/>");
sb_xmlProcDef.append("<ExtendedAttribute Name=\"FR\"
   Value=\"Remplir le formulaire de commande\"/>");
sb_xmlProcDef.append("</ExtendedAttributes>");
InputStream in_xmlActivity = new
ByteArrayInputStream(sb_xmlProcDef.toString().getBytes("UTF-8"));
```

   There are some restrictions on the names you can use for extended attributes. For more information, refer to section *Names of Extended Attributes* on page 124.

2. Read the XML documents containing the extended attributes.

```
DocumentBuilderFactory dbf =
   DocumentBuilderFactory.newInstance();
DocumentBuilder builder = dbf.newDocumentBuilder();
Document extAttPlan = builder.parse(in_xmlProcDef);
Document extAttNode = builder.parse(in_xmlActivity);
```

3. Assign the extended attributes to the element.

   The following sample code shows how to assign extended attributes to a process definition and to a node:

```
procDef.startEdit();
procDef.setExtendedAttributes(extAttPlan);
nodePR.setExtendedAttributes(extAttNode);
procDef.commitEdit();
```

Once you have assigned extended attributes to an element, you can use `getExtendedAttributes()` to retrieve them for further processing.

6.8.2 Retrieving the Value of an Extended Attribute

Prerequisite: You have assigned extended attributes to an element as explained in section *Assigning Extended Attributes* on page 122.

To retrieve the value of a particular extended attribute:

1. Read the extended attributes that have been assigned to the element.

   The following sample shows how to read the extended attributes that have been assigned to a process definition:

   ```
   Document resExtAttPlan = procDef.getExtendedAttributes();
   ```

   Alternatively, you can use your own custom Java action to read the extended attribute assigned to a process definition. Refer to section *Accessing Workflow Data Using the Server Enactment Context Interface* on page 92 to create your own custom Java action. Make sure that you use `ServerEnactmentContext.getProcessDefinitionExtendedAttributes()` API in your customized Java action.

2. Build an XPath expression that specifies the location of the extended attribute within the XML document.

   In the following sample, `exAttName` stores the name of an extended attribute that is to be retrieved.

   ```
   String xpath = "/ExtendedAttributes/ExtendedAttribute[@Name=\""
            + exAttName + "\"]";
   ```

   If `exAttName` has the value `EN`, the resulting XPath expression is:

   ```
   "/ExtendedAttributes/ExtendedAttribute[@Name=\"EN\"]"
   ```

3. Select the XML node that corresponds to the extended attribute:

   ```
   org.w3c.dom.Node ndLang =
                  XPathAPI.selectSingleNode(resExtAttPlan, xpath);
   ```

   The resulting XML node might be, for example:

   ```
   <ExtendedAttribute Name="EN" Value="Purchase Order"/>
   ```

4. To access the value of the extended attribute:

   ```
   String strRes =
             ndLang.getAttributes().getNamedItem("Value").getNodeValue();
   ```

   If an element has no extended attributes assigned, `getExtendedAttributes()` or `selectSingleNode()` return a null value.

6.8.3 Retrieving all Extended Attributes of an Element

Prerequisite: You have assigned extended attributes to an element as explained in section *Assigning Extended Attributes* on page 122.

To retrieve all extended attributes of a particular element:

1. Build an XPath expression that specifies the location of the extended attributes within the XML document:

```
String xpath = "/ExtendedAttributes/ExtendedAttribute";
```

2. Read the extended attributes that have been assigned to the element.

   The following sample retrieves all nodes of a process definition and checks whether extended attributes have been assigned to a node. If a node has extended attributes, they are assigned to a list of XML nodes.

```
Node[] nodes = procDef.getNodes();
for (int iNodeCnt=0; iNodeCnt<nodes.length; iNodeCnt++) {
   Node tmpNode = nodes[iNodeCnt];
   Document resExtAttNode = tmpNode.getExtendedAttributes();
   if (resExtAttNode != null) {
      NodeList nodelistNode =
         XPathAPI.selectNodeList(resExtAttNode, xpath);
   . . . }
}
```

   Alternatively, you can use your own custom Java action to read the extended attribute assigned to an activity definition. Refer to section *Accessing Workflow Data Using the Server Enactment Context Interface* on page 92 to create your own custom Java action. Make sure that you use `ServerEnactmentContext.getActivityExtendedAttributes()` API in your customized Java action.

3. To access a particular extended attribute within the list of XML nodes:

```
for (int iNListCnt=0; iNListCnt<nodelistNode.getLength();
           iNListCnt++) {
   org.w3c.dom.Node exAttNode = nodelistNode.item(iNListCnt);
. . . }
```

   Suppose `nodelistNode` contains the following XML nodes:

```
<ExtendedAttribute Name="DE"
   Value="Bestellformular ausfuellen"/>
<ExtendedAttribute Name="EN"
   Value="Fill out Purchase Requisition"/>
<ExtendedAttribute Name="FR"
   Value="Remplir le formulaire de commande"/>
```

   In this case, `nodelistNode.item(1)` returns the following XML node:

```
<ExtendedAttribute Name="EN"
   Value="Fill out Purchase Requisition"/>
```

6.8.4 Names of Extended Attributes

When defining extended attributes, make sure that their names do not interfere with names used by Interstage BPM or by the XPDL standard.

Extended Attributes Native to Interstage BPM

When converting a process definition to XPDL, Interstage BPM encodes some information that is native to Interstage BPM into extended attributes. Below is the list of names used by Interstage BPM for its native extended attributes.

Note: Do not use these names as extended attribute values.

XML nodes having these attribute values may be removed when the process definition is converted to XPDL. Interstage BPM uses some of these XML nodes to map internal data; they are extracted from the extended attributes using `getExtendedAttributes()`.

```
Associations

Artifacts

ChildPlan

ChildPlanId

CommitJavaActionSet

Coordinates

CustomNodeType

DataMapping

DataMappings

Description

EnableFutureWorkItems

EndPoint

EpilogueJavaActionSet

ExposedField

FormList

FormsList

InitJavaActionSet

InTransaction

IsWorkItemUDA

IteratorCountUDA

NodeType

Organization

ParentVersion

PrivateData

ProcessDefinitionId

ProcessOwnerRole

ProcessOwnerRoleJavaActionSet

ProcessTypeId

PrologueJavaActionSet

RoleJavaActionSet

SameSubPlanVersion
```

`StartPoint`

`State`

`subProcessDefinitionURI`

`SWIM_LANES`

`TemplateIdentifier`

`TimerDefSet`

`Title`

`TriggerDefSet`

`VersionComment`

`ViewerScript`

Example: The following XML fragment defines two extended attributes, `StartPoint` and `StartPoint1`. The definition of `StartPoint` is faulty because this name is used by Interstage BPM for one of its native extended attributes.

```
<ExtendedAttributes xmlns:x="http://example.com">
   <ExtendedAttribute Name="StartPoint">
      <x:StartPoint>
         PointA
      </x:StartPoint>
   </ExtendedAttribute>
   <ExtendedAttribute Name="StartPoint1">
      <x:StartPoint1>
         PointB
      </x:StartPoint1>
   </ExtendedAttribute>
</ExtendedAttributes>
```

When the process definition is converted to XPDL, the `StartPoint` XML node will be removed. The resulting XPDL fragment would look like this:

```
<ExtendedAttributes xmlns:x="http://example.com">
   <ExtendedAttribute Name="StartPoint1">
      <x:StartPoint1>
         PointB
      </x:StartPoint1>
   </ExtendedAttribute>
</ExtendedAttributes>
```

6.8.5 Namespace of Extended Attributes

Interstage BPM generates namespace-aware XPDL that fully complies with the XPDL 2.0 standard.

Note: If you are using extended attributes, you are recommended to prefix their names with valid namespaces. This ensures that the XPDL generated by Interstage BPM remains fully compliant with the XPDL standard.

If you do not use a namespace when defining extended attributes, Interstage BPM adds the default `ibpm` namespace to generate namespace-aware XPDL that fully complies with the XPDL 2.0 standard.

Example 1

The following sample defines extended attributes that are prefixed with the namespaces `x` and `y`:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtendedAttributes xmlns:x="http://example.com"
   xmlns:y="http://example.net">
   <ExtendedAttribute Name="myAttribute">
      <x:myTag1>
         myData1
      </x:myTag1>
      <y:myTag2>
         <myTag3>
            myData2
         </myTag3>
      </y:myTag2>
   </ExtendedAttribute>
</ExtendedAttributes>
```

After importing such a process definition, `getExtendedAttributes()` returns the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtendedAttributes xmlns="http://www.wfmc.org/2004/XPDL2.0alpha"
xmlns:xpdl="http://www.wfmc.org/2004/XPDL2.0alpha"
xmlns:ibpm="http://fujitsu.com/ibpm1" xmlns:x="http://example.com"
xmlns:y="http://example.net">
   <ExtendedAttribute Name="myAttribute">
      <x:myTag1>
         myData1
      </x:myTag1>
      <y:myTag2>
         <myTag3>
            myData2
         </myTag3>
      </y:myTag2>
   </ExtendedAttribute>
</ExtendedAttributes>
```

Note that additional namespace attributes (the `ibpm` namespace and the XPDL 2.0 standard namespaces) have been added to the root element. The immediate children of the `ExtendedAttribute` element have not been prefixed with `ibpm` because appropriate namespaces were already present in the original XML document.

Example 2

This XML fragment defines some extended attributes without namespace. In the context of the XPDL standard, this XML is invalid because the children of the `ExtendedAttribute` element have no namespace prefix.

```
<!-- INVALID -->
<ExtendedAttributes>
   <ExtendedAttribute Name="myAttribute">
         <myTag1>
             myData1
         </myTag1>
         <myTag2>
            <myTag3>
               myData2
            </myTag3>
         </myTag2>
    </ExtendedAttribute>
</ExtendedAttributes>
```

After importing such a process definition, `getExtendedAttributes()` returns the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<ExtendedAttributes xmlns="http://www.wfmc.org/2004/XPDL2.0alpha"
xmlns:xpdl="http://www.wfmc.org/2004/XPDL2.0alpha"
xmlns:ibpm="http://fujitsu.com/ibpm1">
   <ExtendedAttribute Name="myAttribute">
      <ibpm:myTag1>
         myData1
      </ibpm:myTag1>
      <ibpm:myTag2>
         <myTag3>
            myData2
         </myTag3>
      </ibpm:myTag2>
   </ExtendedAttribute>
</ExtendedAttributes>
```

The returned XML document has changed in two aspects:

- The `ibpm` namespace and the XPDL 2.0 standard namespace attributes have been added to the root element.
- The `ibpm` prefix has been added to the immediate children of the `ExtendedAttribute` element.

6.9 Transaction Control

The Interstage BPM transaction control mechanism allows for rolling back transactions to a specific point in process execution. This mechanism also improves the system’s overall performance.

A process often consists of a sequence of nodes. You can decide whether or not the transaction stays open after node completion. If the transaction stays open, the next node in the process will be processed in the same transaction.

Consider the following:

- Performance: Committing a transaction saves everything to the database, and might force everything to be read from the database again to start the subsequent transaction. If you have a long chain of nodes that do very little computation (for example, a chain of branch nodes deciding among many possible directions) the overhead of saving to the database every time will be significant and unnecessary.
- Rollback upon errors: If a low level error occurs during processing, the current transaction will be rolled back to the last commit point. If four nodes are all part of the same transaction, all four nodes will be rolled back in case of an error, and upon restart all four will be executed again. Committing the transaction somewhere in the middle means that it will rollback only to that point.

Controlling Transactions with the Model API

There are two methods in the `com.fujitsu.iflow.model.workflow.Node` interface supporting the transaction control feature. These are:

- `setNodeInTxn()`: Sets the transaction flag on this node. This transaction flag is used to decide whether the transaction should be committed when this node completes.

- `getNodeTxnStatus()`: Gets the node transaction status. Depending upon whether the node is set to complete in the same transaction.

For more information on these methods or the Model API in general, refer to the API Javadoc.

For a node to automatically commit a transaction with the next commit operation (and not directly), set `setNodeInTxn()` to `true`, for example:

```
fillOutNode.setNodeInTxn(true);
```

For checking the transaction status of a node, use `getNodeTxnStatus()`, for example:

```
logger.log(Logger.INFO, "");
logger.log(Logger.INFO, "Get status of node transaction");
boolean nodeTxnStatus =
   workItem.getNodeInstance().getNodeTxnStatus();
logger.log(Logger.INFO, "Status: " + nodeTxnStatus);
```

Refer to the `ComplexPlan.java` and `ProcessExecution.java` samples for the complete node definition.

Rolling Back Transactions in External Systems

You can define actions for Java Actions which perform a "cleanup" before a transaction is rolled back and a process instance is set to the error state. This includes a rollback of all Java Actions in a Java Action Set, and allows you to perform general actions (e.g. sending notification emails in any error case), or executing some specific actions before setting the process instance to error state.

If you do not define any error handling for a Java Action, the following happens: When an exception is thrown in this Java Action, the transaction will be rolled back. A rollback, however, is only possible for changes in the Interstage BPM Application Server context. Any transactions in external systems cannot be rolled back, for example, if a row has been added to a database. Therefore, it is sometimes necessary to manually clean up external systems to ensure a consistent state of all systems used in the transaction. Optionally, you can use Compensate Action Sets.

Refer to section *Dealing with Errors in Java Actions* on page 102 for more information.

6.10 Using Triggers

The purpose of a trigger is to move data, typically XML files, coming from an external system’s data source into Interstage BPM process instances. These process instances are either started or directed by the incoming XML data files, and the process instances perform functions that use the data source data. The XML data files are called event data files.

Triggers can be configured to recognize the data in the event data files, so that the data can be mapped to User Defined Attributes (UDAs) in the Interstage BPM process instances, or triggers use the event data files directly and write the data to the defined UDAs.

Triggers are very versatile and can be associated with a process definition, an Event Activity Node or an Activity Node. Each of these provides a different scope of behaviour.

- When defined on process definition level, the trigger creates a new process instance and starts it.
- When defined on node level, the trigger makes a choice so that process execution moves forward to the next node.

A trigger contains two components: the event that drives the trigger and the action that the trigger takes when it is fired.

6.10.1 How It Works

Event data files (XML files) are sent from an external data source to a directory configured in the Interstage BPM File Listener. The Trigger Handler checks all the triggers configured on Interstage BPM and activates them in case the File Listener notifies the Trigger Handler of incoming XML files. The trigger then acts according to its definition:

- If the trigger is defined on process definition level, it starts a process instance from the process definition in which the trigger is defined and moves the XML data into the User Defined Attributes (UDAs) of the process according to the data mapping specified in the trigger definition.
- If the trigger is defined on node level, it makes a choice on the node for which it is defined and moves the XML data into the UDAs of the process instance according to the data mapping specified in the trigger definition.

Triggers can not only be fired by the File Listener, but also explicitly using the `WFObjectFactory.processTriggerEvent()` method of the Model API in `package com.fujitsu.iflow.model.workflow`. This method processes the event data supplied in XML format for all active trigger definitions.

Trigger State

A trigger is allowed to be in one of three possible states: Default, Active and Inactive. The state of a trigger is a property at definition level, i.e. changing a trigger obtained on instance level, e.g. by calling `ProcessInstance.getTriggers()`, will affect all triggers of all process instances belonging to the same process definition as well. Changes to the state of the process definition or process instance have no effect on the state of the contained trigger(s).

Control Conditions

Triggers may be designed to operate on certain process instances only. In such a case, you can provide control conditions to facilitate the trigger to locate the right process instance to operate on when the trigger is fired. If you do not specify control conditions, the trigger is always fired as soon as the File Listener detects a new XML file in the specified directory.

For example, for triggers defined on process definition level: If you define triggers for several process definitions, every time the File Listener finds an XML file, a process instance for every process definition will be created. This means that an XML file can create an arbitrary number of process instances of different process definitions. You can avoid this behavior by specifying control conditions so that the creation of a process instance depends on the content of the XML file. Only if the XML file content matches the control conditions, the trigger will be fired, i.e. a process instance will be created.

Control conditions operate on User Defined Attributes. For each attribute used, you specify an element of the incoming event data via an XPath expression. When the trigger is fired, only those processes are selected in which the value of the specified attribute matches the value of the corresponding element in the incoming event.

Action Specifications

Action specifications define the operation that the trigger will perform. Since each trigger is defined to handle only one type of operation, all the actions defined in a trigger relate to the same operation. This means, for example, that a trigger defined to make a choice will not be able to start a process.

For triggers defined on node level (make choice triggers), it is required to specify the actions that can be performed using the method `addAction(String, String)`. The first parameter contains the name of the outgoing arrow that is to be chosen if the condition specified in the second parameter is true.

When defining a trigger on an Event Activity Node, exactly one action has to be defined because the node has exactly one outgoing arrow.

When defining a trigger on an Activity Node, the number of actions that can be defined depends on the number of outgoing arrows defined for the node. The sequence of evaluating the conditions, which is identical to the second parameter in the method, depends on the sequence of the actions. The condition of the action with `index [0]` is evaluated first. If its condition is true, the conditions of the other actions will not be evaluated. The process instance continues with the arrow defined in the action whose condition was found to be true.

6.10.2 Defining a Trigger

This section explains how to define a trigger on process definition level. You can find the complete programming code of the examples presented in this section in the sample file `TriggerTimerSample.java`.

Note: The `TriggerTimerSample.java` file contains an additional sample for adding a trigger to an Activity Node. For information about triggers defined on Event Activity Nodes, refer to section *Using Event Activity Nodes* on page 133.

To define a trigger:

1. Configure the File Listener that starts the triggering mechanism so that it looks into the directory where incoming event data files are stored. For instruction on configuring and testing the File Listener, refer to section *File Listeners* on page 134.
2. Create the relevant User Defined Attributes (UDAs) for your process definition.

   In the sample file, the following UDAs are created:

```
plan.addDataItemRef("OrderID",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("OrderPerson",
   DataItemRef.TYPE_STRING,"Initial Value");
plan.addDataItemRef("Name",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("Address",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("City",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("State",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("ZIP",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("Title",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("Note",
   DataItemRef.TYPE_STRING, "Initial Value");
plan.addDataItemRef("Quantity",
   DataItemRef.TYPE_INTEGER, "1");
plan.addDataItemRef("Price",
   DataItemRef.TYPE_BIGDECIMAL, "0.00");
```

3. Add the trigger to your process definition, define control conditions and set its state to active.

In the sample below, two control conditions are defined. The first one requires that the value for the price of the article must be bigger than 10, and the second one states that the name of the adressee ("shipto" tag) must be "Pete Gagnet".

```
TriggerDef myTrigger = plan.addTrigger("myTrigger",
   TriggerDef.TYPE_START_PROCESS);
myTrigger.setDescription("Will start a process instance.");

// add control conditions
String[] conditions = new String[2];
conditions[0] = "toFloat(eventData.getXMLData
   ( \"/shiporder/item/price/text()\" ))  > 10";
conditions[1] = "eventData.getXMLData
   ( \"/shiporder/shipto/name/text()\" ) == \"Pete Gagnet\"";
myTrigger.setControlConditions(conditions);

// activate trigger
myTrigger.setState(TriggerDef.STATE_ACTIVE);
```

4. Specify the data mapping for extracting the values of the event elements of the XML file that will be used to fire the trigger, and write them to the UDAs, for example:

```
myTrigger.addDataMap("OrderID","shiporder/@orderid");
myTrigger.addDataMap("OrderPerson","shiporder/orderperson/text()");
myTrigger.addDataMap("Name","shiporder/shipto/name/text()");
myTrigger.addDataMap("Address","shiporder/shipto/address/text()");
myTrigger.addDataMap("City","shiporder/shipto/city/text()");
myTrigger.addDataMap("State","shiporder/shipto/state/text()");
myTrigger.addDataMap("ZIP","shiporder/shipto/zip/text()");
myTrigger.addDataMap("Title","shiporder/item/title/text()");
myTrigger.addDataMap("Note","shiporder/item/note/text()");
myTrigger.addDataMap("Quantity","shiporder/item/quantity/text()");
myTrigger.addDataMap("Price","shiporder/item/price/text()");
```

5. Create an XML file that will be used by the trigger to start an instance of a process definition and fill the file with values. You can either use an existing XML file or create it using the API, for example:

```
// Create XML file in the folder specified by the FILELISTENER_PATH
File xml = new File(FILELISTENER_PATH + "shiporder.xml");

// Write XML data into the file
FileWriter fw = new FileWriter(xml);
BufferedWriter bw = new BufferedWriter(fw);
bw.write("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>");
bw.newLine();
bw.write("<shiporder orderid=\"889923\">");
bw.newLine();
bw.write("<orderperson>Joan Smith</orderperson>");
bw.newLine();
bw.write("<shipto>");bw.newLine();
bw.write("<name>Pete Gagnet</name>");
bw.newLine();
bw.write("<address>325 Eastwood Ave</address>");
bw.newLine();
bw.write("<city>" + CITY_OF_ORDER + "</city>");
bw.newLine();
bw.write("<state>OH</state>");
bw.newLine();
bw.write("<zip>20131-1234</zip>");
bw.newLine();
bw.write("</shipto>");bw.newLine();
bw.write("<item>");bw.newLine();
String titleStr = "<title>" + ITEM_TITLE + "</title>";
bw.write("(titleStr);bw.newLine();
bw.write("<note>Special Edition</note>");bw.newLine();
titleStr = "<quantity>" + QUANTITY_ORDER + "</quantity>";
bw.write("(titleStr);bw.newLine();
bw.write("<price>39.99</price>");bw.newLine();
bw.write("</item>");bw.newLine();
bw.write("</shiporder>");bw.newLine();
bw.close();
```

Note: When specifying dates in the XML file, make sure to use a date format that matches the locale setting of the Interstage BPM Server.

6.11 Using Event Activity Nodes

An Event Activity Node represents a step in a process which is driven by an external event. Each Event Activity Node has a trigger defined that is supposed to fire when data, typically XML files, comes in from an external system. Once the data arrives, the trigger moves the data into User Defined Attributes (UDAs) according to the trigger's definition. The trigger then makes a choice and process execution moves forward to the next node. No human interaction is required.

This section explains the typical steps involved in working with Event Activity Nodes. You can find the complete programming code for the example presented in this section in the sample file `EventActivityNodeSample.java`.

For general information about triggers and how they work, refer to section *Using Triggers* on page 129.

To use Event Activity Nodes:

1. Create an Event Activity Node using `addNode()`. Set the constant `nodeType` to `TYPE_EVENT_ACTIVITY`.

```
Node activity = plan.addNode("EventActivity",
   Node.TYPE_EVENT_ACTIVITY);
```

2. Add exactly one outgoing arrow to the Event Activity Node, for example

```
plan.addArrow("Exit", activity, exit);
```

3. Define a make choice trigger for the Event Activity Node that you previously created.

```
TriggerDef trigger = activity.addTrigger("ChoiceTrigger",
   TriggerDef.TYPE_MAKE_CHOICE);
```

4. Optionally, add control conditions and correlation maps.

   Control conditions check whether the content of the incoming XML data is valid for the trigger. Correlation maps are used to select the process instances the trigger will operate on.

The following sample defines that the trigger will fire if the `<doMakeChoice>` element in the incoming XML data has `true` as its value. The trigger will only fire for process instances where a particular UDA has the same value as the `<newValue>` element in the incoming XML data.

```
trigger.addControlCondition("eventData.getXMLData"
                + "('/root/doMakeChoice/text()[1]') == 'true'");
trigger.addCorrelationMap(UDA_NAME, "/root/currentValue/text()[1]");
```

5. Specify the outgoing arrow to be chosen by the trigger using the method `addAction(String, String)`.

   The first parameter contains the name of the outgoing arrow that is to be chosen if the condition specified in the second parameter is true.

   In the programming sample, the trigger chooses the outgoing arrow if the value of a particular UDA is less than 10.

```
trigger.addAction(exit.getName(), "uda." + UDA_NAME + " < 10");
```

6. Map any information of the incoming XML data that you require for further processing to UDAs.

   In the programming sample, a single data mapping is defined, which maps the value of the `<newValue>` element to a UDA.

```
trigger.addDataMap(UDA_NAME, "/root/newValue/text()[1]"
```

7. Set the trigger's state to active.

```
trigger.setState(TriggerDef.STATE_ACTIVE);
```

The definition of the Event Activity Node and the trigger is now complete. At runtime, for the trigger to fire, some incoming XML data has to be sent to the trigger. This data is created by the sample program and sent to the trigger using `WFObjectFactory.processTriggerEvent()`:

```
StringBuffer bw = new StringBuffer();
bw.append("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>");
bw.append("<root>");
bw.append("<currentValue>5</currentValue>");
bw.append("<doMakeChoice>true</doMakeChoice>");
bw.append("<newValue>7</newValue>");
bw.append("</root>");
WFObjectFactory.processTriggerEvent(bw.toString(), adminSession);
```

6.12 File Listeners

File Listeners monitor files in specified directory locations. When they detect new or newly modified files, they notify the Interstage BPM file handlers, so they can perform automatic functions like starting Interstage BPM process instances or making choices on activities. File listeners are typically used for integrating Interstage BPM with other enterprise applications.

6.12.1 Using File Listeners

To use a file listener:

1. The File Listener configuration file `fileListenerConf.xml` is part of the workflow application project which you create and import from Interstage BPM Studio. It should be located in `<DMSRoot>/apps/<application ID>`.
2. If you changed your File Listener configuration file, restart your workflow application.
3. Create a process definition and add a trigger that will cause the File Listener to start a process instance. Refer to section *Using Triggers* on page 129 for instructions.
4. Create a File Listener XML file for your trigger and save it into the filelistener directory. Refer to section *Using Triggers* on page 129 for instructions.

   If the trigger is working, the XML file will disappear and a process instance will start from the process definition containing the trigger. If the trigger is not working, the XML file will move to the `filelistener\error` directory.

6.12.2 Configuring File Listeners

The File Listener configuration file `fileListenerConf.xml` is located in `<DMSRoot>/apps/<application ID>`.

The following is an example of a `fileListenerConf.xml` configuration file:

```
<FileListener>
   <Directory>
      <ScanInterval>60000</ScanInterval>
      <StabilizationPeriod>2000</StabilizationPeriod>
      <PostProcessing>
         <onSuccess>
            <Delete></Delete>
         </onSuccess>
         <onError>
            <Move>
            </Move>
         </onError>
      </PostProcessing>
   </Directory>
</FileListener>
```

The following table describes the XML tags used in `fileListenerConf.xml`:

Tag	Description
`<Directory>`	Encloses configuration parameters for the directory to be monitored for new or newly modified files.
`<Path>`	Path of the directory to be monitored. In non-SaaS mode, this can be any directory In SaaS mode, or in non-SaaS when path is not specified, path to be used is `<ServerSharedRoot>/tenants/<tenant name>/apps/<application ID>/filelistener`

Tag	Description
`<ScanInterval>`	Time interval in milliseconds after which the directory is scanned for any new incoming file.
`<StabilizationPeriod>`	Time interval in milliseconds for which the size of the file is observed (to see if it has changed) before it is processed.
`<PostProcessing><onSuccess>`	Action to be taken if the file is successfully processed. Possible actions are `<Delete>` or `<Move>`.
`<PostProcessing><onError>`	Action to be taken if the file is not successfully processed. Possible actions are `<Delete>` or `<Move>`.
`<Delete>`	Deletes the file.
`<Move>`	Moves the file to the directory specified. On success: • In non-SaaS mode, move file to any specified directory • In SaaS mode, or in non-SaaS when path is not specified, move file to `<ServerSharedRoot>/tenants/<tenant name>/apps/<application ID>/filelistener/success` On error: • In non-SaaS mode, move file to any specified directory • In SaaS mode, or in non-SaaS when path is not specified, move file to `<ServerSharedRoot>/tenants/<tenant name>/apps/<application ID>/filelistener/error`

6.13 Email Listeners

Email Listeners monitor incoming email messages containing data files intended for Interstage BPM. When they detect messages containing these files, they notify the Interstage BPM file handlers, so they can perform automatic functions like starting Interstage BPM process instances or making choices on activities. Email listeners are typically used for integrating Interstage BPM with other enterprise applications.

6.13.1 Using the Email Listener for Triggers

You must perform the following actions to use the Email Listener:

- You must configure the Email Listener.

Note: This configuration must be made by a Tenant Administrator.

To configure your Email Listener to listen for the email responses you will send, the following parameters must be set to configure the Email Listener:

- EmailListenerAutoReplyEnabled
- EmailListenerDeleteInvalidMessages
- EmailListenerEmailAddress
- EmailListenerEnabled
- EmailListenerPassword
- EmailListenerPollingInterval
- EmailListenerPOPPort
- EmailListenerPropertiesFile
- EmailListenerServerHost
- EmailListenerUserName
- EmailStyleSheetFile

For a description of these configuration parameters, see the *Interstage Business Process Manager Server Administration Guide*.

- Have a simple process definition with a Start Process or Make Choice (Activity-level) trigger with or without data mapping available and enable its trigger.

The Email Listener works just like the File Listener with one exception. The File Listener transfers any XML data file that is copied into the File Listener directory to the trigger handler for processing. The Email Listener transfers any XML data file that is contained in a specially formatted email message to the trigger handler for processing. This data file is typically used by the trigger handler to trigger an Interstage BPM event. An application access key is the special formatting required.

Note: These instructions use the System application and the default tenant.

