Forte Fusion Process Development Guide

Release 2.1 of Forte Fusion™

Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 U.S.A. 1-800-786-7638

Part No. 806-7140-01
November 2000, Revision A

Contents

4 Defining a User Profile

5 Defining Assignment Rule Dictionaries

8 Defining Validations

9 Writing Fusion Process Definition Methods

A Fusion Process Management Examples

Preface

The *Forte Fusion Process Development Guide* starts with a general introduction to Forte Conductor, a development environment for creating and executing Fusion enterprise applications. Conductor provides a process engine and associated graphical tools for designing, automating, and managing the flow of business processes (workflow) in a Fusion application.

This manual provides complete information on how to visually design business process definitions with the Conductor process development workshops. It also provides some overview information on Fusion system management and Conductor client programming. For information on those parts of the design process, refer to the *Forte Fusion Process Management System Guide* and the *Forte Fusion Process Client Programming Guide*.

The first chapter of this manual is intended for anyone using Conductor. The rest of the manual concentrates on the process development workshops. It is intended for application system designers (who design the flow of business processes in Fusion applications) and process developers (who create Conductor process definitions).

About Forte Fusion

Forte Fusion is a suite of business integration tools for integrating and coordinating heterogeneous applications. The tools and software components provided with Forte Fusion let you integrate newly developed applications, legacy applications, and off-the-shelf packages into business processes that are automated and controlled by a process engine.

A Fusion system is a set of tools and software modules installed on top of a compatible version of the Forte Application Environment. It is composed of two subsystems, a process management system and an XML-based backbone system.

Fusion Process Management System The Fusion Process Management System (formerly known as Conductor) provides a set of tools and software modules that support the development, execution, and management of business processes. The heart of this system is the Fusion Process Engine, which controls and manages business processes from beginning to end, coordinating the work of the different resources or applications that participate in the processes.

Forte Fusion customers use the Fusion Process Management System to:

- develop process logic with the graphical process development workshops
- manage sessions and processes, and the engine itself, using the Fusion Console and other tools
- build applications, called process clients, that make direct API calls to the process engine, using the process client APIs (Forte 4GL, CORBA/IIOP, JavaBeans, ActiveX, or C++)

Backbone System The Fusion Backbone System provides a set of tools and software modules that use XML messaging over HTTP or JMS to simplify communication and coordination between applications. A Fusion backbone can support different styles of integration, but the backbone is always installed on top of the Fusion process engine runtime. The heart of a backbone system is a set of application proxies that perform message brokering and data transformation on behalf of applications. For business process support, proxies interact with the Fusion process engine on behalf of any applications that participate in a common business process. The main purpose of these interactions is to communicate the initiation and completion of work activities.

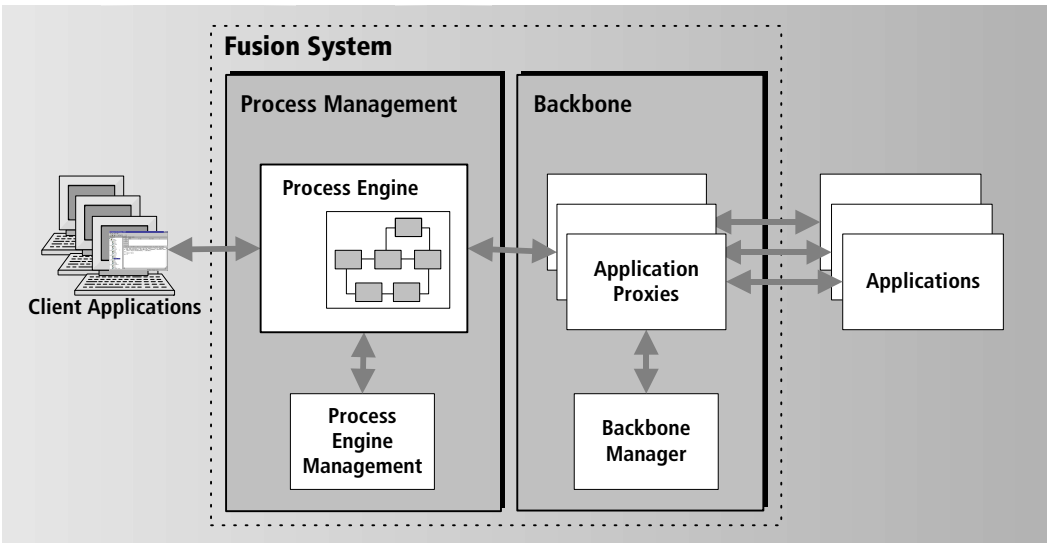


Figure 1 *Fusion System and Subsystems*

Forte provides adapters as well as an adapter toolkit to integrate packages or custom applications that lack a native XML/HTTP interface into a Fusion backbone.

Forte Fusion customers use the Fusion Backbone System primarily to:

- provide an XML/HTTP interface between proxies and applications
- configure application proxies to participate in a managed business process

The XML/XSL Workshops provided with Fusion facilitate the development, testing, debugging, storage, and management of sample XML documents and the XSL stylesheets used for message transformation between applications.

The Organization of this Manual

This manual begins with a brief conceptual overview of Fusion process management design concepts, followed by an introduction to the basic aspects of designing a process definition. The rest of the manual provides information about the workshops you use to develop your process logic. Briefly, the chapters are:

Chapter	Description
Chapter 1, "Fundamentals"	Begins with general Fusion application concepts, explains the Fusion application architecture, describes a Fusion system and how to use it, and discusses a number of application design topics.
Chapter 2, "Getting Started: the Process Development Workshops"	Describes how to start Conductor and how to exit the program, and provides an overview of the workshops.
Chapter 3, "Managing Fusion Plans: the Repository Workshop"	Describes how to use the Conductor-specific features of the Repository Workshop.
Chapter 4, "Defining a User Profile"	Describes user profiles and how to use the User Profile Workshop to create, modify, and distribute them.
Chapter 5, "Defining Assignment Rule Dictionaries"	Describes assignment rule dictionaries and how to use the Assignment Rule Workshop to create, modify, and distribute them.
Chapter 6, "Defining Application Dictionaries"	Describes application dictionaries and how to use the Application Dictionary Workshop to create and modify them.
Chapter 7, "Creating Process Definitions"	Describes process definitions, gives a detailed description of the components of a process definition, and explains how to use the Process Definition Workshop to create, modify, and distribute process definitions.
Chapter 8, "Defining Validations"	Describes session and user validations and how to use the Validation Workshop to create, modify, and distribute them.
Chapter 9, "Writing Fusion Process Definition Methods"	Provides information that is useful in writing methods in any of the process development workshops.
Appendix A, "Fusion Process Management Examples"	Describes the example applications shipped with Conductor, and how to install and use them.

Conventions

This manual uses standard Forte documentation conventions in specifying command syntax and in documenting Forte 4GL TOOL code.

Command Syntax Conventions

The specifications of command syntax in this manual use a “brackets and braces” format. The following table describes this format:

Format	Description
bold	Bold text is a reserved word; type the word exactly as shown.
italics	Italicized text is a generic term that represents a set of options or values. Substitute an appropriate clause or value where you see italic text.
UPPERCASE	Uppercase text represents a constant. Type uppercase text exactly as shown.
underline	Underlined text represents a default value.
vertical bars |	Vertical bars indicate a mutually exclusive choice between items. See braces and brackets, below.
braces { }	Braces indicate a required clause. When a list of items separated by vertical bars is enclosed in braces, you must enter one of the items from the list. Do not enter the braces or vertical bars.
brackets [ ]	Brackets indicate an optional clause. When a list of items separated by vertical bars is enclosed by brackets, you can either select one item from the list or ignore the entire clause. Do not enter the brackets or vertical bars.
ellipsis …	The item preceding an ellipsis may be repeated one or more times. When a clause in braces is followed by an ellipsis, you can use the clause one or more times. When a clause in brackets is followed by an ellipsis, you can use the clause zero or more times.

Forte 4GL TOOL Code Conventions

Where this manual includes documentation or examples of Forte 4GL TOOL code, the TOOL code conventions in the following table are used.

Format	Description
parentheses ( )	Parentheses are used in TOOL code to enclose a parameter list. Always include the parentheses with the parameter list.
comma ,	Commas are used in TOOL code to separate items in a parameter list. Always include the commas in the parameter list.
colon :	Colons are used in TOOL code to separate a name from a type, or to indicate a Forte name in a SQL statement. Always include the colon in the type declaration or statement.
semicolon ;	Semicolons are used in TOOL code to end a TOOL statement. Always type a semicolon at the end of a statement.

Fusion Example Programs

Forte provides a number of example applications that illustrate Fusion features.

Process Client Example Programs

There are five different APIs available to build a Fusion process client. Example application programs are provided for each API. Each API has its own example files in a subdirectory of FORTE_ROOT/install/examples/conductr/. The PDF file in the examples subdirectory explains how to install and run the example application.

The examples are described in the appendix of the *Forte Fusion Process Development Guide*.

Backbone Example Programs

There are three Forte Fusion Backbone example programs: one illustrating the use of the Forte 4GL TOOL with Fusion, another is an example written in 'C', and a third example illustrates JMS messaging.

The example programs are installed under FORTE_ROOT/install/examples/fusion.

The directory containing each example includes a readme file. The readme file contains the background information and configuration instructions. The Forte 4GL example is a complete system, while the 'C' example is an optional replacement for some parts of the Forte 4GL example.

Documentation

The Fusion online documentation includes the complete documentation set and a master index as PDF (Portable Document Format) files as well as online help. For details on viewing and searching these files, see "Viewing and Searching PDF Files" on page 19.

When you are using a Fusion development application, press the Help key or use the Help menu to display online help. The help files are also available at the following location in your Fusion distribution: FORTE_ROOT/userapp/forte/cl*n*/*.hlp (*n* indicates the release number).

When you are using a script utility, such as Conductor Script (Cscript) or Fusion Script (FNscript), type help from the script shell for a description of all commands, or help <*command*> for help on a specific command.

Forte Fusion Documentation Resources

The *Forte Fusion Installation Guide* explains installation options and how to install the Fusion product (both the Fusion Process Engine and the Fusion Backbone).

Other useful resources available in the Fusion product documentation directory are:

- the fndoc.pdf file which serves as a home page for the entire documentation set
- a master index for all Fusion PDF documentation
- a glossary of terms
- a list of resources for learning more about the underlying technologies

Forte Fusion Process Management

The complete documentation set for Forte Fusion Process Management consists of the following manuals and online help:

- *Forte Fusion Process Development Guide*. Explains how to create business process logic using the graphical process development workshops.
- *Forte Fusion Process Management System Guide*. Explains system management concepts and facilities, how to register process definitions, how to configure and manage the process engine, and other related tasks.
- *Forte Fusion Process Client Programming Guide*. If you are building new applications that interact directly with the process engine, this manual explains how to use one of the provided process client APIs for that purpose. Use in conjunction with the API reference in the online help.
- Online help. Provides complete API reference for the process client APIs as well as task Help on the workshops and the Fusion Console.

Forte Fusion Backbone

The documentation set for the Forte Fusion Backbone consists of the following manuals and online help:

- *Forte Fusion Backbone System Guide*. Explains the backbone architecture, proxy concepts and features, and how to configure backbones and proxies. It includes a reference for FNscript, the Fusion scripting language. This should be the first manual you read if you are integrating an application or an adapter into a Fusion backbone.
- *Forte Fusion Backbone Integration Guide*. Explains how to develop XSL stylesheets and perform other integration tasks so that appropriate XML message transformations can occur between applications and proxies. The manual is used in conjunction with the *Forte Fusion Backbone System Guide* and the Fusion Backbone online help.
- Fusion Backbone online help. Explains proxy document XML, how to use the XML/XSL workshops to create, debug, and manage XML documents and XSL stylesheets, how XSLT and standard XML parsers are supported in Fusion, and provides a complete reference for the HTTPSupport (formerly the HTTP-DC) API. The Forte HTTPSupport API enables the development of standard HTTP communication services for transporting and managing HTML and XML messages.
- *Forte Fusion Adapter Development Guide*. Explains how to use the Forte Fusion software development kit to build a custom Fusion adapter for HTTP services. Discusses the design and functioning of a Fusion adapter, and includes guidance on using the 'C' and Forte 4GL TOOL adapter SDKs. Use in conjunction with the Fusion Backbone online help, which explains the HTTPSupport API, as well as with the adapter readme files. The following table indicates the location of readme files that contain further information about the 'C' adapter functions and the 'C' and TOOL adapter example programs:

Contents	Location
Installing TOOL adapter example program	FORTE_ROOT/install/examples/fusion/toolcon/readme.htm
Installing 'C' adapter example programs	FORTE_ROOT/install/examples/fusion/ccon/readme.htm
Explanation of 'C' adapter functions and #defines	FORTE_ROOT/install/fusion/ccon/fnconect.htm

Forte Application Environment

Forte provides a comprehensive documentation set describing the libraries, languages, workshops, and utilities of the Forte Application Environment. For the complete Forte Release 3 documentation set, see the Forte documentation listed on the Forte CyberSupport page at *http://www.forte.com/support*.

Viewing and Searching PDF Files

You can view and search Fusion PDF files directly from the product CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

There are two ways you can look up information in the Fusion documentation set:

- view and search PDF files directly from the product CD-ROM

  The Fusion documentation set has been indexed with Acrobat Catalog. Use Acrobat Reader with Search to search for text strings across the book set and click hypertext links to display the specified content.

- look up index entries in the *Forte Fusion Master Index* included on the product CD-ROM

  The master index also helps you find content across the full documentation set. It is a composite of all Fusion book indexes and is intended to be displayed online or printed to your local printer. It does not provide hypertext links to entries as the individual book indexes do.

Note You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from http://www.adobe.com. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

▶ **To copy the documentation to a client or server:**

1. Copy the fortedoc directory and its contents from the CD-ROM to the client or server hard disk.

   You can specify any convenient location for the fortedoc directory; the location is not dependent on the Forte distribution.

2. Set up a directory structure that keeps the fndoc.pdf and the fusion directory in the same relative location.

   The directory structure must be preserved to use the Acrobat search feature.

Note To uninstall the documentation, delete the fortedoc directory.

▶ **To view and search the documentation:**

1. Open the file fndoc.pdf, located in the fortedoc directory.
2. Click the **Search** button at the bottom of the page or select **Edit > Search > Query**.
3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click **Search**.

   A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

Note For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

   All occurrences of the word or phrase on a page are highlighted.

5 Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

Toolbar Button	Keyboard Command
Next Highlight	Ctrl+]
Previous Highlight	Ctrl+[
Next Document	Ctrl+Shift+]

6 To return to the fndoc.pdf file, click the Homepage bookmark at the top of the bookmarks list.

7 To revisit the query results, click the **Results** button at the bottom of the fndoc.pdf home page or select **Edit > Search > Results**.

Where to Go from Here

What you should read in this manual depends on your responsibilities in your workflow development team.

- All users of Conductor should read Chapter 1, "Fundamentals," to get an overview of how the entire development process works.
- The remainder of the book is intended for programmers who want to convert business process definitions to Conductor process definitions.

If you are responsible for maintaining any part of the Conductor runtime environment (part of which is the process engine), you should also read the *Forte Fusion Process Management System Guide*.

If you are responsible for developing an end-user client application, you should also read the *Forte Fusion Process Client Programming Guide*.

Chapter 1

Fundamentals

This chapter explains some of the key concepts involved in designing, automating, and managing the flow of business processes in a Fusion enterprise application.

The chapter begins with general process management concepts, explains the Fusion system architecture, describes how to use the process engine to manage business processes, and discusses a number of design topics.

The chapter covers the following topics:

- introduction to enterprise process management
- system architecture
- system components
- how to automate business processes
- application design
- working with a process engine

Enterprise Process Management

In today's increasingly competitive business environment, companies must innovate at an ever-faster pace to retain customers and capture market share. Such innovation is largely driven by information systems, which deliver a growing percentage of the competitive differentiation perceived by customers. These issues are highlighted by the proliferation of B2B (business-to-business) and B2C (business-to-customer) electronic commerce, the success of which depends upon a flexible, scalable and reliable infrastructure.

Although it is critical for today's information systems to evolve in step with the business, the systems environments in many companies include multiple, disparate packaged implementations and legacy applications. To rapidly deploy new functionality, a company's systems require a flexible communications infrastructure and a standard means to translate shared data between applications. In addition, there must be a way to ensure the coordination of the activities of the multiple systems that participate in a business process.

Enterprise process management is the end-to-end control and automation of the set of procedures and processes that constitute an enterprise's business. These business processes are typically well-defined, multi-step processes, and can extend beyond the company firewalls. Enterprise business processes often involve the exchange of many kinds of data, transactions with high rates of throughput, multiple participating resources that are both human and mechanical, timed activities, complex subprocesses, and parallel streams of work. Processes can also have long durations. A typical example of a complex, end-to-end business process is order fulfillment, in which customers place orders on the Web and the orders are sequentially routed to and processed by other applications that perform discrete functions to fulfill the order, such as credit verification, inventory management, shipping and billing.

The Forte Fusion Solution

The Forte Fusion product family provides a comprehensive solution to the need for rapid deployment and integration of new functionality across disparate systems with its suite of enterprise application integration and process management tools and infrastructure. Any new or legacy application or package can be integrated into the Fusion Backbone, a flexibly-designed XML/HTTP communications backbone that allows applications to directly exchange XML messages, as well as participate in a business process managed by the Fusion process engine. The process engine, built with Forte's proven technology and supported on various platforms, including OS/390, is highly scalable, reliable and fault-tolerant, making it an excellent choice for supporting mission-critical business processes.

Applications that participate in a Fusion-managed business process may be tightly-coupled or loosely-coupled to the process engine. Tightly-coupled applications, called *Fusion process client applications*, can be rapidly developed using one of the Forte-supplied APIs. (For details, see the *Forte Fusion Process Client Programming Guide*.) Loosely-coupled applications can be quickly integrated using a native or adapter-supplied XML interface. These applications are represented on the Fusion backbone by application proxies that use XSL stylesheets for data transformation. For details on integrating applications using XML and XSLT, see the *Forte Fusion Backbone System Guide*, the *Forte Fusion Backbone Integration Guide*, and the *Forte Fusion Adapter Development Guide*.

Regardless of whether an application is tightly or loosely coupled, when the Fusion process engine is used to manage a business process, you separately develop and maintain the process logic independently of the application logic, making it easy to implement dynamic changes in the business process. With loosely-coupled applications, there is a further separation of integration logic, in which XSL stylesheets specify the actions to be taken and provide the means to transform exchanged data.

This manual explains the design concepts and elements involved in creating a process definition and how to use the graphical process development workshops for this purpose. It also includes relevant information for those designing Fusion process client applications.

Fusion Application Architecture

A business process automation application includes process logic that determines both the routing between the various activities in a process and the users who can perform the various activities.

There are two general approaches—two application architectures—for incorporating process logic into a process management application: the traditional monolithic architecture and the process controller architecture employed in Fusion applications.

Traditional Monolithic Architecture

Consider the application architecture illustrated in Figure 2.

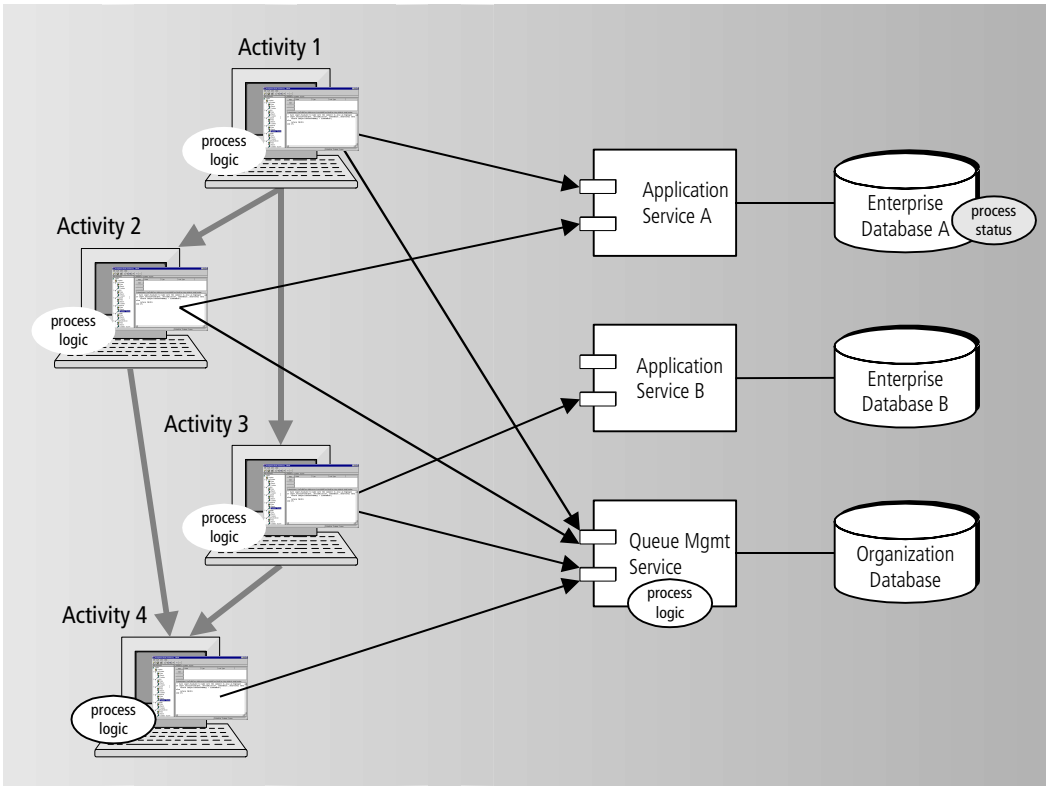


Figure 2 *Traditional Process Automation Application Architecture: Process Logic Throughout*

In this application, a number of client applications perform the various activities involved in an enterprise business process. These client applications interact with several shared application services, some of which access enterprise databases. Suppose the normal business process involves the process flow indicated by the arrows in the illustration: Activity1 is first performed, and depending on the result, either Activity2 or Activity3 or both are performed, followed by Activity4.

In the traditional process automation architecture, Activity1 would need logic that determines whether to pass the result of Activity1 to Activity2 or to Activity3 or to both. It would make this decision by consulting an internal routing table, which specifies the routing based on the value of some selected process data. It might set a status variable or place an entry in a queue for users of Activity2 or Activity3. The queue might reside on a queue manager service in the application, and the queue manager service might post an event to notify any Activity2 or Activity3 application users of an addition to the corresponding queue.

In the meantime, Activity1 would write information concerning the status of its work to a database and free any locks it had on enterprise data. A user of the next activity, say Activity2, would need to log in to the queue manager service, which would establish the user's permission to perform Activity2 (this would normally involve validation against organizational data). Once validated, a user would either respond to events indicating the arrival of work items on a queue or poll the queue manager for the arrival of work items.

Even the acceptance of work could involve significant process logic. For example, Activity4 would need to consult status fields in a database and check whether the conditions that trigger Activity4 have been met, namely if Activity2 or Activity3 or both (depending on the process logic) have been completed.

In short, in a monolithic architecture, process logic is embedded in application logic, and process data is stored in enterprise databases. This application architecture is not very well suited to production process automation for the following reasons:

- All changes to the business process require a change to the application. Even the slightest modification to the business process requires that the entire application be changed and tested before it can be deployed. Database schemas might also have to change. This greatly increases the overhead for developing and maintaining production workflow applications.
- The business has little or no overview of its business processes. Since the process logic is embedded within an application, there is no simple mechanism for the detailed tracking of process status. Business analysts have no easy way to analyze how work is flowing or where process bottlenecks might exist.

Process Controller Architecture

The example application in Figure 2 on page 26 would be implemented very differently using a process controller application architecture, as shown in Figure 3 on page 28.

In this architecture, application and process logic are completely separate. All process logic and all process status are isolated in a unique application service—the process engine—and its corresponding database.

The applications that perform the various activities in a business process do not include any embedded process logic. Instead, they interact with the engine (either directly or through a backbone proxy), which controls and manages the flow of business processes, coordinating the work of the different users (human or machine) that perform the various activities that make up a process. In addition no process status data is stored in enterprise databases.

This architecture—used in Fusion enterprise applications—has a number of powerful advantages over the traditional monolithic architecture:

- Business processes can be redefined without impacting the application logic that supports the business activity; both the logic for routing between activities and the rules for assigning activities to users are handled in the process controller engine.
- The engine is programmable. It can perform concurrent execution of any number of *process definitions*, programs containing all the process logic of a business process automation application.
- Process definitions are defined using graphical tools that provide a visual representation of the process. Process definitions make use of reusable components and are, themselves, reusable.

- The engine takes active control of process execution. It notifies users of pending work, assigns work as users log on, notifies users of changes in process data, determines when to trigger activities, and so on.
- The engine database facilitates process tracking and analysis. It provides the information to determine the status of any individual item of work, to obtain historical data on process execution, and to uncover bottlenecks within a business process.

The engine supports high-end production systems by providing failover support and replicated access to the engine database. In addition, configurations using multiple engines provide scalability and system growth. One or more engines can support thousands of process executions and can be distributed across different computers.

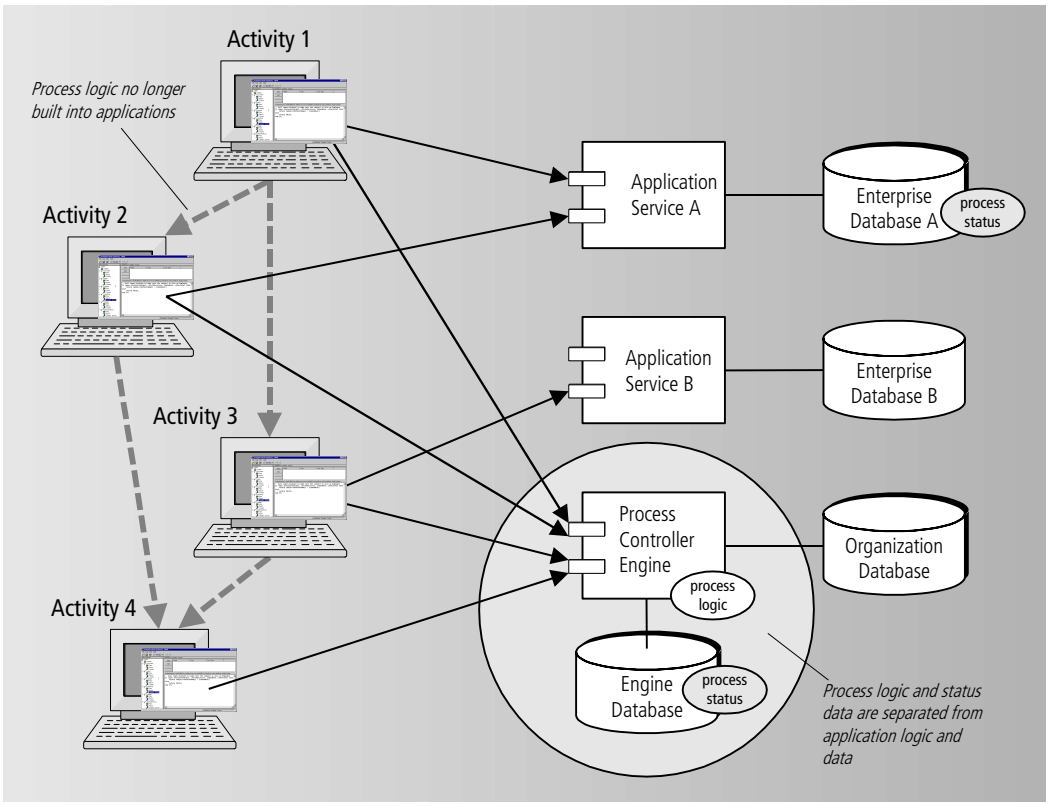


Figure 3 *Fusion Application Architecture: Process Logic Localized in Engine*

Fusion System Components

The Fusion architecture described in the previous section provides a powerful superstructure for managing enterprise business processes. This architecture contains the set of tools and software components that support the integration, development, execution, and management of applications.

Fusion is a flexible, open integration and development environment. It is not restricted to specific business process domains, such as document management, financial transactions, or human resources management, as are some EAI products. With Fusion you can produce process management systems for the broad range of business processes that exist at most companies.

Some of the components of a Fusion process management system have already been mentioned in the previous section—the process controller engine, its database, and a graphical tool for programming process definitions. This section describes these and other components in more detail. Figure 4 illustrates the complete set of components:

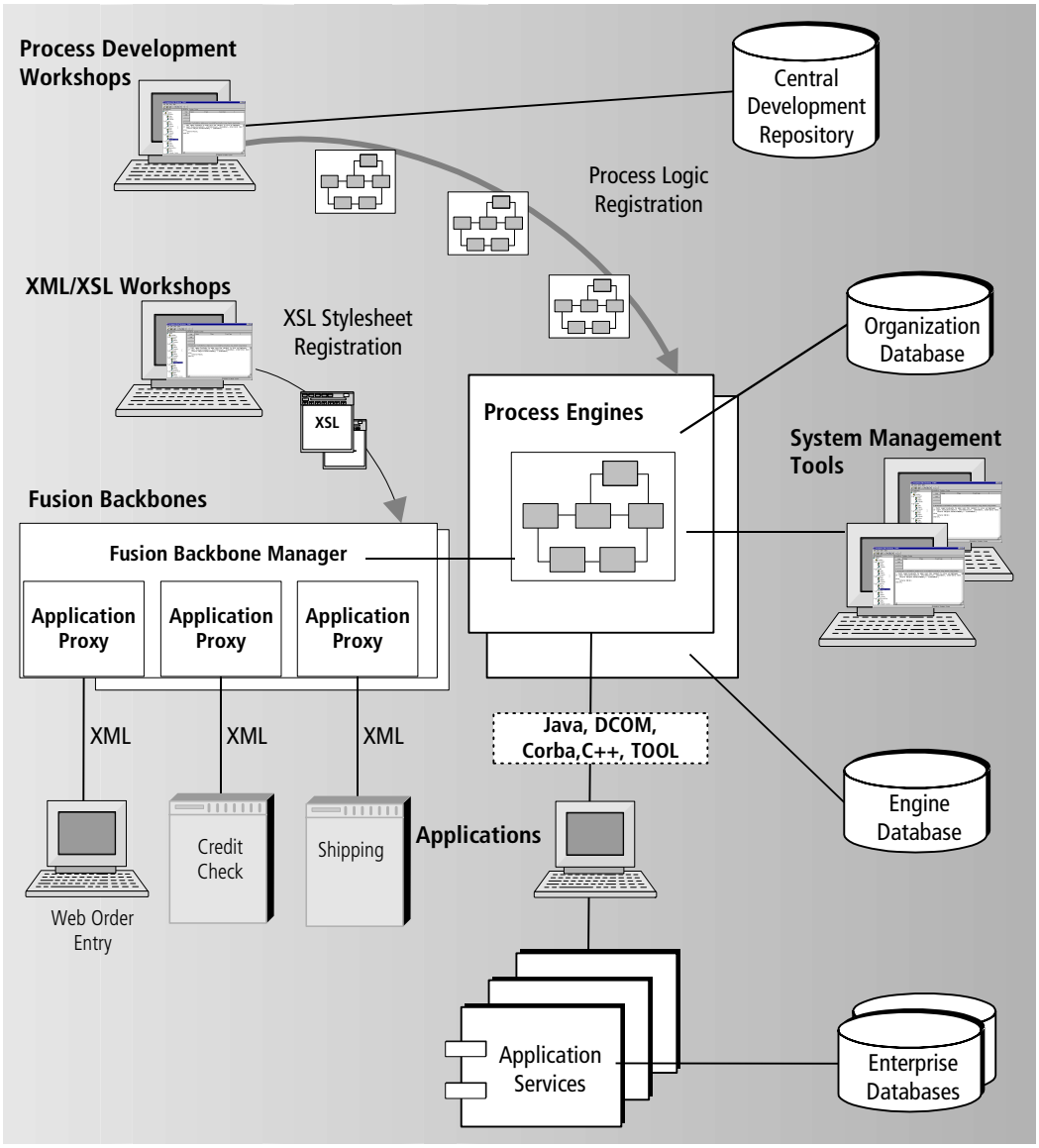


Figure 4 *Fusion System Components*

Fusion provides the following integration, development and runtime components:

Process engine The system includes one or more business process engines. The engine is the heart of the system. It controls and manages the flow of business processes, coordinating the work of the applications that let users perform the various process activities. The engine does this by executing "programs" called *process definitions*; the engine can be thought of as a programmable, shared application service. Process definitions are dynamically registered with the engine, allowing them to be loaded and executed without shutting down the engine. In addition to executing process definitions, the engine validates user logins, interacts with other engines, writes current state and history information to the engine database, and makes performance information available to system management tools.

Process development workshops The workshops are a set of graphical programming tools used for creating process definitions. The workshops store information in a central development repository. When development of a process definition is complete, it is registered with an engine, which then executes the process on behalf of client applications in the system.

XML/XSL workshops These two workshops make it easy to develop, test, debug, store and manage the XSL stylesheets used to integrate loosely-coupled applications into a Fusion backbone. For details, see the Fusion Backbone online Help.

Fusion process client application interfaces Fusion process client applications provide one way for users to perform the activities that make up a process. The client application can perform this work either directly, or by invoking other application services within a distributed application. Process client applications establish sessions with the engine, start instances of processes, obtain appropriate activities, perform these activities, and notify the engine of completion. The client applications are developed with Forte or with another development environment available at the site. They access the engine through an API supplied with the Fusion product. Fusion supports client application implementations in JavaBeans, CORBA/IIOP (Internet Inter-ORB Protocol), ActiveX, Forte TOOL, and C++.

System management tools Fusion Console and its command line counterpart, Conductor Script (Cscript), are used to manage engines (including registration) and to monitor and manage process execution within an engine. For the Fusion backbone, Fusion Script (FNscript) is used to configure and manage backbones and proxies, including the configuring of XSL stylesheets.

Engine database The engine database provides persistent storage for current engine state information. It can also be used to store historical state information for tracking process execution over time. (A status report tool that accesses the engine database will be developed by most sites.) The engine database system is not supplied with the product, but must be provided on site; however, the engine creates all required database tables at startup time. The Fusion process engine can use any relational database system supported by Forte, such as Oracle and Informix.

Central development repository The central development repository supports team development of process definitions. The repository provides all the base classes needed for the process development workshops. It also supports the development of Fusion process client applications in TOOL.

Organization database The organization database is not supplied with the product—it must be provided on site. (A functional sample database, however, is supplied with the product.) The organization database is an important component of a Fusion-managed system. The engine must validate users against this database, which provides the user information required by the engine to assign activities to users of client applications. For loosely-coupled applications, the FNscript utility provides other means of authenticating sessions.

System Implementation

The components of a Fusion development system through a number of software modules running on various nodes in your computing environment. Some of the components are provided by Fusion system software (engine components, process development workshops, and system management tools), some must be developed on site and require Fusion system software (process definitions and process client applications), and some must be developed on site using other software products (organization database).

The Fusion development system provides a tremendous amount of flexibility and power. Just as Forte application architecture isolates application logic from the underlying technical infrastructure, Fusion enterprise application architecture isolates application logic from the underlying business process logic. It lets you implement the application logic (the user interface components and shared application services) of your applications while independently defining the process logic with Fusion development tools. The application logic and process logic are then brought together at runtime by the process engine, which, by executing the process logic, controls and manages the execution of the application logic.

The remainder of this chapter focuses on concepts and design elements needed to implement process logic. It has a wider scope than the rest of this manual, which focuses on how to develop the process logic components.

Creating and Using Process-Based Applications

As described in "Fusion System Components" on page 29, a Fusion development system contains a number of components, some used for development, some used for runtime execution, and some used for both. In most cases different individuals work with different components of the system.

This section describes in very general terms how different individuals can use various components to design, integrate, develop, execute, and manage an application.

The Project Team

The following roles are generally needed on a Fusion project, whether performed by the same or different individuals:

Role	Tasks
Application system designer	▮ Design the overall enterprise application. ▮ Coordinate the work of developers and integrators.
Process developer	▮ Create process definitions, following guidelines provided by the application system designer.
Application integrators	▮ Integrate packages and legacy applications for use with a Fusion backbone, including development of XSL stylesheets and integration of adapters, if needed. One person may be a Fusion specialist, and another is familiar with the APIs of the applications being integrated. For more information, see the *Forte Fusion Backbone System Guide*, the *Forte Fusion Backbone Integration Guide*, and the *Forte Fusion Adapter Development Guide*.
Application developer	▮ Create Fusion process client applications, if needed, following guidelines provided by the application system designer.
System manager	▮ Install the Fusion system ▮ Use system management tools to manage engines, the Fusion backbone, and process execution.

Design, Develop, Execute, and Manage

Before using any of the development system components, the *application system designer* must design the overall application. This requires an understanding of the work the application must perform and the processes by which that work takes place. The development system provides the tools for implementing the process logic of the application—the application logic must either exist already or be developed using other tools.

Following the high level design phase, the s*ystem manager* can install Fusion software by using the product's installation program. The system manager decides in advance which nodes in the environment host engines, which are used for process development, which are used to develop process client applications, which host system management tools, and which host the Fusion Backbone. For more information, see the *Forte Fusion Installation Guide* and the *Forte Fusion Backbone System Guide*.

Following installation, the *application system designer* normally uses a subset of the process development workshops to create important design elements—*user profiles, assignment rule dictionaries, application dictionaries,* and a *validation*. For a discussion of design elements and concepts, see "Application and Process Logic" on page 35.

Process developers can now begin implementing the process logic of the application using the Fusion Process Definition Workshop, where the process definitions executed by the engine are created. For more information on this workshop, see Chapter 7, "Creating Process Definitions."

If applications or packages are being integrated using the Fusion Backbone, *application integrators* must specify the integration logic and advise the *system manager* how to configure application proxies.

If new process client applications are being written to perform the activities specified in the process definition, *application developers* can begin developing them. Process client applications use a Fusion process client API to establish sessions with the engine, start instances of a process, obtain appropriate activities, perform the activities, and notify the engine of completion. For more information, see the *Forte Fusion Process Client Programming Guide*.

To test the development work, you must have a running engine. The *system manager* must configure a test engine and start it. For more information, see the *Forte Fusion Process Management System Guide*.

Once the engine is started, the *application system designer* can register design elements (the user profile, validation, and assignment rule dictionaries) and *process developers* can register their process definitions with the engine. The system manager can also perform registrations using the system management tools.

To test newly developed or integrated applications, *application developers* or *application integrators* must start the applications (and all supporting application services), log on to the test engine, and create instances of the registered process definitions. Once a process instance is started, the engine controls the routing and assignment of activities in accordance with the process definition, and the applications let users perform those activities. The *system manager* can use system management tools to monitor the state of processes being executed by the engine and perform process management tasks, if necessary.

If execution does not proceed as expected, fixes in the process definition or in the applications are required. *Process developers* can modify process definitions and register the modified versions, while *application developers* modify client applications, start new process instances, and so on, until testing proves successful.

At this point, the *system manager* deploys the client applications (and any required shared application services), starts a production engine, registers the completed process definitions, and places the required shared application services on line.

Note If you are using the Forte Fusion backbone to integrate legacy applications or packages, see the *Forte Fusion Backbone System Guide* and the *Forte Fusion Backbone Integration Guide*.

To summarize, the following tasks are typically part of a Fusion project.

To use the Fusion development system:

1. *Designer* creates high level design of the enterprise application.
2. *System manager* installs the engine and runtime components.
3. *Designer* creates important design elements (user profile, assignment rule dictionaries, application dictionary, and validation).
4. *Process developers* use design elements to create process definitions.
5. If existing applications or packages are being integrated, *application integrators develop the XSL stylesheets to* integrate them.
6. *If new* process client applications are being written, *application developers* create them.
7. *System manager* configures and starts up a test engine. If a Fusion backbone is being implemented, the application integrator should assist in configuring proxies with XSL stylesheets and other information.
8. *Designer* registers a user profile, validation, and assignment rule dictionaries, and *process developers* register process definitions with test engine.
9. *Application developers* or *application integrators* start applications and all supporting application services, and log on to the test engine.
10. Testing takes place followed by modifications in process definitions and client applications.
11. *System manager* deploys application, starts a production engine, and registers process definitions (as well as a user profile, validation, and assignment rule dictionaries) with a production engine.
12. Maintenance cycle begins.

Application and Process Logic

The flexibility that Fusion provides means you must pay careful attention to the overall architecture of your application. An application system designer needs to develop a "big picture" of the process automation application and formulate guidelines that process developers and client application developers follow.

The designer needs to understand the basics of the business processes being automated. Ideally, during the design phase, the application system designer is part of the team doing the business process analysis. Participating in the business analysis allows the designer to make effective decisions on issues ranging from the work capacity of the overall system to the level of graphical user interface (GUI) interaction for the end users.

Business object model

For the application logic, the designer ideally starts from an *object model* of the business. The designer needs to determine the business objects (data needed by the application and stored in enterprise databases), as well as the shared application services required to manage the business objects, apply business rules, or perform other services. In some cases, many of the application components may already exist, and need to be tied together.

Business process model

Process logic flows from a related but different model—a *business process model*. This model, normally created by business process analysts, specifies relationships between the many individual work activities in a business process. It describes the sequencing, coordination, and routing among activities, and specifies who performs different activities. It is a visual model of the dynamic flows within a business process. In a Fusion application, the business process model is implemented as a *process definition* executed by a process engine.

User interface model

Activities within a process are often performed by users. If you are developing a new interactive process client, its user interface must communicate with the process engine as well as with other shared application services. Nevertheless, design of the client interface components of an application mainly falls within the application logic domain.

A *user interface model* specifies the relationships among the different client windows used to perform various application tasks. In the case of process client applications, this model must also take into account how the client applications interact with the process engine. This consideration could affect the packaging or grouping of client software components.

While a designer typically deals with both application and process logic, the following discussion focuses on the process logic domain. It presents high-level concepts and structural elements that the designer must consider in providing a framework for application development or integration. This section covers the following three design topics:

- process logic concepts and design elements
- user profile design concepts
- client application design concepts

Process Logic Concepts and Design Elements

In a Fusion system, process logic is encapsulated in code—a process definition—which, when registered with and executed by the process engine, implements a business process. A business process model is normally represented graphically, as illustrated in Figure 5.

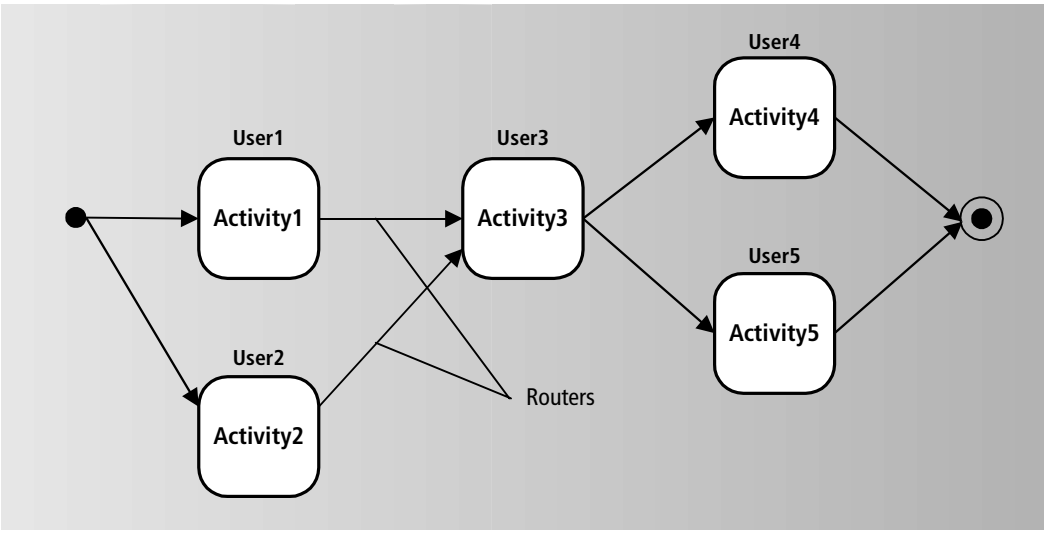


Figure 5 *Business Process Model*

The basic unit of a process model is the *activity*. An activity represents a unit of work in a process. The process logic involves, principally, answering the following questions in the order shown:

1 What is the general sequencing (or routing) of activities in the process?

2 What conditions must be met before an activity can be performed (trigger conditions)?

3 Which user or users can perform each activity?

4 What activities should be performed subsequent to the completion of each activity?

To execute process logic, the process engine creates an instance of a process definition, and, at the appropriate time, creates an instance of each activity defined in the process definition. The engine takes each activity instance through a series of states. The states are shown in Figure 6 on page 37, a magnified view of Activity3 in Figure 5.

Each instance of an activity progresses through the following states:

PENDING Once created, an activity is placed in a PENDING state. In this state a *Trigger method* is executed to determine if all the conditions needed for the activity have been met. When its trigger method returns TRUE, a *Ready method* is evaluated to perform any desired processing before placing an activity in a READY state.

READY In this state *assignment rules* are evaluated to determine which users are permitted to perform the activity. An activity remains in the READY state until a user (or possibly the engine) places it in an ACTIVE state.

ACTIVE In this state, the activity is performed, generally by a user (some activities can be performed automatically by the engine).

COMPLETED (or ABORTED) In this state, a set of routers is activated if the activity completes successfully (or is aborted). In each activated router, a *Router method* is executed to determine if the conditions needed to route process flow to a successive activity have been met.

Some activity types skip one or more of the above states, but in general Figure 6 represents the progression of states through which an activity passes. For a more complete discussion of activity states and the methods that apply to them, see Chapter 7, "Creating Process Definitions" or the *Forte Fusion Process Management System Guide*.

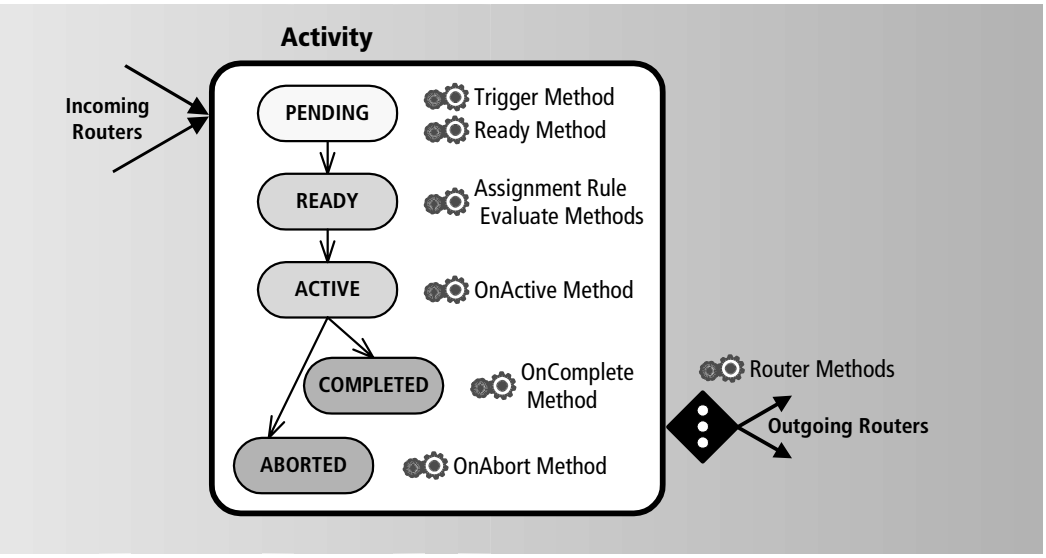


Figure 6 *Activity States for Process Logic Execution*

Activities and activity states play a crucial role in process logic. In addition, a number of system design elements are needed to specify both the routing between activities and who performs activity work. These design elements, created by an application system designer, are:

- process attributes
- assignment rule dictionaries
- user profiles

These elements are introduced below, and illustrated in Figure 7 on page 38 and Figure 8 on page 39.

Routing Between Activities

Each activity in a process (except for the first and last activity) is normally preceded and succeeded by one or more other activities. In general, routing from one activity to another depends on any number of conditions: whether the first activity was completed successfully (or within a specified time period) or aborted, whether activities are to be performed sequentially or in parallel, and whether routing decisions depend upon specific process attribute values.

Process attribute

A *process attribute* is a data value associated with each instance of a process. Process attributes have many uses, one of the main ones being for routing logic. As an application system designer, you specify process attributes needed for routing. These attributes might represent the state of a process, a condition set by an activity, or a business data value—such as the dollar amount of an invoice—which would be used for making routing decisions.

Primary process attribute

You should specify one process attribute as a process identifier. This attribute would be assigned a value as each instance of a process is created. For example, you might specify an invoice number, a job number, an order number, a batch ID, or another identifier. This *primary process attribute* would not normally be used for routing purposes, but rather for identifying data corresponding to a particular process instance.

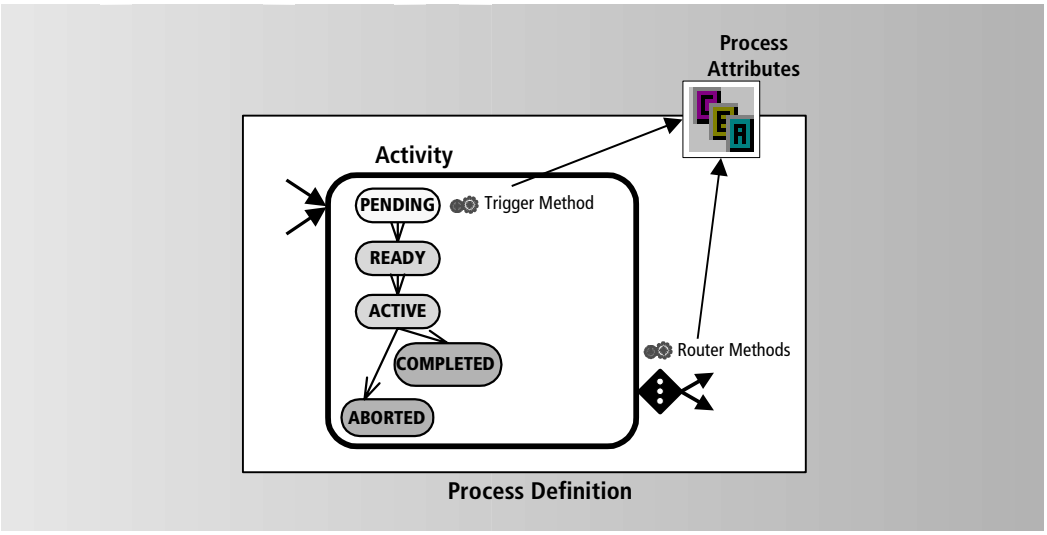


Figure 7 *Process Attributes for Specifying Routing Logic*

Routers

Most activities in a process (except for the last one) are normally succeeded by one or more activities. A router is a pointer from an activity (or a timer) to a successive activity. Associated with each router is a method (a router method) that specifies the conditions under which process flow will be routed to a successive activity. These router methods normally test for certain values of process attributes (for example, whether a batch production sample tests at a Ph value greater than 7.0).

Any number of routers can be defined for an activity. One set of routers can be activated if the activity reaches a COMPLETED state and another set of routers activated if it reaches an ABORTED state. Routers can also be specified for a timer, indicating the routing of process flow if the timer expires.

Triggers

Each activity instance in a process is created when one of its incoming routers is activated. The activity is placed in a PENDING state until all conditions needed to place it in a READY state—that is, to trigger it—are met. Trigger conditions can include the completion of one or more predecessor activities, the aborting of an activity, or the expiration of a timer. These conditions are indicated by activation of the corresponding routers.

A trigger (that is, a trigger method) can also depend upon the values of one or more process attributes. For example, a trigger might require that a customer's order has been verified—a completed activity—*and* that the customer has passed a credit check—a boolean process attribute value—before the order can be shipped.

Routers and triggers work together to provide general sequencing of activities in a process. When a router method returns TRUE, an instance of the activity the router points to is created and placed in a PENDING state. If the activity is already in a PENDING state, the trigger method specifies the remaining routers, which must be activated before an activity is ready to be performed.

Who Performs Activities

A process developer specifies which users can perform each activity by using one or more *assignment rules* created by an application system designer. An assignment rule specifies a user role in the organization or any other factors that might determine work responsibilities or access to information. For example, an assignment rule might specify that a loan transaction (activity) must be approved by a bank officer at a particular location who has sufficient sign-off authority and who is not currently on vacation.

Each assignment rule contains an Evaluate method that can define almost any kind of rule. Normally the *user profile* of each client application user is checked against the rule. The user profile is a template that you, as a designer, build for specifying all the important attributes or characteristics of users of your workflow system. (See "User Profile Design Concepts" on page 40.) If a user's profile matches the conditions specified in the Evaluate method of an assignment rule associated with an activity, then that user is permitted to perform the activity.

Assignment rules can also specify assignments based on the value of process attributes, or can specify, for example, that only the same person—or the manager of the person—who performed a previous activity can perform a current activity.

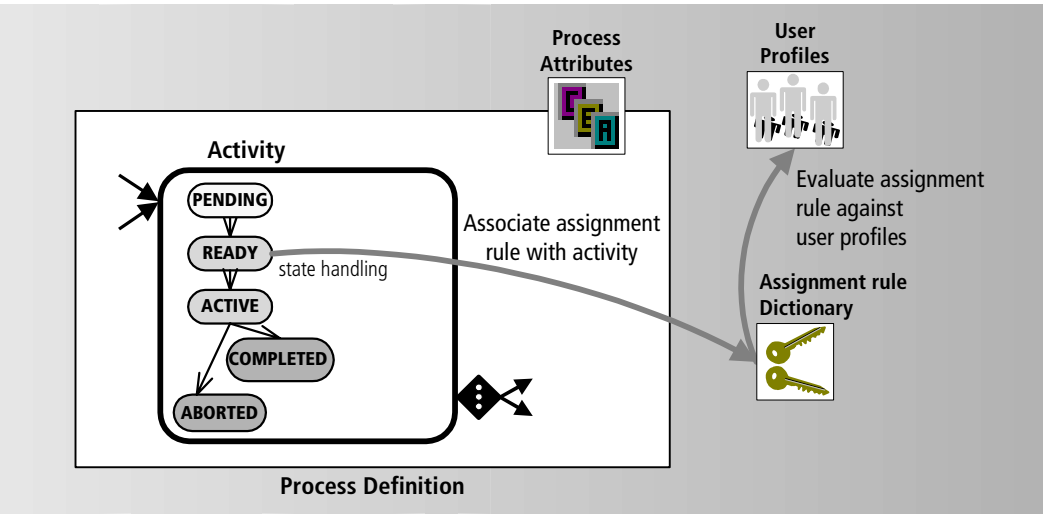


Figure 8 *Design Elements for Specifying Assignment Logic*

As a designer, you build a dictionary of assignment rules that can be applied to user-performed activities —the *assignment rule dictionary*. Process developers can then associate any rule or rules in the dictionary with an activity. The Evaluate method of each assignment rule is executed by the process engine when an activity reaches the READY state. All users whose profiles match the conditions specified in the Evaluate method are permitted to perform the activity.

The *assignment rule dictionary* provides a very flexible approach to specifying who can perform activities. Instead of hard-coding an assignment rule for each activity in a process definition, one or more assignment rules are referenced by name. These references lets you dynamically change the content of assignment rules without having to modify the process definitions that use them. For more information on assignment rules, see Chapter 5, "Defining Assignment Rule Dictionaries."

User Profile Design Concepts

As described in "Process Logic Concepts and Design Elements" on page 36, the user profile is a template that you, as a designer, build for specifying all the important user characteristics in your workflow system. You can build an enhanced or extended user profile template from a default template provided with the Fusion product.

The *user profile* is used to evaluate assignment rules. It is also used to authenticate users when they open a session with a process engine. These two functions are closely related; the information about each individual user needed to evaluate assignment rules is extracted from (or validated with) an organization database when the user opens a session with an engine.

You, as a designer, must create a ValidateUser method that performs the authentication and validation operations. You write this method in a *validation* that you create and register with the engine. The ValidateUser method is executed whenever a user attempts to open a session from a client application.

The user profile and validation scheme is illustrated in Figure 9.

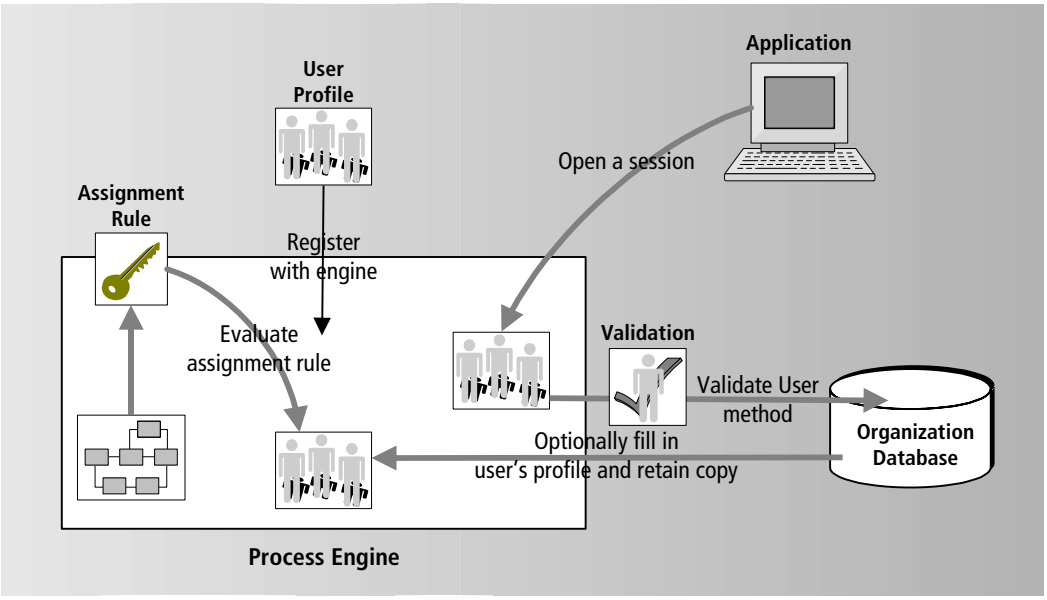


Figure 9 *User Profile and Validation Scheme*

The user profile template and validation definition are first registered with a process engine. When the user of a client application attempts to establish a session with an engine, the client application passes the user's login information to the engine, which creates a user profile object and executes the ValidateUser method. The ValidateUser method authenticates the user, verifies any user-supplied information against the database, and optionally extracts information from the database to fully populate the user profile object. The engine thus serves as a central security clearinghouse.

When the engine creates a session for the user, it retains a copy of that user's profile. This user profile is used when the engine evaluates an assignment rule to decide which sessions are permitted to perform a given activity.

The default user profile template can be used for simple role-based assignment rules. If you need assignment rules that are more sophisticated than checking for roles, you must extend the user profile template to include additional user profile attributes. You must also write a ValidateUser method to extract such information from the database, and provide a database (and schema) that includes all the information you need.

In short, your assignment rules, user profile, validation, and organization database need to be designed together and must be consistent with one another. Your assignment rules check for user attributes that are contained in a user profile, and your validation provides information to a user profile that is contained in your organization database.

Assignment rule dictionaries, consisting of one or more assignment rules, are registered with the process engine. If, in the future, you need to make modifications to your assignment rules that require changes to your user profile (such as adding new user profile attributes), you must modify your user profile and validation accordingly, and then register all three (user profile, validation, and assignment rule dictionary) with the engine. The engine will dynamically load and use the new versions.

If the new user profile requires changes in the organization database, then the database should be changed and updated before registering the new user profile and validation.

Application Dictionary Concepts

Within a Fusion-managed business process applications can act as *service requestors* or *service providers*. An application acting as a service requestor establishes a session with the process engine and creates a new process instance. A service provider application generally supports the following operations, in the order shown:

1. Establish a session with an engine.
2. Maintain a list of activities offered by the engine to the session (a worklist).
3. Select an item in a worklist ("heads-up" approach) or take an item from the top of a queue ("heads-down" approach).
4. Invoke the correct application to perform the activity.
5. Inform the engine when work on an item has been completed.

For applications that perform activities, the application system designer needs to provide guidance to application developers and integrators regarding design and implementation choices. To this end, an application design element—the *application dictionary*—is used to develop work definitions that can be assigned to each activity.

In addition to the information contained in the application dictionary itself, a system designer needs to provide application integrators and developers with other design and implementation information, most of which is normally contained in process definitions:

- where each activity fits in an overall process definition
- the activity type of each activity
- which activities are available to each type of end user
- which process attributes need to be updated for each activity

Note If you are integrating applications using the Fusion Backbone, you configure the application proxy as a service provider or service requestor, to match its application. When the process definition is registered and the proxy is properly configured, it can automatically perform the appropriate interactions with the engine. For details on configuring proxies, see the *Forte Fusion Backbone System Guide*. If you are writing a process client application, you use a Fusion process client API to enable the operations described above; for details, see the *Forte Fusion Process Client Programming Guide*.

Work Definition of an Activity

The work represented by each activity is defined by three properties:

- an activity description
- an identification string, called an application code, that specifies the window, applet, or application that is used to perform the activity
- a set of process attributes needed to perform the activity (and the locks that should be applied to the attributes when the activity is being performed)

As a designer, you build a dictionary of work definitions—called an *application dictionary*—that corresponds to the activities in one or more process definitions. Each item in the dictionary (consisting of the three properties listed above) represents a high level description of the work to be done and the resources needed to accomplish it.

A process developer assigns one item in the dictionary to each activity in a process definition, as shown in Figure 10. The dictionary items are used by the process engine to pass work definition information to the client applications and to lock the appropriate process attributes when an activity is being performed. For example, a dictionary item would be used by the client application to display a description of work in a worklist (activity description), launch the appropriate data entry screen to perform the activity (application code), and find the data needed to perform the work (process attributes).

The process designer uses the application dictionary items by name, without requiring detailed knowledge of their content. Often, the application dictionary items are referenced by more than one activity. The application system designer can make changes to application dictionary items with only limited impact on the process definitions that use them. The application dictionary thus provides important reusability and results in simplified maintenance of the application system.

The application dictionary is the way an application system designer provides both the process developer and client application developer with a consistent definition of the work associated with each activity. It is also a mechanism for communication between developers. The process developer needs to know which application dictionary item to associate with each activity, and the client application developer needs to understand the details of each item, such as the meaning of application codes, in order to successfully write client applications.

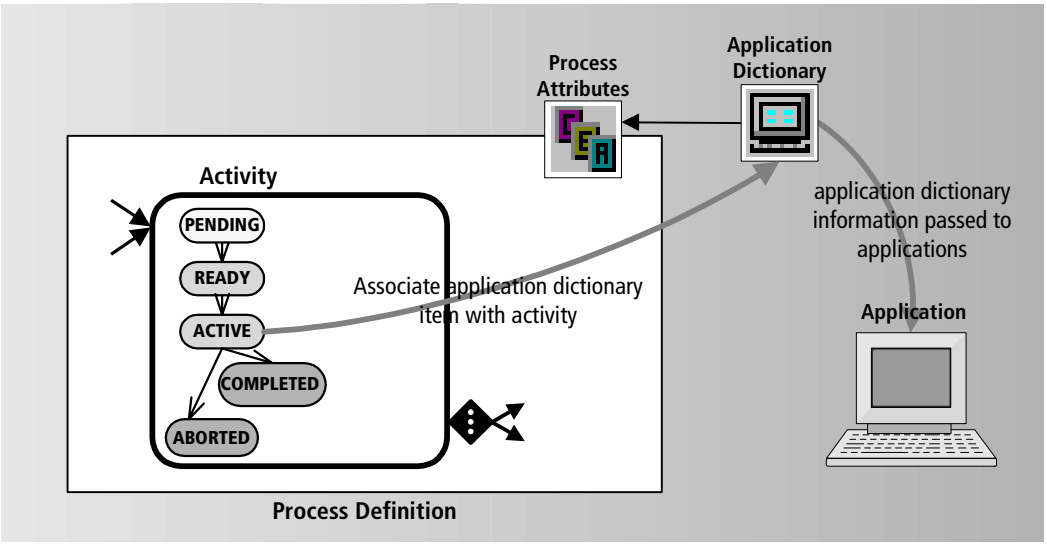


Figure 10 *Application Dictionary Design Element Represents Work of Client Applications*

Activity Type

The process engine does not impose any particular structure on client applications. Some implementation options are:

- Display a worklist allowing the user to select which item to work on, and, in a separate window, launch an existing software program to perform a selected activity.
- Integrate the worklist and application processing for several activities into a single client, thereby allowing the process engine to invoke the next window transparently in the application. For example, if a credit check fails (activity A), bring up a window alerting the operator and go to a loan approval screen (activity B).
- Implement a "heads-down" client in which the user is automatically assigned the first item in a queue. An example would be an inbound telephone sales application where each incoming call is automatically assigned to the next available operator.

The implementation options available depend on the *activity type* specified for each activity. Two activity types are of particular interest in this regard:

Offered activity Activities that are offered to all end users whose user profiles match the assignment rules specified for the activity. Users can normally select from amongst all activities offered them by choosing items from a worklist displayed by the client application.

Queued activity Activities that the engine places in a queue based on a priority established by the process designer. The "heads-down" client application requests the activity at the top of the queue and gives it to the user.

