Java™ Enterprise Edition 5 Deployment API Specification, Version 1.2

Jerome Dochez

Part No. 8xx-xxxx-xx
June 2,2006, Version 1.2

Specification: JSR-000088 Java(tm) EE Application Deployment ("Specification")

Version: 1.2

Status: Maintenance Release 2

Release: 8 May 2006

LIMITED LICENSE GRANTS

1. _License for Evaluation Purposes_. Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. _License for the Distribution of Compliant Implementations_. Sun also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

3. _Pass-through Conditions_. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Sun's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.

4. _Reciprocity Concerning Patent Licenses_.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b With respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Sun that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c Also with respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Sun that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. _Definitions_. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Sun's source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Sun which corresponds to the Specification and that was available either (i) from Sun's 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Sun if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED IN ANY WAY TO YOUR HAVING, IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

If you provide Sun with any comments or suggestions concerning the Specification ("Feedback"), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Rev. April, 2006

Sun/Final/Full

CHAPTER 1

Java™ EE Deployment API

This is the specification of the Java ™2 Enterprise Edition Deployment API. The Deployment architecture defines the contracts that enable tools from multiple providers to configure and deploy applications on any Java EE platform product. The contracts define a uniform model between tools and Java EE platform products for application deployment configuration and deployment. The Deployment architecture makes it easier to deploy applications: Deployers do not have to learn all the features of many different Java EE deployment tools in order to deploy an application on many different Java EE platform products.

1.1 Overview

The Deployment architecture defines implementation requirements for both tools and Java EE platform products. The primary responsibilities of a tool are

- To access the Java EE application archive.
- To display for editing the deployment configuration information retrieved from the Java EE platform product.

The Java EE platform product's primary responsibilities are to

- Generate the product-specific deployment configuration information.
- To deploy the application.

The Deployment architecture uses the JavaBeans™ architecture for the components that present the dynamic deployment configuration information required by a provider's Java EE platform product. The JavaBeans architecture

was chosen because of its versatility in providing both simple and complex components, as well as its platform neutrality. Beans enable the development of simple property sheets and editors, as well as sophisticated customization wizards for guiding the Deployer through the application deployment configuration steps.

1.2 Scope

The API in this specification describes

- A minimum set of facilities, called a plugin, that all Java EE Platform Product Providers must provide to Deployment Tool Providers so that portable Java EE applications can be deployed to the Product Provider's platform
- A minimum set of facilities that all Tool Providers must provide in order to interact with the Product Provider's plugin.

This API describes two of the three deployment processes described in (*the Java EE platform specification*), installation and configuration. The third process, execution, is left to the Platform Product Provider.

We expect that Java EE product providers will extend these base facilities in their own deployment tools, thus allowing competition with other products on various quality of service aspects. Platform Product Providers may choose to make their extensions available to other tool providers or not.

1.2.1 Relationship to the J2EE Management Specification (JSR-77)

Deployment is an integral part of platform management. It depends on management functionality to start deployed applications, stop deployed applications, report the status of applications, and the like. We determined, however, that Java EE platform deployment and management should be addressed in separate JSRs, because of the ways in which these two topics need to be addressed. J2EE platform management needs to be defined as a metadata model and not as an API in order to best address the issues of interoperability with different management systems and protocols. Deployment, on the other hand is best addressed as an API. It is expected that the Platform Product Provider will integrate the Deployment API with its management model implementation. This

specification describes its interactions with the J2EE Platform Management Model.

1.2.2 Replacing a Java EE Application

We recognize that over time applications evolve and need updates of various types. The Java EE specification does not currently address this issue, nor does it prohibit the Platform Product Providers from doing so.

We believe that this API provides a sufficient infrastructure to enable Platform Product Providers to continue providing application update solutions that are appropriate for their implementations. In addition this specification defines a very basic type of application redeployment. A redeploy method is provided. It is an optional feature for the Platform Product Provider to implement.

1.3 Organization

- Chapter 2, “Roles”, describes the responsibilities of the various implementors of this specification.
- Chapter 3, “Interface Overview”, provides a short description of each interface in the API.
- Chapter 4, "DeploymentManager", discusses the functions and responsibilities of the deployment manager.
- Chapter 5, "Deployment Configuration Components", describes the mechanisms for creating and collecting the deployment configuration information.
- Chapter 6, "Deployment Target", describes an object used to represent a server.
- Chapter 7, "DeploymentTargetID", describes a structure used to identify deployed applications.
- Chapter 8, "ProgressObject", describes the object used to monitor and report the status of a deployment action.
- Chapter 9, "DeploymentManager Discovery", describes the discovery mechanism for acquiring a platform provider’s DeploymentManager.
- Chapter 10, “Exceptions”, describes the exception types of the Deployment

API.

1.4 Object Interaction Diagram Notation

Several object interaction diagrams (OID) are presented in this document. The diagrams contain a mix of API class names and method signatures, with general descriptive information about the interactions. The descriptive information identifies vendor-specific facilities that are needed to support the deployment activities, and additional actions that need to occur in relation to the diagram but whose details are outside the scope of the drawing.

The notation used in the diagrams is as follows:

- Plain font text is used for class names and method signatures.
- Italic font text is used to denote roles such as Deployer, Tool, Java EE Platform Product and to note vendor specific facilities and describe general actions.
- A plain text word in a box represents a class.

1.5 Acknowledgments

This specification was developed under the Java Community Process 2.0 as JSR-88. It includes contributions from many partner companies, as well as groups at Sun. We would like to thank the members of the JSR-88 Expert Group in particular for their contributions:

- Skylight Systems - Aaron Mulder
- WebGain - Mark Romano and Omar Tazi
- Forte - George Finklang
- Oracle - Gerald Ingalls
- SilverStream - Helen Herold
- Sybase - David Brandow

- BEA- Mark Spotswood and Reto Kramer and Vadim Draluk
- iPlanet - Byron Nevins and Darpan Dinker
- IBM - Michael Fraenkel and Leigh Williamson
- Verge Technologies Group Inc - Jason Westra
- IONA - David Hayes

2

Roles

This chapter describes the roles and responsibilities specific to the deployment architecture.

2.1 Java EE Product Provider

The Java EE Product Provider is the implementor and supplier of a Java EE compliant product. A Java EE Product Provider is typically an operating system vendor, database system vendor, application server vendor, or web server vendor.

The Java EE Product Provider is responsible for providing an implementation of the interfaces defined in the `javax.enterprise.deploy.spi` package . A vendor's implementation of this package will be referred to as the plugin or server plugin.

The product must be able to communicate with any third-party deployment tool that adheres to this specification.

The Product Provider is responsible for implementing

- A deployment manager.
- Deployment factories, for accessing their product's deployment manager.
- The deployment configuration components for their product.

2.2 Tool Provider

The Tool Provider is the implementor and supplier of software tools that can be used in the development and packaging of application components, and the deployment, management, or monitoring of applications. A Tool Provider is typically a Java EE Product Provider that provides tools for its product, an Integrated Development Environment (IDE) Provider, or a specialty tool provider.

The Tool Provider is responsible for providing an implementation of the interfaces defined in the `javax.enterprise.deploy.model` package. In addition, the tool must provide a means to discover and interact with a designated Java EE product's deployment manager and to display the configuration beans provided by it.

2.3 Deployer

The Deployer is responsible for configuring and deploying Java EE modules on a specific Java EE product. Deployment is typically a three-stage process:

1. Configuration: The Deployer follows the assembly instructions provided by the Application Assembler and resolves any external dependencies declared by the Application Component Provider.

2. Distribution: The application archive and the deployment configuration information are installed on the servers via the Deployment API.

3. Start execution: The Deployer requests the server to start the application running.

3 Interface Overview

The Deployment API consists of eight packages. Two are implemented by the Tool Provider. Four are implemented by the Java EE Product Provider. Two are provided by this API.

This section provides a quick overview of the interfaces. More detail is provided in the following chapters and in the accompanying API documentation.

3.1 Tool Provider Interfaces

The interfaces for the Tool Provider are in the package, `javax.enterprise.deploy.model`.

- **DeployableObject** represents a Java EE deployable module, an EAR, JAR, WAR, or RAR archive.
- **J2eeApplicationObject** represents a Java EE application, an EAR archive. It is a special type of `DeployableObject`.
- **DDBean** is a component used for introspecting a deployment descriptor. It extracts deployment descriptor information on behalf of the server plugin. It can represent all or part of a module's deployment descriptor.
- **DDBeanRoot** is the topmost `DDBean` for a given module's deployment descriptor.
- **XpathListener** receives `XpathEvents`.

3.1.1 javax.enterprise.deploy.model.exceptions package

- **DDBeanCreateException** is thrown when a DDBean object could not be created for the root of a named XML instance document.

3.2 Tool Provider Classes

- **XpathEvent** is an event that identifies `DDBean objects` being added, removed, or changed in a deployment configuration.

3.3 Tool Provider Interfaces Diagrams

Figure 3.1 shows the relationship of the primary interfaces described above to each other and to a deployment tool. The figure shows the logical relationships of the elements; it is *not* meant to imply a physical partitioning of elements on machines, into processes, or address spaces.

In figure 3.1 the tool is preparing to deploy the Java EE application `mystore.ear`. This EAR file contains a deployment descriptor for itself and two sub-modules: `customer.jar`, an EJB module: `storeFront.war`, a WEB module as a web service.

The tool creates a `J2eeApplicationObject` and associates the `mystore.ear` file with it. The J2eeApplicationObject's function is to provide access to the EAR file's contents. It is an abstract container for its sub-modules and deployment descriptor.

A Java EE module contains one or more deployment descriptors. Each deployment descriptor has associated with it one `DDBeanRoot bean`. The `DDBeanRoot bean` is the reference to the deployment descriptor root.

As Java EE applications can now express their nature or their dependencies through standard java annotations, the existence of XML deployment descriptors is now optional in deployment archives. The possibility of providing partial XML deployment descriptors which completes or overrides annotations also complicates how deployment information can be represented in Java EE 5.

Moreover, the interfaces in the javax.enterprise.deploy.model packages (mainly DeployableObject and DDBean) closely bind deployment descriptors to XML by the use of XPath to identify them. Deployment information changes are represented with events containing XPath values identifying the changed descriptors.

In Java EE 5, the Java EE product provider must merge the java annotations and optional partial or complete deployment descriptors into a model which will support XPath queries and the APIs defined in this chapter.

Some tools may choose to translate the annotations into XML then process the optional xml deployment descriptors to have what is called in the Java EE platform specification a “metadate-complete” XML deployment descriptor. Once this “metadata-complete” XML deployment descriptors is calculated, current APIs can be used as is. Other tools may choose a more dynamic approach of translating annotation into XPath when necessary.

Zero or more `DDBean` objects may be associated with the deployment descriptor. A `DDBean` represents a fragment of a deployment descriptor. The bean contains the text of an XML tag. The server plugin code designates which XML tag information is to be extracted. For example the Platform Product Provider might request the information for all the `env-entry` XML tags for all the session beans in the EJB deployment descriptor. A `DDBean` would be provided for each `env-entry` tag found in the file. The `DDBean` would contain the text for the `env-entry`.

The primary function of the `DDBeanRoot` and `DDBean` beans are to extract data from the deployment descriptor on behalf of the Platform Product Provider’s code.