1. Obtain the application access key for the application upon which the prerequisite process definition is defined.

Note: You must be a Tenant Administrator or Application Owner to obtain this key using the substeps given below, otherwise you must obtain this key from Tenant Administrator or Application Owner.

   a) Open **Application Settings** under the **System Administration** tab for the application upon which the prerequisite process definition is defined.

      Following the example, you would open the System application.

   b) Click **Access Key**.

      The application access key will be displayed.

   c) Without editing the key, copy it to a secure location on your network.

      This key must be added unedited to the message body of email messages to be used to activate the Email Listener. If non-administrators will be creating these special Email Listener messages, you must distribute this key to them.

2. Open your email client and create a new message.
3. Address the message to the email address specified in the tenant property **EmailListenerEmailAddress**.

   If you do not know this information, see the application administrator mentioned in Step 1.

4. Attach to the email message the Data Event file that will activate your trigger and transfer data to the process containing it. The file must be in the XML format, and it must conform to the schema defined on your trigger.

   This Data Event file must be of the same format that you would use with the File Listener.

   Only one file should be attached. If more than one file is attached, only the first file attached will be used.

   In summary, the following is needed in email messages that will activate the Email Listener:

   - Application Access Key (see Step 1)
   - Email Listener Email Address (see Step 3)
   - XML Data Event File (see top part of Step 4, this step)

5. Send the Email message.

   Interstage BPM will process the message and send an email response back to you.

   That message will notify you of the pass or fail status of the Email Listener.

6. Check the results of sending the Email Listener message.

   This results obtained will depend on the type of trigger to be activated by the message and the success or failure of the message to activate the trigger.

   If the message activates a Start Process trigger, a new process instance will be started from the process definition defined in the trigger. Interstage BPM will also send you an email response indicating the successful operation.

   If the message activates a Make Choice trigger, the Activity defined on the trigger will be completed along the path of the choice specified in the Data Event file. Interstage BPM will also send you an email response indicating the successful operation.

   If data was mapped in the trigger, it will be added to the process according to the trigger definition.

   If the message fails to activate the trigger specified in the Data Event file, Interstage BPM will send you an email response indicating the unsuccessful operation.

   If you send an email message containing an invalid or already expired application access key, Interstage BPM will discard the message and send you a response indicating that you are using an invalid key.

6.14 JMS Listeners

JMS Listeners monitor incoming JMS messages containing data intended for Interstage BPM. When they detect messages containing this data, they notify the Interstage BPM trigger handler, so it can perform automatic functions like starting Interstage BPM process instances or making choices on activities. JMS listeners are typically used for integrating Interstage BPM with other enterprise applications.

6.14.1 Using the JMS Listener for Triggers

You must have the following to use the JMS Listener:

- The knowledge to create a simple JMS client application
- A simple process definition with a Start Process or Make Choice (Activity-level) trigger with or without data mapping available and enable its trigger.

The JMS Listener works just like the File Listener with one exception. The File Listener transfers any XML data file that is copied into the File Listener directory to the trigger handler for processing. The JMS Listener transfers XML data from a JMS message to the trigger handler for processing. This data is typically used by the trigger handler to trigger an Interstage BPM event. An application access key must be included in the JMS message as indicated below.

Note: The instructions in Step 1 use the System application and the default tenant.

1. Obtain the the application access key for the application upon which the prerequisite process definition is defined.

Note: You must be a Tenant Administrator or Application Owner to obtain this key using the substeps given below, otherwise you must obtain this key from Tenant Administrator or Application Owner.

   a) Open **Application Settings** under the **System Administration** tab for the application upon which the prerequisite process definition is defined.

   Following the example, you would open the System application.

   b) Click **Access Key**.

   The application access key will be displayed.

   c) Without editing the key, copy it to a secure location on your network.

   This key must be added unedited to the JMS message to be used to activate the JMS Listener. If non-administrators will be creating these special JMS messages, you must distribute this key to them.

2. Write theData Event STRING that will activate your trigger and transfer data to the process containing it. The STRING must be in the XML format, and it must conform to the schema defined on your trigger.
3. Determine the name of the Interstage BPM Server to which you want to post the JMS message.
4. When an Interstage BPM Server is deployed, it comes with the JMS Topics like CommandTopic and ResponseTopic. Interstage BPM server will listen on to the CommndTopic for any incoming JMS message to process, and once the processing is done, the Interstage BPM Server will post a response message to the ResponseTopic. The user application can listen to the ResponseTopic to get the response back from the Interstage BPM Server. Create a simple JMS client application to post a text message onto the CommandTopic. This will be your JMS Message. You must set the xmlString variable to the XML Data Event STRING (see Step 2), the APPLICATION_ACCESSKEY property to the Application Access Key STRING (see Step 1), and the SERVER_ID property to the name of the Interstage BPM Server to which you want to post the JMS message expressed as a STRING (see Step 3). The following is an example of how you would set these properties in the JMS message:

```
...
TextMessage msg = pubSession.createTextMessage();

String xmlString = [XML Data Event STRING];
String serverName = [name of Interstage BPM Server STRING];
String applicationAccessKey = [the Application Access Key STRING];

msg.setText(xmlString);
msg.setStringProperty("SERVER_ID", serverName);
msg.setStringProperty("APPLICATION_ACCESSKEY", applicationAccessKey);
...
```

5. Post the JMS message.

Interstage BPM server will process the JMS message and send a JMS response to ResponseTopic about the pass or fail status of the JMS Listener. The response will be of type Text Message or Object Message. When there is an exception accessing the application, a Text Message will be posted back explaining the failure and if the XML Data Event STRING was successfully processed then, the com.fujitsu.mode.workflow.TriggerResult Object will be posted back in the form of Object Message.

6. Check the results of posting the JMS message.

   This results obtained will depend on the type of trigger to be activated by the message and the success or failure of the message to activate the trigger.

   If the message activates a Start Process trigger, a new process instance will be started from the process definition defined in the trigger. Interstage BPM will also send you a response indicating the successful operation.

   If the message activates a Make Choice trigger, the Activity defined on the trigger will be completed along the path of the choice specified in the Data Event file. Interstage BPM will also send you a response indicating the successful operation.

   If data was mapped in the trigger, it will be added to the process according to the trigger definition.

   If the message fails to activate the trigger specified in the Data Event file, Interstage BPM will send you a response, described above, indicating the unsuccessful operation.

   If you post a message containing an invalid or already expired application access key, Interstage BPM will discard the message and send you a response indicating that you are using an invalid key.

6.15 Using Agents

Agents in Interstage BPM are set up to run automatically and act asynchronously on your behalf. You can use Agents to access systems that are external to Interstage BPM. The external systems to which you can connect might be legacy systems or Web Services, both inside and outside of company firewalls. Using Agents, you can incorporate these external services into your Interstage BPM process instances.

Note:
- To include external libraries, so they can be used in Interstage BPM Agents, copy your external classes or JARs to a directory as specified in *Integrating Interstage BPM with External Applications* on page 88.
- If you are using Agents in a cluster installation, configure your Agents on each cluster node. The Agent's configuration must be identical on all cluster nodes.

6.15.1 Agents Overview

Agents are application-specific, and are configured using an XML file called `agentsConfig.xml`. This file is part of the workflow application project when you import it from Interstage BPM Studio, and can be found in `<DMSRoot>/apps/<application id>`.

Once an Agent is configured in Interstage BPM by adding it to `agentsConfig.xml`, it can be run as an activity in an Interstage BPM process instance by assigning it to the Activity Node. The agent is then run in place of the activity. The work item is actually assigned to a system user that has the same name as the agent.

Configuration File agentsConfig.xml

The following code listing is from `agentsConfig.xml`:

```
<ActionAgentList>
   <ActionAgent>
      <Name>@TestFrameAgent</Name>
      <Description>Use for Test, return first choice</Description>
      <RetryInterval>20</RetryInterval>
      <EscalationInterval>1</EscalationInterval>
      <ClassName>com.fujitsu.iflowqa.testframe.TestFrameAgent
      </ClassName>
      <ConfigFile></ConfigFile>
   </ActionAgent>
</ActionAgentList>
```

The following table describes the XML tags used in `agentsConfig.xml`:

Tag	Description
`<ActionAgentList>`	Contains a set of Agents.
`<ActionAgent>`	Contains a definition for a single Agent. Each Agent must have its own definition that is contained under this tag.
`<Name>`	Name of the Agent. The name must begin with "`@`". This designates it as an Agent. The name has to be specified as the assignee of an Activity Node, making the activity an "Agent" activity. As such, it no longer behaves as a "normal" Interstage BPM activity but as an Agent.
`<Description>`	Short description of the Agent. It typically explains form and function.
`<RetryInterval>`	If the Agent fails to call its external service, the Agent retries to call it after a specified time. `<RetryInterval>` specifies the time interval in seconds between attempts.
`<EscalationInterval>`	Specifies the number of failures after which the System Administrator is notified via email. An email is sent after each escalation interval. For example, with a value of 1 the System Administrator is notified each time the Agent fails. The email is sent to the address specified in the `ServerEmailAddress` parameter of the Interstage BPM Server.
`<ClassName>`	Name of the Java class associated with the Agent. This class is the functional part of the Agent.

Tag	Description
`<ClassPath>`	CLASSPATH of the Java class associated with the Agent. In SaaS mode, this value is ignored.
`<ConfigFile>`	It is the configuration file name

Agent Class

The class specified in the `<ClassName>` tag in `agentsConfig.xml` is the Agent's access to its external service. Henceforth, it is called the Agent Class. The Agent Class must implement the following interface:

```
package com.fujitsu.iflow.server.intf;
public interface ActionAgentInvoke
   {
      public String invokeService(
         ServerEnactmentContext sec, String configFile) throws Exception;
}
```

The Agent Class calls the external service using `invokeService()`, receives data from the service, and returns a STRING to Interstage BPM that indicates the action of the service.

There are three possible results of Agent Class execution. These are:

- Return of a NULL STRING or an empty STRING: If the Agent Class returns a NULL STRING or an empty STRING, the Agent repeats its attempt to execute its service. The time interval is specified in the `<RetryInterval>` tag. Each time, the agent attempts execution the number of times specified in the `<EscalationInterval>` tag, the Agent sends an email message to the address specified in the `ServerEmailAddress` parameter of the Interstage BPM Server
- Return of any other STRING: If the Agent Class returns any other STRING, the Agent evaluates it to see if it matches the name of one of the arrows outgoing from the Agent activity. If there is a match, the process instance continues with the matching arrow. If the return STRING does not match an arrow, the Agent throws a cannot-find-arrow exception, and the process instance goes into error state.
- Agent Class throws an exception: The Agent sends the exception to Interstage BPM, and the process instance goes into error state.

AgentSimulator Example

A sample process definition using an Agent is provided in the `<Interstage BPM Server Installation Directory>/client/samples/examples/sources` directory. `AgentSimulator.xpdl` is a process definition that models a simple bank loan approval process. An Agent called `AgentSimulator` is used to simulate a loan decision maker.

For instructions on running the sample, refer to the comments in the source file `AgentSimulator.java`.

6.15.2 Configuring FTP Agents

FTP Agents automatically transfer files attached to process instances that contain them. Like all Agents, FTP Agents take the place of Interstage BPM activities. Process instances started from a process definition containing an FTP Agent activity automatically transfer the files that arge attached to them. They transfer the attached file(s) to the FTP Server of any machine to which the Interstage BPM Server machine can connect.

An FTP Agent, like all Agents, is configured with `agentsConfig.xml`. This configuration file contains an `<ActionAgent>` section with default settings for the FTP Agent. Usually, you don't need to change the FTP Agent settings in this file.

The FTP Agent uses Interstage BPM's `ServiceAgent` class. It also uses a special FTP Agent configuration file called `ftp.xml`. This configuration file defines FTP settings, e.g. the address of the FTP host. Check this configuration file and adapt it to your needs.

Configuration File ftp.xml

The FTP Agent configuration file `ftp.xml` is located in in `<DMSRoot>/apps/<application id>`.

The following is a sample `ftp.xml` file that configures an FTP Agent:

```
<Services>
   <Service>
      <ServiceType>FTP</ServiceType>
      <ServiceStatusUDA>AgentServiceStatus</ServiceStatusUDA>
      <ServiceResultUDA>AgentServiceResult</ServiceResultUDA>
      <ServiceSpecificInfo>
         <FTPHost><HOSTNAME or IP Address></FTPHost>
         <FTPPort></FTPPort>
         <FTPUser>anonymous</FTPUser>
         <FTPPassword></FTPPassword>
         <FTPType>ASCII</FTPType>
         <FTPAppend>FALSE</FTPAppend>
         <FTPDirectory>%%REMOTE_FTP_DIRECTORY%%
            </FTPDirectory>
         <FTPFileNames>%%REMOTE_FTP_FILES%%</FTPFileNames>
         <Documents>%%OUTGOING_FILES%%</Documents>
      </ServiceSpecificInfo>
   </Service>
</Services>
```

Some of the tags in this file are used by all agents using the `ServiceAgent` class. Some of them are specific to FTP Agents.

The following table describes the XML tags used in `ftp.xml`:

Tag	Description
`<ServiceType>`	For the FTP Agent, the service type is always FTP. If there is no service type specified or the service type is invalid, the process instance goes into error state. This tag is used by all agents using the `ServiceAgent` class.
`<ServiceStatusUDA>`	Name of the User Defined Attribute (UDA) that stores the status of the FTP Agent activity. Possible values are `Done` or `Failed`. Removing this tag has no effect on the FTP Agent; however, the user cannot see the status of the FTP Agent activity.

Tag	Description
`<ServiceResultUDA>`	Name of the UDA that stores the error message, if the FTP Agent throws an exception. Removing this tag has no effect on the FTP Agent; however, if the FTP Agent fails, the user cannot see the error that occurred.
`<FTPHost>`	Host name or IP address of the machine that receives the transferred file. The default value is 127.0.0.1, i.e. the local host. If there is no FTP host specified or the FTP host is invalid, files are transferred to the local FTP server, if available. This tag is specific to the FTP Agent.
`<FTPPort>`	Port used by the FTP server. If no port is specified, the default FTP port 21 is used. This tag is specific to the FTP Agent.
`<FTPUser>`	User name of the FTP user that transfers the file. If the user name is invalid, file transfer fails and the process instance goes into error state. This tag is specific to the FTP Agent.
`<FTPPassword>`	Password of the FTP user that transfers the file. This tag is specific to the FTP Agent.
`<FTPType>`	Type of files to be transferred. Allowed values are `ASCII` or `BINARY`. Default value is `BINARY`. Use `ASCII` if you are transferring text files. Use `BINARY` if you are transferring binary files. If any other value than `ASCII` or `BINARY` is specified, the FTP Agent throws an exception. This tag is specific to the FTP Agent.
`<FTPAppend>`	Specifies whether a file to be transferred is appended to a remote file. Allowed values are `TRUE` or `FALSE`. Default value is `FALSE`. Use `TRUE` if you want to append the contents of the local file to an already existing remote file. Use `FALSE` if you want to overwrite an already existing remote file. If any other value than `TRUE` or `FALSE` is specified, the FTP Agent throws an exception. This tag is specific to the FTP Agent.

Tag	Description
`<FTPDirectory>`	Name of the User Defined Attribute (UDA) that specifies the directory to which files are transferred. A path relative to the FTP server is specified. For example: if `/iflow` is the value of the `REMOTE_FTP_DIRECTORY` UDA and you are working on a Windows system, the file is transferred to `c:\Inetpub\ftproot\iflow`. If an invalid directory is specified, the process instance goes into error state. This tag is specific to the FTP Agent.
`<FTPFileNames>`	Name of the UDA that specifies the file names to be used for the remote files. This parameter is used to rename files while transferring them. For example, if `LocalFile.txt` is attached to the process instance and `REMOTE_FTP_FILES` has `RemoteFile.txt` as its value, `LocalFile.txt` is transferred to the FTP directory and renamed to `RemoteFile.txt`.`REMOTE_FTP_FILES` can contain several file names separated by semicolon (;). If `REMOTE_FTP_FILES` contains no file names, the original file names are used.
`<Documents>`	Name of the UDA that specifies the files that are transferred to the FTP directory. `OUTGOING_FILES` can contain several file names separated by semicolon (;). If `OUTGOING_FILES` contains no file names, all files attached to the process instance are transferred.

Note: Files that have a semicolon (;) in their names cannot be transferred to an FTP directory. Semicolons are considered as file name delimiters and are therefore not allowed within names of files that are to be transferred.

6.15.3 Using FTP Agents

To use an FTP Agent in your process definition:

1. Create an Activity Node that represents the FTP agent.
2. Assign "`@FTP`" to the Activity Node.
3. Add all User Defined Attributes (UDAs) that correspond to parameters in the FTP Agent configuration file `ftp.xml` to your process definition.

   These are the UDAs you might need to add:

   - `AgentServiceStatus`
   - `AgentServiceResult`
   - `REMOTE_FTP_DIRECTORY`

- `REMOTE_FTP_FILES`
- `OUTGOING_FILES`

  If you don't want to add `REMOTE_FTP_FILES` or `OUTGOING_FILES`, remove the contents of the corresponding tag from the `ftp.xml` file. Example: You want to keep the original names of the files to be transferred. Therefore, you don't add the `REMOTE_FTP_FILES` UDA to your process definition. In this case, you need to remove the contents of the `<FTPFileNames>` tag from your `ftp.xml` file: `<FTPFileNames></FTPFileNames>`.

  If you don't add `REMOTE_FTP_FILES` or `OUTGOING_FILES` while the corresponding parameter is specified in the `ftp.xml` file, the process instance goes into error state once the FTP Agent Activity becomes active.

  Refer to section *Configuring FTP Agents* on page 142 for more information about the `ftp.xml` configuration file.

4. Add a `Done` arrow that originates from the FTP Agent activity to activate the next activity.

The FTP Agent either returns `Done` or `null`:

- If it returns `Done`, the process instance continues with the `Done` arrow.
- If it returns `null`, the process instance goes into error state. The `AgentServiceStatus` UDA indicates the error state, and the `AgentServiceResult` UDA stores the error message that has been issued.

In the following cases, the process instance goes into error state, but `AgentServiceStatus` and `AgentServiceResult` UDAs are NOT updated with error information:

- The FTP Agent activity has no outgoing `Done` arrow.
- Some data in `agentsConfig.xml` is incorrect or missing.

To see the error reason, the user needs to check the history of the process instance.

6.15.4 Configuring HTTP Agents

HTTP Agents are used to send data to external locations on the Internet and receive data from the Internet. An Agent sends data to the URL specified in its configuration and receives response data from that URL. It sends the value of the User Defined Attributes (UDAs) specified in the `<HTTPRequestUDA>` tag to the URL specified in the `<HTTPBaseURL>` tag and assigns the return data as the value of the UDA specified in the `<HTTPReponseUDA>` tag.

An HTTP Agent, like all Agents, is configured in the `agentsConfig.xml` file. This configuration file contains an `<ActionAgent>` section with default settings for the HTTP Agent. Usually, you don't need to change the HTTP Agent settings in this file.

To configure an HTTP Agent in Interstage BPM:

Create an HTTP Agent configuration file and place it in `<DMSRoot>/apps/<application id>`.

Configuration File HTTPagent.xml

The following is a sample HTTP Agent configuration file:

```
<HTTPAgent>
   <HTTPBaseURL method="POST"
      followRedirects="true">{{Field HTTP_URL}}</HTTPBaseURL>
   <HTTPHeaderParams name="Content-Type">text/xml;
      charset=UTF-8</HTTPHeaderParams>
   <HTTPHeaderParams name="User-Agent">I-BPM HTTP Agent
   </HTTPHeaderParams>
   <HTTPHeaderParams name="accept-charset">UTF-8
   </HTTPHeaderParams>
   <QueryParams name= "ParamName">Param value</QueryParams>
   <HTTPRequestUDA>HTTP_REQUEST</HTTPRequestUDA>
   <HTTPResponseUDA>HTTP_RESPONSE</HTTPResponseUDA>
   <HTTPAgentStatusUDA>HTTP_STATUS</HTTPAgentStatusUDA>
</HTTPAgent>
```

The following table describes the XML tags used in `HTTPagent.xml`:

Tag	Description
`<HTTPBaseURL>`	URL end point. The value for the `<HTTPBaseURL>` tag specifies the external service to be called by the HTTP Agent, e.g. a Servlet. This value can also be defined to come from a process instance UDA as QuickForm expression, such as `{{Field HTTP_URL}}`. Attributes: `method="POST"` or `"GET"`. Defines the request method to be used for the HTTP request. If you do not specify the `method` attribute, the process instance will go into the error state. If you specify an attribute value other than `"POST"` or `"GET"`, the HTTP agent will fail. `followRedirects="true"`. Since the default value for `followRedirects` is `false`, you must specify this attribute.

Tag	Description
`<HTTPHeaderParams>`	Sets the HTTP header properties like the encoding style, the content type, etc. HTTP headers are used to define a variety of web object properties, for example, the properties of request and response objects for the current HTTP session. Attributes: `name="Content-type"` or `"User-Agent"`. `"Content-type"` defines the type of requested content and takes the following value: `value="text/xml", charset=UTF-8`. This value can also be defined to come from process instance UDAs as QuickForm expression such as `{{Field contentType}}`. `"User-Agent"` contains information about the client (user agent) from where the request originates. Note that currently there is no restriction for the value that can be specified in the `name` attribute of the `<HTTPHeaderParams>` tag. Any valid values for this attribute are valid HTTP headers.

`<QueryParams>`	This tag is required in case the `"GET"` method is used with the HTTP Agent. The value of this tag is passed to the external service as Query String. Note that the value of the UDA specified in the `<HTTPRequestUDA>` tag is not used when the `"GET"` method is used.
	Correct usage of this tag is as follows:
	`<QueryParams name= "ParamName">Param value </QueryParams>`
	This will result in the following URL specification during HTTP Agent execution:
	`URL?ParamName=Param Value`
	You can specify any valid string for the name attribute of the `<QueryParams>` tag.

`<HTTPRequestUDA>`	Value of the UDA specified in this tag is part of the query string for the HTTP request.
`<HTTPAgentStatusUDA>`	UDA where the HTTP service status will be stored.
`<HTTPResponseUDA>`	UDA where the HTTP response will be stored.

6.15.5 Using HTTP Agents

To use an HTTP Agent in your process definition:

1. Create an Activity Node that represents the HTTP agent.
2. Assign `@HTTPAgent` to the Activity Node.
3. Add all the User Defined Attributes (UDAs) that correspond to parameters in the `HTTPAgent.xml` file to your process definition.

   These are the UDAs you might need to add:

   - `HTTP_URL`
   - `HTTP_REQUEST`
   - `HTTP_RESPONSE`
   - `HTTP_STATUS`
4. Add a `Done` and a `Failed` arrow that originate at the HTTP Agent activity for activating the next node.

   The HTTP Agent either returns `Done` or `Failed`:

   - If it returns `Done`, the process instance continues with the `Done` arrow.
   - If it returns `Failed`, the process instance continues with the `Failed` arrow.

HTTP Agent Example

A pre-configured HTTP Example is provided for your use in the `<Interstage BPM Server Installation Directory>/client/samples/examples/sources/HTTPAgentExample` directory. This example provides an easy-to-implement HTTP Agent. The example agent demonstrates the complete functionality of an HTTP Agent.

This example includes a `.war` file that needs to be deployed on the application server where the Interstage BPM server has been installed. Proceed with the implementation of the HTTP Agent as described in the `HTTPAgentExampleInstructions.txt` file located in the `HTTPAgentExample` directory.

Below, you find some additional explanations:

- The configuration files `agentsConfig.xml` and `HTTPAgent.xml` include the configuration of the HTTP Agent used in this example. The HTTP Agent settings are contained in the `HTTPAgent.xml` file. The `agentsConfig.xml` file specifies the `HTTPAgent.xml` file as an agent.
- You must restart the Interstage BPM Services to have your newly configured HTTP Agent take effect.
- The imported template `HTTP Agent Example V1.xpdl` contains the HTTP Agent. An agent is assigned to an Activity Node in the same manner that a User Group is assigned to an Activity Node. However, the special designation for the agent name is the `@` sign. In this case the agent name is `@HTTPAgent`.

6.16 Using Timers

Timers trigger certain actions when they expire. These timers can be Activity-Node-Level timers that start running when the Activity Node is activated or Process-Level timers that start running whenever a new process instance is created from the process definition containing the timer.

Timers are controlled by time and action parameters, specified as User Defined Attributes and Javascript, and set through Java Actions or forms. The expiration or firing time of a timer can be an absolute, specific time or relative to another event (such as the start of the activity or process instance).

The timer can also be set to trigger actions periodically. Timers can also perform a variety of actions, including escalation (assigning the activity to an additional list of users) or sending emails.

Execution and expiration of timers can also be controlled for the actions such as cancel, reschedule, expire or reset active timres while the timer is running.

Refer to sections *Controlling Execution of Timers* on page 152 and *Defining a Timer* on page 150.

Timers can be set with the following nodes:

- Activity Node
- Delay Node
- Voting Activity Node
- Compound Activity Node

Refer to section *Timers* on page 32 for an overview of available timer types.

6.16.1 Defining a Timer

In the following example a timer is added to a Delay Node, which waits 200 milliseconds and then sends an e-mail.

To add a timer to a node:

1. Define a Java Action and the User Defined Attributes (UDAs) for a `TimerAction` object.

```
JavaActionSet timerJASet = WFObjectFactory.getJavaActionSet();
timerJASet.setActionSetDescription("Timer controlled execution"
   + " of actions");
timerJASet.setActionSetName("TimerActions");
JavaAction[] timerJA = timerJASet.createJavaActions(1);

timerJA[0].setActionDescription("Sends an email ");
timerJA[0].setActionName("EmailSend");
timerJA[0].setClassName(CLASS_NAME_JAVA_ACTION);
timerJA[0].setMethodName("sendEmail(String,String,String,String,"
   + "String,String,ServerEnactmentContext)");
timerJA[0].setArgumentsUDANames("<E>uda.to</E><E>uda.from</E>"
   + "<E>uda.cc</E><E>uda.bcc</E><E>uda.subject</E>"
   + "<E>uda.body</E><E>sec</E>");
timerJASet.setJavaActions(timerJA);

plan.addDataItemRef("Timer_JavaActionSet",
DataItemRef.TYPE_STRING,
timerJASet.toString());
```

2. Define a `TimerAction` object. Use `getTimerDef()` from the `WFObjectFactory` class to create a timer of a specified type.

```
TimerAction timerAction = WFObjectFactory.getTimerAction();
timerAction.setJavaActionSet("Timer_JavaActionSet");
```

3. In this step, define the type of the used timer with `setType(timerType)`. You find the possible values of the `timerType` constant in the `TimerDef` interface:
   - `RELATIVE`: Defines a Relative timer.
   - `ABSOLUTE`: Defines an Absolute timer.
   - `PERIODIC`: Defines a Periodic timer.
   - `BUSINESSPERIODIC`: Defines a Business Periodic timer.

- `BUSINESSRELATIVE`: Defines a Business Relative timer.

```
TimerDef timer = WFObjectFactory.getTimerDef();
timer.setName("Send mail timer");
timer.setDescription("This timer waits 2000 miliseconds and then"
   + " sends an email.");
timer.setType(TimerDef.RELATIVE);
timer.setTime("TimeOfExpiration");
timer.setTimerTimeExpression("DateAdd(Packages.java.util.Date(), 4,
\"hh\").getTime();");
timer.addTimerAction(timerAction);
plan.addDataItemRef("TimeOfExpiration",
   DataItemRef.TYPE_LONG, "2000");
```

4. Define a `TimerDef` object, i.e. add the `TimerAction` object to `TimerDef` object.
5. Set the name of the UDA which contains the time of expiration.
6. You can also set a Javascript that gets evaluated to calculate the expiry date of a timer.

Note: You can set the Javascript only for Absolute timers, Relative timers, and Periodic timers.

- In case of an Absolute timer, the result of script evaluation is considered as the expiry time. However, if script evaluation does not result in a `date-time` value in `Long` format, then the UDA value is considered as the expiry time. That is, expiry time = script evaluation result or UDA value.
- In case of relative timer, the result of script evaluation is considered as the reference time and time specified in UDA value is considered as relative time from the reference time. That is, expiry time = script evaluation result + UDA value.

This example sets the expiry time as four hours from the time the script is executed.

```
timerDef.setTimerTimeExpression("DateAdd(Packages.java.util.Date(), 4,
 \"hh\").getTime();");
```

This example sets expiry time as two days from `RequestDate` in XML UDA.

```
timerDef.setTimerTimeExpression("sec.getProcessXMLAttributeElementValue(\"xmlUda\",
 "//Request/RequestDate/text()\");");
 timerDef.setTime("timerUda"); // timerUda holds the value for 2 days
in long format
```

`TimerAction` objects and `TimerDef` objects are distinct from one another, and they do not need to be created in a specific order. However, to use a `TimerAction` object, it must be associated with the `TimerDef` object by using `TimerDef.addTimerAction(TimerAction action)`.