Note The queued activity type is not applicable for applications participating on a Fusion backbone.

Since the activity type determines how a user can interact with the activity, it has a big impact on the type of client application that performs the activity. For example, if the user is going to be performing queued activities, then the client application would be a "heads-down" implementation that would not display a list, but would merely select the next activity off the top of a queue.

As a designer, you need to communicate design decisions regarding activity type so that the work of process developers and application integrators or developers will be coordinated.

There are additional activity types that do not directly impact the client application, but are used for more specialized purposes:

Automatic activity Performed by the engine itself, not by users.

Subprocess activity Represents a separate process, and therefore is not directly associated with a client application or the work of a single activity.

Junction activity Used to improve the screen layout of activities in designing a process definition and to economize the use of router and trigger methods.

Design Element Dependencies

The previous sections describe the roles played by each design element in the design of a Fusion application system. These elements are used—directly or indirectly—by both process developers and client application developers.

Process developers directly use process attributes, assignment rules, and application dictionaries to create process definitions. Process client application developers directly use application dictionary items to write client applications, and indirectly use the user profile and validation to log in to an engine and open a session.

Despite the unique role played by each of these design elements and the flexibility and power that derives from this modular design structure, the various design elements are not strictly independent of one another: they have a number of inter-dependencies. For example, both assignment rule dictionaries and validations point to (depend upon), and must be consistent with the user profile. In addition, both assignment rules and application dictionary items might depend on some number of process attributes.

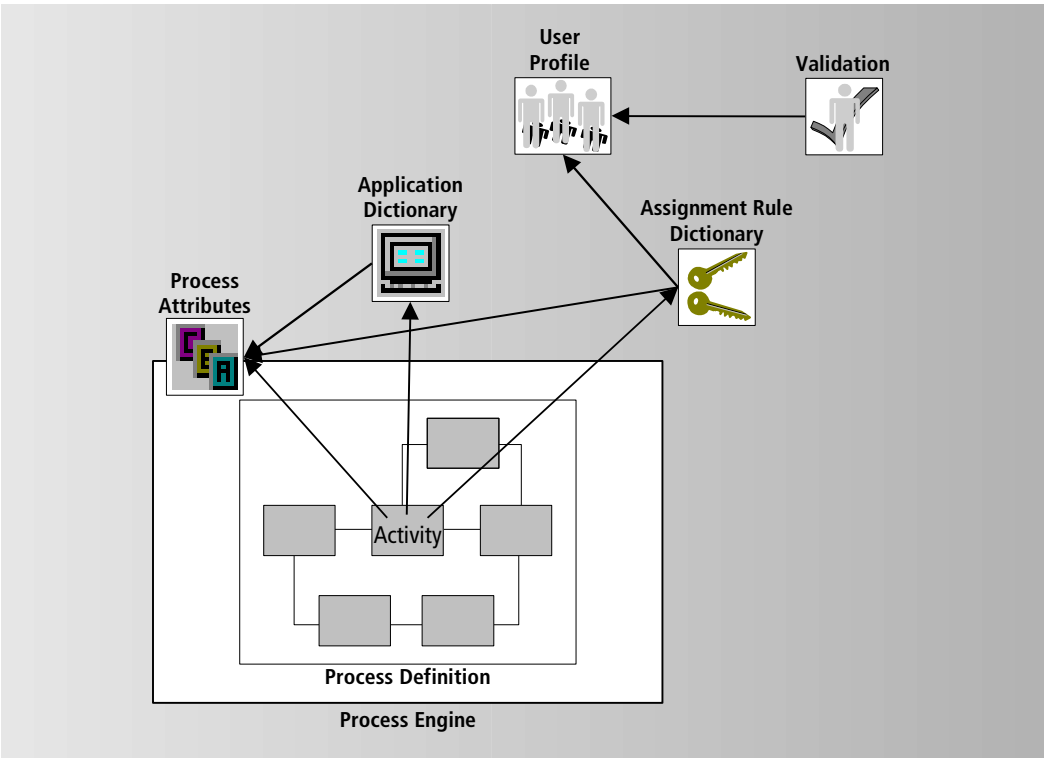


Figure 11 *Design Element Dependencies*

Figure 11 shows dependencies between the various design elements (as well as the dependence of process definitions on process attributes, assignment rules, and application dictionary items).

The designer of a Fusion application system puts information about the application domain (the work or applications required to perform the activities in business processes) into an application dictionary, and information about the user domain (the criteria for deciding who performs the various activities in those processes) into the user profile. This information, in turn, helps define the assignment rule dictionary, which represents general rules that match various user profile roles or characteristics with the various application dictionary items.

Each of these elements should be specified in the preliminary design of your application system. Changes are likely, however, as design proceeds. An assignment rule, for example, might require a user role or attribute not already included in the initial user profile. Or a

new application dictionary item may require an assignment rule not already included in the assignment rule dictionary. Refinements of both application dictionary items and individual assignment rules might depend upon the values of certain process attributes.

While Figure 11 shows the direct dependencies that must be taken into account in the creation of application design elements, all design elements are ultimately related to one another in the implementation of process definitions and client applications.

Modifying Process Logic

While the early stages of application system design are likely to be iterative, the goal is to create design elements that are relatively stable: so they can be used over an extended period of time to create (and modify) a range of process definitions and workflow applications.

In general, the process logic domain is likely to be much more dynamic than the application logic domain. Process definitions normally undergo significant modifications as an enterprise implements new business processes, while business functions and shared application services remain relatively unchanged. Process logic is also likely to change much more rapidly than organizational structure.

Fusion makes it easy to modify process definitions and dynamically register them with an engine without changing design elements, such as the user profile, validation, application dictionary, and assignment rule dictionaries. If your system is properly designed, you should be able to modify process definitions without changing these design elements.

However, there might be times when you need to modify design elements to achieve particular changes in process logic. The difficulty of modifying design elements depends on two factors:

- how extensive the modifications are
- whether your production engine can be shut down to move from old design to new, or whether it must support both old and new in a smooth transition

This section lays out some general guidelines for making changes in the various design elements, based largely on the dependencies illustrated in Figure 11 on page 44.

Modifying an Assignment Rule Dictionary

Assignment rules are the most likely candidates for change. You may want to add new conditions to an existing rule or rules, or to add completely new assignment rules. Normally, these modifications are straightforward.

However, if your changes require new process attributes or an extension to the user profile, the impact of the change can be far reaching and more difficult to achieve.

For example, if your changes require new process attributes, then all process definitions using the new assignment rules have to be modified to include the new process attributes and re-registered with your engine. (It is also possible that client applications that create new process instances may have to be revised and redeployed.)

If your changes require modifying the user profile, then almost all your design elements are likely to be affected. The validation, and assignment rule dictionaries, which depend upon the user profile, may have to be modified and re-registered with your engine. (It is also possible that the login code of all client applications has to be modified and the client applications redeployed.)

For more information, see Chapter 5, “Defining Assignment Rule Dictionaries.”

Modifying a User Profile

The most likely reason to modify the user profile is to extend the user profile to include user characteristics needed by one or more assignment rules. (The information needed to provide the extended user characteristics must be stored in the organization database.) Another reason to modify the user profile is to accommodate changes to the organizational structure.

In any case, a change to the user profile is far reaching and should only be undertaken when absolutely necessary. For more information, see Chapter 4, “Defining a User Profile.”

Modifying an Application Dictionary

Generally speaking, you modify application dictionaries to add new application dictionary items in response to an expansion of the application logic domain—that is, to include new application functionality within existing processes or for new processes. It is also possible, however, that you would modify dictionary items to accommodate changes in process attributes. For more information, see Chapter 6, “Defining Application Dictionaries.”

Summary of Process Design Elements

The following table summarizes the main design elements discussed in the previous section and provides additional information about each.

Element	Description	Other Information
User Profile 	Template used to specify characteristics of a user. Engine compares assignment rules with each user's profile when deciding who can perform an activity. Also used to validate user logins. User profile must be consistent with organization database. Validation and assignment rules must be consistent with user profile.	Created in User Profile Workshop. User profile must be registered with a process engine. In some situations, more than one user profile can be concurrently registered. For more information, see Chapter 4, "Defining a User Profile."
Assignment Rule Dictionary 	A set of one or more assignment rules that process developers can associate with activities in one or more process definitions. Assignment rules are used to determine who can perform an activity or create a process instance. Each assignment rule must be consistent with the user profile. One or more assignment rules can be associated with each user-performed activity or process definition.	Created in Assignment Rule Workshop. Assignment Rule dictionaries must be registered with a process engine; any number of dictionaries can be concurrently registered. For more information, see Chapter 5, "Defining Assignment Rule Dictionaries."
Application Dictionary 	A set of one or more application dictionary items that process developers can associate with activities in one or more process definitions. Application dictionary items are used to specify both the work a service provider application must do in performing an activity and the resources required to do it. The process attributes used in a dictionary item must be a subset of the process attributes defined for the process definition in which the dictionary item is used.	Created in Application Dictionary Workshop. Application developers and integrators must know the meaning of all application dictionary items and communicate requirements back to the designer. For more information, see Chapter 6, "Defining Application Dictionaries."
Process Attributes 	Set of attributes assigned to a process definition. Process attributes are used to identify an instance of a process and to perform process logic, such as in router methods, trigger methods, and assignment rules.	Created in Process Definition Workshop. Process attributes are specified for each process definition. The process definition must be registered with a process engine. Process definitions can also be included as suppliers to other process definitions. For more information, see Chapter 7, "Creating Process Definitions."
Validation 	Used to validate user logins and extract user information for user profile from organization database. Validation must be consistent with user profile and organization database.	Created in Validation Workshop. Validation must be registered with a process engine; only one validation can be concurrently registered. For more information, see Chapter 8, "Defining Validations."

Working with the Process Engine

The process engine is the heart of a Fusion development system, shown in Figure 3 on page 28. The engine controls and manages the flow of business processes from beginning to end, coordinating the work of the different client applications that perform the various activities that make up each process instance.

At each stage of a process, the engine evaluates whether an activity is ready to be performed and if so, assigns that activity to the appropriate resources. When an activity is completed, the engine routes the work to the next activity or set of activities.

The engine knows how to manage the flow of a process because it is programmed to control and track that process: the engine executes a process definition, which is created in the development environment and then registered with the engine.

Once a process definition has been registered with the engine, a client application can open a session with the engine and create an instance of the process. The client application provides any data required to start the process instance, and then the engine takes over, assigning activities directly to client application sessions or to queues where they can be accessed. The client applications perform the work required for each activity by using whatever shared application services or desktop applications necessary, and then notify the engine that the activity has been completed.

The relationship of the engine to client applications, on the one hand, and to the process development workshops, on the other, is illustrated in Figure 3 on page 28. Client applications maintain sessions with a process engine in order to initiate a business process or perform activities within it. As process definitions (as well as assignment rule dictionaries, user profiles, and validations) are created or modified, they can be registered with the engine and become the basis for further process execution.

The engine is thus the centerpiece of a process management system, implementing business processes that require many activities to be performed by many users.

Engine Functions

A process engine performs a number of different functions, including:

- managing client sessions

  The engine opens, suspends, and closes client sessions. When opening a session, the engine first validates a user's login against an organizational database.

- executing business processes

  The engine creates instances of a process, and manages their execution from start to finish. During each process, successions of activities are performed by client applications that have opened sessions with the engine. The engine manages and tracks these activities to their final completion.

- registering distributions

  Process definitions, access rule dictionaries, user profiles, and a validation together constitute the process logic components executed by the engine during process execution. Registration lets you change these components dynamically as business processes and organizational structures change.

- maintaining an engine history database

  The engine can maintain a history of each process execution, writing all state changes to an engine history database. The recorded information can include changes of state in activities, process attributes, timers, and so on.

About Registration

Most businesses are not static—they typically create new processes or modify existing ones. Their organizational structures may also change, impacting the rules by which work gets assigned. To provide the business flexibility required, the process engine is able to dynamically load and execute new and revised process definitions, assignment rules, and other programmatic information.

Registration is the procedure by which process logic created in the process development workshops is made available to a running engine.

The entities that get registered with a process engine include the following:

- user profile
- validation
- assignment rule dictionaries
- process definitions

Registration capability is provided by the process development workshops. It lets application system designers and process developers test their work by registering it with a test engine.

Registration capability is also provided by system management tools (Fusion Console and Conductor Script), letting system managers register process definitions, assignment rules, a user profiles, and a validation with production engines in production environments. (For more information, see the *Forte Fusion Process Management System Guide*.)

What Does Registration Do?

The entity actually registered when you register a process definition, assignment rule, user profile, or validation is a *library distribution*.

A library (often referred to as a shared library) is code that can be loaded into memory at runtime, and then referenced by any number of executing programs. A library distribution is the set of distribution files used to install one or more libraries on any particular node. To be loaded, a library must be registered with the program that needs to reference the code it contains.

Library distributions are installed on nodes hosting process engines and registered with the engines. The engines can then dynamically load and execute the libraries.

Your library distributions are generated automatically by designers or developers using commands in the process development workshops. For a description of the workshops, see Chapter 2, “Getting Started: the Process Development Workshops.” Code in the central development repository is extracted and made into library distribution files, which are then placed in a standard location on the central server node.

The same commands can also be used to register library distributions with a test engine. Registration of at least one user profile, validation, and assignment rule dictionary is needed to test a process definition, and to test any corresponding client applications.

Registration consists of two steps, both transparent to the user:

1. installation of library distribution files on the node hosting the target engine
2. placing an entry in the registration table of the target engine’s database

Registration Sequence

The distributions you register with an engine can be registered in any order with one exception: user profiles should be registered before the validations or assignment rule dictionaries that depend upon them. (See “Extended User Profile as Supplier” on page 73, for the conditions under which this exception holds.)

Since design elements are created before the process definitions that use them, it is normal for a user profile, validation, and assignment rule dictionary to be registered in that order before process definitions are registered. However, in the course of development, assignment rules may change, or new ones may be developed, and these can generally be registered at any time.

Engine Registration Manager

Each engine has a registration manager that tracks library distributions registered with the engine. The manager ensures that all engine references to a library are references to the most current registered version of the library.

For example, if you register a new version of a process definition with an engine, all subsequent instances of that process definition created by that engine are based on the new version. Process instances based on the older version, however, continue to execute to completion (process termination). When instances of the older version no longer exist, the engine registration manager automatically unregisters the old process definition.

When you register a new version of an assignment rule dictionary with an engine that has an older version already registered, the new versions of assignment rules are retroactively applied to all existing offered and queued activities. Offered activities are re-offered to sessions based on the new rules, and access to activities in queues is also governed by the new rules. The engine registration manager automatically unregisters the old versions of any assignment rules included in the new assignment rule dictionary.

Unregistering a library distribution removes the corresponding entry from the registration database table, but does not delete the library files from the engine unit’s host server node.

Chapter 2

Getting Started: the Process Development Workshops

This chapter introduces the Fusion process development workshops and explains how they relate to one another. It also describes how to enter, exit and perform basic tasks in the workshops—the Fusion development environment.

The process development workshops include the following:

- Repository Workshop
- User Profile Workshop
- Validation Workshop
- Assignment Rule Workshop
- Application Dictionary Workshop
- Process Definition Workshop

Introduction to the Process Development Workshops

The Fusion process development environment consists of graphical tools—the process development workshops—for implementing the process logic required for a Fusion enterprise application. These workshops are briefly described below:

Repository Workshop This workshop is the central point of access for the remaining workshops: to perform work in any other workshop, you must first launch the Repository Workshop. By providing access to the central development repository, this workshop lets you manage all Fusion components, including exporting them from the repository or importing them back into the repository.

User Profile Workshop This workshop is used primarily by application system designers to enhance the user profile supplied with Fusion. The workshop is used to specify the important user information needed by assignment rules to determine who should be permitted to perform process activities.

Assignment Rule Workshop This workshop is used primarily by application system designers to create assignment rules and group them into dictionaries. The workshop is used to specify simple role-based rules as well as more sophisticated rules. More sophisticated rules—that depend on user profile attributes, process attribute values, or any condition that might determine user access to one or more activities—are implemented in the workshop by writing an Evaluate method. The assignment rule dictionaries are needed by process developers to specify activity properties.

Application Dictionary Workshop This workshop is used primarily by application system designers to create application dictionaries. The workshop is used to create items that define the work associated with activities, including the process attributes needed to perform the work. Application dictionaries are needed by process developers to specify activity properties.

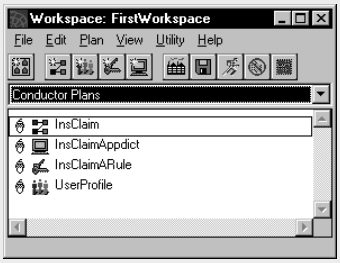
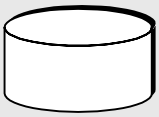
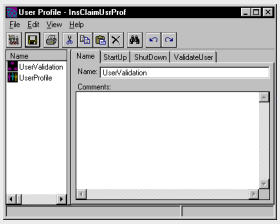
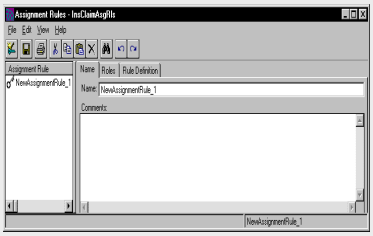
Process Definition Workshop This workshop is used primarily by process developers to create process definitions. It is a tool in which the developer creates a visual model of a process and specifies the properties of each of the components of the model: activities, routers, timers, and so on. The workshop is also used to write methods that specify process logic, such as trigger methods and router methods. This is also where process attributes, an important design element, are specified.

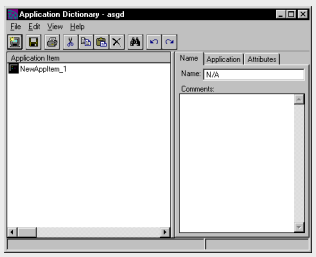
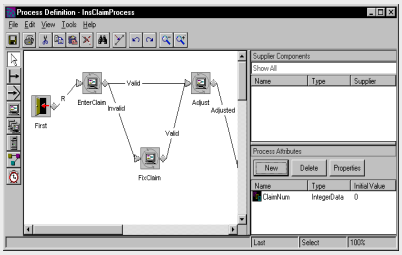
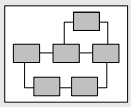
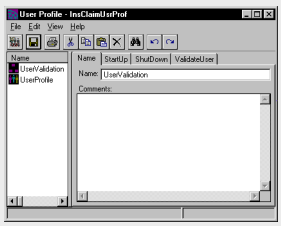
Validation Workshop This workshop is used primarily by application system designers to write validation methods that authenticate users when they open a session with the Fusion process engine, and which may also populate users' profiles by extracting data from an organization database.

XML/XSL Workshop This workshop is an interactive tool for creating, editing, testing, and debugging the XSLT rules that you register with the Fusion process engine. XML rules are supplied to a Fusion Backbone for use in Fusion application proxies. You can create XML and XSLT documents in the workshop, or import the documents from files. The XML documents you use to test your XSLT rules can be actual proxy documents or sample documents that you create just for test processing purposes.

The XML/XSL Workshop is only available to systems that have installed a Fusion Backbone. For more information on this workshop, refer to the XML/XSL Workshop section of the Fusion Backbone online help.

The workshops, their main purpose, and the relationships between them are summarized in the following table.

Workshop	Purpose	Relationships
Repository Workshop 	Manage Fusion components in a central development repository. **Central Repository** 	Provides access to the remaining five workshops and lets you export/import components developed in other workshops from/into the central development repository. For more information, see Chapter 3, "Managing Fusion Plans: the Repository Workshop."
User Profile Workshop 	Create a user profile. **User Profile** 	User profile attributes and methods are used in defining assignment rules and the validation. User profiles must be registered with a Fusion process engine: any number of user profiles can be concurrently registered. For more information, see Chapter 4, "Defining a User Profile."
Assignment Rule Workshop 	Create assignment rule dictionaries. **Assignment Rule Dictionary** 	Assignment rule dictionaries depend upon the user profile; *extended* user profiles must be included as suppliers to assignment rule dictionaries. Assignment rules are used in creating process definitions. Assignment Rule dictionaries must be registered with a Fusion engine: any number of dictionaries can be concurrently registered. For more information, see Chapter 5, "Defining Assignment Rule Dictionaries."

Workshop	Purpose	Relationships
Application Dictionary Workshop 	Create application dictionaries. **Application Dictionary** 	Application dictionary items are used in creating process definitions. Application dictionaries are not registered with a Fusion process engine. Instead they are compiled directly into the process definition in which they are used. For more information, see Chapter 6, "Defining Application Dictionaries."
Process Definition Workshop 	Create process definitions. **Process Definition**  **Process Attributes** 	Process definitions reference assignment rules and application dictionary items; assignment rule dictionaries and application dictionaries must be included as suppliers to process definitions. Process definition components can be used in other process definitions; they should be included as suppliers to those process definitions. Process attributes are defined in the Process Definition Workshop, but are also used in defining application dictionary items and assignment rules (and are redefined in those contexts.) Process definitions must be registered with a Fusion process engine: any number of process definitions can be concurrently registered. For more information, see Chapter 7, "Creating Process Definitions."
Validation Workshop 	Create a validation. **Validation** 	The validation's ValidateUser method depends upon the user profile; *extended* user profiles must be included as suppliers to a validation. A validation must be registered with a Fusion process engine; only one validation can be concurrently registered. For more information, see Chapter 8, "Defining Validations."

Workshop Road Map

The process development workshops as a whole support the various tasks needed to implement the process logic required for a Fusion enterprise application: design, development, and repository management. The workshops also support important runtime operations required for testing or production-level execution—namely, the registration of user profiles, validation, assignment rule dictionaries, and process definitions with a Fusion process engine.

The relationships between the workshops and the different functions they support are shown in Figure 12. The illustration highlights the following points:

- The Repository Workshop is the point from which you access the remaining five workshops.
- An *extended* user profile is a supplier to the Assignment Rule and Validation workshops. The assignment rule's Evaluate method and the validation's ValidateUser method depend upon user profile attributes and methods. (For information on extended user profiles, see "Extended vs. Standard User Profile" on page 72.
- Assignment rule dictionaries and application dictionaries are suppliers to process definitions.
- Registrations of user profiles, validation, assignment rule dictionaries, and process definitions can be performed from the workshop in which each is created.

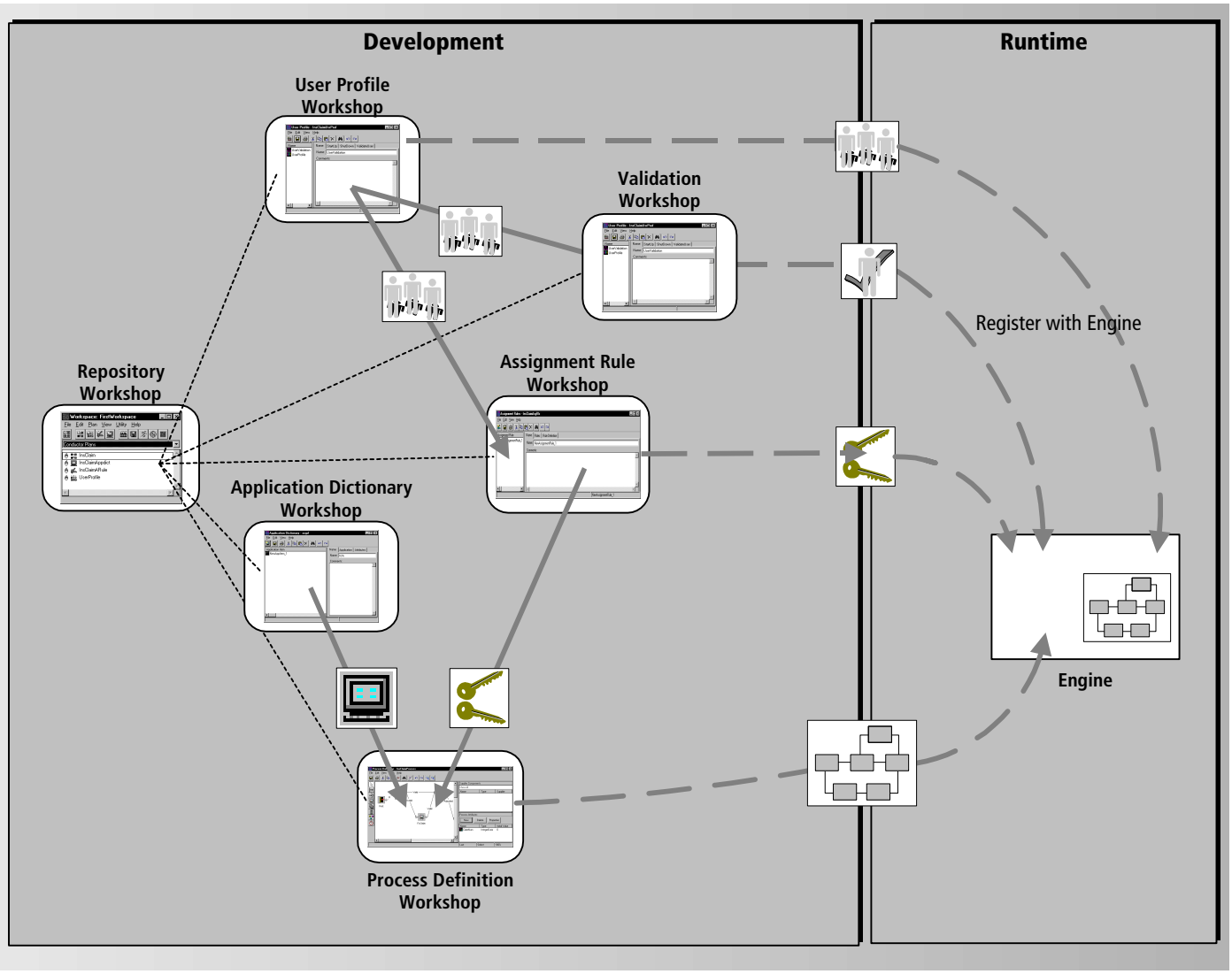


Figure 12 *Relationships Between Process Development Workshops (and Engine)*

Workshop Products: Plans, Projects, Library Distributions

Most of the process development workshops have, as their end product, a library distribution that is registered with one or more engines. Along the way, however, a couple of other, intermediate products are created, as shown in Figure 13.

The type of plan you create depends on the workshop: the User Profile Workshop creates user profiles, the Assignment Rule Workshop creates assignment rule dictionaries, and so on. The plan is stored in the development repository.

When you compile a plan in the workshops (**File > Compile** command) the workshop creates a Forte TOOL project and compiles it. The project is named after the plan, with a suffix appended depending on the plan type (user profile, validation, process definition, and so forth). The project is also stored in the development repository.

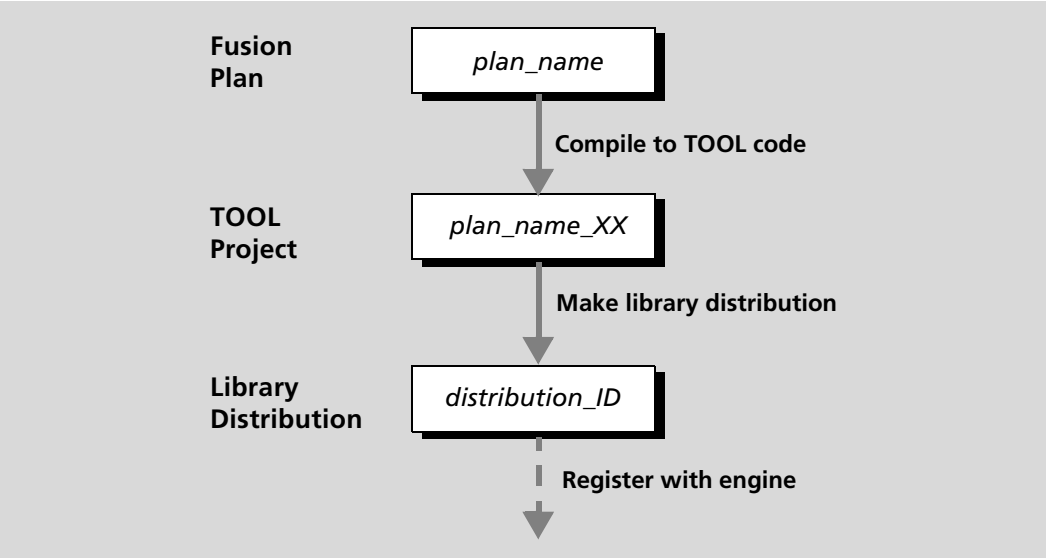


Figure 13 *Process Definition Workshop Products*

When you distribute a model in the workshops (**File > Distribute** command) the workshop first performs the **File > Compile** command described above, and then creates a library distribution from the project. The library distribution is stored in the following location on the central server node in your Fusion environment:
FORTE_ROOT/appdist/*environment_ID*/*distribution_id*/

The **File > Distribute** command also gives you the option of registering the distribution with one or more engines. When you choose to register the library distribution with an engine, it is installed on the nodes hosting the target engine and an entry is placed in the registration table of the target engine's database (see "About Registration" on page 49).

Entering and Leaving Workshops

Before Using Fusion

Before you can use Fusion for process development, you must have a fully functional Forte environment in which Fusion process management system software has been installed by your Fusion system manager.

The node on which you are working should be installed, minimally, as a Fusion development client node. (If your node is an engine server node or a central server node, then development client software should also be installed.) See the *Forte Fusion Process Management System Guide* for a list of distributions required by the process development workshops.

In addition, you must have access to a development repository containing the Fusion plans and libraries needed for Fusion process development. In most cases, your system manager sets up a Fusion central development repository in your environment and sets a Forte environment variable to point to that repository.

Starting the Process Development Environment

When Fusion is properly installed, you access the Fusion process development workshops by starting up your Forte development environment. Forte detects the presence of Fusion components, and modifies the development environment accordingly. For information on installing and setting up Fusion, refer to the *Forte Fusion Installation Guide* and the *Forte Fusion Process Management System Guide*.

Depending on your installation and platform, you can enter the Fusion process development environment from the Windows Start menu or by using the command line. For UNIX and OpenVMS platforms, only the command line method is available.

forte command

The following command line starts the Fusion development environment:

Portable syntax

forte [**-fs**] [**-fr** *repository*] [**-fw** *workspace*] [**-fnd** *node_name*]
[**-fmn** *model_node_name*] [**-fm** *memory_flags*] [**-fl** *logger_flags*] [**-fcons**]

On VMS systems, use the following syntax:

OpenVMS syntax

VFORTE FORTE
[**/STANDALONE**]
[**/REPOSITORY**=*repository_name*]
[**/WORKSPACE**=*workspace_name*]
[**/NODE**=*node_name*]
[**/MODEL_NODE**=*model_node_name*]
[**/MEMORY**=*memory_flags*]
[**/LOGGER**=*logger_flags*]
[**/FCONS**]

The following table explains each of the command line flags:

This Flag	Specifies
-fs /STANDALONE	Run the development session in stand-alone mode. In stand-alone mode, the node is not connected to the distributed development environment, and you cannot use a central repository. The default mode is *distributed*, which connects you to the name server and environment manager (see the *Forte 4GL System Management Guide* for information about the distributed development environment).
-fr *repository* **/REPOSITORY=***repository_name*	The repository to be used for the development session. Information on specifying a repository follows this table.
-fw *workspace* **/WORKSPACE**=*workspace_name*	The workspace for the development session. Information on specifying a workspace follows this table.
-fnd *node_name* **/NODE**=*node_name*	Specifies the node name to use for this session. If you do not specify the node name in the **forte** command, the default node name depends on the operating system. On Windows, the default node name is set by the FORTE_NODENAME environment variable. On all other platforms, the actual node name is used.
-fmn *model _node_ name* **/MODEL_NODE**=*model_node_name*	Specifies the model node name to use for this session. If you do not specify a model node name in the **forte** command, Forte uses the value of the FORTE_MODELNODE environment variable. If the environment variable is not set, the node is not treated as a model node.
-fm *memory _flags* **/MEMORY=***memory_flags*	Specifies the space to use for the memory manager. See Appendix B in *A Guide to the Forte 4GL Workshops* for details.
-fl *logger_flags* **/LOGGER=***logger_flags*	Specifies the logger flags to use for the session. See Appendix B in *A Guide to the Forte 4GL Workshops* for details. If you do not set the logger flags in the **forte** command, Forte uses the value of the FORTE_LOGGER_SETUP environment variable. Note that you can change the logger settings from the Repository Workshop.
-fcons /FCONS	Displays the trace window. By default, the trace window is iconified on Windows. Use this flag to display the trace window on startup.

Following are examples of the forte command:

```
forte -fs
forte -fr bt:$FORTE_ROOT/repos/examples
forte -fl "%stdout(trc:os:1:1 trc:err)" -fm "(n:4000,x:8000)"
```

Selecting a repository

To specify the repository in the forte command:

- For a central repository, specify a repository service name. See your system manager for information about your central repository.
- For a private B-tree repository, specify the repository name using the following format: bt:p*rivate_repository_name*.
- For a shadow repository, specify the repository name using the following format: bt:s*hadow_repository_name*.

See *A Guide to the Forte 4GL Workshops* for information about the different kinds of repositories.

Default repository

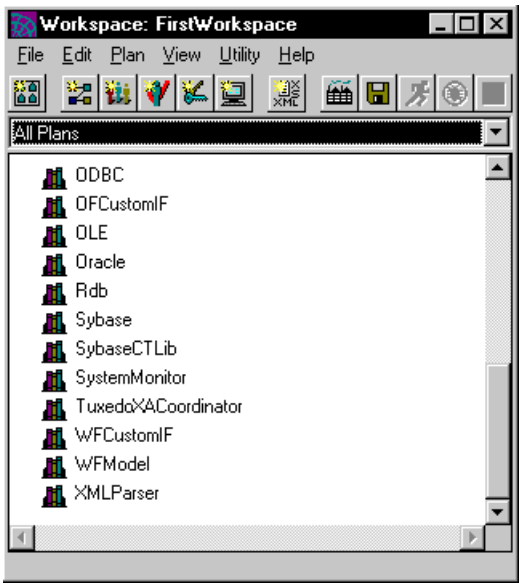
If you do not specify a repository with the forte command, the default repository depends on the operating system. For all platforms, Forte uses the setting of the FORTE_REPOSNAME environment variable. If the environment variable is not set, Forte uses the default distributed repository called *CentralRepository*. For Windows platforms, Forte uses the last repository you opened in the Repository Workshop.

Selecting a workspace

If you do not specify a workspace with the forte command, the default workspace depends on the operating system. For all platforms, Forte uses the setting of the FORTE_WORKSPACE environment variable, and if the environment variable is not set, Forte opens the Repository Workshop without a workspace. For Windows platforms, Forte uses the last workspace you opened in the Repository Workshop.

The Repository Workshop

If the Fusion development environment started successfully, the Repository Workshop displays on your screen.



Your Fusion installation provides additional functionality to the Forte Repository Workshop that allows you to do Fusion process development. For general information about using the Forte Repository Workshop (and other Forte Workshops), see *A Guide to the Forte 4GL Workshops*. If you have Forte Express installed, see *A Guide to Forte Express* for information on the Forte Express workshops.

Starting the Remaining Process Development Workshops

For Fusion plans that have been previously defined, you can double-click a plan (in the list of plans) to launch the associated process development workshop.

To open a workshop to create a new plan, you can either choose the type of plan from the Plan menu or click its corresponding tool button in the toolbar.

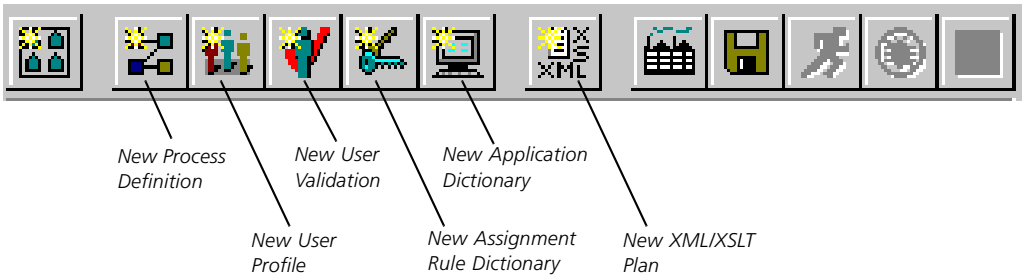


Figure 14 *Repository Workshop Toolbar Buttons*

For example, to create a new process definition plan, open the Process Definition Workshop either by choosing **Plan > New Process Definition** or by clicking the **New Process Definition** button in the toolbar.

Before the workshop opens, a dialog prompts you for a name for the new process definition. Enter a name and click **OK**.

For more information on using the Repository Workshop, see Chapter 3, “Managing Fusion Plans: the Repository Workshop.”

Leaving the Process Development Workshops

Close and Exit commands

You can leave the workshops individually by choosing **File > Close**. If you want to leave the workshops altogether, in the Repository Workshop, choose **File > Exit**. If any of your workshops have unsaved changes, you are asked if you want to save them first.

Process Development Workshops Overview

The process development workshops for creating Fusion design elements—the User Profile, Assignment Rule, Application Dictionary, and Validation Workshops—all have a common user interface appearance and screen elements, described in the following paragraphs. The other Fusion process development workshops—the Repository Workshop and the Process Definition Workshop are described in their respective chapters.

An example design workshop—the Assignment Rule Workshop—is shown below:

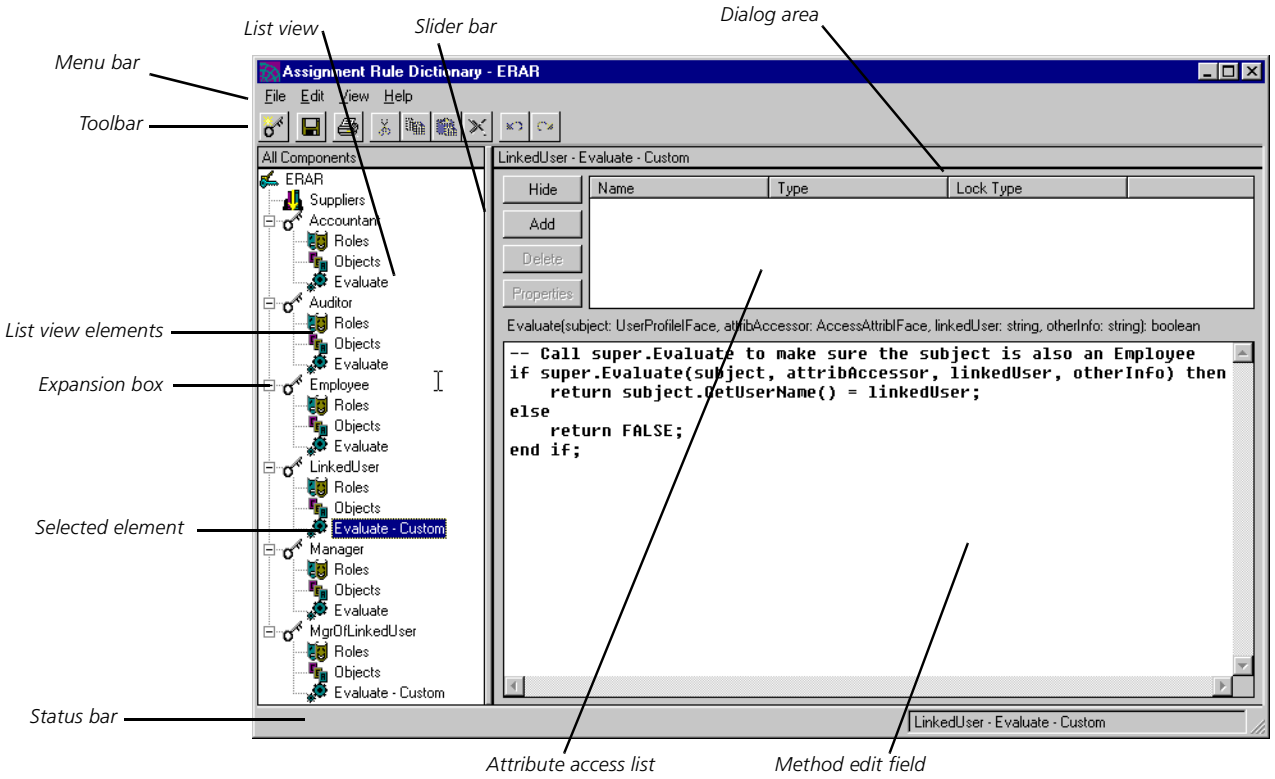


Figure 15 *Sample Workshop User Interface Elements*

The workshop window has two main areas:

List View The list view on the left side of the workshop is a hierarchical view of all the elements that can be created and modified in the workshop. The root level of the list view represents the plan defined in the workshop (in this case, an assignment rule dictionary). At the next indented level are the various elements that comprise the plan (in this case, the suppliers and the individual assignment rules that are defined for the assignment rule dictionary). At the next indented level are the individual elements defined for each assignment rule (in this case Roles, Object attributes, and an Evaluate method). To view the sub-elements comprising an element, click on the expansion box for the element. The expansion box changes from a + to a – to indicate that it has been opened.

Dialog Area The dialog area on the right side of the workshop displays a dialog corresponding to the element selected in the list view. The dialog can be a property inspector or other screen needed to define the selected element. In the case of Figure 15, for example, the Evaluate method corresponding to a LinkedUser assignment rule has been selected; the dialog area displays the fields needed to define that method. In this case, the dialog consists of a field for specifying the method's attribute access list and a field for entering or editing method text.

In Figure 15, the Evaluate method element selected in the list view is designated as "Custom" to differentiate it from default method implementations in the list view.

The relative size of the list view and dialog areas can be changed using the slider bar positioned between them.

The workshop also contains a toolbar for performing common operations in the workshop. The buttons in the toolbar represent commands found in the menus at the top of the workshop.

The User Profile, Assignment Rule, Application Dictionary, and Validation Workshops all contain the user interface elements shown in Figure 15. Each workshop, and how to use it, is described in more detail in its respective chapter of this book.

Cut, Copy, and Paste

You can cut, copy, and paste list view elements from one Fusion plan to another. For example, you can copy an assignment rule from one assignment rule dictionary to another, or an application dictionary item from one application dictionary to another. You can also copy text. For example, you can copy the Evaluate method text from one assignment rule to another.

Undoing Work

All the process development workshops let you undo work you have performed, but do not want to save, using either the Undo/Redo or the Cancel commands.

Undo/Redo You can undo one or more sequential operations by selecting the **Edit > Undo** command one or more times. Similarly you can restore the same operations by selecting the **Edit > Redo** command.

Note The Undo/Redo commands do not apply when you drag and drop assignment rules or application dictionary items from the Supplier Component list to the layout area of the Process Definition Workshop. To undo a drag and drop operation, you must use the Cancel command.

Cancel You can undo all operations since the last save command using the **File > Cancel** command. This discards all changes since the last save operation.

Online Help

Online help is available for Fusion from any of the process development workshops.

- To display the Contents tab dialog for Fusion process development help, choose **Help > Help Topics**.
- To display help for the current window, press the **F1** or **Help** key (depending on your operating system platform).

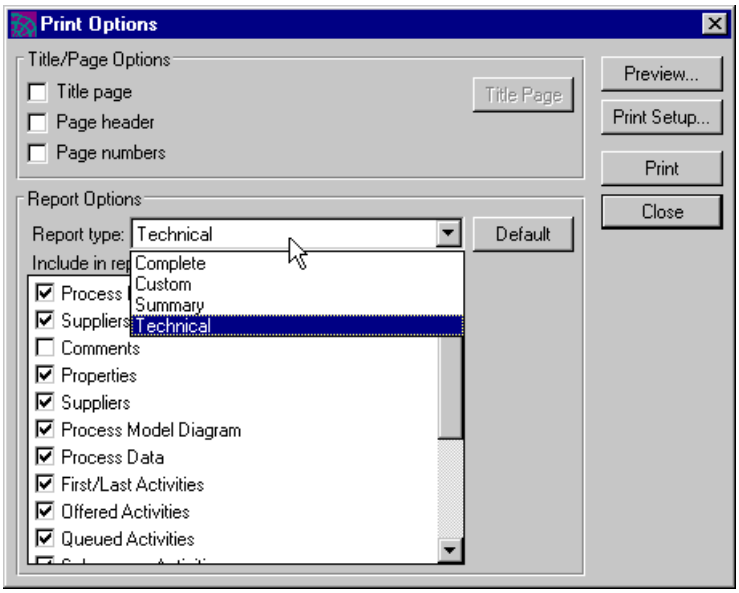
Printing Reports

You can print reports that present details of plans developed in the process development workshops. For example, in the Process Definition Workshop, you can print a process diagram and list the elements of a process definition, including all element properties. In addition, you can customize any report before printing it.

To print a report:

1 In any Fusion development workshop, choose **File > Print**.

The Print Options window displays:



2 Set the appropriate options according to the descriptions in the following table:

Option	Description
Title page	Enable this option to add a title page to your report. Then click the TItle Page button (see "Creating a Title Page" on page 64).
Page header	Enable this option to print the report title and the current date and time on all but the first page of the report.
Page number	Enable this option to print page numbers on all but the first page of the report.
Title Page button	Click this button to display the Title Page dialog, where you add and define a title page for the report.
Report Type	Select the report type from the following: **Complete**—include all the items available for the report **Custom**— select individual items from the Include in Report droplist for inclusion in the report **Summary**—produce a report that includes the following items: Process Model, Comments, Process Model Diagram, First/Last Activities, Offered Activities, Queued Activities, Subprocess Activities, Junctions, Timers, Method/Expressions, Routers, Timers Links. **Technical**—produce a report that includes Suppliers and Properties in addition to items included in the Summary report.
Default button	Set the report's options back to the default options for the currently selected report type.
Include in report	Display available items for inclusion in a report. You can customize any report by enabling or disabling report options. If you select a Custom report no options are included; you must select the items you want to print.
Preview button	Display the report online before printing.
Print Setup button	Displays the windowing system's Print Setup dialog.

Previewing the report

3 (Optional) To view the report before printing it, click **Preview**.

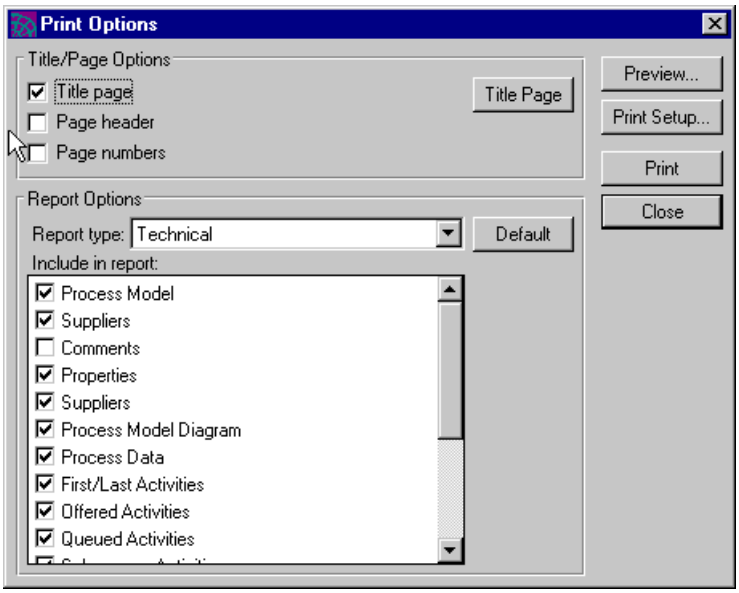
4 Click **Print** to print the report.

Creating a Title Page

To add and define a title page to a report:

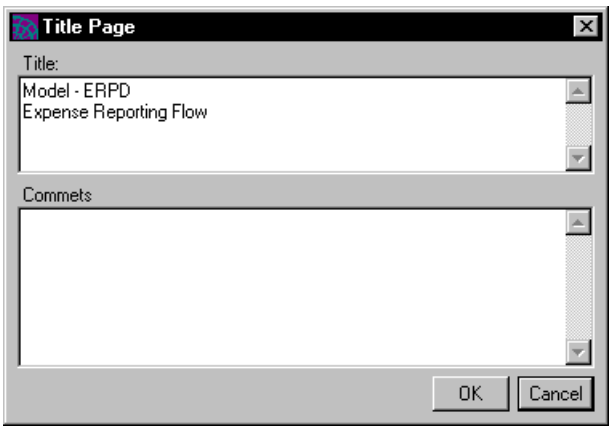
1 Choose **File** > **Print**.

The Print Options dialog appears.



2 In the Print Options dialog, enable the Title Page option and click the **Title Page** button.

The Title Page dialog appears.



3 In the Title field, enter the title of the report.

4 In the Comments field, enter any information comments about the report.

5 Click **OK**.

Preview button

6 To see how the report will look when you print it, click **Preview**.

7 To print the report, click **Print**.

Chapter 3

Managing Fusion Plans: the Repository Workshop

Fusion design elements—user profiles, validations, assignment rule dictionaries, and application dictionaries—and Fusion process definitions are all stored in your development repository.

This chapter provides information about using the Repository Workshop to manage the Fusion plans in your development repository. Specifically, this chapter covers the following topics:

- perform standard repository, workspace, and plan tasks
- open Fusion plans
- check out and branch Fusion plans
- import and export Fusion plans
- compile all out-of-date plans

The complete description of this workshop, including how to work with repositories and workshops, is in *A Guide to the Forte 4GL Workshops*.

Using the Repository Workshop

When you start the Fusion process development environment (as described in "Entering and Leaving Workshops" on page 57), the Repository Workshop opens.

In this workshop, you manage the various Fusion plans (as well as any Forte plans and projects) in your development repository. The Repository Workshop also provides access to the remaining process development workshops—the User Profile, Validation, Assignment Rule, Application Dictionary, and Process Definition workshops. From this workshop, you can perform the following tasks:

- create a workspace
- update a workspace
- open Fusion plans
- save all your work
- check out and branch plans to get write access to them
- import and export plans
- compile plans

This chapter describes briefly how to perform each of these tasks. It does not explain what repositories and workspaces are, nor does it go into detail about how to use the workshop. See *A Guide to the Forte 4GL Workshops* for complete information on this workshop and on repositories and workspaces.

Creating and Opening Workspaces

When you start the Repository Workshop, it opens the current workspace. You can use the Forte Control Panel to designate which workspace to open when you start the workshop. By default, this workspace is called FirstWorkspace.

New Workspace command

You can create a new workspace by choosing **File > New Workspace** and naming your workspace. When your new workspace opens, you can immediately begin working with the process development workshops.

Open Workspace command

You can open an existing workspace by choosing **File > Open Workspace** and picking a workspace from the list that is displayed.

Updating a Workspace

As with any Forte plan, if someone else has changed and checked in a Fusion plan that you use for process development (and that you have therefore included in your workspace), you must update your workspace to be able to use the changed plan. (For example, someone working in another workspace but in the same repository is developing the application dictionary that you use as a supplier to a process definition.)

Update Workspace command

To update your workspace, choose **File > Update Workspace**. All the plans in your workspace are updated and optionally compiled. Recompiling is a good idea, just to ensure that the changes received from the repository work with your plans and projects.

Creating and Opening Fusion Plans

When you create a new Fusion plan or open an existing Fusion plan the appropriate process development workshop is opened.

Creating New Plans

You can create new Fusion plans by selecting commands in the Plan menu or by clicking the appropriate button in the toolbar at the top of the workshop. The toolbar selections for creating new plans are shown in Figure 14 on page 60.

When creating a new plan, a dialog first prompts you to name the plan. After providing a name, the appropriate workshop for the plan opens.

Opening an Existing Plan

To open an existing Fusion plan in its associated workshop:

1 Use any of the following methods:

- double-click the plan name in the list of plans
- select the name of the plan in the list of plans and choose **Plan > Open**
- select the name of the plan in the list of plans and press Enter

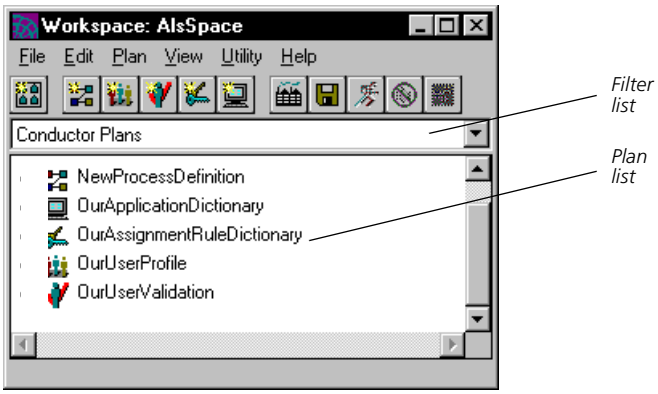


Figure 16 *Fusion Plans in the Repository Workshop*

Note Be sure to select the Fusion plan and not its compiled project—the compiled project is read-only generated TOOL code. To ensure that only the plans are listed, choose Fusion Plans from the filter list as shown in Figure 16.

Saving Plans

Save All command

You can save all the plans you have changed in the current workspace by choosing **File > Save All**. It is a good idea to save your work often—there is no automatic save in the workshops. (You can also save the contents of all workshops by choosing **File > Save All** from any workshop.)

If you attempt to exit the Repository Workshop with unsaved changes, a dialog asks you if you want to save your workspace before exiting.

Checking out and BranchingFusion Plans

To get write access to an existing Fusion plan, you must either check out or branch the entire plan. You check out a plan to make permanent changes. You branch a plan to make temporary changes or if the plan is already checked out by another user.

The Repository Workshop provides these capabilities with the **Plan > Checkout** and **Plan > Branch** commands. The Repository Workshop also provides commands (described in "Undoing Changes to a Plan" on page 69) that let you:

- revert changes you have made to a plan that you checked out or branched
- undelete a plan that you deleted after checking it out

Checking out a Plan

Checkout command

The Checkout command in the Repository Workshop gives you an exclusive write lock on a Fusion plan. Before you can check out a plan, the workspace must be open for modifying. Only one workspace can check out a plan at a time. After you have finished your modifications, you do not have to check the plan back in. It is automatically checked in when you integrate your workspace (**File > Integrate Workspace**).

▶ **To check out a Fusion plan:**

1. In the plan browser, select the Fusion plan you want to check out.
2. Choose **Plan > Checkout**.

When the plan is checked out, the plan browser displays a checkout icon by the plan name.

The Checkout command fails if the plan is already checked out by another workspace. The error message gives the name of the workspace that has checked out the plan. Checkout also fails if the workspace does not have the latest version of the plan. In this case, use the **File > Update Workspace** command to bring the latest version of the plan into your workspace.

If you want to undo the changes you have made since the Checkout command, and revert the plan to the state it was in after your last **File > Update Workspace** command, you can use the **Plan > Undo Checkout/Branch** command as described under "Undoing Changes to a Plan" on page 69.

Branching a Plan

Branch command

The Branch command gives you temporary write access to a Fusion plan so you can test a change while someone else has the plan checked out. You cannot integrate a changed, branched plan into the system baseline.

Before you can branch a plan, the workspace must be open for modifying. Any number of workspaces can branch a plan at the same time.

▶ **To branch a Fusion plan:**

1. In the plan browser, select the Fusion plan you want to branch.
2. Choose **Plan > Branch**.

When the plan is branched, the browser displays a branch icon by the plan name.

If you have already checked out the plan you branch, the checkout is converted to a branch. When a checkout is converted to a branch, your changes to the plan are retained in the workspace, but the plan is free to be checked out by another workspace after the next **File > Save All** command.

If you want to undo changes you have made since a Branch command, reverting the plan to the state it was in before your last Branch command, you can use the Undo Checkout/Branch command described in the next section.

Undoing Changes to a Plan

Choosing **Plan > Undo > Checkout/Branch** erases all changes you made to a plan since the last Checkout or Branch command. The plan reverts to the state it was in before the Checkout or Branch command, and the plan is freed for checkout by another workspace.

To revert a plan:

1. In the Plan browser, select the plan you want to revert.
2. Choose **Plan > Undo > Checkout/Branch**.

Importing and Exporting Fusion Plans

To move a plan from one repository to another, you must export it from your current repository to a text file (.pex file) and then import the .pex file into the other repository. This may be necessary, for example, when an application dictionary and an assignment rule dictionary are developed using one repository and an associated process definition is developed using another repository (not recommended, but sometimes unavoidable).

Export command

To export a plan to a .pex file, select the plan you want to export, then choose **Plan > Export**. Enter the name of the file without an extension and take the default file type, "Exported Project Files." The file will be saved with a .pex extension.

Import command

To import a plan from a text file, first check that the file is available to your system, then choose **Plan > Import**. If you get an error saying that a required library cannot be found, try including the library explicitly in your workspace (choose **Plan > Include Public**) and then do another import.

Compiling Plans

Compile All Plans command

The **File > Compile All Plans** command compiles all Fusion plans that have changed since your last compilation. The plans are compiled into TOOL projects (see "Workshop Products: Plans, Projects, Library Distributions" on page 56).

This command generates TOOL code for each Fusion plan, compiles the TOOL code, and saves the code in a read-only project. The TOOL project name has an extension, depending on the type of Fusion model being compiled, as shown in the following table.

Extension	Model type
UP	User profile
UV	Validation
AR	Assignment rule dictionary
AD	Application dictionary
PD	Process definition

Force Compile command

If you want to compile all your Fusion plans, regardless of whether they have changed, choose **Utility > Force Compile**. The submenu for the **Force Compile** command lets you choose either All Plans, to force compilation of all plans in the workspace, or Selected Plan, to force compilation of the currently selected plan.

Compile Error

If you try to compile a Fusion plan and you receive an error message similar to the following, it means that the library referred to could not be found in your workspace.

There are many reasons why the library could not be found: the library might have been deleted from your repository, you might be using an old repository that does not contain the library, or the library simply has never been included in your workspace.

If you see this prompt, first try to include the missing library in your workspace.

To include the missing library in your workspace:

1. Go to the Repository Workshop.
2. Choose **Plan** > **Include Public**.
3. In the Include Public Plan dialog, select the library you want to include, and click OK.

If you cannot find the library in the Include Public Plan dialog, you have to import it into your repository.

To import the missing library:

1. Exit the error dialog.
2. Go to the Repository Workshop.
3. Choose **Plan** > **Import**.
4. Navigate to the FORTE_ROOT\userapp*library_name*\cl0 directory and import *library_name*.pex.

After the library has been imported, you can include it in your workspace. Then return to the workshop you were previously working in and compile the plan again.

Chapter 4

Defining a User Profile

This chapter describes what a user profile is and how to define one with the User Profile Workshop. In particular, the chapter covers the following topics:

- description of a user profile
- using the User Profile Workshop
- creating new versions of a user profile
- UserProfile class reference

About User Profiles

A user profile provides a template that is used in the following ways:

- to hold a user's name, roles, and other important user characteristics
- to pass user information around within a Fusion process management system
- to evaluate user information (the template carries a set of methods for that purpose)

A user profile plays a key role in two operations within a Fusion process management system:

- Authentication of users who are opening sessions with a Fusion process engine (by using the ValidateUser method of a validation)
- Determining who is permitted to perform the various activities in an executing process (by using assignment rules)

For an overview of user profiles and how they fit into a Fusion process management system, see "User Profile Design Concepts" on page 40.

Extended vs. Standard User Profile

The user profile you create in the User Profile Workshop is derived from a base UserProfile class —it is a subclass of the UserProfile class. Your user profile embodies all the attributes and methods of the UserProfile class, plus any enhancements that you make to it.

You do not have to make enhancements to the UserProfile class; you can use it as is.

The UserProfile class defines a number of methods. These methods are used by assignment rules (described in Chapter 5, "Defining Assignment Rule Dictionaries") to check if a user's profile matches a given assignment rule. These methods are also used by a validation to check a user login against an organization database and populate the user's user profile. The methods are described in detail under "UserProfile Class" on page 81. You do not need to customize these methods; the default implementation can be used without modification in defining assignment rules and a validation.

Standard user profile

A user profile also has a built-in array that can handle a set of roles. Therefore, if your user information consists simply of a user name and a number of roles, you can use this *standard* user profile without alteration. If you want to add other user characteristics, such as signing authority or manager name, you can use the User Profile Workshop to extend the user profile attributes.

Extended user profile

If you enhance a user profile by adding user profile attributes, then you are creating an *extended* user profile. You must treat extended user profiles differently from standard user profiles (those which at most have customized UserProfile methods): *You must explicitly include an extended user profile as a supplier library to assignment rule dictionaries and validations that reference the user profile.*

If you create an extended user profile, you will have to perform the procedures outlined below for making it a supplier library to assignment rules and validations. If you create a standard user profile (at most you customize UserProfile methods), you do not have to perform these procedures.

In any case—whether or not your user profile is extended—your user profile must be registered with the Fusion process engine. Even if you are using a standard user profile with default methods, you need to open the User Profile Workshop, create a user profile of a unique name, and use the workshop commands to register it with your engine (see "Making and Registering User Profile Library Distributions" on page 78).

Extended User Profile as Supplier

As described in "Design Element Dependencies" on page 44, extended user profiles created in the User Profile Workshop are needed by both assignment rules and validations.

Because of this dependency, and because the assignment rules and validations (and the user profiles they depend on) are dynamically loaded and executed by the Fusion process engine, it is necessary that an extended user profile be a supplier *library* to assignment rule dictionaries and validations (as shown in Figure 11 on page 44).

The requirement that the user profile be a supplier library carries with it increased procedural overhead. In particular, to include the user profile as a supplier, you have to first create it, compile it into a TOOL project, and generate a user profile library distribution (see Figure 13 on page 56). Then you have to import the user profile *library* back into your development repository. This procedure is described in more detail in "Including a User Profile as a Supplier Library" on page 79.

Multiple User Profiles: Rolling Upgrades

Normally there is only one registered user profile per engine. This is less a matter of principle than it is of recommended design. You want to design a user profile that is consistent with your organization database, which can be used by all client applications to open sessions with the engine, and by all assignment rules to decide which users are permitted to perform activities in executing processes.

Nevertheless, sometimes more than one user profile must be supported. For example, in the course of an application's life cycle, you may need to enhance a user profile, say, to include additional user characteristics (user profile attributes) required by some new assignment rules. In this case, you can maintain two user profiles for some period of time: one that supports the old design of your workflow applications and one that supports the new one.

In this scenario, both user profiles are registered with the engine as you incrementally upgrade your applications from the old version to the new one. In this type of rolling upgrade you do not have to shut down your production system. The old user profile is used by old client applications and the new user profile is used by new client applications.

In a rolling upgrade, you also have to modify your assignment rules and your validation to support both the old and new user profiles, and then register the modified assignment rule dictionaries.

Note To register more than one user profile with an engine, each user profile must have a unique name. Therefore, when you modify or upgrade a user profile with the intent of performing a rolling upgrade, you should give it a unique name. Otherwise you can only perform a monolithic upgrade, in which all client sessions are shut down and Fusion distributions are unregistered.

Working with a User Profile

This section describes the series of tasks you are likely to perform when you create or modify a user profile. It is followed on page 81 by a reference section on the UserProfile class.

This section covers the following topics:

- opening the User Profile Workshop
- editing a user profile
- saving, compiling, and registering a user profile

Opening the User Profile Workshop

This section contains procedures for creating a new user profile and opening an existing user profile from the Repository Workshop (illustrated in the following figure).

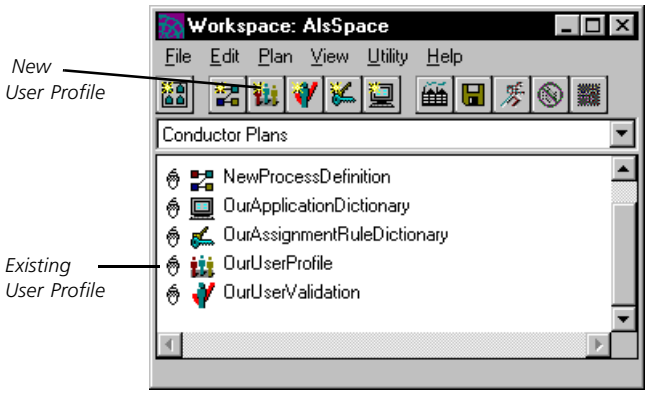
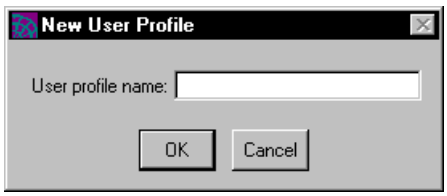


Figure 17 *Opening a User Profile in the Repository Workshop*

To open the User Profile Workshop to create a new user profile:

1 In the Repository Workshop, click the New User Profile toolbar button, or choose **Plan > New Fusion Plans > User Profile**. (See Figure 17, above.)

A dialog opens prompting you to name the user profile.



2 Name the user profile, and click **OK**.

A new user profile plan opens in the User Profile Workshop.