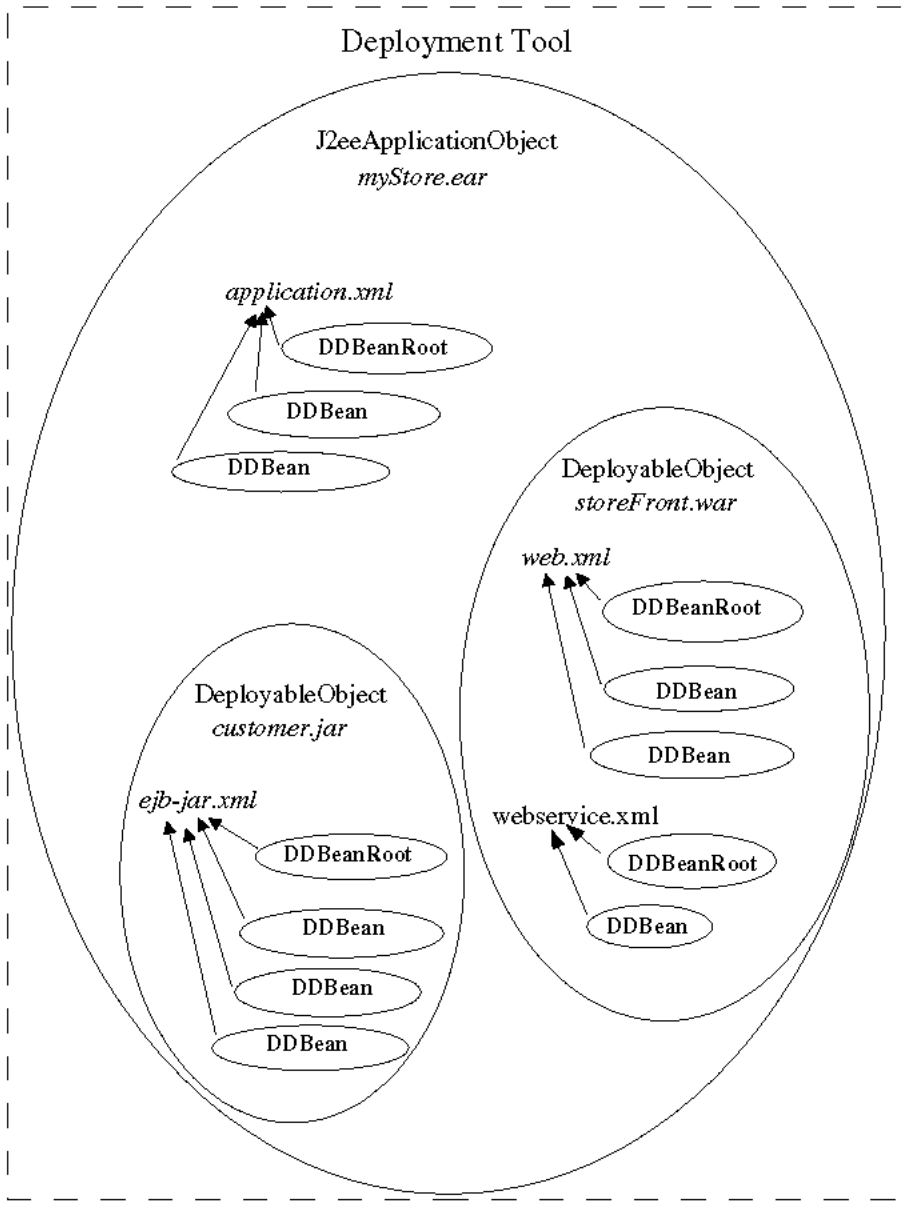


Figure 3.1 Java EE Application

In figure 3.2 the tool is preparing to deploy a stand-alone Java EE module, `storeFront.war`. It creates a `DeployableObject` object instead of a `J2eeApplicationObject` object because it only needs to represent a single module.

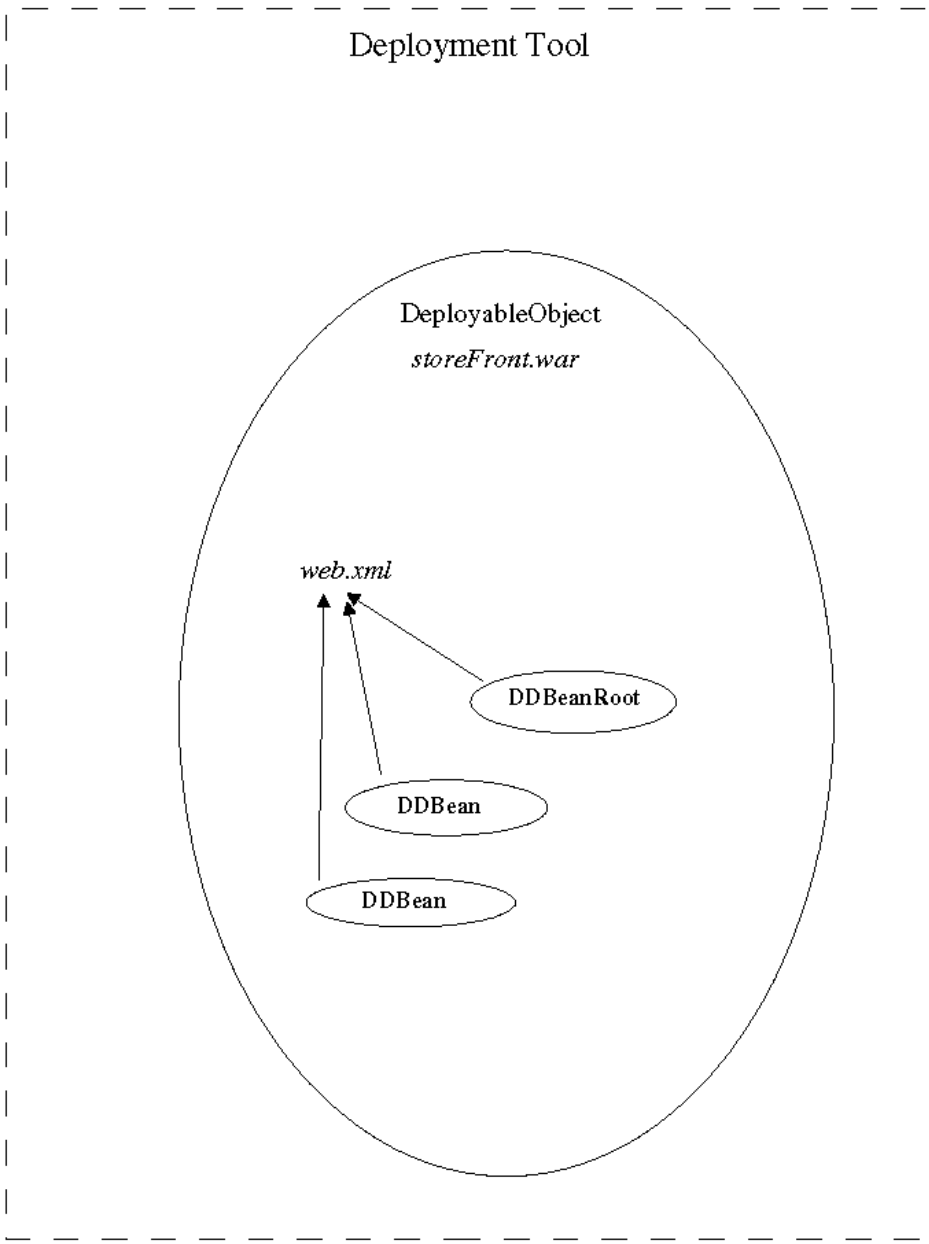


Figure 3.2 Java EE Standalone Module

3.4 Java EE Product Provider Interfaces

The interfaces for the Java EE Product Provider are contained in the package `javax.enterprise.deploy.spi`.

- **DeploymentManager** is the access point for the Tool Provider to a Java EE Platform Product's deployment functionality.
- **DeploymentConfiguration** is the top-level component for deployment configuration information. It is a container for all Java EE platform product-specific configuration objects.
- **DConfigBean** is a JavaBeans component used for conveying platform-product-specific deployment configuration information to the tool. It represents all or part of a deployment descriptor.
- **DConfigBeanRoot** is the topmost `DConfigBean` for a given deployment descriptor.
- **Target** represents an association between a server or group of servers and a location to deposit a Java EE module that has been properly prepared to run on the server or servers.
- **TargetModuleID** is a unique identifier associated with a deployed application module. Each `TargetModuleID` represents a single module deployed to a single server target.

3.4.1 javax.enterprise.deploy.spi.factories package

- **DeploymentFactory** is a deployment driver for a Java EE platform product. It returns a `DeploymentManager` object that represents a connection to a specific Java EE platform product.

3.4.2 javax.enterprise.deploy.spi.status package

- **ProgressObject** tracks and reports the progress of potentially long-lived deployment activities.
- **ProgressEvent** is an event that indicates a status change in a deployment activity.
- **DeploymentStatus** is an object that contains detailed information about a status event.
- **ProgressListener** receives progress events.
- **ClientConfiguration** is a JavaBeans object that installs, configures and executes an application client.

3.4.3 javax.enterprise.deploy.spi.exceptions package

- **ConfigurationException** is thrown when the `ConfigBean` could not be created.
- **DeploymentManagerCreationException** is thrown when a `DeploymentManager` could not be created by the `DeploymentFactory`.
- **InvalidModuleException** is thrown when the Java EE archive module type is unknown by the `DeploymentManager`.
- **TargetException** is thrown when the `Target` is unknown by the `DeploymentManager`.
- **BeanNotFoundException** is thrown when the child `ConfigBean` could not be found by the parent `ConfigBean`.
- **DConfigBeanVersionUnsupportedException** is thrown when the DConfigBeans for a particular Java EE platform verions can not be provided by the platform.

- **ClientExecuteException** is thrown when the application client run environment could not be setup properly.

3.5 Java EE Product Provider Interfaces Diagram

Figure 3.3 shows the relationship of the primary interfaces described in section 3.2 to each other and to a Java EE product . This figure shows the logical relationships of the elements; it is *not* meant to imply a physical partitioning of elements into processes, address spaces or on machines.

In figure 3.3, the `DeploymentFactory` is an object which a tool discovers and uses to retrieves an instance of the Java EE product's `DeploymentManager` object.

The `DeploymentManager` provides the Java EE product's deployment functionality. It is the intermediary between the tool and the server.

A `Target` object is a reference to a server. It can represent a specific application server installation on a single host or it can represent a cluster of servers over many hosts. A `Target` represents an atomic element; for example a `Target` can represent a cluster of servers as a single deployable target. The tool and Deployer need not know the server configuration that a `Target` represents. A `DeploymentManager` can have many deployment targets.

A TargetModuleID object is a reference to a Java EE module that has been deployed to a Target. A module's TargetModuleID is unique within the platform's domain. The association of a TargetModuleID with a module exists only as long as the module is deployed. Once the module is undeployed the TargetModuleID can be reassigned. The TargetModuleID is used by the Deployer to identify the module on which the DeploymentManager is to perform administrative operations, such as start and stop.

A `ProgressObject` provides a means to monitor and report on the status of a deployment operation. There are several operations not depicted in this diagram for which `ProgressObject objects` are provided.

One of the functions of the `DeploymentManager` is to configure Java EE modules for deployment. Some of the configuration information requires input from the Deployer. The `DConfigBean` objects provide the list of external

references and other deployment information the platform needs resolved in order for the module to be deployed.

The `DeploymentConfiguration` object is a container for all the `DConfigBeans` created during a deployment session. A `DConfigBeanRoot` object is associated with a deployment descriptor via the `DDBeanRoot` object. A `DConfigBeanRoot` object can have zero or more `DConfigBean` child objects.

A `DConfigBean` represents deployment information that is associated with XML tag (see section 5.2) information in a deployment descriptor. A `DConfigBean` provides zero or more XPaths that identify the XML information it requires for evaluation. A `DConfigBean` is associated with a `DDBean` (see section 3.3) provided by a tool. A `DConfigBean` object can have zero or more `DConfigBean` child objects.

The primary function of the `DConfigBeanRoot` and `DConfigBean` beans is to tell the tool what data it needs from the deployment descriptor and to allow the Deployer to edit the deployment configuration information the platform requires for the Java EE module.