7. Add the `TimerDef` object to the node.

```
delayNode.addTimer(timer);
```

6.16.2 Adding a Due Date

You can add due date timer at plan or node level by specifying the time by when the process or node should complete. The following sample code illustrates adding and retrieving DueDate on the Process Definition.

```
 //Adding DueDate on Plan:
plan.startEdit();
TimerDef timerDef = WFObjectFactory.getTimerDef();
timerDef.setName("DueDateTimer");
timerDef.setType(TimerDef.RELATIVE);
plan.addDataItemRef("DueDateExpTime",DataItemRef.TYPE_STRING, "300000");
timerDef.setTime("DueDateExpTime");

//To add DueDate on Plan
plan.setDueDateDef(timerDef);
plan.commitEdit();

//To get Due date on Plan
TimerDef dueDateDef = plan.getDueDateDef(timerDef);
ProcessInstance pi = plan.createProcessInstance();

//To retrieve DueDate of Process
Date dueDate = pi.getDueDate();
```

6.16.3 Controlling Execution of Timers

Interstage BPM allows you to control the execution of active timers. You can cancel an active timer, re-schedule it to a new absolute time, reset it to recalculate the timer expiry time, or expire it immediately. By controlling the execution of a timer, you can either change the actual expiry of a timer or stop it from being executed.

Note: Timer control is only accessible to the process owners and the administrators.

You can control the execution of an active timer in the following ways:

- **Cancel a timer:** You can cancel an active timer. The cancelled timer will not execute even after the expiry time is reached. However, it can be reactivated by re-scheduling or resetting it.

Note: You can reshedule and reset a cancelled timer.

- **Re-schedule a timer:** You can re-schedule an active or cancelled timer by specifying an absolute time at which they should expire.

Note: Rescheduling any timer can only be achieved by mentioning an absolute time.

The following code samples show usage of `reschedule` API

```
Calendar cal = Calendar.getInstance();
cal.clear();
cal.set(2009, 2, 3, 13, 30, 00);
timerInstance.reschedule( cal.getTimeInMillis() );
```

Note: Assignee of an activity can also re-schedule a DueDate timer along with process owners and the administrators.

- **Reset a timer:**When you reset an active or cancelled timer, the system gets the timer value again from the UDA or recalculates it from the script. This is useful when the UDA is changed after the timer is activated or cancelled. Once reset, the timer will expire at the new expiry time that is defined.
- **Expire a timer:** Active timers can be forcefully fired immediately, irrespective of their scheduled expiry time.

Note: A DueDate Timer cannot be forcefully fired.

Note: If a timer is rescheduled, reset, or expired, it is implicitly canceled and a new timer instance is started.

6.16.4 Timer Instance History

The timer instance history information is stored for audit purposes. Each time there is a change to a timer instance, the trail of the change is stored and can be retrieved to be viewed at a later time.

The following sample code shows how to retrieve the history of a timer.

```
TimerInstance[]  tInsts = processInstance.getTimerInstances(timerDefId);
 for (int i = 0; i < tInsts.length; i++) {
   TimerInstance tInstance = tInsts[i];
   System.out.println(tInstance.getId() + "\t"
       + tInstance.getName() + "\t"
       + tInstance.getDescription() + "\t"
       + tInstance.getTimerInstanceId() + "\t"
       + new Date(tInstance.getTimerExpirationTime()) + "\t"
       + tInstance.getState() + "\t" + tInstance.getActor();
  }
```

6.16.5 Using Business Calendars

Interstage BPM allows you to create timers that trigger certain events associated with the timer upon its expiration. The Business Calendars feature allows you to create business timers. A business timer is a special type of timer that will only "count" business hours and days and expires only during business hours. Business hours and days are specified in a Business Calendar.

The Business Periodic timers and the Business Relative timers are used with Business Calendars. Since Business Calendars are used in combination with timers, they can be used with the following types of nodes:

- Activity Node
- Delay Node
- Voting Activity Node

Refer to section *Timers* on page 32 for an overview of available timer types.

You can find the complete programming code of the examples presented in this section in the sample file `TriggerTimerSample.java`.

Configuring Business Calendars

Interstage BPM allows you to use many different business calendars. If you haven’t defined a business calendar for the process definition you are using, Interstage BPM uses the default business calendar installed with Interstage BPM.

You can create and use your own custom business calendars or modify the default business calendar to meet your needs. You may create as many Business Calendars as necessary to meet the needs of your organization and use a different Business Calendar for every process definition or process instance. The name of the calendar corresponds to the name of the calendar file. The calendar file is given a `.cal` file extension, so it can be recognized as a Business Calendar.

A default calendar (named `Default.cal`) is provided at installation time. If no other calendar is configured for use with Interstage BPM, you can still add a business timer to your Interstage BPM process instances, and the default calendar will specify the business days and hours. The default calendar also provides an example of a fully functional Business Calendar.

For instructions in creating and using your own custom business calendars or modifying the default business calendar, refer to the *Interstage Business Process Manager Server Administration Guide*.

Assigning Business Calendars

You can assign a particular calendar to a process definition or process instance by assigning its name (calendar file name without the `.cal` extension) to the `__businessCalendar` User Defined Attribute (UDA). This UDA is used exclusively for Business Timers and Due Dates.

For example, say you had a calendar that you wanted to use for process instances run in your German subsidiary and you called it `German.cal`. You want a certain activity to be assigned to one of your employees, and if this employee is not available after a certain period of time, you want the activity to be assigned to another employee. The time when the activity is reassigned depends on your definition of the Business Calendar and the timer you use.

To assign a business calendar to a process definition:

1. Add a UDA of type STRING to your process definitions and assign the name of the calendar file as its value.

   For example, you could add a STRING type UDA to your German subsidiary division process definitions and assign it a value of `German`:

   ```
   plan.addDataItemRef("__businessCalendar", DataItemRef.TYPE_STRING,
   "German");
   ```

2. Add a UDA with a given name, type, and initial value to the process definition. The value of the UDA is to be specified as STRING and represents the start time for the timer.

   In the example, the business time is set relative to the current time of the day by two minutes:

   ```
   plan.addDataItemRef("__udaStartTime", DataItemRef.TYPE_STRING,
   "BT(00:02:00)");
   ```

   For a list of time and day codes that you can use for the business time, refer to section *Time and Day Codes for Timers* on page 155.

3. Add a node to which the business periodic or business relative timer is to be assigned, for example, an Activity Node.

   ```
   Node orderActivity = plan.addNode("Order",Node.TYPE_ACTIVITY);
   ```

4. Create the timer by specifying its name, a description, the start time defined by the `__udaStartTime` UDA, and by defining whether it is to be a business relative or periodic timer.

In the example, a business relative timer is used:

```
TimerDef busCalTimer = WFObjectFactory.getTimerDef();
busCalTimer.setName("busCalTimer");
busCalTimer.setDescription("Timer to show the business calendar 
functionality");
busCalTimer.setTime("__udaStartTime");
busCalTimer.setType(TimerDef.BUSINESSRELATIVE);
```

5. Add the timer to the Activity Node.

```
TimerAction busCalTimerAction = WFObjectFactory.getTimerAction();
// Set the name of the UDA that contains the JavaActionSet to be
// executed when the timer expires.
busCalTimerAction.setJavaActionSet("BusCalActionSet");
// Add a TimerAction that specifies a JavaActionSet to this timer
// definition. The defined JavaActionSet of this action is executed
// when this timer expires.
busCalTimer.addTimerAction(busCalTimerAction);
//Add timer to the activity node
orderActivity.addTimer(busCalTimer);
```

You can also assign a business calendar to a particular Business Timer or Due Date. The Business Timer or Due Date is then calculated based on this business calendar. Use the following expression:

`UC(<business calendar>);`

For `<business calendar>`, specify the name of your business calendar without the .cal extension.

In the following example, a business calendar called `California.cal` is used for the calculation of the Business Timer:

```
plan.addDataItemRef("__udaStartTime", DataItemRef.TYPE_STRING,
     "UC(California);BT(00:02:00)");
. . .
TimerDef busCalTimer = WFObjectFactory.getTimerDef();
. . .
busCalTimer.setTime("__udaStartTime");
busCalTimer.setType(TimerDef.BUSINESSRELATIVE);
. . .
```

You can change calendars while a process instance is running using the `setProcessAttribute` JavaScript function and the `__businessCalendar` UDA. For example to change the business calendar to `England.cal`, you would use the following script:

```
sec.setProcessAttribute("__businessCalendar","England");
```

6.16.6 Time and Day Codes for Timers

When defining a timer, you can use time and/or day codes to specify the time. In the following example, a time code is used to set the business time relative to the current time. The time code is marked bold.

```
plan.addDataItemRef("__udaStartTime", DataItemRef.TYPE_STRING,
"BT(00:02:00)");
```

These are the time and day codes that you can use.

Time Codes

Code	Meaning	Example
`AT`	Sets the absolute time of the day.	`AT(16:30:00)`: 4:30pm on that day.
`CT`	Sets the business time relative to the closing time of the day. Allowed values: `00` or negative hours. Typically you will use negative hours with closing time in order to calculate a relative time before closing time.	`CT(00)`: Closing time. `CT(-02:00:00)`: 2 hrs before the closing time.
`OT`	Sets the business time relative to the opening time of the day. Allowed values: `00` or positive hours.	`OT(00)`: At the opening time. `OT(02:00:00)`: 2 hrs after the opening time.
`BT`	Sets the business time relative to the current time of the day.	`BT(04:30:00)`: After 4 & 1/2 business hours from the current time. `BT(-02:00)`: 2 business hours earlier. `BT(00)`: Use this to search forward to the next business time, without changing the time if the current time is not a business time. You may need this if different business days have different hours. `BT(-00)`: Use this to search backward to the last previous business time. Has no effect if the current time is already during business time.

Day Codes

Code	Meaning	Example
`BD`	Sets a business day.	`BD(4)`: Four business days from today. `BD(0)`: Same day if it is a business day, else the next day. `BD(-0)`: Same day if it is a business day, else the previous day.
`RD`	Sets a relative day from the current day.	`RD(7)`: After one week. `RD(-1)`: One day earlier.
`WD`	Sets a day of the week. Allowed values: `1` to `7`.	`WD(1)`: Sunday of that week. `WD(7)`: Saturday of that week.
`WN`	Sets the next weekday after today. Allowed values: `1` to `7`.	`WN(1)`: The next Sunday after today. `WN(7)`: The next Saturday after today.

Code	Meaning	Example
`RM`	Sets a relative month in the future. If the month does not have enough days to be the same day, the day will be the last day of the month.	`RM(3)`: After 3 months.
`DM`	Sets an exact day of the month. Allowed values: Any number except `0`.	`DM(1)`: The first day of the month. `DM(-1)`: The last day of the month.
`BM`	Sets an exact business day of the month. Allowed values: Any number except `0`.	`BM(1)`: The first business day of the month. `BM(-1)`: The last business day of the month.
`DY`	Sets a day of the year. Allowed values: Any number except `0`.	`DY(1)`: The first day of the year. `DY(-1)`: The last day of the year.
`BY`	Sets a business day of the year. Allowed values: Any number except `0`.	`BY(1)`: The first business day of the year. `BY(-1)`: The last business day of the year.

6.17 Process Scheduler

Process Scheduler is a periodic, workflow application-level timer that starts process instances of specified process definitions contained in a workflow application based on the schedule set in the timer. The timer is configured using the `ProcessScheduler.xml` file, which should be placed in the `<DMSRoot>/apps/<workflow applicationName>/` folder.

Apart from setting the schedule in `ProcessScheduler.xml`, you also specify the Business Calendar the timer should use. Optionally, you can also specify an expiration date after which the timer will no longer be valid. This expiration date can be extended and timer will execute till new extended period.

Note the following points about Process Schedulers:

- For a workflow application you can configure multiple timers using that workflow application's `ProcessScheduler.xml` file. Each timer can be configured to start multiple process definitions.
- A workflow application must be in online state for its Process Scheduler to work properly.
- Any process definitions specified within a timer must be in `published` state and present in the workflow application for which the `ProcessScheduler.xml` has been defined.
- A Process Scheduler logs exception, audit messages in the server log file. The states of the timers are also displayed in this log file.

6.17.1 Defining and Using a Process Scheduler

You cannot create or work directly on Process Schedulers using APIs.

The only way to create and operate Process Schedulers is through the `ProcessScheduler.xml` file. This file can either be created in Interstage BPM Studio, or it can be created manually.

To define Process Scheduler manually:

1. Create the `ProcessScheduler.xml` file using the format specified below. Note that the file name is case sensitive.

2. Place it in the `<DMSRoot>/apps/<workflow applicationName>/` folder
3. Start the workflow application.

To edit or delete a timer in the Process Scheduler:

1. Stop the workflow application that contains the timer.
2. Edit or delete the timer in the `ProcessScheduler.xml` file.
3. Re-start the workflow application.

When a workflow application is started, the process scheduler timer instance is:

- Created in the database if it has not been created earlier but exists in `ProcessScheduler.xml`.
- Updated in the database if it exists in both `ProcessScheduler.xml` and the database.
- Deleted from the database if it exists in the database and not in the ProcessScheduler.xml.

ProcessScheduler.xml Details

The format of the `ProcessScheduler.xml` file is as shown in the following sample.

Note: All tag names are case-sensitive.

```
<ProcessScheduler>
   <Timers>
      <Timer>
        <Name>CheckClearanceTimer</Name>
          <ProcessDefinitions>
            <ProcessDefinition>CheckClearanceProcess1</ProcessDefinition>

            <ProcessDefinition>CheckClearanceProcess2</ProcessDefinition>

          </ProcessDefinitions>
          <Calendar>MyCalendar</Calendar>
          <Schedule>BT(05:00:00)</Schedule>
          <ExpirationDate>2015/12/31</ExpirationDate>
      </Timer>
      <Timer>
        <Name>CashTransferTimer</Name>
          <ProcessDefinitions>
             <ProcessDefinition>CashTransferProcess</ProcessDefinition>
          </ProcessDefinitions>
          <Calendar>MyCalendar</Calendar>
          <Schedule>WN(6);CT(-01:00:00)</Schedule>
          <ExpirationDate>2015/12/31</ExpirationDate>
      </Timer>
   </Timers>
</ProcessScheduler>
```

The tag details are as below.

Tag	Description
`<Name>`	Unique name of the timer which must not contain any of following characters: `\ | / : * ? " < >` and must not be longer than 64 characters. For example, `MyProcessScheduler`. This is a mandatory tag.

Tag	Description
`<ProcessDefinition>`	Name of the process definition in this workflow application whose process instance is to be started by the process scheduler. State of specified process definition must be published. For example, `MyProcessDefinition`. This is a mandatory tag.
`<Calendar>`	Name of the business calendar file, without the `.cal` extension. The system uses `Default` when no calendar is specified. For example, `MyBusinessCalendar`. This is an optional tag.
`<Schedule>`	Periodic timer specified using business calendar format. For example, `WN(1);BT(01:00:00)` Refer section *Time and Day Codes for Timers* on page 155 for a list of time and day codes. As listed in the restrictions below, certain time and day codes and combinations may lead to errors. This is a mandatory tag.
`<ExpirationDate>`	Expiration date of this timer. The timer will not be scheduled if this date has lapsed. Expiration date will be used from the time zone specified in the business calendar file. Format: `yyyy/mm/dd` For example `2020/12/31` This is an optional tag. If this tag is not specified, the timer will continue to be valid for infinite time.

Restrictions

Note the following restrictions when using Process Schedulers:

1. If a periodic timer instance is configured in `ProcessScheduler.xml` for which the difference between the 'next-fire time' and 'previous-fire time' is less than 1 minute, then the first timer instance will be executed but the next timer instance will not be scheduled, and will be put into error state. The error message will be logged into the server log file.

   This error can be recovered by stopping the workflow application, updating the schedule to correct code which will result in more than one minute interval between two timer executions, and then restarting the workflow application.

2. Workflow applications will not be started and no timer instances created if `ProcessScheduler.xml`:
   - contains invalid parameters (such as timer name containing `\ | / : * ? " < >` characters, and so on)

- does not contain mandatory tags
- contain mandatory tags that are empty or contain an empty string
- specifies a business calendar file that does not exist within the workflow application

3. If a process definition specified in `ProcessScheduler.xml` does not exist or does not have a published version, then this timer instance will log a WARN message in the server log file specifying that the process definition does not exist. The timer instance will start the process instance for those process definitions which are present and published and then schedule next timer instance till the Expiration date is not reached.
4. While all time and day codes mentioned in section *Time and Day Codes for Timers* on page 155 can be used for Process Scheduler, use them carefully since for some codes and combinations whenever the next expiration time is calculated using the business calendar, the next and previous expiration time will be same, leading to error. For example:
   - `WD(X);AT(XX:XX:XX)`
   - `CT(-XX:XX:XX)`

Missed Events

Missed events for timers are generated and the timers fired for these events in any of the following cases:

- If a (valid or expired) timer is updated and the workflow application is re-started.

  For example: A timer was last fired at 2:00 pm and is scheduled to fire at 4:00 pm. At 2:30 pm, the workflow application is stopped and timer is updated to fire every hour instead. If the workflow application is restarted at 5:30 pm, there will be three immediate timer executions for missed timer events corresponding to 3:00pm, 4:00 pm and 5:00 pm.

- If a workflow application is stopped (either manually, or for other reasons such as tenant-deactivation, server stop) and then restarted.

  For example: A timer was last fired at 2:00 pm and is scheduled to fire at 4:00 pm (every 2 hours). At 2:30 pm, the workflow application is stopped. Then the workflow application is started at 6:30 pm. In this case the timer will immediately be fired for each missed timer instance, that is, timer will be executed for 4:00 pm, 6:00 pm and then schedule for 8:00 pm.

Timer Instance States in Server Log File

Timer instance states (used in the log file) are as follows. This information is helpful for troubleshooting Process Scheduler-related errors.

- **`NotHandled`**: implies the Timer is configured and will fire at the set time.
- **`workflow applicationStopped`**: implies the Timer will not fire since the workflow application has been stopped.
- **`Error`**: implies the Timer will not fire due to an error condition, for example, when 'next evaluate time' - 'first evaluate time' is less than 1 minute.
- **`Handled`**: implies the Timer will not be scheduled and not be fired since it has expired. This state also occurs if a timer is updated and the new scheduled time is beyond the expiration date. A `Handled` timer can be re-used by updating its expiration date to be later than the current date.

6.18 Modeling Remote Subprocesses

Remote subprocesses are subprocesses that run on a remote workflow server. A remote workflow server might be, for example, another Interstage BPM Server.

A remote subprocess is represented by a Remote Subprocess Node in the parent process definition. The interaction between parent and remote subprocess comprises the following steps:

1. When a Remote Subprocess Node is reached, the local workflow server sends a Start Process request to the remote workflow server, carrying with it the User Defined Attribute (UDA) values that the remote subprocess instance will work on.
2. The remote workflow server must receive this request, and start a process instance.
3. When the remote process instance completes, its workflow server sends a Process Completed message back, carrying with it the results of the subprocess instance.
4. The local workflow server must receive this message, incorporate the results, and complete the Remote Subprocess Node allowing the process instance to continue to the next node.

For a successful interaction, the following must be defined with the parent process definition:

- The URI of the remote process definition
- The identifiers of the UDAs that will be passed back and forth
- Return values of the subprocess definition.

6.18.1 Designing a Parent and Remote Subprocess Definition

This section explains how to design the parent and the subprocess definition. It explains the central steps based on the following sample process definitions:

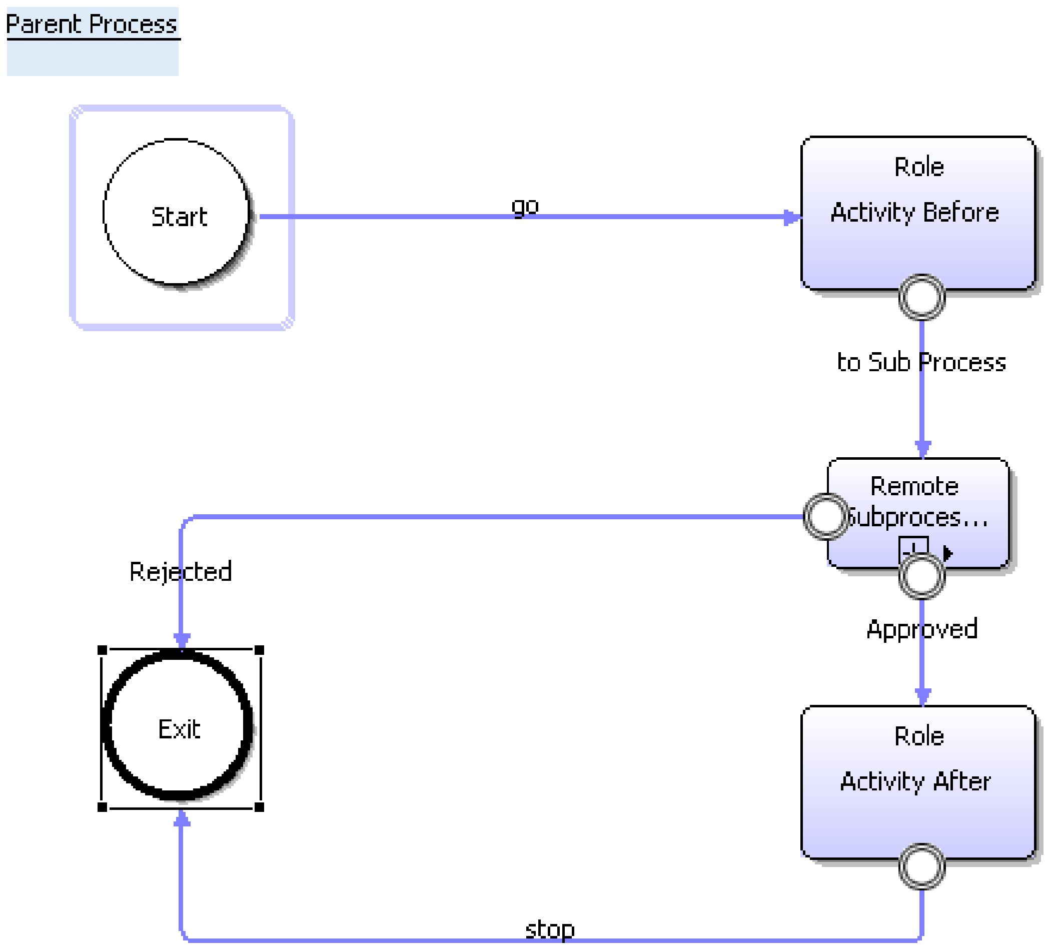


Figure 19: Parent Process Definition with a Remote Subprocess Node

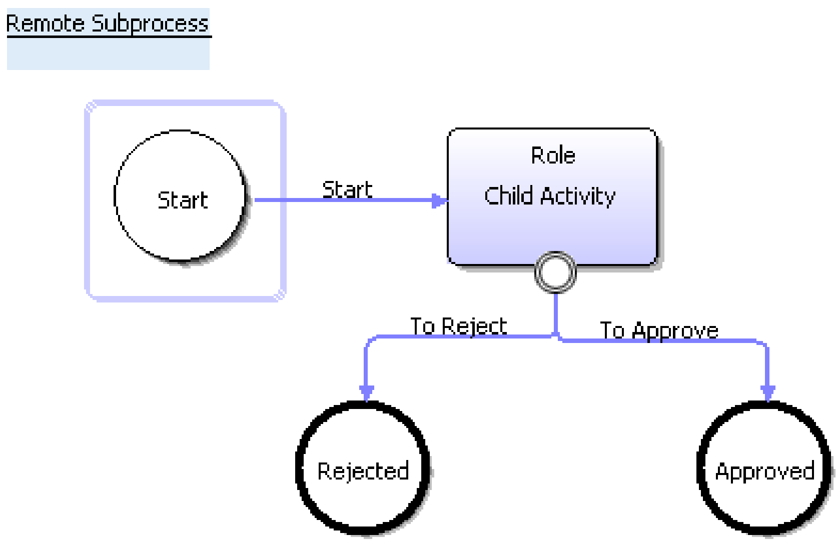


Figure 20: Remote Subprocess Definition

You can find the complete programming code in the sample file `RemoteSubProcess.java`.

To design a parent and a remote subprocess definition:

1. Create the subprocess definition. Add a Start Node, one or more Exit Nodes and the required Activity Nodes to the subprocess definition. Also, specify the UDAs that will be passed back and forth.

   In the sample program, `createRemoteSubPlan()` is called to create a subprocess definition:

   ```
   Plan subPlan = createRemoteSubPlan();
   ```

   The sample subprocess definition consists of a Start Node, an Activity Node and two Exit Nodes. The Exit Nodes define the possible results of the subprocess definition that need to be handled by the parent process definition:

   ```
   Node exitNodeApp = plan.addNode("Approved", Node.TYPE_EXIT);
   exitNodeApp.setPosition(new Point(500, 350));
   Node exitNodeRej = plan.addNode("Rejected", Node.TYPE_EXIT);
   exitNodeRej.setPosition(new Point(200, 350));
   ```

   The following UDAs are created in the sample subprocess definition:

   ```
   plan.addDataItemRef("Name",
         DataItemRef.TYPE_STRING, "John Smith");
   plan.addDataItemRef("Number",
         DataItemRef.TYPE_STRING, "0123456789");
   plan.addDataItemRef("Amount",
         DataItemRef.TYPE_FLOAT, "0.0");
   plan.addDataItemRef("Limit",
         DataItemRef.TYPE_FLOAT, "0.0");
   ```

2. Create the parent process definition and set it to edit-mode.

```
plan = WFObjectFactory.getPlan();
plan.setWFSession(adminSession);
plan.startEdit();
```

3. In the parent process definition, create the UDAs that will be passed back and forth. In the sample program, the following UDAs are created:

```
plan.addDataItemRefWithId("Name", "Name",
      DataItemRef.TYPE_STRING, "John Smith");
plan.addDataItemRefWithId("SSN", "SSN",
      DataItemRef.TYPE_STRING, "0123456789");
plan.addDataItemRefWithId("LoanAmount", "Loan Amount",
      DataItemRef.TYPE_FLOAT, "0.0");
plan.addDataItemRefWithId("ApprovalLimit", "Approval Limit",
      DataItemRef.TYPE_FLOAT, "0.0");
```

Note: You can use different names for the same UDA in the process definitions involved. However, the data type must be identical.

4. Add a Start Node, one or more Exit Nodes and the required Activity Nodes to the parent process definition.
5. Add a Remote Subprocess Node to the parent process definition.

```
Node RemoteSubProcessNode = plan.addNode("Remote",
   Node.TYPE_REMOTE_SUB_PROCESS);
RemoteSubProcessNode.setPosition(new Point(400, 300));
```

6. Define data mappings for the UDAs that need to be passed back and forth. Use the following method of the `Node` interface:

```
addDataMappingElement(<UDA identifier in parent process definition>,
     <UDA identifier in subprocess definition>,<direction of data flow>)
```

When the data value passes from the parent to the remote subprocess, specify `DataItemMappingElement.IN` as the direction of data flow. When the data value passes from the remote subprocess to the parent process, specify `DataItemMappingElement.OUT`. When the data value passes is both directions, specify `DataItemMappingElement.INOUT`.

In the sample program, the following data mappings are defined:

```
RemoteSubProcessNode.addDataMappingElement("Name","Name",
      DataItemMappingElement.IN);
RemoteSubProcessNode.addDataMappingElement("SSN","Number",
      DataItemMappingElement.IN);
RemoteSubProcessNode.addDataMappingElement("LoanAmount", "Amount",
      DataItemMappingElement.IN);
RemoteSubProcessNode.addDataMappingElement("ApprovalLimit", "Limit",
      DataItemMappingElement.OUT);
```

7. Connect the parent and subprocess definition using the following method. To do so, specify the communication protocol to be used by the workflow servers involved and the URI of the remote subprocess definition:

   ```
   RemoteSubProcessNode.setSubPlanURI("asap:" +
   subPlan.getPlanURI());
   ```

   Interstage BPM supports two open protocols for communication between workflow servers: Simple Workflow Access Protocol (SWAP) and Asynchronous Service Access Protocol (ASAP). These protocols pass XML messages over HTTP between workflow servers.

   You are recommended to use ASAP (specified as `asap`) when the processes run on integrated Interstage BPM Servers. Use SWAP (specified as `swap`) with Collaboration Ring only. For details about Collaboration Ring integration, contact your local Fujitsu Support Organization.

8. Add arrows to connect the nodes in the parent process definition. Make sure to add an outgoing arrow to the Remote Subprocess Node for each result that the subprocess might return.

   The result values correspond to the names of the Exit Nodes in the remote subprocess definition.

   A Remote Subprocess Node may have one or more outgoing arrows. Only one arrow is chosen when the parent process resumes. The arrow that is chosen is the one that matches the result of the remote subprocess. In the sample program, the subprocess returns either `Approved` or `Rejected`.

   ```
   Arrow ExitSubProcessApp = plan.addArrow("Approved", RemoteSubProcessNode,
    activityNodeAfter);
   Arrow ExitSubProcessRej = plan.addArrow("Rejected", RemoteSubProcessNode,
    exitNode);
   ```

Note: The names of the Exit Nodes in the subprocess definition and the names of the outgoing arrows must be identical, also regarding uppercase/lowercase.

6.18.2 Running Remote Subprocess Definitions

To be able to run remote subprocess definitions, an Interstage BPM Linkage User must be configured on the local and the remote Interstage BPM Server:

- On the local Interstage BPM Server, a user must be specified that can be authenticated on the remote Interstage BPM Server. This user will create the subprocess instance on the remote Interstage BPM Server.
- On the remote Interstage BPM Server, a user must be specified that can be authenticated on the local Interstage BPM Server. This user will return the result of the remote subprocess instance to the local Interstage BPM Server.