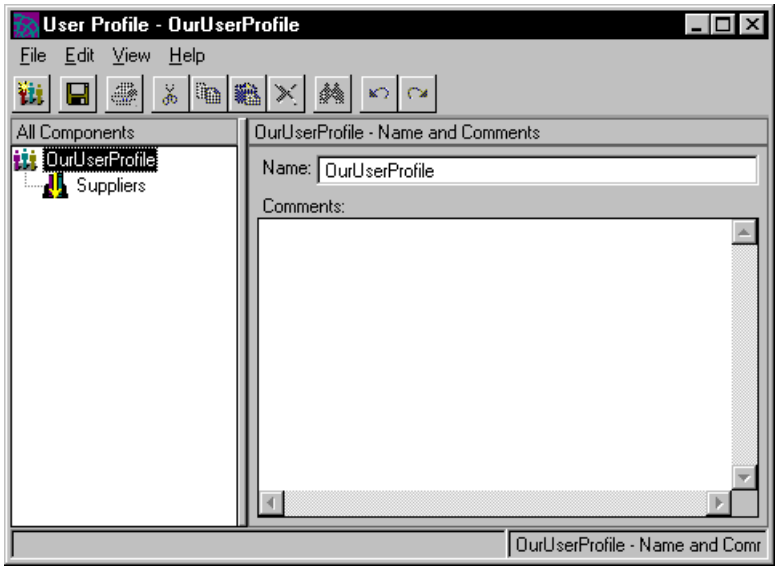
To open the User Profile Workshop for an existing user profile:

1 Double-click the name of an existing user profile in the plan list, or select the name of an existing user profile in the plan list and press **Enter**, or select the name of an existing user profile in the plan list and choose **Plan > Open**. (See Figure 17, above.)

See Chapter 3, "Managing Fusion Plans: the Repository Workshop" for more information on the Repository Workshop.

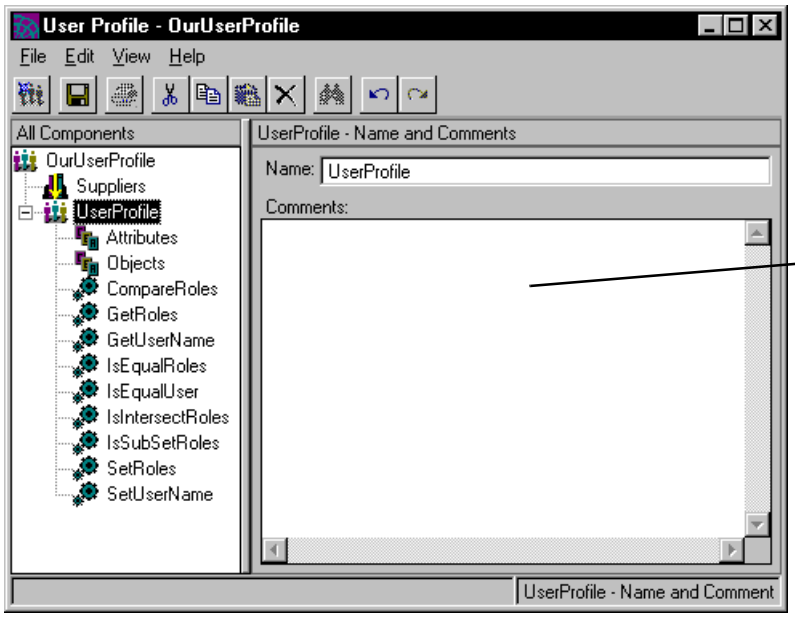
Creating and Editing a User Profile

When creating a new user profile, the User Profile Workshop opens with a new user profile plan, as shown in the following figure.



If you are creating a standard user profile, you do not have to define anything new for the UserProfile to be functional. By default, it can store and retrieve an array of roles as TextData, and all the methods have default functionality, as described in "UserProfile Class" on page 81. For a standard user profile, you can simply choose **File > Distribute**, as described in "Making and Registering User Profile Library Distributions" on page 78, to compile and register the plan.

If you are creating an extended user profile, or modifying a default method, then you have to do some work in the workshop. First, you create a user profile class by clicking the New User Profile button at the top left of the toolbar or choose **File > New User Profile**. The list view changes to display the new elements:



The list view now displays a new user profile class and the elements of that class: scalar attributes, object attributes, and a number of user profile methods. Depending on the element you select in the list view, the dialog area changes.

As you edit the user profile, be sure to save your changes periodically, as described in "Saving Changes" on page 78.

You can also add new user profile attributes (other than roles) to the class, and you can alter any of the methods using the appropriate dialog. If you want to upgrade a user profile that is currently registered with an engine and in production, copy the current user profile elements to a new user profile plan of a different name, and make the appropriate changes.

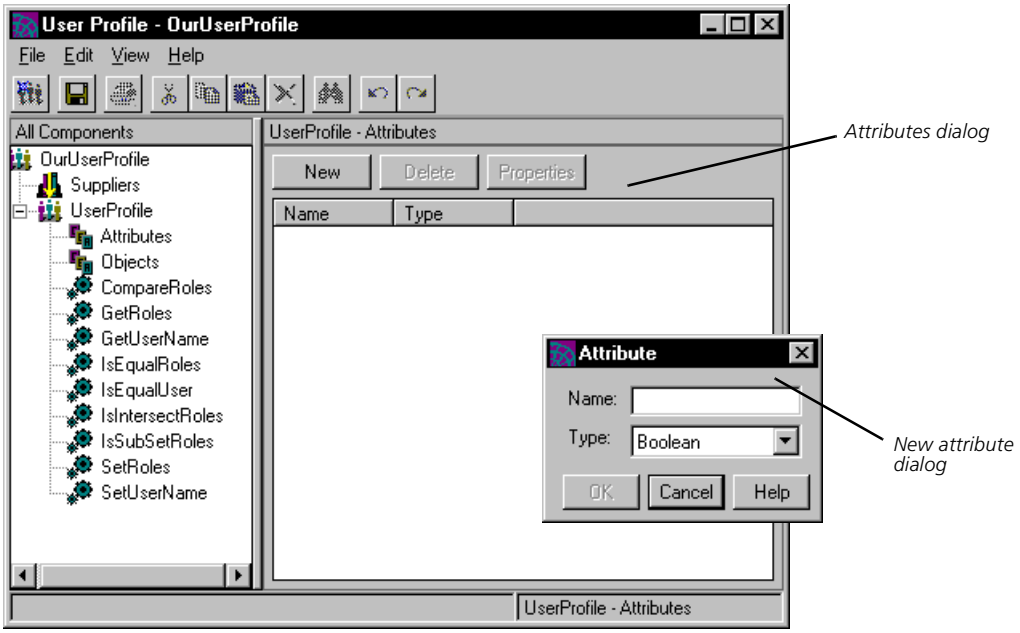
Specifying User Profile Properties

To specify user profile properties, select the user profile class element. The corresponding dialog is displayed on the right. It enables you to enter some comments about the user profile and to change the name of the user profile class.

In general, you do not need to change the user profile class name. However, if you are creating more than one user profile, or creating a new user profile with the intent of performing a rolling upgrade (as described in "Multiple User Profiles: Rolling Upgrades" on page 73), then each user profile class name should be unique. This is because you need to distinguish between multiple user profile classes in your assignment rules and validations, as described in "Creating New Versions of a User Profile" on page 80.

Specifying User Profile Attributes

To specify user profile attributes, select the Attributes element in the list view. The corresponding dialog is displayed on the right. It allows you to add user profile attributes other than roles to the class. Clicking the New button opens a dialog that prompts you for the name and type of the new user profile attribute.



You can create user profile attributes with simple types: boolean, double, float, integer, long, and string. The following table describes these Forte TOOL data types (also described in "An Introduction to The TOOL Language" on page 199):

Data Type	Description
boolean	A variable that can take one of two logical values, TRUE or FALSE.
double	Approximately 10E-308 to 10+308 with about 15 digits of precision, depending on your platform.
float	Approximately 10E-38 to 10+38 with about 7 digits of precision, depending on your platform.
integer	A signed, 4-byte integer ranging from -2,147,483,648 to 2,147,483,647 on all platforms.
long	At least -2,147,483,648 to 2,147,483,647--perhaps greater depending on your platform.
string	A simple data type that stores a string constant. There are no string expressions, and although you can compare strings in boolean expressions, there is no way to manipulate the string other than to copy it to a TextData object, manipulate it, and copy it back.

You can access user profile attributes through simple TOOL *object.attribute* syntax. For example, use the following expression to set a string attribute called ManagerName to the string constant 'Aix':

```
UserProfile.ManagerName = 'Aix';
```

Specifying User Profile Object Attributes

If you need to reference a service object in any of your user profile methods, you can define an object attribute which references the service object and serves as a handle to it. To define such an attribute, click the New button. A dialog opens, prompting you for the name and class type of the new object attribute. The class type should be the same as the service object you are referencing—its definition must be included as a supplier library to your user profile plan.

For information on accessing service objects from process definition methods, see "Writing Code that Accesses Forte Service Objects" on page 192. For more information on object attributes, see "Saving a Handle to a Service Object" on page 197.

Overriding Default User Profile Methods

You can redefine a process definition method in the list view by selecting it and entering TOOL code in the method edit field in the dialog area on the right. The default method implementations are not shown. The signature of each method is listed at the top of the method edit field. See "UserProfile Class" on page 81 for a description of the methods.

Saving and Compiling User Profiles

As you work on your user profile, it is a good idea to save your work regularly. As you make changes to the UserProfile class, you can periodically compile your user profile plan into a TOOL project to ensure that the syntax is correct.

Saving Changes

Save All command

As you edit the user profile, be sure to save your changes periodically (choose **File > Save All**). When you save changes, the current user profile plan in your workspace is updated.

Note

If you have any other workshops open for editing, they are saved at the same time.

Compiling a User Profile

Compile command

As you modify a user profile by adding user profile attributes and possibly modifying default UserProfile methods, you might want to compile to ensure that your code is syntactically correct. To compile, choose **File > Compile**. Fusion generates TOOL code from the user profile plan and compiles it, saving the result in a read-only TOOL project that has the extension _UP. (This file is a by-product of the compile process; you do not use it.)

If there are compilation errors, Fusion displays them for you. You can then go back to the workshop, fix the errors in your method code, and recompile.

Hint

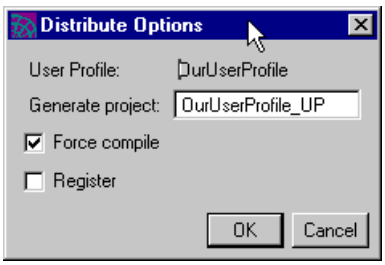
If too many of these generated files begin to clutter your list of plans in the Repository Workshop, you can filter them out by setting the Filter drop list to Fusion Plans. (Figure 17 on page 74 shows the Repository Workshop with the filter set this way.)

Making and Registering User Profile Library Distributions

Finally, when you have completed all work on a user profile and are ready for it to be used by an engine, you make it into a library distribution and register it with one or more engines.

Distribute command

To perform these operations, choose **File > Distribute**. You see the Distribute Options dialog box:



Force Compile option

The **File > Distribute** command performs a compile operation if this option is enabled, then uses the resulting TOOL project to make a library distribution. The Generate Project field shows you the name of the generated TOOL project. You can enter another name if you like.

Register option

To register the resulting library distribution with an engine, enable the Register option. If the Register option is enabled, you are prompted with a list of engines. Choose the engines you want to register with, then click **OK**. The library distribution is saved in the FORTE_ROOT/appdist directory on the central server node in your Fusion system.

Note

The node hosting a Fusion process engine must be online and the engine running in your environment before you can perform a registration with that engine.

If the engine you want to register with is not available in your environment, copy the generated library from your FORTE_ROOT/appdist directory to the remote environment. Then use the Fusion Console to register the distribution. Refer to the *Forte Fusion Process Management System Guide* for more information.

Including a User Profile as a Supplier Library

Once you have made a user profile library distribution and registered it with an engine, you must make the user profile available as a supplier library to the assignment rules and validations.

To make the user profile a supplier library:

1. Generate a user profile library distribution.

   In the User Profile Workshop, generate a user profile library distribution and register it with your Fusion engine (see "Making and Registering User Profile Library Distributions" on page 78, for more information).

   The user profile library distribution contains a .pex file, which can be found in the following location on the node hosting the engine you used to register the user profile:

   ```
   FORTE_ROOT/userapp/distribution_ID/clN/UserProfileName.pex
   ```

   N is the compatibility level number of your user profile distribution.

2. Import the user profile library back into your development repository.

   In the Repository Workshop, import (**Plan > Import**) the .pex file contained in the user profile library distribution into the development repository). See Chapter 3, "Managing Fusion Plans: the Repository Workshop" for more information.

   The .pex file is imported into your repository and saved as a library, *UserProfileName*_UP, overwriting the corresponding TOOL project.

Caution

If you integrate your workspace, be sure to do it after compiling your user profile plan, but before importing the library back into your repository. Otherwise you could lose access to your TOOL source code.

3. In the Validation Workshop, include the user profile library as a supplier library to your validation model (see Chapter 8, "Defining Validations."").

4. In the Assignment Rule Workshop, include the user profile library as a supplier library to your assignment rule dictionary (see Chapter 5, "Defining Assignment Rule Dictionaries").

Creating New Versions of a User Profile

There are two general reasons to modify a user profile:

- alter the default method implementations
- change the user profile attributes

For altering default method implementations, no special considerations or procedures are needed—Fusion already provides the necessary support. However, changing user profile attributes could affect assignment rules and validations. This change can be far reaching and should only be undertaken when absolutely necessary.

The most likely reason to modify the user profile is to extend the user profile to include user characteristics needed by one or more assignment rules. (Of course, the information needed to provide the extended user characteristics must be stored in the organization database.) However, there could also be changes to the organizational structure that require changing the user profile.

If you extend or make changes to an already extended user profile, such as adding a new user profile attribute, you should use the following general procedure.

To modify a user profile:

1. Create a new extended user profile with a different name (and different User Profile class name) and with the required changes in the User Profile Workshop.
2. Register the new user profile with the engine.
3. Make the modified user profile a supplier library to assignment rules and validation.
4. Modify all assignment rules based on the old user profile to accommodate the new user profile, and register the modified assignment rule dictionaries with the engine. (Normally the assignment rules are modified to be able to use both the new and old user profile—for more information see "Creating New Versions of an Assignment Rule Dictionary" on page 101.)
5. Modify the validation to accommodate the new user profile and register it with the engine. (Normally the validation is modified to be able to use both the new and old user profile—for more information see "Creating New Versions of a Validation" on page 177.)
6. For each client application, modify the login code to invoke the new user profile and deploy the modified client applications.

Note If you can perform a *monolithic* upgrade, that is, close all sessions with the engine and unregister your assignment rule dictionaries and current user profile—or if you can shut down your engine to perform the changeover from old to new—then you do not need to rename the new user profile, nor accommodate the old user profile (as well as the new one) in either assignment rules or the validation. You just register—or restart your engine and register—the new user profile, new validation, new assignment rule dictionaries, and all needed process definitions.

UserProfile Class

Method Summary

Method	Parameters	Returns	Source Class	Purpose
CompareRoles	**objectRoles**=*Array of TextData*	integer	●	Tests the relationship between a role or list of roles associated with the userprofile object and the role or list of roles in the objectRoles parameter. The return value is used to indicate whether the user profile's roles are subordinate to, superior to, or have some other relationship to the objectRoles array of roles.
GetOtherInfo*	none	string	●	Gets the value of any auxiliary (otherInfo) information stored for this user profile object, such as the name of the user's manager.
GetRoles	none	Array of TextData	●	Returns the roles associated with the user profile object.
GetUserName	none	string	●	Returns the name of the user associated with the user profile object.
GetSessionType*	none	integer	●	Returns a value (ADMIN or STANDARD) indicating that the user profile is to be validated, or has been validated, for an administrative or standard (non-administrative) session.
IsEqualRoles	**objectRoles**=*Array of TextData*	boolean	●	Tests to see if a role or list of roles associated with the user profile object is equal to the role or list of roles in the objectRoles parameter.
IsEqualUser	**otherUser**=*UserProfileIFace*	boolean	●	Tests to see if the user profile of one user is equal to the user profile object.
IsIntersectRoles	**objectRoles**=*Array of TextData*	boolean	●	Tests to see if a role or list of roles associated with the user profile object is equal to at least one of the roles in the objectRoles parameter.
IsSubsetRoles	**objectRoles**=*Array of TextData*	boolean	●	Tests to see if a role or list of roles associated with the user profile object are a subset of the roles in the objectRoles parameter.
SetRoles	**newRoles**=*Array of TextData*	none	●	Sets the roles associated with the user profile. object.
SetOtherInfo*	**otherInfo**=*string*	none	●	Sets the value of auxiliary (otherInfo) information stored for this user profile object, such as the name of the user's manager, to the value of the parameter.
SetUserName	**newUserName**=*string*	none	●	Sets the user name of the user profile object to the name of the parameter.

* These methods are not displayed in the user profile list view, because you cannot modify or override them. You must use the default implementation.

Using UserProfile

User profile objects, whether standard or extended, are used by the engine to identify the user associated with the current session and to determine access to activities and processes.

The engine passes a user profile object through to your site-defined ValidateUser method, which uses the object to authenticate a user's login and populate the user profile with information from the site's organization database. For more information, see Chapter 8, "Defining Validations."

The engine also passes a user profile object to assignment rules, where user profile methods are used to determine which users can perform an activity. For more information, see "Understanding the Evaluate Method" on page 95.

If your UserProfile methods need to access service objects, you must follow a special technique described in "Writing Code that Accesses Forte Service Objects" on page 192.

Methods

CompareRoles

The **CompareRoles** method tests the relationship between a role or list of roles associated with the user profile object and the role or list of roles in the **objectRoles** parameter. The return value indicates whether the user profile's roles are subordinate to, superior to, or have some other relationship to the **objectRoles** array of roles.

CompareRoles (objectRoles=*Array of TextData)*

Returns integer

Parameters	Required?	Input	Output
objectRoles	●	●	

You can use the **CompareRoles** method when you want to design a set of assignment rules that do not depend on the hierarchy of roles in your organization. Instead, you can use this method to look at a hierarchy of roles and provide a meaningful relationship between sets of roles. You override the default implementation and create a **CompareRoles** method based on your organization's role hierarchy and assignment rule needs.

For example, an assignment rule's Evaluate method could test if a user's roles are superior (or subordinate) to those specified in the assignment rule's role list. If the user's roles are superior to those in the role list, then you could offer an activity to the user. If the user's roles are subordinate to those in the role list, then you probably do not want to offer the activity to the user.

When you write a **CompareRoles** method, you determine the meaning of the return values, and use them appropriately in your assignment rules. The default implementation has the following return values:

Integer value	Meaning
0	IsEqual() is TRUE
1	IsSubset() is TRUE)
-1	No relationship (that is, neither IsSubset() or ISEqual() is TRUE)

GetOtherInfo

The **GetOtherInfo** method gets the value of any auxiliary (otherInfo) information stored for this user profile object, such as the name of the user's manager. The method is normally used in assignment rule Evaluate methods, where the return value is compared with the otherInfo parameter. (For more information see "Understanding the Evaluate Method" on page 95.)

GetOtherInfo ( )

Returns string

GetRoles

The **GetRoles** method returns the roles of the user associated with the user profile object.

GetRoles ( )

Returns Array of TextData

GetUserName

The **GetUserName** method returns the name of the user associated with the user profile object.

GetUserName ( )

Returns string

GetSessionType

The **GetSessionType** method returns a value—ADMIN or STANDARD—indicating that the user profile is to be validated, or has been validated, for an administrative or standard (non-administrative) session.

GetUserName ( )

Returns integer

IsEqualRoles

The **IsEqualRoles** method tests to see if a role or list of roles associated with the user profile object is equal to the role or list of roles in the **objectRoles** parameter.

IsEqualRoles (objectRoles=*Array of TextData)*

Returns boolean

Parameters	Required?	Input	Output
objectRoles	●	●	

IsEqualUser

The **IsEqualUser** method is used by the engine to test if the user associated with a session object in the engine (say, a suspended session) is the same as a user requesting that the session be opened (or reactivated). The implementation of the method represents the algorithm for reconnecting OpenSession requests to currently suspended sessions.

IsEqualUser (**otherUser**=*UserProfileIFace*)

Returns boolean

Parameters	Required?	Input	Output
otherUser	●	●	

If you are using an extended user profile, you might want to customize this method to test for the added attributes.

IsIntersectRoles

The **IsIntersectRoles** method tests to see if at least one role in the list of roles of the user profile object is equal to at least one role in the list of roles in the **objectRoles** parameter.

IsIntersectRoles (objectroles=*Array of TextData*)

Returns boolean

Parameters	Required?	Input	Output
objectRoles	●	●	

IsSubsetRoles

The **IsSubsetRoles** method tests to see if a role or list of roles of the user profile object is equal to or a subset of the role or list of roles in the **objectRoles** parameter.

IsSubsetRoles (objectroles=*Array of TextData*)

Returns boolean

Parameters	Required?	Input	Output
objectRoles	●	●	

SetRoles

The **SetRoles** method replaces the role list for the user profile object with the roles of the parameter.

SetRoles (**newRoles**=*Array of TextData*)

Returns none

Parameters	Required?	Input	Output
newRoles	●	●	

SetOtherInfo

The **SetOtherInfo** method sets the value of auxiliary (**otherInfo**) information stored for this user profile object, such as the name of the user's manager, to the value of the parameter.

SetOtherInfo (**otherInfo**=*string*)

Returns none

Parameters	Required?	Input	Output
otherInfo	●	●	

otherInfo parameter

The **otherInfo** parameter is auxiliary information placed in the user profile by the ValidateUser method at session login time. For an example, see "Understanding the ValidateUser Method" on page 173.

SetUserName

The **SetUserName** method sets the user name of the user profile object to the name of the parameter.

SetUserName (**newUserName**=*string*)

Returns none

Parameters	Required?	Input	Output
newUserName	●	●	

Chapter 5

Defining Assignment Rule Dictionaries

This chapter discusses assignment rule dictionaries and describes how to use the Assignment Rule Workshop.

For a general description on using assignment rules as part of a Fusion process management system, refer to “Application and Process Logic” on page 35.

For a description of how to add an assignment rule to an activity, refer to “Working with Offered Activities” on page 140.

This chapter covers the following topics:

- description of assignment rules
- using the Assignment Rule Workshop
- creating new versions of an assignment rule dictionary

About Assignment Rules

An assignment rule dictionary is a container for a set of assignment rules. The Fusion process engine uses assignment rules to control the following:

- who can perform an activity (start an activity in client applications)
- who can create a new instance of a process definition

If an activity has no assignment rule associated with it, anyone who logs in with a client application can see it in a worklist and start it (offered activity) or retrieve it from a queue (queued activity). If a process has no assignment rule associated with it, anyone can start an instance of the process. However, it is likely that you want to restrict who can perform activities and who can start process instances.

In their simplest form, assignment rules support a straightforward role-based system of setting such restrictions. Typically, the roles have meaning in your organization—the engine uses a registered user profile and registered validation to retrieve the roles from your organization database. By appropriately modifying and enhancing assignment rules in tandem with the user profile, you can implement a custom work assignment system of arbitrary power and complexity. However, see the note on page 88 regarding performance issues when using complex assignment rules.

In the role-based expense report reimbursement process (introduced in Chapter 1, “Fundamentals”), there is no restriction on who can start a process instance: any employee can submit an expense report, starting a new process instance of the Expense Report process for each expense report submitted. Once submitted, an expense report goes to the employee’s manager for review. If it is not approved, it goes back to the employee for more work. If the expense report requests reimbursement of over $1,000, the request has to be submitted to the manager’s manager for approval. Once approved, the report is forwarded to an accountant and an auditor, who perform activities required before a reimbursement check can be processed.

To ensure that the right people get offered the right activities, each activity has an assignment rule associated with it. The assignment rule tests that a given user is in the correct role to perform the activity. In this example, the Review Expense activity has an assignment rule that requires that the user performing the review be a manager of the employees who submitted the expense report (see the assignment rule attached to the

Review Expense activity in Figure 18). For information on how to associate assignment rules with activities and process instances, see "Working with Offered Activities" on page 140 and "Working with Process Definitions" on page 132, respectively.

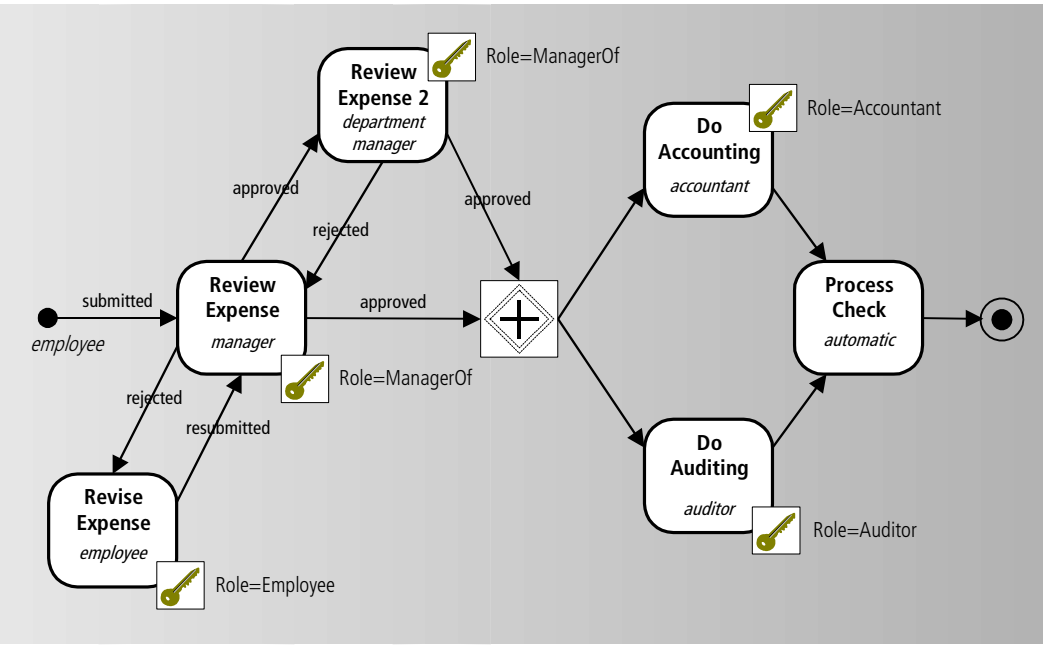


Figure 18 *Assignment Rules in an Expense Report Reimbursement Process*

It is likely that all the assignment rules for this process definition are in a single assignment rule dictionary. There is no requirement that associated rules be in the same dictionary, but keeping assignment rules in the same dictionary makes it simpler to work with them.

Adding Complexity to an Assignment Rule

By default, an assignment rule is *role-based*—it performs simple role checking, making sure that at least one role in a user's profile matches at least one role specified in the assignment rule. You only need to specify one or more roles for each assignment rule, and associate that assignment rule with an activity.

You can also define *custom assignment rules* that do more than simple role checking. To define a custom assignment rule, modify the rule's Evaluate method (using the Assignment Rule Workshop) to implement your own logic for determining assignment. Some examples of custom behavior you might implement are:

- test a list of roles in a specific order
- test the value of one or more process attributes

  For example, determine if a user has sufficient authority to approve a purchase order of a certain amount.
- compare the user's username to a *linked user* (the username of the person who completed another activity)

  For example, ensure that the person who reviews an expense report is the manager of the person who submits it.
- access a database

  For example, retrieve data that helps determine the assignment.

These are just a few of the things you can do with assignment rules. See "Understanding the Evaluate Method" on page 95 for more information on writing an Evaluate method.

Note

Applications that typically use a large number of sessions (user logons) and process instances may encounter performance issues when implementing complex assignment rules. The evaluation of complex rules by the Fusion engine cannot take advantage of the optimizations typically used for role-based assignments. Additionally, if a complex rule requires database access or calls to an external Forte object, engine performance can be slowed even further. Refer to "Assignment Rules During Process Execution" for more information about how assignment rules are used in the Fusion engine.

Assignment Rules and Activities

Offered activities

The previous section's discussion of how to extend assignment rule behavior applies only to assignment rules associated with *offered* activities, where the assignment rule determines who is permitted to perform a given activity. These assignment rules can be arbitrarily complex (although there is a performance cost associated with increased complexity).

Queued activities

Assignment rules that are associated with *queued* activities, where the assignment rule determines who is permitted to access a queue, would not have the same level of complexity as rules associated with offered activities. Users who access a queue can perform any activity in the queue. The activities in the queue belong to many process instances. You typically do not employ a queued activity (where a pool of users is available to perform the activity) when you want complex assignment rules to apply.

You can associate more than one assignment rule with an offered or a queued activity. Assignment rules, however, cannot be associated with automatic activities, subprocess activities, the First activity, or the Last activity, because none of these activities are performed by a user.

Assignment Rules During Process Execution

During development, you use the Process Definition Workshop to an associate assignment rule with a process, or with offered or queued activities. At runtime, the engine invokes the assignment rule's Evaluate method to check each user's profile against the rule. (The user profile is constructed when a user opens a session with the engine from a client application.)

Multiple versus Single Instance Assignment Rules

The Fusion engine allows for two implementations of assignment rules: single instance and multiple instance.

Single instance An assignment rule is shared among all activities and process definitions that refer to it. Single instance assignment rules have the advantage of requiring less engine memory, and any service objects accessed need only be referenced once. To provide effective sharing, the engine ensures that there is only one concurrent execution of the Evaluate method. Depending on your process design and load conditions, this may improve or degrade your engine's overall performance, as compared to a multiple instance implementation.

Multiple instance An assignment rule has a separate instance created for each activity that refers to it. Because each activity has a private instance of the rule, there can be multiple concurrent executions of the Evaluate method. However, this will consume more memory, as compared to a single instance implementation.

Process Instance Creation

The engine verifies that a user is authorized to create an instance of a process by checking the process assignment rule (or rules) against the user's profile, much the same way as it does for activities, as explained in the following sections.

Offered Activities

When a process instance is created, its First activity completes automatically. This activity typically routes to one or more activities in the process definition that require work to be done by a user. For example, in the insurance claim example described on page 87, the First activity routes directly to the initial processing activity performed by the receiving clerk. This activity is an offered activity.

When an offered activity reaches a READY state, the engine uses the activity's assignment rule or rules to evaluate the user profile of each active session. (Specifically, the engine invokes each assignment rule's Evaluate method for each session's user profile, and if it returns TRUE, there is a match.) The engine offers the activity to the sessions that match any of the assignment rules for the activity. When an activity is offered to a session, it is placed on the session's activity list.

Assignment rules are also evaluated for each new session. When a new user profile is validated (when a user opens a new session), the engine checks the user profile against every assignment rule of every offered activity that is in a READY state. If there is a match, the engine adds the activity to the new session's activity list.

The associated client application can either retrieve the session activity list to build a worklist and display it to the user or respond to activity list update events sent by the engine. Typically, both approaches are used.

Queued Activities

When a queued activity reaches a READY state, the engine adds it to a queue of READY activities of the same activity name but from many instances of the same process definition. The queued activity's assignment rule or rules are associated with the queue, not with an individual instance of the activity.

When a client application requests a queued activity, the engine applies the queue's assignment rules to the session's user profile to determine if the user is permitted to access the queue. Because the queue contains activities from many process instances, an assignment rule would not test the value of a process attribute or use the name of the person who performed a prior activity in a process instance. Therefore, assignment rules associated with queued activities do not check process attribute values or linked user information.

Performance Issues with Assignment Rules

When a system demands high-powered work assignment, you should provide the appropriate CPU, network, and database system resources. There might be many instances of many process definitions active in the engine at any one time. In a large system, many assignment rules might be simultaneously evaluated against hundreds of user profiles for thousands of activity instances. Complex assignment rules have additional performance impact, as discussed on page 88.

Thus, the performance of the Evaluate method is a prime consideration in a large system. References to local user profile and assignment rule objects should be favored over references to external services and database queries.

Working with Assignment Rules

This section describes the series of tasks you are likely to perform when you create or update an assignment rule dictionary. It covers the following topics:

- opening the workshop
- creating and editing an assignment rule
- specifying roles
- specifying a list of attributes
- writing Evaluate methods
- saving, compiling, and registering an assignment rule dictionary

Opening the Workshop

This section contains procedures for creating a new assignment rule and opening an existing assignment rule from the Repository Workshop (illustrated in the following figure).

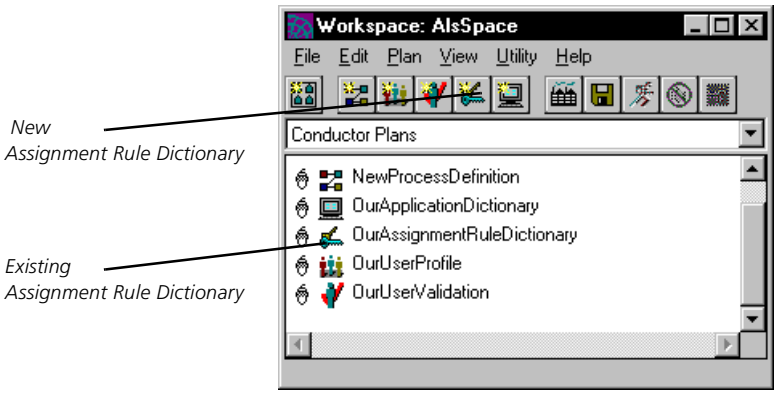


Figure 19 *Opening an Assignment Rule Dictionary in the Repository Workshop*

To open the Assignment Rule Workshop to create a new plan:

1 From the Repository Workshop, click the New Assignment Rule toolbar button, or choose **Plan > New Fusion Plans > Assignment Rule**.

A dialog displays prompting you to name the assignment rule.

2 Name the assignment rule, and click **OK**.

A new assignment rule plan opens.

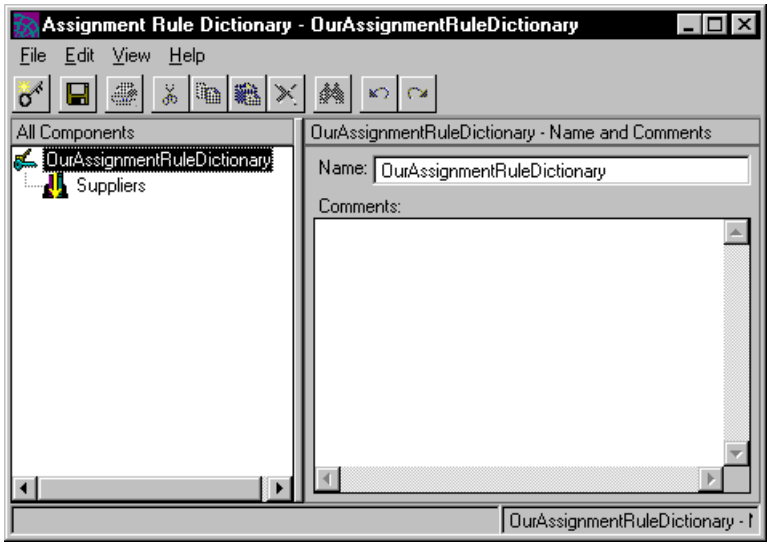
To open the Assignment Rule Workshop for an existing plan:

1 From the Repository Workshop, double-click the name of an existing assignment rule in the plan list, or select the name of an existing assignment rule in the plan list and press Enter, or select the name of an existing assignment rule in the plan list and choose **Plan > Open**.

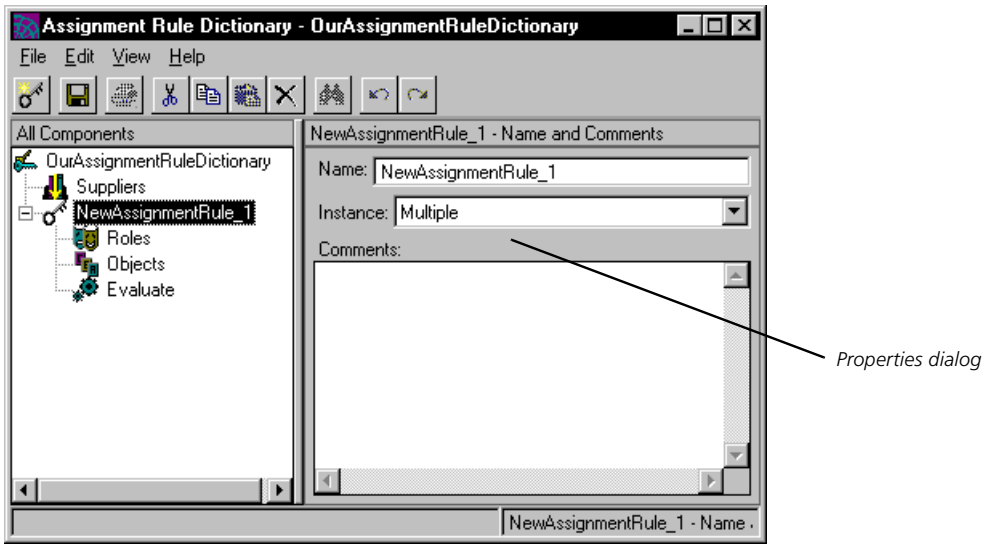
See Chapter 3, "Managing Fusion Plans: the Repository Workshop" for more information on the Repository Workshop.

Creating and Editing an Assignment Rule

If you are creating a new assignment rule dictionary, the Assignment Rule Workshop opens with a new assignment rule dictionary plan, as shown in the following figure:



To create a new assignment rule in the dictionary, click the New Assignment Rule button at the top left of the toolbar or choose **File > New Assignment Rule**. The list view changes to display the new elements.



Note

If your assignment rules depend upon an extended (rather than standard) user profile, you must include the user profile as a supplier library to your assignment rule dictionary. This requires that the user profile library first be imported into your process development library as described in "Including a User Profile as a Supplier Library" on page 79. To include the user profile as a supplier, click on the Suppliers element in the list view, click the Edit Supplier List button (or choose **File > Supplier Plans**), select the library, and click OK.

As you edit the assignment rule dictionary, be sure to save your changes periodically, as described in "Saving Changes" on page 99.

Specifying Assignment Rule Properties

To specify assignment rule properties, select the assignment rule element in the list view. The corresponding dialog is displayed on the right. It enables you to enter an assignment rule name, specify the number of runtime instances, and write comments about the assignment rule.

Name Specifies a name for the assignment rule.

Instances Indicates whether you want single or multiple instances of the assignment rule at execution time. The default is multiple. For more information, see "Multiple versus Single Instance Assignment Rules" on page 88.

Comments Allows you enter internal comments about the assignment rule, which may be useful to other developers.

Specifying Roles

The roles your users can assume are defined in your organization database, which must be set up or already exist at your site. Your application system designer defines a ValidateUser method in the Validation Workshop that retrieves and verifies roles from this database for a user. (For more information on validations, see Chapter 8, "Defining Validations.")

The default behavior of an assignment rule's Evaluate method is to compare the role or list of roles you specify for an assignment rule in this workshop to the role or list of roles in a user's profile. If there is at least one match between the assignment rule's roles and the user's profile roles, the result is TRUE, and the activity is placed on the corresponding session's activity list. If you only specify roles and do not write your own Evaluate method, you get this default Evaluate behavior.

Role names used in this default behavior are not case sensitive. For example, the following role names are equivalent: AreaManager, AREAMANAGER, areamanager. Spaces are not allowed.

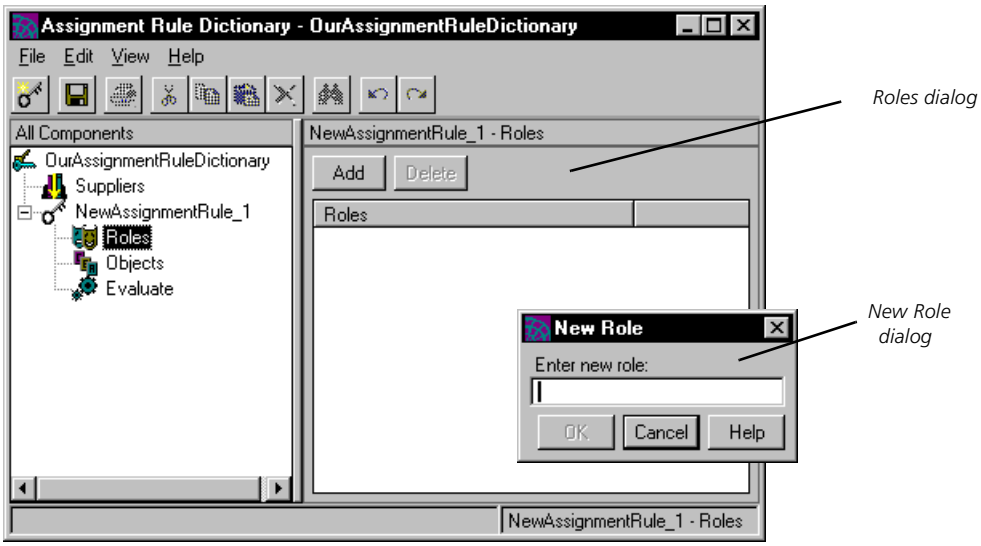
You can specify a single role or a list of roles. What you specify depends on how you expect this assignment rule to be applied. There are a number of ways you can specify that the engine check a user profile against a list of roles:

- Create multiple assignment rules in this dictionary, each specifying a single role. Then in the Process Definition Workshop, the process developer picks from this list of assignment rules, associating several with a given activity.
- Specify multiple roles for each assignment rule in the dictionary, tailoring the rule to be used for a given activity or set of activities. The process developer associates the appropriate assignment rule with a given activity. This approach provides a clearer display in the Process Definition Workshop and results in better engine performance.

To specify roles for an assignment rule:

1 Choose the Roles element of the assignment rule in the list view.

The Roles dialog is displayed on the right:



2 Click **Add** add a new role.

3 In the New Role dialog, enter the name of a role.

Be sure to enter the name correctly—it must match the name in your organization database.

4 Click **OK**.

5 Repeat 2 through 4 for each role you want to add.

To delete roles from an assignment rule:

1 Choose the Roles element of the assignment rule in the list view.

2 In the Role list dialog, if there is more than one role in the list, highlight the role you want to delete.

3 Click **Delete** (or choose **Edit** > **Delete**).

Specifying Object Attributes

If you need to reference a service object in an Evaluate method, you can define an object attribute which references the service object and serves as a handle to it. To define such an attribute, click the New button. A dialog opens, prompting you for the name and class type of the new object attribute. The class type should be the same as the service object you are referencing, and it's definition must be included as a supplier library to your assignment rule dictionary plan.

For information on accessing service objects from process definition methods, see "Writing Code that Accesses Forte Service Objects" on page 192. For more information on object attributes, see "Saving a Handle to a Service Object" on page 197.

Defining an Evaluate Method

As mentioned in "Specifying Roles" on page 92, an Evaluate method has the default behavior of finding the intersection of a user profile's list of roles and an assignment rule's list of roles. If there is at least one role in the user profile that corresponds to a role in the assignment rule, the method returns TRUE, indicating that the engine can place the activity on the corresponding session activity list. If there is no match, the method returns FALSE. (See "Understanding the Evaluate Method" on page 95 for a fuller description of the method.)

If you want an assignment rule to behave differently, you must write a custom Evaluate method. As with all methods in the process definition workshops, you write this method in Forte's TOOL language. (For a description of TOOL, see the "An Introduction to The TOOL Language" on page 199.)

Using the Method Definition Dialog

To use the Evaluate Method Definition Dialog:

1 Choose the Evaluate method element in the list view.

The Method definition dialog is displayed:

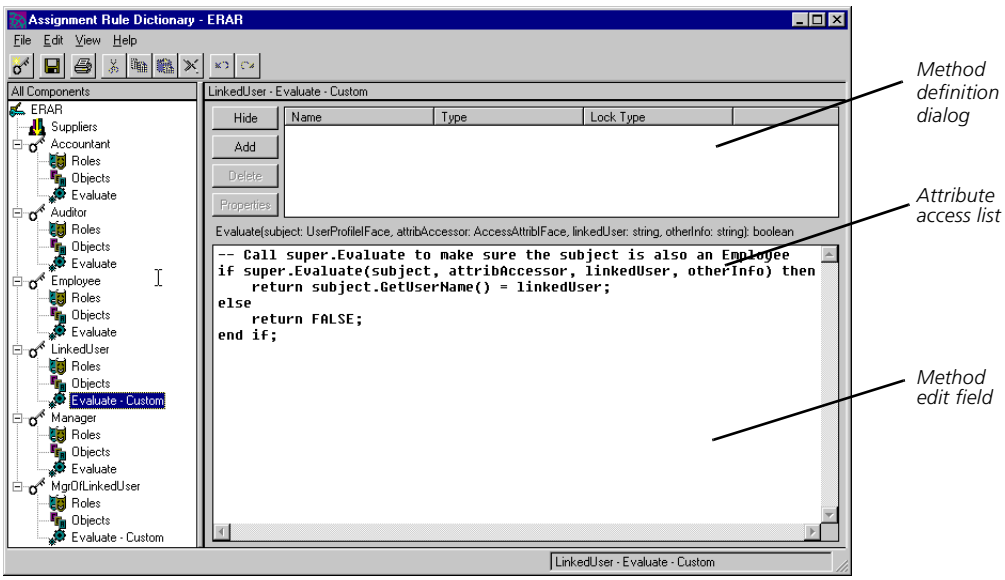


Figure 20 *Evaluate Method Definition Dialog*

2 If you want your Evaluate method to access process attributes, click the **Attributes** button to display the attribute access list, then see the next section, "Specifying Process Attributes."

3 Enter your TOOL code in the method edit field below the Evaluate method declaration.

Specifying Process Attributes

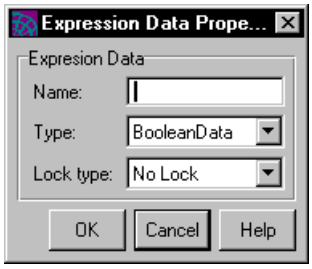
You can specify one or more process attributes to use in your Evaluate method. The process attributes you specify here must be defined in each process definition which uses this assignment rule. (For more information on defining process attributes, see "Defining Process Attributes" on page 138.)

For example, an alternative to the routing used in the expense report reimbursement process, illustrated in Figure 18 on page 87, would be to employ process attributes to determine who reviews an expense report. Suppose, in the Review Expense activity, that if the expense amount is under $1,000, the user performing this activity can be a Manager,

but if it is over $1,000, the user must be a Director. The process definition stores the expense report amount in the TotalAmount process attribute. To determine the amount claimed in the expense report, the Evaluate method must be able to read the TotalAmount process attribute and test for the appropriate roles.

If you want your Evaluate method to access one or more process attributes, click the Attributes button to display the attribute access list.

To add a process attribute, click the Add button. A dialog opens, prompting you for the name of the attribute, its type, and the type of lock request when the Evaluate method accesses the attribute.



Name Enter the name of a process attribute that is defined in every process definition that uses this assignment rule. Process attribute names are case sensitive; the attribute must exactly match the definitions of the process attribute in the process definitions.

Type Choose a data type that matches that of the defined process attribute. You can choose from the drop list. The data types for process attributes are described in "Process Attribute Data Types" on page 189.

Lock type Choose the type of lock request that is sent to the engine when the Evaluate method is executed. You can choose from the drop list. The lock types for process attributes are described in "Specifying Lock Types" on page 187.

Note If this assignment rule is applied to a queued activity, any process attributes you specify are ignored by the engine because the assignment rule is attached to the queue and cannot read process attribute values for an individual process instance.

Understanding the Evaluate Method

In the **Evaluate** method definition dialog shown in Figure 20 on page 94, you can see the declaration for the **Evaluate** method. The method syntax is:

Evaluate (**subject**=*UserProfileIFace*, **attribAccessor**=*AccessAttribIFace*, **linkedUser**=*string*, **otherInfo**=*string*)

Returns *boolean*

Parameters	Required?	Input	Output
subject	●	●	
attribAccessor	●	●	
linkedUser	●	●	
otherInfo	●	●	

This method is executed by the engine, which passes in all the parameters when it invokes the method. For more information on how the engine uses this method, refer to "Assignment Rules During Process Execution" on page 88.

The method returns TRUE or FALSE, indicating whether the user profile matched the criteria being checked by the method.

subject parameter

The **subject** parameter is the user profile object of the session being evaluated. The UserProfileIFace type represents the user profile defined in the User Profile Workshop and registered with the Fusion engine.

The following user profile methods are some of the more useful ones to use in writing an **Evaluate** method. These methods are described in more detail in "UserProfile Class" on page 81.

Method	Parameters	Returns	Purpose
GetRoles	none	Array of TextData	Returns the roles associated with the user profile.
GetUserName	none	string	Returns the name of the user associated with the user profile.
GetOtherInfo	none	string	Gets the value of any auxiliary (otherInfo) information stored for this user profile object, such as the name of the user's manager.
CompareRoles	**objectRoles**=*Array of TextData*	integer	Tests the relationship between a role or list of roles associated with the current user and the role or list of roles in the objectRoles array of roles. The return value is used to indicate whether the current user's roles are subordinate to, superior to, or have some other relationship to the objectRoles array of roles.
IsEqualRoles	**objectRoles**= *Array of TextData*	boolean	Tests to see if a role or list of roles associated with the current user is equal to the role or list of roles in the objectRoles array of roles.
IsIntersectRoles	**objectRoles**= *Array of TextData*	boolean	Tests to see if at least one of the roles of the current user is equal to at least one of the roles in the objectRoles array of roles.
IsSubsetRoles	**objectRoles**= *Array of TextData*	boolean	Tests to see if all the roles of the current user are in the objectRoles array of roles.

attribAccessor parameter

The **attribAccessor** parameter is an attribute accessor for the **Evaluate** method's attribute access list specified in the method edit dialog, described under "Specifying Process Attributes" on page 94. For information on how to use this parameter see "Working with Process Attributes" on page 188.

linkedUser parameter

The **linkedUser** parameter is the user name of the user who completed another activity that has been linked to this one. Activities are linked in the Process Definition Workshop, and described in "Activity Links" on page 120. It is the **Evaluate** method that gives meaning to the link: the process developer must not only link the two activities, but must also associate an assignment rule with the activity containing the link. (See "Evaluate Method Example: Linked Activity (linkedUser)" on page 98.)

otherInfo parameter

The **otherInfo** parameter is information that is passed, along with the **linkedUser** parameter, to the **Evaluate** method from a linked activity. The information passed in the **otherInfo** parameter is provided by the user profile of the user who performed the linked activity, in the same way as is the user name. For this information to be available, however, it must be placed in the user profile (see "SetOtherInfo" on page 84), by the ValidateUser method executed when the user logs in to the engine to open a session. The interpretation of otherInfo is determined by the system design—the interaction of the **Evaluate** method, the user profile, and the process definition—hence, the rather vague name. (See "Evaluate Method Example: Linked Activity (otherInfo)" on page 98.)

Using the Evaluate Method

Invoking the default

When you write an Evaluate method you override the default Evaluate implementation. However, if you want to write logic that adds to the default implementation, rather than replace it, you can invoke the default implementation first. For example you can write the following code:

```
if super.Evaluate(subject, attribAccessor, linkedUser, otherInfo)
      then
      . . . ; --put additional code in here.
end if;
```

In this case the method first checks the subject against the roles specified for the assignment rule (the default implementation), and only continue if the default returns TRUE.

Accessing the role list

In writing your Evaluate method, you may need to access the assignment rule's list of roles, described in "Specifying Roles" on page 92. The role list is a TextData array attribute named "roles" that can be referenced in code either as "self.roles" or by using the internal GetRoles method, which returns an array of "roles." For example:

```
return user.IsEqualRoles(self.GetRoles());
```

Extended user profile

If you have an extended user profile, and you want your Evaluate method code to access extended user profile attributes, you have to cast the **subject** parameter to your extended user profile type. For example, if your extended user profile class type is ExtendedUserProfile, you would write code similar to the following:

```
MySubject : ExtendedUserProfile;
MySubject = (ExtendedUserProfile)(subject);
return (MySubject.attribute = ImportantAssignmentRuleCriteria);
```

Evaluate Method Example: Checking Process Attributes

The section "Specifying Process Attributes" on page 94 describes an employee expense report process in which there is a Review Expense activity for approving an expense. Its Evaluate method checks the total amount of the expense. If it is under $1,000, the user performing this activity must be a Manager. If it is over $1,000, the user must be a Director.

The process definition stores the claim amount in the TotalAmount process attribute. To determine the amount claimed in the expense report, the Evaluate method must be able to read the TotalAmount process attribute and check for the appropriate roles.

You can enter the following code in the method edit dialog to implement the Evaluate method:

```
checkRoles : Array of TextData = new();
if TotalAmount < 1000 then
  checkRoles.AppendRow(TextData(Value = 'Manager'));
else
  checkRoles.AppendRow(TextData(Value = 'Director'));
end if;
return subject.IsIntersectRoles(checkRoles);
```

This example accesses the TotalAmount process attribute directly using a virtual attribute. For more information on using process attributes in process definition methods, see "Working with Process Attributes" on page 188.

Evaluate Method Example: Linked Activity (linkedUser)

The section "About Assignment Rules" on page 86 discusses an example in which a user in the role of Employee started the Expense Report process by submitting an expense report to his or her manager. If the manager rejects the expense report, then the Revise Expense activity should be performed by the employee who originally submitted the expense report.

To implement this scenario, link the Revise Expense activity to the user who started the process. By definition, the user who completed the *First* activity is the user who started this process instance. Therefore, the process developer links the Revise Expense activity to the First activity (see "Activity Links" on page 120 and "Setting an Activity Link" on page 142).

For the link to have the intended meaning, the process developer must also attach to Revise Expense an assignment rule whose Evaluate method checks to see if the user who completed First (passed in as the linkedUser parameter), has the same user name as the user profile object associated with the session currently being evaluated. If so, the user of that session gets the Revise Expense activity.

The code you enter in the method definition dialog is:

```
return subject.GetUserName() = linkedUser;
```

Notice that this code is generic. You can use this assignment rule for any activity that needs to link in the same way.

Evaluate Method Example: Linked Activity (otherInfo)

A case that is a bit more complicated than the preceding example is when the user who performs an activity is not the same user who performed a linked activity, but, say, the manager of that user.

It would be possible for an Evaluate method to perform a lookup in the organization database to find the manager of linkedUser. However, this lookup would have to be performed and compared to the user profile of each session being evaluated. A more efficient approach is to perform that lookup only once, when authenticating each user, and adding the manager's name to the user profile using a user profile method called SetOtherInfo.

When you link one activity to another, both the linkedUser and otherInfo parameters are passed to the Evaluate method of the successor activity.

In the example discussed in "About Assignment Rules" on page 86, if you want the Review Expense activity to be performed by the *manager* of the person who completed the First activity, you first set otherInfo to the user's manager in the ValidateUser method (see "Writing a ValidateUser Method" on page 173), and then enter the following code in the method definition dialog:

```
if super.Evaluate(subject, attribAccessor, linkedUser, otherInfo)
    then return subject.GetOtherInfo() = otherInfo;
else
    return FALSE;
end if;
```

See Expense Reporting example

Plan: ERAR • **Class**: MgrOfLinkedUser • **Method**: Evaluate

You can use this type of assignment rule for any condition you represent by the otherInfo parameter.

Saving and Compiling an Assignment Rule Dictionary

As you work on assignment rules, it is a good idea to save the dictionary regularly. If you write your own Evaluate method, you can compile it to see if the code is correct.

Saving Changes

Save All command

As you edit an assignment rule, be sure to save your changes periodically (choose **File > Save All**). When you save changes, the current assignment rule dictionary is updated in your workspace.

Note

If you have any other workshops open for editing, they are saved at the same time.

Compiling an Assignment Rule Dictionary

Compile command

If you write custom Evaluate methods, you may want to compile the dictionary each time you finish a method to ensure that your code is correct. To compile, choose **File > Compile**. Fusion generates TOOL code from the assignment rule dictionary and compiles it, saving the resulting TOOL code in a read-only file that has the extension _AR. (This file is a by-product of the compile process: you do not use it.)

If there are compilation errors, Fusion displays them for you. You can then return to the workshop, fix the errors in your TOOL code, and recompile.

Hint

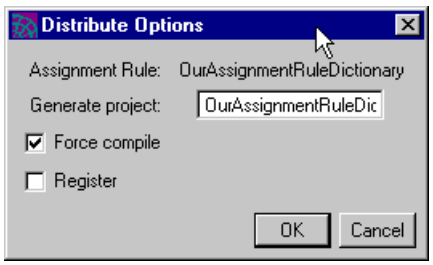
If too many of these generated files clutter your list of plans in the Repository Workshop, you can filter them by setting the Filter drop list to Fusion Plans. (Figure 19 on page 90 shows the Repository Workshop with the filter set this way.)

Making and Registering an Assignment Rule Dictionary

Distribute command

When you have completed all the assignment rules for a dictionary and are ready to use the rules in an engine, you make the dictionary into a library distribution and register it with one or more engines.

To perform these operations, choose **File > Distribute** to open the Distribute Options dialog:



Force Compile option

The **File > Distribute** command performs a compile operation if this option is enabled, then uses the resulting TOOL project to make a library distribution. The Generate Project field shows you the name of the generated TOOL project. You can enter another name if you like.

Register option

To register the resulting library distribution with an engine, enable the Register option. If the Register option is enabled, you are prompted with a list of engines. Choose the engines you want to register with, then click **OK**. The library distribution is saved in the FORTE_ROOT/appdist directory on the central server node in your Fusion system.

Note

The node hosting a Fusion process engine must be online and the engine running in your environment before you can perform a registration with that engine.

If the engine you want to register with is not available in your environment, copy the generated library from your FORTE_ROOT/appdist directory to the remote environment. Then use the Fusion Console to register the distribution. Refer to the *Forte Fusion Process Management System Guide* for more information.

Creating New Versions of an Assignment Rule Dictionary

Assignment rules are the design elements you are most likely to change. You may want to add new conditions to an existing rule or rules, or add completely new assignment rules. If your changes require new process attributes or an extension to the user profile, the impact of the change can be far reaching.

For example, if your changes require new, currently undefined process attributes, then all process definitions using the new assignment rules have to be modified to include the attribute definitions and registered again with your engine. Possibly, some client applications creating instances of the process definitions have to be revised and redeployed as well.

If the modification in assignment rules requires an extension to the user profile or a change in an extended user profile (see "Extended vs. Standard User Profile" on page 72), then almost all your design elements are affected. The user profile, validation, and assignment rule dictionaries have to be modified and reregistered with your engine. In addition, the login code of client applications may need to be modified and the client applications redeployed.

If these changes can be implemented in a *monolithic* upgrade of your Fusion enterprise application, then your assignment rules need support only the current user profile. In a monolithic upgrade you shut down all sessions and unregister the old versions of user profile and assignment rule dictionaries—or shut down your engine to perform the changeover—and then reregister the new versions of Fusion distributions.

However, if you need to perform a *rolling* upgrade of your Fusion enterprise application, as described in "Multiple User Profiles: Rolling Upgrades" on page 73, it might be necessary for your assignment rules to support more than one user profile. If this is the case, then your Evaluate methods must include a "case" statement or an "if" statement that references more than one user profile.

For example, if two user profiles of class type UserProfile1 and UserProfile2 have been registered with an engine, you might have code similar to the following code fragment:.

```
profile1 : UserProfile1 = new;
profile2 : UserProfile2 = new;
if user.IsA(UserProfile1) then
  profile1 = (UserProfile1)(user); --cast to UserProfile1
  . . .  --perform evaluation on profile1
else
  profile2 = (UserProfile2)(user); --cast to UserProfile2
  . . .  --perform evaluation on profile2
end if;
```

Note All user profiles referenced should be included as supplier libraries to your assignment rule dictionary (see "Creating and Editing an Assignment Rule" on page 91).

How to Modify an Assignment Rule Dictionary

The following guidelines describe how to modify, add, or delete assignment rules under a number of scenarios:

Modifying an Existing Assignment Rule

▶ **To modify an assignment rule that does not require changes in the user profile or process attributes:**

1. Modify the rule.
2. Register the modified assignment rule dictionary with the engine.
3. Make no change in process definitions.

▶ **To modify an assignment rule that requires a change in process attributes:**

1. Create a newly named assignment rule in addition to the old one and register the modified assignment rule dictionary with the engine (so that both the old and the modified assignment rule dictionaries are registered).
2. For each process definition that uses the old assignment rule, create a newly named process definition that has the new process attribute list and activities that use the new assignment rule, and register each new process definition with the engine (so that both the old and new process definitions are registered).
3. For client applications that create instances of the old process definitions, modify the code to create instances of the new process definitions and deploy the modified client applications.

Note If you can perform a monolithic upgrade, then you do not need to rename anything in the above steps, and you do not need to modify the client applications.

▶ **To modify an assignment rule that requires an extended user profile or a change in an extended user profile (see Chapter 7, "Creating Process Definitions"):**

1. Create a newly named user profile with the required changes and register the new user profile with the engine.
2. Modify the assignment rule (and all other assignment rules) to accommodate the new user profile and register the modified assignment rule dictionary with the engine. (Normally the assignment rules are modified to be able to use both the new and old user profile.)
3. Modify the validation to accommodate the new user profile and register it with the engine. (Normally the validation is modified to be able to use both the new and old user profile.)
4. For each client application that performs activities associated with the modified assignment rule dictionary, modify the login code to invoke the new user profile and deploy the modified client applications.

Note If you can perform a monolithic upgrade, then you do not need to accommodate the old user profile (as well as the new one) in either assignment rules or validation.

Adding a New Assignment Rule

▶ **To add a new assignment rule that does not require a modification in the user profile or process attributes:**

1. Add the new assignment rule to the assignment rule dictionary.
2. Register the modified assignment rule dictionary with the engine.
3. Make no change in process definitions.

▶ **To add a new assignment rule that requires a change in process attributes:**

1. Add the new assignment rule to any assignment rule dictionary and register the modified assignment rule dictionary with the engine (so that both the old and modified assignment rule dictionaries are registered).
2. For each process definition that requires the new assignment rule, create a newly named process definition that has the new process attribute list and activities that use the new assignment rule, and register each new process definition with the engine (so that both the old and the new process definitions are registered).
3. For client applications that create instances of the old process definitions, modify the code to create instances of the new process definitions and deploy the modified client applications.

Note If you can perform a monolithic upgrade, then you do not need to rename anything in the previous steps, and you do not need to modify the client applications.

▶ **To add a new assignment rule that requires an extended user profile or a change in an extended user profile (see Chapter 7, "Creating Process Definitions"):**

1. Create a newly named user profile with the required changes, and register the new user profile with the engine.
2. Add the new assignment rule to the assignment rule dictionary, modify all other assignment rules to accommodate the new user profile, and register the modified assignment rule dictionary with the engine. (Normally the assignment rules are modified to be able to use both the new and old user profile.)
3. Modify the user validation to accommodate the new user profile and register it with the engine. (Normally the user validation is modified to be able to use both the new and old user profile.)
4. For each client application that performs activities associated with the modified assignment rule dictionary, modify the login code to invoke the new user profile and deploy the modified client applications.

Note If you can perform a monolithic upgrade, then you do not need to accommodate the old user profile (as well as the new one) in either assignment rules or user validation.

Deleting an Existing Assignment Rule:

▶ **To delete an assignment rule that *is not* used by any process definitions:**

1. Delete the assignment rule and register the modified assignment rule dictionary with the engine.
2. Make no change in process definitions.

▶ **To delete an assignment rule that *is* used by process definitions:**

1. Delete the assignment rule and register the modified assignment rule dictionary with the engine (so that both the old and modified assignment rule dictionaries are registered).
2. Modify each process definition that uses the old assignment rule (so that it uses a different assignment rule) and register the modified process definitions with the engine.

Registering a New Version of an Assignment Rule Dictionary

When you register a new version of an assignment rule dictionary with an engine that has an older version already registered, the new version is retroactively applied to all existing offered and queued activities, as described in the following sections.

Offered Activities

The new version of assignment rules take effect immediately. In addition, offered activities that became READY prior to registering the new version are re-evaluated, and offered to sessions once again, based on the new assignment rules. The old versions of any assignment rules included in the new assignment rule dictionary are automatically unregistered.

Queued Activities

Queued activities work differently from offered activities because their assignment rules apply to the queue and not to individual activities:

- An activity queue is first created because an instance of a queued activity becomes READY. That activity's assignment rules become the queue's assignment rules.

  A queue will also be created if a client application requests an activity from a queue which has not yet been created.

- If a new version of an assignment rule dictionary is registered that has one or more of an existing queue's assignment rules in it, the queue's assignment rules are updated. Sessions which accessed activities in the queue prior to registering the new version might no longer have access to the queue.

  The old versions of any assignment rules included in the new assignment rule dictionary are automatically unregistered.

There is an additional, related consideration for queued activities in the case where a new version of a process definition containing the same queued activity, but with different assignment rules, is registered. When the new queued activity becomes READY, the new assignment rules are attached to the existing queue, replacing the previous assignment rules.

Even if there are activities in the queue that used the old assignment rules, the queue is now accessed with the new set. For example, if the old queued activity had an assignment rule that made it accessible to managers, and the new activity has that rule replaced with one for vice presidents, once the queued activity from the newly registered process definition becomes READY, only vice presidents would be permitted to get activities in the queue.

Note If you are making a radical change in access to a queued activity when you modify a process definition, you may want to rename the activity in the revised process definition and change the client applications that access the activity to use the new activity name.

Chapter 6

Defining Application Dictionaries

This chapter discusses application dictionaries and describes how to use the Application Dictionary Workshop to define them.

For a general description of how application dictionary entries are used in a Fusion process management system, see "Application and Process Logic" on page 35.