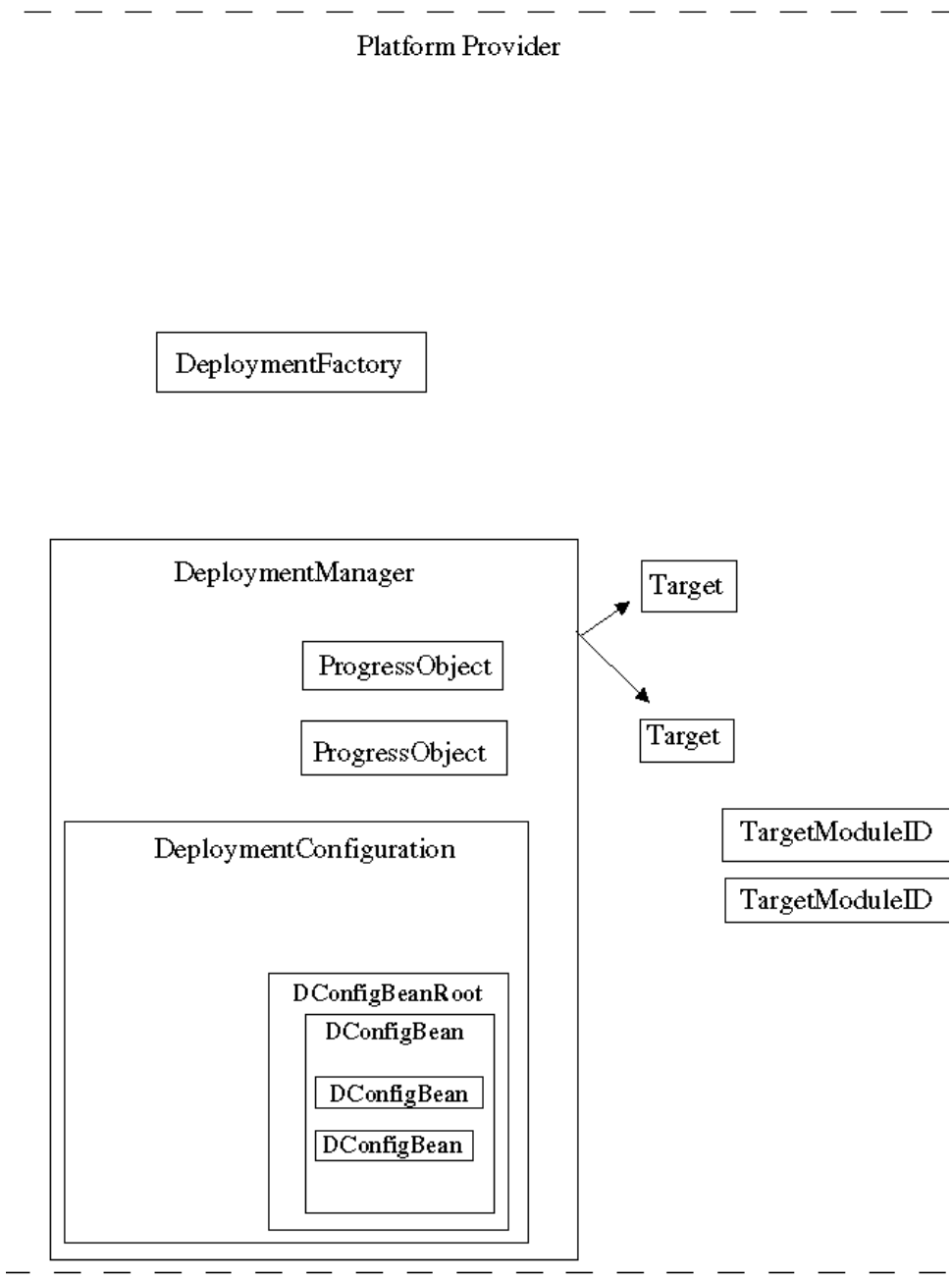

Platform Provider
DeploymentFactory
DeploymentManager
ProgressObject
ProgressObject
DeploymentConfiguration
DConfigBeanRoot
DConfigBean
DConfigBean
DConfigBean
Target
Target
TargetModuleID
TargetModuleID

Figure 3.3 Product Provider Interfaces Diagram

3.6 Shared Classes

There are several constants that both the Tool Provider and Platform Product Provider use. These constants have been grouped into four classes and are provided in the package `javax.enterprise.deploy.shared`.

3.6.1 javax.enterprise.deploy.shared package

- **ModuleType** provides values used to identify the Java EE module type represented by a `DeployableObject` instance.
- **DConfigBeanVersionType** provides values used to identify the Java EE version for which the deployment descriptor beans and deployment configuration beans where compiled.
- **CommandType** provides values use by `DeploymentStatus` to identify the deployment operation it represents.
- **StateType** provides values use by `DeploymentStatus` to identify the state of the deployment operation.
- **ActionType** provides values use by `DeploymentStatus` to identify if a cancel or stop action on the current operation is being performed.

3.6.2 javax.enterprise.deploy.shared.factories package

- **DeploymentFactoryManager** is a central registry for DeploymentFactory objects. The tool discovers the DeploymentFactory objects in a Product Provider's supplied JAR file and registers them with the DeploymentFactory-

Manager. The tool contacts the DeploymentFactoryManager when it requires a DeploymentManager.

3.7 Environment Requirements

Each version of the Java 2 Platform Enterprise Edition Specification defines the Java Compatible™ runtime environment it requires. It is a version of the Java 2 Platform, Standard Edition (J2SE). This specification requires its runtime environment to be the same J2SE edition the platform requires. This information can be found in the platform specification in the section titled, "Container Requirements".

Tools must be able to access the `DeploymentManager`, `DConfigBeans` and helper classes through the classpath or via a classloader.

3.7.1 Tool's Security Permission Set

The `DeploymentManager` must have a minimum set of security permissions in the tool's environment in order to perform its functions. They are listed below.

TABLE 3-1 Security Permission Set

Security Permission	Target	Action
java.lang.RuntimePermission	loadLibrary	
java.net.SocketPermission	*	connect
java.net.SocketPermission	localhost:1024-	accept,listen
java.io.FilePermission	*	read/write
java.util.PropertyPermission	*	read

4 DeploymentManager

The `DeploymentManager` is a service that enables Java EE applications to be deployed to Java EE platform products . It is a deployment tool's access point to a product's deployment functionality. The `DeploymentManager` provides administrative operations for

- Configuring an application.
- Distributing an application.
- Starting the application.
- Stopping the application
- Undeploying the application.

4.1 DeploymentManager Requirements

- At least one `DeploymentManager` object must be provided per Java EE product.
- The `DeploymentManager` must be able to distribute a configured Java EE module to the designated targets.
- A `DeploymentManager` can run either *connected to* or *disconnected from* its Java EE product. A `DeploymentManager` running disconnected from its Java EE product can only configure modules but not perform administrative operations. It might not have access to any product resources. If any of the administrative operations, distribute, start, stop, undeploy, or redeploy are called, an `IllegalStateException` must be thrown. A disconnected `DeploymentManager` is acquired by calling the single argument method `DeploymentFactory.getDisconnectedDeploymentManager(name)`.

A connected DeploymentManager is associated with a specific Java EE product instance. It is identified by a URL and may require a valid user name and password. This DeploymentManager can use the product resources to assist in the resolution of deployment configuration information and can execute all administrative operations.

A DeploymentManager running in connected mode can be notified by the tool to run in disconnected mode. This notification signals to the DeploymentManager that it may release any Java EE resource connections it had established during deployment configuration and clean up resources. It should allow any active operations to finish processing. The DeploymentManager must throw an IllegalStateException if any administrative operations are called when running in disconnected mode.

- The DeploymentManager processes only properly packaged Java EE application or stand-alone module archives (EAR, JAR, WAR, and RAR) files. It does not participate in the predeployment assembly or packaging of the archives.

4.2 DeploymentManager Methods

- **getTargets** returns the list of server targets to which this DeploymentManager supports deployment.
- **getAvailableModules** returns the list of all Java EE modules available on a designated server target. The module may or may not currently be running.
- **getRunningModules** returns the list of all Java EE modules currently running on a designated target server.
- **getNonRunningModules** returns the list of all Java EE modules currently deployed but not running on a designated target server.
- **createConfiguration** returns the object that can evaluate and generate the Java EE product's application runtime configuration information.
- **distribute** moves the complete deployment bundle, module, configuration data and any additional generated code to the target. The J2EE 1.4 distribute method was deprecated with the Java EE 5 release and a new distribute method was added to the DeploymentManager interface.
- **start** makes an application runnable and available to clients. This operation is valid for TargetModuleIDs that represent a root module. A root TargetMod-

uleID has no parent. The root TargetModuleID module and all its child modules will be started. A child TargetModuleID module cannot be individually started. If the application is currently running no action should be taken and no error should be reported. The start operation is complete only when this action has been performed for all the modules.

- **stop** makes a running application unavailable to clients and stopped. This operation is valid for TargetModuleIDs that represent a root module. A root TargetModuleID has no parent. The root TargetModuleID module and all its child modules will be stopped. A child TargetModuleID module cannot be individually stopped. If the application is currently not running, no action should be taken and no error should be reported. The stop operation is complete only when this action has been performed for all the modules.

- **undeploy** removes the application from the target. This operation is valid for TargetModuleIDs that represent a root module. A root TargetModuleID has no parent. The root TargetModuleID module and all its child modules will be undeployed. A child TargetModuleID module cannot be undeployed. The root TargetModuleID module and all its child modules must be stopped before they can be undeployed. The undeploy operation is complete only when this action has been performed for all the modules.

- **isRedeploySupported** designates whether this Java EE product provides application redeployment functionality. A value of true means it is supported.

- **redeploy** is an *optional* operation. Redeploy replaces a currently deployed application with an updated version. The runtime configuration information for the updated application must remain identical to the application it is updating.

  When an application update is redeployed, any transition of clients from the existing application to the application update must be transparent to the client.

  This operation is valid for TargetModuleIDs that represent a root module. A root TargetModuleID has no parent. The root TargetModuleID module and all its child modules will be redeployed. A child TargetModuleID module cannot be individually redeployed. The redeploy operation is complete only when this action has been performed for all the modules.

- **release** signals to the DeploymentManager that the tool does not need it to continue running connected to the Java EE product. This is a signal from the tool that it wants to run in disconnected mode or that the tool is preparing to shutdown.

When release is called, the `DeploymentManager` cannot accept any new operation requests. It can release any Java EE resource connections it had established during deployment configuration and clean up resources. It should finish processing any active operations.

- **getDefaultLocale** returns the default locale supported by this implementation. A default locale must be provided.
- **getCurrentLocale** returns the active locale of this implementation. A current locale must be provided.
- **setLocale** set the active locale for this implementation. Support for locales other than the default locale is optional.
- **getSupportedLocales** returns a list of supported locales of this implementation. At minimum it must return the default locale.
- **isLocaleSupported** returns *true* if the specified locale is supported and *false* if it is not.
- **getDConfigBeanVersion** returns the Java EE platform version number for which the deployment configuration beans are provided.
- **isDConfigBeanVersionSupported** returns true if the deployment configuration beans support the Java EE platform version specified otherwise it returns false.
- **setDConfigBeanVersion** sets the deployment configuration beans to the Java EE platform version specified.

4.3 Starting and Stopping Applications

The time at which an application's running environment is initialized or shut down is not specified. A vendor may choose to initialize an application's running environment when the archive is distributed to the system or wait until the start action is called. The only requirement is that the application is not available to clients until the start action is called. Similarly a vendor may choose to shut down a running application's environment when stop is called or wait until undeploy is called. The only requirement is that the application is made unavailable to clients when the stop action is called.

4.4 Internationalization

Tool Providers and plugin providers may choose to offer internationalized Deployment API implementations to their users. Support for locales other than the default locale is not required. A locale setting is in effect for all the Deployment API subpackages in the provider's implementation for the duration of a deployment session.

4.5 Object Interaction Diagrams for DeploymentManager

This section contains object interaction diagrams (OID) that illustrate the interaction of the parties that participate in an application deployment. The diagrams illustrate a hypothetical deployment session between a Deployer, a tool, and a Java EE product's `DeploymentManager`. Where possible the corresponding method calls and data types are used. A general description of the interaction is provided for those action that are implementation-specific.

The order of the interactions listed should be considered illustrative of an implementation rather than prescriptive.