The Interstage BPM Linkage User is configured in the parameters `SWAPLinkageUserName` and `SWAPLinkagePassword` of the Interstage BPM Server. For more information, refer to the *Interstage Business Process Manager Server Administration Guide*.

Only published subprocess definitions can be called remotely unless the subprocess definition belongs to the Interstage BPM Linkage User. The Interstage BPM Linkage User can call remote subprocess definitions that are in Draft state. Therefore, use one of the following options to test your subprocess definitions:

1. Publish the remote subprocess definition before running it.
2. Find out the Interstage BPM Linkage User configured for the local Interstage BPM Server; use it as the owner of the subprocess definition.

3. Change the Interstage BPM Linkage User configured for the local Interstage BPM Server; specify the owner of the subprocess definition as the Interstage BPM Linkage User.

6.18.3 Error Handling for Remote Subprocesses

You can handle the case when the starting of a remote subprocess fails, e.g. because the remote server is not started by defining an Error Java Action for the Remote Subprocess Node in the parent process definition. In this way you can avoid that the process instance goes into error state as soon as the subprocess cannot be started.

An Error Java Action Set on the Remote Subprocess level will become active if the remote subprocess fails to start. Refer to section *Using Error Java Actions* on page 101 for more information on this type of Java Action.

To design and assign an Error Action Set to a Remote Subprocess Node:

1. Use `getJavaActionSet()` from the `WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet errorJavaActionSet =
   WFObjectFactory.getJavaActionSet();
```

2. Generate the required number of Java Actions for `JavaActionSet`.

   In the following example, one Java Action is generated.

```
JavaAction[] errorJavaActions =
   errorJavaActionSet.createJavaActions(1);
```

3. Define the Java Action Set.

```
errorJavaActions[0]
    .setActionName("WriteLogEntry");
errorJavaActions[0]
    .setActionDescription("Writes a log entry in case the Remote
                                  Subprocess could not be started");
errorJavaActions[0]
    .setMethodName("writeLogEntryWithException(String,ServerEnactmentContext)");
errorJavaActions[0]
    .setClassName(CLASS_NAME_JAVA_ACTION);
errorJavaActions[0]
    .setArgumentsUDANames("<E>uda.LogEntry</E><E>sec</E>");
errorJavaActionSet.setJavaActions(errorJavaActions);
```

4. Assign the `JavaActionSet` to the Remote Subprocess Node.

```
RemoteSubProcessNode.setJavaActionSet(errorJavaActionSet,
    JavaActionSet.NODE_ERROR);
```

6.19 Using Compound Activity Nodes

A Compound Activity node encompasses a phase of a process and the milestones to be achieved at the end of the phase. A phase is a container that contains various nodes and arrow transitions. Once a phase is complete a milestone in the process is said to be achieved and the process can move to the next activity or another phase. You can also define due date for a phases. This due date can be defined using different types of Timers.

The Compound Activity node has child nodes nested inside it. It should have Start and Exit nodes as child nodes. Refer to *Defining a Compound Activity Node* on page 167 for more information on how to define a Compound Activity node.

The Compound Activity node and the nodes nested inside a Compound Activity node will have the same Process Definition ID as the parent process. The nodes nested inside the Compound Activity node cannot be accessed outside this node.

The following figure illustrates an example of a Compound Activity node.

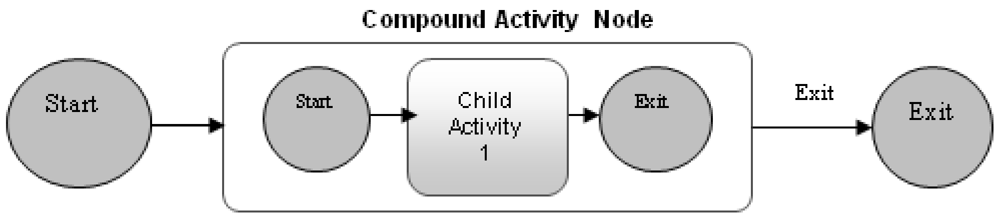


Figure 21: Compound Activity Node

Workflow for Compound Activity Nodes

When the control reaches the compound activity node in a process,

1. Workitem for the Compound Activity is created and WorkItem state is changed to `WaitingForSubProcess`.
2. The child Start node is activated.

Note:	A Compound Activity node must have Start and End nodes inside it.

3. The status of Compound Node is changed to `WaitingOnSubProcess` and the child nodes will be executed.
4. When the Child Exit node is reached
   - The Compound Activity Node is closed and its state is changed to `Completed`.
   - Workitems of the Compound Node are deleted.
   - Outgoing arrow from the Compound Activity node that has the same name as child Exit node is activated.

Note:	Make sure that the child Exit node name matches any one of the outgoing arrow names from the Compound Activity node.

Note:	When the work item for Compound Activity is completed by making choice, then any active child node instance will be aborted, its Work items are deleted and the Compound Activity state is changed to `Completed`.

For more information on functions that are supported on Compound Activity node, refer to information in *Node Types* on page 33.

6.19.1 Defining a Compound Activity Node

Pre-requisites

The Process Definition to which you are adding a Compound Activity node should be in the edit mode.

To define a Compound Activity Node

1. Create an Compound Activity node using `addNode()`. Set the constant `nodeType` to `TYPE_COMPOUND_ACTIVTY`

```
 Node activity = plan.addNode("CompoundActivity",
Node.TYPE_COMPOUND_ACTIVTY);
```

   Following sample code adds Compound Activity `CompoundNode` to the process Definition `Plan`

```
 // Adding Compound Activity Node
Node compoundNode =
plan.addNode("CompoundNode",Node.TYPE_COMPOUND_ACTIVITY);
compoundNode.setRole("SampleGroup");
compundNode.setSize(new Dimension(200,100));
```

2. Add child nodes using the `addChildNode()` API of the Interface `com.fujitsu.iflow.model.workflow.Node`

   Following sample code adds child nodes to the Compound Activity node.

```
 //Adding child nodes
Node subStartNode = compoundNode.addChildNode("SubStart",
Node.TYPE_START);
Node subActivityNode =
compoundNode.addChildNode("SubActivity",Node.TYPE_ACTIVITY);
subActivityNode.setRole("SampleGroup");

//Exit Node name should be same as Compound Activities' outgoing Arrow
 name
Node subExitNode = compoundNode.addChildNode ("CommonName",
Node.TYPE_EXIT);
```

3. Add arrows for the child nodes. Make sure that name if an outgoing arrow is same as the exit node name under the Compound Activity Node

   The following sample code adds arrows to connect various Child Nodes

```
 plan.addArrow("SubArrow1", subStartNode, subActivityNode);
 plan.addArrow("SubArrow2", subActivityNode, subExitNode);
```

   The following sample code adds an outgoing arrow with the same name as the exit node.

```
 plan.addArrow("CommonName", compoundNode, actNode);
```

Note: You cannot define arrow transitions from child nodes inside Compound Activity to nodes outside Compound Activity.

6.20 Using Dynamic Subtasks

When a task is activated, it might be required to create subtasks to that task and assign these subtasks to different users, in order to complete the task. Creating dynamic subtasks to a task enables you to assign these subtasks to different users.

To add a subtask to an active task (Activity1), a dynamic child node instance needs to be created and assigned. The workitem that is generated for this dynamic child node instance becomes the subtask for Activity1.

Note: To add a subtask to a task, the task should be either in the `Active` or `WaitingForSubProcess` states.

When you add node instance to an activity, the task and the associated node instances states are changed to `WaitingforSubProcess`. The dynamic child node instance and associated subtasks states are changed to `Active`. When all the subtasks of a task are complete, the task is said to be complete.

Note: The workitems of dynamic child node instance is deleted when dynamic node instance is completed or aborted.

You can also add dynamic child node instances to existing dynamic child node instances thus, nesting dynamic child node instances.

A dynamic child node instance contains:

- Name
- Description
- Priority
- Comments
- Owner
- Order (The order of task that can be used when listing)
- Assignee
- Point (The position of task that can be used when displaying on screen)

You can set due date on a dynamic child node instance.

You cannot set the following on dynamic child node instances and subtasks:

- JavaAction
- Javascript
- Timers
- Triggers
- Iterator Count
- Recall Flag
- Transaction Flag
- Subprocess
- Chained Process
- Remote Subprocess
- Arrows (When creating dynamic child node instance, the arrow named complete is created as default.)
- Forms

To add subtasks to tasks:

The following steps demonstrate adding of dynamic child node instance (newNodeInst) to a node instance (nodeInstance).

1. Start a process instance and activate a node instance. For example, nodeInstance is activated.

2. On an active node instance, create a dynamic child node instance (for example, newNodeInst) using the API addChildDynamicNodeInstance(). Refer to the sample code below.

```
// create dynamic node instance
ProcessInstance procInst = … // Get process instance
String[] assignees = {"user1","user2"}
procInst.startEdit();
NodeInstance newNodeInst = nodeInstance.addChildDynamicNodeInstance("Sub
 Task",assignees) ;
newNodeInst.setDesc("Description of this sub task");
newNodeInst.setOrder(10);
newNodeInst.setPriority(Node.PRIORITY_MEDIUM);
procInst.commitEdit();
```

Note: addChildDynamicNodeInstance() API belongs to the NodeInstance of com.fujitsu.iflow.model.workflow package. Refer to the API JavaDoc for more information.

Now, newNodeInst is in `Active` state and nodeInstance is in `WaitingForSubProcess` state.

The workitem that is generated for newNodeInst becomes the subtask for workitem generated for nodeInstance.

Note: You can get the parent node instances and dynamic child node instances using the APIs `getParentNodeInstance()` and `getChildNodeInstances()`.

6.21 Using Dynamic Processes

During execution of a process instance or a workitem, you might have the need to complete some activities that are not part of the original process definition. In other words, you might want to create activity nodes dynamically at run-time and complete their associated tasks when these activities are not part of the original plan.

Dynamic process instance is a process instance that is created without being associated with any process definition. Dynamic processes do not have any of arrow transitions. They are just a collection of tasks that need to be completed for the dynamic process to be completed.

To create a dynamic process instance use the `WFObjectFactory.createDynamicNodeInstance()` method.

Note: When you create an application space, application ID is defined when `WFAdminSession.createApplicationSpace(String applicationId)` is executed. To create dynamic process, it is necessary to know the application ID for the application in which dynamic process is to be created. You must use `WFSession.chooseApplication()` to determine the application ID before `WFObjectFactory.createDyanmicNodeInstance()` is executed to create dynamic process.

When you execute `WFSession.chooseApplication()` method, it is necessary to specify the `applicationId` other than the `System`.

To create a dynamic process and generate dynamic node instances

The following steps demonstrate creating a dynamic process instance and generating node instances and tasks.

1. Create and start a dynamic process instance using `WFObjectFactory.createDynamicNodeInstance()` API. Refer to the following sample code that creates a dynamic node instance called Root Task.

```
 // create dynamic process instance
NodeInstance nodeInst =
WFObjectFactory.createDynamicNodeInstance(wfSession, "Root Task");
```

2. Create node instances for this dynamic process instance. The dynamic process instance name and node instance name are set as same value. You can modify name and other attributes. Refer to the sample code below.

```
 // create dynamic node instance
 ProcessInstance procInst = nodeInstance.getProcessInstance();
procInst.startEdit();
nodeInst.setName("Modified task name");
nodeInst.setDesc("Description of this sub task");
nodeInst.setOrder(10);
nodeInst.setPriority(Node.PRIORITY_MEDIUM);
procInst.commitEdit();
```

Note: After a dynamic node instance is created, the process instance creator is set as node instance owner, and work items are assigned to the node instance owner.

3. On this active dynamic node instance, add child dynamic node instances. The dynamic node instance state changes to `WaitingForSubProcess` and the child dynamic node instance becomes `Active`. Refer to the sample code below.

```
 procInst.startEdit();
String[] assignees = String{"user01","user02"};
nodeInst.addChildDynamicNodeInstance("NewTask", assignees);
nodeInst.setDesc("Description of this sub task");
nodeInst.setOrder(10);
nodeInst.setPriority(Node.PRIORITY_MEDIUM);
procInst.commitEdit();
```

6.22 Decision Tables

Decision Tables allow you to design advanced rules for decision making without programming. Decision Tables use a simple yet powerful table based approach for managing rules dynamically. They avoid the need to learn, develop, and support a complex rules engine infrastructure.

Decision tables (.dt files) can exist independently of any process definition. This allows the designing of decision tables to be done by business users using business terms, and as such, it can be abstracted out from the more technical activity of creating a process definition.

6.22.1 Decision Table Concepts

A Decision Table is defined using the following:

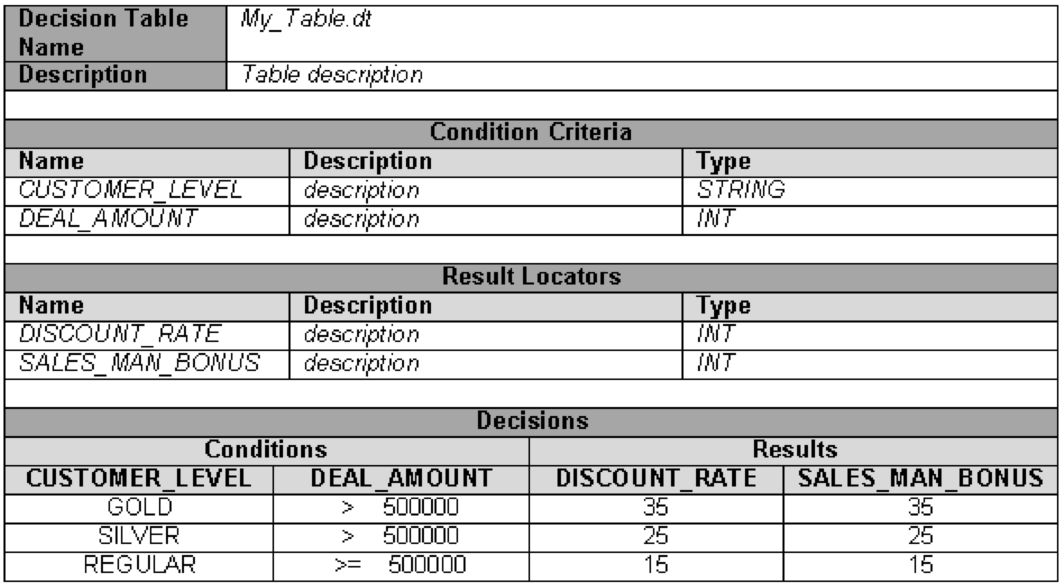
- A list of input variables, called Condition Criteria. For eg., ‘`Deal Amount`’
- A list of output variables, called Result Locators. For eg., ‘`Discount Rate`’

- Conditions, which are pre-defined values with conditions for an input variable. For eg., '`Deal Amount > 5000`' or '`Deal Amount <= 3000`'
- Results, which are pre-defined values for an output variable. For eg., '`Set Discount Rate to 35`'
- Decisions, or rules, which map different conditions to different results. For eg, "`If the condition 'Deal Amount is > 5000' is found to be true, the result 'Set the Discount Rate to 35' should be applied`".

Depending on what values the decision table receives for its input variables (or Condition criteria), it evaluates if any of the listed conditions is met. For the condition that is met, it selects its corresponding pre-defined result, that is, it sets the specified values for its output variables (the result locators).

Creating a decision table mainly involves defining its condition criteria, result locators, conditions, results, and mapping the conditions to results, that is, defining a decision.

A sample Decision Table can conceptually be illustrated as follows:



Decision Table Name	*My_Table.dt*
Description	*Table description*

Condition Criteria		
Name	**Description**	**Type**
CUSTOMER_LEVEL	*description*	*STRING*
DEAL_AMOUNT	*description*	*INT*

Result Locators		
Name	**Description**	**Type**
DISCOUNT_RATE	*description*	*INT*
SALES_MAN_BONUS	*description*	*INT*

Decisions			
Conditions		**Results**	
CUSTOMER_LEVEL	**DEAL_AMOUNT**	**DISCOUNT_RATE**	**SALES_MAN_BONUS**
GOLD	> 500000	35	35
SILVER	> 500000	25	25
REGULAR	>= 500000	15	15

Figure 22: Decision Table concept

In the example above, `CUSTOMER_LEVEL`, `DEAL_AMOUNT` are the input variables or condition criteria. `DISCOUNT_RATE`, `SALES_MAN_BONUS` are the pre-defined output variables or result locators. The different decisions listed are:

- If `CUSTOMER_LEVEL` is `GOLD` and `DEAL_AMOUNT` is greater than `500000`, set the `DISCOUNT_RATE` and `SALES_MAN_BONUS` to `35`
- If `CUSTOMER_LEVEL` is `SILVER` and `DEAL_AMOUNT` is greater than `500000`, set the `DISCOUNT_RATE` and `SALES_MAN_BONUS` to `25`
- If `CUSTOMER_LEVEL` is `REGULAR` and `DEAL_AMOUNT` is greater than or equal to `500000`, set the `DISCOUNT_RATE` and `SALES_MAN_BONUS` to `15`

The decision table is evaluated in a top-down order. Result values are set as per the table for the first condition that is found to be met.

For example, if the `CUSTOMER_LEVEL` is `SILVER`, and the `DEAL_AMOUNT` is `800000`, the first condition is found not to be met. The second condition is found to be met, so the `DISCOUNT_RATE` and `SALES_MAN_BONUS` are set to `25`. Since the second decision has already been selected, the third and fourth decisions are not evaluated.

Data Dictionary

Condition criteria have a feature called the Data Dictionary. This allows storing of synonym values for any pre-defined value of a condition criteria. For example, consider a condition criterion called 'Country' and the condition created is 'Country = Japan'. Pre-defining allowable alternatives for 'Japan', such as 'JP','JPN', etc. will ensure the condition 'Country = Japan' is found to be met even if the end-user enters 'JP' or 'JPN' instead of 'Japan'.

The Data Dictionary feature makes Decision Tables more flexible.

6.22.2 Using a Decision Table within a Process Definition

To use a decision table in a process definition, a pre-defined JavaAction to evaluate Decision Tables (called the Decision Table JavaAction) is provided. The variables of the decision table (condition criteria and result locators) then need to be mapped to UDAs of the process definition. For details on how to do this, refer the *Interstage BPM Studio User's Guide* and/or the *Interstage BPM Console Online Help*.

In a process instance, when the node of the Decision Tables JavaAction is active, if the input UDA values match any of the conditions specified in the Decision Table, the Decision Table JavaAction will set the values of the output UDAs as per the results of the Decision Table.

6.22.3 Decision Table Specifications

ConditionCriterion and ResultLocator

ConditionCriterion and ResultLocator contain following information:

- **Name:** Abstract names of ConditionCriteria and ResultLocators. These names are mapped to actual UDA names when defining a Decision Table JavaAction.
- **Description:** A brief explanation of ConditionCriteria and ResultLocators. It does not affect behavior of the decision table.
- The type of ConditionCriteria and ResultLocators. Following types are supported.
  - BOOLEAN
  - INTEGER
  - LONG
  - FLOAT
  - BIGDECIMAL
  - STRING
  - DATE

Decisions

Decisions have following information:

- conditions for each ConditionCriterion
- values to assign to each ResultLocator

The following operators can be used within conditions:

- `=`, `!=` for exact match of any data type
- `>`, `<`, `>=`, `<=`, for greater/less than comparisons of numeric types
- `in(a,b,c)` for comparison to one of a number of literals
- `between(x,y)` for numeric range testing
- `LIKE()`, `NOT_LIKE()` for pattern matching expressions; question mark (`?`) will match any single character, and asterisk (`*`) will match any number of characters. For example, if `LIKE(?OLD)` is used in a condition, all values such as `GOLD`, `BOLD`, `TOLD` will satisfy this condition; if `LIKE(JAP*)` is used in a condition, all values such as `JAPAN`, `JAPANESE` will satisfy this condition.

Different operators are supported depending on the type of the condition criterion. See the following table:

Operator	Boolean	Integer	Float	Long	BigDecimal	Date	String
`<`	No	Yes	Yes	Yes	Yes	Yes	No
`>=`	No	Yes	Yes	Yes	Yes	Yes	No
`<=`	No	Yes	Yes	Yes	Yes	Yes	No
`>=`	No	Yes	Yes	Yes	Yes	Yes	No
`!=`	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`=`	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`in(a,b,c)`	No	Yes	Yes	Yes	Yes	Yes	Yes
`between(x,y)`	No	Yes	Yes	Yes	Yes	Yes	No
`like`	No	No	No	No	No	No	Yes
`not_like`	No	No	No	No	No	No	Yes

6.22.4 Managing Decision Tables

To manage Decision Tables, use the `com.fujitsu.iflow.decisiontable` package. It contains APIs for managing Decision Tables. For a detailed description of this package and its APIs, refer to the API Javadoc. The following important operations are discussed:

1. Creating decision tables
2. Creating condition criteria, result locators, and decisions
3. Validating decision tables
4. Evaluating decision tables
5. Saving decision tables
6. Loading decision tables

Creating Decision Tables

You can create decision table files by either using the API, or by creating an XML file which conforms to the Decision Table XML Schema.

Creating Decision Tables using API

You can create a Decision Table by using the `createDecisionTable()` method of the `DecisionTableFactory` class. Refer to the API Javadoc for details of this method. For example:

```
DecisionTable dt = DecisionTableFactory.createDecisionTable();
dt.setName("DecisionTableName");
dt.setDescription("example of decision table");
```

Creating Decision Tables manually by conformance to the XML Schema

If you want to create a complete decision table without using the APIs, ensure the Decision Table XML file (.dt file) you create conforms to the XML Schema for the .dt file, `DecisionTable.xsd`, stored in the `<Interstage BPM Installation Directory>/client/DecisionTable` directory.

Creating condition criteria, result locators, decisions

Creating Condition Criteria

Use the `createConditionCriterion()` method of the `DecisionTable` class to create a new condition criterion. Refer to the API Javadoc for details of this method.

In the following example, `CUSTOMER_LEVEL` and `DEAL_AMOUNT` are the two condition criteria created. Also, for `CUSTOMER_LEVEL`, the Data Dictionary feauture is used to create the synonyms 'normal' and 'none' for the main condition criterion value 'Regular'.

```
ConditionCriterion criterion1 = dt.createConditionCriterion();
criterion1.setName("CUSTOMER_LEVEL");
criterion1.setType(DataItemRef.TYPE_STRING);
Dictionary dic = criterion1.createDictionary();
DictionaryItem regularDictionaryItem = dic.createDictionaryItem();
regularDictionaryItem.setWord("Regular");
String synonyms[] = new String[2];
synonyms[0] = "Normal";
synonyms[1] = "None";
regularDictionaryItem.setSynonyms(synonyms);

ConditionCriterion criterion2 = dt.createConditionCriterion();
criterion2.setName("DEAL_AMOUNT");
criterion2.setType(DataItemRef.TYPE_INTEGER);

ConditionCriterion criteria[] = new ConditionCriterion[2];
criteria[0] = criterion1;
criteria[1] = criterion2;
dt.setConditionCriteria(criteria);
```

Creating Result locators

Use the `createResultLocator()` method of the `DecisionTable` class to create a new Result Locator. Refer to the API Javadoc for details of this method. The following example creates the `DISCOUNT_RATE` and `SALES_MAN_BONUS` result locators.

```
ResultLocator locator1 = dt.createResultLocator();
locator1.setName("DISCOUNT_RATE");
locator1.setType(DataItemRef.TYPE_STRING);

ResultLocator locator2 = dt.createResultLocator();
locator2.setName("SALES_MAN_BONUS");
locator2.setType(DataItemRef.TYPE_STRING);

ResultLocator locators[] = new ResultLocator[2];
locators[0] = locator1;
```

```
locators[1] = locator2;
dt.setResultLocators(locators);
```

Creating Decisions

The example below creates the following decisions:

- If `CUSTOMER_LEVEL` = `'GOLD'`, and, `DEAL_AMOUNT` > `'500000'`, set `DISCOUNT_RATE` and `SALES_MAN_BONUS` to `'35'`
- If `CUSTOMER_LEVEL` = `'SILVER'`, and, `DEAL_AMOUNT` > `'500000'`, set `DISCOUNT_RATE` and `SALES_MAN_BONUS` to `'25'`

```
Decision decision1 = dt.createDecision();
Condition condition1 = decision1.createCondition();
condition1.setCriterionName("CUSTOMER_LEVEL");
condition1.setOperator(Condition.CONDITION_OPERATOR_EQUAL);
condition1.setValue("Gold");
Condition condition2 = decision1.createCondition();
condition2.setCriterionName("DEAL_AMOUNT");
condition2.setOperator(Condition.CONDITION_OPERATOR_GREATER_THAN);
condition2.setValue("500000");
Condition conditions1[] = new Condition[2];
conditions1[0] = condition1;
conditions1[1] = condition2;
decision1.setConditions(conditions1);
Result result1 = decision1.createResult();
result1.setLocatorName("DISCOUNT_RATE");
result1.setValue("35");
Result result2 = decision1.createResult();
result2.setLocatorName("SALES_MAN_BONUS");
result2.setValue("35");
Result results1[] = new Result[2];
results1[0] = result1;
results1[1] = result2;
decision1.setResults(results1);

Decision decision2 = dt.createDecision();
... //add code for creating decision2, similar to creating decision1
...

Decision decisions[] = new Decision[2];
decisions[0] = decision1;
decisions[1] = decision2;
dt.setDecisions(decisions);
```

Validating Decision Tables

A decision table should be validated before it can be evaluated.

Validating a decision table involves some basic checks such as:

- Ensuring the Decision Table name is specified
- Ensuring ConditionCriterion name is specified
- Ensuring ConditionCriterion type is specified, and so on

Use the `validate()` method of the `DecisionTable` class to validate a decision table. Refer to the API Javadoc for details of these methods.

```
ValidationResult[] validationResults = dt.validate();
System.out.println(validationResults.length + " Problems detected.");
```

```
for (int i = 0; i < validationResults.length; i++) {
  ValidationResult validationResult = validationResults[i];
  System.out.println("Severity = " + validationResult.getSeverity());
  System.out.println("ID = " + validationResult.getIdentifier());
  System.out.println("Problem = " + validationResult.getProblem());
}
```

Evaluating a Decision Table

Use the `evaluate()` method of the `DecisionTable` class to evaluate or test a decision table with sample input values. Refer to the API Javadoc for details of this method.

Note that to successfully evaluate a decision table, the decision table must be valid. If `validate()` returns any results whose severity is `ERROR`, evaluation of the decision table fails.

The following example sets `CUSTOMER_LEVEL` to a sample value of '`SILVER`' and `DEAL_AMOUNT` to a sample value of '`800000`'. The decision table is then tested using these sample values, and the result values are set accordingly.

```
Properties conditionValues = new Properties();
conditionValues.setProperty("CUSTOMER_LEVEL","Silver");
conditionValues.setProperty("DEAL_AMOUNT","800000");
DecisionResult decisionResult = newDt.evaluate(conditionValues);

int decisonIndex = decisionResult.getExecutedDecision();
Decision allDecisions[] = newDt.getDecisions();
Decision selectedDecision = allDecisions[decisonIndex];

Result results[] = decisionResult.getResults();
for (int i = 0; i < results.length; i++){
  Result result = results[i];
  String locatorName = result.getLocatorName();
  String value = result.getValue();
  System.out.println("result : " + locatorName + " = " + value);
}
```

Saving a Decision Table

Use the `save()` method of the `DecisionTable` class to evaluate or test a decision table with sample input values. Refer to the API Javadoc for details of this method.

```
FileOutputStream outFile = new FileOutputStream("/decisiontable.dt");
dt.save(outFile);
```

Decision tables are application-specific, and are to be stored in the `<DMSRoot>/apps/<application ID>/rule/<category>` directory.

Loading an existing Decision Table

Use the `load()` method of the `DecisionTable` class to load an existing decision table. Refer to the API Javadoc for details of this method.