For a description of how to add an application dictionary entry to an activity, see "Working with Offered Activities" on page 140.

This chapter covers the following topics:

- description of application dictionaries
- using the Application Dictionary Workshop
- creating new versions of an application dictionary

About Application Dictionaries

An *application dictionary* is a container for a set of application dictionary items. Each application dictionary item is a work definition containing information about an activity that the Fusion process engine sends to a process client application—both when the client application first gets its list of available activities (if it is a heads-up client application) and when it accepts an activity for a client user to perform.

The client application uses this work definition to display a description of the activity for the user, to automatically start up software programs for the user, and to query and update process attributes associated with the activity.

An application dictionary item has the following three components:

- *activity description*—a text description of the activity that the client application can display to the user (normally in a "to-do" worklist)
- *application code*—an arbitrary text string used by the client application, usually to start software on the client needed by the user to do the work
- *attribute accessor*—access to a list of process attributes that the client application reads information from or updates as the user works on the activity

For example, if the activity is the step in an expense report reimbursement process in which a manager has to approve or reject an expense report, the activity description might be something like "expense review," the application code might cause the client application to start a program that displays the expense report on a database form, and the list of process attributes might include read-only access to an ExpenseReportID attribute and write access to a Status attribute.

The client application programmer is the person most likely to use application dictionary items, since it is the client application that needs this information. The application system designer creates an application dictionary to specify high level descriptions of client applications that are used in a process management system. The application dictionary documents communication between an activity in a process definition and the client application used to perform the activity.

Application dictionary items are often associated with more than one activity, either in the same or across multiple process definitions. For example, an item that describes an Expense Review application might be used for an activity offered to a manager and for another activity offered to a departmental director.

When you have defined the set of items for an application dictionary, you save it, which writes a copy of it to the repository. Then, in the Process Definition Workshop, the process designer adds the application dictionary as a supplier plan to the process definition. At that point, the application dictionary items show up in the Process Definition Workshop's Supplier Components list, and the process developer can associate the items with the appropriate activities in the process definition. See "Working with Offered Activities" on page 140 for more information on using application dictionary items in process definitions.

Working with Application Dictionaries

This section describes the series of tasks you are likely to perform when you create or update an application dictionary. It covers the following topics:

- opening the workshop
- creating and editing an entry
- specifying an activity description and an application code
- specifying a list of attributes
- saving and using an application dictionary

Opening the Application Dictionary Workshop

This section contains procedures for creating a new application dictionary and opening an existing application dictionary from the Repository Workshop (illustrated in the following figure).

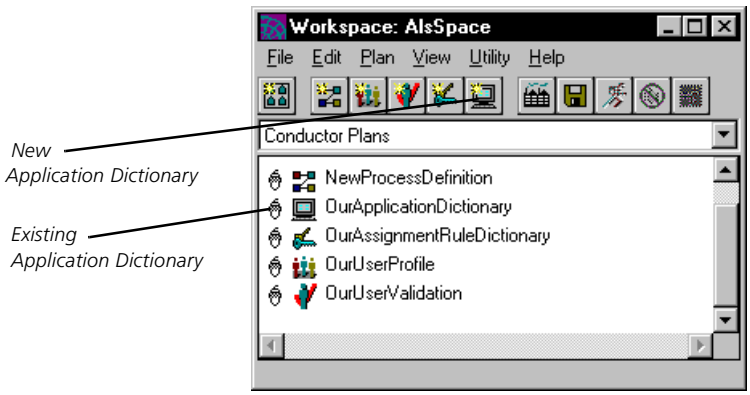


Figure 21 *Opening an Application Dictionary in the Repository Workshop*

To open the Application Dictionary Workshop to create a new plan:

1 Click the New Application Dictionary toolbar button, or choose **Plan > New Fusion Plans > Application Dictionary**.

A dialog opens prompting you to name the application dictionary.

2 Name the application dictionary, and click **OK**.

A new application dictionary plan opens in the Application Dictionary Workshop.

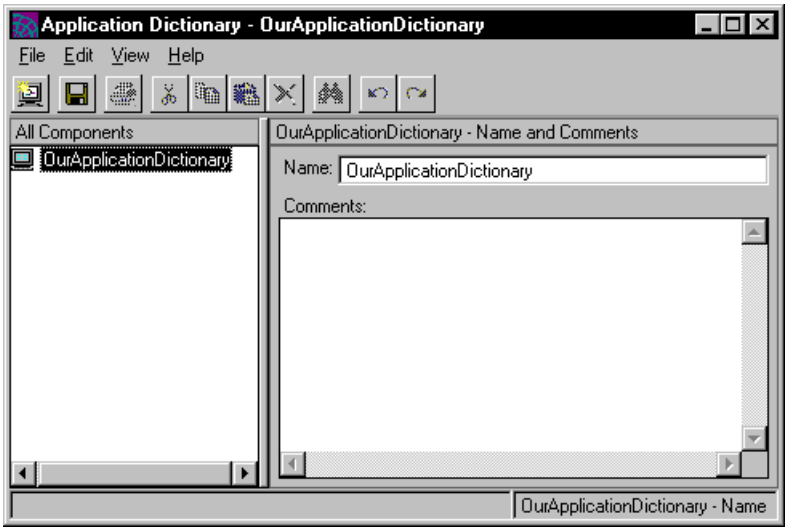
To open the Application Dictionary Workshop for an existing plan:

1 Double-click the name of an existing application dictionary in the plan list, or select the name of an existing application dictionary in the plan list and press **Enter**, or select the name of an existing application dictionary in the plan list and choose **Plan > Open**.

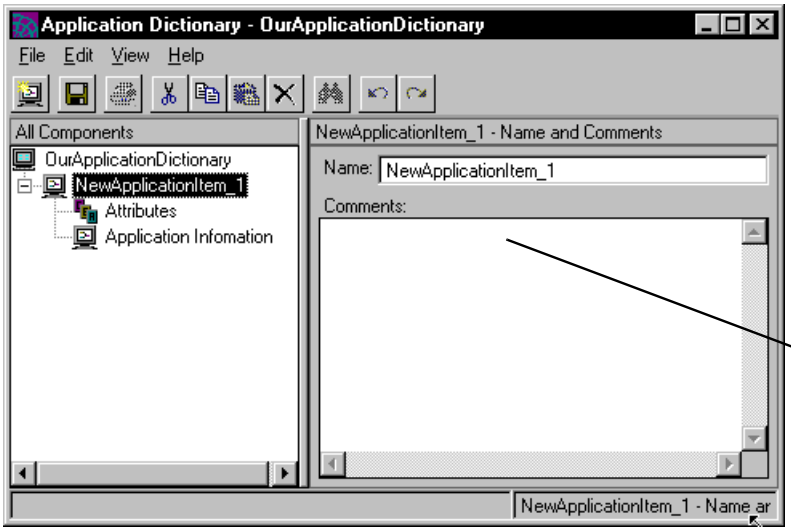
See Chapter 3, "Managing Fusion Plans: the Repository Workshop" for more information on the Repository Workshop.

Creating and Editing an Application Dictionary Item

When creating a new application dictionary, the Application Dictionary Workshop opens with a new application dictionary plan, as shown in the following figure:



To create a new dictionary item in the application dictionary, click the **New Dictionary Item** button at the top left of the toolbar or choose **File > New Application Entry**. The list view changes to display the new elements.



As you edit the application dictionary, be sure to save your changes periodically, as described in “Saving and Using an Application Dictionary” on page 111.

Specifying Application Dictionary Item Properties

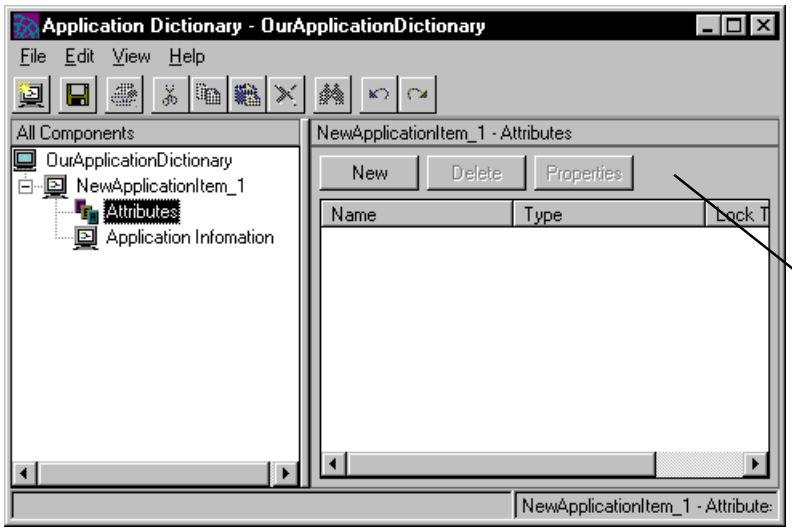
To specify application dictionary item properties, select the application dictionary element in the list view. The corresponding dialog is displayed on the right. It enables you to enter a dictionary item name and write comments about the dictionary item.

Specifying a List of Attributes

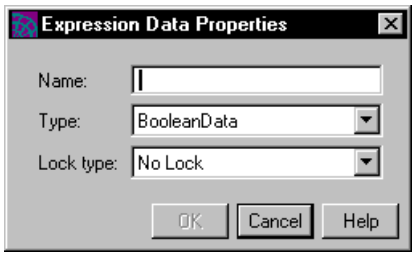
As described at the beginning of this chapter, along with the activity description and the application code, you can specify a set of process attributes to be made available to the client application when it gets this activity. This list of attributes is sent as a list in an attribute accessor, an object that groups attributes for the client application. (For more information on attribute accessors, see the description of WFAttributeAccessor in the *Forte Fusion Process Client Programming Guide*.)

If the process definition already exists, you can see which attributes are available by opening the Process Definition Workshop and looking at the Process Attributes list. You can also print a report that lists the attributes and their types from the Process Definition Workshop. You can also coordinate with the process developer to obtain the names and types of the attributes.

To specify a list of attributes, select the Attributes element in the list view. The Attributes dialog displays on the right:



To add a process attribute, click **New**. You see a properties dialog that lets you enter the name of the attribute, its data type, and the type of lock that is requested when the corresponding activity is placed in an ACTIVE state:



Name Enter the name of a process attribute that is defined in every process definition that uses this application dictionary. Process attribute names are not case sensitive—otherwise the attribute name must exactly match the definitions of the process attribute in the process definitions.

Type Choose a data type that matches that of the defined process attribute. You can choose from the drop list. The data types for process attributes are described in "Process Attribute Data Types" on page 189.

Lock type Choose from the drop list the type of lock request that is sent to the engine when the Evaluate method is executed. The lock types for process attributes are described in "Specifying Lock Types" on page 187.

Note A client application gets the accessor associated with an *offered* activity twice: once, when the client application is notified that the activity has been offered to the user (is in a READY state and placed on the session's activity list) and is available to be added to the worklist, and again, when the client application accepts the activity to perform it (is made ACTIVE).

The accessors associated with the READY state does not lock attributes. This is because many client applications might be offered this activity and place it on their work lists. Instead, the engine sets a NO_LOCK lock type. When the activity is made ACTIVE, the engine sets the locks specified in the dictionary entry. In this case, only one client application is getting access to the activity and it needs to be able to get and set attribute values accordingly.

A client application gets the accessor associated with a *queued* activity only once: when the activity is taken off the queue and made ACTIVE.

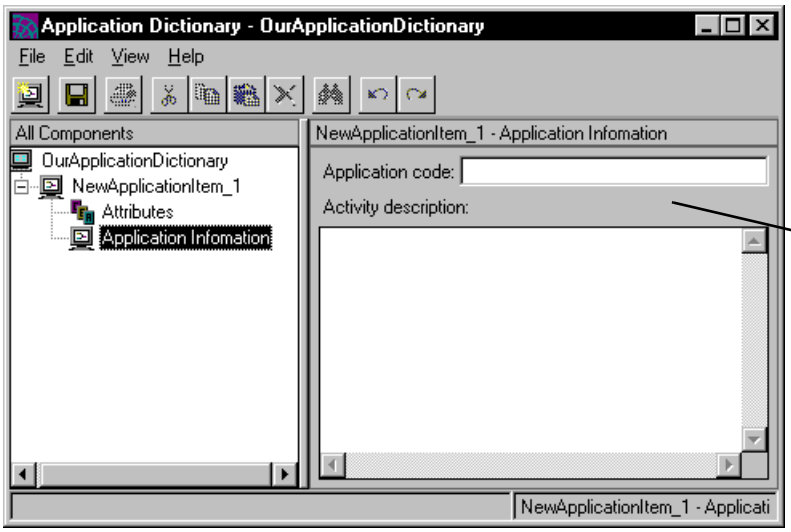
Specifying an Activity Description and Application Code

As described at the beginning of this chapter, the activity description is descriptive text that the client application can use when it displays the activity to a user.

The application code is a string of characters meaningful to the client application that tells the application what program to launch so the user can perform the activity. For example, the client application could start a commercial application for the user. Another possibility would be to open a data entry screen that is part of the client application. In any case, this code is arbitrary and is meaningful only to the client application programmer.

The activity description and application code are fixed strings defined in the Application Dictionary Workshop; any variable data that must be passed to a client application from the engine must be specified as process attributes and passed in the list of attributes, as described in "Specifying a List of Attributes" on page 109.

To specify an activity description and an application code for an application dictionary item, select the Application Information element in the list view. The Application Information panel displays on the right:



Application Information dialog

Application code Enter a code in the appropriately labeled top text entry field. The client application must be programmed to understand the meaning of this code.

Activity description Enter an activity description that might appear in a client's worklist. Note that even though this edit field is large, it is not a comment field; the client application is likely to display all the text you enter.

Saving and Using an Application Dictionary

Save All command

When you have completed work on the application dictionary, you can save it to the repository by choosing **File > Save All**. When you save changes, the current application dictionary is updated in your workspace.

Note If you have any other workshops open for editing, they are saved at the same time.

You do not need to compile or generate an application dictionary, and you do not register it with the engine. To use an application dictionary in a process definition, you only have to include it as a supplier plan to the process definition. Any application dictionary entries used in the process definition are automatically bound into that process definition when it is compiled.

As with any supplier plan, the application dictionary must be in your repository and you must have included it as a plan in your workspace. In addition, as with any other plan that is being changed at your site, every time an application dictionary you use in a process definition is changed and saved to the repository—if this change is made outside your workspace—you must update your workspace to get the changes.

If an application dictionary changes, you must go into the Process Definition Workshop, recompile the process definition, and register it again with the Fusion engine. Otherwise the changes do not propagate into the process definition.

Creating New Versions of an Application Dictionary

Generally speaking, you modify application dictionaries to add new application dictionary items in response to an expansion of the application logic domain—that is, to include new application functionality within existing processes or for new processes. It is also possible, however, that you would modify dictionary items to accommodate changes in process attributes.

How to Modify an Application Dictionary

The following guidelines show you how to modify, add, and delete application dictionary items under a number of scenarios:

Modifying an Existing Application Dictionary Item

▶ **If the modification does not require a change in process attributes:**

1. Modify the application dictionary item.
2. Make no change in process definitions.
3. Recompile and register with the engine each process definition that uses the old application dictionary item.

▶ **If the modification requires a change in process attributes:**

1. Modify the application dictionary item.
2. Modify each process definition that uses the old application dictionary item to include the new process attribute list.
3. Recompile and register each such process definition with the engine.

Adding a New Application Dictionary Item

▶ **If the new application dictionary item does not require a change in process attributes:**

1. Add the new application dictionary item.
2. Make no change in process definitions.
3. Recompile and register with the engine each process definition that uses the old application dictionary item.

▶ **If the new application dictionary item requires a change in process attributes:**

1. Add the new application dictionary item.
2. Modify each process definition that uses the new application dictionary item to include the new process attribute list.
3. Recompile and register each such process definition with the engine.

Deleting an Existing Application Dictionary Item

If the application dictionary item is not used by any process definitions:

1. Delete the application dictionary item.
2. Make no change in process definitions.

If the application dictionary item is used by process definitions:

1. Delete the application dictionary item.
2. Modify each process definition that uses the old application dictionary item (so that it uses a different application dictionary item)
3. Recompile and register each modified process definition with the engine.

Chapter 7

Creating Process Definitions

This chapter is an introduction to working with the Process Definition Workshop. It picks up where "Application and Process Logic" (Chapter 1, "Fundamentals") stops in its overview of process definitions and their components. This chapter describes how to:

- create a process definition
- create the various kinds of activities
- associate an application dictionary item and assignment rules with an activity
- write activity methods and router methods
- connect an activity to other activities and to timers
- save, compile, and register a process definition

The descriptions in this chapter assume that you have read Chapter 1, "Fundamentals" and Chapter 2, "Getting Started: the Process Development Workshops," in this manual and have a general understanding of Fusion process management, the process definition workshops, what a process definition is, and how the plans produced by the other workshops are used in a process definition.

About Process Definitions

A process definition is a representation of a business process. You create process definitions in the Process Definition Workshop, laying them out graphically as a series of connected activity definitions.

To these activities you assign application dictionary entries (to connect them to the client applications) and assignment rules (that determine which users perform them). You connect the activities with routers, optionally writing short methods that direct the flow of control—depending, possibly, on the values of process attributes that you define. You might also include timers to control the flow of process execution.

When you are done, you save the process definition, and compile it into a project. You then make a process definition distribution and register it with one or more Fusion process engines. An engine creates instances of the process definition and tracks the work defined by it.

Process definitions are composed of *activities* and timers that are linked in meaningful ways by *routers*. A process definition looks something like a flow diagram, as shown in Figure 22. A process definition also has a set of properties, which you set in its property inspector. For a discussion of how to set process definition properties and what they mean, see "Working with Process Definitions" on page 137.

The following discussion briefly describes the elements of a process definition. These are discussed in further detail later in this chapter.

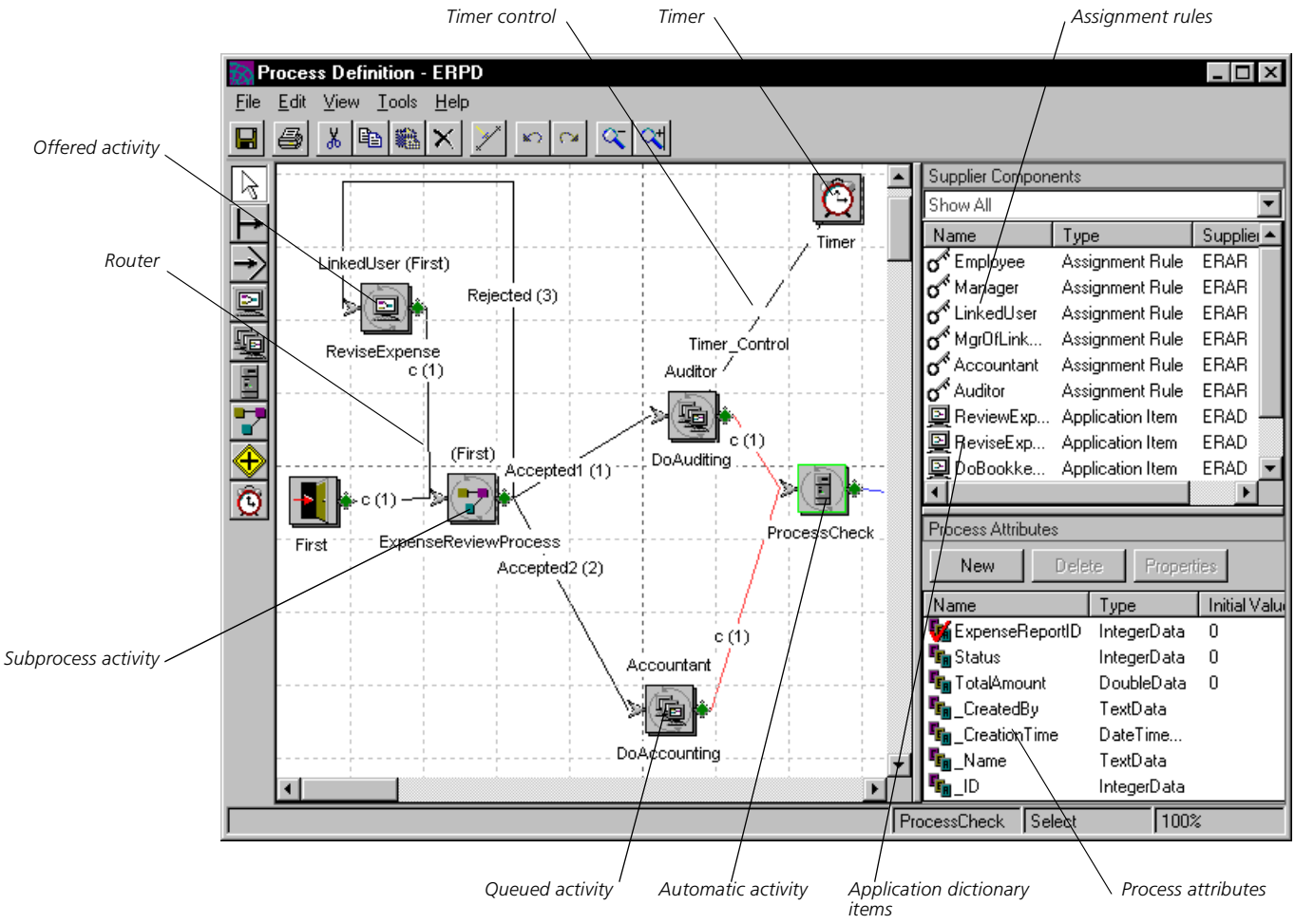


Figure 22 *Elements of a Process Definition*

Activities

A typical process definition is composed primarily of *activities* that represent sequenced units of work performed by the process. There are two primary kinds of activities—offered activities and queued activities—and a number of other special purpose activities:

- **offered activity** (described on page 121)

  An activity that the engine offers to all users who, according to the assignment rules associated with the activity, are qualified to perform the activity.

- **queued activity** (described on page 121)

  An activity that the engine stores in a queue. The queue can be accessed by all users who are qualified to perform the activity by the assignment rules associated with the activity.

- **subprocess activity** (described on page 122)

  An activity that represents a process definition. It passes control to another process definition to perform a complex set of activities, much like a method call. Subprocesses are either performed synchronously or in parallel with the invoking (parent) process.

- **automatic activity** (described on page 123)

  An activity performed by the engine rather than a user. The engine executes a method—and might invoke a service object—depending on how you define the activity.

- **junction activity** (described on page 124)

  An activity used to improve the layout of activities in the Process Definition Workshop by representing a joining or splitting of routers. A junction activity is not performed by a user.

- **First activity** (described on page 124)

  The first activity in a process definition. The First activity represents the starting point of a process instance.

- **Last activity** (described on page 125)

  The last activity in a process definition. The Last activity causes the termination of a process instance.

A process definition must contain a *First activity*, required as the initial activity in a process and a *Last activity*, required as the final activity. The Process Definition Workshop automatically inserts a First activity and a Last activity when it creates a new process definition.

Timers

A process definition can also contain *timers*, objects that can be set for a duration of time (elapsed timer) or with a deadline (deadline timer), usually to ensure that activities are completed on time. For example, in the Expense Report process definition shown in Figure 22 on page 116, if expense reports reimbursements must be handled by an auditor within a set time limit, an elapsed timer could be started when that activity becomes ACTIVE. If the timer expires (for example, after 2 days), it can trigger an activity that specifies a corrective action, or it can raise the priority of that expense report.

Timer Controls

Activities that control timers are connected to the corresponding timers through timer controls. These controls let an activity turn a timer on or off or reset it, depending on the state of the activity. Unlike routers, a *timer control* does not route process control to an activity.

Routers

Activities are connected with *routers*, lines that represent where control passes when the current activity completes. Control can pass to any number of activities simultaneously or to certain activities depending upon process conditions specified in router methods. A timer, when it expires, can also pass control to an activity through a router. Although you draw routers between activities separately from defining the activities themselves, you should regard routers as an extension of the activity or timer from which they emerge, since they represent the last processing elements of the activity or timer.

Activity Links

Activities can be linked to other activities through activity links. Information about the user of another activity is captured and passed to the referencing activity. For example, an activity link lets an assignment rule specify that one activity is performed by the same user who performed the linked activity, or by a user related in some specified way to the user who performed the linked activity.

Process Attributes

A process definition contains *process attributes*, which are variables that are global to a process instance and contain data that is meaningful to the process definition and to the activities in the process. Process attribute values are typically used in routing logic. For example, in an Expense Report process definition, you can define a process attribute to store the expense report reimbursement amount, which might affect who has to approve an expense report reimbursement request.

Suppliers

A process definition requires assignment rules and application dictionary items supplied from the corresponding supplier plans. These design elements are normally created by an application system designer before you create a process definition that uses them. When used in a process definition, the system automatically checks that any process attributes referenced by the supplier are also defined in the process definition.

About Activities

An *activity definition* is the specification of an activity in a process definition. Because the different types of activities (offered, queued, automatic, and subprocess) perform such varied tasks, they have different properties, different methods associated with them, and different assignment rules and application dictionary items associated with them.

"Application and Process Logic" on page 35 has an overview of how assignment rules and application dictionary items are used with activities. In general, these elements are used with activities that are performed by users of client applications (offered and queued activities). An assignment rule specifies who can perform the activity, and an application dictionary item sends information to the client application about what the activity is and the program used to perform it. An application dictionary item can also optionally send a set of process attributes that the client application can view and use. For more information on assignment rules, see Chapter 5, "Defining Assignment Rule Dictionaries." For application dictionaries, see Chapter 6, "Defining Application Dictionaries."

Activity States

As an activity is reached in a process instance's flow of control, the engine takes it through a series of states, as shown in Figure 6 on page 37. The states an activity can go through are described in the following table:

State	Description
PENDING	In this state a Trigger method is executed to determine if all the conditions needed for the activity to be performed have been met. If there are multiple routers coming into an activity, the Trigger method might let the first activated router make the activity READY, or it might hold the activity in PENDING state until all the routers are activated (until all required activities are complete). When the Trigger method returns TRUE, any Ready method defined for the activity is executed.
READY	If the activity is an offered or a queued activity, assignment rules are evaluated to determine which users should be permitted to perform the activity.
ACTIVE	In this state, an offered or queued activity's work is performed by the user of a client application. An automatic activity calls its OnActive method, which often executes a program. A subprocess activity calls a subprocess.
COMPLETED	If an activity completes successfully, it goes to COMPLETED state and executes its OnComplete method, and then activates one or more OnComplete routers, indicating the next activity (or activities) to be performed. Each activated router executes a corresponding router method that determines if the conditions have been met for its successor activity to be performed.
ABORTED	If an activity fails, it goes to ABORTED state and executes its OnAbort method, then activates one or more OnAbort routers, indicating the next activity (or activities) to be performed. Each activated router executes a corresponding router method that determines if the conditions have been met for its successor activity to be performed.

Activity Methods

An activity can contain various methods that control when the activity becomes READY and what to do internally when an activity changes state—for example when the associated work is completed, if the activity is aborted, and so on. You create these methods in Forte's TOOL language, using the editors or code generators provided in the Process Definition Workshop.

An important purpose of these methods is to control routing. A Trigger method controls the conditions under which an activity leaves the PENDING state. The OnActive, OnComplete, and On Abort methods control what happens when their respective states are reached. In addition, there are router methods, that control which activated routers pass control to successor activities. It is not necessary for you to write any of these methods to get basic routing functionality. If you draw routers coming into an activity and going out of one, default router and trigger methods are executed automatically.

Sharing process attributes

Some of the methods use common sets of process attributes. For example, the Trigger and Ready methods share the same set of process attributes, and the OnComplete method and the OnComplete router methods share another set. Changing the process attributes for one method changes the set of attributes for the associated method or methods. The Process Definition Workshop limits where and how you can set process attributes for these kinds of methods to ensure that as you are setting the attributes of one method, you do not unknowingly change the attributes of another.

The following table summarizes the activity methods:

Method	Description
Trigger	Defines when the activity can leave a PENDING state and become READY (available to a client). For example, this method can test if several other activities have completed before executing the activity's Ready method, if any. The default Trigger method returns TRUE when any router arrives. This method shares a set of process attributes with the Ready method.
Ready	If necessary, this method initializes process attributes or performs other work before the activity is made READY. This method shares a set of process attributes with the Trigger method.
OnActive	Performed in the ACTIVE state, this method initializes process attributes or performs other work before the engine passes control to a client application. In automatic activities, the OnActive method performs the work represented by the activity. By default the OnActive method returns TRUE.
OnComplete	Performs cleanup work necessary after the activity completes successfully, but before control is passed to any OnComplete routers. This method has a router list that specifies all the associated OnComplete routers, and shares a set of process attributes with the router methods of these OnComplete routers.
OnAbort	Performs cleanup work necessary after the activity is aborted (terminated abnormally), but before control is passed to any OnAbort routers. This method has a router list that specifies all the associated OnAbort routers, and shares a set of process attributes with the router methods of these OnAbort routers. This method and the OnAbort routers can be used to prevent the process from aborting, which is the default behavior in their absence.

The router methods determine whether a router is used and shares process attributes with the associated OnComplete or OnAbort activity method (see previous table). For more information on router methods, see "About Routers" on page 128.

Activity Links

An assignment rule evaluates characteristics of a generic user, such as the user's role. However, what if you need to specify that an activity be performed by the same user who performed another activity in the process instance, or by the manager of the user who performed the last activity? If you only had assignment rules at your disposal, you would have to use a process attribute to store the name of the user who performed another activity, then test that process attribute in an assignment rule's Evaluate method. (For more information on the Evaluate method, see "Understanding the Evaluate Method" on page 95.)

This situation is handled automatically with *activity links*. When you specify an activity link to another activity, you specify that user information from the other activity can be passed to the current activity. When the other activity is completed or aborted, the engine saves the user name and other specified information. When the engine offers the current activity, it passes the information to the Evaluate method of the current activity's assignment rules. For this mechanism to be of any use, at least one of the assignment rules must have an Evaluate method that handles the linked user name or other information, and applies it as you want for the current activity.

For example, in the Expense Report reimbursement process, there is a Review Expense activity that is offered only to the manager of the employee who files an expense report. To ensure that the correct manager performs the review, the Review Expense activity has an activity link that specifies the First activity (that is, the creator of the process instance—in this case, the user who submitted the expense report). It also has an assignment rule with an Evaluate method that takes the employee's manager's name in its otherInfo parameter. When the Review Expense activity becomes READY, the engine uses this assignment rule to evaluate all users and offers this activity instance only to this particular manager. (For an example of this Evaluate method, see "Evaluate Method Example: Linked Activity (otherInfo)" on page 98.)

Activity links have special behavior regarding the passing of linked activity information between activities in a parent process and a subprocess. See "Setting the Subprocess Activity Link" on page 154 for more information and an example.)

Offered and Queued Activities

Offered and queued activities are performed by users of client applications.

Offered activities

The engine *offers* an offered activity to all users who are qualified to perform the activity by the assignment rules associated with the activity. An offered activity is typically displayed by the client application on a work list from which the user picks an activity to perform. The first user to pick an activity gets it. Offered activities are typically performed by users of heads-up client applications (see the *Forte Fusion Process Client Programming Guide* for a description of heads-up applications), but can also be performed by automated clients or services that create sessions with a Fusion engine.

Queued activities

The engine stores a queued activity in a queue, which is ordered according to criteria you specify. The queue can be accessed by all users who are qualified to perform the activity by the assignment rules associated with the activity. Each defined queued activity has its own queue in the engine (identified by the name of the activity and process definition), and activity instances are added to the queue from the various process instances executing in an engine. A queued activity is pulled from the top of the queue and performed by a client application. Queued activities are typically performed by users of heads-down client applications (see the *Forte Fusion Process Client Programming Guide* for a description heads-down applications), but can also be performed by automated clients or services that create sessions with a Fusion engine.

Queued and offered activities go through four states: PENDING, READY, ACTIVE, and COMPLETED/ABORTED. The following figure shows the states of an offered activity and the associated methods:

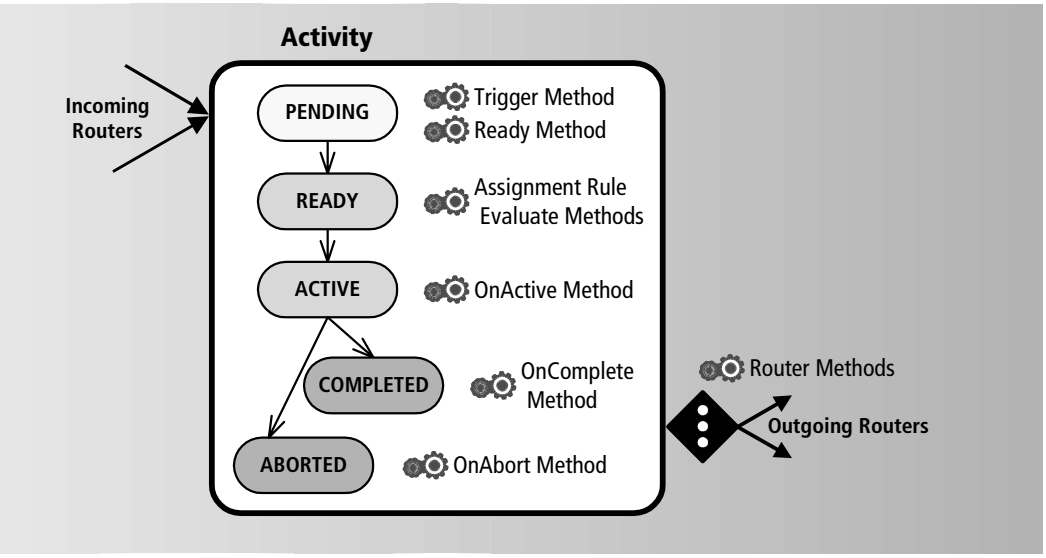


Figure 23 *Activity States and Associated Methods for Offered and Queued Activities*

Both an offered activity and a queued activity can have assignment rules associated with them that tell the engine who can perform the activity. Both these types of activities also have an application dictionary item associated with them. The application dictionary item tells the client applications what information to display for the activity and what operation or program to invoke to perform the activity. It can also supply a list of locked process attributes for the activity.

Figure 24 represents the contents of an offered or a queued activity.

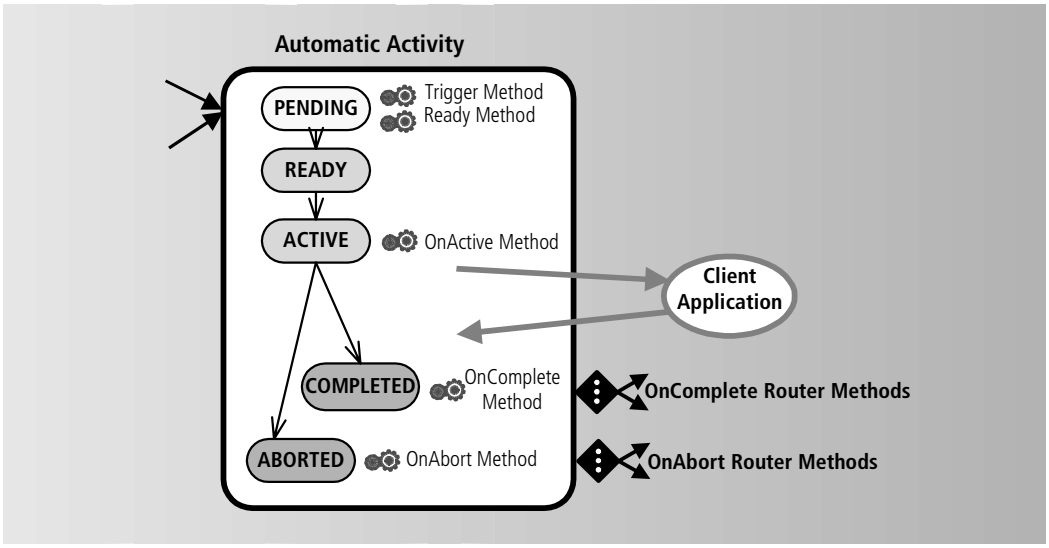


Figure 24 *Offered and Queued Activity Elements*

Subprocess Activities

A subprocess activity represents a process definition. It passes control to another process definition to perform a complex set of activities, much like a method call.

A subprocess activity does not directly interact with a client application and therefore does not have assignment rules or an application dictionary item associated with it. Since it represents a subprocess, it passes directly from a PENDING state to an ACTIVE state, and does not have associated Ready or OnActive methods.

Figure 25 represents the contents of a subprocess activity:

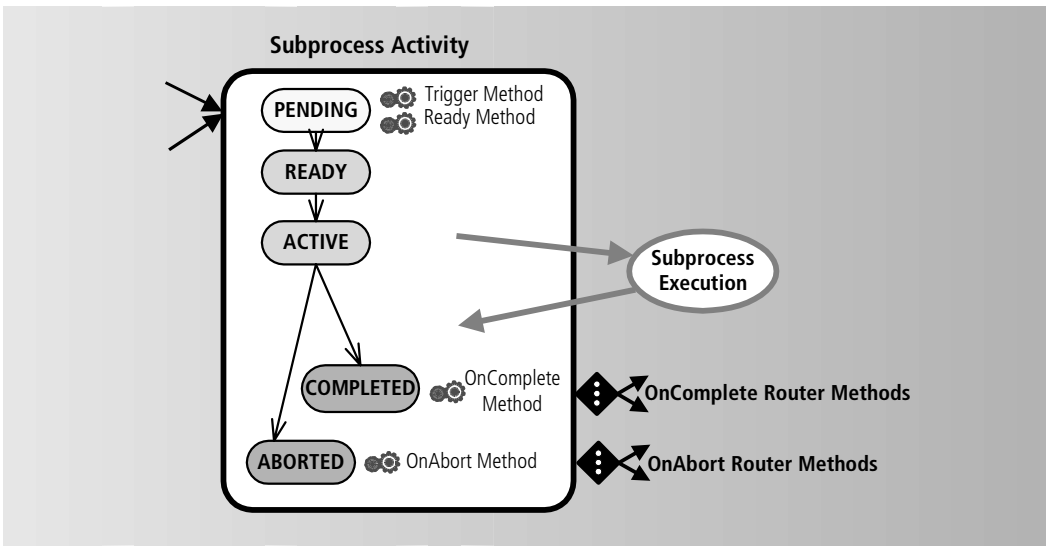


Figure 25 *Subprocess Activity Elements*

Subprocess activities are the only activities that have input and output attributes. Just as with a method, input attributes are attributes passed from the subprocess activity to the subprocess itself and are used by the subprocess in doing its work. Output attributes are passed back from the subprocess to the calling subprocess activity. The handling of these attributes also depends on whether the subprocess is performed synchronously or in parallel with the invoking (parent) process. For more information on subprocess activities, see "Working with Subprocess Activities" on page 152.

Automatic Activities

An automatic activity executes a program or invokes a service object to perform the activity, calling it directly from the engine without requiring interaction with the user of a client application. Because it does not interact with client applications, an automatic activity does not have assignment rules or an application dictionary item associated with it. However, because it is performed by the engine, an automatic activity can hold up process execution.

An automatic activity has the same states and associated methods as offered and queued activities, shown in Figure 23 on page 121.

Figure 26 represents the contents of an automatic activity.

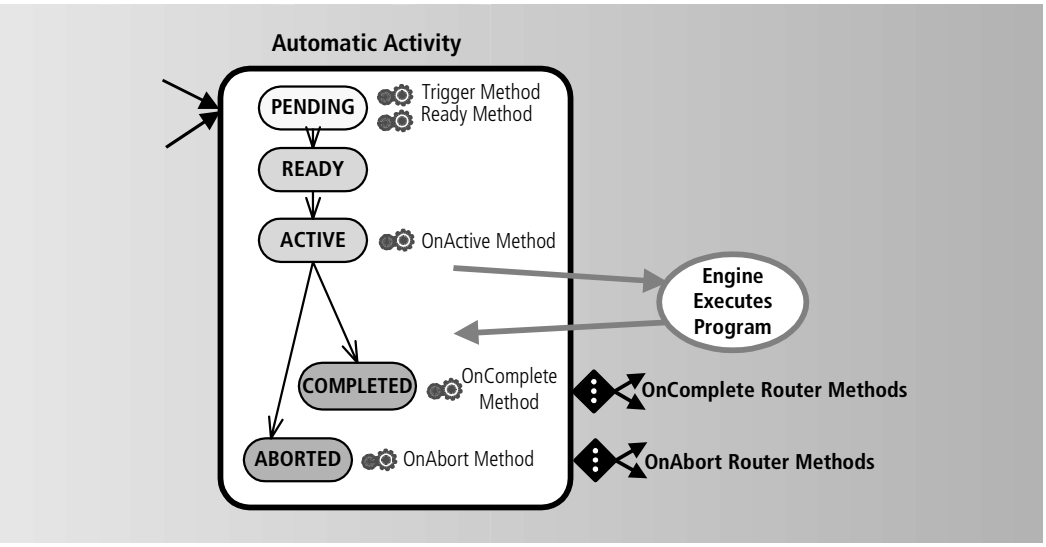


Figure 26 *Automatic Activity Elements*

For more information on automatic activities, see "Working with Automatic Activities" on page 155.

Junction Activities

A junction activity can improve the layout of activities in the Process Definition Workshop by representing a joining or splitting of routers. It can also economize the use of router and trigger methods in complex routing situations, as shown in Figure 27.

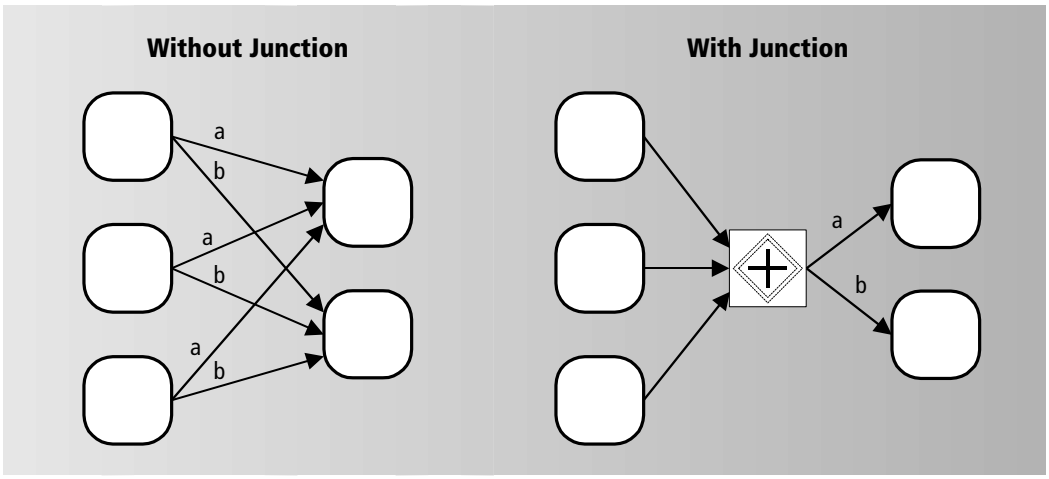


Figure 27 *Use of a Junction Activity*

A junction activity is not performed by a user. It simply serves as a convenient way to improve your process definition layouts.

First Activity

The First activity is a required activity that represents the beginning of a process. There is only one in a process definition. The First activity goes automatically through PENDING, READY, and ACTIVE states and always completes successfully. The only method associated with a state is the OnComplete method, which can have multiple OnComplete routers associated with it, just like any other activity. Timer controls are processed normally and can refer to any state. (See "Timer Controls" on page 126 for more information.)

The First activity is not performed by a user in the usual sense, and therefore does not have assignment rules or an application dictionary item. However, because it represents the creation of a process instance, the First activity does have an associated user: the user who created the process instance through a client application. Therefore, an activity that needs to designate the user who started a process instance as a linked user can do so by creating an activity link to the First activity. (See "Activity Links" on page 120 for more information on linked users.)

For information on how linked activities apply to subprocess activities, see "Setting the Subprocess Activity Link" on page 154.

Figure 28 represents the contents of a First activity.

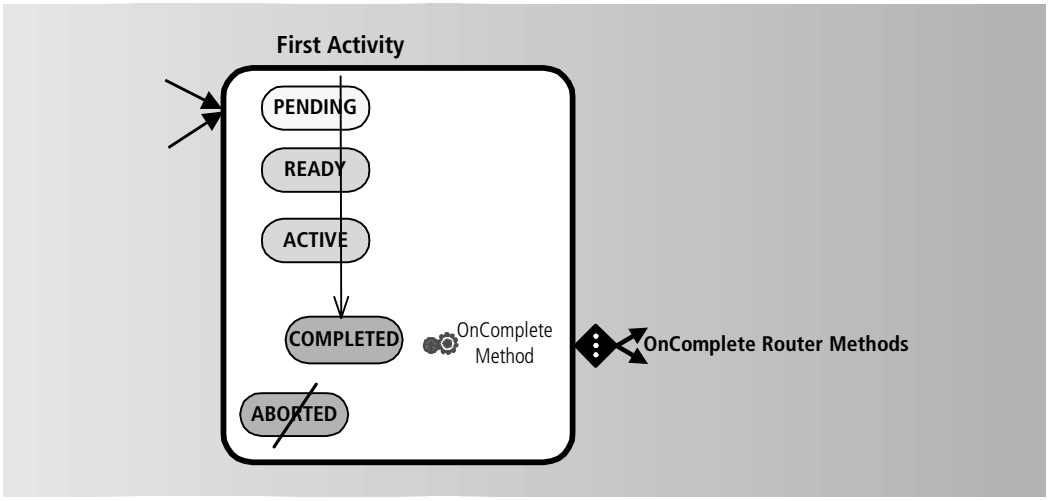


Figure 28 *First Activity Elements*

Last Activity

The Last activity is a required activity that represents the end of a process. There can be only one in a process definition. The Last activity goes through normal PENDING and READY states, passes automatically through the ACTIVE state, and then ends the process instance by entering a COMPLETED state. It never enters the ABORTED state. Because routers come into it, the Last activity has a Trigger expression, which controls when the process can end. It also has a Ready method, which does any final cleanup required before the process is ended.

The Last activity does not get performed by a user and therefore does not have assignment rules or an application dictionary item; however, in the case of a subprocess, it can link to another activity. For more information on this case, see "Setting the Subprocess Activity Link" on page 154.

Because the Last activity does not route to other activities, it has no OnAbort or OnComplete method and does not connect to any routers.

Figure 29 represents the contents of a Last activity.

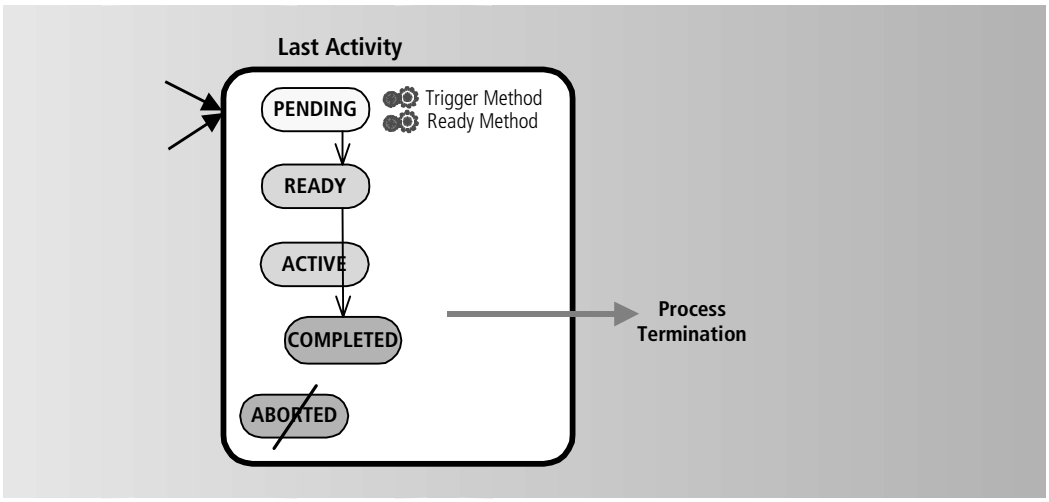


Figure 29 *Last Activity Elements*

About Timers

A timer is an object that can be set for a period of time, like a kitchen timer (an *elapsed timer*), or to a date and time, like an alarm clock (a *deadline timer*). If the time period expires or the date and time is reached, the timer performs an appropriate expiration action, usually routing to an activity to handle the timeout condition.

For example, in the expense report reimbursement process shown in Figure 30, expense reports are submitted to a manager, who has to approve or reject the expense report request. When an expense report is created (a process instance is created), it starts a timer and routes to the Review Expense activity. The Review Expense activity is where a manager checks the information entered in the expense report, and there is a time limit of three days for this work to begin. If the work does not begin in three days, the timer expires. When the timer expires, an automatic activity can notify the manager and restart the timer.

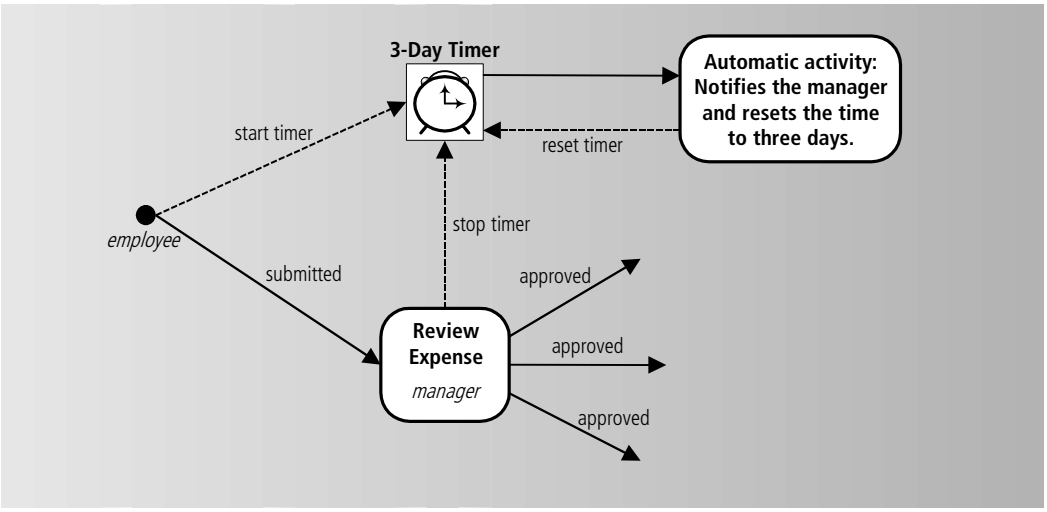


Figure 30 *An Elapsed Timer in a Expense Report Reimbursement Process Definition*

A timer has an OnExpiration method (similar to an activity's OnComplete method) and one or more associated Expiration routers that share a common set of process attributes, as shown in the following diagram.

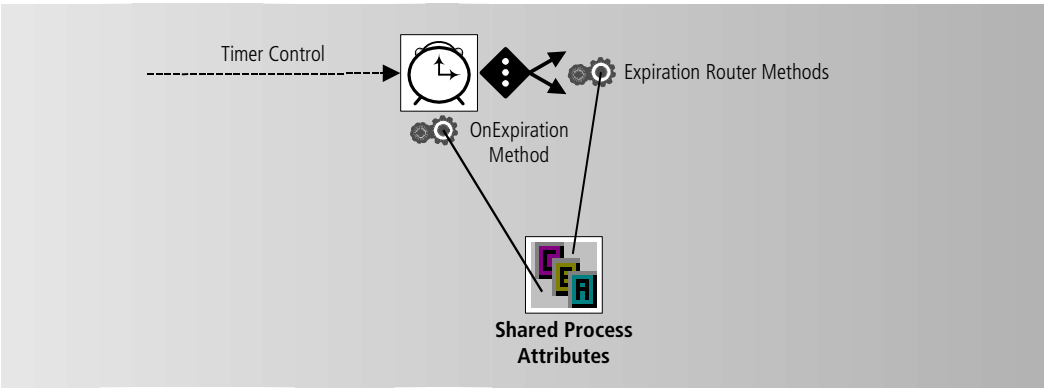


Figure 31 *OnExpiration Method of a Timer and Router Methods of its Timers*

Timer Controls

The connector from an activity to a timer is called a *timer control*. It is not considered a router because starting a timer does not affect the flow of control or cause work to be done. (For example, an activity can start a timer when it becomes READY, then stop the timer when it becomes COMPLETED.)

Timer controls and activity states

A timer control is activated by a transition to a specific activity state. For example, if there were a requirement that an activity be performed within two hours from the time a user picked it, you could define an elapsed timer that runs for two hours, and then have one timer control turn the timer on when the activity becomes ACTIVE, and another timer control turn the timer off when the activity becomes COMPLETED.

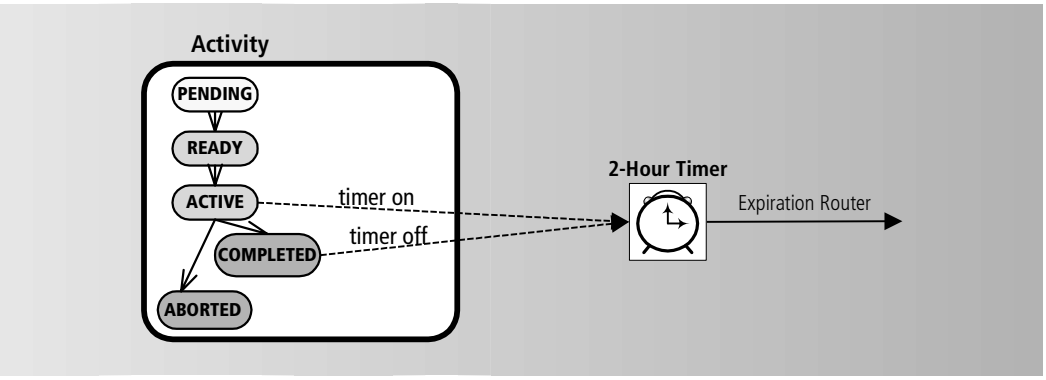


Figure 32 *Timer Controls and Activity States*

Although the connector from an activity to a timer is a control, the connector from a timer to an activity is a router. If the timer expires, an activity must be started to handle the expiration.

In the previous example, in Figure 30 on page 126, the timer starts when the process is created. The flow of control goes to the Review Expense activity, which is waiting in READY state for someone to perform it. The flow of control does not go to the timer. However, if the timer expires, it creates another activity, affecting the flow of control.

Types of Timers

There are two types of timers:

- elapsed timer—a timer that is set for a duration of time (as illustrated in Figure 30)
- deadline timer—a timer that is set to expire on a date and time

Elapsed Timers

ElapsedOn method

In its simplest form, an elapsed timer is set for a period of seconds, minutes, hours, days, or greater increments, and it expires when that amount of time has passed. This behavior is set by the timer's ElapsedOn method, which is called by the engine every time an elapsed timer is turned on. By default this method simply calculates how much time is left by adding the block of time indicated in the timer to the current date and time indicated by the computer system, yielding an expiration time.

ElapsedOff method

Whenever an elapsed timer is turned off, the engine calls its ElapsedOff method, which calculates how much time is left from the current date and time to the expiration time of the timer.

You have the option of specifying more complex behavior with these two methods, such as calculating by business days (skipping two days for each weekend). For example, a business day timer that is set for three business days might be started on a Thursday at 9 am. Its ElapsedOn method calculates its expiration time as the same time on the following Tuesday. If it is subsequently stopped on Friday at 9 am, the timer's ElapsedOff method indicates that two business days are left on the timer. If it is then started up on Monday at 9 am, the timer's ElapsedOn method calculates the expiration time as Wednesday at 9 am.

Deadline Timers

DeadlineInit method

The simplest type of deadline timer is one that is set to expire on a specific date at a specific time. This behavior is set by the timer's DeadlineInit method, which is called by the engine every time a deadline timer is initialized or reset. By default this method simply returns a set date and time.

For example, Brunhilda's husband, Trog, has a birthday on February 29, 2004. In 1997, she sets a deadline timer for 10:00 am on February 26, 2004, to remind her to get a card and a present. She subsequently stops the timer for a year, and then restarts it in 1999. The timer is still set for the same date and time because its DeadlineInit method simply returns that setting.

However, Brunhilda wearies of having to set the timer to a new value every leap year, especially after forgetting once and having to deal with a moping Trog for the subsequent two years, so she writes her own DeadlineInit method that calculates the birthday from the current date and time. Now when the timer expires, she can reset it, and it determines when the next leap year is and sets the date and time itself.

About Routers

To indicate process flow in the Process Definition Workshop, an activity is graphically connected to another with a router. Although you draw routers between activities separately from specifying the activities themselves, you should regard a router that leaves an activity as an element of that activity, because it represents the last processing elements of the activity. An activity designates in its OnComplete and OnAbort router lists which routers are activated when the activity completes or is aborted. Each router has a router method that determines whether the router actually transfers process control to the activity to which it points. The router transfers process control if the router method evaluates TRUE.

An activity can be connected by routers to more than one successive activity. When the current activity completes or is aborted, the engine calls its OnComplete or OnAbort method, and then evaluates the router methods of all its OnComplete or OnAbort routers, respectively, to determine where the flow of control is to go next. You can specify the order in which the router methods are executed. Control can go to more than one activity, in which case the successor activities execute in parallel. In addition, an activity can route back to itself when an activity needs to be performed multiple times.

Router methods typically use process attributes to determine where to transfer process control. For example, in the Expense Report process definition, when a manager approves an expense report, the Review Expense activity's OnComplete method passes control to two routers. One routes control to the manager's manager for approval (for expense report amounts over $1,000), and another routes control directly to accounting and auditing for further processing (for amounts under $1,000).

To determine which of these routers transfers process control when the activity completes successfully, each router has a router method that compares the value of the process attribute TotalAmount to the value $1,000. The method that evaluates TRUE is the one which transfers control.

The set of process attributes used in a router method are specified in the corresponding OnComplete or OnAbort method. For example, the TotalAmount process attribute is designated in the Review Expense OnComplete method and applies to both OnComplete router methods.

Routers are also used to transfer process control from an expired timer to an activity. Like activities, timers can have multiple Expiration routers, each with a corresponding router method that can use process attributes to determine whether to transfer process control.

Abort Router Handling

The engine handles abort routers as follows:

If a READY or PENDING activity is aborted and either has no routers or all return FALSE, the engine takes no action.

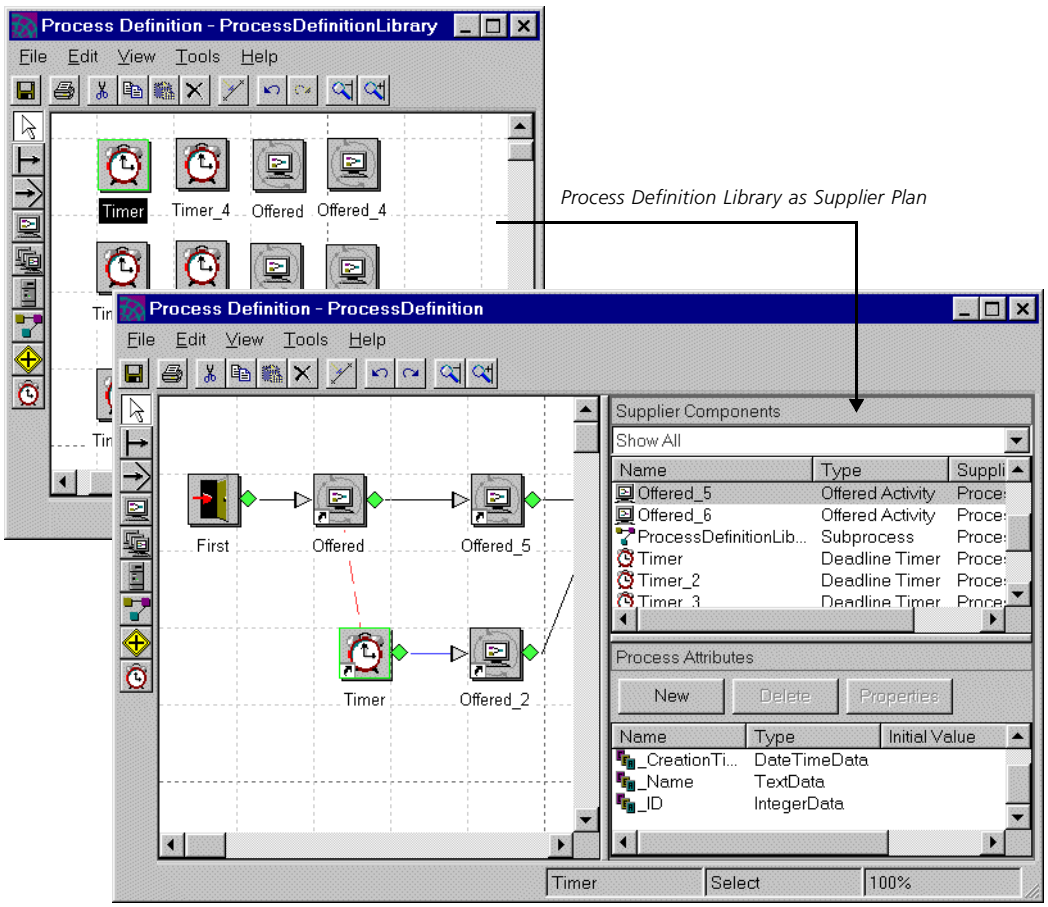
If the activity is ACTIVE when aborted and there are no routers or all return FALSE, the process instance is aborted.

If any router expression returns TRUE, no matter what state the activity is in when aborted, those routes are taken.

Creating a Process Definition Library

At times, you may want to use the same activity or timer in various process definitions of your workflow application. For example, you may have a timer definition that is used throughout your application with the same settings. You can place multiple copies of the timer in various process definitions, but if you later want to modify the behavior of the timer, you need to update each copy.

However, you can create a *process definition library* that contain a set activities and timers that you intend to reuse. You supply this library to your process definitions, and then place *references* to the activities and timers in your process definitions. If you want to modify the behavior of a referenced activity or timer, you make the changes in the library. The changes are then automatically propagated to each process definition containing a reference to the activity or timer.



Only offered activities, queued activities, automatic activities, and timers can be supplied by a process definition library to another process definition. Also, a process definition supplier plan does not necessarily have to be a library—you can use any process definition in your workflow application to supply activities or timers. However, you typically keep all reused components in libraries to simplify maintenance.

Note A process definition library must contain a first and last activity even though these activities are not used in the library and cannot be supplied to another process definition.

Reference Properties

Some properties of activities or timers supplied by a library are maintained in the library and propagated to all references. Others are set and maintained in the reference. For example, in a supplied timer, the Timer Type and Timer Value are maintained in the library because you want the behavior of each reference to be the same. But the OnExpiration Router for a supplied timer is maintained in each reference because each reference uses the timer in a different area of the workflow application.

The following table lists the properties that are maintained in the library and those that are maintained in the reference.

Activity/Timer	Properties Maintained in the Library	Properties Maintained in the Reference
Offered Activity Queued Activity Automatic Activity	Application Dictionary Item Assignment Rules Comments OnActive Method/Attributes	Action On removal Activity Link Name OnComplete Method/Attributes/Routers On Session Suspend Priority Attribute Ready Method/Attributes Trigger Type Trigger Method/Attributes
Timer	Comments ElapsedOn Method/Attributes Elapsed Off Method/Attributes Timer Type Timer Value	Name Timer on at process start OnExpiration Method/Attributes/Routers

Note Properties of an activity or timer in a process definition library that are not referenced are set to standard default values.

Working with Process Definition Libraries

A process definition library is created by setting the Supplier Library property for a process definition.

▶ **To create a process definition library:**

1. From a Repository Workshop, choose **Plans > New Conductor Plans > Process Definition** to create a new Process Definition plan.
2. In the Process Definition Workshop that opens, choose **File > Properties** to open the Property Inspector.
3. In the Name tab of the Property Inspector, select the Supplier library toggle and click **OK**.

Note If you do not enable the Supplier library option, compiling the process definition may fail.

4 Within the Process Definition Workshop, create the activities and timers that you intend to supply to other process definitions.

Do not specify routers or timer controls in the process definition library. Properties for routers and timer controls are not propagated. Activities and timers in a library should only be used by process definitions that specify the library as a supplier plan.

To supply a process definition library to another process definition:

1 In the workshop for the process definition that you want to supply the library to, choose **File > Supplier Plans**.

2 In the Supplier Plans dialog that opens, drag the process definition library from the Available Plans to the Supplier Plans.

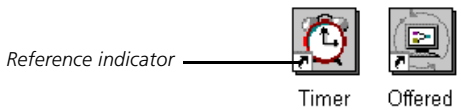
3 Click **OK**.

After supplying a process definition library to another process definition, the activities and timers in the plan are listed in the Supplier Components list of the Process Definition Workshop. You can use the drop down list for Supplier Components to filter the display of components. For example, select Timers to list all the supplied timers.

To add an activity or timer reference to a process definition:

1 In the Supplier Components list, click and drag a supplied activity or timer to the Layout Area.

The referenced activity or timer appears in the layout area. The icon for the referenced activity or timer indicates that it is a reference:



You can change an activity or timer reference in a process definition so it loses the reference and behaves as if it were created directly within the process definition. It becomes a copy of the referenced activity (or timer), with all properties now being maintained within the process definition. Because it is now a copy, changes made to it no longer propagate to the original reference.