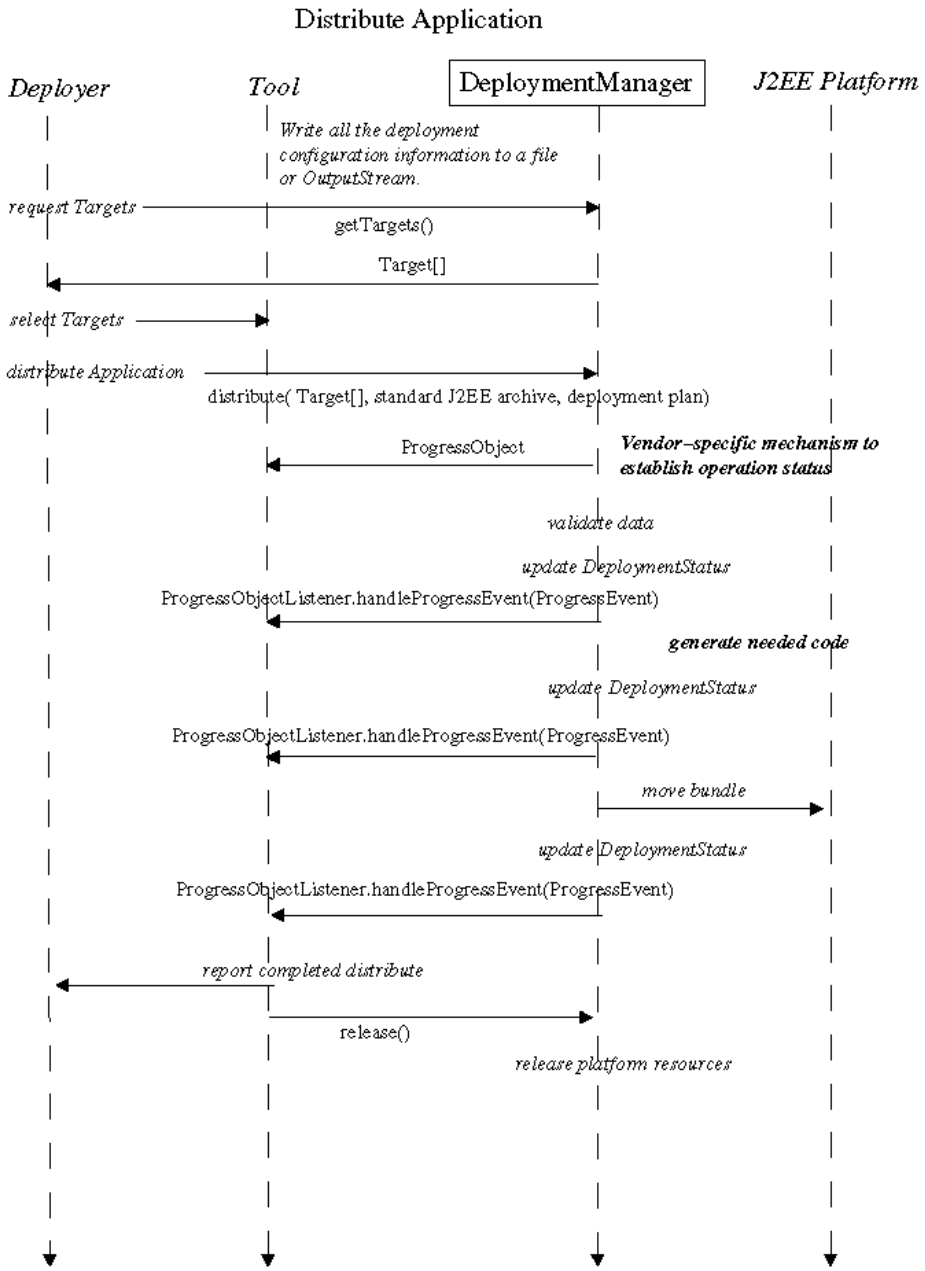


Figure 88Info.4.1 Distributing an Application

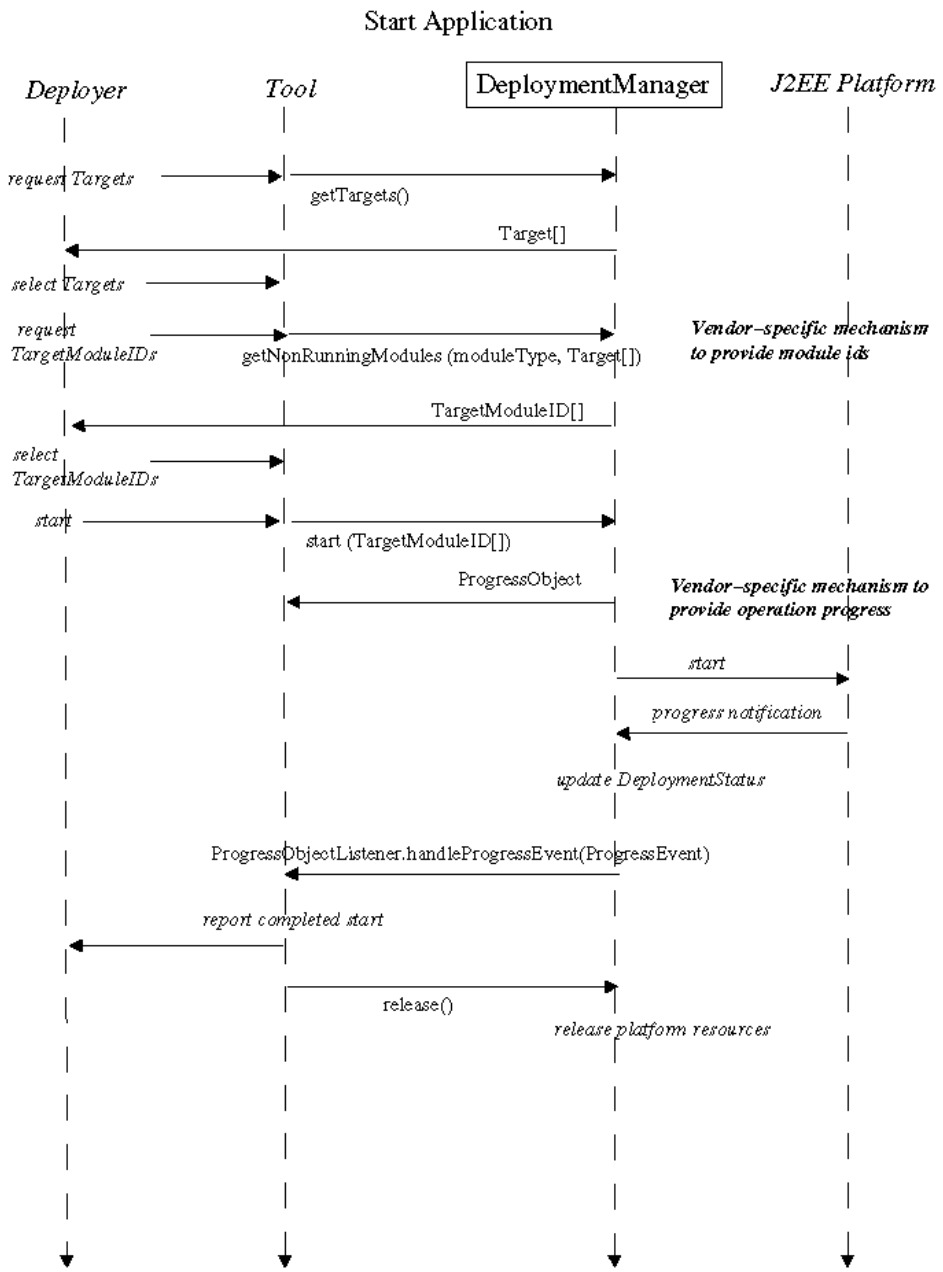


Figure 88Info.4.2 Starting an Application

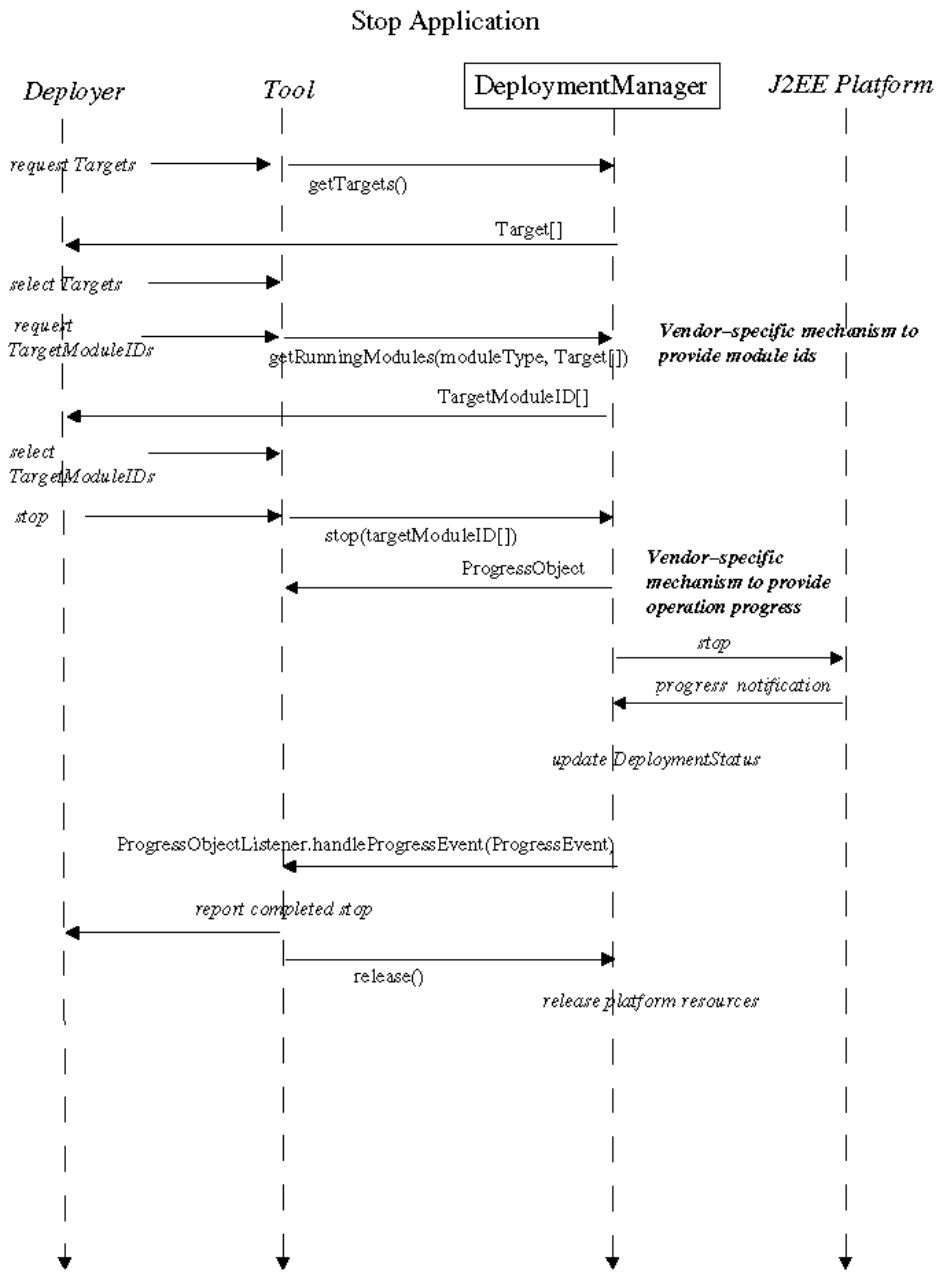


Figure 88Info.4.3 Stopping an Application

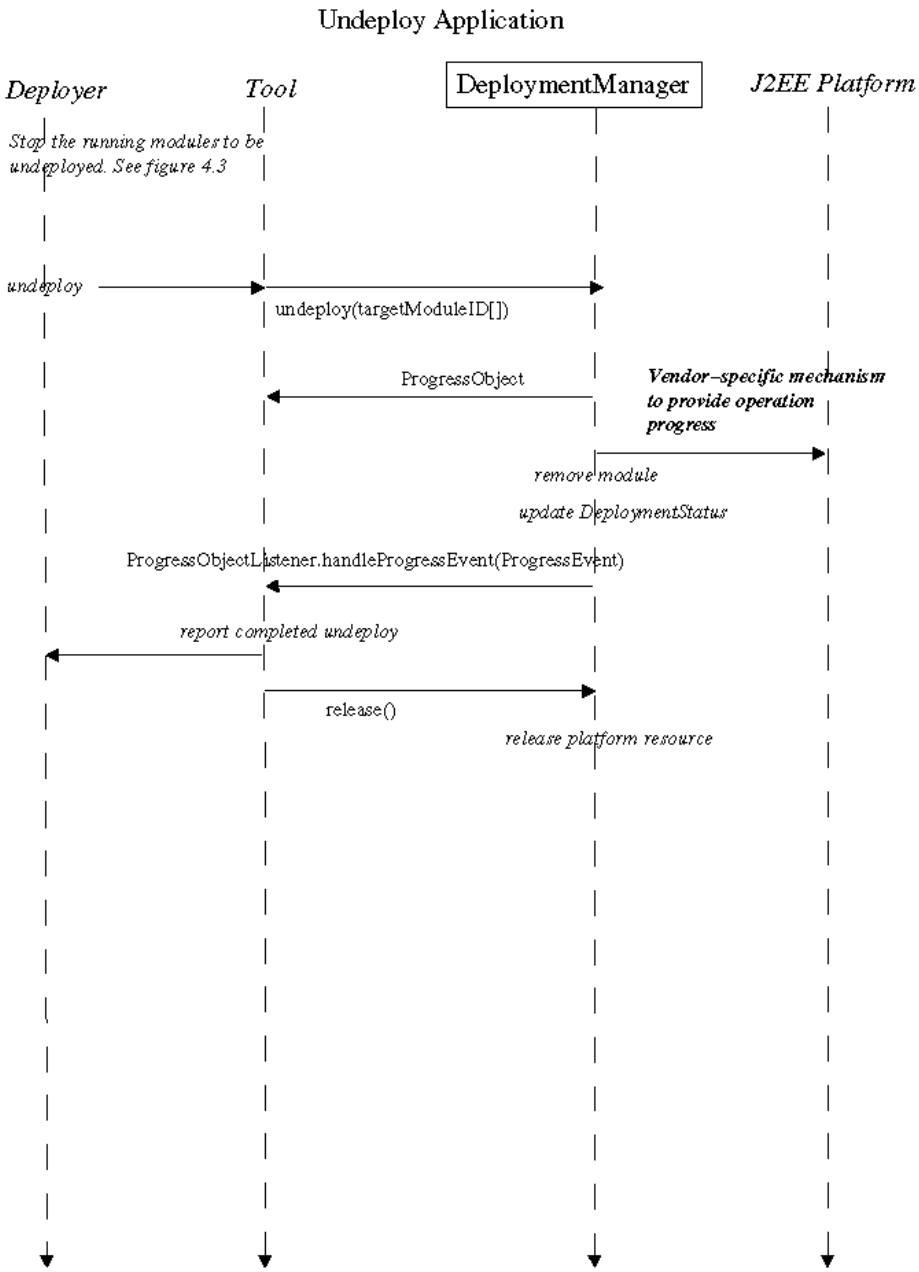


Figure 88Info.4.4 Undeploying an Application

4.6 DeploymentManager and the J2EE Management Specification (JSR 77)

The J2EE Management Specification defines a model for platform management. Deployment is an integral part of J2EE platform management. Deployment depends on management functionality to start installed applications, stop running applications, and report the status of applications. This section describes the recommended mappings of the DeploymentManager functionality to the management model.

4.6.1 Listing Deployed Modules

The management model provides access to all managed objects on the Java EE platform through the J2EE Management EJB component (MEJB). The MEJB is registered in the Java Naming and Directory Interface ™ (JNDI) service. The DeploymentManager may use the MEJB to acquire the list of deployed modules on the platform. See chapter 7, "J2EE Management EJB" in the Java 2 Enterprise Edition Management Specification.

4.6.2 Module Start and Stop

The management model provides a facility for state management of managed objects. This is an optional feature. The state management facility allows applications to start and stop deployed modules. The DeploymentManager may use this facility to start and stop modules that support state management. See chapter 5, "State Management" in the Java 2 Enterprise Edition Management Specification.

5 Deployment Configuration Components

The deployment plan is a file or a stream that contains the deployment configuration information. The data is the Java EE product provider-specific information required in order to deploy the application to the product provider's platform. It is recommended that the file format be XML.

5.1 Runtime Configuration Components

The components that present to the Deployer the dynamic deployment configuration information for a Java EE product are JavaBeans. This specification requires the JavaBeans API Specification version 1.01 be followed for these components.

The deployment configuration components are the contracts between the Java EE Product Provider and the Tool Provider. The components are as follows:

- Deployment Configuration Beans
- Deployment Descriptor Beans

5.1.1 Deployment Configuration Beans

Deployment Configuration Beans (config beans for short) are the components that present to the Deployer the dynamic deployment configuration information: the external dependencies that must be resolved. They are JavaBeans components that enable the deployment information to be presented as simple property sheets with property editors or with custom wizards. The properties are expected to have

default values when possible. (It is important to note that the Deployer's acceptance of the default values does not guarantee optimum performance on the Java EE Provider's product.) The Java EE Product Provider provides the configuration beans for its product.

A config bean represents a logical grouping of deployment configuration information that will be presented to the Deployer. A config bean has a one-to-one relationship to the text of an XML tag in a deployment descriptor through its association with a `DDBean`. A config bean may contain other config beans and regular JavaBeans. It provides zero or more XPaths for XML information it requires. The topmost parent config bean is the root config bean which represents a single deployment descriptor file; it is associated with a `DDBeanRoot`.

Config beans can be represented as a tree structure. The root of the tree is a `DConfigBeanRoot`. The nodes of the tree are D`ConfigBean` objects. A config bean with zero XPaths or one which has no child config beans is an end node in the tree.

An application can contain many Java EE component modules. A component module contains one or more deployment descriptors. A component module must contain a deployment descriptor for its component type. This is the primary deployment descriptor for the module. See the corresponding component specification for details. It may contain other deployment descriptors that extend its basic component functionality. These are secondary deployment descriptors for the module. See the Web Services 1.1 specification.

A `DeploymentConfiguration` object is a container for all `DConfigBeanRoot` objects. They represent primary deployment descriptors. A `DConfigBeanRoot` object is a container for any secondary deployment descriptors in the same component module.

5.1.1.1 DConfigBean Methods

DConfigBean is a bean for configuring a vendor-specific deployment descriptor or a subset of one.

- **getDConfigBean** returns the server-specific configuration bean for a given sub-element of the standard deployment descriptor.
- **getDDBean** returns the `DDBean` storing the concrete deployment descriptor fragment this D`ConfigBean` is configuring.

- **removeDConfigBean** removes a child DConfigBean from this bean.
- **getXpaths** returns a list of XPath strings representing the deployment descriptor information that a DDBean must retrieve
- **notifyDDChange** indicates that the DDBean provided in the event has changed and that this bean or its child beans need to reevaluate themselves.