```
DecisionTable newDt = DecisionTableFactory.createDecisionTable();
FileInputStream inFile = new FileInputStream("/decisiontable.dt");
newDt.load(inFile);
```

API Sample for Decision Tables

For a complete sample of managing a Decision Table, refer section *Samples Related to Decision Tables* on page 217

6.23 Iterator Nodes

Normally, for each node created in a process definition, only a single node-instance is generated at run-time. Enabling iteration on a node with an iteration count 'n' allows you to create 'n' instances of the same node during run-time.

An iterator node has the following general features:

- Iteration is enabled during design-time.
- The number of iterations is specified using a UDA. If a specified UDA for an iterator node does not exist, process validation will fail and no process instance can be started from the process definition until the UDA is added.
- You can enable only the following types of nodes for iteration:
  - Activity nodes
  - Subprocess nodes
  - Chained-process nodes
- All iterated node instances are executed in parallel.
- To enable iteration, iteration count should be > 0.
- If the iteration count for a node is set to <=0:
  - Even though a single instance of a node would already have been created as soon as the process instance is created, that node instance will not be activated
  - Execution of the node instance will be skipped, and the next node will be activated.
- If a completed iterator node is reactivated, node instances created during the earlier execution of the iterator node are re-used by reactivating them. For example, consider during the first run of an iterator node 3 instances were created.
  - If during its second run iteration count is set to 10, then only 7 new instances will be created; 3 instances from the previous run will be re-used by reactivating them.
  - If during the second run iteration count is set to 2, no new instances will be created; of the 3 instances from the previous run, only 2 instances will be reactivated for re-use, and one instance will remain inactive.
- For ad hoc activation, deactivation, if some iterated instances of a node are running, and some are closed:
  - You cannot reactivate a closed instance.
  - If you deactivate a running instance, all running instances will be deactivated.
- Prologue and Epilogue JavaActions are executed only once irrespective of the number of iterations.
- You cannot change the source or target nodes of the outgoing arrows of an iterated node instance.
- If you delete an iterated node instance, all iterated instances for that node are deleted.
- If you delete the outgoing arrow instance of an iterated node instance, all outgoing arrow instances of all iterated instances for that node are deleted.
- If you add an outgoing arrow instance to an iterated node instance, outgoing arrow instances are added for all iterated instances of that node.

- Setting iterator count on a running node instance will not be effected during that run of the node. It will be effected only during the next run (if any) of that node instance.
- Each iterated node instance can be identified and accessed using its iterator index number. Index count ranges from 1 to 'n', where 'n' is the iteration count.
- From a running process instance containing an iterator node, the iterator count UDA specified for the iterator node cannot be deleted.

6.23.1 Iterated Activity Nodes

Iterated Activity nodes have the following features:

- For an Activity node, consider you set iteration count as 'n'. As soon as the process instance is created, the first instance of that node is created. After the process instance starts, when control reaches that node, the following occurs:
  - For an iterated Activity node, 'n-1' instances are created; work items are created for all users of an assigned role, for each of the 'n' node instances. For example, if there are 10 users for an assigned role, and iteration count is 5, 50 work items are generated.
- For each of the instances, if any user completes the work item, the work items for other users within that node instance are deleted, and that node instance is complete.
- When all iterated instances of a node will complete, the activity of that node is completed, and the next node is activated.
- The number of outgoing arrows from an activity iterator node is restricted to one.
- All instances of an iterator node share all the UDAs in the process. If one instance changes a UDA's value, the change is reflected for all instances immediately.
- You cannot use triggers on an iteration-enabled Activity node.
- Each iterated instance has the same properties (name, description, and so on).
- The iterator index number of an iterated instance can be retrieved using a Model API function, JavaScript function or a custom JavaAction. For more information, refer *Retrieving the Iterator Index of a Node Instance using Model API* on page 181 and *Retrieving the Iterator Index of an Activity Node Instance using a Custom JavaAction Class* on page 182.
- You can assign different users to each iterated instance of a node. For more information, refer *Retrieving the Iterator Index of an Activity Node Instance using a Custom JavaAction Class* on page 182. Role Actions are executed for each iterated node instance.

Recalling Work items for Iterator Nodes

Iterator Nodes support the work item recall feature in the following scenarios.

- When an iterator node is a source node for recall, recall is possible only if at least one of the iterator node instances has been closed.
  - If all the iterator node instances are closed, and the next activity is still active - On recall, the active target activity is de-activated. The compensate Java Action defined for prologue Java Action of target activity is executed, the compensate Java Action defined for epilogue Java Action of source activity is executed. The recalled iterator activity is activated, and a work item is created for the user that recalled the work item.
  - If all of the iterator node instances have not been closed - Recall is possible only for any of the closed iterator node instances. On recall, the recalled activity instance is re-activated and a work item created for the user that recalled the work item.

- When an iterator node is a target node for recall, recall is possible only if all iterator node instances are running. On recall, all iterated node instances are de-activated. The compensate Java Action defined for prologue Java Action of target activity will be executed, the compensate Java Action defined for epilogue Java Action of source activity will be executed. The recalled activity is re-activated, and a work item is created for the user that recalled the work item.

For more information about recalling work items, refer *Recalling Work Items* on page 83.

6.23.2 Iterated Subprocess and Chained-process Nodes

Iterated Subprocess and Chained-process nodes have the following features:

- For a subprocess or chained-process node, consider you set iteration count as 'n'. As soon as the process instance is created, the first instance of that node is created. After the process instance starts, when control reaches that node, the following occurs:
  - For an iterated Subprocess node, 'n-1' node instances will be created. For each of the 'n' node instances, an instance of the linked subprocess will be created and started. Control will shift to the next node only after all iterated instances of the subprocess node are complete.
  - For an iterated Chained-process node, no further instances will be created. For the single node instance that was created when the process instance was created, 'n' instances of the linked child process will be created and started. Control will shift to next node as soon the 'n' linked processes are started.
- If the process instance containing a subprocess iterator node is aborted, all iterated subprocess instances are aborted.

Data Mapping for Subprocess and Chained-process Iterator Nodes

- Each iterated instance of a linked process (subprocess or chained-process) has its own UDAs. An instance cannot interact directly with UDAs of its sibling iterated instances. It can directly interact only with UDAs of its parent process.
- You can transmit different data between each iterated child process instance and its parent process instance (and vice-versa) by using an XML UDA and XPath with the predefined variable **`$index`**. The variable `$index` within the Xpath expression of a parent process XML UDA helps map different values from the parent XML UDA to XML UDAs of different iterated child instances. For details of how such data mapping can be implemented, refer *Data Mapping between Parent and Iterated Child Process Instances* on page 183.

6.23.3 Working with Iterator Nodes

Creating an Iterator Node

While designing a process definition, you can enable a node for iteration by using the `setIteratorCount(String udaName)` method of the `com.fujitsu.iflow.model.workflow.Node` interface. Note that the UDA should be of type integer.

You can also retrieve the name of an iterator UDA using the `getIteratorCount()` method of the `com.fujitsu.iflow.model.workflow.Node` interface.

For more information about these methods, refer the API Javadoc.

The following sample code sets the UDA `IteratorCount` as the iterator count UDA name for activity node `Activty`, and also sets the iteration count to `5`.

```
//lock a Process definition before calling the API
plan.startEdit();
plan.addDataItemRef("IteratorCount", DataItemRef.TYPE_INTEGER, "5");
Node activityNode = plan.addNode("Activity", Node.TYPE_ACTIVITY);
activityNode.setIteratorCount("IteratorCount");
//when you save the plan, the Iterator count of
//the above node will also get saved
plan.createProcessDef();
//Now you can check the Iterator count set above
String iteratorCountUda = activityNode.getIteratorCount();
```

An iteration enabled node can be reverted back to a normal node by setting an empty ("") string as the iterator count UDA name.

Note: It is recommended to set iterator count of a node by taking into consideration the number of assignees for each Iterated activity instance, so that it should not result in creation of more than 2000 work items per Process instance for the iterated activity. Otherwise, depending upon the server environment setup and system load, the iterator node activation transaction may take too long and time out, causing the process to go into error state.

Retrieving the Iterator Index of a Node Instance using Model API

You can retrieve the iterator index of a node instance using the `getIteratorIndex()` method of the `com.fujitsu.iflow.model.workflow.NodeInstance` interface. For more information about this method, refer the API Javadoc.

Sample code:

```
 //retrieve the Node instance using Model API, then use code below
//for retrieving iterator index of node instance
//Let 'ni' represent retrieved Node Instance
int iteratorindex ;
iteratorIndex = ni.getIteratorIndex();

//this iteratorIndex can be used for manipulation of an XML UDA as follows

//Create the XPath using above retrieved iteratorIndex
String xpath="//OrderIds/orderId[position()=" + iteratorIndex + "]/text()";


//Retrieve the XML UDA from the process instance
:
:
//Assume 'dataItem' is the retrieved XML UDA

//Set the new value of the XML UDA using the XPath
dataItem.setElementValue(XPath, "Updated");
```

```
//Retrieve the new value
String udaValue = dataItem.getElementValue(xpath);
```

Retrieving the Iterator Index of an Activity Node Instance using a Custom JavaAction Class

You can retrieve the iterator index of an activity instance by calling the `getActivityIteratorIndex()` method of the `com.fujitsu.iflow.server.intf.ServerEnactmentContext` interface, and use this index for reading values from an XML UDA for a particular node instance in a custom JavaAction class.

For example, you would need to define a custom JavaAction class to retrieve the iterator index of activity instances if you want to set different assignees to each instance.

To set Activity Instance Assignees, perform the following steps:

1. Create a custom JavaAction class that uses `getActivityIteratorIndex()` to define a custom method that can be used to set assignees.
2. The custom JavaAction class defined above can be used in a Role Action method to assign different assignees to different iterations of an activity instance.

1. **Creating a Custom JavaAction Class**

   When creating a custom JavaAction class:

   a) Import the `ServerEnactmentContext` interface.

   b) Make sure that one parameter of the method that you want to call from a JavaAction is of type `ServerEnactmentContext`.

   c) When defining the Java Action, specify the values to pass to your method. Use the `sec` identifier as a value for the `ServerEnactmentContext` parameter. At run time, Interstage BPM will pass an object with this interface to the custom method.

   For example, consider the XML UDA `UdaAssignees`

```
<Assignees>
    <Assignee>ibpm_user1</Assignee>
    <Assignee>ibpm_user2</Assignee>
    <Assignee>ibpm_user3</Assignee>
</Assignees>
```

   The sample code below is for the custom JavaAction class `MyClass` that uses the `getActivityIteratorIndex()` method to define a custom method named `setAssignee` that can be used to set assignees.

```
import com.fujitsu.iflow.server.intf.ServerEnactmentContext;
public class MyClass {
    public void setAssignee(ServerEnactmentContext sec)
           throws Exception {
       String xpath = "//Assignees/Assignee[position()=" +
sec.getActivityIteratorIndex() + "]/text()";
       String assignee =
sec.getProcessXMLAttributeElementValue("UdaAssignees", xpath);
//here "UdaAssignees" is the name of the XML UDA containing the
assignees

       String[] assignees = { assignee };
       sec.setActivityAssignees(assignees);
    }
}
```

2. **Using Role Actions to assign different assignees to different iterator instances**
   a) Add an Activity Node. The name of the first activity in the process definition uses the '`protected`', '`final`', '`static`' modifiers.

```
String NODE_FILL_OUT_PR = "Fill out Purchase Request";
protected final static String NODE_FILL_OUT_PR = "Fill out Purchase
Request";
Node fillOutNode = plan.addNode("NODE_FILL_OUT_PR",Node.TYPE_ACTIVITY);
fillOutNode.setRole("SampleGroup");
fillOutNode.setPosition(new Point(450, 40));
```

   b) Use `getJavaActionSet() WFObjectFactory` class to create a new `JavaActionSet` object.

```
JavaActionSet asRJavaActionSet = WFObjectFactory.getJavaActionSet();
```

   c) Generate the required number of Java Actions for JavaActionSet.

```
JavaAction[] asRJavaActions = asRJavaActionSet.createJavaActions(1);
```

   d) Define the JavaActions that set the assignee of an activity

```
asRJavaActions[0].setActionDescription("Sets Activity assignee");
asRJavaActions[0].setActionName("Set assignee");
asRJavaActions[0].setMethodName("setAssignee(ServerEnactmentContext");
asRJavaActions[0].setClassName("MyClass");
```

   e) Assign JavaActionSet to the Activity Node as Role Action.

```
asRJavaActionSet.setJavaActions(asRJavaActions);
fillOutNode.setJavaActionSet(asRJavaActions,JavaActionSet.NODE_ROLE);
```

Note: You can also use the `getActivityIteratotIndex()` in a JavaScript as follows:

`var iteratorIndex =` **`sec.getActivityIteratorIndex();`**

You can further use `iteratorIndex` as required.

Data Mapping between Parent and Iterated Child Process Instances

You can transmit different data between each iterated child process instance and its parent process instance (and vice-versa) by using an XML UDA and XPath with the predefined variable **`$index`**.

Consider the following XML UDAs:

- Parent process XML UDA, named `ParentXMLUDA`

```
<Items>
    <Item>A1</Item>
    <Item>A2</Item>
</Items>
```

- Child instance #1 with XML UDA `ChildXMLUDA_P`

```
<ChildItems>
    <ChildItem>P</ChildItem>
</ChildItems>
```

- Child instance #2 with XML UDA `ChildXMLUDA_P`

```
<ChildItems>
    <ChildItem>P</ChildItem>
</ChildItems>
```

Note that since child instances come from the same child process, all child instances will have UDAs with same names and **initial** values.

You can implement data mapping between parent and child XML UDAs using the `addDataMappingElement()` method of the `com.fujitsu.iflow.model.workflow.Node` interface.

Syntax:

```
addDataMappingElement(java.lang.String parentDataItemRefName,java.lang.String
xPathForParentDataItem,java.lang.String childDataItemRefName,java.lang.String
xPathForChildDataItem,int direction)
```

Why `$index` is needed

Using the `addDataMappingElement()` method without `$index`, for example `addDataMappingElement("ParentXMLUDA", "//Items/Item[2]/text()", "ChildXMLUDA_P", "//ChildItems/ChildItem/text()", DataItemMappingElement.INOUT)` will cause the following to happen:

1. At the start of the child process, UDAs of **both** child instances will receive the **same** value (as specified by `"//Items/Item[2]/text()"`) that is, the **second** item of `ParentXMLUDA`, which is **`A2`**. The child instance UDAs will then be as follows:
   - UDA of child instance **#1**:

   ```
   <ChildItems>
       <ChildItem>A2</ChildItem>
   </ChildItems>
   ```

   - UDA of child instance **#2**:

   ```
   <ChildItems>
       <ChildItem>A2</ChildItem>
   </ChildItems>
   ```

2. At the end of the child instances, both instances will modify the **same** item in the parent instance (as specified by `"//Items/Item[2]/text()"`) that is, the **second** item of `ParentXMLUDA`, thus overwriting each others transferred data. For example, child instance #1 will change the second item of `ParentXMLUDA` to **P1**, and then child instance #2 will overwrite the second item of `ParentXMLUDA` to **P2**. The parent XML UDA would be as follows:

```
<Items>
    <Item>A1</Item>
    <Item>P2</Item>
</Items>
```

Essentially, this means you cannot map different values from a parent process to the different child instances. Such a problem will occur even when you use non-XML UDA data mapping.

To be able to map different values from a parent process to the different iterated child instances, you need to use XML UDAs and the `$index` pre-defined variable.

Using `$index`

Using the `addDataMappingElement()` method with `$index`, for example
`addDataMappingElement("ParentXMLUDA", "//Items/Item[$index]/text()", "ChildXMLUDA_P", "//ChildItems/ChildItem/text()", DataItemMappingElement.INOUT)`
will cause the following to happen:

1. UDAs of **both** child instances will receive **different** values (as specified by `"//Items/Item[$index]/text()"`). The `$index` variable in the Xpath expression of the parent UDA will automatically map the **first** item of the parent XML UDA to the XML UDA of the **first** child instance, the **second** item of the parent XML UDA to the XML UDA of the **second** child instance, and so on. The child instance UDAs will then be as follows:
   - UDA of child instance **#1**:

     ```
     <ChildItems>
         <ChildItem>A1</ChildItem>
     </ChildItems>
     ```

   - UDA of child instance **#2**:

     ```
     <ChildItems>
         <ChildItem>A2</ChildItem>
     </ChildItems>
     ```

2. At the end of the child instances, each child instance will have modified **different** items of the parent XML UDA. The `$index` variable in the Xpath expression of the parent UDA will automatically map the **first** item of the parent XML UDA to the XML UDA of the **first** child instance, the **second** item of the parent XML UDA to the XML UDA of the **second** child instance, and so on. The parent XML UDA will then be as follows:

   ```
   <Items>
       <Item>P1</Item>
       <Item>P2</Item>
   </Items>
   ```

Thus, using `$index` you can map different parent UDA values to UDAs of different iterated child instances.

Note:
- `$index` forces serial order mapping (first item to first child instance, second item to second child instance); you need to ensure that items in your parent XML UDA are ordered corresponding to the child instances to which they are to be mapped.
- `$index` can be used only in Xpath expressions of the **parent** process XML UDA.
- If `$index` is used for non-iterator node XML UDA XPaths, the variable will be replaced by '`1`' during processing.

For other information related to data mapping, refer *Designing a Parent and Remote Subprocess Definition* on page 162.

7 Administration

There exist two administrative roles in Interstage Business Process Manager:

- **Super User**: an administrator whose only responsibilty is to create and manage tenants, and manage the Interstage BPM Server. A Super User cannot administrate tenant users, process definitions, process instances, or work items. A Super User uses the **Interstage BPM Tenant Management Console** for administration of tenants. For information about administrative functions performed by a Super User refer the *Interstage Business Process Manager Console Tenant Management Guide* and *Interstage BPM Administration Guide*.
- **Tenant Owner**: a tenant user in an administrative role. The Tenant Owner is created by the Super User.

In non-SaaS mode, since there is only a single default tenant, the Super User and Tenant Owner may be the same person but may need to log in as a Super User or Tenant Owner to perform the different administrative functions.

This chapter provides programming examples, using the Model API, for administrative operations performed by a Tenant Owner.

A Tenant Owner can perform the following administrative functions:

- Logging in/logging out a tenant owner
- Tenant user administration (hereafter referred to as user administration)
- Process definition administration
- Process instance administration
- Work item administration

You can find the complete programming code of the examples presented in this chapter in the `Adminstration.java` and the `SampleLocalUserManagement.java` sample files.

For implementing the examples, you need the following packages from the Model API:

- **`com.fujitsu.iflow.model.util`**

  Contains low level utility classes that are commonly used by other classes and interfaces.

- **`com.fujitsu.iflow.model.workflow`**

  Contains interfaces that manage information required by process definitions and process instances. This includes providing objects that represent nodes, arrows, forms, attachments, work items and permission levels.

For more information on the Model API in general, refer to the API Javadoc.

7.1 Logging In/Logging Out an Administrator (Tenant Owner)

For administering process definitions or tenant users, an administrator (tenant owner) must be logged in to the Interstage BPM Server. Logging in an administrator means to create a session, i.e. a `WFAdminSession` object. The session is finished when an administrator is logged out again.

To log in/log out an administrator:

1. Create a new `WFAdminSession` object.

   ```
   adminSession = WFObjectFactory.getWFAdminSession();
   ```

   Use the `WFObjectFactory` class for accessing workflow objects. `getWFAdminSession()` then creates a `WFAdminSession` object.

2. Initialize the session with the appropriate configuration file.

   You can use the sample `iFlowClient.properties` file located in `<Interstage BPM Server Installation Directory>/client/samples/examples/classes`.

   Either use this file or write the properties into a new file, which must be located in the current runtime directory. For the location of the current runtime directory, refer to section *Location of Properties Files* on page 61.

Note: Ensure you specify the tenant name logged in for the property file used. The tenant name is specified for value of `WFObjectFactory.TENANT_NAME(TenantName)`. `TenantName=Default` is specified for login to `Default` tenant.

Note: In the `iFlowClient.properties` file, any backslashes "\" or colons ":" are escaped by backslashes. For example, a server address is specified like this:

`ibpmhost\:49950`

When loading the `iFlowClient.properties` file using the `java.util.Properties.load()` method, escape characters will automatically be taken into account. If you use another way to load the properties, make sure that you handle any escape characters correctly. For details about escape sequences that may occur in the `iFlowClient.properties` file, refer to the Java documentation of the `java.util.Properties.store()` method.

   Load the configuration file `iFlowClient.properties` for the session:

```
Properties sessionProps = new Properties();
   sessionProps.load(new FileInputStream("iFlowClient.properties"));
```

   Initialize the session:

```
adminSession.initForApplication(null, sessionProps);
```

3. Log in an administrator.

```
String server = sessionProps.getProperty("HostName") + "Flow";
   adminSession.logIn(server, adminName, password);
```

   For implementation reasons, always attach the constant `"Flow"` to the given host name.

4. After the administration tasks are completed, the administrator has to log out from the `WFAdminSession`. To log out an administrator:

```
if (adminSession != null ) {
   adminSession.logOut();
   }
```

7.2 Choosing a Workflow Application

After logging in, choose a workflow application within which you want to operate, using `WFSession.chooseApplication()`

```
adminSession.chooseApplication(myApplicationID);
```

Note: If the `ApplicationModeSecurity` server parameter is set to `Relax`, choosing an application is optional. To know about detailed behavior when an application is not chosen, refer the `ApplicationModeSecurity` parameter in the *Interstage BPM Server Administration Guide.*

7.3 User and Group Administration

Every tenant user (hereafter referred to as a user) that is to work with Interstage BPM needs a user account and must be assigned to one or more groups.

User Accounts

Depending on how Interstage BPM has been configured during deployment, user accounts are managed either in Interstage BPM's local user store or in a Directory Service (remote user store).

If you are using the local user store, you can add users, delete users and reset their password using the `WFAdminSession` interface.

If you are using a Directory Service, you use the functions of the Directory Service to add and delete users.

Groups

A group is a collection of users who share a function within an organization. For example, a `Manager` group might contain the first-line managers in an organization. In Interstage BPM, groups are used to determine who is responsible for carrying out a task in a process.

Groups can be managed in Interstage BPM's local group store, in a Directory Service (remote group store), or in both systems. If you are using the local group store, you can manage it using the `WFAdminSession` interface. The `WFAdminSession` interface has operations for creating groups, adding groups to other groups, adding users to groups, and so on.

Because groups can be members of other groups, you can model your organizational hierarchy into a corresponding group hierarchy. Groups can be nested to any depth.

Note: Users and groups can be managed in different stores. For example, an organization might manage user accounts in a Directory Service and groups in the local group store.

7.3.1 Managing Local Users

Prerequisite: Interstage BPM has been configured to use its local user store. For more information, refer to the *Interstage Business Process Manager Server and Console Installation Guide*.

The `WFAdminSession` interface allows you to retrieve a list of all local users, add local users, reset their password and delete local users.

- **To retrieve a list of all local users:**

```
User[] localUser = adminSession.getLocalUsers();
```

- **To add a local user:**

```
//Check whether the user already exists
User testUser = WFObjectFactory.getUser(sampleUser);
adminSession.createUser(testUser, usrPassword);
```

  The user ID and the password can contain up to 200 characters. Use only alphanumeric characters, hyphens and underscore characters ("_"). User names must not begin with an at character ("@") as this is used to identify Agents in Interstage BPM.

- **To reset the password of a local user:**

```
adminSession.resetPassword(testUser, usrPassword);
```

Note: If you want users to be able to change their password, use `changePassword()` from the `WFSession` interface.

- **To delete a local user:**

```
adminSession.deleteUser(userID);
```

The user will be deleted and removed from all local groups to which the user belongs.

7.3.2 Managing Local Groups

This section introduces the most important operations for managing the local group store. In the programming sample, a hierarchy of three groups is created and users are assigned to those groups. The following figure shows the organizational structure that will be created:

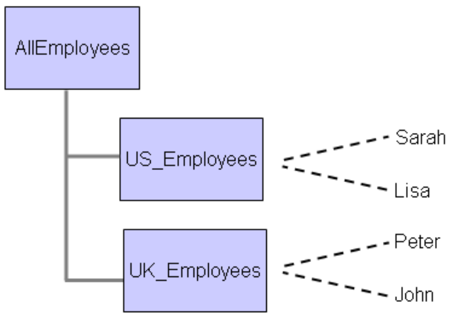


Figure 23: Sample Groups and Users

To manage local groups:

1. Create the required local groups using `createGroup()` from the `WFAdminSession` interface.

   The following sample creates an `AllEmployees`, `US_Employees` and `UK_Employees` group. Before creating a group, the sample checks whether the group already exists.

```
group = WFObjectFactory.getGroup("AllEmployees ",
        "ALL Employee group");
adminSession.createGroup(group);
group = WFObjectFactory.getGroup("US_Employees ",
        "US Employee group");
```

```
adminSession.createGroup(group);
group = WFObjectFactory.getGroup("UK_Employees ",
        "UK Employee group");
adminSession.createGroup(group);
```

The group name can contain up to 200 characters. Use only alphanumeric characters, hyphens and underscore characters ("_"). Group names must not begin with an at character ("@") as this is used to identify Agents in Interstage BPM.

2. You can retrieve all groups available in Interstage BPM using `getAllGroups()`.

```
Group[] LocalGroupList  = adminSession.getAllGroups();
```

3. Create the users you want to assign to the local groups in Interstage BPM. In this sample, the users Lisa, Sarah, Peter and John are created:

```
testUser = WFObjectFactory.getUser("UserSarah");
adminSession.createUser(testUser, "SomePassword");
testUser = WFObjectFactory.getUser("UserLisa");
adminSession.createUser(testUser, "SomePassword");
testUser = WFObjectFactory.getUser("UserPeter");
adminSession.createUser(testUser, "SomePassword");
testUser = WFObjectFactory.getUser("UserJohn");
adminSession.createUser(testUser, "SomePassword");
```

4. Assign the newly created users to the local group using `addUserToGroup()` from the `WFAdminSession` interface. In the sample, Lisa and Sarah are assigned to `US_Employees`, John and Peter are assigned to `UK_Employees`.

```
adminSession.addUserToGroup("UserSarah", "US_Employees");
adminSession.addUserToGroup("UserLisa", "US_Employees");
adminSession.addUserToGroup("UserPeter", "UK_Employees");
adminSession.addUserToGroup("UserJohn", "UK_Employees");
```

You can assign the same user to as many groups as you need.

You may assign any Interstage BPM user to any local group. The users to be assigned may be stored in Interstage BPM's local user store or in a Directory Service.

5. If you wish to create group hierarchies, use `addGroupToGroup()` to assign a child group to a parent group. In the sample, `US_Employees` and `UK_Employees` are associated with the `AllEmployees` group.

```
adminSession.addGroupToGroup("US_Employees", "AllEmployees");
adminSession.addGroupToGroup("UK_Employees", "AllEmployees");
```

When assigning a child group to a parent group, all members of the child group also become members of the parent group.

You can assign the same group to multiple groups.

Note: Do not create cyclic groups. For example, do not add `Group1` as a child group of `Group2` and `Group2` as a child group of `Group1`.

6. If a user shall no longer be a member of a group, you can remove the user from that group.

```
adminSession.removeUserFromGroup(sampleUser, roleName);
```

7.3.3 Listing Logged-In Users

To retrieve a list of all logged-in users:

- Use `getAllUserAgentInfo()` from the `WFAdminSession` interface.

```
UserInfo[] userInfoList = adminSession.getAllUserAgentInfo();
```

7.3.4 Logging Out Users

- **To log out a single user:**

  Use `logoutUser()` from the `WFAdminSession` interface.

- **To log out all users:**

  Use `logoutAllUsers()` from the `WFAdminSession` interface.

The following example shows how to log out all users from a given `userInfoList`, except the administrator:

```
if (userInfoList != null) {
   boolean loggedOut = false;
   for (int i = 0; i < userInfoList.length; i++) {
      // Control the name of the user to get a user which is not the
      // current administrator.
      if (!userInfoList[i].getName().equals(adminName)) {
         adminSession.logoutUser(userInfoList[i].
         getClientID());

         loggedOut = true;
      }
   }
}
```

7.3.5 Resetting the User and Group Cache

In an administration session, `WFAdminSession`, an administrator can reset the cached data for groups and users from the Directory Server.

At startup, the Interstage BPM Server retrieves the groups and users from the local or remote store and stores them in a cache. In a `WFAdminSession`, an administrator can reset the cached data for groups and users.

- **To reset the group list:**

  ```
  adminSession.resetLDAPGroupsList();
  ```

  `resetLDAPGroupsList()` clears the cache for groups immediately. It resets the cached memory object of the groups defined in the local or remote store. Any modifications to the group list in the local or remote store are reloaded into the memory object.

- **To reset the users list:**

  ```
  adminSession.resetLDAPUsersList();
  ```

  `resetLDAPUsersList()` clears the cache for users immediately. It resets the cached memory object of the users defined in the local or remote store. Any modifications to the users list in the local or remote store are reloaded into the memory object.

Note: When using a remote store, the group cache is automatically refreshed at a regular interval. The expiration date for the LDAP group cache is set in the `LDAPGroupCacheAgeSec` parameter of the Interstage BPM Server. Refer to the *Interstage Business Process Manager Server Administration Guide* for more information.

7.4 Process Definition Administration

This section describes the following common process definition administration tasks:

- Listing process definitions
- Publishing process definitions
- Archiving process definitions
- Deleting process definitions
- Deleting archived process definitions
- Importing a process definition from an XPDL file
- Exporting a process definition to an XPDL file

7.4.1 Listing Process Definitions

For listing process definitions, use the `WFObjectList` interface from the `com.fujitsu.iflow.model.workflow` package. The interface retrieves a list of the existing `WFObjects` from the Interstage BPM Server.

To list process definitions:

1. Build a filter using the `Filter` class.

   Possible filter criteria for process definitions are:

   - `AllArchivedPlans`: Retrieves all archived process definitions.
   - `AllPlans`: Retrieves all process definitions.
   - `MyInactivePlans`: Retrieves all inactive process definitions.
   - `MyPlans`: Retrieves all process definitions belonging to the logged in user.

2. Use `openBatchedList()`.
3. Iterate the list of returned objects matching the filter criteria set with `openBatchedList()`.

Example

```
Plan[] planList = null;
if (filter == Filter.AllArchivedPlans
   || filter == Filter.AllPlans
   || filter == Filter.MyInactivePlans
   || filter == Filter.MyPlans) {

      WFObjectList wfObjectList =
         WFObjectFactory.getWFObjectList(adminSession);

      wfObjectList.openBatchedList(filter);

      Vector tmpList = new Vector();
      int batchSize = 50;

      Object[] elements = wfObjectList.getNextBatch(batchSize);
      Plan plan = null;

      while (elements != null && elements.length > 0) {
         for (int i = 0; i < elements.length; i++) {

            plan = (Plan) elements[i];
            // Add process definition to temporary list
            tmpList.add(plan);
         }
      // Read next list of process definitions regarding to filter
      elements = wfObjectList.getNextBatch(batchSize);
   }
   // Copy the process instance of temporary list into a
   // ProcessInstance array that will be returned
   planList = new Plan[tmpList.size()];
   planList = (Plan[]) tmpList.toArray(planList);

   return planList;
}
```

7.4.2 Publishing Process Definitions

To publish a process definition:

- Use `publishPlan()` from the `WFAdminSession` interface.

When a process definition is published, its state changes to "published".

Example

The following example shows how to publish the first process definition from a list of process definitions (`planList`) that are in private or draft state.