To convert a reference:

1 In the Layout Area, select the activity or timer reference.

2 Choose **Tools > Remove Reference**.

You can also use the Based-on property of an activity or timer to change the behavior or activities or timers in your process definition.

To change or remove references for an activity or timer using the Based-on property:

1 In the Layout Area, right-click on the activity or timer and select Properties to open its Property Inspector.

2 In the Property Inspector, open the Based-on drop down list (from the Name tab) to view the available choices.

Selecting None removes any reference to the activity or timer. You can also select from any of the imported components displayed in the drop down list to change the reference to another supplied activity or timer.

3 Close the Property Inspector.

Working with Process Definitions

This section describes the series of tasks you are likely to perform when you create or update a process definition. It covers the following topics:

- opening the workshop
- workshop overview
- working with the various kinds of activities and writing their methods
- working with routers
- working with timers
- saving, compiling, and registering a process definition

Opening the Process Definition Workshop

This section contains procedures for creating a new process definition and opening an existing process definition from the Repository Workshop (illustrated in the following figure).

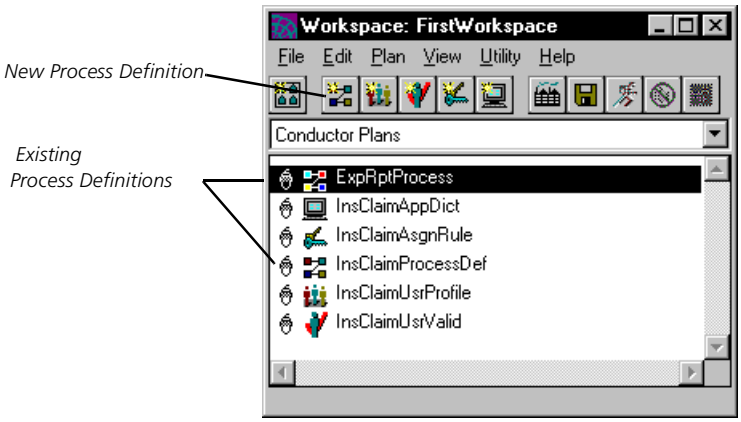
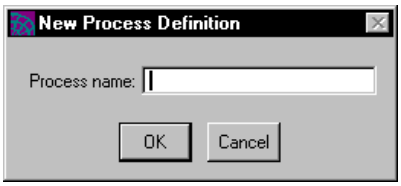


Figure 33 *Opening a Process Definition in the Repository Workshop*

To open the Process Definition Workshop and create a new plan:

1 From the Repository Workshop, click the New Process Definition toolbar button, or choose **Plan > New Fusion Plans > Process Definition**.

A dialog opens prompting you to name the process definition.



2 Name the process definition, and click **OK**.

A new user profile plan opens in the Process Definition Workshop.

To open the Process Definition Workshop for an existing plan:

1 From the Repository Workshop, double-click an existing process definition in the plan list, or select a process definition in the plan list and press Enter, or select a process definition in the plan list and choose **Plan > Open**.

See Chapter 3, "Managing Fusion Plans: the Repository Workshop" for more information on the Repository Workshop.

Workshop Overview

The Process Definition Workshop has three main areas, illustrated in Figure 34:

- The layout area

  An area for arranging and defining the components of the process definition, such as activities, timers, and routers.

- The Supplier Components list (to the right of the layout area)

  This list shows *predefined* components that you can add to the process definition, such as assignment rules, application dictionary entries, and subprocesses. The contents of this list are defined in other workshops, such as the Assignment Rule Workshop. The plans produced from these other workshops must be specifically included in your current process plan as supplier plans. (For a description of how to include a supplier plan see "Adding Supplier Components" on page 135.)

- The Process Attributes list (below the Supplier Components list)

  This list shows *process attributes* that you add to the process definition and use in methods, application dictionary entries, activity methods, and so on.

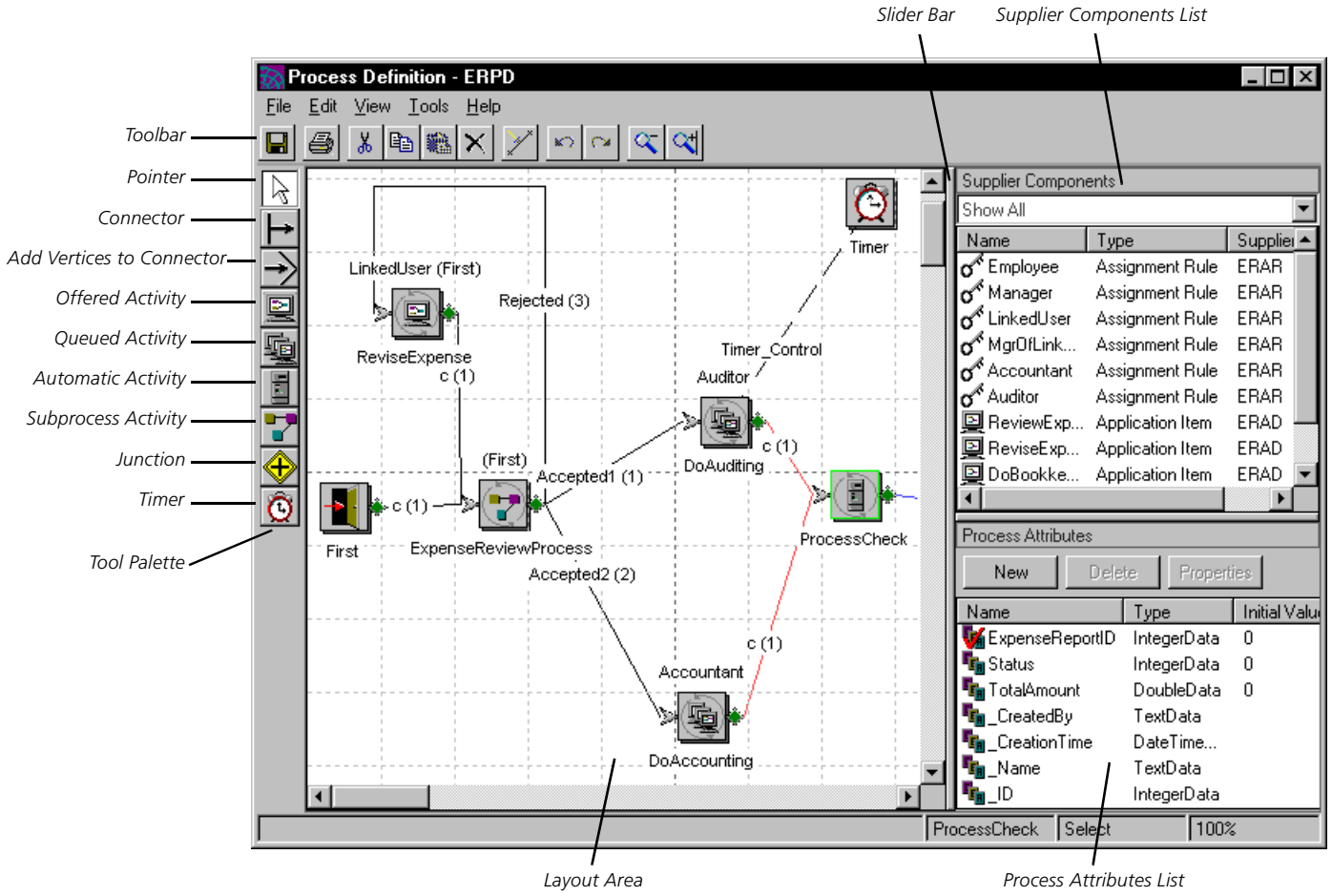


Figure 34 *Process Definition Workshop*

Adding Objects to the Layout Area

The simplest way to add objects, such as activities and timers, to the layout area is to click the tool you want in the tool palette, then click the spot on the layout area where you want to add the object. (You can also click the Tools menu to choose from a list of tools.) If you want to use the same tool repeatedly, choose **Tool > Repeat**, and you won't have to select the tool each time you want to add an additional object of the same type.

You can add preexisting objects from another process definition in two ways:

- Open the source process definition and use the **Edit > Copy** and **Paste** commands to copy the activity or timer from the source process definition. The Fusion engine checks the consistency of process attributes between the two process definitions.
- Include the source process definition as a supplier and then drag the activity or timer from the Supplier Components list and drop them in the layout area.

Note

Everything on the tool palette has been described previously in this chapter except the Pointer, which is the default selection tool, and the Add Vertices tool. Use the Add Vertices tool to move line segments of routers and controls. For example, in Figure 34, the router coming out of the timer at the top of the figure has had three vertices added to it with this tool.

To remove vertices in a router or linker, select the vertex, then click the Straighten Connector button on the toolbar.

Menu Bar

The Process Definition Workshop menubar provides all the commands you can execute in the workshop. The menus are summarized below. The main window menus are:

File menu Provides commands relevant to the process definition as a whole: setting properties, including supplier plans, saving, distributing (registering with an engine), printing, and so on.

Edit menu Provides commands relevant to a selected item in the layout: setting properties, cutting, copying, pasting, deleting, and so on.

View menu Provides commands for altering the appearance of the workshop window: displaying the tool palette, Supplier Components list, Process Attributes list, toolbar, status bar, and so on.

Tools menu Provides commands for selecting the tools that appear on the tool palette.

Help menu Provides online help for Fusion.

Right Mouse Button

The Process Definition Workshop supports a popup menu activated by the right hand mouse button. The commands on the popup menu depend upon the item selected in the layout area and represent a subset of the commands you can access from the menu bar:

Undoing Work

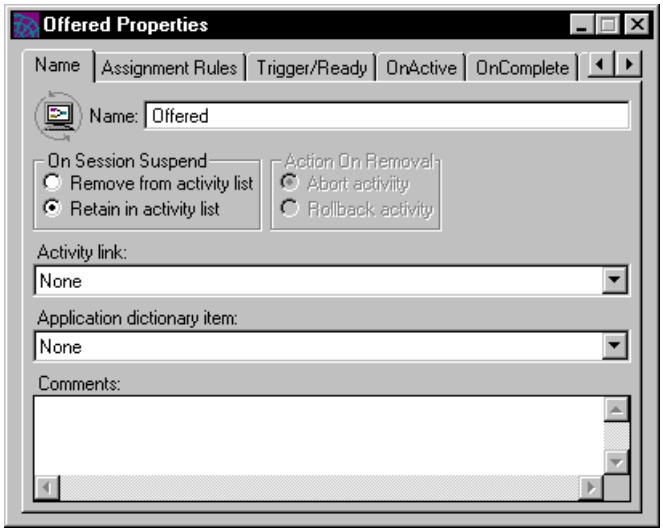
The Process Definition Workshop has the following mechanisms for undoing work you have performed but don't want to save:

Undo/Redo You can undo one or more sequential operations by selecting the **Edit > Undo** command one or more times. Similarly you can restore these operations by selecting the **Edit > Redo** command.

Cancel You can undo all operations since the last **File > Save All** command using the **File > Cancel** command. This discards all changes since the last save operation.

Working with Property Inspectors

To customize the objects you add to the layout area, use the object's property inspector. To display an object's property inspector, double-click the object (or select the object, then choose **Edit > Properties**). For example, if you double-click an offered activity, the following property inspector opens:



A property inspector is a tab folder. Clicking each tab gives you a separate tab page for editing information on the object. This particular property inspector is displaying the Name tab page. From this page, you can change the name of the activity, add activity links and application dictionary entries, and enter comments about the activity.

Other tab pages for this property inspector allow you to add assignment rules to the activity and edit the activity's methods. (Editing properties for an offered activity is discussed in detail in "Working with Offered Activities" on page 140).

Notes

Before opening a property inspector to specify an object's properties, it is useful to have previously added activities, process attributes, and other elements of the process definition, and to have connected objects with routers. Then when you open the property inspector for activities and timers, routers are already connected, process attributes already exist, and if you are linking to any activities, they are already there.

A property inspector always stays on top of other windows on your display. It displays the properties of only one object at a time. If you select other objects in the layout area while a property inspector is open, the property inspector changes to show you the properties of the new object.

The process definition itself has properties, which you can modify by choosing **File > Properties**. These properties are described later in "Working with Process Definitions" on page 137.

Adding Supplier Components

The objects in the Supplier Components list, such as assignment rule dictionaries, application dictionaries, and other process definitions, are from plans created in their corresponding workshops. In these workshops, you can save and compile the corresponding plan, and (in some cases) make a distribution and register it with an engine, as described in the individual workshop chapters.

Supplier components are needed to create a process definition. In order to specify the activities in a process definition, you need to associate assignment rules and application dictionary items from the Supplier Components with each activity. In addition, if you want

to use activities and timers specified in another process definition in your current process definition (as described in "Creating a Process Definition Library" on page 129), the source process definition must be included in the Supplier Components list.

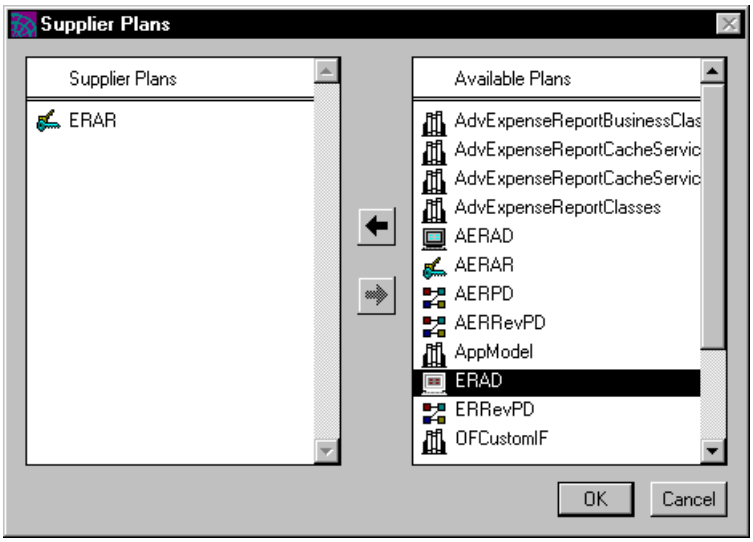
Note If you only plan on referring to another process definition using a subprocess activity in the subprocess definition, the source process definition does not need to be included in the Supplier Components list.

If you are working in the Application Dictionary Workshop or the Assignment Rule Workshop in your own workspace, all you need to do is save the plan to have access to it from the Process Definition Workshop. If someone else is working on one of these plans in their own workspace, they must integrate their workspace so you can get access to it.

When a plan you want to include in your process definition is available in your repository, you include it in your process definition by using the **Supplier Plan** command.

To include supplier plans:

1 In the Process Definition Workshop, choose **File** > **Supplier Plan**. You see the Supplier Plans dialog.



2 Select a plan from the Available Plans list on the right.

3 Click the left arrow between the lists to move the plan to the Supplier Plans list. (You can also double-click on the plan or drag and drop it.)

4 Add the other supplier plans you need.

5 Click **OK**.

You see the contents of the supplier plans appear in the Supplier Components list (see Figure 22 on page 116).

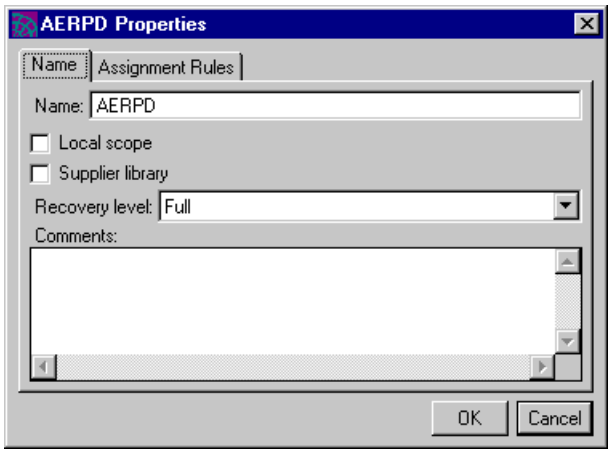
If these supplier plans are updated, for example, if new assignment rules are added to an assignment rule dictionary, you do not have to make any changes in your supplier plans. You might have to update your workspace to get the latest versions of the plans from the repository, and you might have to close and reopen your Process Definition Workshop to see any new elements.

Working with Process Definitions

The process definition itself is an object, just as its elements are, and it has properties: a name, a comment field, and assignment rules for determining who can create a new process instance.

Note Technically speaking, process attributes are also properties of the process definition, but because they are used extensively by activity methods, assignment rules, application dictionary entries, and so on, there is a separate, more visible list for them. How to add process attributes to a process definition is described later in "Defining Process Attributes" on page 138.

You set a process definition's properties by choosing **File > Properties**, which displays the following property inspector:



Specifying Process Definition Properties

To specify process definition properties, select the Name tab page in the process definition property inspector. The Name tab page gives you access to the following properties:

Name Displays the process definition name. You cannot rename the process definition. To change its name, you have to create a new process definition and copy all the elements from the old process definition to the new one.

Supplier library Only enable this option if the process definition is being used as a supplier library. Supplier libraries allow you to reuse activities and timers in other process definitions. For more information on supplier libraries, refer to "Creating a Process Definition Library" on page 129.

Recovery level Indicates the level at which you want to recover this process in the event of a failure. Lowering the recovery level is useful if full recovery is not needed for the process in the event of engine or system failure, and you want to conserve system resources. The recovery level options are Full (the default), Process Only, and None. If full recovery is not needed, choose Process Only or None.

- **Process Only** restarts the process instance; there is no recovery of process state information.
- Selecting **None** specifies that no recovery is to be performed for this process.
- Specifying **Full** (Recovery) for the process means during recovery, all current state information for the process is recovered from the Fusion engine database. This includes the state of each process, activity, timer, and process attribute lock that is created in the course of process execution. The Full Recovery default automatically applies to any processes previously defined in earlier versions of Fusion.

Note

If you specify full recovery, state information for the process is recovered only if the engine is also configured for state recovery. If logging options for the engine are turned off, however, the process definition recovery level setting do not override the engine setting. For more information, refer to *Forte Fusion Process Management System Guide*.

Local Scope Indicates whether the process definition, at runtime, can be accessed from a remote engine. Normally a process definition is accessed from a remote engine if it is being used as a subprocess of a process definition executing on the remote engine. If you enable Local scope, the process definition can only be accessed by a parent process definition executing on the same engine.

Comments TAllows you to enter internal comments about the process definition. Comments are a way to document information that may be useful to other developers.

Specifying Assignment Rules for Process Creation

Assignment rules for a process definition control which users of client applications can create a new instance of the process definition. As described in "Adding Supplier Components" on page 135, you must have already added the corresponding assignment rule dictionary as a supplier plan before you can use the individual assignment rules in the process definition. (See Chapter 5, "Defining Assignment Rule Dictionaries" for a complete description.)

To add assignment rules for process instance creation, click the Assignment Rules tab to display the tab page, then click the Add button to display the Add Assignment Rules dialog. shown in Figure 35.

All the assignment rules in your supplier list appear in the Add Assignment Rules dialog. Choose the rules you want to add, then click OK to add them. The names of the added rules appear in the Assignment Rules tab page.

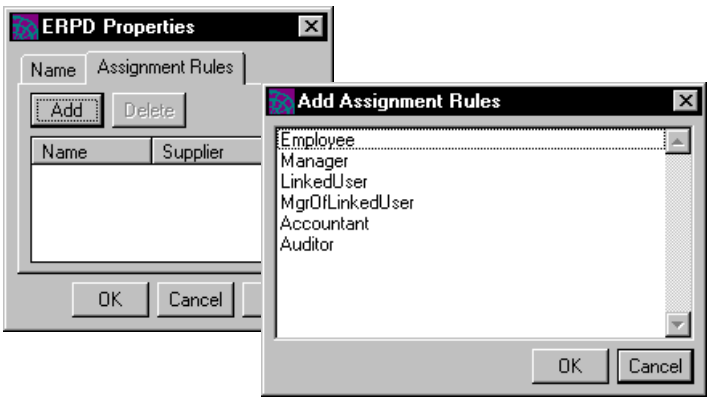


Figure 35 *Process Definition Add Assignment Rules Dialog*

To delete an assignment rule you added, choose the rule you want to delete in the Assignment Rules tab page, then click Delete.

Defining Process Attributes

Process attributes are variables of simple types that hold data important to the process. Each instance of a process has values assigned to each process attribute—sometimes at process creation time and sometimes as a result of process execution.

One of the main purposes of process attributes is for routing logic. They are used as process data in router methods, trigger methods, and possibly in assignment rules.

Primary process attribute

One special case is the primary process attribute. A primary process attribute is one that identifies the current process instance and possibly serves as a key value to associated database entries. For example, the InvoiceNum attribute of a Customer Order process

would identify each customer order by invoice number and allow the client application to look up the order in a database. The primary process attribute is the only process attribute the system administrator can see when monitoring processes.

An example of process attributes used for process logic is a process that handles employee expense reports that include an activity for approval of the report. If the expense total is under $500, the user performing the activity can be a Manager. If it is over $500, the user must be a Director. The process definition stores the claim amount in the ExpenseAmt process attribute. To determine the amount claimed in the expense report, the activity's assignment rule Evaluate method must be able to read the ExpenseAmt process attribute and test for the appropriate roles. (For information on the Evaluate method, see "Evaluate Method Example: Checking Process Attributes" on page 97.)

XmlData

A process attribute can be of type XmlData. Attributes of this type are typically used to hold XML data that can be supplied to applications that integrate with a Fusion Backbone. The XML data must be well formed, otherwise the Process Attribute property inspector rejects it. Fusion imposes no limits on the size of the XmlData. For more information on using XmlData, refer to *Forte Fusion Backbone Integration Guide* and the Forte Fusion Backbone online Help.

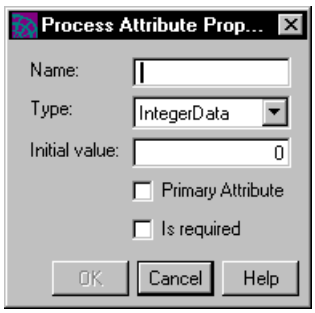
System attributes

In addition to process attributes that you define explicitly, there are four process attributes defined automatically by Fusion and assigned values when a process instance is first created. The system attributes represent the process id, process name, creation time, and creator, as listed in the following table. System attributes show up automatically in the Process Attributes list and can be used as you would other process attributes, except they cannot be deleted.

System Attribute	Description
_CreatedBy	The creator of the process
_CreationTime	Time the process was created
_Name	Name of the process
_ProcessID	Unique ID assigned by the Fusion Engine for the process

You work with process attributes in the Process Attributes list, displayed in the main workshop to the right of the layout area and below the Supplier Components list (see Figure 34 on page 133).

To add a process attribute, click the New button in the Process Attributes list to open the Process Attribute property inspector:



You can set the following properties for a process attribute:

Name Enter a name for the process attribute. Do not use an underscore ("_") as the first character of a process attribute name.

Type Choose a data type from the drop list. The allowed data types for process attributes are described in "Process Attribute Data Types" on page 189.

Initial value You can optionally choose an initial value for the process attribute, or you can leave it set to the default value displayed in this field. Primary process attributes would typically be set by the client application that creates new process instances.

Primary attribute Use this option to designate the attribute as a primary process attribute—an attribute that identifies the process instance, rather than one used for coding process logic. There can only be one primary process attribute.

Is required Use this option to indicate that a process attribute must be initialized. If this option is enabled, then process instance creation fails unless the creator provides an initial value for the attribute. This is critical for process definitions that might be referenced as subprocess activities in another process definition. The option also serves as a check on client applications that create process instances.

Deleting a process attribute

To delete a process attribute, select it in the Process Attributes list, then click Delete.

Changing a process attribute

To change the properties of a process attribute that is already defined, you can double-click it in the Process Attributes list or you can select it, then click the Properties button. Either choice displays the Process Attribute property inspector.

Working with Offered Activities

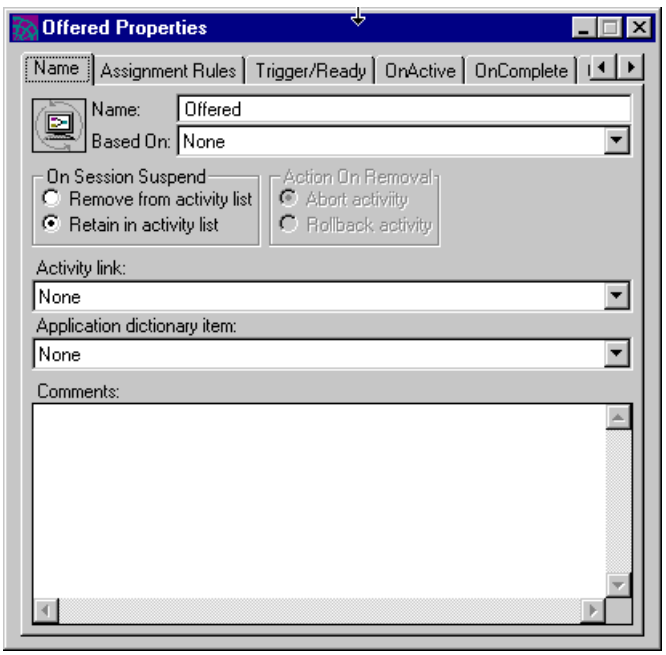
As described in "Offered and Queued Activities" on page 121, an offered activity is an activity that the engine offers to all users who have the roles and other qualifications to perform the activity, as determined by the activity's assignment rules.

To add an offered activity to the layout area, click its icon in the toolbar (or choose **Tools > Offered Activity**), then click the location in the layout area where you want to place the activity.

Note

You can also reuse an activity specified in another process definition by including the source process definition as a supplier to your current process definition. You then drag the activity from the Supplier Components list to the layout area. The workshop checks that the activity is consistent with your current process definition.

To see what properties you can set for an offered activity, double-click it to display its property inspector (or select it, then choose **Edit > Properties**):



As you can see from the property inspector, you can add and modify the following properties of an offered activity:

- name
- “based on” property
- session suspend action
- activity link
- application dictionary item
- comments
- assignment rules
- Trigger method
- Ready method
- OnActive method
- OnComplete method
- OnAbort method

See “Offered and Queued Activities” on page 121 for a description of offered activities that shows the relationships between methods, routers, and the states of the activity, and also shows how methods share sets of process attributes.

The rest of this section describes the activity’s properties and how to work with them.

Setting the “Based on” Property

This field indicates inheritance of like components from other process definitions that supply the current process definition.

Setting the Session Suspend Action

You can specify the action that the engine takes if the session that makes an offered activity ACTIVE is suspended. The session might be suspended because of a network or engine failure, or because a user or system manager explicitly suspended it. You can specify two kinds of action if the session is suspended:

- Remove from Activity List

  The activity is no longer ACTIVE, even if the session is re-established. You then specify whether the activity is aborted or rolled back to a READY state (so another user session can make it ACTIVE).

- Retain in Activity List

  The activity remains in the ACTIVE state, and is available for continued work when the session is re-established

Setting an Activity Link

You set the activity link on the Name tab page of the offered activity property inspector by selecting an activity in the Activity link drop list. The list shows all activities currently defined in the process definition. Once set, the name of the linked activity appears near the current activity in the layout area.

As described in "Activity Links" on page 120, an activity link specifies a relationship between the current activity and the user of another activity in the process definition. What the link itself does is to ensure that the name of the user that completed the other activity, plus other specified user profile information, is saved by the engine, and that the engine passes that information to the current activity's assignment rules.

You can link to only one other activity, and the activity must already exist in your process definition. You can only link to First, Last, offered, queued, and subprocess activities. In the case where the link is to a First activity the user name and related information is obtained from the creator of the process definition. The creator can be an individual user or, in the case of a subprocess, it can be a parent process. For information on the latter case, see "Setting the Subprocess Activity Link" on page 154.

To make the link meaningful, you must add at least one assignment rule to the current activity that makes use of the user name or the specified user profile information in its Evaluate method.

For example, if the current activity must be performed by the same user that performed a previous activity, choose the name of that activity from the drop list in the Activity Link field. Next, add an assignment rule whose Evaluate method compares its linkedUser parameter to the user name of its subject parameter. (See "Understanding the Evaluate Method" on page 95 for more information.)

Associating an Application Dictionary Item

You associate an application dictionary item with an offered activity by dragging the application dictionary item from the Supplier Components list and dropping it on the activity.

Alternatively, you can select the dictionary item from the drop list on the Name tab page of the offered activity property inspector. The drop list contains all the application dictionary items listed in the process definition's Supplier Components list. (For information on adding supplier components to a process definition, see "Adding Supplier Components" on page 135.)

When using elements in the Supplier Components list, the workshop checks for consistency. For example, when you associate an application dictionary item with an activity, any new process attributes referenced by the application dictionary item are added to the process definition's Process Attribute list. If the referenced attributes already exist in the attribute list, the data types are compared.

For a complete description of application dictionaries, see Chapter 6, "Defining Application Dictionaries."

Adding Comments

You can provide internal documentation about the activity in the Comments field of the Name tab page of the offered activity property inspector. This field is a good place to document how you expect the activity to be used and any unusual characteristics of the activity that are not immediately obvious. Information you include here can be useful to developers writing Fusion process client applications.

Associating Assignment Rules

You associate an assignment rule with an offered activity by dragging the assignment rule from the Supplier Components list and dropping it on the activity.

Alternatively, you can click the Assignment Rule tab in the offered activity property inspector to bring up the Assignment Rule tab page. To associate an assignment rule with an activity, click the Add button to display the Add Assignment Rules dialog shown in Figure 36 on page 143.

The dialog contains all the assignment rules listed in the process definition's Supplier Components list. You can associate any number of assignment rules with an activity. Before you can add them in the dialog, you must have included the assignment rule dictionary as a supplier to your process definition (its assignment rules must be in the Supplier Components list). Pick one or more assignment rules from the list, then click OK. (For information on adding supplier components to a process definition, see "Adding Supplier Components" on page 135.)

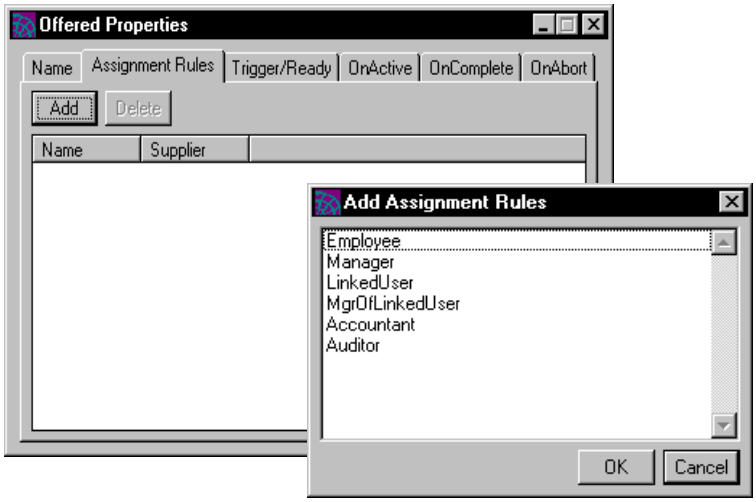


Figure 36 *Offered Activity Assignment Rules Tab Page*

When using elements in the Supplier Components list, the workshop checks for consistency. For example, when you associate an assignment rule with an activity, any new process attributes referenced by the assignment rule will be added to the process definition's Process Attribute list. If the referenced attributes already exist in the attribute list, the data types will be compared.

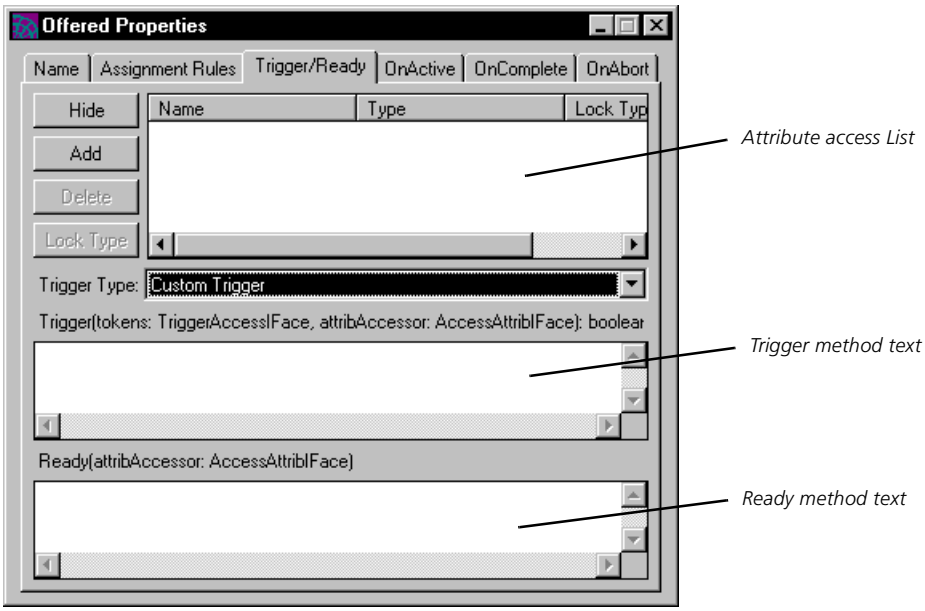
To delete an assignment rule, select it from the list in the Assignment Rule tab page, then click the Delete button.

For a complete description of assignment rule dictionaries, see Chapter 5, "Defining Assignment Rule Dictionaries."

Defining a Trigger Method

Click the Trigger/Ready tab to define a Trigger method.

When this tab page initially displays, it contains a Trigger method drop list, which offers several trigger method. If you select Custom Trigger from the list, a text field opens where you can write code for a custom trigger method. If you click the Attributes button, the attribute list displays.The following figure shows the Trigger/Ready tab:



Specifying the Attribute Access List

The attribute access list allows you to add, delete, and view the set of process attributes that can be accessed by both the Trigger and Ready methods. You can hide the attribute list by clicking the Hide button. Click the Attributes button, to display the list again.

Both the Trigger and the Ready method are listed on the same tab page to indicate that they share the same process attribute access list. You can designate one or more process attributes for your Trigger and Ready methods to access. The process attributes you designate here must already have been defined for the process definition, as described in “Defining Process Attributes” on page 138.

The procedure for specifying the attribute access list is common for all process definition activity methods and is described in “Specifying an Attribute Access List” on page 185.

Specifying the Trigger Type

An activity remains in a PENDING state until its Trigger method returns TRUE. At that time, the activity executes a Ready method, if any is defined, before placing the activity in a READY state.

The Trigger method can respond to a number of conditions:

- a router arriving from a predecessor activity
- a router arriving from a timer
- a process attribute changing value

Any of these conditions alone or in any combination can cause a Trigger method to return TRUE, depending on the logic in the Trigger method. If the Trigger method returns FALSE, the triggering conditions have not yet been met and the activity remains in the PENDING state.

You can write Trigger methods to implement quite complex triggering logic. However, the following two cases are the most common:

Trigger when any router arrives If more than one router points to the current activity, the first one to arrive causes the Trigger method return TRUE. This case, which is the default, handles a single incoming router pointing to an activity.

Trigger when all routers arrive If there is more than one incoming router pointing to an activity, all must arrive for the Trigger method to return TRUE. This case is a common one: All activities that are predecessors to this one must be COMPLETED (or ABORTED) before the current activity can be performed.

In these two standard cases, you do not have to write a trigger method—both these methods are already provided—you simply choose one from the Trigger Type drop list.

Writing a Custom Trigger Method

If your triggering logic is more complex than these two cases—for example, if it depends on process attribute values—you must write your own *custom* Trigger method. To write a custom **Trigger** method, choose Custom Trigger from the Trigger Type drop list. The method item field becomes enabled.

On the Trigger/Ready tab page shown in Figure 36 on page 143, you can see the declaration for the **Trigger** method. The method declaration is:

Trigger (**tokens=*TriggerAccessIface*, attribAccessor**=*AccessAttribIface*)

Returns boolean

Parameters	**Required?**	**Input**	**Output**
attribAccessor	●	●	

This method evaluates all the incoming routers according to criteria that you specify in the method. The method returns TRUE or FALSE, indicating whether the incoming router conditions match the criteria being checked by the method.

attribAccessor parameter

The **attribAccessor** parameter is an attribute accessor for the method's attribute access list specified on the Trigger/Ready tab page, described under "Specifying the Attribute Access List" on page 144. For information on how to use this parameter see "Working with Process Attributes" on page 188.

You can write custom Trigger methods using one of the two following virtual attributes to represent the number of times a router arrives from an activity or an expired timer:

- _Count*Activity_Name*
- _Count*Timer_Name*

Suppose you want to write a Trigger method that implements the following trigger condition: both Activity1 and Activity2 complete successfully, and in addition, Activity1 must complete three times (it has some kind of loop back router). The Trigger method code for this condition is the following:

```
Return ((_CountActivity1 = 3) and (_CountActivity2 = 1));
```

Note

The virtual attributes _Count*Activity_Name* and _Count*Timer_Name* can only be referenced by the activity or junction immediately following the activity or timer whose count is being referenced. Attempts to access these attributes further down the process definition results in a compiler error.

For general information on how to write process definition methods, see "Writing Code in Process Definition Methods" on page 182.

Defining a Ready Method

The **Ready** method is defined in the same Trigger/Ready tab page as the Trigger method. See the previous section "Defining a Trigger Method" on page 144 for more information about using this tab page and specifying the attribute access list.

The engine executes the **Ready** method when the activity's Trigger method returns TRUE and the activity is about to be placed in a READY state. You can use this method for internal housekeeping chores, like setting attribute values prior to their being used by assignment rules. For an overview of what this method is doing, see "Offered and Queued Activities" on page 121.

Note

The **Ready** method accesses the same attribute access list as the Trigger method (see "Defining a Trigger Method" on page 144).

The Ready method declaration is:

Ready (**attribAccessor**=*AccessAttriblFace*)

Returns none

Parameter	Required?	Input	Output
attribAccessor	●	●	

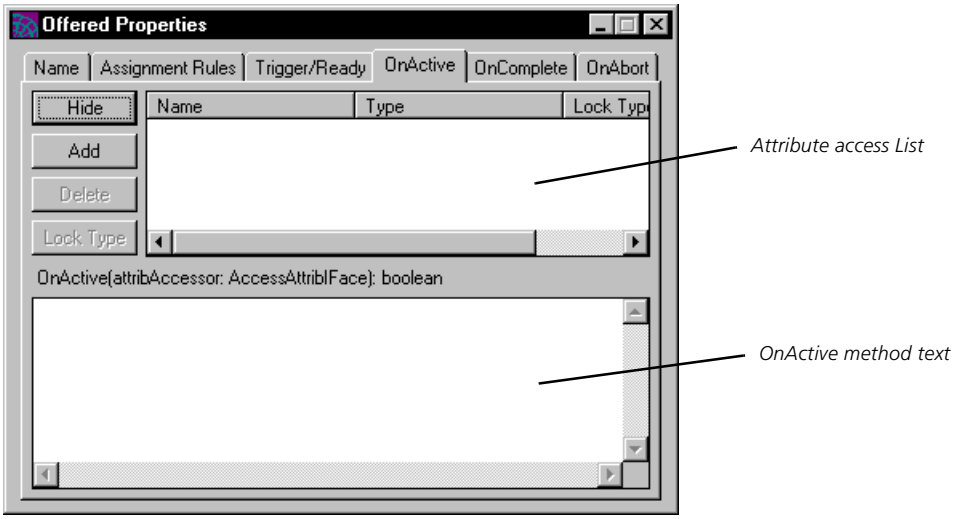
attribAccessor parameter

The **attribAccessor** parameter is an attribute accessor for the method's attribute access list specified on the Trigger/Ready tab page, described previously under "Specifying the Attribute Access List" on page 144. For information on how to use this parameter see "Working with Process Attributes" on page 188.

For general information on writing process definition methods, see "Writing Code in Process Definition Methods" on page 182.

Defining an OnActive Method

You define the **OnActive** method on the OnActive tab page:



The engine executes the **OnActive** method when an offered activity reaches an ACTIVE state, before the activity is performed by a user. You can use this method for internal housekeeping chores, such as setting attribute values prior to their being used by a client application.

As with all process definition methods, you have to set an attribute access list if you want to access process attributes from your **OnActive** method. The attribute access list is displayed at the top of the tab page, allowing you to add, delete, and view the set of process attributes that can be accessed by the **OnActive** method. You can hide the attribute list by clicking the

Hide button. If you do so, you see in its place an Attributes button, which you can click to display the list again. To create an attribute access list, see "Specifying an Attribute Access List" on page 185.

The **OnActive** method declaration is:

OnActive (**attribAccessor**=*AccessAttribIFace*)

Returns boolean

Parameter	Required?	Input	Output
attribAccessor	●	●	

The **OnActive** method returns a boolean, TRUE or FALSE. If it returns TRUE, then activity execution continues, and the activity is performed by a user. If the method returns FALSE, the activity is placed in an ABORTED state.

attribAccessor parameter

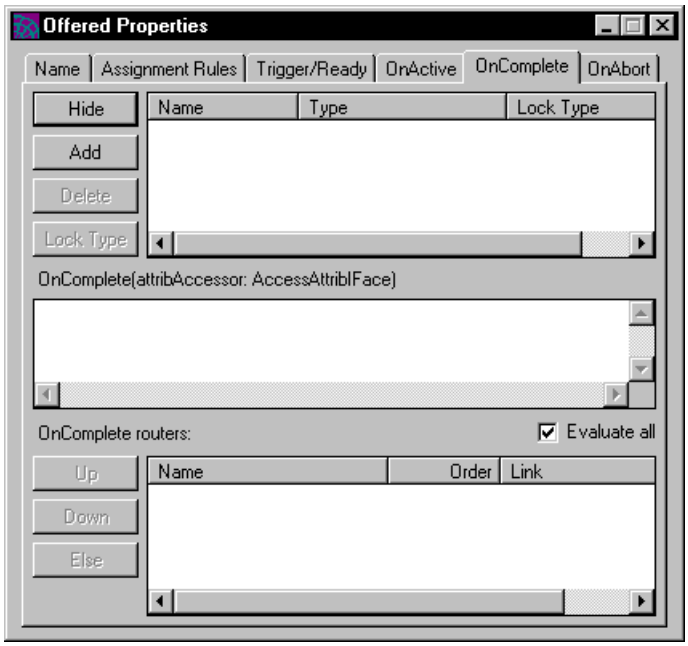
The **attribAccessor** parameter is an attribute accessor for the method's attribute access list specified on the OnActive tab page. For information on how to use this parameter see "Working with Process Attributes" on page 188.

For general information on how to write process definition methods, see "Writing Code in Process Definition Methods" on page 182.

Defining an OnComplete Method

The engine executes the **OnComplete** method when an offered activity reaches a COMPLETED state, after it is performed by a user. You can use this method for internal housekeeping chores, such as setting attribute values prior to OnComplete routing.

To specify the OnComplete attribute access list and define the **OnComplete** method choose the OnComplete tab page:



As with all process definition methods, you have to set an attribute access list if you want to access process attributes from your **OnComplete** method. The attribute access list is displayed at the top of the tab page, allowing you to add, delete, and view the set of process attributes that can be accessed by the OnComplete method. You can hide the attribute list by clicking the Hide button. If you do so, you see in its place an Attributes button, which you can click to display the list again. To create an attribute access list, see "Specifying an Attribute Access List" on page 185.

OnComplete *router* methods use the same attribute access list as the **OnComplete** method. If you change this attribute access list, it can affect all OnComplete router methods.

The **OnComplete** method declaration is:

OnComplete (**attribAccessor**=*AccessAttribIFace*)

Returns none

Parameter	Required?	Input	Output
attribAccessor	●	●	

attribAccessor parameter

The **attribAccessor** parameter is an attribute accessor for the method's attribute access list specified on the OnComplete tab page. For information on how to use this parameter see "Working with Process Attributes" on page 188.

For general information on how to write process definition methods, see "Writing Code in Process Definition Methods" on page 182.

Specifying OnComplete Router Execution

The OnComplete Routers list at the bottom of the tab page shows the routers that are activated when the activity reaches a COMPLETED state. (To specify the routers in the list, see "Working with Routers" on page 163.) Each router contains a router method that is executed after the OnComplete method executes. If the router method returns TRUE, the router transfers process control to its successor activity (the activity to which it points). If the router method returns FALSE, that router does not transfer control.

Ordering routers By default, any router you draw in the layout area from the current activity to another is an OnComplete router, and displayed in the OnComplete router list. Also by default, the order of execution of router methods is the order in which you draw the routers in the layout area. To change the order in which the router methods are executed, select a router and click the Up or Down buttons.

Else router In some cases, you might choose to have a router method executed *only if* all other router methods return FALSE. You can designate such a router as an *else* router by selecting it and clicking the Else button. There can only be one else router in a router list. By default, if there is only one router in the list, it is an else router. Like all routers, the else router can return TRUE or FALSE.

Evaluate all By default, the methods of all routers in the OnComplete router list are executed sequentially. However, you can specify that once any router method returns TRUE, subsequent routers in the list are ignored—their router methods are not executed. The first router to return TRUE transfers control. You specify this routing behavior by disabling the Evaluate All option.

Defining an OnAbort Method

The engine executes the **OnAbort** method when the activity is placed in an ABORTED state. You can use this method for internal housekeeping chores, like resetting attribute values prior to OnAbort routing.

To specify the OnAbort attribute access list and define the OnAbort method choose the OnAbort tab page, shown in Figure 37 on page 149.

As with all process definition methods, you have to set an attribute access list if you want to access process attributes from your OnAbort method. The attribute access list is displayed at the top of the tab page, allowing you to add, delete, and view the set of process attributes that can be accessed by the OnAbort method. You can hide the attribute list by clicking the

Hide button. If you do so, you see in its place an Attributes button, which you can click to display the list again. To create an attribute access list, see "Specifying an Attribute Access List" on page 185.

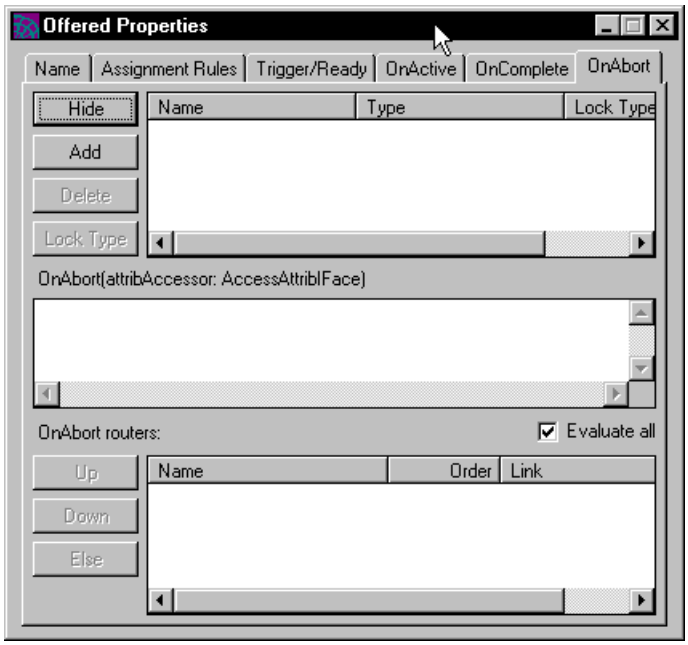


Figure 37 *Offered Activity OnAbort tab page*

OnAbort router methods use the same attribute access list as the **OnAbort** method. If you change this attribute access list, it can affect all OnAbort router methods.

The **OnAbort** method declaration is:

OnAbort (**attribAccessor**=*AccessAttribIFace*)

Returns none

Parameter	Required?	Input	Output
attribAccessor	●	●	

attribAccessor parameter

The **attribAccessor** parameter is an attribute accessor for the method's attribute access list specified on the OnAbort tab page. For information on how to use this parameter see "Working with Process Attributes" on page 188.

For general information on how to write process definition methods, see "Writing Code in Process Definition Methods" on page 182.

Specifying OnAbort Router Execution

The OnAbort Routers list at the bottom of the tab page shows all the routers that are activated when the activity reaches an ABORTED state. (To specify the routers in the list, see "Working with Routers" on page 163.) Each router contains a router method that is executed after the OnAbort method executes. If the router method returns TRUE, then the router transfers process control to its successive activity (the activity to which it points). If the router method returns FALSE, then that router does not transfer control.

OnAbort router execution is specified in the same way as OnComplete routers. See "Specifying OnComplete Router Execution" on page 148.

Working with Queued Activities

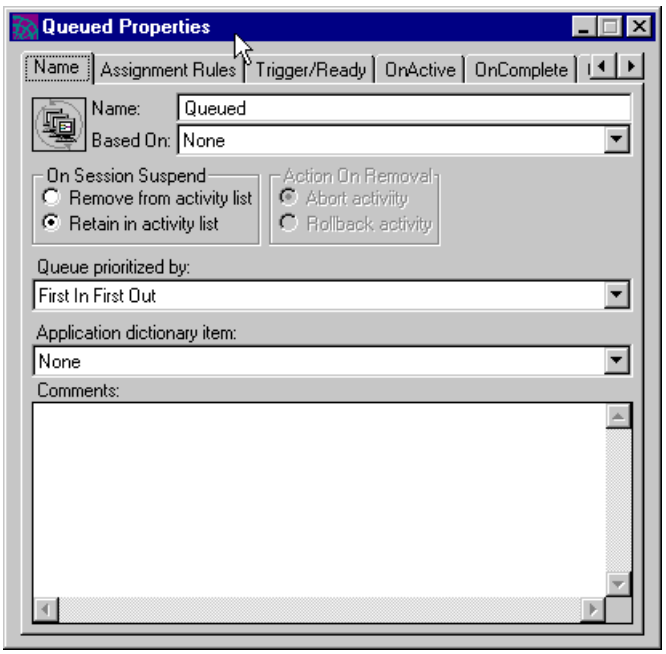
As described under "Offered and Queued Activities" on page 121, a queued activity is not offered to users, but is stored by the engine in a queue. Access to the queue is available to all users whose profiles match the activity's assignment rules.

From a process development point of view, the fact that a queued activity is put in a queue by the engine is the most significant difference between a queued activity and an offered activity. Queued activities are placed in a queue containing activities from many process instances. Ordering of the queue can depend on criteria based on individual process instances, however access to the queue cannot. This distinction has the following implications:

- Activities can be ordered in a queue based either on the order in which they are placed in the queue or on the value of a specified process attribute. In the latter case, as process attribute values change, activities are reordered in the queue.
- Assignment rules for queued activities cannot depend upon process attribute values, since users with access to a queue can perform all activities in the queue.
- Queued activities cannot be linked to other activities. The user who performed a previous activity in one instance is not the same as in other process instances, so an assignment rule could not use that kind of criteria. However, other activities in a process definition can be linked to a queued activity.

To add a queued activity to the layout area, click the queued activity icon in the toolbar (or choose **Tools > Queued Activity**), then click the location in the layout area where you want to place the activity.

To view the properties of a queued activity, double-click the activity in the layout area (or select it, then choose **Edit > Properties**) to open the property inspector:



From the property inspector, you can add and modify the following properties of a queued activity:

- name
- "based on" property
- session suspend action

- queue prioritizing process attribute
- application dictionary item
- comments
- assignment rules
- Trigger method
- Ready method
- OnActive method
- OnComplete method
- OnAbort method

Except for queue priority, these properties are the same as for offered activities and are discussed in "Working with Offered Activities" on page 140. Setting queue priority is discussed in the following section, "Setting Queue Priority."

Setting the "Based on" Property

This field indicates inheritance of like components from other process definitions that supply the current process definition.

Setting the Session Suspend Action

You can specify the action that the engine takes if the session that makes a queued activity ACTIVE is suspended. The session might be suspended because of a network or engine failure, or because a user or system manager explicitly suspended it. You can specify two kinds of action if the session is suspended:

- Remove from Activity List

  The activity is no longer ACTIVE, even if the session is re-established. You then specify whether the activity is aborted or rolled back to a READY state (so another user session can make it ACTIVE).

- Retain in Activity List

  The activity remains in the ACTIVE state, and is available for continued work when the session is re-established

Setting Queue Priority

You can control the way activities are prioritized (ordered) in a queue. By default, activities are added to the bottom of the queue as they are placed in the READY state. However, you can order the activities in a queue by using any IntegerData process attribute. Each activity in the queue is prioritized based on the value of its associated process attribute: the activity whose process attribute has the highest value becomes first in the list, and the one with the smallest value becomes last.

To set a prioritizing mechanism, choose a process attribute from the Queue prioritize attribute drop list. The list displays all IntegerData process attributes defined in the process definition, as well as the default priority, First In First Out.

If you select a process attribute to be used to prioritize the queue, the queue is reordered whenever an activity is placed in the queue and whenever the queue prioritizing attribute changes value for any activity in the queue.

Working with Subprocess Activities

As described in "Subprocess Activities" on page 122, a subprocess activity represents another process definition, which must be registered with some engine. The process definition a subprocess activity references is treated much like a method or subroutine: it is a regular process, but is instantiated by the subprocess activity.

To add a subprocess activity to the layout area, click the subprocess activity icon in the toolbar (or choose **Tools > Subprocess Activity**), then click the location in the layout area where you want to place the activity.

To see the properties of a subprocess activity, double-click the activity in the layout area (or select it, then choose **Edit > Properties**) to open the property inspector:

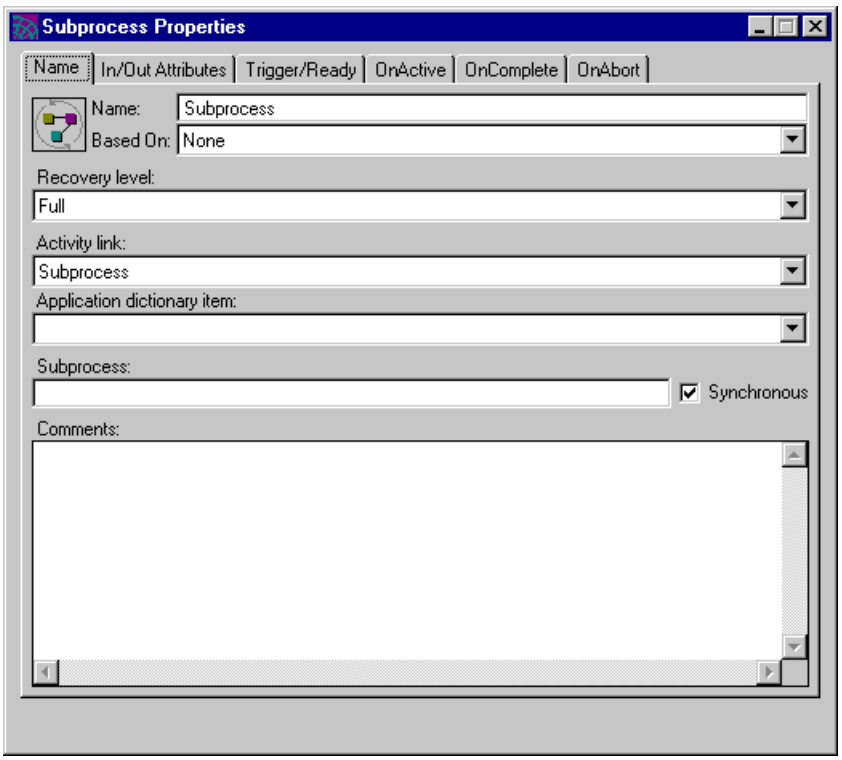


Figure 38 *Subprocess Activity Property Inspector*

From the subprocess activity property inspector, you can add and modify the following properties of a subprocess activity:

- name
- based on
- recovery level
- activity link
- application dictionary item
- subprocess name
- synchronous/asynchronous property
- comments
- input/output attributes
- Trigger method
- Ready method

- OnActive method
- OnComplete method
- OnAbort method

Because a subprocess activity does not interact directly with a client application (it creates an instance of another process), it does not have assignment rules or an application dictionary item associated with it. However, it does go through a READY state and has a Ready method, which allows you to set attributes and perform other necessary tasks before invoking the subprocess definition.

A subprocess activity can be synchronous (wait for the subprocess to complete) or asynchronous (continue to completion without waiting for the subprocess to complete).

A unique feature of a subprocess activity is its input and output attributes. Just as with method calls, input attributes are attributes passed from the subprocess activity (the parent process) to the subprocess and are used by the subprocess in doing its work. Output attributes are passed back from the subprocess to the subprocess activity (the parent process). These attributes must be defined both in the parent process definition and in the subprocess definition.

Note

If the subprocess activity is asynchronous, it does not need to wait for the subprocess to complete and does not need to receive values from the completed subprocess; therefore, it is not possible to set output attributes for an asynchronous subprocess activity.

The unique properties of subprocess activities are discussed in the following sections. For information about activity methods (Trigger, Ready, OnActive, OnComplete, and OnAbort) see "Working with Offered Activities" on page 140.

Specifying the Subprocess

To specify the name of the subprocess represented by a subprocess activity and indicate whether the subprocess activity is synchronous or asynchronous, double-click the subprocess activity to display its property inspector, and, if necessary, click the Name tab to display the Name tab page, shown in Figure 38 on page 152.

Synchronous/asynchronous property

To indicate whether a subprocess activity is synchronous or asynchronous, use the Synchronous option. The two choices are:

Synchronous By default, a subprocess executes synchronously. The subprocess activity cannot continue to a COMPLETED state unless the subprocess goes to completion. Thus, the parent process is dependent on successful completion of the subprocess. If the subprocess cannot be created or fails to complete normally for any reason, the subprocess activity is placed in an ABORTED state.

Asynchronous The subprocess activity continues to a COMPLETED state without waiting for the subprocess to complete. Thus the parent process continues to execute independently of the subprocess. In this context, output attributes for the subprocess activity have no meaning. If the subprocess is not successfully created, the subprocess activity is placed in an ABORTED state.

Subprocess name

In the Subprocess field, enter the name of the process to be called. By default, it is assumed that the subprocess is executed on the same engine as the parent process, however this is not always the case. Your system manager can indicate the engine on which the subprocess definition is executed by registering an alias with the engine of the parent process.

Setting the Subprocess Activity Link

Activity links for subprocess activities work basically the same as for offered activities (see "Setting an Activity Link" on page 142) with a few additional subtleties.

The activity link you specify for a subprocess activity does not apply *directly* to the subprocess activity, because a subprocess activity is not performed by a single user. Instead the user who completed the linked activity is considered to be the *creator* of the subprocess, meaning the user associated with the First activity of the subprocess.

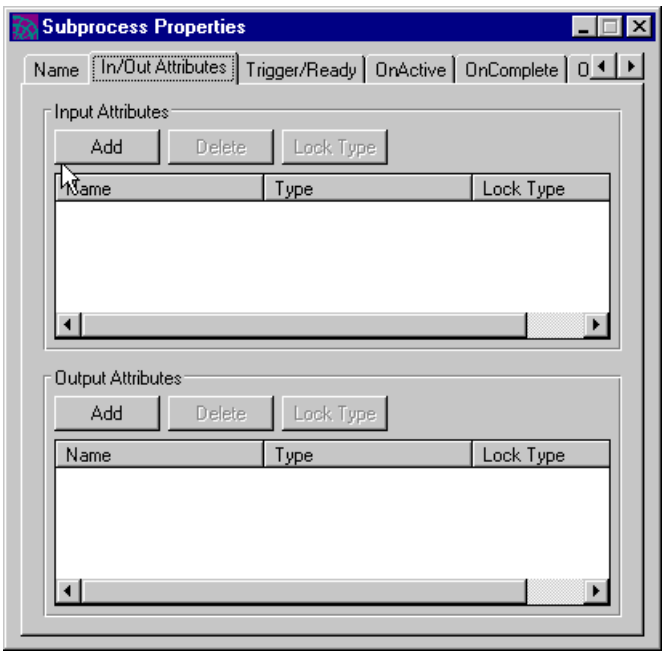
Any activity in the subprocess definition that links to the First activity gets this linked user. If the subprocess activity lacks an activity link, then the First activity in the subprocess has no user information—no activity in the subprocess definition should link to it.

As a further subtlety, if any activity in the parent process definition has an activity link to the subprocess activity, the linked user for the subprocess activity is the "user" of the Last activity in the subprocess definition. Since there is no direct user associated with a Last activity, this information is only available if the Last activity is itself linked to some other activity in the subprocess.

Specifying Input and Output Attributes

You can define attributes that pass values into the subprocess and attributes that pass values back out. Input attributes are similar to a method's Input parameters: they pass values into the subprocess to be used there. Output attributes are similar to a method's output parameters: they pass values back out of the subprocess to be used in the parent process. Input and output attributes must exist as process attributes both in the parent process (the one with the subprocess activity) and the subprocess.

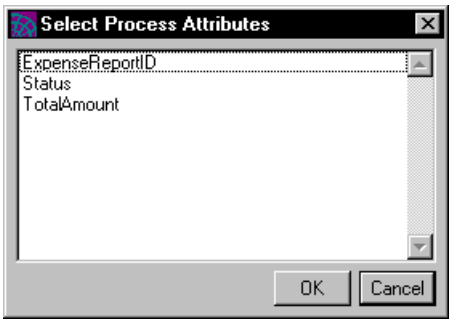
To set input and output attributes, click the In/Out attributes tab in the property inspector to open the In/Out tab page:



To add process attributes to the Input or Output attribute list:

1 Click **Add**.

The Select Process Attributes dialog appears. The dialog displays the list of process attributes defined in the process definition:



2 Choose one or more process attributes.

3 Click **OK** to add them to the attribute access list.

Note Input attributes automatically have No_lock lock type. Output attributes only have meaning and can only be specified for synchronous subprocess activities; they automatically have Write lock type.

To delete an attribute from the list:

1 Select the attribute from the list

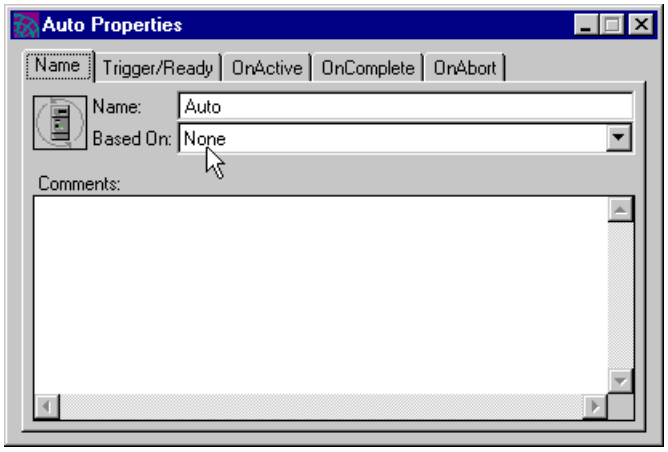
2 Click **Delete**.

Working with Automatic Activities

As described under "Automatic Activities" on page 123, an automatic activity is performed by the engine, not by a user of a process client application.

To add an automatic activity to the layout area, click its icon in the toolbar (or choose **Tools > Automatic Activity**), then click the location in the layout area where you want to place the activity.

To see the properties of an automatic activity, open the property inspector by double-clicking the activity in the layout area (or select the activity, then choose **Edit > Properties**):



From the property inspector, you can add and modify the following properties of an automatic activity:

- name
- "based on" property
- comments
- Trigger method
- Ready method
- OnActive method
- OnComplete method
- OnAbort method

An automatic activity specifically does *not* have assignment rules, an activity link, or an application dictionary item associated with it.

The most important aspect of an automatic activity is its *OnActive* method. This method, performed by the engine, represents work that can be performed automatically rather than by a user of a client activity. In general, this work should be simple (from the engine's point of view) so it does not impact process execution performance. As a general rule, you write this method to invoke some service external to the engine.

Often, fully automated work can be satisfactorily performed using either an automatic activity or by creating a service that opens a session with a Fusion engine to perform the automated work, much like a client application user. In the latter case, the tasks performing the activity execute outside and in parallel with the engine, while in the former case, they are executed by the engine itself.

The mechanics of writing an OnActive method for an automatic activity are the same as for an offered activity and are described in "Defining an OnActive Method" on page 146. The only significant distinction for an automatic activity's OnActive method is that when it returns TRUE, the activity is placed directly in a COMPLETED state.

Typically, you code your OnActive method to invoke a Forte service object. For more information, refer to "Writing Code that Accesses Forte Service Objects" on page 192.

For information about other automatic activity methods (Trigger, Ready, OnComplete, and OnAbort) refer to "Working with Offered Activities" on page 140.

Working with Timers

Timers are introduced in "About Timers" on page 126. To summarize, there are two types of timers: *Elapsed* (works like a kitchen timer, with a duration of time whose expiration depends on when the timer starts) and *Deadline* (works like a deadline in a schedule, with a set date and time that it expires). Each type of timer has its own default behavior, which can be modified by changing its properties and rewriting its associated methods. Each type of timer also has its own set of methods.

As you can with an activity, you can use a router to connect a timer to an activity, and you can have multiple routers connecting to multiple activities. Also, as with an activity, the attribute access list for a timer's router methods is shared with the OnExpiration method of the timer.