- **addPropertyChangeListener** supports standard JavaBeans property change notification registration.
- **removePropertyChangeListener** supports standard JavaBeans property change notification de-registration.

5.1.1.2 DConfigBeanRoot Methods

DConfigBeanRoot is a config bean associated with the root of a primary deployment descriptor. A DConfigBeanRoot object may have child DConfigBean objects representing secondary deployment descriptors.

- **getDConfigBean** returns a DConfigBean object for a DDBeanRoot of a secondary deployment descriptor.

5.1.1.3 DeploymentConfiguration Methods

DeploymentConfiguration is a container for all the server-specific configuration information for a single application

- **getDConfigBeanRoot** returns the vendor-specific DConfigBeanRoot for a primary deployment descriptor.
- **getDeployableObject** returns the top-level DeployableObject for this configuration.
- **removeDConfigBean** removes the DConfigBeanRoot and all its children.
- **restore** restores a deployment configuration session that was saved to disk.
- **restoreDConfigBean** restores the designated DConfigBean that was saved to disk.
- **save** writes a deployment configuration session to disk.
- **saveDConfigBean** writes the designated DConfigBean to disk.

5.1.2 Deployment Descriptor Beans

Deployment Descriptor Beans (DD beans for short) are the components that present the text, based upon the XPath string, back to the config bean. They are the mechanism for reading and extracting data from the application's deployment descriptor files. The Tool Provider provides the DD beans for its product.

A DD bean is associated with a deployment descriptor. It can have child DD beans. The topmost DD bean is the root DD bean, which represents a single deployment descriptor file.

A deployable Java EE application can be an EAR file that contains one or more modules or a single stand-alone module (JAR, WAR, or RAR) file. There are separate configuration-related containers for these two categories of deployable modules:

- The `J2eeApplicationObject` object is the container for an EAR file. It is a special type of `DeployableObject` that contains a `DeployableObject` for each module in the archive. It provides accessor methods to access the information in a single `DeployableObject` or a group of them, which are provided by the Tool Provider.
- The `DeployableObject` object is the container for a single module. It maintains references to the deployment descriptor files, the `DDBeanRoot` objects and all the child DD beans for the module.

5.1.2.1 DDBean Methods

`DDBean` is a bean that represents a fragment of a standard deployment descriptor.

- **getChildBean** returns a list of child `DDBean` objects based upon the designated XPath.
- **getRoot** returns the `DDBeanRoot` object of this bean.
- **getText** returns the deployment descriptor text associated with this bean.
- **getId** returns a tool-specific reference for attribute ID on an element in the deployment descriptor.
- **getXpath** returns the original XPath string provided by the DConfigBean.

- **getAttributeNames** returns the list of attribute names associated with the XML element.
- **getAttributeValue** returns the string value of the named attribute.
- **addXpathListener** supports registration of XPath listener objects.
- **removeXpathListener** supports de-registration of XPath listener objects.

5.1.2.2 DDBeanRoot Methods

`DDBeanRoot` is a `DDBean` that represents the root of a deployment descriptor.

- **getDeployableObject** returns the containing `DeployableObject`.
- **getModuleDTDVersion** returns the DTD version number.
- **getDDBeanRootVersion** returns the version number of an XML instance document.This method is replacing the methods DDBeanRoot.getModuleDTDVersion and DeployableObject.getModuleDTDVersion.
- **getFilename** returns the filename relative to the root of the module of the XML instance document this DDBeanRoot represents.
- **getType** returns the deployment descriptor type.

5.1.2.3 DeployableObject Methods

`DeployableObject` is a bean that represents a Java EE module within an EAR file or an independently deployable module.

- **getChildBean** returns a list of `DDBean` objects associated with the designated XPath.
- **getClassFromScop**e returns a class from the component module associated with this deployment descriptor.
- **getModuleDTDVersion** returns the DTD version number of the module's component deployment descriptor file. This method is being deprecated. With the addition of multiple deployment descritors in components for J2EE 1.4 this method is being replaced by `DDBeanRoot.getDDBeanRootVersion`.
- **getDDBeanRoot** returns the `DDBeanRoot` object for the component's primary deployment descriptor.
- **getDDBeanRoot** returns a DDBeanRoot object for the XML instance document named in the input parameter.

- **getText** returns the deployment descriptor text associated with the designated XPath.
- **entries** returns an enumeration of the module's file entries.
- **getEntry** returns the InputStream for the given file entry name.
- **getType** return the module type of this `DeployableObject`.

5.1.2.4 J2eeApplicationObject Methods

`J2eeApplicationObject` is a bean that represents a Java EE application EAR file. It is a special type of `DeployableObject.` It has a `DeployableObject` for each module in the archive.