```
if (planList != null ) {
   boolean published = false;
   for (int I = 0; I < planList.length; I++) {
      if (planList[i].getState() == Plan.STATE_DRAFT
         || planList[i].getState() == Plan.STATE_PRIVATE) {
         adminSession.publishPlan(planList[i].getId());
      }
   published = true;
   }
}
```

7.4.3 Archiving Process Definitions

When a process definition is archived, its state changes to "archived". If you try to archive a process definition that is already archived, Interstage BPM assumes that you want a published process definition to be archived. In this case, the process definition still exists in the regular process definition list having the state "deleted".

To archive a process definition:

- Use `archivePlan()` from the `WFAdminSession` interface.

```
adminSession.archivePlan(planList[i].getId());
```

Once you have archived a process definition, you can retrieve it again as follows:

1. Get the list of all archived process definitions for retrieving the IDs of the archived process definitions. Use the `AllArchivedPlans` filter criterion for doing so.
2. To retrieve a specific archived process definition, use `getArchivedPlan(id)` from the `WFAdminSession` interface.

```
adminSession.getArchivedPlan(planList[i].getId());
```

Note that you can only perform a subset of functions on archived process definitions, such as retrieving information on them. All functions that modify a process definition are not supported for archived process definitions (e.g. `setName`).

7.4.4 Deleting Process Definitions

An administrator can delete process definitions from all users. A user, however, can only delete a process definition of which he/she is the owner.

To delete a process definition:

- Use `deletePlan()` from the `WFAdminSession` interface.

```
adminSession.deletePlan(planList[i].getId());
```

The method selects a specified process definition from the database. The process definition must be in "draft" or "private" state. If the process definition is in any other state, its state changes to "deleted", but it will not be physically deleted from the database.

7.4.5 Deleting Archived Process Definitions

To delete an archived process definition from the database:

- Use `deleteArchivedPlan()` from the `WFAdminSession` interface.

```
adminSession.deleteArchivedPlan(archivedPlanList[i].getId());
```

7.4.6 Importing a Process Definition from an XPDL File

To import an XPDL file:

1. Create an empty process definition object with `WFObjectFactory.getPlan()`.
2. Set the current session to the process definition object with `setWFSession()`.
3. Create a process definition object from an XPDL file with `convertFromXPDL()`.

Example

For the following example to work properly, make sure that the defined import directory `DIR_PATH_IMPORT` contains some files.

```
File importDir = new File(DIR_PATH_IMPORT);

if (importDir.exists() && importDir.isDirectory()) {
   File[] fileList = importDir.listFiles();
   String filePath;
   boolean fileFound = false;
   Plan plan = null;

   for (int i = 0; i < fileList.length; i++) {
      filePath = fileList[i].getAbsolutePath();

      if (filePath.endsWith(".xml") ||
         filePath.endsWith(".xpdl")) {
         plan = WFObjectFactory.getPlan();
         plan.setWFSession(adminSession);
         plan.convertFromXPDL(new FileInputStream(fileList[i]));
         break;
      }
   }
}
```

7.4.7 Exporting a Process Definition to an XPDL File

You can convert a process definition to XPDL data using `convertToXPDL()` from the `Plan` interface. You need to extend the import declaration in the same way as when importing process definitions.

To export a process definition to an XPDL file:

1. Take the first process definition from a given list of process definitions.

```
if (planList != null && planList.length > 0) {
   Plan plan = planList[0];
   . . .
}
```

2. Check whether the destination directory exists. If not, a new directory is created.

```
File exportDir = new File(DIR_PATH_EXPORT);
   if (!exportDir.exists() || !exportDir.isDirectory()) {
      exportDir.mkdir();
}
```

3. Generate an absolute file name for the export file.

```
String fileName = DIR_PATH_EXPORT + "/" + "PLAN_" +
plan.getName() + "_" + plan.getId() + ".xml";
```

4. Create a new file object.

```
File file = new File(fileName);
if (!file.exists()) {
   file.createNewFile();
   . . .
}
```

5. Convert the process definition to XPDL.

```
plan.convertToXPDL(new FileOutputStream(file));
```

7.5 Process Instance Administration

This section describes the following process instance administration tasks:

- Listing process instances
- Changing the ownership of a process instance
- Archiving process instances
- Suspending process instances
- Aborting process instances
- Deleting process instances
- Deleting archived process instances

7.5.1 Listing Process Instances

For listing process instances, use the `WFObjectList` interface from the `com.fujitsu.iflow.model.workflow` package. This interface retrieves a list of the existing `WFObjects` from the Interstage BPM Server.

To list process instances:

1. Build a filter using the `Filter` class.

   Possible filter criteria for process instances are:

   - `AllActiveProcesses`: Retrieves all active process instances.
   - `AllArchivedProcesses`: Retrieves all archived process instances.
   - `AllInactiveProcesses`: Retrieves all inactive process instances.
   - `AllProcesses`: Retrieves all process instances.
   - `AllProcessesInErrorState`: Retrieves all inactive process definitions.
   - `MyActiveProcesses`: Retrieves all active process instances initiated by the logged in user.
   - `MyInactiveProcesses`: Retrieves all inactive process instances initiated by the logged in user.
   - `MyProcesses`: Retrieves all process instances initiated by the logged in user.

2. Use `openBatchedList()`.
3. Iterate the list of returned objects matching the filter criteria set with `openBatchedList()`.

Refer to the `Administration.java` sample source file located in your `<Interstage BPM Server Installation Directory>/client/samples/examples/sources` directory for the detailed sample code on how to list process instances.

7.5.2 Changing the Ownership of a Process Instance

Only an administrator and the current process instance owner can change the ownership of a process instance. Process ownership must be set at the process definition level. The process owner is determined by the setting of the attribute "Owner Role". The attribute "Owner Role" is evaluated by the Interstage BPM Server before activating a new process instance.

Use `setOwners()` from the `ProcessInstance` interface to set the owner of a running process instance. Before actually changing the ownership, this method checks whether the user or group who is to become the new owner exists; if one of the new owners does not exist, the process instance will go into error state.

In the following example, the ownership of the first process instance from a given list is assigned to the first found user from a given user list.

To change the ownership of a process instance:

1. List all existing user groups.

```
if (procInstList != null) {
   String[] owners = null;
   String role = "";
   DirectoryServices ds = WFObjectFactory.getDirectoryServices(
      adminSession, adminName, password);
   String[] groups = ds.getUserGroups();
   if (groups != null && groups.length > 0) {
      role = groups[0];
   }
   owners = ds.getUserList(role);
   if (owners != null) {
      boolean assigned = false;
      ...
   }
}
```

2. Take the first user to get the new owner for a process instance, and start the edit-mode.

```
for (int i = 0; i < procInstList.length; i++) {
   procInstList[i].startEdit();
      ...
   }
```

3. Set the owner to the process instance.

```
procInstList[i].setOwners(owners);
```

4. Commit the edit-mode.

```
procInstList[i].commitEdit();
assigned = true;
```

7.5.3 Archiving Process Instances

Only process instances that are completed, aborted or in error state can be archived. Therefore, before archiving a process instance, you have to check for the process instance's current state. You can identify the state of a completed process instance using the constant `STATE_CLOSED` from the `ProcessInstance` interface. Use the constants `STATE_ERROR` or `STATE_ABORTED` to check whether a process instance is in error state or has been aborted.

To archive a process instance:

1. Check a given list of process instances for process instances that are completed, aborted or in error state. To do so, check the current state of the process instance using `getState()` from the `ProcessInstance` interface.

```
 if (procInstList[i].getState() == ProcessInstance.STATE_CLOSED ||
  ProcessInstance.STATE_ABORTED || ProcessInstance.STATE_ERROR)
 {
   ...
}
```

2. To archive a process instance, use `archiveClosedProcess()` from the `WFAdminSession` interface.

```
adminSession.archiveClosedProcess(procInstList[i].getId());
```

Once you have archived a process instance, you can retrieve it again as follows:

1. Get the list of all archived process instances for retrieving the IDs of the archived process instances. Use the `AllArchivedProcesses` filter criterion for doing so.
2. To retrieve a specific archived process instance, use `getArchivedProcess(id)` from the `WFAdminSession` interface.

```
adminSession.getArchivedProcess(procInstList[i].getId());
```

Note that you can only perform a subset of functions on archived process instances, such as retrieving information on them. All functions that modify a process instance are not supported for archived process instances (e.g. `setName`).

7.5.4 Suspending Process Instances

When suspending a running process instance, its state changes to `STATE_SUSPENDED`. If this process instance has any work items, they are also suspended. If the process instance has any running subprocess instances, they are also suspended.

To suspend a process instance:

1. Check a given list of process instances for running process instances. To do so, check the current state of the process instance using `getState()` from the `ProcessInstance` interface.

```
 if (procInstList[i].getState() == ProcessInstance.STATE_RUNNING)
 {
    ...
}
```

2. To suspend a running process instance, use `suspend()` from the `ProcessInstance` interface.

```
procInstList[i].suspend();
```

Note: If defined on process definition level or Activity Node level, a set of Java Actions may be executed before the process instance changes to `STATE_SUSPENDED`. This set of Java Actions will become active as soon as you call the `suspend()` command. For example, a notification email is sent to the owner of the process instance, or additional log file entries are written. Refer to section *Using onAbort, onSuspend, onResume Java Actions* on page 109 for details on how to define an `onSuspendJavaActionSet`.

You can resume a suspended process instance using `resume()` from the `ProcessInstance` interface. The state of the process instance is then changed to `STATE_RUNNING`.

Again, if defined on process definition level or Activity Node level, a set of Java Actions may be executed before the process instance changes to `STATE_RUNNING` again. This set of Java Actions will become active as soon as you call the `resume()` command.

7.5.5 Aborting Process Instances

When aborting a process instance, its state changes to `STATE_ABORTED`. If this process instance has any work items, they are removed. If the process instance has any running subprocess instances, they are also aborted.

To abort a process instance:

- Use `abort()` from the `ProcessInstance` interface.

  This method can only be used by an administrator or the process instance owner. Other users are not allowed to abort a process instance.

Example

In the following example, a given list of process instances is checked for running (constant `STATE_RUNNING`) and suspended (constant `STATE_SUSPENDED`) process instances using `getState()` from the `ProcessInstance` interface. `getState()` returns the current state of the process instance. Then the first running or suspended process instance which is found inside the list is aborted.

```
if (procInstList != null ) {
   boolean aborted = false;
   for (int i = 0; i < procInstList.length; i++) {
      if (procInstList[i].getState() ==
         ProcessInstance.STATE_RUNNING
        || procInstList[i].getState() == ProcessInstance.STATE_SUSPENDED)
 {
            procInstList[i].abort();
            aborted = true;
      }
   }
}
```

Note: If defined on process definition level or Activity Node level, a set of Java Actions may be executed before the process instance changes to `STATE_ABORTED`. This set of Java Actions will become active as soon as you call the `abort()` command. For example, a notification email is sent to the owner of the process instance, or additional log file entries are written. Refer to section *Using onAbort, onSuspend, onResume Java Actions* on page 109 for details on how to define an `onAbortJavaActionSet`.

7.5.6 Deleting Process Instances

To delete a process instance:

- Use `deleteProcessInstance()` from the `WFAdminSession` interface.

  This method deletes a process instance from the database, using a process instance ID.

Example

In the following example the first valid process instance from a given list is deleted:

```
if (procInstList != null) {
   boolean deleted = false;
   for (int i = 0; i < procInstList.length; i++) {
      adminSession.deleteProcessInstance
      (procInstList[i].getId());
      deleted = true;
      break;
   }
}
```

7.5.7 Deleting Archived Process Instances

To delete an archived process instance from the database:

- Use `deleteArchivedProcess()` from the `WFAdminSession` interface.

```
adminSession.deleteArchivedProcess(archivedProcInstList[i].getId());
```

7.6 Work Item Administration

This section describes the following work item administration tasks:

- Reassigning work items from one user to another
- Refreshing work items

7.6.1 Reassigning Work Items to Another User

To reassign a work item to another user:

- Use `reassign()` from the `WFSessionAdmin` interface.

Example

The following example shows how to reassign work items from one user to another existing user.

```
if (workItemList != null && workItemList.length > 0) {
   WorkItem workItem = workItemList[0];
   UserInfo[] userList = listUsers();
   String oldAssignee = "";
   String newAssignee = "";
   if (userList != null && userList.length > 0) {
      oldAssignee = workItem.getAssignee();
      newAssignee = userList[0].getName();
      adminSession.reassign(oldAssignee, newAssignee,
      workItem.getProcessInstanceId(),
      workItem.getId(),false);
   }
}
```

7.6.2 Refreshing Work Items

If an administrator adds or deletes a user from a group in the directory, any running activities that have that group as an assignee will not be automatically updates, and the outstanding workitems will not be created or removed to match the group. In general operation, workitems are created to match the group as it is at the time that the activity is started, and changes in the group after that are purposefully ignored. In some situations, however, one would like the workitems of an activity that is already started to be altered to match the changed group members. To update the workitems, the administrator has to refresh the work items.

Refreshing the work items can take some time because often several iterations are necessary to complete the task. The time needed depends on the number of the active tasks in the database. The execution time can be reduced by using options for selecting the work items to be refreshed:

- The work items of a process instance.
- The work items of all process instances that belong to a process definition.
- The work items for a group.
- All existing work items.
- **To refresh the work items of a process instance:**

```
boolean ignoreActivityWithRoleScript = true;
String refreshResult = null;

if (procInst != null) {
   long procInstId = procInst.getId();
   refreshResult = adminSession.refreshWIforProcess(procInstId,
      ignoreActivityWithRoleScript);
}
```

- **To refresh the work items of all process instances that belong to a process definition:**

```
if (procDef != null) {
   long procDefId = procDef.getId();
   refreshResult = adminSession.refreshWIforPlan(procDefId,
      ignoreActivityWithRoleScript);
}
```

- **To refresh the work items for a group:**

```
String group = "SampleGroup";
refreshResult = adminSession.refreshWIforGroup(group,
   ignoreActivityWithRoleScript);
```

- **To refresh all work items:**

```
refreshResult = adminSession.refreshWorkItems
   (ignoreActivityWithRoleScript);
```

Note: If the `SupportGroupWorkItem` parameter of the Interstage BPM Server is set to `true`, the filter `Filter.MyActiveWorkItems` is not supported, i.e. the work items cannot be retrieved with this filter.

In this case you have to define your own filter for retrieving the work items to be refreshed. Refer to section *Listing Work Items* on page 81 for details on building a filter for work items. Refer to the *Interstage Business Process Manager Server Administration Guide* for details on the `SupportGroupWorkItem` parameter.

The result of the refreshing is formatted as an XML string, which lists the successful and the failed operations. The following gives an example of such an XML string:

```
<?xml version="1.0" encoding="UTF-8" ?>
   <Processes>

      <success>
         <Process Name="procName" id="123"></Process>
      </success>

       <failed>
         <Process Name="procName" id="456" Status="ignored"
            message="error message" />
         <Process Name="procName" id="123">
            <Activities>
               <Activity Id="123456" Name="activityName"
                  Status="ignored" message="error message" />
            </Activities>
         </Process>
      </failed>
   </Processes>
```

The `<success>` tags contain a list of process instances for which the refreshing of the work items was successful. The `<failed>` tags contain a list of process instances for which the refreshing failed. The `<Activities>` tags contain activities of a process instance for which the refreshing failed.

8 Retrieving History Information

For each process instance, the information concerning the workflow is stored in an Interstage BPM database called `teamflowdb`. In the History table of the `teamflowdb` database, each single step in a workflow of a process instance is stored.

You have the following possibilities to retrieve history information from the `History` table:

- With the Model API
- From a Hashtable
- With SQL Statements

8.1 Retrieving History Information With the Model API

To retrieve history information from a process instance using the Model API:

1. Use the following code as basis:

```
import com.fujitsu.iflow.model.workflow.ProcessInstance;
import com.fujitsu.iflow.model.util.IflowEnumeration;

int processHistoryFields[] = {
   ProcessInstance.HISTORY_ID,
 ProcessInstance.HISTORY_TIME_STAMP,
   ProcessInstance.HISTORY_EVENT_CODE,
   ProcessInstance.HISTORY_EVENT_TYPE,
   ProcessInstance.HISTORY_RESPONSIBLE,
   ProcessInstance.HISTORY_PRODUCER_TYPE,
   ProcessInstance.HISTORY_PRODUCER_ID,
   ProcessInstance.HISTORY_CONSUMER_TYPE,
   ProcessInstance.HISTORY_CONSUMER_ID,
   ProcessInstance.HISTORY_PROCESSINSTANCE_ID,
   ProcessInstance.HISTORY_ISHANDLED_CODE,
   ProcessInstance.HISTORY_EVENTDATA
};

ProcessInstance procInst =
   WFObjectFactory.getProcessInstance(procID, session);

IflowEnumeration sElements =
   processInstance.getHistory(processHistoryFields);
```

   `getHistory()` retrieves all the history data from the related process instance. The columns that are to be retrieved are defined by the integer array, which is passed as argument to the method.

2. Navigate through the resulting objects of type `IflowEnumeration` to access each row for the history entries.

```
while(sElements.hasMoreElements())
   {
      Hashtable htblEvent = (Hashtable) sElements.nextElement();
      // Here the code from the section "Retrieving the History
Information From a Hashtable"
   }
```

The following table shows the mapping of the constants in the `ProcessInstance` class to the columns of the `History` table.

Constants in ProcessInstance Class	Name of Column
HISTORY_ID	historyID
HISTORY_TIME_STAMP	CreatedTime
HISTORY_EVENT_CODE	EventCode
HISTORY_EVENT_TYPE	EventType
HISTORY_EVENTDATA	EventData
HISTORY_PRODUCER_TYPE	ProducerType
HISTORY_PRODUCER_ID	ProducerId
HISTORY_CONSUMER_TYPE	ConsumerType
HISTORY_CONSUMER_ID	ConsumerId
HISTORY_ISHANDLED_CODE	IsHandled
HISTORY_PROCESSINSTANCE_ID	ProcessInstanceId
HISTORY_RESPONSIBLE	Responsible

Note: There is a constant defined to retrieve the column `ServerName`.

8.2 Retrieving History Information From a Hashtable

With the code in section *Retrieving History Information With the Model API* on page 203, you can retrieve all history data from a process instance. The results are stored in a hashtable. The following sections show how to retrieve the values for the single constants of the `ProcessInstance` class.

8.2.1 HISTORY_ID

To retrieve the history ID of an entry, use the following code:

```
long historyId = ((Long)htblEvent.get
   (new Integer(ProcessInstance.HISTORY_ID))).longValue();
```

Each entry in the `History` table has a unique identifier.

8.2.2 HISTORY_TIME_STAMP

To retrieve the timestamp of an entry, use the following code:

```
long timestamp = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_TIME_STAMP))).longValue();
```

To convert the long value of the date into a readable form, use standard Java statements.

8.2.3 HISTORY_EVENT_CODE

To retrieve the event code of an entry, use the following code:

```
long historyEventCode = ((Long) htblEvent.get
    (new Integer(ProcessInstance.HISTORY_EVENT_CODE)).longValue();
```

The event code indicates the type of the event. The following table lists possible values and their meaning:

Event Code	Description
0	Start (Default)
1	Activate
2	Make choice
3	Accept
4	Decline
5	Reassign
6	Exit
7	Create subprocess
8	Suspend work item
9	Resume work item
10	Start work
11	Stop work
12	Voting complete
13	Process created
14	Process migrated
15	Process recall
16	Deactivate target node
17	Activate source node
25	Generate future work items

8.2.4 HISTORY_EVENT_TYPE

To retrieve the event type of an entry, use the following code:

```
String eventType = (String) htblEvent.get(
   new Integer(ProcessInstance.HISTORY_EVENT_TYPE));
```

The event type specifies the name of an activity or the name of an arrow in Interstage BPM. If, for example, an arrow has the name `Create Purchase Requisition`, the result of `HISTORY_EVENT_TYPE` is `Create Purchase Requisition`.

8.2.5 HISTORY_EVENTDATA

To retrieve additional data about an event, use the following code:

```
String eventData = (String) htblEvent.get(
   new Integer(ProcessInstance.HISTORY_EVENTDATA));
```

The retrieved value is in most cases `NULL`.

8.2.6 HISTORY_PRODUCER_ID

To retrieve the producer ID of an entry, use the following code:

```
long producerId = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_PRODUCER_ID))).longValue();
```

Depending on the producer type, the producer ID specifies different types of information. Refer to section *History Table* on page 209 for more details.

8.2.7 HISTORY_PRODUCER_TYPE

To retrieve the producer type of an entry, use the following code:

```
long lProdType=((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_PRODUCER_TYPE))).longValue();
```

The producer type indicates the event responsible for creating the current event. The following table lists possible values and their meaning:

Producer Type	Description
0	Arrow
3	Activity
7	Process
15	Timer

8.2.8 HISTORY_CONSUMER_ID

To retrieve the consumer ID of an entry, use the following code:

```
long consumerId = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_CONSUMER_ID))).longValue();
```

Depending on the consumer type, the consumer ID specifies different types of information. Refer to section *Rules for Navigating Through the History Entries* on page 208 for details.

8.2.9 HISTORY_CONSUMER_TYPE

To retrieve the consumer type of an entry, use the following code:

```
long lConsType = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_CONSUMER_TYPE))).longValue();
```

The consumer type indicates the type of the element that the event is intended for. The following table lists possible values and their meaning:

Consumer Type	Description
3	Activity
7	Process

For more information on how to interpret the consumer type, refer to section *Rules for Navigating Through the History Entries* on page 208.

8.2.10 HISTORY_ISHANDLED_CODE

To retrieve the value `IsHandled` of an entry, use the following code:

```
long isHandledCode = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_ISHANDLED_CODE)))
   .longValue();
```

The following table lists possible values and their meaning:

isHandled Code	Description
0	Not handled
1	Handled
2	Ignored
3	Error
4	Audit
5	Suspended

8.2.11 HISTORY_PROCESSINSTANCE_ID

To retrieve the process instance ID of an entry, use the following code:

```
long procInstId = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_PROCESSINSTANCE_ID)))
   .longValue();
```

8.2.12 HISTORY_RESPONSIBLE

To retrieve the responsible for the entry in the `History` table, use the following code:

```
String responsible = (String) htblEvent.get(
   new Integer(ProcessInstance.HISTORY_RESPONSIBLE));
```

Possible values are:

- `__process`

  System user ID to indicate the activity of the workflow engine.

- `<userID>`

Any valid user ID to indicate the user who performed an action.

8.2.13 Rules for Navigating Through the History Entries

To navigate through the entries in the `History` table of a process instance, the following rules are defined.

Rule 1

To navigate through the entries of the `History` table, only entries fulfilling one of the three following conditions are evaluated:

- **Activation event**

  Applies if all of the following criteria are met:
  - `ConsumerType` is defined as `3` (Activity)
  - `EventType` is equal to `__Activate`
- **Activity Node**

  Applies if all of the following criteria are met:
  - `ConsumerType` is defined as `3` (Activity)
  - `EventType` is not equal to `__Activate`
  - `ProducerType` is defined as `7` (Process)
- **Arrow**

  Applies if all of the following criteria are met:
  - `ConsumerType` is defined as `3` (Activity)
  - `EventType` is not equal to `__Activate`
  - `ProducerType` is defined as `0` (Arrow)

Rule 2

In case there is an entry of type 1, i.e. an activation event, only the timestamp is retrieved.

Rule 3

In case there is an entry of type 2, i.e. an Activity Node, the consumer ID represents the ID of the node instance in the process instance. Using the Model API, you can retrieve the node instance as follows:

```
// CONSUMER_ID represents the node instance ID

   long consumerId = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_CONSUMER_ID))).longValue();

   NodeInstance nodeInst = procInst.getNodeInstance(consumerId);

   // Now continue to retrieve the information from the object
   // of the class NodeInstance
```

Rule 4

In case there is an entry of type 3, i.e. an arrow, the producer ID represents the ID of the arrow instance in the process instance. Using the Model API, you can retrieve the node instances of source and target nodes as follows:

```
// PRODUCER_ID represents the arrow instance ID

   long producerId = ((Long)htblEvent.get(
   new Integer(ProcessInstance.HISTORY_PRODUCER_ID))).longValue();

   ArrowInstance arrow = procInst.getArrowInstance(producerId);

   NodeInstance sourceNodeInst = arrow.getSourceNodeInstance();
   NodeInstance targetNodeInst = arrow.getTargetNodeInstance();

   // Now continue to retrieve information from the source and
   // target objects of the arrow instance
```

8.3 History Table

In the `History` table, each step in the workflow of a process instance is defined by a separate entry, i.e. a row. A step can be defined by a system action, for example "Creation of a process instance", or by the execution of an activity which is explicitly defined in the process definition, for example "Vote for Approval".

The content of the `History` table is used to export workflow information from Interstage BPM to ARIS Process Performance Manager. For more information, refer to the *Interstage Business Process Manager ARIS PPM Integration Guide*.

Note: The database schema described in this section could be changed in a future version of Interstage BPM. If you develop your own application based on the database schema, you need to change the application when the database schema is changed.

Field	Value/Sample	Remarks
`historyID`	`888919`	Unique identifier for each row in the `History` table.

Field	Value/Sample	Remarks

`EventCode`	`0=Start (default)`
	`1=Activate`
	`2=MakeChoice`
	`3=Accept`
	`4=Decline`
	`5=Reassign`
	`6=Exit`
	`7=CreateSubProcess`
	`8=SuspendWorkItem`
	`9=ResumeWorkItem`
	`10=StartWork`
	`11=StopWork`
	`12=VotingComplete`
	`13=ProcessCreated`
	`14=ProcessMigrated`
	`15=ProcessRecall`
	`16=DeactivateTargetNode`
	`17=ActivateSourceNode`
	`25=GenerateFutureWorkItems`
		Each code uniquely identifies a separate event.

`EventType`	`"Approve"`
	or
	`"Submit Purchase Order:99198"`
		Name of the event.

`EventData`	`NULL`	Additional information, but in most cases `NULL`.

`CreatedTime`	`"2004-11-12 13:19:12.820"`
	or
	`"22.12.2004 16:48"`
		Timestamp (string format) of the creation of this activity.

`Responsible`	`"jim"`
	or
	`"__process"`
		Name of the user or system who was responsible for creating the event. If there is no User Group assigned to the node, the system executes the node and the name `"__process"` is taken.

Field	Value/Sample	Remarks

`IsHandled`	`0=notHandled`
	`1=handled`
	`2=ignored`
	`3=Error`
	`4=audit`
		State of the event.

`ProducerType`	`0: arrow`
	`7: node`
		Element responsible for creating the current event. Either this is an arrow or a node.

`ProducerId`	`99130`	If `ProducerType = 0`, the `ProducerId` denotes the arrow instance ID of the current event. Using the arrow instance ID, the source and target node instances of the event can be retrieved.

`ConsumerType`	`0=ArrowType`
	`3=ActivityType`
	`7=ProcessType`
		The element type that the event is intended for.

`ConsumerId`	`99116`	If `ProducerType = 7` the `ConsumerId` denotes the node instance ID of the current node. `If ProducerType = 0` the `ConsumerId` denotes the node instance ID of the target node. The same node instance can be retrieved by the arrow instance.
`ProcessInstanceId`	`88892`	Unique identifier of a process instance.
`ServerName`	`localhost`	The name of the server for which these events are generated and handled.
`applicationContainerId`	`2`	Unique identifier of the application that contains the process instance

8.4 Retrieving History Information With SQL Statements

You can retrieve history information using SQL statements on the `teamflowdb` database.

- **To get an overview about all stored data in the History table:**

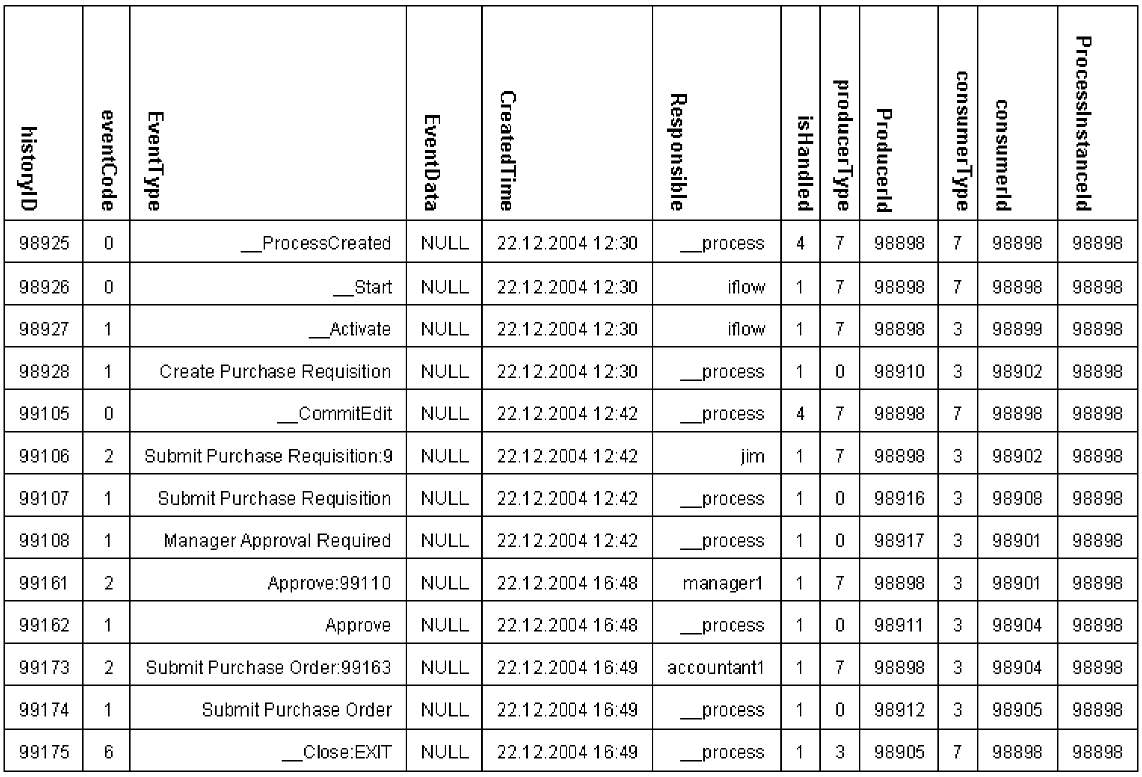
  ```
  SELECT * FROM HISTORY
  ```

- **To retrieve all rows linked to a specific process instance:**