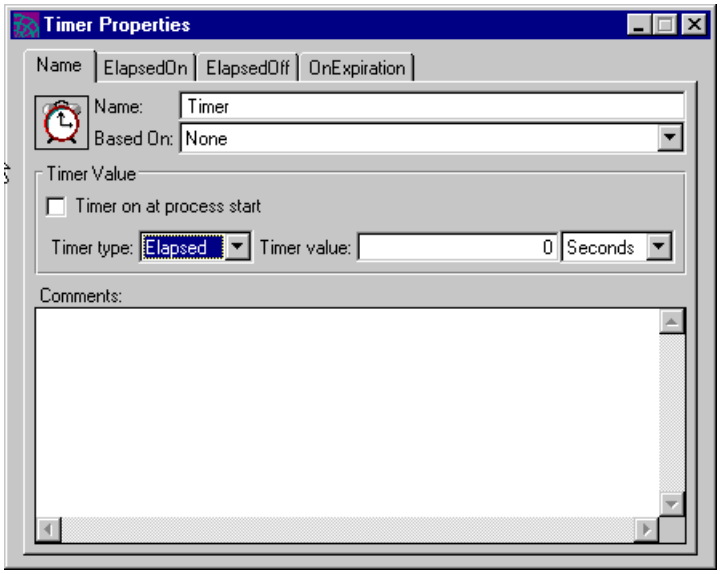
See "Working with Timer Controls" on page 162 for information on changes in activity states can be used to control a timer.

To add a timer to the layout area, click its icon in the toolbar (or choose **Tools** > **Timer**), then click the location in the layout area where you want to place the timer.

Working with an Elapsed Timer

When you first create an elapsed timer its default duration (timer value) is set to zero and it's default initial state (when a process instance starts) is off.

To set the timer's properties, double-click the timer to display its property inspector:



The property inspector has tabs for setting the name of the timer and other properties, and tabs for defining its ElapsedOn, ElapsedOff, and OnExpiration methods. The OnExpiration tab page is where timer expiration routers are specified.

Specifying Timer Properties

In the Name tab page of an elapsed timer's property inspector, you can set the following properties:

Name Enter a name for the timer.

Based on Indicates inheritance of like components from other process definitions that supply the current process definition.

Timer On at Process Start Lets you set the initial state of the timer when the process instance is first created (started up). By default the initial state is OFF.

Timer Type Choose between Elapsed or Deadline. Set this value early because it determines the other properties you can set.

Timer Value and Units You can set an elapsed timer for any duration of time. The value has meaning only when you specify the units, in the drop list. Note, however, that setting units to milliseconds might not work on all platforms. The practical minimum unit is seconds.

Comments Enter an internal comment that describes the timer. Comments can be useful to developers writing process client applications.

Defining the ElapsedOn Method

The ElapsedOn method is executed when the timer is turned on. By default, it calculates the expiration time of the timer by adding the block of time indicated in the timer's Timer Value setting (passed to it in the onTime parameter) to the current date and time indicated by the computer system, yielding an expiration time.

You have the option of specifying more complex behavior using the **ElapsedOn** and **ElapsedOff** methods, such as calculating by business days, skipping two days for each weekend. For example, a business day timer that is set for three business days might be started on a Thursday at 9 am. Its **ElapsedOn** method calculates its expiration time as the same time on the following Tuesday. If it is subsequently stopped on Friday at 9 am, the timer's **ElapsedOff** method indicates that two business days are left on the timer. If it is then started up on Monday at 9 am, the timer's ElapsedOn method calculates the expiration time as Wednesday at 9 am.

To specify the ElapsedOn attribute access list and define the **ElapsedOn** method choose the ElapsedOn tab page.

As with all process definition methods, you have to set an attribute access list if you want to access process attributes from your **ElapsedOn** method. The attribute access list is displayed at the top of the tab page, allowing you to add, delete, and view the set of process attributes that can be accessed by the **ElapsedOn** method. You can hide the attribute list by clicking the Hide button. If you do so, you see in its place an Attributes button, which you can click to display the list again. To create an attribute access list, see "Specifying an Attribute Access List" on page 185.

The **ElapsedOn** method declaration is:

ElapsedOn (**onTime**=*IntervalData*, **attribAccessor**=*AccessAttriblFace*)

Returns DateTimeData

Parameter	Required?	Input	Output
onTime	●	●	
attribAccessor	●	●	

onTime parameter

The **onTime** parameter is the internally known time period until expiration. It is passed into the method, and the method manipulates it as necessary to produce an expiration time as the return value. Initially the value is taken from the Timer value field in the property inspector; however on subsequent restarts, it is supplied by the engine (from the return value of the ElapsedOff method).

attribAccessor parameter

The **attribAccessor** parameter is an attribute accessor for the method's attribute access list specified on the ElapsedOn tab page. For information on how to use this parameter see "Working with Process Attributes" on page 188.

For general information on how to write process definition methods, see "Writing Code in Process Definition Methods" on page 182.

Defining the ElapsedOff Method

The **ElapsedOff** method is executed when the timer is turned off. By default, it calculates how much time is left from the current date and time to the expiration time of the timer (as calculated by the ElapsedOn method).

To specify the ElapsedOff attribute access list and define the **ElapsedOff** method choose the ElapsedOff tab page.

As with all process definition methods, you have to set an attribute access list if you want to access process attributes from your **ElapsedOff** method. The attribute access list is displayed at the top of the tab page, allowing you to add, delete, and view the set of process attributes that can be accessed by the ElapsedOff method. You can hide the attribute list by clicking the Hide button. If you do so, you see in its place an Attributes button, which you can click to display the list again. To create an attribute access list, see "Specifying an Attribute Access List" on page 185.

The **ElapsedOff** method declaration is:

ElapsedOff (**offTime**=*DateTimeData*, **attribAccessor**=*AccessAttriblFace*)			
Returns IntervalData			
Parameter	**Required?**	**Input**	**Output**
offTime	●	●	
attribAccessor	●	●	

offTime parameter

The **offTime** parameter is the internally known expiration time of the elapsed timer (as returned by the ElapsedOn method). It is passed into the method, and the method manipulates it as necessary to produce the time interval until expiration.

attribAccessor parameter

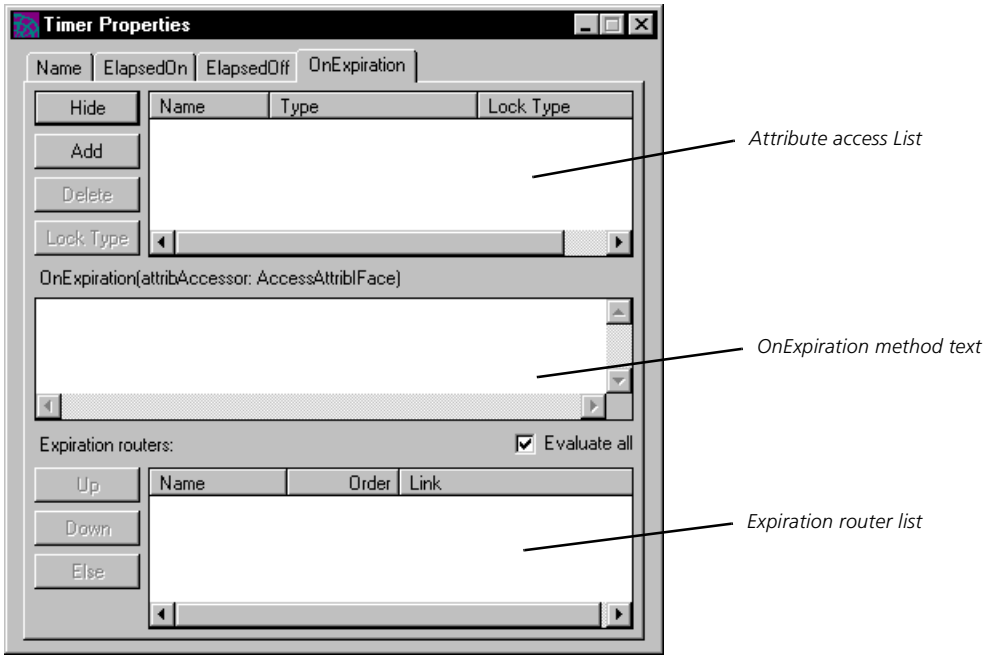
The **attribAccessor** parameter is an attribute accessor for the method's attribute access list specified on the specified on the ElapsedOff tab page. For information on how to use this parameter see "Working with Process Attributes" on page 188.

For general information on how to write process definition methods, see "Writing Code in Process Definition Methods" on page 182.

Defining the OnExpiration Method

The engine executes the **OnExpiration** method when the timer expires. You can use this method for internal housekeeping chores, like setting attribute values prior to Expiration routing.

To specify the OnExpiration attribute access list and define the **OnExpiration** method choose the OnExpiration tab page.



As with all process definition methods, you have to set an attribute access list if you want to access process attributes from your **OnExpiration** method. The attribute access list is displayed at the top of the tab page, allowing you to add, delete, and view the set of process attributes that can be accessed by the **OnExpiration** method. You can hide the attribute list by clicking the Hide button. If you do so, you see in its place an Attributes button, which you can click to display the list again. To create an attribute access list, see the procedure in "Specifying an Attribute Access List" on page 185.

Timer expiration router methods use the same attribute access list as the **OnExpiration** method. If you change this attribute access list, it can affect all Expiration router methods.

The **OnExpiration** method declaration is:

OnExpiration (**attribAccessor**=*AccessAttribIFace*)			
Returns none			
Parameter	**Required?**	**Input**	**Output**
attribAccessor	●	●	

attribAccessor parameter

The **attribAccessor** parameter is an attribute accessor for the method's attribute access list specified on the OnExpiration tab page. For information on how to use this parameter see "Working with Process Attributes" on page 188.

For general information on how to write process definition methods, see "Writing Code in Process Definition Methods" on page 182.

Specifying Expiration Router Execution

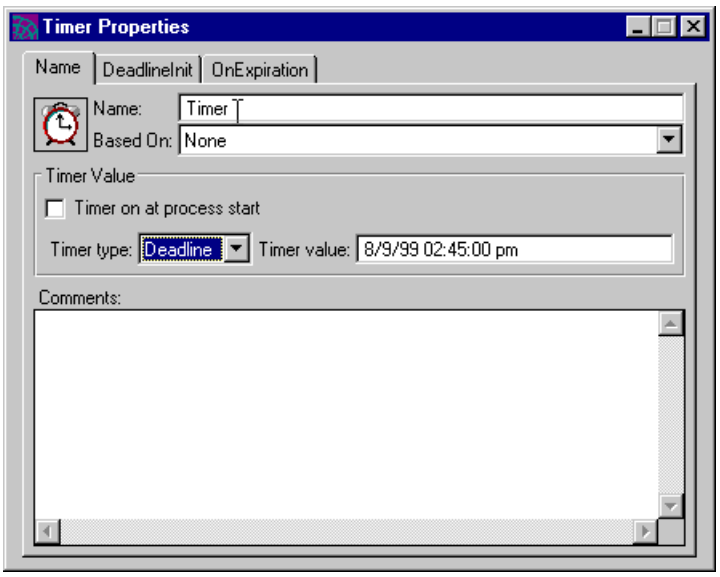
The Expiration Routers list at the bottom of the tab page shows all the routers that are activated when the timer expires. (To specify the routers in the list, see "Working with Routers" on page 163.) Each router contains a router method that is executed after the OnExpiration method executes. If the router method returns TRUE, then the router transfers process control to its successive activity (the activity it points to). If the router method returns FALSE, then that router does not transfer control.

Expiration routers are specified in the same way as OnComplete routers. See "Specifying OnComplete Router Execution" on page 148.

Working with a Deadline Timer

When you first create an elapsed timer its default expiration time (timer value) is set to the current date and time and it's default initial state (when a process instance starts) is off.

To set the timer's properties, double-click the timer to display its property inspector:



The property inspector has tabs for setting the name of the timer and other properties, and tabs for defining its DeadlineInit and OnExpiration methods. The OnExpiration tab page is where timer expiration routers are specified.

Specifying Timer Properties

In an elapsed timer's Name tab page, you can set the following properties:

Name Enter a name for the timer.

Timer On at Process Start Sets the initial state of the timer when the process instance is first created (started up). By default the initial state is OFF.

Timer Type Select Elapsed or Deadline. Set this value early because it determines the other properties you can set.

Timer Value Enter the date and time the timer expires. The format required in this field is the MM/DD/YY hh:mm format—Month/Day/Year Hours:Minutes, with the year not showing the century and the hours using a 24-hour clock. If you have not entered a value in this field, you see the current date and time.

Comments Enter an internal comment that describes the timer. Comments can provide useful information to developers writing process client applications.

Defining the DeadlineInit Method

The **DeadlineInit** method calculates an expiration date and time whenever the deadline timer is turned on. By default, it returns the date and time entered in the Timer value field of the property inspector. As discussed in "About Timers" on page 126, a deadline timer can be defined to handle custom deadlines such as the end of a business quarter. This calculation would be performed by the DeadlineInit method, which would round up an initial date to the end of a business quarter.

To specify the DeadlineInit attribute access list and define the **DeadlineInit** method, choose the DeadlineInit tab page.

As with all process definition methods, you have to set an attribute access list if you want to access process attributes from your **DeadlineInit** method. The attribute access list is displayed at the top of the tab page, allowing you to add, delete, and view the set of process attributes that can be accessed by the **DeadlineInit** method. You can hide the attribute list by clicking the Hide button. If you do so, you see in its place an Attributes button, which you can click to display the list again. To create an attribute access list, see "Specifying an Attribute Access List" on page 185.

The **DeadlineInit** method declaration is:

DeadlineInit (**initTime=***DateTimeData*, **attribAccessor**=*AccessAttribIFace*)

Returns DateTimeData

Parameter	Required?	Input	Output
initTime	●	●	
attribAccessor	●	●	

initTime parameter

The **initTime** parameter is the initial deadline time—set in the Timer value field of the property inspector. It is passed into the method, and the method manipulates it if necessary to produce an actual expiration time as the return value.

attribAccessor parameter

The **attribAccessor** parameter is an attribute accessor for the method's attribute access list defined in the DeadlineInit tab page. For information on how to use this parameter see "Working with Process Attributes" on page 188.

For general information on how to write process definition methods, see "Writing Code in Process Definition Methods" on page 182.

Defining the OnExpiration Method

The procedure for defining the OnExpiration method for a deadline timer is the same as for an elapsed timer. See "Defining the OnExpiration Method" on page 159.

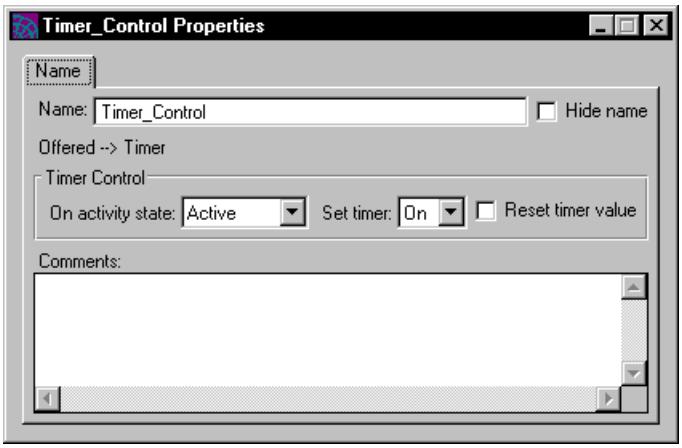
Specifying Expiration Router Execution

The procedure for specifying the Expiration routers for a deadline timer is the same as for an elapsed timer. See "Specifying Expiration Router Execution" on page 160.

Working with Timer Controls

To have an activity control the behavior of a timer, click the Connect tool in the tool palette, then in the layout area click in the originating activity and drag to the destination timer, dropping it there. This creates a timer control.

To set the properties of a timer control, double-click the timer control to display its property inspector:



In the Timer Control property inspector, you can enter a name for the timer control, provide internal, descriptive comments about the timer, and set its activity state and its effect on the timer.

On Activity State A timer control is connected to a specific state of the originating activity. By default, the timer control becomes active (affects the timer) when the activity becomes ACTIVE. However, you can designate that the timer control become active in any state of the activity, and you can have multiple timer controls from an activity to a timer, making it possible, for example, to start a timer when the activity becomes READY and stop it when the activity completes (reaches COMPLETED state). See "Activity States" on page 119 for more information on activity states.

SetTimer This setting designates whether the timer is automatically set ON or OFF when the timer control activates.

Reset Timer Value A timer control can reset the timer when it activates, or it can leave it at its current value. The effect of a reset depends on the type of timer:

- Elapsed timer—resetting causes it to recalculate the expiration time (ElapsedOn method). For example, assuming a default ElapsedOn method, if the Time Value is ten minutes and the timer has run for five, a reset causes it to expire in ten minutes again. If you have redefined its ElapsedOn method, resetting it affects its behavior as you define in this method.
- Deadline timer—resetting causes it to recalculate the expiration time (DeadlineInit method). For example, assuming a default DeadlineInit method, the reset causes the deadline time to remain the same. If you have defined your own DeadlineInit method for a deadline timer, resetting the timer might cause the deadline to change, depending on your definition of this method.

See "Working with Timers" on page 156 for more information on timer methods.

Working with Routers

To route process control from one activity to another or from a timer to an activity, click the Connect tool in the tool palette, then in the layout area click and drag from the originating object to the destination object. This creates a Router.

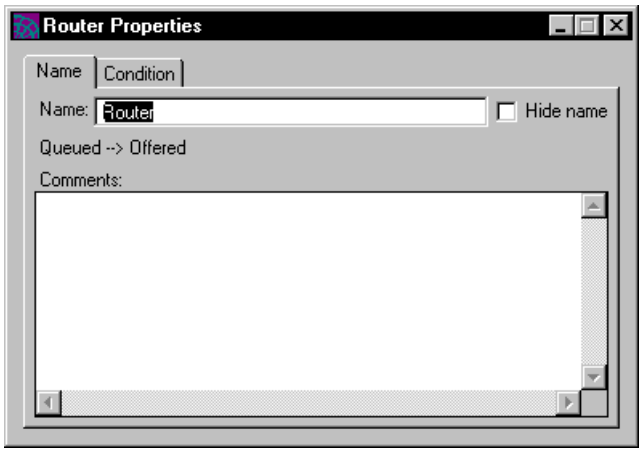
There are three types of routers:

- OnComplete—routers that are activated when an activity completes successfully (reaches a COMPLETED state)
- OnAbort—routers that are activated when an activity fails to complete successfully (reaches an ABORTED state)
- Expiration—routers that are activated when a timer expires

When you draw a router from one activity to another, it is by default an OnComplete router. The router is automatically added to the activity's OnComplete router list. You can turn an OnComplete router into an OnAbort router by displaying the router's property inspector and, in the Condition tab page, choosing OnAbort from the Router Type drop list (unless the router originates from a First activity).

When you draw a router from a timer to an activity, it is always an Expiration router. The router is automatically added to the timer's Expiration router list.

To display a router's property inspector, double-click the router (or select the router and choose **Edit > Properties**).



Specifying Router Properties

In the router's Name tab page, you can set the following properties:

Name Enter a name for the router.

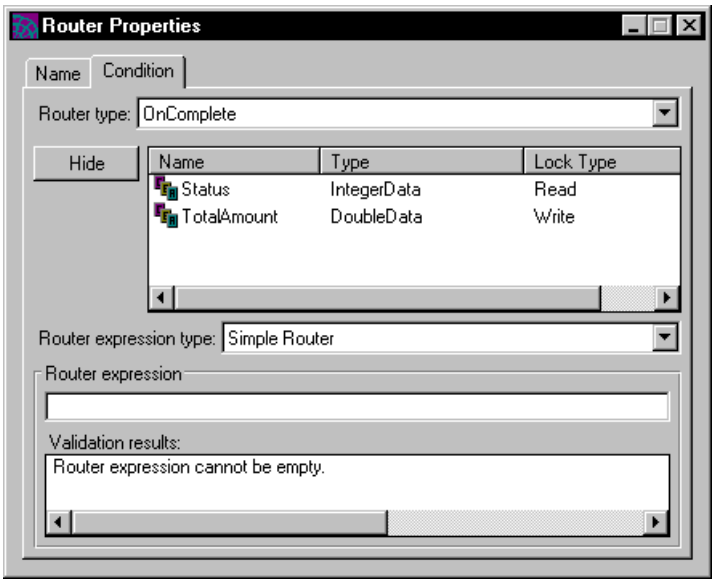
Hide name Choose to not display the router name in the layout area by enabling the Hide name option. Router names can sometimes clutter the layout area.

Comments Enter an internal comment that describes the router. Internal comments can be useful to developers writing process client applications.

Above the Comments field there is a statement indicating the router's origin and its destination.

Defining Router Methods

In the router property inspector, click the Condition tab to display the Condition tab page:



In this tab page, you can set the router type, OnComplete or OnAbort. You can also view the attributes access list for the **Router** method, and can write a **Router** method.

The router method by default returns TRUE, meaning that by default the router always transfers control to its destination activity. If you want to evaluate criteria for the router, such as testing a process attribute to see if its value is appropriate for transferring control to the destination activity, you can write a method of either a simple router type or a custom router type, as described in the following sections.

As with all process definition methods, you have to set an attribute access list if you want to access process attributes from your Router method. However, Router methods use the attribute access lists of their corresponding parent methods:

- If the router is an OnComplete router, its attribute access list is set in the originating activity's OnComplete tab page (see "Defining an OnComplete Method" on page 147).
- If the router is an OnAbort router, its attribute access list is set in the originating activity's OnAbort tab page (see "Defining an OnAbort Method" on page 148).
- If the router is an Expiration router, its attribute access list is set in the originating timer's OnExpiration tab page (see "Defining the OnExpiration Method" on page 159).

You can only see the attributes, not edit them in a router's Condition tab page. To change the attribute access list you need to change them in the corresponding OnComplete, OnAbort, or OnExpiration tab pages.

You can hide the attribute list by clicking the Hide button. If you do so, you see in its place an Attributes button, which you can click to display the list again.

Simple Router Type Methods

You can construct router methods that express the desired router conditions using a method generator that does not require knowledge of TOOL language syntax. To use this method generator, choose Simple Router from the Router expression type drop list.

In the Router expression field you can now enter a simple algebraic expression—using process attributes in the router method's attribute access list and standard algebraic operators—that describes the condition under which the router will transfer control to its destination activity. As you type the expression, the Validation results field will dynamically change to reflect the status of your expression, reporting errors in syntax.

A simple router expression would be the following:

```
Status = 3 and TotalAmount > 1000
```

Custom Router Type Methods

You can write router methods that express the desired router conditions using TOOL code, by choosing Custom Router from the Router expression type drop list. The **Router** method declaration is:

Router (**attribAccessor**=*AccessAttribIFace*)

Returns boolean

Parameter	Required?	Input	Output
attribAccessor	●	●	

attribAccessor parameter

The **attribAccessor** parameter is an attribute accessor for the method's attribute access list shown on the Condition tab page. For information on how to use this parameter see "Working with Process Attributes" on page 188.

The router method corresponding to the simple expression illustrated in "Simple Router Type Methods" would be the following:

```
return (Status = 3 and TotalAmount > 1000);
```

Of course, you can write considerably more complicated router methods as needed. For general information on how to write process definition methods, see "Writing Code in Process Definition Methods" on page 182.

Saving and Compiling Process Definitions

As you work on your process definitions, it is a good idea to save them regularly. As you write your own activity, router, or timer methods, you can periodically compile your process definition plan into a TOOL project to ensure that the syntax is correct.

Saving Changes

Save All command

As you edit the process definition, be sure to save your changes periodically (choose **File > Save All**). When you save changes, the current process definition is updated in your workspace.

Note

If you have any other workshops open for editing, they are saved at the same time.

Compiling a Process Definition

Compile command

If you write your activity, router, or timer methods, you might want to compile the process definition each time you finish a method to ensure that your code is syntactically correct. To compile, choose **File > Compile**. Fusion generates TOOL code from the process definition and compiles it, saving the resulting TOOL code in a read-only TOOL project that has the extension _PD. (This file is a by-product of the compile process: you do not use it.)

If there are compilation errors, Fusion displays them for you. You can find the source of compilation errors by double-clicking on a compile error item. The corresponding method text item field will be displayed with the code that contains the error highlighted. You can fix the errors in your TOOL code, and recompile.

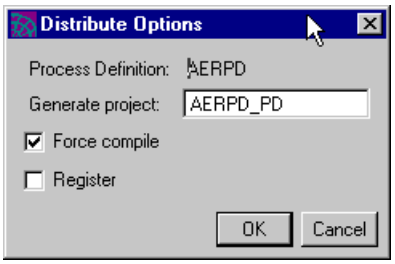
Hint If too many of these generated files clutter your list of plans in the Repository Workshop, you can filter them by choosing Fusion Plans from the Filter drop list. (Figure 33 on page 132 shows the Repository Workshop with the filter set this way.)

Making and Registering Process Definition Library Distributions

Finally, when you have completed all work on a process definition and are ready for it to be used by an engine, you make it into a library distribution and register it with one or more engines.

Distribute command

To perform these operations, choose **File > Distribute**. You see the Distribute Options dialog box:



Force Compile option

The **File > Distribute** command performs a compile operation if this option is checked, then uses the resulting TOOL project to make a library distribution. The Generate Project field shows you the name of the generated TOOL project. You can enter another name if you like.

Register option

To register the resulting library distribution with an engine, click the Register option. If the Register option is enabled, you are prompted with a list of engines. Choose the engines you want to register with, then click **OK**. The library distribution is saved in the FORTE_ROOT/appdist directory on the central server node in your Fusion system.

Note The node hosting a Fusion process engine must be online and the engine running in your environment before you can perform a registration with that engine.

If an engine you want to register the process definition with is not in your environment, you must copy the library distribution to the remote environment, then use the Fusion Console to register the distribution. See the *Forte Fusion Process Management System Guide* for more information.

Registering a New Version of a Process Definition

When you register a new version of a process definition with an engine that has an older version already registered, the new version of the process definition is the one that is used as users create new process instances. If there are any instances of the old process definition executing, they continue to execute until they are done.

Chapter 8

Defining Validations

This chapter describes how to use the Validation Workshop to write the methods that validate users and sessions against the site's organization database. The most important of these methods is ValidateUser, which authenticates a user at login time and then constructs a user profile object for that user.

The chapter covers the following topics:

- descriptions of user and session validations
- using the Validation Workshop
- creating new versions of a validation
- class reference for the Validation class

About Validations

A validation is a method invoked by a Fusion engine to authenticate an operation based on information in a site's organization database. The validation methods consist of two general types: user validation and session validation.

User validation Verifies a user's request to open an engine session. The engine invokes a ValidateUser method, which checks a user profile object and login password against information in the organization database. The method can also add information it finds in the database to the user profile object. When the engine has fully authenticated a user profile object, it can pass it to assignment rules that determine the activities (and possibly processes) a user can access. For an overview of user profiles and user validations and how they fit into a Fusion enterprise application, see "User Profile Design Concepts" on page 40.

Session validation Implement a site's session control policy, which generally regulates the number of concurrent sessions a user can open. Two methods, SessionOpen and SessionClose, are invoked to write session control information into the organization database and use that information to decide when a session can be opened.

The user validation and session validation work in tandem. The Fusion engine first performs the user validation, then the session validation. Both user validation *and* the session control criteria must be satisfied for a new session to be opened, or for a suspended session to be restored to an ACTIVE state. The authenticated user profile object is then associated with the ACTIVE session.

The validation methods access the site-defined organization database that holds all employee information pertinent to performing work in an enterprise application. They retrieve or write information for a specific user.

Validation class

The Validation class defines a number of methods that are executed by the Fusion engine. You write these methods to gain access to your organization database, verify a user profile object against that database, implement a session control policy, and exit the database. Because these methods access your organization database and perform actions that depend on site policies, you should define these methods yourself. The methods have default implementations that are used if you do not provide your own. For more information on these methods, refer to "Validation Class" on page 178.

Validation methods are dependent on the user profile. Therefore, any extended user profile must be supplied to the validation plan, as discussed in "Extended User Profile as Supplier" on page 73.

Note A validation must always be defined at your site and registered with the engine. You can only register one validation with an engine. An extended user profile must be registered before registering any validation that references it.

Working with a Validation

This section describes the tasks you are likely to perform when you create or modify a user validation. It is followed by a reference section on the Validation class.

This section covers the following topics:

- opening the Validation Workshop
- editing a validation class
- saving, compiling, and registering a validation

Opening the Validation Workshop

This section contains procedures for creating a new validation and opening an existing validation from the Repository Workshop (illustrated in the following figure).

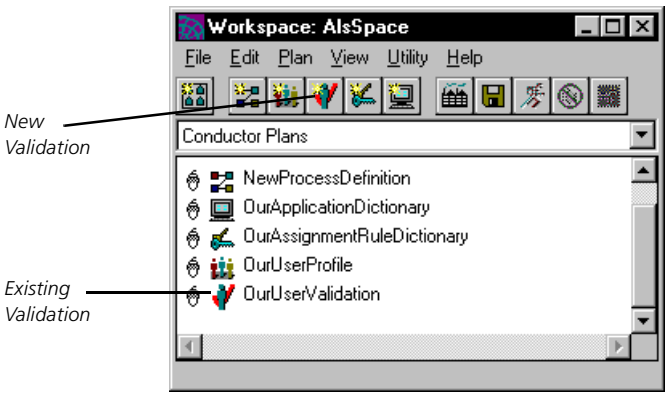
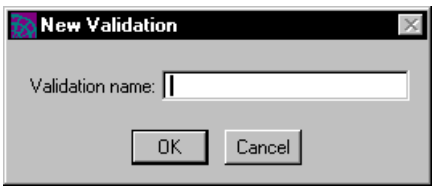


Figure 39 *Opening a Validation in the Repository Workshop*

To open the Validation Workshop and create a new plan:

1 From the Repository Workshop, click the New Validation toolbar button, or choose **Plan > New Fusion Plans > Validation**.

A dialog opens prompting you to name the new validation.



2 Name the validation, and click **OK**.

A new user profile plan opens in the Validation Workshop.

To open the Validation Workshop for an existing plan:

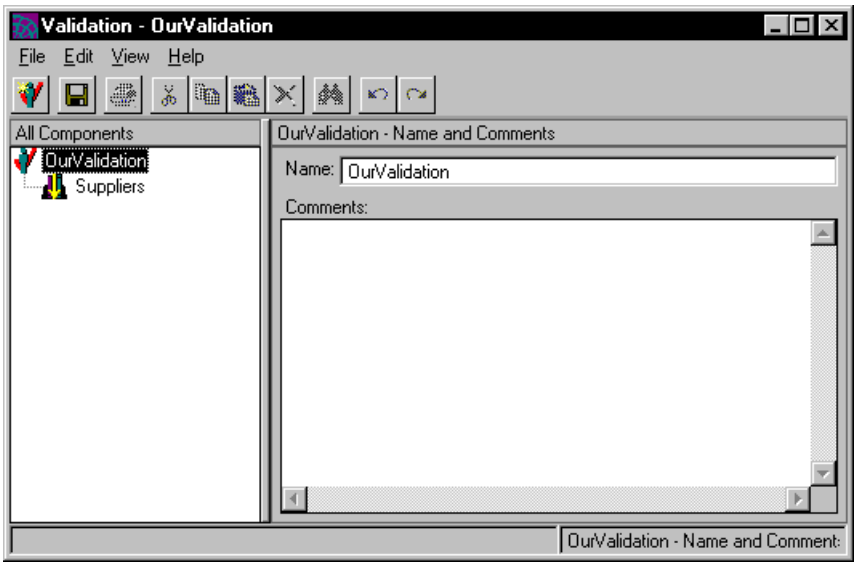
1 From the Repository Workshop, double-click the name of an existing validation in the plan list, or select the name of an existing validation in the plan list and press **Enter**, or select the name of an existing validation in the plan list and choose **Plan > Open.**

See Chapter 3, "Managing Fusion Plans: the Repository Workshop" for more information on the Repository Workshop.

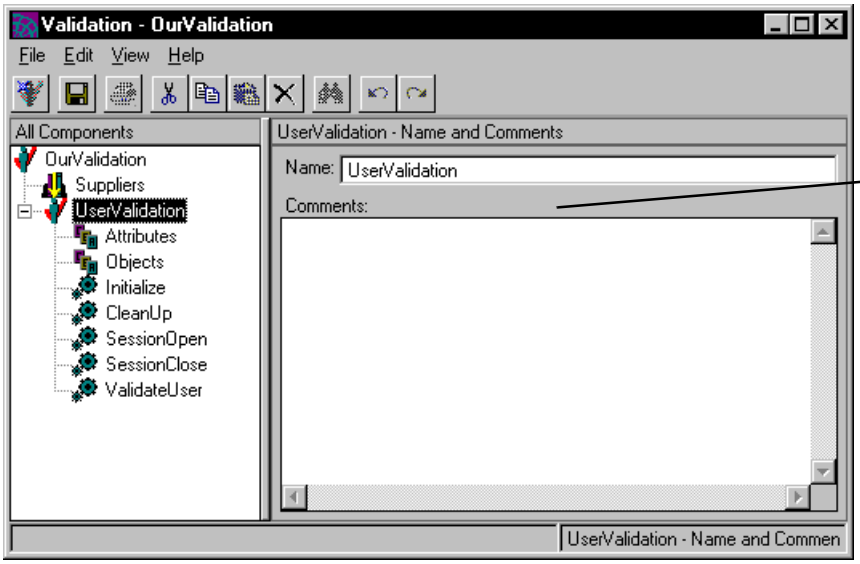
Creating and Editing a Validation

You have to define a custom Validation for your Fusion process management system—in particular you have to write a ValidateUser method that validates users who are opening a session with an engine.

When you create a new validation, the Validation Workshop opens with a new validation plan:



To create a validation class, click the New Validation button at the top left of the toolbar or choose **File > New Validation**:



The list view now displays a new validation class and the elements of that class: attributes, object attributes, and a number of validation methods, the most important of which is the ValidateUser method. Depending on the element you select in the list view, the dialog area changes accordingly.

Note If your validation depends upon an extended (rather than standard) user profile, you must include the user profile as a supplier library to your validation. This requires that the user profile library first be imported into your process development library as described in "Including a User Profile as a Supplier Library" on page 79. To include the user profile as a supplier, click on the Suppliers element in the list view, click the **Edit Supplier List** button (or choose **File > Supplier Plans...**), select the library, and click **OK**.

As you edit the validation, be sure to save your changes periodically, as described in "Saving Changes" on page 175.

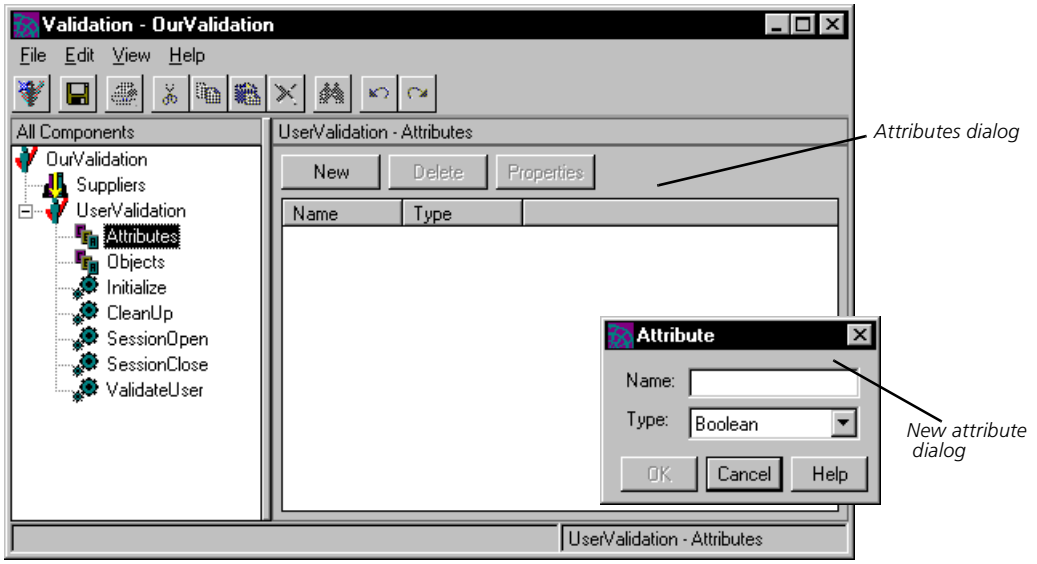
Specifying Validation Properties

To specify user validation properties, select the validation class element. The corresponding panel is displayed on the right. It enables you to enter the name of the validation class and comments about it.

Specifying Validation Attributes

There may be situations when you want to store simple, validation-related information. For example, you might store the name of the engine invoking a validation method, the number of times the method is invoked, information that is passed to an external validation service, or information about the identity of a database accessed by an external validation service. You can store such information as a validation attribute.

To specify an attribute, select the Attributes element in the list view. The corresponding dialog is displayed on the right. It allows you to add validation attributes to the class. If you click the **New** button, you see a dialog prompting you for the name and type of the new validation attribute:



You can create validation attributes with simple types: boolean, double, float, integer, long, and string. The following table describes these Forte TOOL data types (also described in "An Introduction to The TOOL Language" on page 199):

Data Type	Description
boolean	A variable that can take one of two logical values, TRUE or FALSE.
double	Approximately 10E-308 to 10+308 with about 15 digits of precision, depending on your platform.
float	Approximately 10E-38 to 10+38 with about 7 digits of precision, depending on your platform.
integer	A signed, 4-byte integer ranging from -2,147,483,648 to 2,147,483,647 on all platforms.
long	At least -2,147,483,648 to 2,147,483,647--perhaps greater depending on your platform.
string	A simple data type that stores a string constant. There are no string expressions, and although you can compare strings in boolean expressions, there is no way to manipulate the string other than to copy it to a TextData object, manipulate it, and copy it back.

You access these validation attributes through simple TOOL *object.attribute* syntax. For example, use the following expression to set a string attribute called DatabaseVersion to the string constant 'A42':

```
Validation.DatabaseVersion = 'A42';
```

Specifying Validation Object Attributes

If you need to reference a service object in any of your validation methods, you can define an object attribute which references the service object and serves as a handle to it. To define such an attribute, click the New button. A dialog opens, prompting you for the name and class type of the new object attribute. The class type should be the same as the service object you are referencing, and it's definition must be included as a supplier library to your validation plan.

For information on accessing service objects from process definition methods, see "Writing Code that Accesses Forte Service Objects" on page 192. For more information on object attributes, see "Saving a Handle to a Service Object" on page 197.

Writing a ValidateUser Method

You write a ValidateUser method in the ValidateUser method panel. As with all methods in the process development workshops, you write this method in Forte's TOOL language. (For a brief description of TOOL, see "An Introduction to The TOOL Language" on page 199.)

To write a ValidateUser method:

1 Select the ValidateUser method element in the list view.

The method panel is displayed on the right:

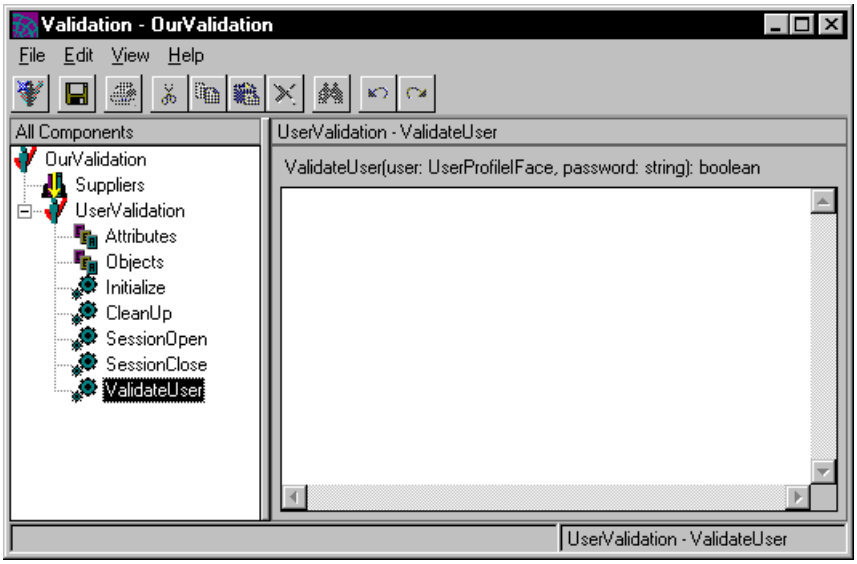


Figure 40 *Validate User Method Panel*

2 Type your TOOL code in the method edit field below the ValidateUser method declaration. The ValidateUser method is described in the following section.

Understanding the ValidateUser Method

In the **ValidateUser** method dialog, shown in Figure 40, you can see the **ValidateUser** method declaration.

ValidateUser (**user**=*UserProfileIFace*, **password**=*string*)

Returns boolean

Parameters	Required?	Input	Output
user	●	●	●
password	●	●	

The method returns TRUE or FALSE, indicating whether the user is validated or not. For more information about the method parameters, see "ValidateUser" on page 180 in the Validation class reference section.

There is no default implementation for **ValidateUser**—you have to write it yourself. The ValidateUser method should verify a user's request to open an engine session. It normally checks a user's name, roles, and login password against information in an organization database. The method can also add information it finds in the database to the user profile object.

To perform these tasks ValidateUser typically employs a number of methods defined on the UserProfile class. The following are some of the more useful methods. (These methods are described in more detail in Chapter 4, "Defining a User Profile.")

Method	Parameters	Returns	Purpose
GetRoles	none	Array of TextData	Returns all the roles this user is using.
GetUserName	none	string	Returns the name of the user.
GetSessionType	none	integer	Returns a value (ADMIN or STANDARD) indicating that the user profile is to be validated, or has been validated, for an administrative or standard (non-administrative) session.
SetOtherInfo	**otherInfo**=*string*	none	Sets the value of any linked user information stored for this user, such as the name of the user's manager. The information is passed as a linked activity parameter (otherInfo) to be used in assignment rule Evaluate methods.
IsEqualRoles	**objectRoles**= *Array of TextData*	boolean	Tests to see if the role or list of roles of the subject is equal the role or list of roles in the objectRoles array of roles.
IsIntersectRoles	**objectRoles**= *Array of TextData*	boolean	Tests to see if at least one of the roles of the subject is equal to at least one of the roles in the objectRoles array of roles.
IsSubsetRoles	**objectRoles**= *Array of TextData*	boolean	Tests to see if all the roles of the subject are in the objectRoles array of roles.

In most cases user information is stored in an organization database external to the Fusion engine. The recommended way to perform user validation is by using an external service that access the organization database and performs the validation tasks using the data stored there. The external service is typically a Forte service object.

Two example ValidateUser methods, taken from Fusion example applications, are presented below: one method performs all the user validation internally, using hard coded user information. The other method calls a service object that provides validation by accessing an external organization database.

ValidateUser Example: Internal Validation

The Expense Report example application uses a ValidateUser method that performs three operations:

- checks if the user/password combination submitted by the user is valid
- checks if the role under which the user is logging in is valid for this user
- if the user has a manager, places the manager's name in the otherInfo attribute of the user's profile (for use by assignment rule Evaluate methods).

```
tmpRoles : array of TextData = user.GetRoles();
...
if(user.GetUsername() = 'Charlotte') and
             password = 'Charlotte') then
  if (tmpRoles[1].IsEqual('Employee')) or
     (tmpRoles[1].IsEqual('Manager')) then
          user.SetOtherInfo('Alice'); -- Alice is Charlotte's manager
          return TRUE;
  end if;
...
return FALSE;
```

See ExpenseReporting example

Plan: ERUV • **Class**: UserValidation • **Method**: ValidateUser

ValidateUser Example: External Validation

In the Advanced Expense Report example application, the ValidateUser method calls a service object to perform the user validations. (For information on using service objects in process definition methods, see "Writing Code that Accesses Forte Service Objects" on page 192.) The "OrgDB" service object performs the same validation as in the basic Expense Report example (see "ValidateUser Example: Internal Validation" on page 174) by accessing data stored in an organization database.

```
anyObject : ObjectWrapper = new;
thisOrgDBSO : OrgDB;
thisOrgDBSO = (OrgDB)(anyObject.FindObject('ODBSO'));
if thisOrgDBSO.VerifyUser(user.GetUserName(),
             password,
             user.GetRoles()) then
user.SetOtherInfo(thisOrgDBSO.GetManagerOfUser(
             user.GetUserName()));
return TRUE;
else
return FALSE;
end if;
```

See Advanced Expense Reporting example

Plan: AERUV • **Class**: UserValidation • **Method**: ValidateUser

Saving and Compiling a Validation

As you work on your validation, it is a good idea to save your work regularly. As you make changes to the Validation class, you can periodically compile your validation plan into a TOOL project to ensure that the syntax is correct.

Saving Changes

Save All command

As you edit the validation, be sure to save your changes periodically (choose **File > Save All**). When you save changes, the current validation plan is updated in your workspace.

Note

If you have any other workshops open for editing, they are saved at the same time.

Compiling a Validation

Compile command

As you write your own validation methods, you might want to compile to ensure that your code is syntactically correct. To compile, choose **File > Compile**. Fusion generates TOOL code from the validation plan and compiles it, saving the result in a read-only TOOL project that has the extension _UV. (This file is a by-product of the compile process; you do not use it.)

If there are compilation errors, Fusion displays them for you. You can then go back to the workshop, fix the errors in your method code, and recompile.

Hint

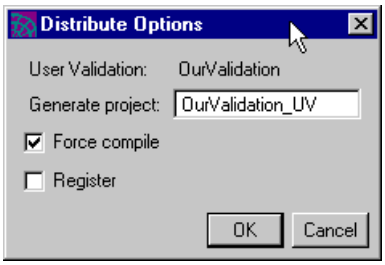
If too many of these generated files begin to clutter your list of plans in the Repository Workshop, you can filter them by choosing Fusion Plans from the plan list filter drop list. (Figure 39 on page 169 shows the Repository Workshop with the filter set this way.)

Making and Registering a Validation

When you have completed all work on a validation and are ready for it to be used by an engine, you make it into a library distribution and register it with one or more engines.

Distribute command

To perform these operations, choose **File > Distribute to open** to open the Distribute Options dialog:



Force Compile option

The **File > Distribute** command performs a compile operation if this option is selected, then uses the resulting TOOL project to make a library distribution. The Generate Project field displays the name of the generated TOOL project. You can enter another name if you like.

Register option

To register the resulting library distribution with an engine, enable the Register option. If the Register option is enabled, you are prompted with a list of engines. Choose the engines you want to register with, then click **OK**. The library distribution is saved in the FORTE_ROOT/appdist directory on the central server node in your Fusion system.

Note

The node hosting a Fusion process engine must be online and the engine running in your environment before you can perform a registration with that engine.

If the engine you want to register with is not available in your environment, copy the generated library from your FORTE_ROOT/appdist directory to the remote environment. Then use the Fusion Console to register the distribution. Refer to the *Forte Fusion Process Management System Guide* for more information.

Creating New Versions of a Validation

Typically, you modify the validation so it can support an extended user profile—for example, to include user profile attributes needed by one or more assignment rules. (The information needed to provide the attribute values must be available in the organization database.) However, there may also be changes to the organizational structure that require changing the validation.

If these changes can be implemented in a *monolithic* upgrade of your Fusion process management system, then your assignment rules only need to support the current user profile. In a monolithic upgrade you shut down all sessions and unregister the old versions of user profile and assignment rule dictionaries—or shut down your engine to perform the changeover—and then reregister the new versions of Fusion distributions.

However, if you need to perform a *rolling* upgrade of your Fusion system, as described in "Multiple User Profiles: Rolling Upgrades" on page 73, it might be necessary for your validation to support more than one user profile. If this is the case, then your ValidateUser method must include a "case" statement or an "if" statement that references more than one user profile.

For example, if two user profiles of class type UserProfile1 and UserProfile2 have been registered with an engine, you might have code similar to the following:.

```
profile1 : UserProfile1 = new;
profile2 : UserProfile2 = new;
if user.IsA(UserProfile1) then
  profile1 = (UserProfile1)(user); --cast to UserProfile1
  . . .  --perform validation on profile1
else
  profile2 = (UserProfile2)(user); --cast to UserProfile2
  . . .  --perform validation on profile2
end if;
```

Note All user profiles referenced should be included as supplier libraries to your validation plan (see "Creating and Editing a Validation" on page 170).

Validation Class

Method Summary

Method	Parameters	Returns	Source Class	Purpose
Cleanup	none	none	●	Performs general cleanup operations such as closing a session with the organization database.
Initialize	**engineName**=*string*, **environment**=*string*	boolean	●	Performs general initialization operations such as opening a session with the organization database.
SessionClose	**sessionName**=*string*, **user**=*UserProfileIFace*, **state**=*integer*	none	●	Implements session control policies in conjunction with the SessionOpen method. Normally deletes session information from the organization database that was inserted by the SessionOpen method.
SessionOpen	**sessionName**=*string*, **user**=*UserProfileIFace*, **connectionType**=*integer*	boolean	●	Implements session control policies in conjunction with the SessionClose method. Normally validates that a session can be opened in accordance with policy and inserts session information into the organization database for future use.
ValidateUser	**user**=*UserProfileIFace*, **password**=*string*	boolean	●	Validates user information against an organization database. Typically verifies that the password and the user profile information in the user object are valid. It can also insert user information from the database into the user object.

Using the Validation Class

The engine uses the Validation class to verify that the user of a session is valid. It accesses the database, verifies any information already in the user profile object, and optionally transfers information from the database to the user profile object.

For your methods to access service objects, you must follow a special technique described in "Writing Code that Accesses Forte Service Objects" on page 192.

Methods

Cleanup

The **Cleanup** method has an empty default implementation—supply your own implementation only if needed. This method defines operations that are performed whenever a validation object is destroyed. The most common operation is to close a session with the organization database. (The **Cleanup** method is invoked whenever the engine shuts down or whenever a new validation is registered with the engine.)

Cleanup ( )

Returns none

Initialize

The **Initialize** method has an empty default implementation—supply your own implementation only if needed. It defines operations that are performed whenever a new validation object is created. The most common operation is to open a session with the organization database. This database session can then be used by the other validation methods to access the organization database. (The **Initialize** method is invoked whenever the engine starts up or whenever a new validation is registered with the engine.)

Initialize (**engineName**=*string,* **environment**=*string*)

Returns boolean

Parameters	Required?	Input	Output
engineName	●	●	●
environment	●	●	

SessionClose

The **SessionClose** method has an empty default implementation—supply your own implementation only if needed. It is an adjunct to the **SessionOpen** method for implementing session control policy, and is invoked whenever a session is suspended or terminated. The **SessionClose** method typically deletes session information from the organization database that was inserted by the **SessionOpen** method.

SessionClose (**sessionName**=*string,* **user**=*UserProfileIFace,* **state**=*integer*)

Returns none

Parameters	Required?	Input	Output
sessionName	●	●	●
user	●	●	●
state	●	●	

state parameter

The **state** parameter indicates the new state requested for the session, letting the **SessionClose** method adjust its behavior depending on whether the session is being suspended or terminated. The values of the parameter are as follows:

State	Value
WFSession.SUSPENDED	The close session request specifies that the session is to be suspended.
WFSession.TERMINATED	The close session request specifies that the session is to be terminated.

SessionOpen

The **SessionOpen** method returns TRUE by default. It is invoked if the ValidateUser method returns TRUE. It typically checks session information in the organization database to confirm that a session can be opened in accordance with a site's session control policy. For example, a policy might restrict a user to a single engine session. If the method confirms the open request, it typically writes session information into the database for use by future session open requests.

SessionOpen (**sessionName**=*string,* **user**=*UserProfileIFace,* **connectionType**=*integer*)

Returns boolean

Parameters	Required?	Input	Output
sessionName	●	●	●
user	●	●	●
connectionType	●	●	

The method returns TRUE or FALSE, indicating whether the user's OpenSession request is accepted or denied.

connectionType parameter

The **connectionType** parameter indicates whether a request to open a session creates a new session object in the engine or reconnect to an existing, suspended session object. The parameter lets the **SessionOpen** method adjust its behavior depending on whether the session to be opened reconnects to a suspended session or creates a new one. The values of the parameter are as follows:

State	Value
WFSession.NEW_SESSION	A new session is being created.
WFSession.RECONNECTED_SESSION	A session is being reconnected to a previously suspended session.

ValidateUser

The **ValidateUser** method returns FALSE by default. It typically accesses the organization database using the user name from the user object and the user's password (passed in by the client application) and verifies that the information in the user object is consistent with information in the organization database. It can also insert user information from the database into the user object.

ValidateUser (**user**=*UserProfileIFace,* **password**=*string*)

Returns boolean

Parameters	Required?	Input	Output
user	●	●	●
password	●	●	

ValidateUser is executed by the engine, which passes in all the parameters when it calls the method.

The method returns TRUE or FALSE, indicating whether the user is authenticated against the organization database.

user parameter

The **user** parameter is the user profile object of the session being evaluated. The UserProfileIFace type represents the user profile class that has been defined at your site in the User Profile Workshop and registered with the engine. (If your user profile has not been extended, it represents the default user profile class provided with Fusion process management system.)

If you have an extended user profile, and you want your ValidateUser method code to access the extended user profile attributes, you have to cast the user parameter to your extended user profile type. For example, if your extended user profile class type is ExtendedUserProfile, you would write code similar to the following:

```
MyUser : ExtendedUserProfile;
MyUser = (ExtendedUserProfile)(subject);
MyUser.attribute = DatabaseValue);
```

password parameter

The **password** parameter is the password used when the user started the client session.

Chapter 9

Writing Fusion Process Definition Methods

This chapter provides background information useful in writing methods in any of the process development workshops. It covers the following topics:

- using the TOOL language to write code in process definition methods
- accessing and using process attributes in process definition methods
- methods you can call from process definition methods
- calling service objects from process definition methods

Writing Code in Process Definition Methods

The process development workshops enable you to do much of the work of designing process definitions without having to write code. However, to implement behavior that is different from the default functionality built into the product, you have to write at least some method code. This customization occurs in *process definition methods*, which are defined in the process development workshops and are executed by the Fusion process engine.

For example, in the Process Definition Workshop, if you create two activities, Act1 and Act2, and connect them with a router, by default the second activity becomes READY as soon as the first one completes. This behavior is controlled by several methods, each of which has default behavior:

- Act1's OnComplete method by default activates the router.
- The router method by default transfers control to Act2.
- Act2's Trigger method by default returns TRUE and Act2's Ready method executes.
- Act2's Ready method by default transitions the activity to the READY state.

If you wanted Act2 to wait until a process attribute changed before the activity could be triggered, you have to alter one of the methods (probably Act2's Trigger method) to test the value of the process attribute and allow the activity to trigger only if that test condition is met.

To alter the method this way, you add the process attribute to the method's attribute access list and then write a single line of code. (For a full description of how to use attributes in a method, see "Process Attribute Data Types" on page 189.)

For example, if the process attribute is TotalAmt and your method triggers only if TotalAmt is greater than $1,000, the following code would work:

```
return (TotalAmt > 1,000);
```

To override an activity's default Trigger method:

1. Choose the activity and display its property inspector.
2. Click the Trigger/Ready tab.

   In the Trigger/Ready tab page, the default Trigger method is set to Trigger When Any Router Arrives.
3. Change the trigger type to Custom Trigger.
4. Click the **Attributes** button to specify an attribute access list.

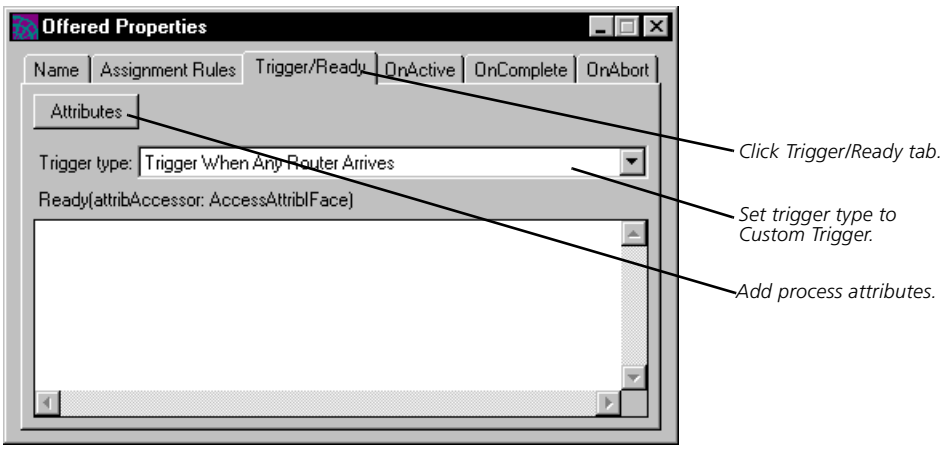


Figure 41 *Editing a Method on the Trigger/Ready Tab Page*

5 Specify process attributes in the attribute access list.

See "Specifying an Attribute Access List" on page 185 for more information.

6 Enter code in the Trigger method edit field.

Basic Language Syntax for Methods

The language used in process definition methods is Forte TOOL (Transactional Object Oriented Language). This language, described completely in the Forte *TOOL Reference Manual*, is a complete, fourth-generation language with capabilities that extend well beyond the typical requirements of a process definition method. For most of your Fusion work, you need to understand only a simple subset of TOOL:

- the syntax of method calls and how to use the *return* statement
- variables and how to use them
- some simple data types
- the operators that let you do comparisons
- possibly some statements like *if* and *while* that let you control the flow of your code

This section covers some basic information about method syntax and describes the return statement in some detail. For information on TOOL variables, data types, operators, and flow control statements, see "An Introduction to The TOOL Language" on page 199.

Method Syntax

When you edit a method in the process development workshops, you see the method's declaration at the top of the edit field. For example, the method declaration for the Trigger method is:

```
Trigger(attribAccessor:AccessAttribIFace):boolean
```

This method has

- a name, Trigger
- a parameter, attribAccessor, that provides information to the method (in this case, the process attributes that have been associated with this method)
- a return value, which is of type boolean (TRUE or FALSE)

You should be aware of the following before customizing a method:

- What is the method's purpose?

  For example, the Trigger method is used to move the activity from a PENDING state to a READY state (after its Ready method, if any, is executed).

- What is its return type?

  The Trigger methods's return type is boolean, which is also the case for all activity and router methods. If the method returns TRUE, execution continues.

- Which process attributes does it need to access?

  For a process definition method, your code can reference by name any attributes in the Process Attributes list, as long as they are in the method's attribute access list. (See "Specifying an Attribute Access List" on page 185 for more information.)

When you write code in the method edit field you are writing the body of the method—the code that the engine executes. This code overrides the default method code originally supplied. You set the return value with the *return* statement, which is described next.

The return Statement

Every activity and router method must return a boolean value, a value of TRUE or FALSE. Use the return statement in a method to return a value:

```
return (expression);
```

The *expression* inside the parentheses evaluates to TRUE or FALSE, such as a comparison of two or more elements (two variables, a variable and a value, and so on) or the constants TRUE or FALSE themselves. This kind of expression is called a *boolean expression*, such as:

```
(a = 3)          //Is the variable a equal to 3?
(TotalAmt > 500) //Is process attribute TotalAmt greater than 500?
(TRUE)           //The boolean value TRUE is always true.
```

You can use a return statement anywhere in your method. However, when a return statement is encountered, the method ends, and any remaining statements in the method are ignored. Therefore, be careful to set up the return statement so that it is the last statement evaluated in the method.

For example, the following three samples of code test if the process attribute *age* is equal to the value 18. If it is, the method returns TRUE. If it is not (age is something other than 18) the method returns FALSE. The code can be written in several ways: one that stores the value to be returned and returns it at the end, another that returns the value when it is determined, and another, even more economical way, that returns the value of the test itself:

Return it at end

```
mustReg : boolean;
if (age = 18) then
  mustReg = TRUE;
else
  mustReg = FALSE;
return mustReg;
```

Return when value is determined

```
if (age = 18) then
  return TRUE;
else
  return FALSE;
```

Directly return value of test

```
return (age = 18); // TRUE if age is 18, FALSE otherwise
```

Accessing and Using Process Attributes

Many process definition methods use process attributes as a basis for making process logic decisions. Foremost among these are Trigger methods, Router methods, and the Evaluate method of assignment rules.

When a process instance changes state, you might also change some process attribute values. Methods you typically use for this purpose are an activity's Ready, OnActive, OnComplete, and OnAbort methods, and a timer's OnExpiration method.

All these methods access process attributes through attribute accessors. An attribute accessor is an object that provides an attribute access list: a set of process attributes with a lock type specified for each. You must specify an attribute access list for each method that uses process attributes (see "Specifying an Attribute Access List"). The lock controls access to the process attribute by other activities in the process. (There is a full description of locks and how to set them later under "Specifying Lock Types" on page 187.)

Each method that uses process attributes has an attribAccessor input parameter. The engine passes the attribute accessor object to the method through this parameter. You can read a process attribute, and change its value if you have set the appropriate lock on the attribute. Each activity method can use a set of methods defined for the attribute accessor object to get an attribute's value, set its value, and get its lock type.

Specifying an Attribute Access List

When you want to access process attributes from a method, you must specify an attribute access list for that method. For each attribute in the list, you can specify the lock type applied to the attribute during execution of the method. (As explained earlier, the attributes in your attribute access list are made available to the method through an attribute accessor object that is passed to the method as an attribAccessor input parameter.)

In some cases two methods share a common access list to make attribute locking more efficient. The following table documents these cases:

Methods Sharing an Access List	Attribute Locking Behavior
Activity: Trigger and Ready method	Attribute locks are released when Trigger returns FALSE. When Trigger returns TRUE, locks are held until Ready method completes execution.
Activity: OnComplete and OnComplete router methods	Attribute locks are held during execution of OnComplete method and until all OnComplete router methods complete execution.
Activity: OnAbort and OnAbort router methods	Attribute locks are held during execution of OnAbort method and until all OnAbort router methods complete execution.
Timer: OnExpiration and Expiration router methods	Attribute locks are held during execution of OnExpiration method and until all Expiration router methods complete execution.

All process definition method entry dialogs provide the same mechanism for specifying the method's attribute access list. For example, Figure 42 on page 186, illustrates the Trigger/Ready tab page used to write both the Trigger and Ready methods of an activity.

The top of the tab page contains the user interface for constructing the attribute access list, which is shared by the Trigger and Ready methods. The bottom of the page contains the method entry fields.

You can hide the attribute list by clicking the Hide button. All the buttons are replaced by an Attributes button, which you can click to display the list again.

You can click the Add button to specify one or more process attributes to access from the corresponding methods. The process attributes you specify must have been previously defined in the process definition, as described in "Defining Process Attributes" on page 138.

Note In addition to the procedure described in the following paragraphs, you can also construct attribute access lists for a method by dragging process attributes from the Process Attributes list in the Process Definition Workshop and dropping them on the corresponding method's attribute access list.

The following figure shows the Trigger/Ready tab page used to write the Trigger and Ready methods of an activity.

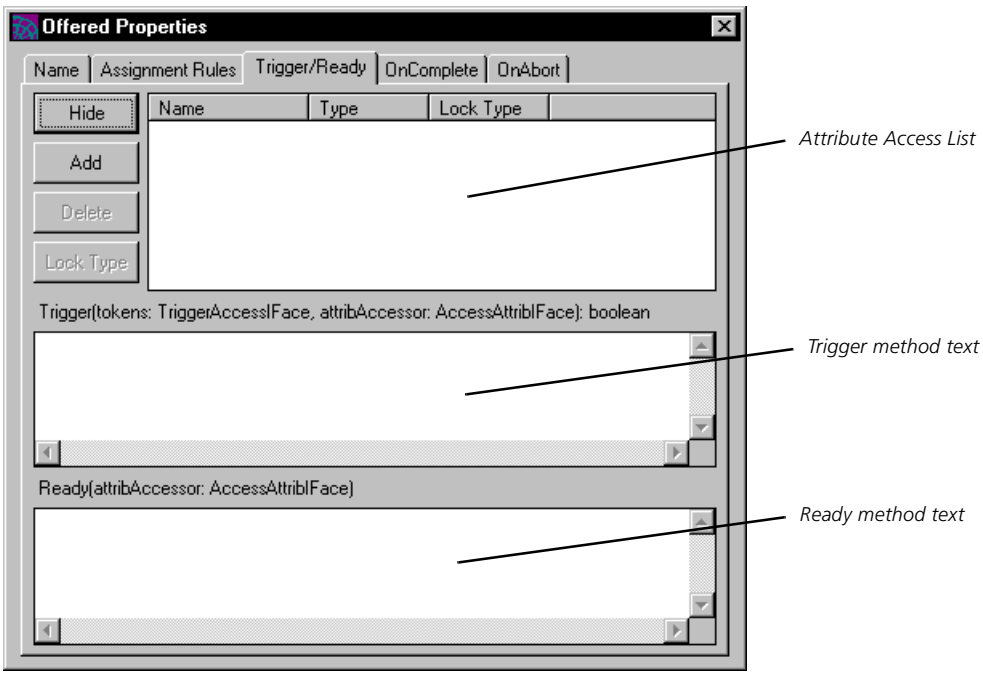


Figure 42 *Trigger/Ready Tab Page*

To add process attributes to the attribute access list:

1 Click the **Add** button.

The Select Process Attributes dialog appears. The dialog displays the list of process attributes defined in the process definition.