- **getChildBean** returns a list of `DDBean` objects based upon the designated XPath and module type.
- **getDeployableObject** returns a `DeployableObject` based upon a URI.
- **getDeployableObjects** returns a list of `DeployableObject` objects based upon the designated module type.
- **getModuleUris** returns the module based upon its URI.
- **getText** returns the deployment descriptor text associated with the designated XPath and module type.
- **addXpathListener** supports registration of XPath listener objects by module type.
- **removeXpathListener** supports de-registration of XPath listener objects by module type.

5.2 Multiple Deployment Descriptor Files

A Java EE component module contains one or more deployment descriptor files and zero or more non-deployment descriptor XML instance documents. A module must contain a component specific deployment descriptor file. It may contain one or more deployment descriptor files that define extra functionality on the component for example webservice.xml and it may contain zero or more non-deployment descriptor XML instance documents.

The tool is required to present the server plugin a `DDBeanRoot` object for each deployment descriptor file in the module. Method `DeploymentConfiguration.getDConfigBean` must be called with the `DDBeanRoot` object for the component specific deployment descriptor and method `DConfiBeanRoot.getDConfigBean` must be called with the `DDBeanRoot` object for the deployment descriptor that extends the base component functionality.

The server plugin provider calls method `DeployableObject.getDDBeanRoot` for each non-deployment descriptor XML instance document it requires a `DDBeanRoot` object for.

5.3 UI Contract between Tool and Server Plugin

JavaBean components present the dynamic deployment configuration information for a Java EE plugin to the deployer. The JavaBeans architecture was chosen because of its versatility in providing both property sheets and property editors, as well as sophisticated customization wizards.

The JavaBean GUI mechanism, that a plugin provider implements in order to enable their DConfigBeans to be displayed by a deploy tool is not specified. The plugin provider may choose to provide a Customizer for one DConfigBean, a Property Editor for a complex datatype for the Property Sheet of another DConfigBean, and to use the default Property Editors for the Property Sheet of yet a third DConfigBean. See the JavaBeans API Specification version 1.01.

The manner in which a tool analyzes a DConfigBean and displays it is not specified. It is recommended that any Customizer or Property Editor provided by the plugin vendor take precedence over similar functionality provided by the tool vendor.

It is expected that a Property Editor will be provided by a plugin vendor for any complex datatype in a DConfigBean that is to be edited by the Deployer. The Property Editor should be implemented and made available to a tool according to the guidelines defined in the JavaBeans API Specification version 1.01.

5.4 ModuleType Enumeration Objects

The Java EE module types are provided in the class javax.enterprise.deploy.shared.ModuleType. Its values are:

- **ModuleType.EAR** indicates the module is an EAR archive.
- **ModuleType.EJB** indicates the module is an Enterprise Java Bean archive
- **ModuleType.CAR** indicates the module is an Client Application archive.
- **ModuleType.RAR** indicates the module is an Connector archive.
- **ModuleType.WAR** indicates the module is an Web Application archive.

5.5 Deployment Descriptor Document Version

All deployment descriptors must indicate the document type definition, DTD or XML Schema version being used. The version number resides in a different location in the DTD than in an XML Schema document. Modules packaged using J2EE 1.3 and 1.2 tools are in DTD format. Modules packaged using 1.4 tools are in XML Schema format.

5.5.1 DTD Document

The version number of the an XML DTD based deployment descriptor instance document is defined in the DOCTYPE statement. The DOCTYPE statement contains the version number in the label of the statement.

The format of the DOCTYPE statement is:

<**!DOCTYPE** root_element **PUBLIC** "-//organization//label//language" "location">

- **root_element** is the name of the root document in the DTD.
- **organization** is the name of the organization responsible for the creation and maintenance of the DTD being referenced.
- **label** is a unique descriptive name for the public text being referenced.

- **language** is the ISO 639 language id representing the natural language encoding of the DTD.
- **location** is the URL of the DTD.

An example J2EE deployment descriptor DOCTYPE statement is:

<!DOCTYPE application-client **PUBLIC**
"-//Sun Microsystems, Inc.//DTD J2EE Application Client 1.3//EN"
"http://java.sun.com/dtd/application-client_1_3.dtd">

In this example the label is, "DTD J2EE Application Client 1.3", and the DTD version number is 1.3. A call to `getDDBeanRootVersion` would return a string containing, "1.3".

5.5.2 XML Schema Document

The version number of the an XML Schema based deployment descriptor instance document is defined in the "`version`" attribute on the root element.

```
<application xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/application_1_4.xsd"
    version="1.4">
```

In this example the value of the version attribute is 1.4. A call to `getDDBeanRootVersion` would return a string containing, "1.4".

5.6 DConfigBean Version

Each version of the Java 2 Platform Enterprise Edition Specification defines the Java Compatible™ runtime environment it requires. It is a version of the Java 2 Platform, Standard Edition (J2SE). This specification requires the Java EE Product Provider to provide its Deployment API implementation based upon this runtime environment, and the Tool Provider to support the runtime environment. The DConfigBean Version number is the version number of the Java EE platform for which the APIs were built.

It is required that the same version of the Tool Provider's APIs and the Java EE Provider's APIs interact. It is not required that differing versions of the APIs interact.

5.6.1 DConfigBeanVersionType Enumeration Objects

The platform version number is provided in the class javax.enterprise.deploy.shared.DConfigBeanVersionType. Its values are:

- **DConfigBeanVersion.V1_3** indicates the beans were built for the J2EE 1.3 platform.
- **DConfigBeanVersion.V1_3_1** indicates the beans were built for the J2EE 1.3.1 platform. **This constant should never be used. Use V1_3 instead.**
- **DConfigBeanVersion.V1_4** indicates the beans were built for the J2EE 1.4 platform.
- **DConfigBeanVersion.V5** indicates the beans were built for the Java 5 platform.

5.7 XPath Syntax

XML Path Language (XPath) Version 1.0 is used as the path notation for navigating the hierarchical structure of the deployment descriptor document. Only the AbsoluteLocationPath and RelativeLocationPath elements of the XPath standard are used by this API, and only a subset of these two elements' grammar is used. The XPath Location Step (that is an axis specifier, a node test, and predicates) is not used in the AbsoluteLocationPaths or RelativeLocationPaths specified by the configuration beans in this API. The path element, '.' selects the context node and '..' selects the parent context node. What remains are AbsoluteLocationPaths and RelativeLocationPaths consisting of '.', '..', and XML tags separated by forward slashes (/).

DTD-based and XML Schema-based deployment descriptors have different XPath naming requirements. This is due to the use of namespaces in XML Schema but not in DTD-based deployment descriptors. The namespace feature requires the addition of a namespace prefix to each XML element in the XPath for a Java EE XML Schema-based instance document. Each element in an XPath

must be specifically qualified with the namespace prefix that is bound to the namespace's URI. The format is:

< prefix>:<XML tag>

Prefix is a name associated with the namespace URI. The colon (:) is a separator between the prefix and the XML tag.

A namespace is uniquely identified using a URI. A prefix may be associated with a namespace URI. The reserved attribute **`xmlns`** is used to define a namespace without an associated prefix; the reserved attribute **`xmlns:`** is used to defined a namespace with an associated prefix. A namespace defined without a prefix is treated as part of the default namespace. The element in which the default namespace is specified and all the contents within the element are associated with the XML Schema Namespace. If there is no prefix defined for a namespace, the <prefix>: is not used in the XPath element qualifier, only the <XML tag> is given. Since namespaces are not supported in DTD-based deployment descriptors, the <prefix>: is never used in XPath element qualifier.

The required namespace for J2EE XML Schema-based deployment descriptors is http://java.sun.com/xml/ns/j2ee. There is no required prefix for this namespace. The prefix can either be specified by the creator of the instance document or it can be left unspecified and thus part of the default namespace.

To build a proper XPath string a Java EE Product Provider plugin will need to determine the active namespaces for elements in a deployment descriptor instance document, by analyzing the attributes on the instance document elements.

5.7.1 AbsoluteLocationPath Syntax

An XPath whose first character is a forward slash '/' designates an AbsoluteLocationPath. It starts at the root of the document.

An XPath whose first character is a forward slash '/' may be followed by zero or more fully qualified element names separated by a forward slash. An XPath consisting of a single forward slash designates the document root.

A DTD based deployment descriptor does not use namespaces, thus the plugin provider does not need to determine the active name space and each fully qualified element consists of the XML tag only. In the example below the AbsoluteLocationPath to the two env-entry tags in the EJB deployment descriptor would be:

```
/ejb-jar/enterprise-beans/session/env-entry
```

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <env-entry>
                <env-entry-name>ejb/mail/SendMail
                </env-entry-name>
                <env-entry-type>java.lang.Boolean
                </env-entry-type>
                <env-entry-value>false</env-entry-value>
            </env-entry>
            <env-entry>
                <env-entry-name>event/SignoutEvent
                </env-entry-name>
                <env-entry-type>java.lang.String
                </env-entry-type>
                <env-entry-value>ejb.SignoutHandler
                </env-entry-value>
            </env-entry>
        </session>
    </enterprise-beans>
</ejb-jar>
```

In the example below the plugin vendor would have determined that there is one active namespace. It is defined in the root element of the XML Schema based instance document. The J2EE namespace prefix is defined to be ‘j’ `xmlns:j="http://java.sun.com/xml/ns/j2ee"`. The AbsoluteLocationPath to the two env-entry tags in the EJB deployment descriptor would be:

```
/j:ejb-jar/j:enterprise-beans/j:session/j:env-entry
```

```
<j:ejb-jar xmlns:j="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd
    version="2.1">
    <j:enterprise-beans>
```

```
        <j:session>
            <j:env-entry>
                <j:env-entry-name>ejb/mail/SendMail
                </j:env-entry-name>
                <j:env-entry-type>java.lang.Boolean
                </j:env-entry-type>
                <j:env-entry-value>false</j:env-entry-value>
            </j:env-entry>
            <j:env-entry>
                <j:env-entry-name>event/SignoutEvent
                </j:env-entry-name>
                <j:env-entry-type>java.lang.String
                </j:env-entry-type>
                <j:env-entry-value>ejb.SignoutHandler
                </j:env-entry-value>
            </j:env-entry>
        </j:session>
    </j:enterprise-beans>
</j:ejb-jar>
```

5.7.2 RelativeLocationPath Syntax

An XPath whose first character is not a forward slash ’/’, but that has one or more qualified elements separated by a forward slash, designates a RelativeLocationPath. It starts from the current location in the document.

For example in a DTD based instance document the RelativeLocationPath to the two `env-entry` tags in the EJB deployment descriptor below would be the following assuming that a previous XPath was simply "/", which is a reference to the root of the file.

```
    ejb-jar/enterprise-beans/session/env-entry

<ejb-jar>
    <enterprise-beans>
        <session>
            <env-entry>
                <env-entry-name>ejb/mail/SendMail
                </env-entry-name>
                <env-entry-type>java.lang.Boolean
                </env-entry-type>
```

```
                <env-entry-value>false</env-entry-value>
            </env-entry>
            <env-entry>
                <env-entry-name>event/SignoutEvent
                </env-entry-name>
                <env-entry-type>java.lang.String
                </env-entry-type>
                <env-entry-value>ejb.SignoutHandler
                </env-entry-value>
            </env-entry>
        </session>
    </enterprise-beans>
</ejb-jar>
```

In a XML Schema based instance document with a defined prefix, `j`, the RelativeLocationPath to the two `env-entry` tags in the EJB deployment descriptor below would be the following assuming that a previous XPath was simply "/", which is a reference to the root of the file.