  ```
  SELECT * FROM HISTORY WHERE PROCESSINSTANCEID=<PROCID>
  ```

  Replace `<PROCID>` by the requested ID (number) of the process instance.

The following figure shows as an example all the entries in the `History` table, assigned to a closed process instance.



historyID	eventCode	EventType	EventData	CreatedTime	Responsible	isHandled	producerType	ProducerId	consumerType	consumerId	ProcessInstanceId
98925	0	__ProcessCreated	NULL	22.12.2004 12:30	__process	4	7	98898	7	98898	98898
98926	0	__Start	NULL	22.12.2004 12:30	iflow	1	7	98898	7	98898	98898
98927	1	__Activate	NULL	22.12.2004 12:30	iflow	1	7	98898	3	98899	98898
98928	1	Create Purchase Requisition	NULL	22.12.2004 12:30	__process	1	0	98910	3	98902	98898
99105	0	__CommitEdit	NULL	22.12.2004 12:42	__process	4	7	98898	7	98898	98898
99106	2	Submit Purchase Requisition:9	NULL	22.12.2004 12:42	jim	1	7	98898	3	98902	98898
99107	1	Submit Purchase Requisition	NULL	22.12.2004 12:42	__process	1	0	98916	3	98908	98898
99108	1	Manager Approval Required	NULL	22.12.2004 12:42	__process	1	0	98917	3	98901	98898
99161	2	Approve:99110	NULL	22.12.2004 16:48	manager1	1	7	98898	3	98901	98898
99162	1	Approve	NULL	22.12.2004 16:48	__process	1	0	98911	3	98904	98898
99173	2	Submit Purchase Order:99163	NULL	22.12.2004 16:49	accountant1	1	7	98898	3	98904	98898
99174	1	Submit Purchase Order	NULL	22.12.2004 16:49	__process	1	0	98912	3	98905	98898
99175	6	__Close:EXIT	NULL	22.12.2004 16:49	__process	1	3	98905	7	98898	98898

Figure 24: Entries in the History Table

Appendix A: Using the Interstage Business Process Manager Samples

The samples in this manual demonstrate various ways to program Interstage BPM with the Model API. After installing Interstage BPM Server, you find the following subdirectories in your `<Interstage BPM Server Installation Directory>` directory (typically `C:\Fujitsu\InterstageBPM` on Windows and `/opt/FJSVibpm` on UNIX or Linux):

`/client/samples/examples/sources`

`/client/samples/examples/classes`

`/client/samples/examples/bin`

The contents of each subdirectory is described in detail below.

Note: To use sample programs in SaaS mode, set the appropriate value for the `TenantName` property in `iFlowClient.properties` available in the `<InterstageBPM installation directory>/client/` directory.

A.1 /client/samples/examples/sources

This directory contains the Java source code of the examples used throughout this manual. To see how the examples work, read the samples' source code found in the `/client/samples/examples/sources` subdirectory of your Interstage BPM Server installation directory. In the following sections you find a list of the available sample sources and information on where to find additional explanations in this manual.

A.1.1 Samples Related to Process Modeling and Execution

SimplePlan.java

This sample creates a simple Interstage BPM process definition.

Additional information can be found in section

Designing a Simple Process Definition on page 63

ComplexPlan.java

This sample creates a complex Interstage BPM process definition and shows, for example, how to add User Defined Attributes and Voting Activity Nodes, use Java Actions, etc.

Additional information can be found in sections

Designing a Complex Process Definition on page 68

Using Java Actions on page 90

Advanced Filtering & Sorting API on page 110

Special User Defined Attribute Properties on page 118

Using Extended Attributes on page 121

Transaction Control on page 128

ProcessExecution.java

This sample shows how to start a new process instance and execute work items.

Additional information can be found in sections

Working with Process Instances on page 78

Transaction Control on page 128

Note: This sample contains an `INSTALL_ROOT` constant containing the path to your Interstage BPM Server installation directory. By default, the path value is set to `C:\Fujitsu\InterstageBPM`. You need to adjust this value in case you installed Interstage BPM Server in a different location or you use UNIX or Linux.

SampleJavaActions.java

This sample contains methods which are used for Java Actions in process definitions like the one created with the `ComplexPlan.java`, for example, adding an attachment and sending an email.

Additional information can be found in section *Using Java Actions* on page 90.

BatchDetailsRetrieval.java

This sample shows how to work with the bulk processing capabilities provided by the `WFDetailsList` interface.

Additional information can be found in section *Retrieving Information about Multiple Objects at a Time* on page 114.

TriggerTimerSample.java

This sample demonstrates the use of triggers, timers, and business calendars. It creates a process definition that is started by a trigger.

Additional information can be found in sections

Using Triggers on page 129

Using Timers on page 149

EventActivityNodeSample.java

This sample shows the typical steps involved in working with Event Activity Nodes.

Additional information can be found in section *Using Event Activity Nodes* on page 133.

RemoteSubProcess.java

This sample shows how to define and use a remote subprocess. It creates a process definition containing a Remote Subprocess Node and another process definition which is used as remote subprocess.

Additional information can be found in section *Modeling Remote Subprocesses* on page 161.

AgentSimulator.java and AgentSimulator.xpdl

`AgentSimulator.xpdl` is a process definition that models a simple bank loan approval process. `AgentSimulator.java` is an example of an Agent Class, which simulates a loan decision maker. For more information, refer to the comments in the `AgentSimulator.java` source file and to section *Using Agents* on page 140.

HTTPAgentExample

The `HTTPAgentExample` subdirectory contains a pre-configured HTTP Example for implementing an HTTP Agent. The example agent demonstrates the complete functionality of an HTTP Agent. For detailed information, refer to the `HTTPAgentExampleInstructions.txt` file and to section *Using HTTP Agents* on page 149.

A.1.2 Samples Related to System Administration

Administration.java

This sample demonstrates various administrative activities, such as listing users, logging out users, resetting data from a user directory, publishing and archiving process definitions, etc.

Note: This sample works with existing process definitions and process instances. Make sure that all process definitions are valid and can be executed. In addition, be aware that this sample starts new instances and changes the status of existing instances.

Additional information can be found in chapter *Administration* on page 186.

SampleLocalUserManagement.java

This sample demonstrates local user and group administration, such as creating local users and groups, assigning users to groups, associating groups with other groups, etc.

Additional information can be found in section *User and Group Administration* on page 188.

A.1.3 Samples Related to the JMS Interface

Note: The JMS interface functionality can be used only in the non-SaaS mode.

The JMS Interface is an interface to the Interstage BPM Server like the Model API. With the Model API, you make method calls to model objects to perform all Interstage BPM operations. The Model API remains the primary interface to the Interstage BPM Server, but as an option, you can interact more directly with it using the JMS Interface. The JMS Interface functionality is, however, limited to certain key Interstage BPM operations.

Two topics are basic to the JMS Interface:

- The Interstage BPM command topic
- The Interstage BPM response topic

A JMS Client posts request messages to the command topic. This message contains an Interstage BPM command and the information necessary to execute it. The command topic is configured to accept requests from JMS Clients, and they are transferred to the Interstage BPM Server for processing. The JMS API executes the requested action on the Interstage BPM Server and retrieves a response. The response message (to all properly formatted requests) contains the requested information and is posted to the response topic. The information is then transmitted to the JMS Client.

Request and response JMS messages are XML that must be formatted according to `iFlowJMS.xsd`, the Interstage BPM JMS Interface schema. You can find the `iFlowJMS.xsd` and related documents in the `<Interstage BPM Installation Directory>/client/samples/doc/jms` directory.

If you want to use the JMS Interface, perform the following steps:

1. **Set up a JMS Client**:
   - Gather the Interstage BPM information that you will need to write the XML for the Interstage BPM commands that you want to run through the JMS Interface.

     To determine the required information, use the Interstage BPM JMS Interface schema `iFlowJMS.xsd` and related documents as a guideline.
   - Write XML for the Interstage BPM commands that you want to run.

     Again, use the Interstage BPM JMS Interface schema and related documents as a guideline. Optionally, the XML can specify a response topic different than the default response topic created by the Interstage BPM installation.
   - Set up your JMS Client to post the XML you have just written to the command topic. Use the samples as a guide for `PATH` and `CLASSPATH` settings. Be sure to add the application server-specific client properties files to your `CLASSPATH`.
   - You are now ready to post the XML that you have written to the command topic.
2. **Post requests to the command topic**:

   There are various JMS related samples available with your Interstage BPM installation in the `<Interstage BPM Installation Directory>/samples/examples/source` directory. These sample files can be recognized by their `Jms*.java` file names.

Note: The JMS interface of Interstage BPM does not provide the functionality to create a process definition. Therefore you find a separate sample, `createPlan.java` which shows how to create a process definition.

Refer to section */client/samples/examples/bin* on page 218 for instructions on how to compile and execute the Interstage BPM samples.

List of Available Samples Related to the JMS Interface

Note: Some of the JMS samples refer to the Server properties file, `server.properties`, located in the `<Interstage BPM Installation Directory>/samples/configuration`. This file is used for reading and/or updating the following attributes: `planId`, `processId`, `workitemId`, `choiceName`, or `UdaName`. If you do not use the process definition created by the `createPlan` sample file, you can update this file before executing the corresponding JMS sample.

For example: If you are executing the `JmsAcceptWorkItem` sample, the `workitemid` specified in `server.properties` should be valid for it to work. `JmsGetMyWorkItems` updates the `server.properties` with the appropriate `workitemid`. Thus you need to execute `JmsGetMyWorkItems` before `JmsAcceptWorkItem` so that the `server.properties` file is updated with the appropriate `workitemid`.

Note: The samples related to the JMS interface must be executed in a specific sequence, as each program uses the workflow objects generated by previous programs. For example, `createPlan` must be executed before executing `JmsGetMyPlans`.

JmsGetMyPlans	Retrieves a list of process definitions owned by the logged-in user.
JmsGetPlanState	Retrieves the state of a process definition.

JmsGetPlanUDA	Retrieves a UDA using the process definition.
JmsCreateProcess	Creates a process instance.
JmsStartProcess	Starts a process instance.
JmsGetMyProcesses	Retrieves a list of process instances owned by the logged-in user.
JmsGetProcessState	Retrieves the state of a process instance.
JmsGetProcessUDA	Retrieves a UDA using the process instance.
JmsGetMyWorkItems	Retrieves a list of work items assigned to the logged-in user.
JmsAcceptWorkItem	Accepts a work item.
JmsGetWorkItemState	Retrieves the state of a work item.
JmsDeclineWorkItem	Declines a work item. **Note:** Before executing this sample, execute the following samples in the shown sequence: 1. `JmsCreateProcess` 2. `JmsStartProcess` 3. `JmsGetMyWorkItems`
JmsGetWorkItemUDA	Retrieves a UDA using the work item.
JmsMakeChoice	Performs a make choice operation. **Note:** Before executing this sample, execute the following samples in the shown sequence: 1. `JmsCreateProcess` 2. `JmsStartProcess` 3. `JmsGetMyWorkItems`
JmsPublishPlan	Publishes a process definition.
JmsObsoletePlan	Sets a process definition to the state "Obsolete".
JmsSuspendProcess	Suspends the execution of a process instance.
JmsResumeProcess	Resumes the execution of a process instance.
JmsAbortProcess	Aborts the execution of a process instance.

A.1.4 Samples Related to Decision Tables

DecisionTableAPI.java

This sample demonstrates various activities related to Decision Tables management, such as creating a decision table, creating condition criteria, result locators, decisions, validating and evaluating a decision table, saving a decision table, and loading a decision table file. Additional information can be found in section *Decision Tables* on page 171.

A.2 /client/samples/examples/classes

The `/client/samples/examples/classes` subdirectory contains the compiled class files for the samples.

A.3 /client/samples/examples/bin

The `/client/samples/examples/bin` subdirectory contains two batch files:

- `StartSamples.bat`: Starts the execution of a sample file. Use the `StartSamples.sh` file on UNIX or Linux.

  You need to edit this batch file and set the directory where your application-server specific `jar` files are stored for the `APP_SERVER_JARS` variable BEFORE you run it.

  In addition, make sure that the `CLASSPATH` variable points to the correct directory where the following files are located:

  - `iflow.jar`
  - When using Interstage Application Server: `fujitsu-ibpm-engine-ejb_jar_client.jar`
  - Any application-server specific `jar` files
  - J2SE Development Kit (JDK)

  Refer to the `StartSamples.bat` or `StartSamples.sh` file for a sample path for Weblogic and JBoss.

  When you execute the batch file, you need to specify the sample file name (see below), the name of a user with administrator rights and his/her password, as well as a the name of the Interstage BPM Server, which must be the same as mentioned in the `IBPMPROPERTIES` table or the `ibpm.properties` file.

- `BuildSamples.bat`: Builds the class files for the Java source files. Use the `BuildSamples.sh` file on UNIX or Linux.

  Again, make sure that the `APP_SERVER_JARS` and the `CLASSPATH` variables point to the correct directory.

Appendix B: Customizing Forms

A QuickForm is a standard HTML page that enables the display of process data in Interstage BPM Clients and provides the ability to change that data. The Interstage BPM engine does not constrain you to any particular form technology. This means that if you have a favorite form support technology, you should be able to attach those forms to an activity, and your customization should be able to provide a client that displays the data through that form.

QuickForms are generated by Interstage BPM Studio and by the Process Designer that is part of the Interstage BPM Console. For information on generating QuickForms using these tools, refer to the *Interstage Business Process Manager Studio User's Guide* and *Interstage Business Process Manager User's Guide*.

This appendix describes how to customize the generated QuickForms. QuickForms can be edited with any HTML editor because they are standard HTML files. You simply need to understand a little bit about the components that are placed into the QuickForm and the effect that they have on the final produced web page. This appendix does not cover how to use specific Web-authoring tools.

B.1 QuickForm Command Overview

When you look at a generated form, you will see that certain components that appear on the form are represented by form tags. These tags have been generated within the HTML at the points where UDA values are expected to appear. These component tags start with two open brace characters and are concluded with two close brace characters. The double-braces tell the Interstage BPM Console that what lies inside is a directive telling what value to place there. Brace characters are not special characters in HTML, they do not require any special handling, so your HTML editor should treat them as normal text that you can type and edit.

The following is a brief overview of the components that appear in a QuickForm; all of them are optional:

- **`{{WorkItemHeader}}`**: This token will generate the top of the form to look the same as the rest of the Console pages, including the navigation tabs and other buttons that appear. Use this when you want the form to look the same as the Console. Without this, you have free capability to make the page look any way you want to.
- **`{{ProcessPanel}}`**: The process panel displays details about the current state of the work item including who it is assigned to, who started the process, when it was assigned, when it is due to be completed, what the choices are, and buttons to accept, decline, or reassign the work item.
- **`{{AttachmentPanel}}`**: This panel lists the documents currently attached to the process, and provides buttons for adding, removing or editing them.
- **`{{control type="Style" uda=udaIdentifier}}`**: This is the main way to put the value of a UDA into the resulting web page. The value will be properly encoded for HTML, which means that the ampersands and angle brackets will be properly replaced with the corresponding HTML entities. This is used for displaying a UDA value, or for passing as an attribute of an HTML input form element.

  When creating the form using the Interstage BPM Studio or Console, you can choose one of the following styles for each UDA - depending on its data type:

Style	BigDecimal, Float, Integer, Long	Boolean	Date	String	Comment
Default	x	x	x	x	The HTML form element that will be generated depends on the data type of the selected UDA. For example, for a UDA of type Date, the Default form element will be a Calendar control. For a UDA of type STRING, the default will be a text area.
Hidden	x	x	x	x	Generates a hidden form element in the web page. This is useful if you are including some JavaScript in the page that will read from the hidden element, and update to the hidden element for posting back to the server.
Text Field	x	-	-	x	Generates a text input field. By default, this field has a width of 10 characters allowing for the entry of a maximum of 10 characters. You can change this default setting.
Read Only	x	x	x	x	Generates a read-only field. By default, this field has a width of 10 characters. You can change this default setting.
Radio Button	x	x	-	x	Generates a radio button for the UDA value. By default, there is one radio button for the selected UDA with a default a value. You can add additional value names and assign default values.
Combo Box	x	x	-	x	Generates a combo box for either choosing or entering a UDA value. By default, there is one entry available for the selected UDA with a default a value. You can add additional value names and assign default values. These can then later be chosen from a list of entries.
Calendar	-	-	x	-	Generates a calendar control for the web page. The default date selected on the caledar is the current date.
Password	-	-	-	x	Generated a text input field in for entering passwords, i.e. the input of a user cannot be read, dots will be displayed for each entered character. By default, this field has a width of 10 characters allowing for the entry of a maximum of 10 characters. You can change this default setting.

Style	BigDecimal, Float, Integer, Long	Boolean	Date	String	Comment
Text Area	-	-	-	x	Generates an area for text input. By default, this text area consists of four columns and two rows. You can change this default setting.
Checkbox	-	-	-	x	Generates a check box for the UDA value. By default, there is one check box for the selected UDA with a default a value. You can add additional value names and assign default values.
List	-	-	-	x	Generates a list for displaying and selecting UDA values. By default, there is one entry for the selected UDA with a default a value. You can add additional value names and assign default values. In addition, multiple selection from the list is enable, and the list will consist of two rows by default. You can change this setting as well.

The following is an example of how you would put the value of the UDA with the ID 'PurchaseAmount' as Text Field into a form:

```
{{control type="TEXT" uda="PurchaseAmount" value="" maxchar="" maxwidth=""}}
```

- **`{{DefaultUDAsPanel}}`**: Rarely used, this token will generate form input elements for every UDA in the process automatically at run time. You might use this if you had a form that was to be used for many different processes with different schema. This is unusual because normally you want to use the form purposefully to control which variables the user sees.

In general you can then lay out your HTML page in any way you wish. You may wish to change the appearance of the form so that there are multiple columns, add a border, add color, etc. You may include graphics at any place in the page. JavaScript may be used to provide interactions, including data validation. Java applets may also be used within the page. When the QuickForm is being used, it has full control of every aspect of the page.

B.2 QuickForm Token Command Parameter Details

`{{ProcessPanel params}}` This generates a hidden tag to carry the work item id, and another hidden tag to carry the process instance sequence id. It also generates a number of sections, and each section is controlled by the parameter. The parameter is a comma-delimited list of identifiers, each identifier turns on a single process panel section. For example, if you only want the due date and the choices to appear in the panel, then you might specify:

`{{ProcessPanel due,choices}}`

The following is a list of the identifiers that may be used, and their effect:

`activity` – specifies that the top of the panel should be included which displays the name of the activity.

`from` – displays the second line of the panel, the “From” line, which displays the owner of the process instance.

`to` – displays the assignee of the work item.

`date` – displays the date that the work item was created and first offered to the assignee.

`process` – displays the name and description of the process instance, including a link to the process instance display page.

`due` – displays the due date for the work item.

`desc` – displays the detailed description of the activity to be performed.

`form` – if the activity has at least one associated form, it will display the list of all forms as hyperlinks that would allow the user to view the activity using the specific form.

`choices` – displays the choices that the user has at this point.

`commands` – displays the commands section of the process panel which includes the start/stop timer, the Save button, the Accept button, the Decline button, and the Reassign button.

`{{Field udaIdentifier}}` The parameter specified is the identifier of a UDA that carries the value that is to be put here. The data is encoded to be proper for putting into HTML. This token can be used to put the value directly into a page for display to the user. It might easily be placed into a table cell, or any other such HTML construct.

It also may be used to set the initial value of a form input element. If you have a requirement for an input box of a particular size, you can specify the <INPUT> tag and use this token to generate the value attribute of the tag. For example, you might use the following in order to have a 10 character edit field for a UDA variable with the identifier 'ShoeSize':

```
<INPUT NAME=”uda_ShoeSize” value=”{{Field ShoeSize}}” maxchar="10" maxwidth="12">
```

The Console will replace the QuickForm token with the value for shoe size. Similarly, if you want to use a <TEXTAREA> tag in order to have a larger editing area for a UDA with the identifier 'Description' you might use something like:

```
<TEXTAREA NAME=”uda_Description”>{{Field Description}}</TEXTAREA>
```

Another technique to consider is composing a URL from a UDA variable. Imagine that you have an existing application that can display detailed information about an item for sale. Given the item number held in a UDA variable with the identifier 'ItemNo', you might include into the form a link like this:

```
<a href=”http://mysite/itemdetail.jsp?item={{Field ItemNo}}”>details</a>
```

Or if the UDA holds the entire URL then it could be directly placed into the `href` attribute of the <a> tag. Clearly, the field token is the most versatile QuickForm token for including values into the form.

If you use the **Hidden** style for the identifier of a UDA variable, this token generates the entire hidden <INPUT> tag so that UDA values will be available to JavaScript running in the form. The value must be carried in a hidden form element if the value is to be sent back to the server. JavaScript can get the value out of the form element, and can update the form element. This is useful when the data that is stored in the UDA is not directly in the form that is convenient for display. In this case, a small JavaScript could run, read the value, convert to a displayable form, and place that form in a visible form element. Then, upon any change, a different JavaScript will be invoked to convert the data from the visible representation back to the hidden field. A similar technique can be used in JavaScript (and Java) to use the value as a key to look up information elsewhere and display the result of the lookup on the page. For example, if a database holds detailed information on an item for sale, that database might be accessed directly by the form, at the time that the form is displayed, in order to include the latest information about the item directly on the form.

B.3 QuickForm Example: Vacation Request

An example process called `Vacation Request` is included with the Interstage BPM Console that demonstrates the kind of things that might be done with QuickForms. The `Vacation Request` sample files are part of the Template Library.

1. If you have not done so already, import the `TemplateLibrary.bar` application using the **Install Application** function of the Interstage BPM Console.
2. Search for the `Vacation Request` sample files and open the `EDF_Employee.htm` form in a text editor.
3. Search for the form and input tags and note how they correspond to the instructions above.

Appendix C: Supported JavaScript Functions

Interstage BPM provides a set of JavaScript functions that you can use with Java Actions, triggers, and Complex Conditional Nodes.

This appendix explains which functions you can use with these elements. Also, it provides a description of some of these functions.

Apart from functions listed in this appendix, you can use the JavaScript functionality that is defined in the ECMA Standard. For information about the ECMA Standard, refer to the document `ecma-262.pdf` included with the Interstage BPM installation.

Note: The size that a method used in a JavaScript may have is limited by the Java Virtual Machine (JVM). Currently, the method byte code size is limited to 65535 bytes (64 KBytes). If you use a method with a larger size, the JVM will throw an error and you have to reduce the size of the method before executing the JavaScript again.

C.1 General JavaScript Functions

You can use the JavaScript functions explained in this section with Java Actions, triggers, and Complex Conditional Nodes.

Note: Since numeric values are treated as values of type Number, you can use only the following range of integers in JavaScript:

- Minimum: -9007199254740992
- Maximum : 9007199254740992

Note that the following minimum and maximum values of type Long are not supported:

- Minimum: Packages.java.lang.Long.MIN_VALUE
- Maximum: Packages.java.lang.Long.MAX_VALUE

new Packages.java.util.Date()

For a description, refer to the Javadoc that comes with the J2SE Development Kit (JDK).

Date DateAdd(Date or Number date, Number offset, String field)

Returns a JavaScript Date object containing the date and time that is the result of adding the offset to the date. `field` determines the unit of time measure for the offset. Valid `field` values are:

Field Value	Description
`"ss"`	Seconds
`"mi"`	Minutes
`"hh"`	Hours
`"dd"`	Days

Example:

```
var now = Packages.java.util.Date();
uda.Date = DateAdd (now, 1, "dd");
```

Note: The example works correctly only if the User Defined Attribute (UDA) `Date` is of type DATE.

Say `now` has the following value:

`Tue Jul 01 2006 14:02:59 GMT-0800 (PST)`

Then, `tomorrow ( DateAdd ( now, 1, "dd" ) )` has the following value:

`Wed Jul 02 2006 14:02:59 GMT-0800 (PST)`

If `date` is of type Number, its value is interpreted as milliseconds since January 1, 1970.

Boolean DateCompare(Date or Number date1, String operator, Date or Number date2)

Compares two Date values and returns `true` or `false` as the result of the comparison. Valid operators are:

Operator	Description
`">"`	Greater than
`"<"`	Less than
`">="`	Greater than or equal to
`"<="`	Less than or equal to
`"=="`	Equal to
`"!="`	Not equal to

Example:

```
var now = Packages.java.util.Date();
var tomorrow = DateAdd (now, 1, "dd");
if (DateCompare (now,"<", tomorrow))
   ...;
   //... Executes because DateCompare evaluates to true.
```

If `date1` and `date2` are of type Number, their values are interpreted as milliseconds since January 1, 1970.

Number DateDiff(Date or Number date1, Date or Number date2, String field)

Subtracts `date2` from `date1`. Returns the difference between these date/times in days, hours, minutes or seconds depending on the `field` value. Valid `field` values are:

Field Value	Description
`"ss"`	Seconds
`"mi"`	Minutes
`"hh"`	Hours
`"dd"`	Days

Example:

```
var now = Packages.java.util.Date();
var tomorrow = DateAdd (now, 1, "dd");
var diff = DateDiff (tomorrow, now, "dd"));
//diff has a value of 1.
```

If `date1` and `date2` are of type Number, their values are interpreted as milliseconds since January 1, 1970. Example:

```
var date = DateDiff (20000000,10000000,"ss");
```

BigDecimal DecimalAdd(BigDecimal value1, BigDecimal value2)

Returns a JavaScript BigDecimal object that is the sum of the parameters specified. The result inherits the precision from the parameter with the most significant figures.

If you want to assign the result to a UDA, make sure that the UDA is of type BIGDECIMAL.

If you pass any other data type as a parameter, the function converts the parameter to a BigDecimal.

Example:

```
int x = 39;
var z = DecimalAdd ("3.1416",x);
```

If `z` is a BigDecimal, its value is `42.1416`.

Boolean DecimalCompare(BigDecimal value1, String operator, BigDecimal value2)

Compares two BigDecimal values and returns `true` or `false` as the result of the comparison.

Valid operators are:

Operator	**Description**
`">"`	Greater than
`"<"`	Less than
`">="`	Greater than or equal to
`"<="`	Less than or equal to
`"=="`	Equal to
`"!="`	Not equal to

If you want to assign the result to a UDA, make sure that the UDA is of type BIGDECIMAL.

If you pass any other data type as a parameter, the function converts the parameter to a BigDecimal.

Example:

```
var smallDecimal = "1.11";
var largeDecimal = "22.22";
if (DecimalCompare (smallDecimal,"<", largeDecimal))
   ...;
   //Executes because DecimalCompare evaluates to true.
```

BigDecimal DecimalDivide(BigDecimal value1, BigDecimal value2, Number scale)

Divides `value1` by `value2` and returns the result of the division. `scale` determines the number of significant digits for rounding. Any Number can be used as a `scale` value. Default value is 2.

The rounding mode is always to round half up; it rounds towards the "nearest neighbor" unless both neighbors are equidistant. In that case, it rounds up.

If you want to assign the result to a UDA, make sure that the UDA is of type BIGDECIMAL.

If you pass any other data type as a parameter, the function converts the parameter to a BigDecimal.

BigDecimal DecimalMultiply(BigDecimal value1, BigDecimal value2, Number scale)

Multiplies `value1` by `value2` and returns the result of the multiplication. `scale` determines the number of significant digits for rounding. Any number can be used for `scale`. Default value is 2.

The rounding mode is always to round half up; it rounds towards the "nearest neighbor" unless both neighbors are equidistant. In that case, it rounds up.

If you want to assign the result to a UDA, make sure that the UDA is of type BIGDECIMAL.

If you pass any other data type as a parameter, the function converts the parameter to a BigDecimal.

BigDecimal DecimalSubtract(BigDecimal value1, BigDecimal value2)

Subtracts `value2` from `value1`. Returns the difference between these values as a BigDecimal.

boolean toBoolean(String or Number value)

Converts `value` to a JavaScript Boolean. The value can be either a String containing `"true"` or `"false"`, or a Number containing zero (for true) or non-zero (for false). Returns `true` or `false` depending on the parameter passed. If `value` cannot be converted into a Boolean, `false` is returned.

BigDecimal toDecimal(BigDecimal value, Number scale)

Converts `value` to a BigDecimal object. Returns the result of the conversion to the number of significant figures specified with `scale`. Any Number can be used for `scale`. Default value is 2.

If you want to assign the result to a UDA, make sure that the UDA is of type BIGDECIMAL.

If you pass any other data type as a parameter, the function converts the parameter to a BigDecimal.

Packages.java.lang.Float.parseFloat

For a description, refer to the Javadoc that comes with the J2SE Development Kit (JDK).

Packages.java.lang.Integer.parseInt

For a description, refer to the Javadoc that comes with the J2SE Development Kit (JDK).

Packages.java.lang.String.valueOf

For a description, refer to the Javadoc that comes with the J2SE Development Kit (JDK).

C.2 JavaScript Functions Supported with Java Actions

With Java Actions, you can use the functions explained in section *General JavaScript Functions* on page 224 and the functions listed below.

The functions listed below use the Server Enactment Context API `com.fujitsu.iflow.server.intf.ServerEnactmentContext` to provide access to workflow information. For a description, refer to the *API Javadoc*.