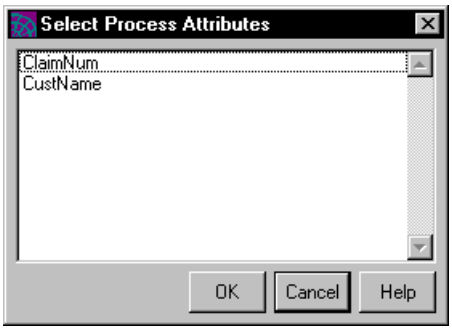


Figure 43 *Select Process Attributes Dialog*

2 Choose one or more process attributes.

3 Click **OK** to add them to the attribute access list.

To delete an attribute from the list:

1 Select the attribute.

2 Click **Delete**.

Specifying Lock Types

When your method accesses a process attribute, a lock request corresponding to the attribute's lock type is sent to the engine. By default, when you add an attribute to your method's attribute list, it gets a lock type of NO_LOCK. You can specify a lock to ensure that the value the method reads from or writes to the attribute is accurate and is not overwritten while the method is accessing the attribute.

To specify a lock type for an attribute:

1 Select the attribute in the attribute list.

2 Click the **LockType** button to display the Expression Data properties dialog.

3 Choose a lock type from the drop list.

4 Click **OK** to update the attribute lock type.

The following table describes the lock request types:

Lock Request	Description
READ	The attribute is to be locked for reading, preventing others from gaining a WRITE lock to it, but allowing others to read it. If the attribute is currently WRITE locked, this request fails.
READQ	This request is the same as READ, except that the request waits—is queued—if the attribute is currently WRITE locked.
WRITE	The attribute is to be locked in preparation for getting or changing its value or both. This lock prevents others from obtaining a READ or WRITE lock (but an accessor using NO_LOCK can still read it). If the attribute currently has a either a READ or WRITE lock, this request fails.
WRITEQ	This request is the same as WRITE, except that the request waits—is queued—if the attribute is currently locked.
NO_LOCK	This lock request type is the default type. The attribute is not locked, and a copy is made of its value, regardless of any locks that might currently be set on this attribute. The purpose is to obtain its current value, with the understanding that the value might change at any time.

Working with Process Attributes

Process definition methods available from within the Process Development Workshop contain an attribAccessor parameter, which provides access to the process attributes defined for the process definition. Using the methods available with the attribAccessor parameter, you can get and set the values of process attributes, get the lock type of an attribute, and also get a list of the attributes in the process definition.

Accessing Process Attributes by Name

The attribAccessor parameter, available to most methods in the Process Definition Workshops, provides generic methods to get and set attribute values. However, you can also access process attributes directly by name.

Most process attributes can be accessed by name using a simple data type, which simplifies the code you write in your methods.

For example, if you want a router method to return TRUE if the IntegerData process attribute *status* has the value 1, you can access the attribute by name as indicated below:

```
return (status = 1);
```

However, you can accomplish the same thing using the GetValue method of the attribAccessor parameter to the router method.

```
return (attribAccessor.GetValue('status') = 1);
```

For process attributes accessible as simple data types, you can use operators to build numeric expressions, as explained in "Numeric Expressions" on page 208. Refer to Table 1 on page 189 for a list of process attribute data types and their corresponding data types when you access the attribute by name.

AttribAccessor Parameter

attribAccessor method list

The following table shows the generic AttributeAccessor methods available from the attribAccessor parameter.

Method	Parameters	Returns	Purpose
GetLockType	**name**=*string*	integer	Returns the lock type of the specified process attribute.
GetNames	*none*	Array of TextData	Returns the names of the attributes accessible from this accessor.
GetValue	**name**=*string*	DataValue	Returns the value of the specified attribute.
SetValue	**name**=*string*, **newValue**=*DataValue*	none	Sets the value of the specified attribute. (A WRITE lock should have been set on the attribute.)

For example, suppose your process definition has a process attribute of type BooleanData named *priority*. In your method (such as the OnActive method), the following line of code sets the value of *priority*.

```
SetValue('priority', FALSE);
```

In this example, if you want to determine the type of lock for the process attribute before setting its value, you can use the following code in your method:

```
integer priorityLockType;
priorityLockType = attribAccessor.GetLockType('priority');
```

Process Attribute Data Types

A process attribute can be of one of the data types shown in the table that follows. All these data types are described in detail as data structure classes in the Forte Framework Library online Help. Most of them can be accessed by name using a corresponding simple data type, which simplifies the coding needed to get and set process attribute values (see "Simple Data Types" on page 202).

The following table shows the process attribute types and the corresponding simple data types (see "Simple Data Types" on page 202 for more information on using these data types). If the simple data type is *none*, then you can access the attribute by name using its attribute data type.

Table 1 *Process Attribute Data Types*

Attribute Data Type	Simple Data Type	Process Data Type Description
BooleanData	boolean	An object that is equivalent to a TOOL boolean data type.
DateTimeData	none	An object that represents any date, timestamp, or time. This type is based on the SQL standard and represents a combination of the SQL DATE, TIME, and TIMESTAMP data types. A DateTimeData object typically starts at the beginning of the Gregorian calendar—Friday, October 15, 1582—and extends into the indefinite future. It contains attributes that allow specification of year, month, day, hour, minute, second, and millisecond, although accuracy of DateTimeData is guaranteed only to the second across all platforms.
DecimalData	none	An object that represents floating point data scaled to a specified decimal precision of as many as thirty decimal places. DecimalData is often used to represent monetary values in greater precision than possible with DoubleData.
DoubleData	double	An object that is equivalent to a TOOL double precision scalar data type, approximately 10E-308 to 10+308 with about 15 digits of precision, depending on your platform.
IntegerData	integer	An object that is equivalent to a TOOL 4-byte integer scalar data type, a signed, 4-byte integer ranging from -2,147,483,648 to 2,147,483,647 on all platforms.
IntervalData	none	An interval of time based on the SQL Interval data type, this object is made up of a set number of years, months, days, hours, minutes, seconds, and milliseconds. You can use the SetUnit method to set the value of an IntervalData object and you can use the GetUnit method to extract each type of unit (such as day, year, minutes, and so on) from an IntervalData object.
TextData	string	An object that is equivalent to a TOOL string data type, but with the addition of TextData conversion functions and automatic memory allocation.
XmlData	none	XML data that is typically supplied to a Fusion Backbone for use in Fusion application proxies. The XML data must be well-formed. Fusion imposes no limit on the size of the Xml data. For more information on using XmlData, refer to the proxy document section of the Fusion Backbone online help.

Interacting with Activities from an Activity Method

This section describes three methods available to activity and timer methods that let you interact with the running instances of activities. The methods are:

- GetManager

  Provides access to the process management system.

- GetPreviousState

  Gives you information about the previous state of the current activity. For information about activity states, see "Activity States" on page 119.

- AbortActivity

  Forces an activity instance to terminate abnormally (to abort).

Activity methods can make GetPreviousState and AbortActivity requests to the process management system for activity instances in the current process instance. The first step in making these requests is to obtain access to the process management system by calling the GetManager method, which returns a Manager object.

Having obtained this object, you can call GetPreviousState or AbortActivity. Typically, you combine the call to GetManager with the call to the requesting method. For example:

```
GetManager().AbortActivity(
             'AnotherActivity', ActivityConstants.READY);
```

Caution AbortActivity should be used with care,—aborting a method can cause work to be lost and the process instance to abort. The activity being aborted should have an OnAbort router that provides the appropriate behavior when the activity is aborted.

See the descriptions that follow for more information on these methods and on how to use them.

GetManager Method

Provides the AbortActivity method and the GetPreviousState method with access to the process management system, which is necessary before calling either of those methods.

GetManager ( )

Returns *Manager*

GetPreviousState Method

Returns the previous state of the current activity instance (the activity from which this method call is made).

GetPreviousState ( )

Returns *integer*

GetPreviousState returns one of the following values:

Activity State	Description
ActivityConstants.PENDING	The activity was previously in the PENDING state.
ActivityConstants.READY	The activity was previously in the READY state.
ActivityConstants.ACTIVE	The activity was previously in the ACTIVE state.

This method requests the previous state of the current activity. Before calling GetPreviousState, you must first obtain the interface to the process execution manager by calling the GetManager method, as shown in the following code sample:

```
activityState : integer;
activityState = GetManager().GetPreviousState();
```

Your activity methods can use the return value to determine how the activity got to the current state.

For example, if GetPreviousState is called from an activity's OnAbort method, the return value can be ACTIVE or READY. If the previous state was ACTIVE, then the OnAbort method may have to perform some cleanup in an enterprise database, whereas if the previous state was READY, this is not necessary.

AbortActivity Method

Requests that all activity instances in the current process instance with the specified name and state be aborted. The method is available in any activity method or timer method written in the Process Definition Workshop.

AbortActivity (**activityName**=*string*, **expectedState**=*integer*)

Returns *boolean*

Parameters	Required?	Input	Output
activityName	●	●	
expectedState	●	●	

The return value is TRUE if an activity was found that was in the specified state or FALSE if none of the activities by that name were in the specified state. A return value of TRUE means that at least one activity by that name was aborted. If no activities are found that have the specified name, the method throws an exception.

This method makes a synchronous request to abort the named activity instance if it is in the specified state. If an activity instance of the correct name is found, but it is not in the expected state, it is ignored.

AbortActivity requires that you obtain the interface to the process execution manager by calling the GetManager method. For example, the following code sample aborts an instance of the activity named AnotherActivity that is in the READY state:

```
GetManager().AbortActivity(
             'AnotherActivity', ActivityConstants.READY);
```

activityName parameter

The **activityName** parameter is the name of the activity in single quotes. If the name is not defined in the process, you get a compile error for the process. In addition, the engine checks for the activity name when the process runs and raises an exception if the activity does not exist.

expectedState parameter

The **expectedState** parameter indicates the state the activity must be in to be aborted. The following table shows the values you can use:

Activity State	Description
ActivityConstants.PENDING	The activity is in the PENDING state.
ActivityConstants.READY	The activity is in the READY state.
ActivityConstants.ACTIVE	The activity is in the ACTIVE state.

Writing Code that Accesses Forte Service Objects

There are a number of situations where you may want to access Forte service objects (shared services) from process definition methods—methods that are executed by the Fusion engine. Whenever such a method requires a computation that uses external data or which is intensive enough to negatively impact engine performance, it's best to let a Forte service object perform the computation. The most typical scenarios are the following:

- You need to authenticate users opening engine sessions against an organization database and provide information for populating user profile objects, so you write a ValidateUser method that calls out to a user validation service.
- Your process definitions contain one or more automatic activities with OnActive methods that are computation intensive or perform work in the application domain. You call out to an application domain service to perform these computations.
- Your process definitions have process attributes that also serve as application data. If activity methods or router methods change the values of these process attributes, you want these methods to also update the database that stores your application data. You call out to a data management service that maintains a session with your application database.
- You've defined assignment rules that need to perform sophisticated calculations or access external data sources. Your assignment rule's Evaluate method calls an external service to perform the calculation.

In each of these cases, methods should access a Forte service object. However, the Fusion engine unit is a Forte partition built without knowledge of service objects in your Forte application environment. This means the engine cannot *directly* access service objects. Consequently, you must use the approach described in this section if you want to access service objects from process definition methods.

The approach is two-sided.

- To access a service object, you must first *explicitly* register it with the Forte name service—as opposed to the *implicit* registration normally performed by the runtime system. Explicit registration involves choosing a name to register a service object with and writing code that performs the registration when the service object is initialized at start-up time.
- Secondly, the method code that accesses the service object must request a reference to it from the name service using the name the service object was registered with at start-up time.

To facilitate explicit registering and referencing of service objects, the Fusion process management system software includes a library distribution named WFAccessServiceObj. This library defines an ObjectWrapper class that provides methods for "wrappering" the service object: explicitly registering it, and subsequently referencing it, from the name service.

While in concept this approach is straightforward, there are a few complicating details that arise from the way Forte partitions (Fusion engine units) dynamically load libraries (Fusion distributions and their supplier libraries). The following conditions must be met for methods in a Fusion plan to be able to access service objects:

- Suppliers to the Fusion plan must be supplier *libraries*.
- Supplier libraries to the Fusion plan (which must be dynamically loaded) cannot contain service object definitions.
- To be dynamically loaded, a supplier library to the Fusion plan must be installed on the node (or nodes) hosting the Fusion engine unit.

These conditions frame the following description of how to implement access to service objects from process definition methods.

Implementing Access to Service Objects

There are two sides to accessing service objects: explicitly registering them with the name service, on one side, and referencing them using the name service registrations, on the other. Each of these operations is described separately in the sections that follow.

In these descriptions, it is assumed that you are using two development repositories: one for developing code in the application logic domain (persistent business objects, shared services, and client UI windows) and another for the code in the process logic domain (Fusion process development). Since Fusion process client applications bridge the two domains, they can be developed in either repository—or even a third—as long as the repository contains all the supplier projects and libraries needed to write client code. Refer to "Process Controller Architecture" on page 27 for a fuller discussion of the overall domain architecture of a Fusion enterprise application.

While it is not a strict requirement that you use separate repositories, doing so makes it easier to modify and maintain library source code.

Explicitly Registering a Service Object

To explicitly register a service object with the name service, you have to write code in the init method of the service object class. The conceptual scheme is illustrated in Figure 44.

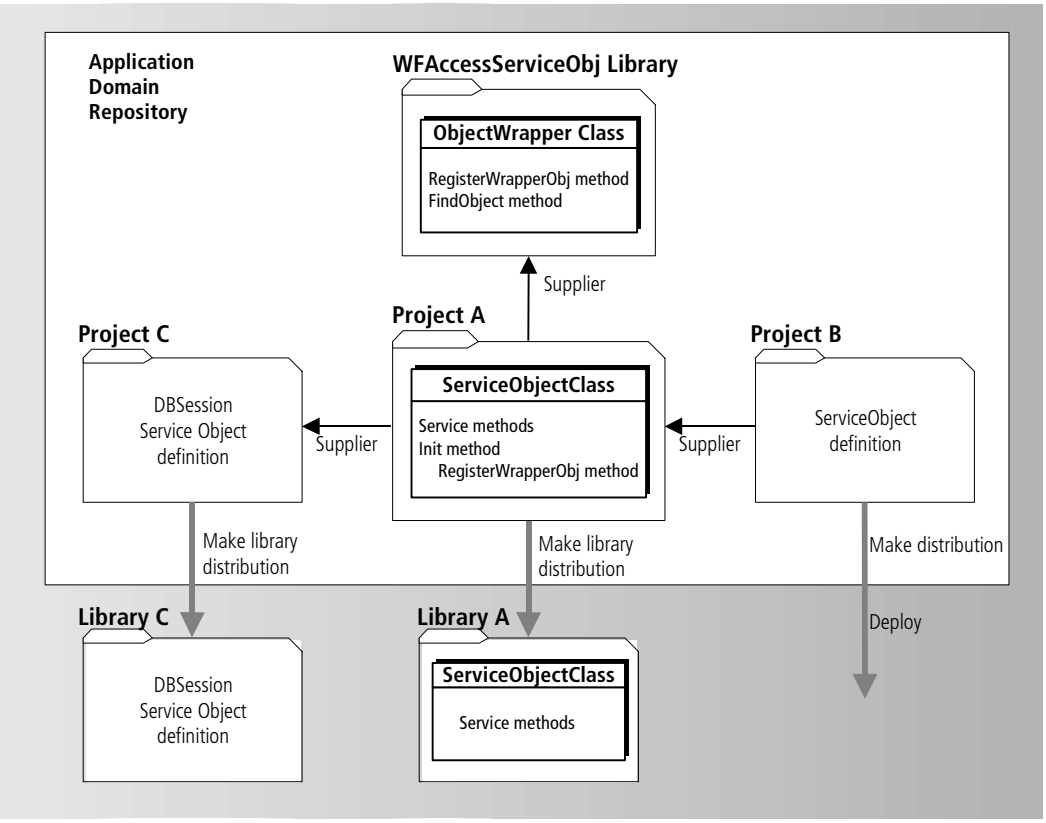


Figure 44 *Scheme for Registering Service Object with Name Service*

Figure 44 shows the Forte projects you need and how to structure them within your application domain repository.

The service object you want to access from a process definition method is defined in Project B. The methods that can be invoked on the service object are defined in the ServiceObjectClass in a separate project, Project A. Project A is a supplier to Project B. If methods of the service object class in Project A invoke other service objects (for example, a DBSession service object, in order to access a database), then those service objects should be defined in one or more separate projects, represented as Project C. Project C is a supplier to project A.

Note The arrangement of projects in Figure 44 is due to the fact that Library A, generated from Project A, is dynamically loaded into the Fusion engine, and dynamically loaded libraries cannot contain service objects.

To explicitly register a service object:

1. In your application domain repository, create a project (Project A) that defines the class (ServiceObjectClass) of the service object.
2. Create another project (Project B), with Project A as a supplier, and define the service object in Project B.
3. Include the Fusion WFAccessServiceObj library in your workspace and make it a supplier to Project A.
4. Create an Init method for ServiceObjectClass that performs an explicit registration. This method:
   a. instantiates an ObjectWrapper object
   b. invokes the RegisterWrapperObj method, supplying a name for explicit registration (see "WFObjectWrapper Methods" on page 198).

   As an example of this step, the Advanced Expense Reporting sample application contains a CheckProcessingMgrSO service object that is accessed from an OnActive method. To register this service object, the Init method of the CheckProcessingMgr class contains the following code, which registers the service object under the name "CheckProcessingSO."

```
tmpReg : ObjectWrapper = new;
tmpReg.RegisterWrapperObj('CheckProcessingSO',self);
```

   Project: AdvExpenseReportClasses • **Class**: CheckProcessingMgr • **Method**: Init

   See Advanced Expense Reporting example
5. In the Partition Workshop, configure Project A (and Project C, if any) as a library.
6. Make a distribution for Library A (and Project C, if any).

   A library distribution contains .pex files for importing the library into a development repository, as well as shared library files that must be installed on any node executing the code. In this case, the library must be installed on the node hosting the Fusion engine unit—where the library will be dynamically loaded.
7. In the Partition Workshop, make an application distribution for Project B and install it in your deployment environment. Once installed, start up the service object partition.

   As an alternative to this step, depending on the function of the service object, you can make Project B a supplier of a client application that invokes the service object, and start up the client—which autostarts the service object. There are a number of scenarios possible, but the point is to get the service object up and running in your Fusion application environment before the engine tries to access it.

Referencing an Explicitly Registered Service Object

To access an explicitly registered service object, your process definition method must reference the service object using its registered name. The conceptual scheme is illustrated in Figure 45.

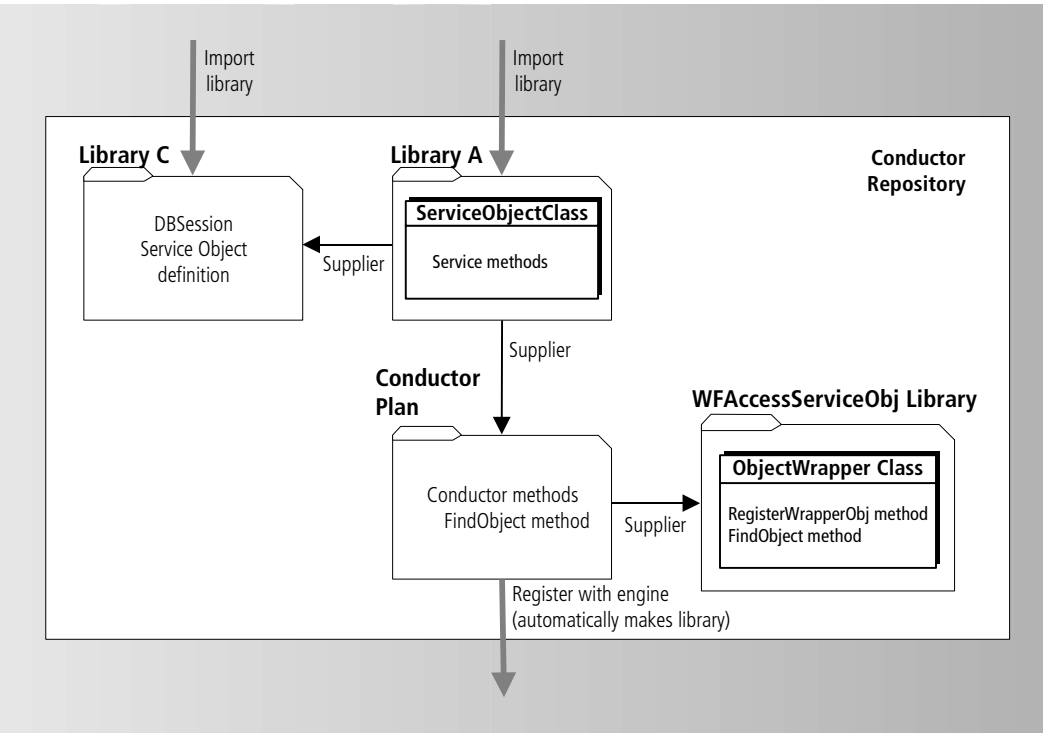


Figure 45 *Scheme for Accessing Explicitly Registered Service Object*

Figure 45 shows the Forte projects you need and how to structure them within your development repository.

In Figure 45, Library A, defining the methods invoked on the service object, is imported into the development repository (as are any supplier libraries of Library A, shown as Library C). Library A is supplied to the Fusion plan defining the method accessing the service object. The WFAccessServiceObj library is also supplied to the Fusion plan.

In the method, the FindObject method, defined in WFAccessServiceObj library, is used to reference the service object. The methods defined in Library A on the ServiceObjectClass can then be invoked on the service object.

When development of the Fusion plan is completed, the plan is registered as a library with the Fusion engine and dynamically loaded. At execution time, the engine also dynamically loads Library A (as well as the WFAccessServiceObj library) so it can invoke the methods defined in Library A. The methods can invoke any of the methods and attributes defined in Library A, but cannot instantiate any classes defined in a supplier to Library A.

Note Fusion process client applications may also need code contained in projects and libraries in the application logic domain, such as Library A and Library C. In that case these libraries need to be imported into whatever repository is used to develop process client applications.

▶ **To access an explicitly registered service object:**

1. In your development repository, import Library A, which contains the ServiceObjectClass definition, and Library C, if any, which supplies Library A.
2. Include WFAccessServiceObj library in your workspace.

3 Make Library A and WFAccessServiceObj library suppliers to the Fusion plan containing the method that you want to access the service object:

   In the Process Development Workshop, choose **File > Supplier Plans** and add these two libraries.

4 In the method code, reference the service object:

   a Instantiate an ObjectWrapper object.

   b Declare an object of type ServiceObjectClass.

   c Call the FindObject method, specifying the name registered by the service object class (see "WFObjectWrapper Methods" on page 198).

   d Cast the object returned by FindObject to the ServiceObjectClass.

5 Invoke any of the service object methods on the ServiceObjectClass object.

   As an example of steps 4 and 5, the Advanced Expense Reporting sample application contains an OnActive method that invokes a method on a CheckProcessingMgrSO service object. The OnActive method contains code that references the service object under the name "CheckProcessingSO" and then invokes a GenerateStatement method defined in the CheckProcessingMgr class, as shown in the following code:

Cast object returned by FindObject

```
myWrapper : ObjectWrapper = new;
myCheckProcessingSO : CheckProcessingMgr;
myCheckProcessingSO =
(CheckProcessingMgr)(myWrapper.FindObject('CheckProcessingSO'));

myCheckProcessingSO.GenerateStatement(ExpenseReportID);
return TRUE;
```

See Advanced Expense Reporting example

```
Plan: AERPD • Activity: ProcessCheck • Method: OnActive
```

6 Register the Fusion plan's library distribution with your Fusion engine.

7 Make sure Library A is installed on the node hosting your Fusion engine.

Implementation and Access Issues

In accessing service objects as described in the previous sections, there are two additional issues that you need to take into consideration:

- How do you access replicated service objects?
- How can you save a handle to a service object in order to boost engine performance?

These issues are discussed in the following sections.

Replicated Service Objects

When service objects have been marked as replicated for failover or load balancing, the approach described in "Implementing Access to Service Objects" on page 193 still works, but because registration with the name service is explicit rather than implicit, some of the failover and load balancing mechanisms built into the Forte runtime system are not available. Depending on the situation, you might have to add some additional code.

If a service object is replicated for load balancing, it gets explicitly registered multiple times in the name service. When you reference the service object using the FindObject method, one of the registrations is selected at random. There is no router mechanism that balances requests for the service among the available replicates.

If a service object is replicated for failover, it also gets explicitly registered multiple times in the name service. When you access the service object using the FindObject method, one of the registrations is selected at random. If that service fails for some reason, the Forte system cannot automatically reroute your reference to another available replicate of the service object. Instead, the system raises a DistributedAccessException. Your code must handle this exception by retrying the FindObject method. When a replicate of the service is available, FindObject returns this object and you can proceed as normal.

Saving a Handle to a Service Object

It is not unusual for process definition methods (activity methods, router methods, assignment rule Evaluate methods, and ValidateUser methods) to be executed thousands of times within a short time interval—depending on the number of process instances being concurrently executed and the number of current active sessions. If such methods need to access a service object, there can be a considerable performance hit taken from having to reference the service object every time the method is executed. Where possible, it is much more efficient to reference the service object once and save a handle to it for subsequent executions of the method.

For the ValidateUser and Evaluate methods, which commonly access a service object, you can save the service object reference as an object attribute of the Validation and assignment rule classes, respectively.

ValidateUser example

For example, if your ValidateUser method needs to access a service object of type OrgDB, registered as "ODBSO," you can define an object attribute in your validation named OrgDBSOHandle of type OrgDB. You can then reference the "ODBSO" service object once in the Initialize method of your validation class (see "Initialize" on page 179 for information on this method) and save the reference as OrgDBSOHandle, as shown in the following code:

```
anyObject : ObjectWrapper = new;
self.OrgDBSOHandle = (OrgDB)(anyObject.FindObject('ODBSO'));
```

In your ValidateUser method, you no longer have to reference the service object. You simply use OrgDBSOHandle (see "ValidateUser" on page 180 for information on this method), as shown in the following code:

```
if self.OrgDBSOHandle.VerifyUser(user.GetUserName(),
             password,
             user.GetRoles()) then
return TRUE;
else
return FALSE;
end if;
```

Assignment rule example

In the case of assignment rule Evaluate methods, there is no Initialize method to use for setting the service object handle, so you need to do it the first time the Evaluate method is executed. For example, if your Evaluate method needs to access a service object of type ARCalc, registered as "ARCalcSO," and you define an object attribute in your assignment rule named ARCalcSOHandle of type ARCalc, you can use code like the following:

```
if self.ARCalcSOHandle = NIL then
anyObject : ObjectWrapper = new;
self.ARCalcSOHandle = (ARCalc)(anyObject.FindObject('ARCalcSO'));
end if;
self.ARCalcSOHandle.DoSomething();
```

WFObjectWrapper Methods

RegisterWrapperObj

The **RegisterWrapperObj** method is used to explicitly register service objects with the Forte name service.

RegisterWrapperObj (**name**=*string,* **serviceObj**=*Object*)

Returns

Parameters	Required?	Input	Output
name	●	●	
serviceObj	●	●	

name parameter

The **name** parameter is the name you use to register the service object with the Forte name service.

serviceObj parameter

The **serviceObj** parameter is the service object to be registered with the Forte name service.

FindObject

The **FindObject** method is used to locate a service object that has been explicitly registered with the Forte name service.

FindObject (**name**=*string*)

Returns Object

Parameters	Required?	Input	Output
name	●	●	

The method returns the service object which has been registered with the Forte name service.

name parameter

The **name** parameter is the name under which the service object is registered in the Forte name service.

An Introduction to The TOOL Language

The sections that follow introduce TOOL (Transactional Object-oriented Language) and describe the features that are useful in Fusion process definition methods. For a complete description of the TOOL language, see the *Forte TOOL Reference Manual*.

TOOL Language Elements

The language elements of TOOL that are immediately useful in process definition methods are:

- statements
- comments
- names
- data types
- comparison, logical, and arithmetic operators
- variables
- named constants
- fixed arrays

TOOL Statements and Comments

Forte methods are composed of TOOL statements and comments.

TOOL is not case sensitive. You can enter statements, comments, and other language elements in uppercase, lowercase, or any combination of the two.

Statements

A TOOL statement must end with a semicolon. You can start TOOL statements anywhere on a line. Although you can put multiple statements on one line, your code will be more readable if you begin each statement on a new line.

You can use newline characters (line breaks) almost anywhere in TOOL code, but you must avoid line breaks in the middle of identifiers, constants, or operators.

For example, the following code has two statements in it:

```
myAge : integer = 104;
return (age > myAge);
```

Statement Blocks

There are TOOL statements that are not simple single statements like those in the previous example, but rather start a *statement block* with one or more statements in it (for example, an **if** statement). Statement blocks are significant because they determine the scope for any variables or constants that are declared in them.

The statement that starts a statement block is not in that statement block. (It is not at the same level of scope.)

The following example shows a method statement block with a simple statement, a statement block that starts with a **for** statement, and a statement that is in the statement block.

1st statement
2nd statement--starts a block
Statement in block

```
-- A for loop to calculate factorial
j : integer = procAttr;
for i in 1 to 10 do
  j = j  * i;
end for;
```

Comments

You can use two types of comments in methods: single-line comments and block comments. Comments are ignored when the code is compiled and executed.

Single-Line Comments

Single-line comments begin with the characters "--" or "//" and end with the end-of-line character. The system ignores all characters between these two delimiters.

Example:
single-line comments

```
// Here is a comment
-- Here is another style comment.
x = 10;         // Comment at end of line
j : integer;    -- Another comment at end of line
```

Block Comments

Block comments begin with "/*" and end with "*/". The system ignores all characters between these two delimiters. Block comments can span any number of lines.

Example:
block comment

```
/*  The start of a long multi-line comment that
  goes for several lines.  The next statement:
    x = 10;
  is not executed because it is part of the comment  */
```

A block comment can contain a single-line comment or another block comment, allowing you to nest block comments. If you nest a block comment in another block comment, be sure to add the final "*/" characters to end the main comment, or you will comment out more than you intend to. The following example shows nested block comments:

Example:
nested block comment

```
/*  Start first block comment
  /* Start second block comment
  End second block comment */
End first block comment */
```

Names

A name contains alphanumeric characters and underscores. The first character must be an alphabetic character or an underscore. Case is not significant, so *myName*, *MyName*, and *myname* are all the same name. The name can be any length. When you create a new name, it must be unique for the current scope (see the next section for a description of scopes).

Restrictions

A name can have no spaces or symbols except the underscore. You cannot use TOOL reserved words (see "TOOL and SQL Reserved Words" on page 220) and you should avoid using SQL reserved words (see "SQL Reserved Words" on page 220). Also, you cannot start any name with "forte" and you cannot end any name with "proxy".

In most TOOL statements, when you want to reference a component, you simply type its name. However, when you are using SQL statements, you may need to preface a Forte name with a colon in order to distinguish it from a column name.

Use double quotes for names that are reserved words

If you must create a name whose name is the same as a reserved word, you must enclose the name in double quotes.

Scope

The scope of variables is determined by statement block. A variable declared in a statement block is *in scope* from the point at which it is declared to the end of the statement block. It stays in scope as long as TOOL is still in that statement block or is in another statement block that is embedded in that statement block.

TOOL searches for a name starting in the current scope and moving out to the enclosing scopes. A name declared in an inner scope can hide the same name declared in an outer scope. Furthermore, a name declared in an inner scope is not available in the outer scope.

In the following code example, there are two scope errors:

- The code attempts to use the variable *i* outside the for loop. Its scope is only the for loop's statement list, so trying to use it in the main method body produces a compilation error.
- It mistakenly redeclares the variable *saveit* in an inner statement block. It masks the variable *saveit* in the outer block, so setting it to TRUE does not affect that variable. When the for loop exits., the outer variable *saveit* is always FALSE.

Example: name resolution

```
CountLoop (maxAllowed:integer): integer
  saveit : boolean = FALSE;
  count : array [maxAllowed] of integer;
  for i in 1 to maxAllowed do
    count[i] = maxAllowed + i;
    if (count[i] > 50) then
      saveit : boolean = TRUE;
      exit;
    end if;
  end for;
  if (saveit) then // Error: saveit always FALSE in outer scope
    return count[i];// Error: i not recognized in outer scope
  else
    return count[maxAllowed]; // Never gets executed
```

new scope

new embedded scope

saveit is redeclared

back in outer (method) scope

Simple Data Types

You use the simple data types described in the next few sections to handle string, boolean, and numeric data. The numeric data types are discussed together because they can be combined in numeric expressions.

Advantages of classes over simple data types

Forte provides an "object" version of each simple data type. The advantage of using an object to store data is that the class provides methods for manipulating the data. Forte also provides classes specifically for storing and manipulating dates and times, time spans, and images. See the Framework Library online Help for general information about using the class data types and reference information on the subclasses of the DataValue and DataFormat classes.

String Data Types

The string data type stores a character string. This data type is very simple. You use a string constant to specify a string value, but there are no string expressions. Although you can compare strings in boolean expressions (see "Boolean Data Type" on page 203), you cannot manipulate strings. If you need to manipulate strings, use the TextData class, which provides many text handling methods and a text string of unlimited size.

To declare a data item with the string data type, use the **string** key word. For example:

```
s : string;
name : string = 'Jones';
```

The default value of a string data item is NIL, which means that it contains no string.

Also see the "Char Data Type" on page 203.

String Constants

A string constant is any series of characters enclosed by single quotes.

Example: string constants

```
s : string;
s = 'Jones';
if s = 'Smith' then
  ... will not execute ...
end if;
```

To specify an empty string, use two single quotes with no characters between them. The following table specifies how to enter special characters in a string:

Special Character	How to Enter It
'	\'
\	\\
new line	\n
carriage return	\r
tab	\t
alert (bell)	\a
backspace	\b
formfeed	\f
vertical tab	\v
octal value	\000, where 0 is 0-7
hexadecimal value	\xhh, where h is 0-9, A-F, or a-f (the character with the hex value)

Char Data Type

The char data type contains a single byte of data. TOOL provides this data type so that you can create a C-like character string—an array of char data items.

You use a string constant to set the value of a char variable. Be careful not to use a string constant that contains a string: char variables can be only to single-byte strings. The following example shows how to set a char variable:

Example: assigning string constant to char variable

```
myCharValue : char = ‘a’;
myCharEscapeSequence : char = ‘\n’;
-- ‘\n’ is new line escape sequence, a single-byte constant

-- the following assignment is invalid: ‘ab’ contains two bytes
myCharValue = ‘ab’;
```

Compilation error

TOOL automatically converts char data to integer data when a char value is assigned to an integer variable, but it does not allow you to assign string constants larger than one byte (more than one character) to integer values. TOOL also automatically converts integer data to char data when an integer value is assigned to char variable, as shown in the following example:

Example: converting char to integer

```
c : char;
i : integer;
-- 'a' is a char constant
c = 'a';
-- Convert char to integer value 97 (ASCII equivalent of ‘a’)
i = c;
-- Convert integer to char value ‘a’ (ASCII equivalent of 97)
c = i;
```

Boolean Data Type

The boolean data type contains two logical values, the boolean constants TRUE and FALSE. Use the boolean data type when a data item has only two values (such as true and false, yes and no, on and off). To declare a data item with the boolean data type, use the **boolean** key word, as in the following example:

```
test : boolean;
test = FALSE;
test2 : boolean = TRUE;
```

The default value for a boolean data item is FALSE.

Boolean Expressions

Boolean expressions are expressions that resolve to a logical value of TRUE or FALSE. You use boolean expressions to specify the conditions for several TOOL programming statements.

The following types of boolean expressions are described in the next sections:

Comparison expression Uses a comparison operator to compare two values (numeric, string, pointer, or object) and produce a value of TRUE or FALSE.

Logical expression Uses a logical operator with one or two boolean values and returns a value of TRUE or FALSE.

Comparison Expressions

A comparison expression uses a comparison operator to compare two numeric, two string, or two object values and produce a value of TRUE or FALSE. The numeric or string values can be constants, variables, attributes, named constants, expressions, and methods that return an appropriate value. The following table describes the comparison operators.

Operator	Meaning	Description
=	Equals	Result is TRUE if left side is equal to right side. Defined for numeric data types, strings, pointers, and objects. Two pointer values are equal if they contain the same address. Two object values are equal if they reference the same object.
<>	Not equals	Result is TRUE if left side is not equal to the right side. Defined for numeric data types, strings, pointers, and objects.
<	Less than	Result is TRUE if left side is less than the right side. Defined for numeric data types and strings.
>	Greater than	Result is TRUE if left side is greater than right side. Defined for numeric data types and strings.
<=	Less than or equal to	Result is TRUE if left side is less than or equal to right side. Defined for numeric data types and strings.
>=	Greater than or equal to	Result is TRUE if left side is greater than or equal to right side. Defined for numeric data types and strings.

The following code fragment uses comparison expressions:

Example: comparison expressions

```
x : integer = 10;
if x < 100 then
  --... this will be executed ...
end if;
```

Use comparison expression to produce a boolean value

```
test : boolean;
test = x < 100;
if test then
  --... this will be executed ...
end if;
```

Logical Expressions

A logical expression uses a logical operator either to negate a boolean value or to compare two boolean values and produce one boolean value, TRUE or FALSE. The boolean values you can use in the expression include comparison expressions, logical expressions, boolean constants, boolean variables, boolean attributes, and methods that return a boolean value.

The following table describes the logical operators.

Operator	Description
not	Negates one boolean value. If the value is TRUE, **not** produces FALSE. If the value is FALSE, **not** produces TRUE.
and	Result is TRUE if both values are TRUE. If one or both values are FALSE, the result of the expression is FALSE.
or	Result is TRUE if either value is TRUE. The expression is FALSE only if both values are FALSE.

The following code fragment uses logical expressions:

Example: logical expressions

```
if not x > 10 then
  ...
end if;
if (x > 10) or (x < 0) then
end if;
  ...
if (x > 10) and (y < 100) then
  ...
end if;
```

The **and** and **or** operators evaluate both operands only if necessary. In the following example, if foundIt is FALSE, then x > maxCount is not evaluated:

```
if foundIt and x > maxCount
```

Operator precedence

The logical expression is evaluated with the following operator precedence (see "Numeric Expressions" on page 208):

Precedence	Operator
1	arithmetic and address operators
2	comparison operators
3	bitwise operators (see "Numeric Expressions" on page 208)
4	not
5	and
6	or

The following sample code shows operator precedence:

Example: operator precedence

```
x : integer = 1;
y : integer = 0;
if  x + y > y - x or not x > y then
  ...same as...
if  ((1+0) > (0-1)) or (not (1>0)) then
  ... or ...
if (1 > -1) or (not (TRUE)) then
  ... or ...
if (TRUE) or (FALSE) then
  ... or ...
if TRUE then
```

Parentheses in expressions

Use parentheses to guarantee the order of evaluation. TOOL evaluates the expressions in the innermost parentheses first.

Example: parentheses in an expression

```
if (((x > y) or (y < 2)) and (x > 2)) then
  ...
end if;
```

You can also use parentheses with the **not** operator. Doing so negates the entire expression in parentheses. The following code negates the entire expression from the previous example:

Example: parentheses with **not**

```
if  not (((x > y) or (y < 2)) and (x > 2)) then
  ...
end if;
```

Numeric Data Types

The numeric data types allow you to store integers and floating point numbers of different sizes. This section describes the integer and float data types and provides general information on numeric constants and numeric expressions.

Integer Data Types

Of the integer data types, only some are guaranteed to be portable because they have the same range on every platform. The non-portable types use different representations on different machines. Keep this in mind when you declare integer data items. The following table lists the integer data types and indicates whether or not they are portable:

Key Word	Description	Portable
int	At least -32,768 to +32,767.	no
long	At least -2,147,483,648 to +2,147,483,647.	no
short	At least -32,768 to +32,767.	no
i2	Signed two byte integer. Exactly -32,768 to +32,767 all platforms.	yes
ui2	Unsigned two byte integer, 0 to +65,535.	yes
integer or i4	Signed four byte integer. Exactly -2,147,483,648 to +2,147,483,647 on all platforms.	yes
ui4	Unsigned four byte integer, 0 to +4,294,967,295.	yes
i1	Signed one byte integer, -128 to +127.	yes
ui1	Unsigned one byte integer, 0 to +255.	yes

To declare a data item of an integer data type, use the appropriate key word. For example:

```
i : integer;
j : short = 32;
```

The default value for an integer data item is 0.

Float Data Types

TOOL supports two float data types whose exact precision depends on your particular machine. The float data types are:

Key Word	Description
float	Approximately 10E-38 to 10E+38, with about 7 digits of decimal precision.
double	Approximately 10E-308 to 10+308, with about 15 digits of decimal precision.

If you want to ensure that your code is completely portable, you should only use the precision that is available on all the machines you plan to use. For precise decimal behavior, you can use the DecimalData class (see the Framework Library online Help for information).

To declare a data item with the float data type, use the appropriate key word.

Example: float types

```
i : float;
j : double = 10;
pi : double = 3.14159268;
```

The default value for a float data item is 0.0.

Numeric Constants

Integer constants

An integer constant is a sequence of digits between 0-9. No other characters are allowed. To indicate a negative number, use a minus sign. A number without a sign is considered positive but you can use a plus sign if you wish. The syntax is:

[+|-]*digits*

```
x : integer;
x = 10;
x = -23;
x = +43;
```

Hexadecimal and octal integers

You can use hexadecimal or octal constants to specify an integer. The syntax for hexadecimal integers is:

0x*hexdigit*

where *hexdigit* is:

0-9, A-F, or a-f (the character with the hex value)

The syntax for octal integers is:

0*octaldigit*

where *octaldigit* is any digit from 0-7. The first non-octal digit terminates the number.

```
x = 0x20;
x = 0xff04;
x: integer = 011; -- returns a value of 9
x: integer = 08; -- returns a value of 0
```

Float or double constant

A float constant is a sequence of digits 0-9 with a single decimal point (.). You can also include an exponent. To indicate a negative number, use a minus sign. A number without a sign is considered positive but you can use a plus sign if you wish. The syntax is:

[+|-]*digits.digits*[**e**|**E**[+|-] *integer*]

Example: numeric constants

```
y : double;
y = 10;
y = -123.456;
y = -1.3e+12;
```

Numeric Expressions

A numeric expression combines two numeric values with an arithmetic operator to produce one numeric value. The numeric values you can use include numeric constants, numeric variables, numeric attributes, methods that return a numeric value, and numeric expressions.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (integer results are truncated, not rounded)
- (unary)	Negative
+ (unary)	Positive
%	Mod
&	bitwise and
|	bitwise or
^	bitwise exclusive or
~	bitwise (unary)

Order of evaluation

Numeric expressions are evaluated from left to right, with the following operator precedence (from highest to lowest):

Precedence	Operator
1	[ ] ->
2	* (pointer dereference) & (address of)
3	- (unary) + (unary) ~
4	* / %
5	+ -
6	< > = >= <= < >
7	& (bitwise)
8	^ (bitwise xor)
9	| (bitwise or)
10	not
11	and
12	or

The following example illustrates the order of evaluation in a numeric expression:

Example: evaluation of numeric expression

```
x : integer = 1;
y : integer = 2;
z : integer;
z = x + y * y;   -- Evaluates to 5
```

Use parentheses to guarantee the order of evaluation. TOOL evaluates the expressions in the innermost parentheses first.

Example: parentheses in numeric expression

```
x : integer = 1;
y : integer = 2;
z : integer = (x + y) * y;
-- Evaluates to 6
```

Expression data type

The data type of a numeric expression's result is determined by the data type of both operands (left and right). The following table shows the data type for each possible pair of operands. Since the table is symmetric, the rows and columns can correspond to either the left or right operand:

	double	**float**	**ui4**	**long**	**integer/i4**	**int**	**i2**	**i1**	**ui2**	**ui4**
double	double	double	double	double	double	double	double	double	double	double
float	double	float	float	float	float	float	float	float	float	float
ui4	double	float	ui4	ui4	ui4	ui4	ui4	ui4	ui4	ui4
long	double	float	ui4	long	long	long	long	long	long	long
integer/i4	double	float	ui4	long	integer	integer	integer	integer	integer	integer
int	double	float	ui4	long	integer	integer	integer	integer	integer	integer
i2	double	float	ui4	long	integer	integer	integer	integer	integer	integer
i1	double	float	ui4	long	integer	integer	integer	integer	integer	integer
ui2	double	float	ui4	long	integer	integer	integer	integer	integer	integer
ui1	double	float	ui4	long	integer	integer	integer	integer	integer	integer

TOOL automatically converts the value whose data type is different than the result type. This takes effect before the operation is executed.

Casting numeric types

Because TOOL automatically performs conversions for the numeric values in expressions, you normally do not need to cast numeric types. However, if you need more control, you can cast the numeric types to the correct type. The syntax is:

numeric_type (*expression*);

or

(*numeric_type*) (*expression*);

For example, normally the result of adding a float and an integer is a float. However, if your code requires that the result be an integer, you can cast the expression, forcing it to be rounded to the nearest integer. For example:

Example: casting numeric expression

```
x : integer = 1;
y : float = 2.14159;
z : integer = integer (x + y);
-- Evaluates to 3
```

Variables

A variable is a name used to refer to a single data item. Every variable has a data type. It can have a simple data type or a class type. If the variable has a numeric, boolean, or string data type, the variable itself contains the data. If the variable has a class type, the variable points to the object or objects that contain the data. (See the Forte *TOOL Reference Manual* for more information on classes, variables, and objects.)

You must declare a variable before you can reference it. After you have declared the variable, you can assign a value to it and include it in expressions.

Declaring a Variable

You can declare a variable anywhere within your TOOL code. You must specify the variable name and type. Optionally, you can specify an expression that sets the initial value of the variable.

variable_name [**,** *variable_name*]… **:** *type* [= *expression*]

The scope of the variable is from the point where you declare it until the end of the current statement block. If you declare it at the start of a method, its scope is for the entire method.

Variable name

The variable name identifies the variable for use within the current statement block. It can be any legal TOOL name and must be unique for the current block. If the variable name is the same as the name of a variable declared in an enclosing statement block, the new variable will "hide" the existing variable. (See "Scope" on page 201 for more information.)

You can declare multiple variables in a single definition, simply by specifying more than one variable name. TOOL creates a separate variable for each name, using the same type and initial value. For example:

Example: declaring two variables

```
i, j : integer = 0;
-- Three new objects
t, u, v : TextData = new(value = 'x');
```

Variable type

The variable type can be any simple type, any class (or interface), or any array. A variable declared to be of a class type is also called an *object* or an *instantiation* of the class. If you are using classes in your methods other than the objects supplied by Fusion, see the Forte *TOOL Reference Manual* for more information on using classes and objects in TOOL.

Initial value (expression)

The *expression* part of the previous variable declaration is the initial value of the variable. When you declare a variable, it is often a good idea to also give it an initial value, which can be any value that is compatible with the data type of the variable. If the variable is a simple data type and you do not specify the initial value, it has the default value for the data type. If the variable is a class type and you do not specify the initial value, it has a default value of NIL.

Default values for simple variables are shown in the following table:

Type	Default Initial Value
boolean	FALSE
char	NIL
double	0.0
integer (all types)	0
float	0.0
string	NIL

Assigning a Value to a Variable

To assign a value to a variable, use the following syntax:

variable = *expression*

The expression can be any value that is compatible with the data type of the variable. For example:

Example: assigning a value to a variable

```
i : integer = 0;
j : integer = 10;
j = j + i;
```

Named Constants

A named constant (as opposed to a literal constant, such as a string constant like 'This a string' or a numeric constant like 3.14159) is a literal string or numeric value that has a name. To declare a named constant, you specify a constant name and a value. You can then use the name in place of the value in your TOOL code. TOOL uses three kinds of named constants: project constants, class constants, and local constants. (For information on project constants and class constants, see the Forte *TOOL Reference Manual*.

Local constants

A local constant is a named constant that can be accessed only within the current statement block. You must declare a local constant in your TOOL code before you can reference it. After you have declared the local constant, you use it in the current statement block.

Declaring a Local Constant

You can declare a local constant anywhere in your TOOL code with the **constant** statement. You must specify the constant name and value. The syntax is:

constant *constant_name* = *value*

As with a variable, the scope of the constant is from the point where you declare it until the end of the current statement block. If you declare it at the start of your method, its scope is for the entire method. The following example illustrates declaring a local constant:

```
constant pi = 3.14159268;
constant SECONDS_PER_HOUR = 3600;
```

Constant name

The constant name identifies the constant for use within the current statement block. It can be any legal TOOL name and must be unique for the current block. If the constant name is the same as the name of a global constant, global variable, or a data item declared in an enclosing statement block, the new named constant will "hide" the existing data item.

Constant value

You declare either a numeric or string value for the named constant. The data type of the value determines the data type of the constant. After you specify the value for a constant, you cannot change it.

Referencing a Named Constant

To reference a named constant, use the constant name. For example:

Example: referencing a named constant

```
circumference, radius : double;
radius = 2.0;
constant PI = 3.14159268;
circumference = 2 * pi * radius;
```

You can use a named constant to specify a value anywhere in the TOOL code as long as the data type meets the requirements of the expression.

If the named constant is a class constant that is not in the current class, other classes must reference the constant with the following syntax:

class_name.constant_name

Using Named Constants in Expressions

Because named constants represent literal values, you can use them in any expression where a literal value is appropriate.

However, because named constants are read-only values, you cannot assign a value to them. This means you cannot pass a named constant to a method as an output parameter or use it on the left side of any assignment statement.

The following example illustrates the use of named constants in an expression:

Example: named constants in expressions

```
perimeter, radius : double;
radius = 2.0;
constant PI = 3.14159268;
perimeter = 2 * pi * radius;
-- following line is ERROR: CANNOT ASSIGN CONSTANT.
PI = 3.14159268;
```

Fixed Arrays

A fixed array in TOOL is an array of a predetermined size that contains values of the same simple data type. ("Simple Data Types" on page 202 describes these data types.) For example, the following code sample declares an array with ten elements of type integer and sets the first element to 1:

```
myIntArray [10] of integer;
myIntArray [0] = 1;
```

Usually a fixed array of simple data types is all you need in a process definition method.

Note TOOL also supports arrays of objects. If you plan to use this type of array, you must know more about using TOOL. See the Forte *TOOL Reference Manual* for information on using arrays of objects.

The following table shows two varieties of syntax for declaring a fixed array. The brackets "[" and "]" represent characters that are part of the syntax, and the ". . ." characters indicate that more dimensions can be added to the array.

name **: array** [*lower***..***upper*] … **of** *data_type***;**
name **: array** [*lower***..***upper, …*] **of** *data_type***;**

- *name* is the variable name for the array.
- *lower* and *upper* define the lower and upper bounds of the array. Their values must be integer constants. A value for the *lower* bound is not required, but if you specify it, the value must be lower than the value of the corresponding *upper* bound.
- *data_type* is the name of the simple data type for all the elements of the array

Note Unless you specify the lower bound of the array, the numbering of the array elements starts at 0.

Using the first variation of the syntax, you can specify arrays as shown in the following examples:

Example: first kind of array syntax

```
-- Declare a one-dimensional array with 10 integer
-- elements numbered 0 to 9:
myArray1 : array [10] of int;
-- Declare a one-dimensional array with 10 char
-- elements numbered 1 to 10:
myArray2 : array [1..10] of char;
-- Declare a two-dimensional array containing 5 arrays, each with
-- 8 float elements whose element numbering starts at 3.
myArray3 : array [5][3..10] of float;
```

Using the second variation of the syntax, you can specify multidimensional arrays as shown in the following example:

Example: second kind of array syntax

```
-- Declare a two-dimensional array containing 6 arrays of 8 integer
-- elements whose element numbering starts at 0
myArrayA : array [6, 8] of int
-- Declare a two-dimensional array containing 5 arrays, each with
-- 8 elements whose element numbering starts at 3
myArrayB : array [5, 3..10] of float
```

TOOL Statements for Methods

The previous section covered basic elements of the TOOL language, but did not describe specific TOOL statements for flow control, looping, and so on. This reference describes the TOOL statements that are useful in process definition methods. For a complete TOOL reference, see the Forte *TOOL Reference Manual*.

case

The **case** statement is similar to a set of **if…then…elseif** statements. Instead of **if** and **elseif** statements, it has a set of statement blocks, each identified by a different integer. The **case** statement evaluates an integer expression, then executes the statement block that matches that number, if any.

Syntax

case *integer_expression* [**is**]
    [**when** *value* **do** *statement_block*]…
    [**else** [**do**] *statement_block*]
end [**case**]**;**

Example

GetState returns integer value

```
t : string;
case GetState()
  when ACTIVE do
    t = 'I\'m busy.';
  when READY do
    t = 'I\'m ready if you are.';
  else do
    t = 'I\'m finished.';
end case;
```

Description

Note

This description covers only functionality that is generally useful in Fusion process definition methods. For a complete description of the **case** statement, see the Forte *TOOL Reference Manual*.

When the **case** statement starts, its **case** expression is evaluated, then each **when** clause is evaluated to see if its value is equal to the **case** expression. If there is a match, the corresponding statement block is executed, then the **case** statement exits. If there is no match and there is an **else** clause, that statement block is executed, then the **case** statement exits. If there is no **else** clause, the **case** statement simply exits.

Expression

The **case expression** must be an integer expression. The data type of the corresponding values in the **when** clauses must also be integer.

when clause

The **when** clause specifies one integer value that is a possible result of the expression and provides a statement block to be executed for that particular value. The value for each **when** clause must be an integer constant.

If you specify the same value in more than one **when** clause, you get a compile time error.

statement block

The statement block for a **when** clause can include any TOOL statements. You can use an **exit** statement in the statement block to exit from the **case** statement.

Example: use of **exit** statement with **case**

```
case ...
  when 1 do
    if ...condition... then
      exit;
      -- Exits case statement.
    end if;
  when 2 do
    ...
end case;
```

constant

The **constant** statement declares a named constant with a scope from the point you declare it to the end of the block.

Syntax

constant *name* = *value* **;**

Example

```
constant PI = 3.14159268;
constant seconds_per_hour = 3600;
constant COMPANY_NAME = 'Forte Software Inc.';
```

Description

The **constant name** can be any legal TOOL name and must be unique for the current statement block. Because constants share the same name scope as several other components, if the constant has the same name as a component in an enclosing scope, the new named constant will "hide" the existing component. See "Scope" on page 201 for information on name resolution.

The **constant value** can be any numeric or string value. The data type of the value determines the data type of the constant. After you specify the value for a constant, you cannot change it.

for

The **for** statement is a loop that repeats a statement block for each number in a range of numbers.

Syntax

for *variable_name* **in** *first_value* **to** *second_value*
    [**by** *step_expression*]
    do *statement_block*
end [**for**]**;**

Example

```
j : array [10] of integer;
for i in 0 to 9 do
  j[i] = i;
end for;
```

Description

Note — This description covers only functionality that is generally useful in Fusion process definition methods. For a complete description of the **for** statement, see the Forte *TOOL Reference Manual*.

The **for** statement uses a numeric variable to loop through its statement block a set number of times. At the start of each loop iteration, the variable is incremented (or decremented) and is compared to the second value in the range. When the variable is no longer in the range, the **for** statement exits.

Declaring the variable — Use a new variable name for the loop control variable. (You cannot use an existing variable.) TOOL automatically declares a new variable whose scope is limited to the **for** statement. The variable's type is automatically set to the same type as the numbers in the range.

The range — To specify a range, you must use either integers or floating point numbers and enter a first value and a second value. By default, the numbers are incremented by 1 (one) each time through the loop. The optional **by** clause allows you to specify a step value to be used for calculating the loop control value each time through the loop. The step value can be a positive or negative number. If the step value is positive, the first value must be lower than the second value. If the step value is negative, the first value must be higher than the second value.

The statement block — The statement block can include any TOOL statements. You can use the **exit** statement to transfer control to the statement after the **end for**. You can use the **continue** statement to force a new iteration of the statement block and assign the loop control variable the next value in the range.

if

The **if** statement executes a statement block when the specified boolean condition is true.

Syntax

if *boolean_expression* **then**
    statement_block
[**elseif** *boolean_expression* **then**
    statement_block]…
[**else** [**do**]
    statement_block]
end [**if**]**;**

Example

```
if (amount <= 0) then
  ErrorCondition = TRUE;
elseif (amount < 500) then
  return TRUE;
else // amount is >= 500
  return FALSE;
end if;
```

Description

Note This description covers only functionality that is generally useful in Fusion process definition methods. For a complete description of the **if** statement, see the Forte *TOOL Reference Manual*.

At most, only one of the statement blocks in the **if** statement can be executed. Each boolean condition is evaluated in order, starting with the **if** condition and continuing if necessary with any **elseif** conditions. Evaluation stops when a true condition is found; that statement block is executed, and the **if** statement exits.

If no conditions are true and there is an **else** condition, its statement block is executed and the **if** statement exits. (The **else** clause specifies a statement block to be executed when all the conditions are false.) If there is no **else** condition, the **if** statement exits without executing any statement block.

Note There is no space in the **elseif** keyword.

The simplest version of the **if** statement specifies one condition to be tested and one statement block to be executed if the expression is true. For example:

Example: simple **if** statement

```
i : integer = 10;
if i > 5 then
  return TRUE;
end if;
```

Boolean Expressions

The expressions in the **if** statement must be boolean expressions (see "Boolean Expressions" on page 203). These expressions can include boolean variables, constants, attributes, and methods that produce a boolean return value. For example.

Example: boolean expression

```
if ((i > 10) and (i < 100)) or not (j > 4) then
  rtnVal = TRUE;
end if;
```

Statement Blocks

The statement blocks in an **if** statement can include any TOOL statements. Note, however, that you cannot use **exit** or **continue** in an **if** statement to close or repeat the **if** statement blocks. Including **exit** or **continue** in an **if** statement block causes control to pass to the closest enclosing loop statement. If there is none, you get an error when you compile the method.

return

See "The return Statement" on page 184.

while

The **while** statement loops through its statement block as long as its boolean expression is true.

Syntax

while *boolean_expression*
  do *statement_block*
end [**while**]**;**

Example

```
i : integer = 1;
while i < maxValue do
  ...
  i = i + 1;
end while;
```

Description

Note

This description covers only functionality that is generally useful in Fusion process definition methods. For a complete description of the **while** statement, see the Forte *TOOL Reference Manual*.

At the start of the **while** loop, its expression is evaluated. If the expression is true, the statement block executes. After the last statement in the block executes, control returns to the beginning of the **while** loop, where the expression is evaluated again. This process continues until either the expression is false or the loop is exited with an **exit** statement or a **return** statement.

To ensure that the **while** statement exits, you must do one of the following:

- use a boolean condition that will eventually be false
- use an **exit** statement to exit the loop
- use the **return** statement to exit the method

If you do none of these things, the **while** statement will loop infinitely.

If you are using a boolean condition to loop through a range of numbers, be sure to increment or decrement your counter. For example:

Example: using a counter

```
x = array [maxLength] of integer;
... fill in x ...
i : integer = 1;
while i <= maxLength do
...process x[i]...
  i = i + 1;
end while;
```

Boolean Expression

The boolean expression specifies a logical condition that has a value of TRUE or FALSE. It can include boolean variables, constants, attributes, and methods that return boolean values. See "Boolean Expressions" on page 203 for information on boolean expressions. Here is an example:

Example: boolean expression

```
while ((i < 10) and (j > 4)) and not (k = 3) do
  ...
end while;
```

Statement Block

The statement block can include any TOOL statements. You can use the **continue** statement to return to the first statement of the statement block and force another iteration of the loop. You can use the **exit** statement to pass control to the statement following the **end while**.

Example: **exit** and **continue** in statement block

```
while TRUE do
  ...processing...
  if continueOK = TRUE then
    continue;
  else
    exit;
  end if;
end while;
```

TOOL and SQL Reserved Words

TOOL Reserved Words

and	exception	loop	return
attribute	exit	method	service
begin	false	new	sl
case	for	nil	start
changed	forward	not	struct
class	from	of	super
constant	handler	output	task
continue	has	post	then
copy	if	postregister	to
cursor	implements	preregister	transaction
do	in	private	true
else	includes	property	typedef
elseif	inherits	public	union
end	input	raise	virtual
enum	interface	register	when
event	is	method	‘where

SQL Reserved Words

all	desc	group	procedure
any	distinct	having	raise
as	escape	immediate	revoke
asc	execute	insert	select
between	exists	into	session
by	extend	like	set
close	extent	minus	some
connect	fetch	null	unique
current	fragment	on	update
default	from	open	values
delete	grant	order	where

Appendix A

Fusion Process Management Examples

This appendix describes the Fusion process management example applications and the sample Organizational Database Access application, which are provided with your Fusion installation. The examples are in the following location of your Fusion installation:

```
$FORTE_ROOT/install/examples/conductr
```

Typically, you run an example application, then examine it in various process development workshops to see how the process definitions are implemented. You can then examine the application source files to see how the application is implemented. You can modify the examples to experiment with Fusion features or to create your own applications based on the examples. It is recommended that you modify private copies of the example files, keeping the original examples intact.

The Fusion examples provide process definitions for both a basic and advanced expense reporting application. Each example illustrates how to use one of the Fusion process client APIs (TOOL, CORBA/IIOP, C++, and ActiveX).

The instructions to the examples assume familiarity with the Forte Application Environment, and also, some familiarity with the Fusion process development workshops and the Fusion Console. The *Forte Fusion Process Development Guide* provides information on the process development workshops. For information on the Fusion Console, refer to the *Forte Fusion Process Management System Guide*. For information on the Fusion process client APIs, refer to the *Forte Fusion Process Client Programming Guide*. Online help is also available for Fusion.

This appendix provides:

- instructions on how to install the examples
- a brief overview of the example applications
- a section describing each example in detail

Installing Fusion Example Applications

Before running the Fusion examples, you must first configure and start a Fusion process engine, import the example .pex files into your repository, and register the example Fusion plans with the Fusion engine.

The procedure for creating and starting an engine is described in the following section, "Configuring and Starting an Engine." Each example comes with a script which automatically imports the .pex files and registers the Fusion plans with the engine. The section for each example provides instructions on using the script and running the example.

Configuring and Starting an Engine

The example applications communicate with a Fusion process engine called "ceengine." The procedure for configuring and starting this engine is described below. For additional information on Fusion process engines, refer to the *Forte Fusion Process Management System Guide* and Fusion online help.

Note If you do not have access to a Fusion engine named ceengine, refer to "Using Alternate Engines" on page 223 for information on how to modify the examples to communicate with another engine.

To configure a Fusion process engine:

1. Start Fusion Console.
2. Select the **Engine** > **New** command to open the Configure New Engine dialog.
3. In the Name tab page, enter ceengine in the Engine Name field.
4. In the Database tab page, fill in the database information fields.

   You must have a valid database connection to configure a Fusion engine.
5. In the Logging tab page, select the History Log categories you want to record.

   If you enable all logging, a significant amount of data can be written to your engine database. For a minimal configuration, disable the current state and history logs. If you want to enable failover or specifically look at the history, you can always enable logging later.
6. In the Components tab page, insert at least two components: an engine unit, and a database service.

   You can name these components whatever you like.
7. Click **Create** to create the engine configuration file.
8. In the Fusion Console main window, select ceengine, and then select the **Engine** > **Start** command.

   The Start Engine window appears.
9. Select Cold in the Startup Options field, then click **Start**.

   It takes a few moments for all the engine components to start. The state column indicates when each component is online. You can close the Start Engine window after both components are online.

Caution Cold starting an engine initializes the logging, state, and registration information for the engine. After shutting down the engine, you typically use a warm start to restart the engine. Otherwise you lose all of the engine's logging, state, and registration information.

10 Verify that the engine started successfully, as follows.

In the Fusion Console main window, select ceengine, and then choose the **Engine > Status** command. Examine the information in each node in the treeview to make sure the engine startup succeeded.

Note If the engine startup fails, check the Engine Partition Log for each component (available from the Engine Status window) for problem descriptions. You can also check any recent files in the log directory, located at $FORTE_ROOT/log.

You can now exit Fusion Console. However, you may find it useful to have Fusion Console running while you run the examples. The Fusion Console allows you to monitor sessions and processes and also provides other useful information.

Importing, Distributing, and Registering Examples

Each example contains a Conductor Script (Cscript) file. Cscript files are identified by the filename extension *.csc*. Executing an example Cscript imports the .pex files for the example, makes library distributions, and registers Fusion plans with the engine named ceengine.

The sections for each example provide specific instructions on how to load and run the example. You can load all the examples into the same development repository, and in any order. The Cscript file provided with each example attempts to create the workspace ConductorExamples. It then attempts to load the Fusion libraries into the ConductorExamples workspace.

Note If the ConductorExamples workspace has been previously created by one of the example scripts, error messages inform you that the workspace and the libraries already exist. You can ignore these messages.

After you run the Cscript file provided for an example, you can run the example without making any manual registrations. However, if you have more than one example loaded, and want to switch back and forth between them, you must reregister the User Validation plan for the example you are about to run. This is because you can only register a single User Validation plan with a Fusion engine at one time.

Using Alternate Engines

If a Fusion engine named ceengine is not available to you, you can use another Fusion engine. However, you must edit the example Cscript file so it refers to the correct Fusion engine. In the Cscript file, search for “ceengine” and replace it with the name of the engine you are using. Additionally, you need to set the environment variable FORTE_EP_ENGINE_NAME to the name of this engine. The client code for the examples checks this environment variable when it attempts to connect to an engine. If FORTE_EP_ENGINE_NAME is not set, it defaults to ceengine.