```
    j:ejb-jar/j:enterprise-beans/j:jsession/j:env-entry

<j:ejb-jar xmlns:j="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd
    version="2.1">
    <j:enterprise-beans>
        <j:session>
            <j:env-entry>
                <j:env-entry-name>ejb/mail/SendMail
                </j:env-entry-name>
                <j:env-entry-type>java.lang.Boolean
                </j:env-entry-type>
                <j:env-entry-value>false</j:env-entry-value>
            </j:env-entry>
            <j:env-entry>
                <j:env-entry-name>event/SignoutEvent
                </j:env-entry-name>
                <j:env-entry-type>java.lang.String
                </j:env-entry-type>
                <j:env-entry-value>ejb.SignoutHandler
                </j:env-entry-value>
            </j:env-entry>
```

```
        </j:session>
    </j:enterprise-beans>
</j:ejb-jar>
```

5.7.3 Multiple Namespaces

A Java EE XML Schema based document has one or more defined namespaces. The required namespace for a J2EE deployment descriptor is `http://java.sun.com/xml/ns/j2ee` . A server vendor may define other namespaces which define the data in a `deployment-extension` tag. For exmaple the root element of the document below defines two namespaces, the J2EE namespace and the `foobar.com` namespace. The `foobar.com` namespace is used for elements contained in the `deployment-extension` tag.

In creating an XPath string for the exmaple below it should be noted that there is no prefix defined for the J2EE namespace, so only the element tag is used for the associated elements and a prefix of `foobar` is defined for namespace `foobar.com, xmlns:foobar="http://foobar.com"`, thus the absoluteLocationPath to element `foobar:comment` would be

/web-app/deployment-extension/foobar:comment

From the element above the RelativeLocationPath to `foobar:version` would be:

foobar:product/foobar:version

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:foobar="http://foobar.com"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd http://foobar.com
    http://foobar.com/foobar.xsd"
    version="2.4">
    <servlet>
        <servlet-name>MyInventoryServlet</servlet-name>
        <servlet-class>com.acme.Inventory</servlet-class>
```

```
            <init-param>
                <param-name>debug</param-name>
                <param-value>0</param-value>
            </init-param>
            <init-param>
                <param-name>defaultCompany</param-name>
                <param-value>ToysRUs</param-value>
            </init-param>
            <load-on-startup>1</load-on-startup>
        </servlet>
        :
        :
        :
    <deployment-extension namespace="http://foobar.com"
        mustUnderstand="false">
        <extension-element xsi:type="foobar:mytype">
        <foobar:comment>This component is generated by Foobar company
        </foobar:comment>
        <foobar:product>Foobar Build Environment</foobar:product>
            <foobar:version>100.5</foobar:version>
        </foobar:product>
        </extension-element>
    </deployment-extension>
```

An alternative implementation of the example above is to define the `foobar` namespace in the `deployment-extension` element. The Java EE Product Provider plugin would have had to analyze the attributes on the root element and the `deployment-extension` element in order to generate the XPath strings.

A server plugin could get the `foobar:version` data with the following steps. One, determine the J2EE namespace prefix from the root element. Two, create the AbsoluteLocationPath `/web-app/deployment-extension`. Three, evaluate each `deployment-extension` element for the `foobar` namespace name. Four, determine the prefix of the `foobar` namespace. Five, create theRelativeLocationPath:

```
extension-element/foobar:comment/foobar:product/foobar:version
```

```
        <web-app xmlns="http://java.sun.com/xml/ns/j2ee"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd
        version="2.4">
        <servlet>
            <servlet-name>MyInventoryServlet</servlet-name>
            <servlet-class>com.acme.Inventory</servlet-class>
                <init-param>
                    <param-name>debug</param-name>
                    <param-value>0</param-value>
                </init-param>
                <init-param>
                    <param-name>defaultCompany</param-name>
                    <param-value>ToysRUs</param-value>
                </init-param>
                <load-on-startup>1</load-on-startup>
    </servlet>
            :
            :
            :
    <deployment-extension namespace="http://foobar.com"
        xmlns:foobar="http://foobar.com"
        xsi:schemaLocation=http://foobar.com
        http://foobar.com/foobar.xsd"
        mustUnderstand="false">
        <extension-element xsi:type="foobar:mytype">
            <foobar:comment>This component is generated byFoobar company
            </foobar:comment>
            <foobar:product>Foobar Build Environment</foobar:product>
            <foobar:version>100.5</foobar:version>
        </extension-element>
    </deployment-extension>
</web-app>
```

5.8 Client Applications

The Java EE platform specification leaves the mechanism used to install the application client to the discretion of the Java EE Product Provider. The specification notes that there are a wide range of possibilities; a vendor might allow the application client to be deployed on a Java EE server and automatically made available to some set of clients. A vendor might require the Java EE application bundle containing the application client to be manually deployed and

installed on each client machine or they could have the deployment tool produce an installation package that can be used by each client to install the application client.

It is recommended that an application client DConfigBean be provided that supports the vendor's application client installation mechanism. For example a Java EE product that requires the manual deployment of an application client bundle might request the Deployer to provide a disk location where the bundle will be copied or the bean might inform the Deployer where to retrieve the bundle.

5.9 Object Interaction Diagrams for Deployment Configuration Beans

This section contains an object interaction diagram (OID) that illustrates the interaction of DD beans and configuration beans.

Figure 5.1 illustrates a hypothetical session for generating configuration beans for a Java EE application. The Tool is the control center. DD Beans activity is noted to the left of the Tool, and a Java EE product provider's configuration beans activity is to the right. Two objects, DeploymentConfiguration and DConfigBean, are not listed at the top of the diagram. They appear in the middle of the diagram. Their activity is mapped after they are created.

The order of the interactions listed should be considered illustrative of an implementation rather than prescriptive.

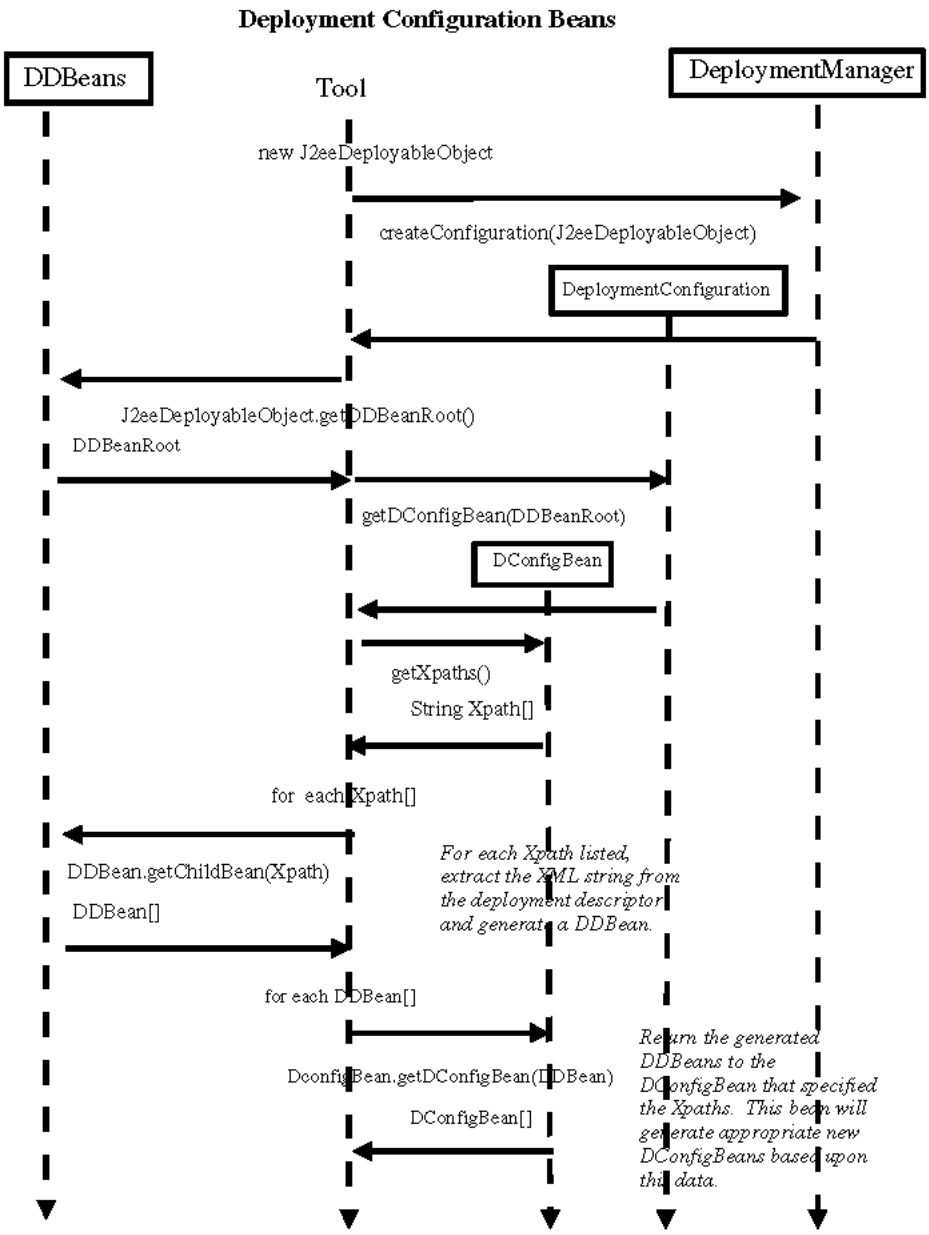

Deployment Configuration Beans
DDBeans
Tool
DeploymentManager
new J2eeDeployableObject
createConfiguration(J2eeDeployableObject)
DeploymentConfiguration
J2eeDeployableObject.getDDBeanRoot()
DDBeanRoot
getDConfigBean(DDBeanRoot)
DConfigBean
getXpaths()
String Xpath[]
for each Xpath[]
DDBean.getChildBean(Xpath)
DDBean[]
For each Xpath listed, extract the XML string from the deployment descriptor and generate a DDBean.
for each DDBean[]
DconfigBean.getDConfigBean(DDBean)
DConfigBean[]
Return the generated DDBeans to the DConfigBean that specified the Xpaths. This bean will generate appropriate new DConfigBeans based upon this data.