```
void sec.addAttachment(String attachmentName, String attachmentPath)
void sec.addProcessXMLAttributeSubstructure(String udaName, String xPath, String
value)
void sec.addProcessXMLAttributeSubstructureByIdentifier(String identifier, String
xPath, String value)
void sec.deleteAttachment(String attachmentName)
void sec.deleteProcessXMLAttributeSubStructure(String udaName, String xPath)
void sec.deleteProcessXMLAttributeSubStructureByIdentifier(String identifier,
String xPath)
void sec.escalateActivity(String assignees)
String sec.getActivityActor(String activityName)
Array sec.getActivityAssignees()
String sec.getActivityName()
String sec.getActor()
Array sec.getAllAttachmentNames()
Array sec.getAllAttributeNames()
String sec.getAttachment(String attachmentName)
Number sec.getCurrentActivityId()
Number sec.getCurrentProcessId()
Array sec.getGroupMembers(String groupName)
String sec.getProcessAttribute(String attName)
String sec.getProcessAttributeByIdentifier(String identifier)
String sec.getProcessAttributeStringType(String udaName)
String sec.getProcessDefinitionId()
String sec.getProcessDefinitionName()
String sec.getProcessDescription()
String sec.getProcessInitiator()
String sec.getProcessName()
Array sec.getProcessOwners()
Number sec.getProcessPriority()
Number sec.getActivityPriority()
String sec.getProcessTitle()
String sec.getProcessXMLAttributeElementValue(String udaName, String xPath)
String sec.joinString(Array)
```

```
Array sec.resolveRelationship(String relationship, String sourceValue)
void sec.sendEmail(String to, String from, String cc, String bcc, String subject,
String body, String mimeType)
void sec.setActivityAssignees(Array assignees)
void sec.setOwners(Array users)
void sec.setProcessAttribute(String name, String value)
void sec.setProcessAttributeByIdentifier(String identifier, String value)
void sec.setProcessDescription(String description)
void sec.setProcessName(String name)
void sec.setProcessOwners(Array users)
void sec.setProcessPriority(Number priority)
void sec.setActivityPriority(Number priority)
void sec.setProcessTitle(String title)
void sec.setProcessXMLAttributeElementValue(String udaName, String xPath, String
value)
void sec.setProcessXMLAttributeElementValueByIdentifier(String identifier, String
xPath, String value)
void sec.setProcessXMLAttributeSubstructure(String udaName, String xPath, String
value)
void sec.setProcessXMLAttributeSubstructureByIdentifier(String identifier, String
xPath, String value)
void sec.validateProcessXMLAttributeValue(String udaName)
void sec.validateProcessXMLAttributeValueByIdentifier(String identifier)
Array sec.splitString(String commaSeparatedList)
```

Using User Defined Attributes (UDAs)

You can use UDAs that have been added to the process definition in your JavaScripts. Use the following syntax:

`uda.<udaIdentifier>`. Note that you must use the identifier and not the name of the UDA, as multibyte characters are not allowed for variable names in JavaScript.

The following example creates a variable and initializes it to the value of a UDA:

```
var someVariable = uda.Price;
```

The following example shows how to assign the value of a variable to a UDA:

```
var lastName = "Jones";
uda.udaIdentifier = lastName;
```

Note: UDAs having the identifier `get` or `set` or `Function` must not be accessed using the syntax `uda.<udaIdentifier>`. The system behaviour is undefined in that case.

As a workaround, do one of the following:

- Use `uda.get("<udaName>")`.
- Assign a different identifier to UDAs to be used in JavaScript, e.g. `uda.myget` where `myget` is the identifier of a UDA having the name `get`.

The methods `uda.get` and `uda.set` allow you to access UDAs by their names:

- `uda.get` returns the value of the specified UDA:

  ```
  var value = uda.get("<udaName>");
  ```

- `uda.set` sets the value of the UDA to the specified value:

  ```
  uda.set("<udaName>", "<udaValue>");
  ```

When assigning a JavaScript return value to a UDA, make sure that their data type matches. Otherwise, the assignment fails.

Note: If the assignment of a value to a target UDA fails due to conversion or any other kind of errors, error details are logged in `IBPMServer.log`. The target UDA is not updated and holds its earlier value.

Interstage BPM maps UDA data types to the following Java data types:

- UDAs of type BIGDECIMAL are mapped to `Packages.java.math.BigDecimal` objects.
- UDAs of type DATE are mapped to `Packages.java.util.Date` objects.

For details about the Java objects, refer to the Javadoc that comes with the J2SE Development Kit (JDK).

C.3 JavaScript Functions Supported with Triggers

With triggers, you use JavaScript expressions to specify control conditions. Control conditions narrow down when the trigger fires.

You can use the functions explained in section *General JavaScript Functions* on page 224 and the function explained below.

String eventData.getXMLData(String xpath)

Returns the text value of the XML element specified in the `xpath` expression.

Example:

Consider the following XML fragment:

```
<Customer>
   <Data>
      <Name>John</Name>
   </Data>
</Customer>
```

The following statement assigns the text value `"John"` of the `<Name>` XML element to the `name` variable:

```
var name = eventData.getXMLData ( "/Customer/Data/Name/text()");
```

Appendix D: Troubleshooting

D.1 Log File Information

If Interstage BPM does not seem to be working properly, check the following log files:

- `IBPMServer.log` - all errors from the server and the Interstage BPM adapters are logged in this file. This log file is often very helpful in troubleshooting. For example, it may indicate that the database server is down.
- `AnalyticsError.log` - all analytics errors are logged in this file.

These files are located at the following location on the Interstage BPM Server computer: `<Interstage BPM Server Installation Directory>/ server/instance/default/logs`

D.2 Resolving Specific Error Situations

D.2.1 Interstage BPM Server Fails to Start

Check the `IBPMServer.log` in the `<Interstage BPM Server Installation Directory>/server/instance/default/logs` directory.

Look for	What to do
`DbService : setConnection: Connection to database server failed. Is the database server running and reachable through the network? {ORA-01089: immediate shutdown in progress - no operations are permitted.`	Check if the database is running. Also check that you can access the database from the machine where the Interstage BPM Server is installed in case the database is running on a different machine. You can use `telnet <Database Server Hostname> <Port>` from the server host machine to check that the connection to the database host/port can be established.

`LdapBroker : getContext: Could not create the directory services. {[LDAP: error code 49 - Invalid Credentials]}`
`LdapBroker : getGroupMembersByDN: Could not retrieve the user groups. {Could not create the directory services. {[LDAP: error code 49 - Invalid Credentials]}}`
`LdapBroker : Could not retrieve the user groups. {Could not create the directory services. {[LDAP: error code 49 - Invalid Credentials]}}`
	Ensure that the user name/password as specified in the `LDAPAccessUserID / LDAPAccessUserPassword` parameters of the Interstage BPM Server are correct and you can login to your Directory Server using the above user name/password.

Look for	What to do

`getContext: Could not create the directory services.`
`LdapBroker : getGroupMembersByDN: Could not retrieve the user groups. {Could not create the directory services.`
`LdapBroker : Could not retrieve the user groups. {Could not create the directory services.`
	Ensure that the LDAP Server is running on the port as specified in the `LDAPServer` parameter of the Interstage BPM Server. You can use `telnet ldapServerHostName port` from the server host machine to check that the connection to the host/port can be established.

`IflowStartup : @(Failed to execute the IBPM startup routine task)Unable to deliver the message for the requested eventClass. {javax.naming.ServiceUnavailableException: A communication failure occurred while attempting to obtain an initial context with the provider URL: "iiop://<ServerName>:<Ports>". Make sure that any bootstrap address information in the URL is correct and that the target name server is running. A bootstrap address with no port specification defaults to port 2809. Possible causes other than an incorrect bootstrap address or unavailable name server include the network environment and workstation network configuration.}`	This is because you installed WebSphere Application Server in a Cell (deployment manager and a managed node) environment, causing the `BOOTSTRAP_ADDRESS` port to be incorrect in some configuration files. 1. Get the correct value of the `BOOTSTRAP_ADDRESS` port from the WebSphere Application Server Console (from **Servers** > **Application Servers** > <ServerName> > **Ports**). 2. Update the value of the `BOOTSTRAP_ADDRESS` port for the following entries in the `iflowClient.properties,ibpm.properties` files, as well as Interstage BPM properties in the database: • `JMSNamingProviderURL` (in the format `iiop://<Host Name>:<BOOSTRAP_ADDRESS>`) • `NamingProviderURL` (in the format `iiop://<Host Name>:<BOOSTRAP_ADDRESS>`)

D.2.2 Error in IBPMServer.log

Check the `IBPMServer.log` in the `<Interstage BPM Server Installation Directory>/server/instance/default/logs` directory.

Look for	What to do
`getGroupMembersByDN: Could not retrieve the user groups. {[LDAP: error code 32 - No Such Object]}`	Possible cause of this error: A user has been deleted from the Directory Server (LDAP Server) but the reference of it is still there in one of the groups.

D.2.3 Timeout During JavaScript Execution

When executing large JavaScripts, the transaction timeout currently set for WebLogic and WebSphere application servers (120 seonds) is insufficient. Due to this setting, script execution may fail with a "transaction timeout".

Your application server administrator can increase the transaction timeout depending on your usage requirements, for example, to 200 seconds. This setting can be changed in the following location:

- WebLogic application server:

  `<WebLogic install dir>/../config/config.xml/<jta>/<timeout-seconds>`

- WebSphere application server:

  In the WebSphere Console: **Servers** -> **ApplicationServers** -> **server1** (default name) -> **Container Services** -> **Transaction Services** -> **Total Transaction lifetime timeout**.

D.2.4 Failure in Writing to an Oracle Database

When the updting of an Oracle database table fails, for example, when you try to archive a process instance, check the Oracle alert log file located in the `<Oracle Installation Dir>/admin/<DB instance name>/bdump` directory, for example:

```
C:\ProgramFiles\Oracle\admin\orcl\bdump\alert_orcl.log
```

The following error may be observed:

```
{Database add/create request failed. {ORA-08103: object no longer exists}}
```

This failure may be due to the fact that the Datafile size reached the file size limit on the hard disk of the database server.

The system administrator of the database server needs to increase the file size on the database server hard disk.

D.2.5 Warning or Error Messages for Interstage Application Server

When Interstage BPM Server is running on Interstage Application Server, warning or error messages are sometimes logged to an event log or a syslog of the Operating System.

If any messages of Interstage Application Server are logged in container log files of Interstage BPM Server or Console work units at the same time when the warning or error messages are logged to an event log or a syslog, it is possible that they are caused by Interstage BPM. In this case, please refer *Contacting Your Local Fujitsu Support Organization* on page 238 and ask the Fujitsu Support Organization to investigate the problem.

However, if messages correspond to any of the following errors or warnings, it is not necessary to ask the Fujitsu Support Organization to investigate the problem:

Error/ Warning	Details
1.	• Message logged to an event log or syslog: `od60002` • Message logged to an EJB container log of the Interstage BPM Server: `IJServer21104` and `IJServer21092` • Description of the message: A timeout is caused by the idle monitoring function for STATEFUL Session Beans and an EJB object in the Interstage BPM Server is automatically deleted by the Interstage Application Server. The message can be ignored because the Interstage BPM Server automatically creates the alternative EJB object.

D.2.6 Access Permissions for Generic Java Action Execution

Generic Java Actions that try to copy or move files to remote machines in the network, may fail if the proper permissions are not setup on the remote machine. To resolve such problems ensure that the proper access permissions are setup on the remote machine.

D.3 Errors During Installation, Deployment and Configuration

This section lists several errors that could occur during the installation, deployment and configuration of Interstage BPM and describes the required action(s) to take:

Installation of IBPM fails when trying to install a new build

I	Cause	The build directory of the exisitng installed build was deleted, without un-installing the build or un-installation failed.

	Action	**For Windows** Manually delete the earlier build's registry-entry, as follows: 1. Go to **Start > Run**, type `regedit`, click **OK**. 2. In the **Registry Editor** screen, go to **HKEY_LOCAL_MACHINE > SOFTWARE > Fujitsu > Install > Interstage BPM Server** . 3. Delete the **Interstage BPM Server** registry entry under **Install** key. 4. Go to **HKEY_LOCAL_MACHINE > SOFTWARE > Microsoft > Windows > CurrentVersion > Uninstall >Interstage Business Process Manager xx.x**. 5. Delete the **Interstage Business Process Manager xx.x** registry entry under **Uninstall** key. This will ensure complete deletion of the registry entry for Windows platform. **For Solaris** Manually delete the Solaris package information, as follows: 1. Open the Command Prompt window and run the command `pkginfo -l FJSVibpm` from any location of the Solaris machine, to check if the Solaris package information still exists. 2. If the Solaris package information is displayed, then delete the package information using the following steps: a. Create a file named **ibpm.uninst** under the **/tmp** location. b. Run the command `pkgrm FJSVibpm` to delete the Solaris package. c. Run the command `pkginfo -l FJSVibpm` once more, to ensure that the Solaris package has been deleted successfully. If no information is displayed, then it confirms that the Solaris package has been completely deleted from the Solaris platform. **For Linux** Manually delete the RPM package information, as follows: 1. Open the Command Prompt window and run the command `rpm -qi FJSVibpm` from any location of the Linux machine, to check if the RPM package still exists. 2. If the RPM package information is displayed, then delete the package information using the command `rpm -e FJSVibpm`. 3. Run the command `rpm -qi FJSVibpm` once more to ensure that the RPM package has been deleted successfully. If no information is displayed, then it confirms that the RPM package has been completely deleted from the Linux platform.

The Interstage BPM installation program was unable to launch on Linux

I	Cause	The `libXp-1.0.0-8.i386.rpm` package is not installed on the Linux machine.

	Action	On the Linux machine, do the following: 1. Download the `libXp-1.0.0-8.i386.rpm` package from your Red Hat Enterprise Linux CD or the Red Hat Network website. 2. Install the package using the `rpm -i libXp-1.0.0-8.i386.rpm` command. 3. Retry launching the Interstage BPM installation program for Linux.

An error occurred during Interstage BPM database creation/update

I	Cause	One of the following values provided during deployment were wrong: • Database Administrator user name • Database Administrator password
	Action	Restore the database from the backup of the database made before deploying Interstage BPM. Run the Deployment Tool again and select **Database Configuration** on the **Welcome** screen. In this way, the database is configured without deploying Interstage BPM again.
II	Cause	One of the following values provided during deployment were wrong: • Host name of the database server • Database SID (database instance name) • Database port
	Action	Restore the database from the backup of the database made before deploying Interstage BPM. Remove Interstage BPM from the application server and deploy it again using the Deployment Tool.
III	Cause	Database server is not running.
	Action	Start the database server. Run the Deployment Tool again and select **Database Configuration** on the **Welcome** screen. In this way, the database is configured without deploying Interstage BPM again.

An error occurred during the execution of `importLDAP.bat`

I	Cause	One of the following values provided during deployment were wrong: • LDAP Key • LDAP Organizational Unit
	Action	Remove Interstage BPM from the application server and deploy it again using the Deployment Tool.

An error occurred during the execution of `importAD.bat`

I	Cause	One of the following values provided during deployment were wrong: • Active Directory Key • Active Directory Organizational Unit
	Action	Remove Interstage BPM from the application server and deploy it again using the Deployment Tool.

D.4 Errors during Starting the Interstage BPM Server

The following tables explain the possible causes of errors during server startup and the appropriate action(s) to take:

Errors Pertaining to Active Directory

I	Cause	The Active Directory Server is remote and Active Directory is not running.
	Action	Start the Active Directory Server, and then start the Interstage BPM Server.
II	Cause	One of the following values provided during deployment was wrong: • User name for the Directory Service Login account • Password for the Directory Service Login account These values are used in the `importAD.bat` file as arguments of the `net user` command.
	Action	Remove Interstage BPM from the application server and deploy it again using the Deployment Tool.
III	Cause	Interstage BPM cannot connect to the Active Directory Server because one of the following values provided during deployment was wrong: • Active Directory Key • Active Directory Organizational Unit
	Action	Remove Interstage BPM from the application server and deploy it again using the Deployment Tool.

Errors Pertaining to Sun Java System Directory Server

I	Cause	Interstage BPM cannot connect to the LDAP Server because one of the following values provided during deployment was wrong: • LDAP Key • LDAP Organizational Unit
	Action	Remove Interstage BPM from the application server and deploy it again using the Deployment Tool.

Errors Pertaining to the Database

I	Cause	The Database Server is not running.
	Action	Start the Database Server, and then start the Interstage BPM Server.

Errors Pertaining to a Hostname Change

I	Cause	You changed the hostname of the computer where Interstage BPM Server has been installed. As the hostname occurs in the names and values of various configuration parameters of the Interstage BPM Server, the server cannot access its configuration settings.
	Action	In the `IBPMProperties` table of the Interstage BPM database, make the following changes: • In the `PROPERTYKEY` column, update any parameter names that have the hostname in the suffix. These parameters have the format `<PARAMETER_NAME>.<HOSTNAME>` or `<PARAMETER_NAME>.<HOSTNAME>.<SERVERNAME>`. • In the `PROPERTYVALUE` column, update any parameter values containing the hostname. To update the `IBPMProperties` table, use the appropriate database commands or a database client software.

D.5 Contacting Your Local Fujitsu Support Organization

If you are unable to troubleshoot your problem:

1. Set the `DebugLevel` parameter of the Interstage BPM Server to `2` using these steps:
   a) Start the Interstage BPM Server Configuration Tool using the following URL:
   ```
   http://<Hostname>:<Port>/fujitsu-ibpm-config-webapp/IBPMConfigServlet
   ```
   b) Log in as an Interstage BPM Super User.
   c) Set the `DebugLevel` parameter to `2`.
   d) Click **Save and Reload properties**.
2. Replicate the actions that caused the error.
3. Contact your local Fujitsu Support organization and provide the following information:

   General Information
   - Operating System
   - Directory Service (type and version)
   - Database server (type and version)
   - JDK or JRE version
   - Application server (type and version)
   - Interstage BPM edition, version and build number
   - Major problem area
   - Priority of the issue
   - Environment in which the problem occurs

Configuration Information

- The configuration file that you exported from the Interstage BPM Server

Log Files

- All log files from `<Interstage BPM Server Installation Directory>/server/instance/default/logs`
- When using Interstage Application Server: All log files from `<Interstage Installation Directory>/J2EE/var/deployment/ijserver/<Your Work Unit>/log`
- When using WebLogic: All log files from `<WebLogic Installation Directory>/user_projects/domain/<Your Domain>/servers/AdminServer/logs`
- When using WebSphere: All log files from `<WebSphere Installation Directory>/profiles/<Your Application Server Profile>/logs/<Your Server>`

OS System Logs

- The Windows event log that you obtain using the Windows Event Viewer
- UNIX system logs stored in `/var/adm/messages`
- Linux system logs stored in `/var/log/messages`

Problem Description

- Description of the steps you performed before the problem occurred
- Frequency with which the problem occurrs

Problem Details

- The application program and its source code that caused the error
- The XPDL file of the process definition that caused the error
- Information about Java Actions, Timers and Agents defined in the process definition
- Screenshot of the process instance history if the process instance goes into error state
- Stack trace if any exception is displayed

  You can obtain the strack trace by clicking **Details** on the error page displayed in the Interstage BPM Console.
- Screenshot of the exception wherever it is displayed
- Screenshot of the process instance (graphical view) if the process instance goes into error state or into an unexpected state
- Calendar files (`*.cal)` if timers are used
- The `agentsConfig.xml` file if agents are used

Glossary

ACID properties	A transaction is a set of actions that obeys the four so-called ACID properties: atomic, consistent, isolated, and durable.
Activity	The description of a task, logical step, or work to be performed in a process. An activity is represented by a work item.
Activity Node	A graphical representation of an activity .
Activity Time	The time it takes to perform a particular activity.
Agents	Components that asynchronously access systems external to Interstage BPM.
AND Node	A node that synchronizes multiple branches in a process.
Annotation	An addition to a process definition allowing for adding explanatory comments to the process definition.
API	Application Programming Interface. The interfaces and classes that programmers may use in their own customized applications to access the server.
ASAP	Asynchronous Service Access Protocol. ASAP is a communication protocol based on SOAP and is used to start, manage, and monitor long running services.
Arrow	A connector between nodes. Arrows guide the process flow from one node to another.
Assignee	The person(s) assigned to perform an activity.
Attachment	A document file generated by any application, which has been associated with a process.
BPR	Business Process Reengineering. The field of study which concentrates on how work may be redefined in terms of processes.
Business Calendar	A calendar that specifies working days and times.
Business Process	See *Process*.
Chained-Process Node	A node representing a complex task that can be accomplished independently from the tasks defined in the parent process definition.
Compensation Action	A Java Action that can be defined as compensation for a regular Java Action, e.g. for cleaning up the system and ensuring a consistent state of external systems involved in a transaction.
Complex Conditional Node	A Conditional Node where the condition is specified as a JavaScript expression.
Conditional Node	A node that directs the process flow along one of several branches, depending on specified criteria.
Database Action	A Java Action allowing for the interaction with external databases that are independent of Interstage BPM.
DB Node	A node that accesses an external database using JDBC.

Delay Node	A node that suspends process execution for a certain amount of time.
Directory Service (DS)	Repository for the entire network's authentication and configuration data. Provides access to services, file servers, databases, and other applications. User and application access to the repository is controlled.
Document Management System (DMS)	The system integrated with Interstage BPM which is used to store attachments, forms, etc. The DMS Adapter is the communication link between the DMS and Interstage BPM.
Due Date	Specifies when an activity is due to be completed once it has become active. A due date also specifies what will happen when it is reached and the activity has not been completed.
EJB	Enterprise JavaBeans.
Email Node	A node that sends out predefined emails.
Error Action	A Java Action that is used to handle specific errors on process definition level, on Remote Subprocess level, and on Java Action level.
Exit Node	A node that identifies the end of a process branch. A process definition has at least one Exit Node.
Form	An HTML or XML file which may be associated with an activity, process instance, or process definition. Forms can be created using Interstage BPM; their appearance can be modified using any XML or HTML editing tool.
Framework Adapter	The Framework Adapter, also called DD Adapter, connects the Directory Service and the Document Management System Adapters. The "DD" is short for "document" and "directory". It handles authentication of the user and manages a consistent user authentication to the Directory Services and Document Management System.
Groups	Sets of individuals who typically share common characteristics. Groups are defined in Interstage BPM's local group store, in a Directory Service or in both systems.
Groupware	A type of software, which facilitates collaboration.
GUI	Graphical User Interface.
Initiator	The person who starts a new process instance.
Integration Action	A Java Action allowing for accessing external functions from within a process definition.
Interstage BPM Console	User interface which allows a user to create process instances, process definitions and access and respond to work items. It is also used by Interstage BPM Super Users to administrate Interstage BPM.
Interstage BPM Form	An XML layout definition plus a Java adapter class from which an HTML file is generated. An Interstage BPM Form is created and designed using the Interstage BPM Form editor.
Java Action	A custom programmed component used for performing the same function several times.
LDAP	Lightweight Directory Access Protocol.

Naming Service	A service that allows clients to find objects based on names.
Node	A graphical representation of a step in a process. Interstage BPM supports different node types, e.g. Activity Nodes, AND Notes, Subprocess Nodes, Conditional Nodes.
No-Operation Action	A built-in Java Action that specifies no operation.
Notification Action	A Java Action allowing for notifying users on events related to process execution. Users can be notified by email, for example, that a process or a single activity has been started.
OnAbort Action	A Java Action that will be executed before a process instance is aborted.
OnResume Action	A Java Action that will be executed before a process instance is resumed.
OnSuspend Action	A Java Action that will executed before a process instance is suspended.
OR Node	A node that splits process flow into multiple parallel branches.
Owner	See *Process Definition Owner* and *Process Instance Owner*.
Participant	A person involved in a process.
Process	A sequence of steps that are performed to reach a business goal. Processes are modeled in process definitions.
Process Definition	The representation of a business process in a form that supports automated manipulation. A process definition defines the behavior and properties of the process instances created from it including the flow of control within the process.
Process Definition Owner	The person who created (or last edited) the process definition.
Process Initiator	See *Initiator*.
Process Instance	Represents a single enactment of a specific process definition. The structure of a process instance is exactly the same as the structure of the process definition on which it is based.
Process Instance Due Date	Represents the due date for a process instance.
Process Instance Owner	By default, the owner of a process instance is the owner of the process definition from which the process instance was created.
Process Participant	See *Participant*.
Project	A container for process definitions, forms, simulation scenarios, attachments, etc. On file system level, a project corresponds to a folder.
QuickForm	A structured, field-based HTML file created using the Interstage BPM Studio.
Remote Subprocess Node	A node representing a subprocess that runs on a remote workflow server.
Role	The name of a person or group (for example, the Manager Role). Each task (activity) in a process is assigned to a role for completion. Roles

	are equivalent to groups, as defined in Interstage BPM's local group store or in a Directory Service.
Rule	Method used to determine choices on activities for which the rules are defined.
Server	In the Interstage BPM context, the component of the workflow management system installed on the computer where the workflow engine provides the run-time environment for a process.
Server Action	A Java Action allowing for interacting with the Interstage BPM Server.
Simulation Scenario	A defined setup for simulating the execution of a process definition on your local computer.
SOAP	Simple Object Access Protocol. SOAP is a standard communication protocol that allows one application to send an XML message to another application. It is used, for example, to access Web Services.
SQL	Structured Query Language.
Start Node	A node that identifies the beginning of a process. Every process definition has one and only one Start Node.
Subprocess Node	A node that represents a complex task. The details of that task are defined in another process definition.
SWAP	Simple Workflow Access Protocol. SWAP passes XML messages over HTTP between workflow servers.
Swimlane	Visual grouping of activities performed by the same Role.
Task	An activity in a process; it typically requires a human decision to be made.
Timer	Expires after a specified interval or at a specified time and date. Timers trigger certain actions when they expire.
User Defined Attribute (UDA)	Data that process participants need to access, modify, or add, such as customer data, order numbers etc. User Defined Attributes are specified in the process definition.
User Profile	User-specific configuration information. This includes information such as whether a users wishes to receive email notifications, email address, and default directory, etc.
Voting Activity Node	A node that allows users to work on an activity in collaboration with one another.
Voting Rule	Rule defined on a Voting Activity Node to determine the outcome of the vote.
Web Service Node	A node that retrieves data from a Web Service and makes it available for further processing.
Workflow	The sequence of activities within a business process.
Workflow Application	A process solution consisting of process definitions, forms, simulation scenarios, attachments, etc. Interstage BPM allows for the creation of Workflow Application projects having a predefined structure. Such

	applications can be deployed on an Interstage BPM server that can be accessed by Interstage BPM clients.
Workflow Server	An Interstage BPM component that provides the run-time environment for process instances.
Work Item	An activity in the worklist.
Worklist	The list of activities.
WSDL	Web Services Description Language. WSDL is an XML-based language that describes the Web Services an organization offers. It also describes how to access the Web Services.
XML Action	A Java Action that allows you to perform specific operations on UDAs of type XML, for example adding XML substructures, setting text or attribute values in XML, or extracting UDA values from XML data.
XPath	XML Path Language. XPath is a language for finding information in an XML document. It is used to navigate through elements and attributes.
XPDL	XML Process Definition Language

Index