Overview of Fusion Process Management Examples

The following tables provide an overview of the example applications, organized by general topic. The first column in each table shows the name of the example subdirectory under $FORTE_ROOT/install/examples/conductr. For the complete description of each example application, refer to "Application Descriptions" on page 225.

Fusion Process Management Examples

	Example	Description
er/	Expense Reporting	Illustrates how to use basic Fusion features in an expense reporting system. The client in this example is implemented with the TOOL client API.
adver/	Advanced Expense Reporting	Builds on the basic features shown in Expense Reporting. It illustrates how to reference service objects in Fusion process definition methods. Its user validation plan makes use of database tables created by the Organization Database example. It also stores its own application data in database tables. The client in this example is implemented with the TOOL client API.
jer/	JExpense	Illustrates Fusion's CORBA/IIOP client API. The client in this example is implemented in Java.
jer/	JExpenseNS	Illustrates Fusion's CORBA/IIOP client API. The client in this example is implemented in Java. It is similar to JExpense, except it uses the CORBA Name Service to communicate with the Fusion engine.
jer/	JExpenseSO	Illustrates Fusion's CORBA/IIOP client API. The client in this example is implemented in Java. It is similar to JExpenseNS—it uses the CORBA Name Service to communicate with the Fusion engine. In addition, the JExpenseSO client communicates with the service object in the expense report application through an ior file.
jer/	JExpenseNB	Illustrates the use of a NetBeans client using Fusion's CORBA/IIOP client API. This example uses the same communication mechanisms as JExpenseSO. The NetBeans client provides a browser based GUI client, using JSPs and a servlet controller.
cer/	C++ Expense Reporting	Illustrates Fusion's C++ client API. A C++ client communicates with an engine that has registrations for the Expense Reporting definitions.
vber/	ActiveX Expense Reporting	Illustrates Fusion's ActiveX client API. A Visual Basic client communicates with an engine that has registrations for the Expense Reporting definitions.

Organization Database Access

	Example	Description
orgdb/	OrganizationDatabase	This application provides a GUI for maintaining a user organization database. A Fusion process client application typically references this database using the ValidateUser method developed in the Validation Workshop. This example does not require a Fusion engine or process development workshops to run.

Application Descriptions

This section provides detailed descriptions for each example application. The section for each example provides the following information about the examples:

Description Describes the purpose of the example, the problems it solves, and the Fusion features it illustrates.

Pex Files Provides the directory and file names of the .pex project files. The examples come with a Conductor Script (Cscript) file. The Cscript file for each example does the following:

- imports the .pex files into your repository
- compiles the plans
- makes the library distributions
- registers the distributions with the engine

The following naming scheme is used to identify the .pex files for an example:

.pex File	Description
*name*_ad.pex	application dictionary
*name*_ar.pex	assignment rule dictionary
*name*_pd.pex	process definition
*name*_up.pex	user profile
*name*_uv.pex	user validation

The Conductor Script files import plans into your workspace. The script compiles the plans, creating projects. The imported plans and resulting projects use the following naming convention:

Plan	Description
*AppName*AD	application dictionary plan
*AppName*AD_AT	compiled application dictionary TOOL project
*AppName*AR	assignment rule dictionary plan
*AppName*AR_AR	compiled assignment rule dictionary TOOL project
*AppName*PD	process definition plan
*AppName*PD_PD	compiled process definition TOOL project
*AppName*UP	user profile plan
*AppName*UP_UP	compiled user profile TOOL project
*AppName*UV	user validation plan
*AppName*UV_UV	compiled user validation TOOL project

In your workspace, double-click a plan to examine it in the corresponding Fusion process development workshop. You can also examine the compiled, read-only, TOOL projects by double-clicking a project.

Additional files in an example directory are client and server files for the Forte application.

Special Requirements Identifies any special setup procedures you may need to follow.

Running the Example Explains how to step through the application's functions.

Note All the examples require access to a running Fusion engine. See "Configuring and Starting an Engine" on page 222 for information about how to start a Fusion engine.

Expense Reporting

Description This basic Expense Reporting example illustrates how to use Fusion to build a simplified expense reporting system. Expense Reporting allows different users to log on as employees, managers, auditors, and accountants. In these roles, the different users can create expense reports, review them, revise them if necessary, and perform accounting and auditing procedures on them.

This example provides Fusion plans that are used by other examples. The Fusion project, ExpenseReportClient, illustrates how to write a Fusion process client using the TOOL API.

Expense Reporting illustrates the following Fusion process management features:

- offered activities
- queued activities
- automatic activities
- routers
- subprocesses
- client API programming in TOOL
- customized user validation
- access rules
- application dictionary items
- timers
- triggers
- linked users

Expense Reporting does not illustrate the following Fusion process management features:

- organization database access
- persistent storage of application data in database
- customized user profile
- junction activity

Pex Files This example uses the following .pex files, located at $FORTE_ROOT/install/examples/conductr/er:

erad.pex	ercso.pex.
erar.pex	erpd.pex
erbc.pex	errevpd.pex
erclient.pex	erup.pex
ercsc.pex	eruv.pex

Special Requirements A Fusion engine called “ceengine” must be running. Refer to “Configuring and Starting an Engine” on page 222 for more information.

The er directory contains a Conductor Script file: er.csc. This script imports the Fusion and Forte plans into a new workspace called ConductorExamples. It then compiles, distributes, and registers the Fusion plans.

To install the Expense Reporting application:

1. If the Fusion engine named ceengine is not already running, start it.
2. From a command window, navigate to the FORTE_ROOT/install/examples/conductr/er directory and issue the following command:

```
cscript -i er.csc -o er.out
```

3. Examine er.out to make sure all the commands completed successfully.

Running the Expense Reporting Example This section describes the Expense Reporting application and shows how to run it, taking a process instance to completion.

Figure 46 shows valid users (and their roles) for the Expense Reporting application. All users are employees. Some are managers, some are accountants or auditors. Only managers can log on as reviewers. For all users, the password is the same as the user name. Case is significant.

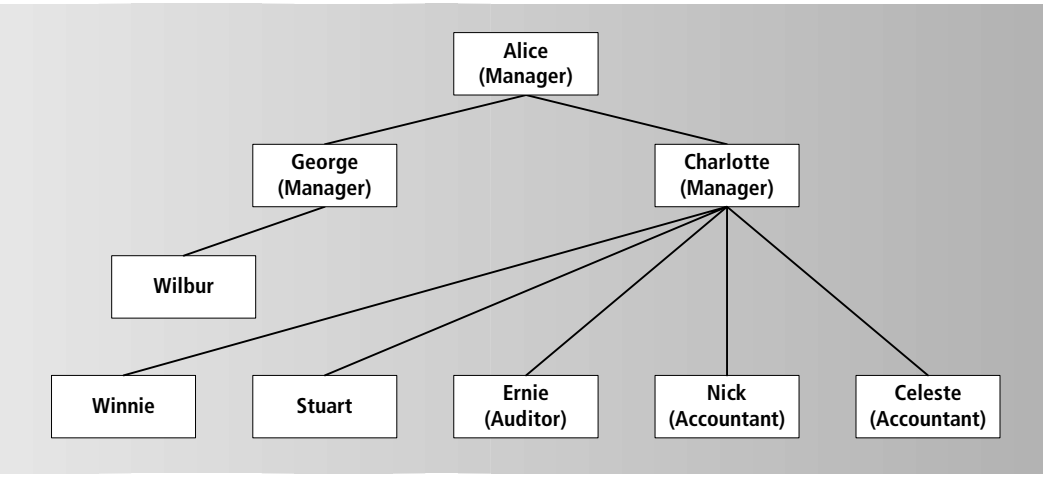


Figure 46 *Valid Users in Expense Reporting System*

To see processes go to completion, log on with at least the following roles:

- an employee
- the employee's manager (George or Charlotte)
- the second level manager (Alice)
- an accountant
- an auditor

Users can log on more than once in different roles For example, Charlotte could log on as an employee, then log on again as a reviewer. Nick could log on as an accountant, then log on again as an employee.

An additional logon option allows you to open a Monitor Window that lets you examine expense reports as they are created in the system. To open the Monitor Window, use "Forte" as the password with no user name. Any role can be selected.

After completing the procedure, you are encouraged to experiment with various processes within the application. You should also examine the Fusion plans created by the example in the Fusion process development workshops to further understand how the example works. The ExpenseReportClient project shows how the client was implemented using the TOOL API.

To run the Expense Reporting application:

1. Start Forte Distributed, and set your workspace to ConductorExamples. Select ExpenseReportClient, and click the Run icon.

   The Expense Reporting logon opens.

2. Open a number of windows:
   - Log on Winnie and Wilbur as employees
   - Log on Charlotte, Alice, and George as reviewers
   - Log on Nick as an accountant
   - Log on Ernie as an auditor
   - Log on with the password Forte to bring up the Monitor Window

3. In Winnie's Employee Expense Report Information window, click Create New Expense Report.

   The CreateExpense Report window opens. Expense Report IDs are assigned automatically, starting at 1001.

4. Enter a description for the expense and an amount greater than $1,000. Then click Submit.

   The expense report appears in the inbox of Charlotte's Expense Report Review window. This is an example of a *heads up* window associated with an offered activity.

   The item does not appear in George's inbox. Although George is a manager, he is not Winnie's manager. It also does not appear in Alice's inbox because she is not a direct manager of Winnie.

   Look at the columns for this item in Charlotte's inbox. The status is listed as Submitted and the priority is initially set to false. If the item remains in Charlotte's inbox for more than one minute, the priority is set to true.

5. In Charlotte's Expense Report Review window, select the expense report item Winnie just submitted, then click Review. Enter something in the comments field, and click Approve.

   After approval the item is no longer in Charlotte's inbox. It arrives in the inbox of Alice's Review Expense Report window. In this example application, expenses exceeding $1,000 must be approved by a second level manager. The status of the expense report has now changed from Submitted to Accepted.

6. In Alice's Review Expense Report window, select the newly arrived item and click Review. Add a remark to the comment field and then click Decline.

   The item disappears from Alice's inbox, reappearing in Charlotte's inbox with the status Rejected. In Charlotte's window, review this expense again, but this time reject it. The item disappears from Charlotte's inbox, reappearing in the inbox of Winnie's Employee Expense Report Information window. Only Winnie, and no other employee, receives this rejected expense.

7. In Winnie's window, select the expense and click the Revise Expense Report. Change the amount to less than $1,000 and click Submit Revision.

8. Charlotte receives the item again. Review the item and accept the report.

   This time Alice does not receive it in her inbox. Expense reports under $1,000 do not require second level manager approval.

9 In Nick's Accounting Window, click Get Report.

This is an example of a *heads down* window, which is associated with a queued activity. Nick cannot choose which report to work on next. He receives whatever is next in the queue.

In this case, Winnie's expense report appears in Nick's window.

10 In Nick's Accounting Window, click Work on Report.

Here, a real accountant would do some work. To keep the example simple, a message dialog comes up indicating that real work usually occurs at this point.

Click Get Report again. A message dialog informs you that no work is available.

11 In Ernie's Auditing Window, follow the same steps Nick just used to get work and complete it.

12 Now examine the Monitor Expense Reports window.

The Monitor Expense Reports window automatically places the value 1001 in the ID field. Click the Fetch button to retrieve expense report information.

In this example application, each time an activity is performed on the expense report, a tagline is written to the WorkHistory attribute on the ExpenseReport object. The Work History field of the Monitor Expense Reports window displays the contents of the WorkHistory attribute, thus listing all (but one) of the activities performed on this expense report. The exception is the automatic activity (ProcessCheck) that issues a check for the expense. This automatic activity writes to the engine partition log file rather than to the WorkHistory attribute. It waits until both the accountant and the auditor have completed their work before it triggers. The following step shows how to view the engine partition log.

As you send more expense reports through the system, you can examine the expense report object in the monitor window by changing the value in the ID field and clicking Fetch.

13 Use the Fusion Console to view the engine partition log, which includes any entries for the automatic activity.

If it is not open, open the Fusion Console, select your engine, and then select **Engine > Status**. In the Engine Status window, select the unit under Engine Components, then click **View Partition Log**.

If you completed the example up to this point, the log contains the following line:

"Check processed for expense report 1001"

14 Exit the application as follows:

Complete all the processes you start and exit all the user windows before exiting the logon window, which shuts down the application. This prevents process instances from a subsequent run from colliding with the current run. (The Advanced Expense Reporting example provides a more elegant way to shut down the application.)

Alternatively, you can use the Fusion Console to terminate processes and sessions.

In the Fusion Console, select your engine and then do either of the following:

- To terminate all processes, select **Monitor > Processes** to open the Process Instances window. Then select **Process > Abort All**.
- To terminate all sessions, select **Monitor > Sessions** to open the Sessions window. Then select **Session > Terminate All**.

You have now seen one process instance go to completion. Experiment with other scenarios. Log on other users in different roles. Also, examine the Fusion process development workshops for the application and the client TOOL code to see how the application is implemented.

Note Because of the basic nature of the Expense Report example, it does not write application data, such as ExpenseReport IDs, to a database, but to an array. The Expense Report ID is also stored by the Fusion engine as a process attribute. If you exit the Expense Report application and subsequently restart it, the application data array is cleared and Expense Report IDs restart at 1001. However, if process instances from an earlier run had not been completed, the engine database still has old Expense Report ID entries. The Advanced Expense Report example shows how to use a database to avoid this situation.

Advanced Expense Reporting

Description Advanced Expense Reporting builds on the Expense Reporting example, showing how to call service objects from Fusion process definition methods and how to provide persistent data between sessions of the application.

In the process definition for Advanced Expense Reporting, the automatic activity ProcessCheck now calls a service object to perform this activity (rather than simply write to a log). Additionally, in the Validation plan, the ValidateUser method calls a service object to verify users based on data in a database. This data is created by the OrganizationDatabase example, which is described in detail in the section "OrganizationDatabase" on page 250.

The Advanced Expense Reporting example now stores application data in a database, providing persistence of data between sessions of the application. In the basic Expense Reporting example, application data is stored in an array that is initialized each time the application runs. This means the process attribute data, which is stored both in the array and in the Fusion engine, can get out of synch. However, in the Advanced Expense Reporting example, activities for any incomplete process instances from a previous session are displayed in the appropriate work lists in subsequent sessions.

The Advanced Expense Reporting example provides Fusion plans and the Fusion project, AdvancedExpenseReportClient, that illustrates how the application is implemented. In addition to features from the previous example, Advanced Expense Reporting illustrates the following:

- service object access
- organization database access
- persistent storage of application data in database

Advanced Expense Reporting does not illustrate the following Fusion process management features:

- customized user profile
- junction activity

Pex Files This example uses the following .pex files:

aerso.pex	aerbc.pex
aerrv_pd.pex	aer_uv.pex
aercso.pex	aer_up.pex
aercsc.pex	aer_pd.pex
aerclnt.pex	aer_ar.pex
aerc.pex	aer_ad.pex

Special Requirements A Fusion engine called "ceengine" must be running. Refer to "Configuring and Starting an Engine" on page 222 for more information. In the Fusion engine, all process instances and sessions from previous example runs must be terminated. Use the Fusion Console to terminate any lingering process or sessions.

You must also do the following, as described in the following procedures:

- Define a resource manager for a database (if not previously defined)
- Set up an organization database for user validation
- Set environment variables for access to the organization database
- Create database tables for persistent storage of application data

Resource Manager

Although Fusion does not require a resource manager, the Forte part of this example needs one to perform database access. If a resource manager is not already defined for your node, use the following procedure to define one.

To define a resource manager:

1. From your Forte installation, run the Environment Console.
2. In the Environment Console, select your node and then lock it.
3. Select **Component > Properties**, and define a resource manager for your database. For this example, this database can be the same database that is running your Fusion engine.

Organization Database

The Advanced Expense Reporting example validates users based on information in an organization database, and stores application data in an application database. (Actually, these databases are different tables in the same physical database, however they are logically distinct.) This database activity is distinct from the writing of state information to the engine database by the Fusion engine. However, for this example, the application database you set up for this application should be the same database system as your Fusion engine database. Typically, the application database would be set up as a separate database.

Environment variables

You must set four environment variables that define access to the application database. These environment variables are read by the Advanced Expense Reporting application before it dynamically creates a DBSession object. Set the following four environment variables:

```
FORTE_EP_DB_TYPE  (Oracle, Sybase, etc)
FORTE_EP_DB_NAME  (Resource name, for example @HILLARY_ORACLE)
FORTE_EP_DB_USER_NAME (user name)
FORTE_EP_DB_USER_PASSWORD (password)
```

Import the example

The Advanced Expense Reporting example contains the Conductor Script file aer.csc. This script imports the Fusion and Forte plans into a new workspace called ConductorExamples, and then compiles, distributes, and registers the Fusion plans. It also creates libraries from some of the Forte projects—Fusion needs to dynamically load those libraries.

To import the Advanced Expense Report example into a workspace:

1. If the Fusion engine named ceengine is not already running, start it.
2. From a command window, navigate to the $FORTE_ROOT/install/examples/conductr/adver directory and issue the following command:

   ```
   cscript -i aer.csc -o aer.out
   ```

   Examine aer.out to make sure all the commands completed successfully.

Import Organization Database data

Before you can run the example, you must load the data from the orgdb.sql file into a database using the Organization Database example.

To load the Organization Database data:

1. Import the Organization Database example into your development repository, as follows:

   From the ConductorExamples workspace, select **Plan > Import**. Then navigate to the $FORTE_ROOT/install/examples/conductr/orgdb example directory and select orgdbacc.pex.

2. In your workspace, select the newly imported OrganizationDatabase project and click the Run icon.

   In the Login to Database window that opens, provide valid access information to the database you are using for this example.

3. In the Organization Database window that opens, select **Database > Import**.

   In the chooser that opens, navigate to the adver directory and select the file orgdb.sql.

   This creates the database tables and inserts the data needed by the Advanced Expense Reporting example. The employees and roles displayed in the window are the same as those described directly in the ValidateUser method for the basic Expense Reporting example.

4. Exit the Organization Database application.

For more information about using the Organization Database example, refer to "Organization Database Access" on page 224.

Expense report data tables

Finally, you need to create two tables in your database that store expense reporting data. To keep this example simple, continue to use the same database from the previous procedures. (Typically, you would use a separate database to store this data.)

From a command window, navigate to the adver directory and execute the script corresponding to the database vendor you are using. The scripts use the naming convention er_*databasetype*.sql, where *databasetype* refers to the vendor of your database.

Note

This example has been tested running on Oracle and Sybase, but not on the other databases. If you are using Sybase, edit the er_syb.sql script to point to a valid database. If you are using Informix, Ingress, or ODBC, you may need to edit the matching er_*databasetype*.sql script before running it.

Running the Advanced Expense Reporting Example The behavior of Advanced Expense Reporting is very similar to Expense Reporting. Follow the steps described in "Running the Expense Reporting Example" on page 227. Note the following differences in the behavior of the application:

- The tag line for the ProcessCheck automatic activity now appears in the Work History field in the Monitor Expense Reports window, rather than being written to a separate log.
- The application data is now stored in a database. This means that when you exit the application with incomplete processes, you do not have to terminate them before starting the application again. (In the basic Expense Reporting example you had to make sure all processes terminated before exiting the application.) When restarting the application, activities for incomplete processes appear in the appropriate work lists and activity queues.

▶ **To run the Advanced Expense Reporting application:**

1 Start Forte Distributed, and set your workspace to ConductorExamples. Select AdvExpenseReportClient, and click the Run icon.

2 Follow the steps in the basic Expense Reporting example on page 228 and observe the new behavior introduced with this example.

As with the basic Expense Reporting example, experiment with other scenarios. Also, examine the Fusion plans and projects for the application to see how it is implemented.

JExpense

Description JExpense illustrates how to use Fusion's CORBA/IIOP process client API. The client application in this example is written in Java. JExpense uses the process logic defined in the basic Expense Reporting example—it does not rely on Forte service objects or use the TOOL client code from the Expense Reporting example.

Note You should be familiar with the basic Expense Reporting example before running JExpense.

The JExpense client application simply starts a process instance, automatically logs in appropriate users, and takes the process instance to completion. It is not intended to be a complete client, but to illustrate the following:

- the CORBA/IIOP interface
- the appropriate Java syntax for referencing classes generated by the IDL to Java conversion

In this example, the Java client reads from an IOR file to locate objects. In the next example, JExpenseNS, the Java client uses a CORBA naming service. Run both examples to compare the two techniques. You do not need to restart the Fusion engine between runs of the two examples, but you do need to restart the IIOP server.

Pex Files Same as Expense Reporting example.

Special Requirements A Fusion engine called "ceengine" must be running before you run the example. Refer to "Configuring and Starting an Engine" on page 222 for more information. In the Fusion engine, all process instances and sessions from previous example runs must be terminated. Use the Fusion Console to terminate any lingering process or sessions.

You must have JDK 1.2.2 installed on the client machine, with your CLASSPATH environment variable properly set.

The example contains both a client and server installation. You can install both the client and server sides on a single machine, or you can install them on separate machines as indicated in the instructions. If you are using a single machine, you may want to open separate command windows for the server side and client side of the example.

The basic Expense Reporting Example (described on page 226) must be installed on the server machine. If you previously installed the Expense Reporting Example, you do not need to reinstall it. If you also installed the Advanced Expense Reporting example, you need to register the UserValidation from the basic example, as described in Step 2 of the following procedure.

▶ **To install the JExpense application:**

Server side

1 If you have not installed the basic Expense Reporting example, install it on the server machine, following the instructions on page 227.

Note JExpense uses only the Expense Reporting Fusion plans installed by the Expense Reporting script er.csc. This script automatically registers these plans with the Fusion engine named ceengine.

2 If you installed the Advanced Expense Reporting example, you need to register the UserValidation from the basic example as follows:

   a If the Fusion engine named ceengine is not already running on the server side for this example, start it.

   b From the Repository Workshop, double-click the plan ERUV to open the Validation Workshop.

   c From the Validation Workshop, select **File > Distribute**.

   d In the Distribute Options window that opens, specify Register (you do not have to specify compile) and click **OK**.

   e Close the Validation Workshop.

Client side

3 On the machine you are using for the client side of the installation, create a working directory. Then copy the files jexpense.java and conductor.jar from the $FORTE_ROOT/install/examples/conductr/jer example directory to the working directory.

4 On the client machine, modify your CLASSPATH environment variable to include the path to the conductor.jar file you copied in Step 3.

5 In a command window on the client machine, navigate to the working directory and compile jexpense.java by issuing the following command:

```
javac jexpense.java
```

Ignore the compiler message about using a deprecated API.

Running the JExpense Example The following procedures show how to start an IIOP server and run the JExpense application from both client and server machines. Unlike the earlier examples, the steps that take a process to completion are automated. Messages printed to standard out (plus an additional message written to a log file) indicate completion of steps in the process. You should examine the file jexpense.java to further understand how the example works.

For information on process automation and workflow in the application, refer to "Expense Reporting" on page 226.

▶ **To start the IIOP server for the JExpense application:**

Server side

1 On the server machine, open a command window and issue the following command to start Conductor Script:

```
cscript
```

2 From the Cscript prompt, issue the following command to start the IIOP server:

```
cscript > iiopserver start
```

Starting the IIOP server creates the file conductr.ior, which contains an initial object reference, at the following location:

```
$FORTE_ROOT/etc/iiopior/conductr.ior
```

Every time the IIOP server is restarted, the conductr.ior file is re-created.

The conductr.ior file provides a node name, TCP/IP port, and other information required to find an object. The client side of the JExpense example uses this file to obtain access information to objects on the server side.

▶ **To run the JExpense client application:**

Server side

1 If the Fusion engine named ceengine is not already running on the server side for this example, start it.

Client side

2 If you are using a separate client machine, on the client machine, mount the drive on the server machine containing the conductr.ior file.

3 In a command window, navigate to the working directory on the client machine and start the client application by issuing the following command:

```
java jexpense [Drive:]%FORTE_ROOT%\etc\iiopior\conductr.ior
```

Drive indicates any drive letter needed to specify the server machine.

The JExpense client application does not require user input. It simply prints out messages describing its actions. The JExpense client displays the following messages as it takes a process to completion:

```
Starting Java Expense Report Client
Attempting to connect with ceengine.
Our engine's name: ceengine

Signing on as employee Winnie...
Session Name = Employee
Building the attribute descriptor array...
Creating the process instance...
Closing employee session...

Signing on as manager Charlotte...
Session Name = Manager
Getting the activity list...
Number of current activities = 1
Getting the attributes...
ExpenseReportID =1001
Priority =false
Status =1
Change status attribute value to 3 - Accepted.
Getting the attributes...
ExpenseReportID =1001
Priority =false
Status =3
Ending the activity...
Closing the manager session...

Opening session for accountant Nick...
Session Name = Accountant
Working on Expense Report #1001
Completing the accounting activity...
Closing the accountant session...

Opening session for auditor Ernie...
Session Name = Auditor
Working on Expense Report #1001
Completing the auditing activity...
Closing the auditor session...
```

In this example, the automatic activity ProcessCheck does not print to standard out, but instead writes to a log file. This automatic activity is triggered only after both the accountant and the auditor have completed their work. The following procedure shows you how to use the Fusion Console to view the engine partition log, which includes any entries for this automatic activity.

To view the engine partition log:

1. If it is not open, open the Fusion Console.
2. Select your engine, and then select **Engine > Status**.
3. In the Engine Status window that opens, under Engine Components, select the engine component and then click View Partition Log.

   If you completed the example, the log contains the following line:

   "Check processed for expense report 1001"

One process instance has now been created and driven to completion by the Java client application. You can run this application as many times as you like. Review the implementation in the jexpense.java file to understand how to write a Java client that uses the Fusion CORBA/IIOP process client API.

JExpenseNS

Description Like the JExpense example, JExpenseNS is a Java client that uses Fusion's CORBA/IIOP client API. In JExpenseNS, the Java client uses the CORBA naming service to obtain access information to server objects. (In JExpense, the Java client reads from an IOR file generated by the IIOP server.) You can run both examples to compare the two techniques. You do not need to restart the Fusion engine between runs, but you do need to restart the IIOP server.

Note You should be familiar with the basic Expense Reporting example and the JExpense example before installing and running the JExpenseNS example.

JExpenseNS uses the process logic defined in the basic Expense Reporting example—it does not rely on Forte service objects or use the TOOL client code from the Expense Reporting example. The JExpenseNS client application simply starts a process instance, automatically logs in appropriate users, and takes the process instance to completion. It is not intended to be a complete client, but to illustrate how to use the CORBA naming service with the CORBA/IIOP client API.

Pex Files Same as Expense Reporting example.

Special Requirements A Fusion engine called "ceengine" must be running before you run the example. Refer to "Configuring and Starting an Engine" on page 222 for more information. In the Fusion engine, all process instances and sessions from previous example runs must be terminated. Use the Fusion Console to terminate any lingering process or sessions.

You must have JDK 1.2.2 installed on the client machine, with your CLASSPATH environment variable properly set.

The example contains both a client and server installation. You can install both the client and server sides on a single machine, or you can install them on separate machines as indicated in the instructions. If you are using a single machine, you may want to open separate command windows for the server side and client side of the example.

The instructions assume that you have installed the basic Expense Reporting example (described on page 226) and the JExpense example (described on page 233).

Note If you also installed the Advanced Expense Reporting example (page 230), you may need to register the UserValidation from the basic example. This is because the Fusion engine can have only one UserValidation registration. Refer to Step 2 of the JExpense installation procedure on page 233 for information on how to register the UserValidation.

The installation procedure for JExpenseNS is similar to the JExpense installation procedure.

▶ **To install the JExpenseNS application:**

Server side

1 If you have not installed the basic Expense Reporting example, install it on the server machine, following the instructions on page 227.

Also, if you need to register the UserValidation from the basic example, follow the instructions from Step 2 of the JExpense installation procedure on page 233.

Client side

2 On the client machine, copy jexpensens.java and nsclient.java from $FORTE_ROOT/install/examples/conductr/jer example directory to the working directory you created for the JExpense example.

This directory should already contain the conductor.jar file used in the previous example.

3 Make sure your CLASSPATH environment variable includes the path to the conductor.jar file in the working directory.

4 In a command window on the client machine, navigate to the working directory and compile jexpensens.java:

```
javac jexpensens.java
```

5 In the same command window, compile nsclient.java:

```
javac nsclient.java
```

Running the JExpenseNS Example The following procedures show how to start a naming service and an IIOP server. It then shows how to run the JExpense application from both client and server machines. The behavior of this example is similar to the JExpense example, except that it uses a CORBA naming service to locate objects. (JExpense reads from an IOR file generated by the IIOP server.) To further understand how the example works, compare the jexpensens.java to the jexpense.java file, and also examine the nsclient.java file.

For information on process automation and workflow in the application, refer to "Expense Reporting" on page 226.

▶ **To start a naming service and the IIOP server:**

Server side

1 If the IIOP server is running on the server machine, issue the following Cscript command to stop it:

```
cscript > iiopserver stop
```

Client side

2 On the client machine, run the Java tnameserv executable in the background. This file should be located in your JDK bin directory.

3 From the working directory on the client machine, run your nsclient application:

```
java nsclient
```

When you run nsclient, it creates a file called ns.ior in the current directory. This file has a node name, TCP/IP port, and other information required to find an object.

Server side

4 Copy the newly created ns.ior file to the following directory on the server machine:

```
$FORTE_ROOT/etc/iiopior
```

The Fusion process uses this file to locate the CORBA naming service.

5 On the server machine, issue the following Cscript command to start the IIOP server:

```
cscript > iiopserver start
```

The IIOP server obtains information from the ns.ior file in $FORTE_ROOT/etc/iiopior.

Every time the IIOP server is restarted, it reads from the ns.ior file.

To run the JExpenseNS application:

1 If the Fusion engine named ceengine is not already running on the server side for this example, start it.

2 From the working directory on the client machine, run the Java client application:

```
java jexpensens
```

The behavior of this example is similar to the JExpense example described on page 233. The only difference is that this example uses a CORBA naming service to locate objects.

JExpenseSO

Description Like the JExpense and JExpenseNS examples, JExpenseSO is a Java client that uses Fusion's CORBA/IIOP client API. In JExpenseSO, the Java client uses the CORBA naming service to communicate with the Fusion engine. JExpenseSO also uses an IOR file to communicate with the Expense Reporting application's service object. Additional techniques are required to generate this IOR file and the Java files to communicate with the service object. You do not need to restart the Fusion engine between runs, but you do need to restart the IIOP server.

Note You should be familiar with the basic Expense Reporting example and the JExpense and JExpenseNS examples before installing and running the JExpenseSO example.

JExpenseSO uses the process logic defined in the basic Expense Reporting example. It also calls a Forte service object to get the next expense report number. JExpenseSO does not use the TOOL client code from the Expense Reporting example. The JExpenseSO client application simply starts a process instance, automatically logs in appropriate users, and takes the process instance to completion. It is not intended to be a complete client, but to illustrate how to use the CORBA naming service with the CORBA/IIOP client API, in conjunction with accessing a Forte service object.

Pex Files Same as Expense Reporting example.

Special Requirements A Fusion engine called "ceengine" must be running before you run the example. Refer to "Configuring and Starting an Engine" on page 222 for more information. In the Fusion engine, all process instances and sessions from previous example runs must be terminated. Use the Fusion Console to terminate any lingering process or sessions.

You must have JDK 1.2.2 installed on the client machine, with your CLASSPATH environment variable properly set.

The example contains both a client and server installation. You can install both the client and server sides on a single machine, or you can install them on separate machines as indicated in the instructions. If you are using a single machine, you may want to open separate command windows for the server side and client side of the example.

The instructions assume that you have installed the basic Expense Reporting example (described on page 226) and the JExpense example (described on page 233).

Note If you also installed the Advanced Expense Reporting example (page 230), you may need to register the UserValidation from the basic example. This is because the Fusion engine can have only one UserValidation registration. Refer to Step 2 of the JExpense installation procedure on page 233 for information on how to register the UserValidation.

The base example, Expense Report, has CorbaFlat added as an extended property on the ExpenseReportBusinessClasses project. This property is not used by any other example, and has no adverse effect on the examples. Setting the CorbaFlat extended property adds data to the idl file so that, when idltojava is run on it, it produces an ExpenseReport_struct.java file. It is often convenient to use structs based on objects in your Java client code.

You do not have to make any changes to Expense Reporting (CorbaFlat is already included in that example). To add the CorbaFlat property to your own application, in the Repository Workshop, select your project. Then select **Plan > Extended Properties**. Select **New**, then and add CorbaFlat with a value of 1.

The installation procedure for JExpenseSO is similar to the JExpenseNS installation procedure, but there are some additional steps.

To install the JExpenseSO application:

Server side

1. If you have not installed the basic Expense Reporting example, install it on the server machine, following the instructions on page 227. You must have a current version of the Expense Reporting example. If you have one from an earlier release it will not work.

   Also, if you need to register the UserValidation from the basic example, follow the instructions from Step 2 of the JExpense installation procedure on page 233.

2. Define the ior file that will be used to connect with the Forte service object.

   a. Start Forte Distributed and select ExpenseReportClient. Open the Partition Workshop.

   b. In the Partition Workshop, click the arrow next to ExpenseReportClient.

   c. Double-click ExpenseReportMgrSO_cl0_Part1 to open it in the Service Object Properties window.

   d. Select the Export tab page.

   e. Select IIOP for External Type to open the IIOP Configuration window.

   f. Enter ercon.ior in the Name field, and make sure it is set to create at runtime.

      Since you did not provide a path in the Name field, the file is created in FORTE_ROOT/etc/iiopior.

   g. Click **OK** in both windows.

   This example uses ercon.ior to connect with the Expense Reporting application's service object. It uses ns.ior to connect with the Fusion engine. The conductor.ior file is not used.

3. Now distribute, install, and run the Expense Reporting application, as described below.

   This step creates the ercon.ior file, and provides a running executable of the service object, which the Java client later needs.

   a. Still in the Partition Workshop, select **File > Make Distribution**.

   b. Turn on Full Make and Install in Current Environment. Then click Make.

   c. Exit from the Partition Workshop, and exit from the Forte Distributed IDE.

   d. Start the Expense Reporting application from the Forte Applications window (not from Forte Distributed).

e Logon as Winnie and click the **Create New Expense Report** button.

When you see the value 1001 appear, you know the service object has been started. Click **Cancel** rather than starting this process, then exit from Winnie's employee window. (If you do not exit from Winnie's window, the Java client complains about it later.)

f Check the FORTE_ROOT/etc/iiopior directory to confirm that a new ercon.ior file has been created.

Client side

4 On the client machine, copy jexpenseso.java from $FORTE_ROOT/install/examples/conductr/jer example directory to the working directory you created for the JExpense example.

This directory should already contain the conductor.jar and nsclient.java files used in the previous examples. If it does not have these files, copy them from $FORTE_ROOT/install/examples/conductr/jer to your work directory.

5 Make sure your CLASSPATH environment variable includes the path to the conductor.jar file in the working directory.

6 Copy the Expense Report idl file (at the location below) from the server machine to the working directory on your client machine.

The idl file that you need from the server machine is located at $FORTE_ROOT/appdist/centrale/expenser/cl0/generic/expens1/corba.idl

7 In a command window, issue the following command to convert the idl file to java files.

```
idltojava corba1.idl
```

The idl to java conversion produces a number of subdirectories.

Note If you do not have the idl compiler, download idltojava.exe from http://www.javasoft.com/products/jdk/idl/index.html.

8 Compile all the .java files in the ExpenseReportBusinessClasses and the ExpenseReportCacheServiceClasses with the following commands:

```
javac ExpenseReportBusinessClasses/*.java
javac ExpenseReportCacheServiceClasses/*.java
```

9 In a command window on the client machine, still in the working directory, compile jexpenseso.java:

```
javac jexpenseso.java
```

10 In the same command window, compile nsclient.java:

```
javac nsclient.java
```

Running the JExpenseSO Example The following procedures show how to start a naming service and an IIOP server. It then shows how to run the JExpenseSO application from both client and server machines. The behavior of this example is similar to the JExpenseNS example, except that it additionally connects with a Forte service object using an IOR file. To further understand how the example works, compare the implementation in the jexpenseso.java and jexpensens.java files.

For information on process automation and workflow in the application, refer to "Expense Reporting" on page 226.

▶ **To start a naming service and the IIOP server:**

Server side

1 If the IIOP server is running on the server machine, issue the following cscript command to stop it:

```
cscript > iiopserver stop
```

Client side

2 On the client machine, run the Java tnameserv executable in the background. This file should be located in your JDK bin directory.

3 From the working directory on the client machine, run your nsclient application:

```
java nsclient
```

When you run nsclient, it creates a file called ns.ior in the current directory. This file has a node name, TCP/IP port, and other information required to find an object.

Server side

4 Copy the newly created ns.ior file to the following directory on the server machine:

```
$FORTE_ROOT/etc/iiopior
```

The Fusion process uses this file to locate the CORBA naming service.

5 On the server machine, issue the following Cscript command to start the IIOP server:

```
cscript > iiopserver start
```

The IIOP server obtains information from the ns.ior file in $FORTE_ROOT/etc/iiopior.

Every time the IIOP server is restarted, it reads from the ns.ior file.

▶ **To run the JExpenseSO application:**

1 If the Fusion engine named ceengine is not already running on the server side for this example, start it.

2 From the working directory on the client machine, run the Java client application:

```
java jexpenseso [Drive:]%FORTE_ROOT%\etc\iiopior\ercon.ior
```

Drive indicates any drive letter needed to specify the server machine.

The behavior of this example is similar to the JExpenseNS example described on page 236. The only difference is that this example is also connected to the Forte service object.

To prove the connection was made, the client calls the GetNewExpenseReport method on the ExpenseReportMgrSO. In the output, note that the report is 1002. Because you already created expense report 1001 earlier (when you ran the application to start the service object), you are now getting the next available expense report number. If you run jexpenseso again, you get expense report 1003.

JExpenseNB

Description JExpenseNB uses a Java client developed in NetBeans, and uses the communication techniques described in JExpenseSO. Like the JExpenseSO example, JExpenseNB is a Java client that uses Fusion's CORBA/IIOP client API. In JExpenseNB, the Java client uses the CORBA naming service to communicate with the Fusion engine. JExpenseNB also uses an IOR file to communicate with the Expense Reporting application's service object. Additional techniques are required to generate this IOR file and the Java files to communicate with the service object. You do not need to restart the Fusion engine between runs, but you do need to restart the IIOP server.

Note You should be familiar with the basic Expense Reporting example and the JExpense, JExpenseNS, and JExpenseSO examples before installing and running the JExpenseNB example.

JExpenseNB uses the process logic defined in the basic Expense Reporting example. It also calls a Forte service object to get the next expense report number.

Note If you have already installed and run JExpenseSO, you do not need to redo the idltojava conversion. You can reuse the ExpenseReportBusinessClasses and ExpenseReportCacheServiceClasses file that you created. The full installation instructions are provided here for completeness.

The GUI client in this example runs in a browser. The interaction is similar to the Expense Reporting TOOL client. Some functionality is eliminated to limit the size of the application. But you can sign on as the familiar users, in each of the four roles, create processes and drive them to completion.

Pex Files Same as Expense Reporting example.

Special Requirements A Fusion engine called "ceengine" must be running before you run the example. Refer to "Configuring and Starting an Engine" on page 222 for more information. In the Fusion engine, all process instances and sessions from previous example runs must be terminated. Use the Fusion Console to terminate any lingering process or sessions.

You must have JDK 1.2.2 installed on the client machine, with your CLASSPATH environment variable properly set.

You must have NetBeans 3.0 installed on the client machine. This example was tested against Build 2000. If you do not have NetBeans, download it from NetBean's Open Source website: http://www.netbeans.org. Click Download. Ignore alarming messages and download the Java Build. You only need the build file. You do not need the source, the additional binaries, or the other downloads. Follow their instructions to install the downloaded files on your client machine.

The example contains both a client and server installation. You can install both the client and server sides on a single machine, or you can install them on separate machines as indicated in the instructions. If you are using a single machine, you may want to open separate command windows for the server side and client side of the example.

The instructions assume that you have installed the basic Expense Reporting example (described on page 226) and the JExpense example (described on page 233).

Note If you also installed the Advanced Expense Reporting example (page 230), you may need to register the UserValidation from the basic example. This is because the Fusion engine can have only one UserValidation registration. Refer to Step 2 of the JExpense installation procedure on page 233 for information on how to register the UserValidation.

The base example, Expense Report, has CorbaFlat added as an extended property on the ExpenseReportBusinessClasses project. This property is not used by any other example, and has no adverse effect on the examples. Setting the CorbaFlat extended property adds data to the idl file so that, when idltojava is run on it, it produces an ExpenseReport_struct.java file. It is often convenient to use structs based on objects in your Java client code.

You do not have to make any changes to Expense Reporting (CorbaFlat is already included in that example). To add the CorbaFlat property to your own application, in the Repository Workshop, select your project. Then select **Plan > Extended Properties**. Select **New**, then and add CorbaFlat with a value of 1.

The installation procedure for JExpenseNB is similar to the JExpenseSO installation procedure, but there are some additional steps related to running the NetBeans GUI client. All steps are provided here for completeness.

To install the JExpenseNB application:

Server side

1 If you have not installed the basic Expense Reporting example, install it on the server machine, following the instructions on page 227. You must have a current version of the Expense Reporting example. If you have one from an earlier release it will not work.

   Also, if you need to register the UserValidation from the basic example, follow the instructions from Step 2 of the JExpense installation procedure on page 233.

2 Define the ior file that will be used to connect with the Forte service object.

   a Start Forte Distributed and select ExpenseReportClient. Open the Partition Workshop.

   b In the Partition Workshop, click the arrow next to ExpenseReportClient.

   c Double-click ExpenseReportMgrSO_cl0_Part1 to open it in the Service Object Properties window.

   d Select the Export tab page.

   e Select IIOP for External Type to open the IIOP Configuration window.

   f Enter ercon.ior in the Name field, and make sure it is set to create at runtime.

   Since you did not provide a path in the Name field, the file is created in FORTE_ROOT/etc/iiopior.

   g Click OK in both windows.

   This example uses ercon.ior to connect with the Expense Reporting application's service object. It uses ns.ior to connect with the Fusion engine. The conductor.ior file is not used.

3 Now distribute, install, and run the Expense Reporting application, as described below.

   This step creates the ercon.ior file, and provides a running executable of the service object, which the Java client later needs.

   a Still in the Partition Workshop, select **File > Make Distribution**.

   b Turn on Full Make and Install in Current Environment. Then click Make.

   c Exit from the Partition Workshop, and exit from the Forte Distributed IDE.

   d Start the Expense Reporting application from the Forte Applications window (not from Forte Distributed).

   e Logon as Winnie and click the Create New Expense Report button.

   When you see the value 1001 appear, you know the service object has been started. Click Cancel rather than starting this process, then exit from Winnie's employee window. (If you do not exit from Winnie's window, the Java client complains about it later.)

   f Check the FORTE_ROOT/etc/iiopior directory to confirm that a new ercon.ior file has been created.

Client side

4 On the client machine, copy jexpenseso.java from $FORTE_ROOT/install/examples/conductr/jer example directory to the working directory you created for the JExpense example.

   This directory should already contain the conductor.jar and nsclient.java files used in the previous examples. If it does not have these files, copy them from $FORTE_ROOT/install/examples/conductr/jer to your work directory.

5 Make sure your CLASSPATH environment variable includes the path to the conductor.jar file in the working directory.

6 Copy the Expense Report idl file (at the location below) from the server machine to the working directory on your client machine.

The idl file that you need from the server machine is located at $FORTE_ROOT/appdist/centrale/expenser/cl0/generic/expens1/corba.idl

7 In a command window, issue the following command to convert the idl file to java files.

```
idltojava corba1.idl
```

The idl to java conversion produces a number of subdirectories.

Note If you do not have the idl compiler, download idltojava.exe from http://www.javasoft.com/products/jdk/idl/index.html.

8 Compile all the .java files in the ExpenseReportBusinessClasses and the ExpenseReportCacheServiceClasses with the following commands:

```
javac ExpenseReportBusinessClasses/*.java
javac ExpenseReportCacheServiceClasses/*.java
```

9 In a command window on the client machine, compile nsclient.java:

```
javac nsclient.java
```

10 Copy the following files and directories to your NetBeans Development directory:

```
conductor.jar
ExpenseReportBusinessClasses/*.*
ExpenseReportCacheServiceClasses/*.*
```

11 Unzip the nber.zip file, placing the JExpenseNB directory directly under your NetBeans Development directory:

```
JExpenseNB/*.*
```

12 Run NetBeans. In the NetBeans explorer window, you should see conductor.jar and the directories you just copied under Development.

13 In the NetBeans Explorer window, expand JExpenseNB. Expand the jexpenseso class under JExpenseNB. Double-click the method initializeConnections, to open it in the NetBeans editor. In the editor window, modify the String variable ourargs to point to the proper drive and path for the ior file on your server machine.

14 In the NetBeans explorer window, select JExpenseNB and click **Build > Build All**. (You can ignore the warning about a deprecated API.)

Running the JExpenseNB Example The following procedures show how to start a naming service and an IIOP server. It then shows how to run the JExpenseNB application from both client and server machines. The behavior of this example is similar to the JExpenseSO example, except that it has a GUI client, developed in NetBeans. To further understand how the example works, examine the Java and JSP files in NetBeans. The controller servlet's processRequest method is where the flow of client activity is controlled. The class jexpenseso is where most of the calls to the service objects and the engine are made.

For information on process automation and workflow in the application, refer to "Expense Reporting" on page 226.

▶ **To start a naming service and the IIOP server:**

Server side

1 If the IIOP server is running on the server machine, issue the following Cscript command to stop it:

```
cscript > iiopserver stop
```

Client side

2 On the client machine, run the Java tnameserv executable in the background. This file should be located in your JDK bin directory.

3 From the working directory on the client machine, run your nsclient application:

```
java nsclient
```

When you run nsclient, it creates a file called ns.ior in the current directory. This file has a node name, TCP/IP port, and other information required to find an object.

Server side

4 Copy the newly created ns.ior file to the following directory on the server machine:

```
$FORTE_ROOT/etc/iiopior
```

The Fusion process uses this file to locate the CORBA naming service.

5 On the server machine, issue the following Cscript command to start the IIOP server:

```
cscript > iiopserver start
```

The IIOP server obtains information from the ns.ior file in $FORTE_ROOT/etc/iiopior.

Every time the IIOP server is restarted, it reads from the ns.ior file.

▶ **To run the JExpenseNB application:**

1 If the Fusion engine named ceengine is not already running on the server side for this example, start it.

2 In the NetBeans explorer window, select logon.jsp under JExpenseNB. Right click and select **Execute**.

3 A browser window opens, providing familiar logon options. Refer to "Running the Expense Reporting Example" on page 227 for appropriate user/password/role combinations. You need to execute logon.jsp each time you want to logon as a different user.

The functionality provided is a subset of the Expense Reporting TOOL client example, so it should be familiar. You can track sessions, processes, and activities in the engine. You can mix using TOOL and Java applications for various roles.

C++ Expense Reporting

Description C++ Expense Reporting illustrates how to use Fusion's C++ client API. The client application in this example is written in C++. C++ Expense Reporting makes use of the process logic defined in the Expense Reporting example, but does not rely on the Forte service objects nor does it use the TOOL client code from the Expense Reporting example.

The C++ Expense Reporting client application lets you select the engine, then log in any of the four roles defined for the Expense Reporting example. You can enter expense reports as an employee, review and approve them as a manager, and perform auditing and accounting tasks on them when signed on as an auditor or accountant. A simple command line interface guides you through the process.

C++ Expense Reporting is not intended to be a complete client, but to illustrate the following:

- the Fusion C++ client interface
- the appropriate syntax for referencing classes in the C++ API

For examples of more realistic clients, refer to the Expense Reporting and Advanced Expense Reporting examples.

Pex Files Same as Expense Reporting example.

Special Requirements A Fusion engine called "ceengine" must be running before you run the example. Refer to "Configuring and Starting an Engine" on page 222 for more information. In the Fusion engine, all process instances and sessions from previous example runs must be terminated. Use the Fusion Console to terminate any lingering process or sessions.

You must have a C++ compiler installed and properly included in your path.

The instructions assume that you have installed the basic Expense Reporting example, described on page 226.

Note If you also installed the Advanced Expense Reporting example (page 230), you may need to register the UserValidation from the basic example. This is because the Fusion engine can have only one UserValidation registration. Refer to Step 2 of the JExpense installation procedure on page 233 for information on how to register the UserValidation.

To install the C++ Expense Reporting application:

1. If you have not installed the basic Expense Reporting example, install it following the instructions on page 227.

Note C++ Expense Reporting uses only the Expense Reporting Fusion plans installed by the Expense Reporting script er.csc. This script automatically registers these plans with the Fusion engine named ceengine.

2. Create a working directory for compiling and linking the C++ client.
3. Copy the following files from $FORTE_ROOT/install/examples/conductr/cer to your working directory:

```
ermain.cpp
ermain.h
ermain.mak (make file for NT)
bldclnt.csh (build script for Unix platforms)
```

4. Examine ermain.mak (if you are running on NT), ermain.h, and ermain.cpp to make sure they properly reflect your environment.

   ermain.mak assumes your FORTE_ROOT directory is c:\forte. Make a global edit if your FORTE_ROOT is set to something else.

5. Navigate to your working directory and build the example as follows:

   On Windows NT:

```
nmake -f ermain.mak
```

   ermain.mak creates a debug directory, which contains the ERClient.exe file.

   On UNIX:

```
bldclnt.csh ermain.cpp erclient
```

6 If you are working on NT, copy the following files into the debug directory created in Step 5 of this procedure:

```
$FORTE_ROOT/userapp/ofcustom/cl1/libofcus.dll
$FORTE_ROOT/userapp/wfclien1/cl1/wfclie0.dll
$FORTE_ROOT/userapp/wfclient/cl1/libwfcli.dll
```

Alternatively, you could include the above directories in your path. However, it is usually easier to copy the files to your working directory.

7 If you are working on a UNIX platform, copy the equivalent libraries to those listed in Step 6 to your working directory. The libraries have the following extensions:

Unix Platform	Library Extension
HP9000	.sl
RS6000	.a
AlphaOSF	.so
Solsparc	.so

Running the C++ Expense Reporting Example The following procedure shows how to run the C++ Expense Reporting application. The behavior of the C++ client is similar to the behavior of the TOOL client created in the basic Expense Reporting example. It allows you to interactively take a process, (which was defined in the Process Definition Workshop) to completion. However, the C++ example does not provide a graphical user interface. Instead, it provides a command line interface for taking the process to completion.

For information on process automation and workflow in the application, refer to "Expense Reporting" on page 226. For information on how the C++ client API is implemented in the example, refer to the source files for the example.

To run the C++ Expense Reporting application:

1 If the Fusion engine named ceengine is not already running, start it.

If you have been running other Fusion example applications, make sure there are no lingering process instances or sessions. If there are any, terminate them in Fusion Console before running this example. ERClient is expecting a clean slate in the engine. For the same reason, do not attempt to run the Expense Report client simultaneously with this example.

Also, if you need to register the UserValidation from the basic example, follow the instructions from Step 2 of the JExpense installation procedure on page 233.

2 In your working directory (on Unix) or the Debug subdirectory (on NT) start the executable:

```
erclient
```

The client prompts you for input and also prints messages describing its actions. It displays the following messages:

```
Starting the C++ client for the Expense Reporting process.
...
Running Conductor engines:
  1 ceengine
Select engine #(1...1)(0 to quit):
```

3 Select the engine that has the Expense Reporting example definitions registered.

4 Follow the guidelines below to take a process in the example to completion.

The client prompts you to log in one of the four roles. If you are not already familiar with how to run the Expense Reporting example, refer to the instructions in "Running the Expense Reporting Example" on page 227. Follow the same sequence recommended for the Expense Reporting example:

- Log in as an employee and enter an expense report
- Log in as a manager and approve the expense
- Log in as an accountant and as an auditor to process the expense

As you complete the process, the client application prints messages describing its actions. However, as with the basic Expense Reporting example, the automatic activity ProcessCheck writes to a log file when it is completed. This automatic activity is triggered only after both the accountant and the auditor have completed their work. To view the results in the log file, refer to the procedure on page 236.

After taking one process instance to completion, feel free to experiment with other scenarios. You can monitor the processes, sessions, and activities you create with this client in the Fusion Console.

ActiveX Expense Reporting

Description ActiveX Expense Reporting illustrates how to use Fusion's ActiveX client API. The client application in this example is written in Visual Basic. ActiveX Expense Reporting uses the process logic defined in the Expense Reporting example—it does not rely on Forte service objects or use the TOOL client code from the Expense Reporting example. The Visual Basic client application provides a GUI. The user can connect to an engine, log in as appropriate users, and take the process instances to completion.

ActiveX Expense Reporting is not intended to be a complete client, but to illustrate the following:

- the ActiveX client interface
- the appropriate Visual Basic syntax for referencing the ActiveX API

For examples of more realistic Fusion clients, refer to the Expense Reporting and Advanced Expense Reporting examples.

Pex Files Same as Expense Reporting example.

Special Requirements A Fusion engine called "ceengine" must be running. For more information, refer to "Configuring and Starting an Engine" on page 222.

You must have Microsoft Visual Basic 5.0 installed.

The instructions assume that you have installed the basic Expense Reporting example, described on page 226.

Note If you also installed the Advanced Expense Reporting example (page 230), you may need to register the UserValidation from the basic example. This is because the Fusion engine can have only one UserValidation registration. Refer to Step 2 of the JExpense installation procedure on page 233 for information on how to register the UserValidation.

To install the ActiveX Expense Reporting application:

1 If you have not installed the basic Expense Reporting example, install it following the instructions on page 227.

Note ActiveX Expense Reporting uses only the Expense Reporting Fusion plans installed by the Expense Reporting script er.csc. This script automatically registers these plans with the Fusion engine named ceengine.

2 In Windows Explorer, navigate to the following example directory, which contains the Visual Basic files for the ActiveX Expense Reporting example:

```
%FORTE_ROOT%\install\examples\conductr\vber
```

3 Double-click the file er.vbp, which invokes the Microsoft Visual Basic development environment with er.vbp as its current project.

Alternately, you could start Visual Basic from the Start Programs menu, choose **File > Open Project**, and then navigate to er.vbp in the example directory.

4 At this point, you may want to examine the forms and modules that comprise this example.

Running the ActiveX Expense Reporting Example The following procedure shows how to run the ActiveX Expense Reporting application in the VisualBasic interpreter. The behavior of the ActiveX client is similar to the behavior of the TOOL client created in the basic Expense Reporting example. It allows you to interactively take a process, (which was defined in the Process Definition Workshop) to completion. However, the ActiveX example provides a graphical user interface built with VisualBasic.

For information on process automation and workflow in the application, refer to "Expense Reporting" on page 226. For information on how the ActiveX client API is implemented in the example, refer to the source files for the example.

To run the ActiveX Expense Reporting application:

1 If the Fusion engine named ceengine is not already running, start it.

If you have been running other Fusion example applications, make sure there are no lingering process instances or sessions. If there are any, terminate them in the Fusion Console before running this example. The Visual Basic client Expense Report expects a clean slate in the engine. For the same reason, do not attempt to run the Expense Report client simultaneously with this example.

Also, if you need to register the UserValidation from the basic Expense Reporting example, follow the instructions from Step 2 of the JExpense installation procedure on page 233.

2 In the Microsoft Visual Basic development environment, click the run icon to start the example.

The Get Engine window opens. If you have followed the normal setup procedures for the Expense Reporting example, the Fusion engine named ceengine appears in the Engine List.

3 In the Get Engine window, select the engine ceengine and click the Open Engine button.

The Expense Reporting logon window opens.

4 Follow the guidelines below to take a process in the example to completion.

The client prompts you to log in one of the four roles. If you are not already familiar with how to run the Expense Reporting example, refer to the instructions in "Running the Expense Reporting Example" on page 227. Follow the same sequence recommended for the Expense Reporting example:

- Log in as an employee and enter an expense report
- Log in as a manager and approve the expense
- Log in as an accountant and as an auditor to process the expense

As you complete the process, the client application prints messages describing its actions. Fewer details are provided for each expense, since this client communicates with the engine only, and not with the Forte service objects.

Note There is one significant difference between this Visual Basic client and the TOOL client. Due to the lack of event support in the ActiveX API, the display of the activity list is handled differently from how it is handled in the TOOL client. Two windows display the activity list: the Employee Expense Report Information window and the Expense Report Review window. In the Visual Basic client, you *must click the Update List button to see the current activity list*. Always click Update List before selecting an activity and performing work on it.

As with the basic Expense Reporting example, the automatic activity ProcessCheck writes to a log file when it is completed. This automatic activity is triggered only after both the accountant and the auditor have completed their work. To view the results in the log file, refer to the procedure on page 236.

After taking one process instance to completion, feel free to experiment with other scenarios. You can monitor the processes, sessions, and activities you create with this client in the Fusion Console.

OrganizationDatabase

Description OrganizationDatabase provides a GUI for maintaining a user organization database, which can be any relational database supported by Forte. This example creates tables for a simplified corporate organization. A Fusion process client application typically references this database using the ValidateUser method developed in the Validation Workshop.

The OrganizationDatabase application is intended as a starting point in the creation of an organization database that you can use with Fusion client applications. Typically, you adapt it to your business organization. Alternatively, if you are a Forte Express customer, you can use Express to build an application to create a database schema and populate tables.

In the OrganizationDatabase application, the organization consists of a set of employees and departments arranged in a typical corporation hierarchy. An employee is a person who belongs to a department. An employee can also be a manager of other employees. Each employee has a set of attributes such as name, badge number, database user name and password. The employees also have a set of roles assigned to them. A role is a text value that describes the employee's job. For example, there may be roles for a Manager, Clerk, or Programmer. An employee can have multiple roles.

Departments are divisions within the organization. They are arranged into a hierarchical structure where a department can be the parent of a set of departments. Employees who are members of a department are also members of all the departments that are direct or indirect parents of the department. Each department has a manager, who is the head of the department.

When you run OrganizationDatabase, it creates the following tables in your database (described here in Oracle's SQL):

Employee Table

Name	Type
NAME	VARCHAR2(32)
BADGENO	NUMBER(38)
MANAGER	NUMBER(38)
DEPARTMENT	NUMBER(38)
USERNAME	VARCHAR2(32)
PASSWORD	VARCHAR2(32)

Department Table

Name	Type
CODE	NUMBER(38)
NAME	VARCHAR2(32)
HEAD	NUMBER(38)
PARENT	NUMBER(38)
DESCRIPTION	VARCHAR2(100)

Roles Table

Name	Type
ID	NUMBER(38)
NAME	VARCHAR2(32)
DESCRIPTION	VARCHAR2(100)

EmployeeRoles Table

Name	Type
EMPLOYEE	NUMBER(38)
ROLEID	NUMBER(38)

The EmployeeRoles table contains the *many to many* relationship between employees and roles. EMPLOYEE is an employee identifier, and ROLEID is a role identifier.

Control Table

Name	Type
ROLEID	NUMBER(38)
DEPARTMENTID	NUMBER(38)

The Control table provides persistent storage for the allocation of the next available role and department identifiers. Because the number stored is always the current maximum, it is incremented and then allocated.

Pex Files orgdb/orgdbacc.pex.

Special Requirements Database connection.

To run the OrganizationDatabase application:

1. In the Repository Workshop, import the following file into your repository:

```
%FORTE_ROOT%\install\examples\conductr\orgdb\orgdbacc.pex
```

2. Select the newly imported OrganizationDatabase plan, and click the **Run** icon to run the application.
3. The first window prompts you for database information. Fill in the fields with valid information, and click Logon.
4. Insert, update, and delete data, and import and export your records as described below in the following section, "OrganizationDatabase Application Details."

OrganizationDatabase Application Details

The OrganizationDatabase application consists of a main window containing a menu bar and the following three tab pages:

- Employees
- Departments
- Roles

Each tab page contains a list of entries of the appropriate type. For example, the Employees tab page contains employee entries. Double-clicking the first column in any of the lists opens a dialog containing the details of the record. You can edit any field in the details dialog to change the record. Click **OK** to write your changes to the database.

The following sections provide detailed information about the commands on the menu bar and the individual tab pages.

Menu Bar The menu bar on the main window contains the following commands:

File >
    Print
    Print setup
    Exit

Database >
    Empty
    Import
    Export

The **File > Print** and **File > Print setup** commands allow standard printing of the contents of the window. The **File > Exit** command exits the program. The **Database > Empty** command allows you to delete all the records in the database tables. As this is a destructive operation, a confirmation dialog allows you to change your mind.

The **Database > Export** and **Database > Import** commands allow you to save the database to a text file and to reload it. The **Import** command also allows you to import a foreign database, provided it is in the appropriate text format.

The text file format consists of a series of lines. Each line represents one entity in the database (employee, department, and role). The line starts with a string followed by a colon. The string specifies the entity type, and can be either EMPLOYEE, DEPARTMENT or ROLE.

Depending on the type string, the rest of the line contains the attributes for that entity. For an EMPLOYEE record, the fields are:

- Employee name (enclosed in double quotes if it contains spaces)
- Badge number (integer)

- Badge number of manager (integer)
- Department number (integer)
- A set of integers separated by commas representing the identifiers for the roles assigned to the employee

For a DEPARTMENT record, the fields are as follows. Each department has a unique code.

- Department name (enclosed in double quotes if it contains spaces)
- Department description (enclosed in double quotes if it contains spaces)
- Badge number of department head (integer)
- Number for parent department (integer)
- Number for this department (integer)

For roles, the fields are as follows. Each role has a unique, integral ID.

- Role name (enclosed in double quotes if it contains spaces)
- Role description (enclosed in double quotes if it contains spaces)
- Role identifier (integer)

Each line is terminated by a newline character. The set of roles must come before the employees in the file, as the employee records refer to them by their IDs. The departments and employees can appear in any order.

When such a file is imported, the IDs and codes are reset to the next in sequence in the current database. This is necessary because some codes may already be used, and there would be a conflict if the same codes were stored.

The following example is an example of an exported text file, which represents the data used by the Advanced Expense Reporting application:

```
#
# Organization database exported on 29-Jul-1997 12:35:23
#
ROLE: "Manager" "Manages employees" 2
ROLE: "Accountant" "Performs accounting tasks" 3
ROLE: "Auditor" "Performs auditing tasks." 4
ROLE: "Employee" "Works in a full time permanent position.” 1
DEPARTMENT: "General" "This company doesn't have departments" 0 0 1
EMPLOYEE: "Charlotte" "Charlotte" "Charlotte" 3 1 1 1,2
EMPLOYEE: "Wilbur" "Wilbur" "Wilbur" 4 2 1 1
EMPLOYEE: "Winnie" "Winnie" "Winnie" 5 3 1 1
EMPLOYEE: "Stuart" "Stuart" "Stuart" 6 3 1 1
EMPLOYEE: "Ernie" "Ernie" "Ernie" 7 3 1 1,4
EMPLOYEE: "Nick" "Nick" "Nick" 8 3 1 1,3
EMPLOYEE: "Celeste" "Celeste" "Celeste" 9 3 1 1,3
EMPLOYEE: "Alice" "Alice" "Alice" 1 1 1 1,2
EMPLOYEE: "George" "George" "George" 2 1 1 1,2
```

Employees Tab Page The Employees tab page shows a list of all the employees stored in the database. To create a new employee, invoke the popup menu on the list and select New. Fill in the details of the employee and click OK. The new employee details are validated, the new record is written to the database, and then it is added to the list of existing employees.

All employees must have unique names and badge numbers. The department and manager may be selected from drop lists. If there are no departments available, then you must create them using the Departments tab. The first employee added to the database does not have a manager available. The manager can be added later if necessary.

The list of roles in the New Employee dialog allows the roles allocated to the employee to be assigned. The roles available are shown in a drop list. If there are no roles available, you need to add some of them using the Roles tab page.

The popup menu in the employees list contains menu entries named for each employee. Each employee menu contains a submenu with two options: **Delete** and **Open.** Selecting **Open** is equivalent to double-clicking on the first column, providing another way to edit the employee details. Choosing **Delete** allows the employee to be deleted from the database. Deleting an employee may affect other employees (for example, the employee may be a manager of a set of subordinates). The **Delete** command presents a dialog allowing you to reassign the subordinates to another manager or delete them.

Departments Tab Page The Departments tab page presents a list of the departments stored in the database. The same menus available for the manipulation employees are available for the employees. Double-clicking an existing department allows the department details to be edited.

To create a new department, invoke the popup menu and choose **New**. To delete a department, invoke the popup menu and choose **Delete**. Deleting a department affects any child departments and any employees who are members of the department. You can delete the child departments or reassign them to another department. Similarly, you can delete any affected employees or reassign them to another department.

Roles Tab Page The Roles tab page shows the available roles. Each role has a name and description. Roles are assigned to employees. The manipulation of roles is similar to the other tabs. Invoke the popup menu to create new roles and to delete existing roles. Deleting a role requires removing it from the employees who have the role.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X