Figure 5.1

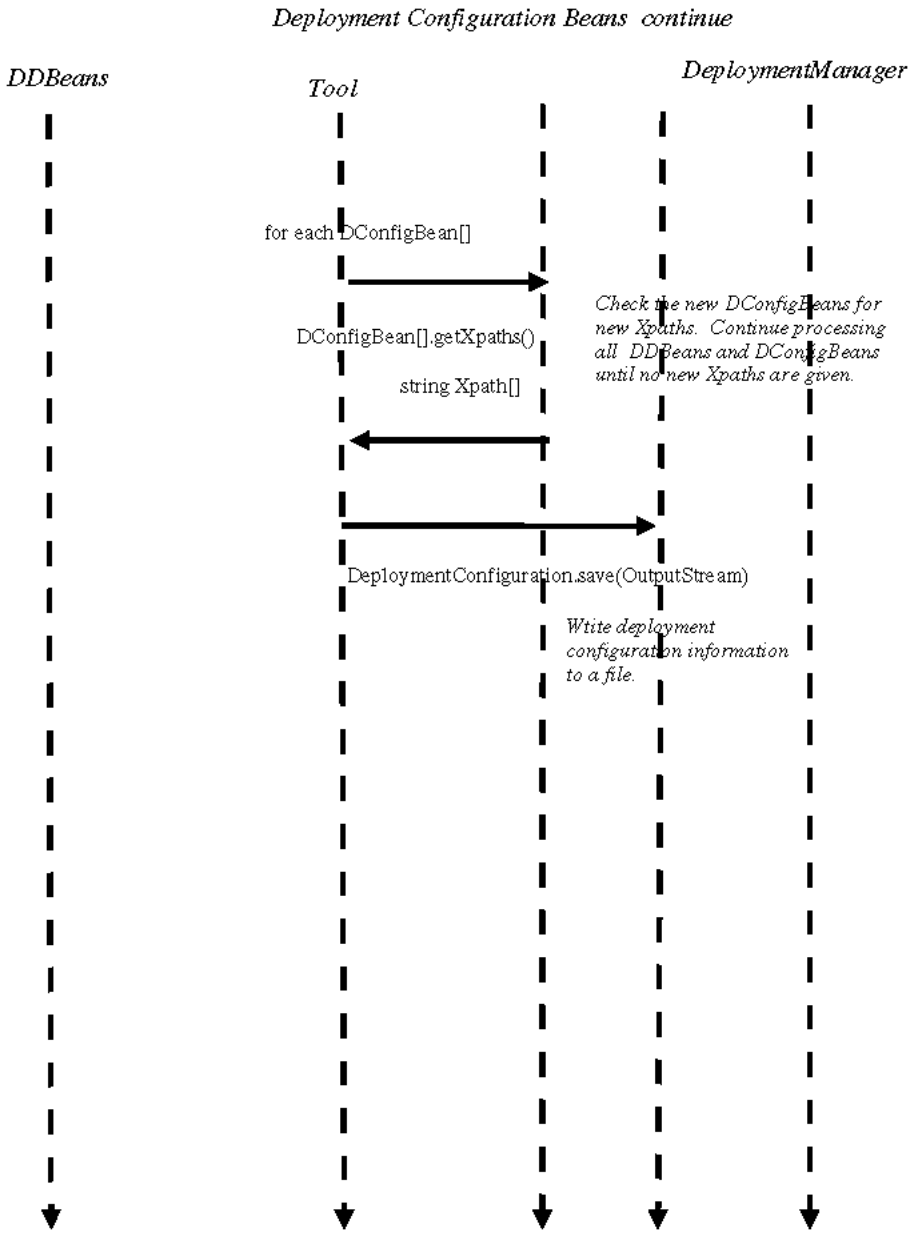
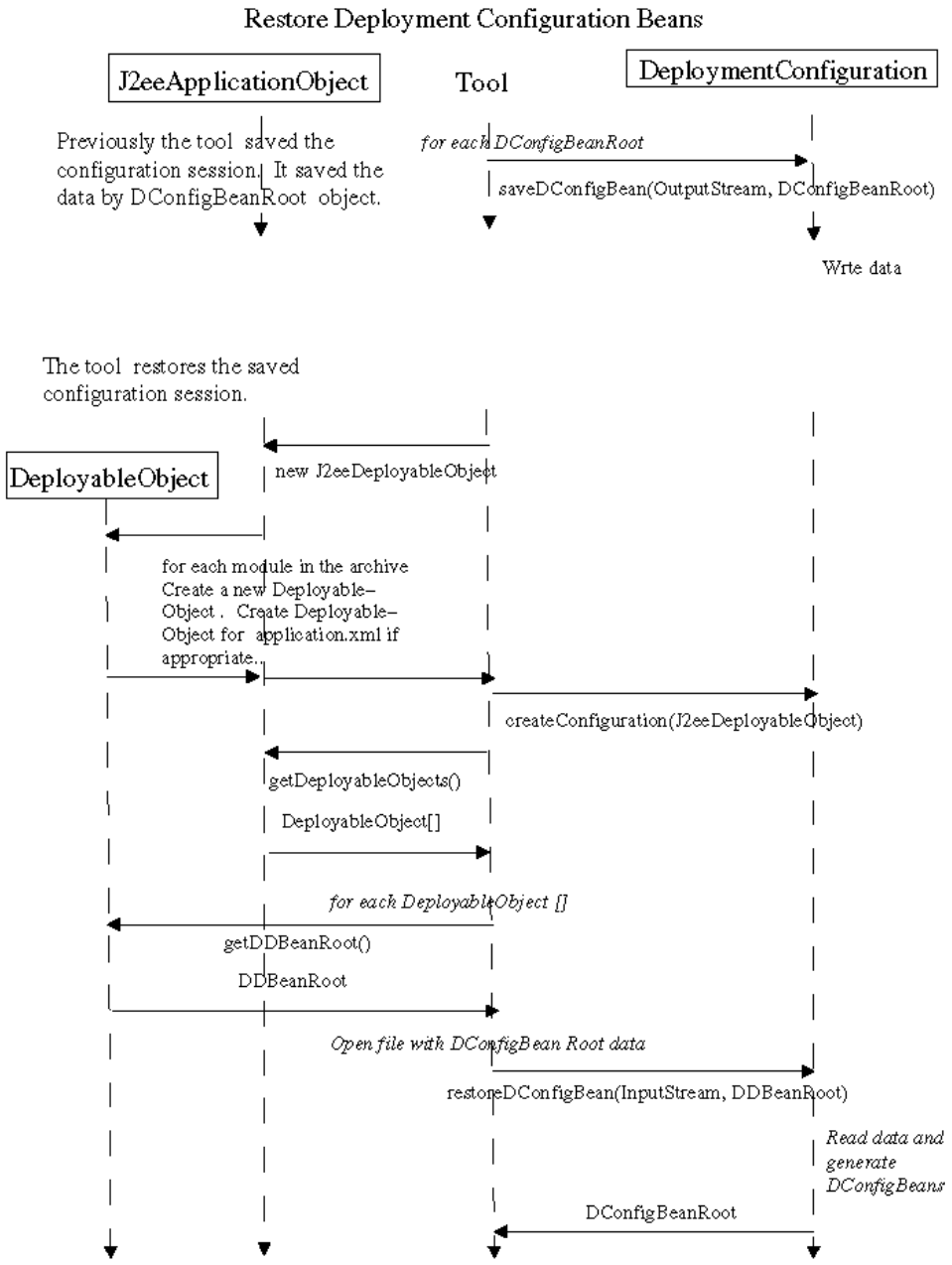


Figure 5.2 Deployment Configuration

5.9.1 Restore Configuration Beans



6 Packaging

A server plugin is packaged into one or more JAR files with a .jar extension. The JAR file or files contain an implementation of the `javax.enterprise.deploy.spi` package and utility classes. The APIs must be implemented in the vendor's namespace.

The entry point class to the server plugin is an implementation of the `DeploymentFactory` interface. There can be one or more `DeploymentFactory` implementation classes in the plugin. Each implementation must be identified by a fully qualified class name in the JAR file's manifest file by the attribute, `J2EE-DeploymentFactory-Implementation-Class`.

The manner in which a vendor's plugin JAR file(s) are made available to the tool is not specified. Some Java EE Product vendors may direct the user to download the plugin from a web site, another may require the user to copy it from their local server installation, and another may provide an initial JAR file whose `DeploymentFactory` implementation contains an automated mechanism that downloads the JAR file(s) on a tool's initial request for a `DeploymentFactory`.

The plugin should not assume that any packages other than the J2SE version required by the plugin's Java EE platform or higher and the `javax.enterprise.deploy` package will be available. The plugin should not provide the Java EE APIs in the JAR files provided to the tool. Plugins should not attempt to load application classes in the tool. The plugin may send the application classes to the server and load them there for reflection, but the plugin should not try to use reflection on application classes in the plugin because doing so is not portable.

The manner in which a vendor's plugin JAR file(s) are made available to the tool is not specified. Some Java EE Product vendors may direct the user to download the plugin from a web site, others may require the user to copy it from

their local server installation, and another may provide an initial JAR file whose `DeploymentFactory` implementation contains an automated mechanism that downloads the JAR file(s) on a tool's initial request for a `DeploymentFactory`.

6.1 Accessing a server plugin

The manner in which a tool makes a server plugin accessible in its classpath is not specified. One tool vendor may designate a directory in which all plugin jar files are saved. It could then process all the jar files in the directory. Another may retain a repository of plugin names and directory locations which it processes. A third vendor may require the user to identify the location of the plugin whenever it runs.

The tool vendor is required to provide the J2SE version required by the plugin or higher and the `javax.enterprise.deploy` package. See `getDConfigBeanVersion` in Section 4.2 for information on how to get the plugin version.

The entry point to a server plugin is the implementation of the `DeploymentFactory` interface. A server vendor must provide at least one implementation of the `DeploymentFactory` interface. The fully qualified name of each `DeploymentFactory` implementation in a JAR file must be identified in the `J2EE-DeploymentFactory-Implementation-Class` attribute of the JAR file's manifest file.

7 Deployment Target

A deployment target (target for short) represents an association between a server or group of servers and a location to deposit a Java EE module that has been properly prepared to run on the server or servers. A target can represent a specific application server installation on a specific host or it can represent a cluster of servers over many hosts. The storage area may be a directory or database or some other storage location. A `Target` represents an atomic element. For example a `Target` can represent a cluster of servers as a single deployable target. The tool and Deployer need not know the server configuration that a `Target` represents. It is left to the product provider to define the type of association that is appropriate for its product.

At least one `Target` object must be defined per Java EE product. The product target information must be accessible to the `DeploymentManager`. An application will be distributed to the target or targets specified at deployment time.

7.1 Target Methods

- **getName** returns a string containing the name of the target.
- **getDescription** returns a string containing descriptive information about the target.

7.2 Target Examples

Figure 6.1 shows three hypothetical targets. The figures show the logical relationships of the elements. They are not meant to imply a physical partitioning of elements into separate machines, processes, or address spaces.

Example 1 illustrates a Java EE product that defines three `Target` objects. Each target represents the association of a server with a separate directory archive repository.

Example 2 illustrates a Java EE product that defines one `Target` object. Three servers use the same directory for the target's archive repository. This example demonstrates a target functioning as a stagging area from which multiple servers pull applications to install and run.

Example 3 there are Java EE product vendors that define a unique server to be defined by a server paired with a database. In this example a single server is paired with two separate databases, thus there are two separate servers by this vendor's definition . A unique target has been defined for each server in this example.

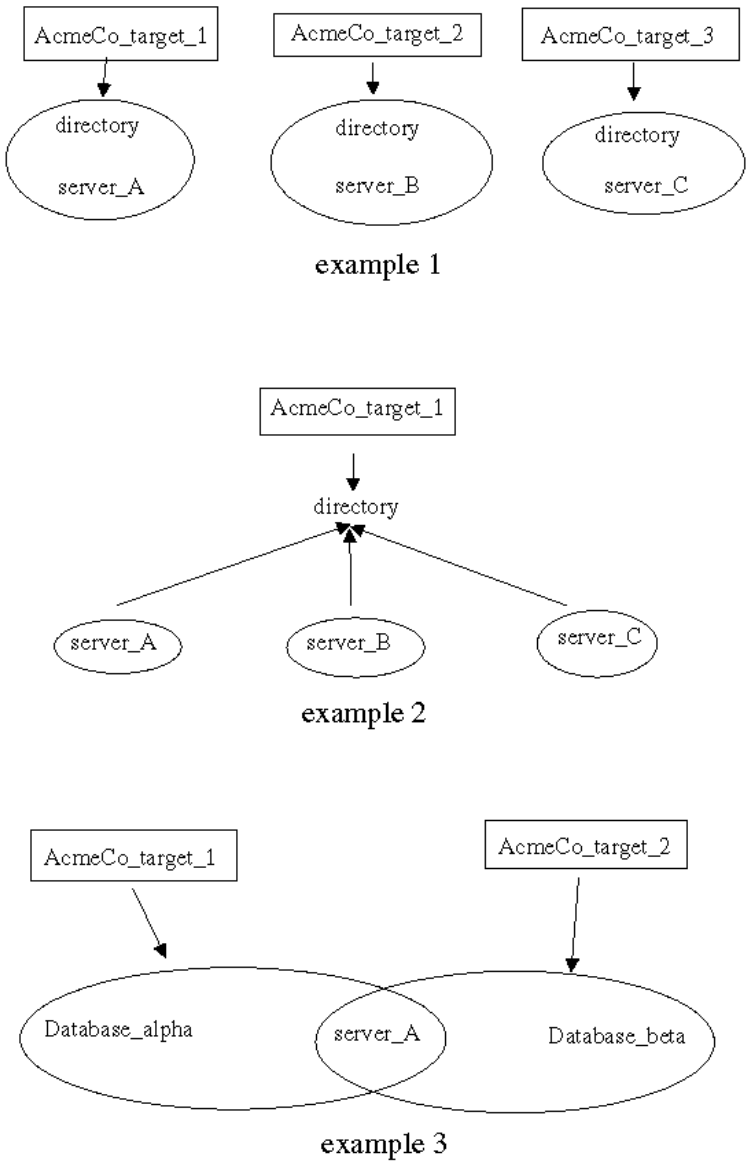


Figure 7.1 Example Targets

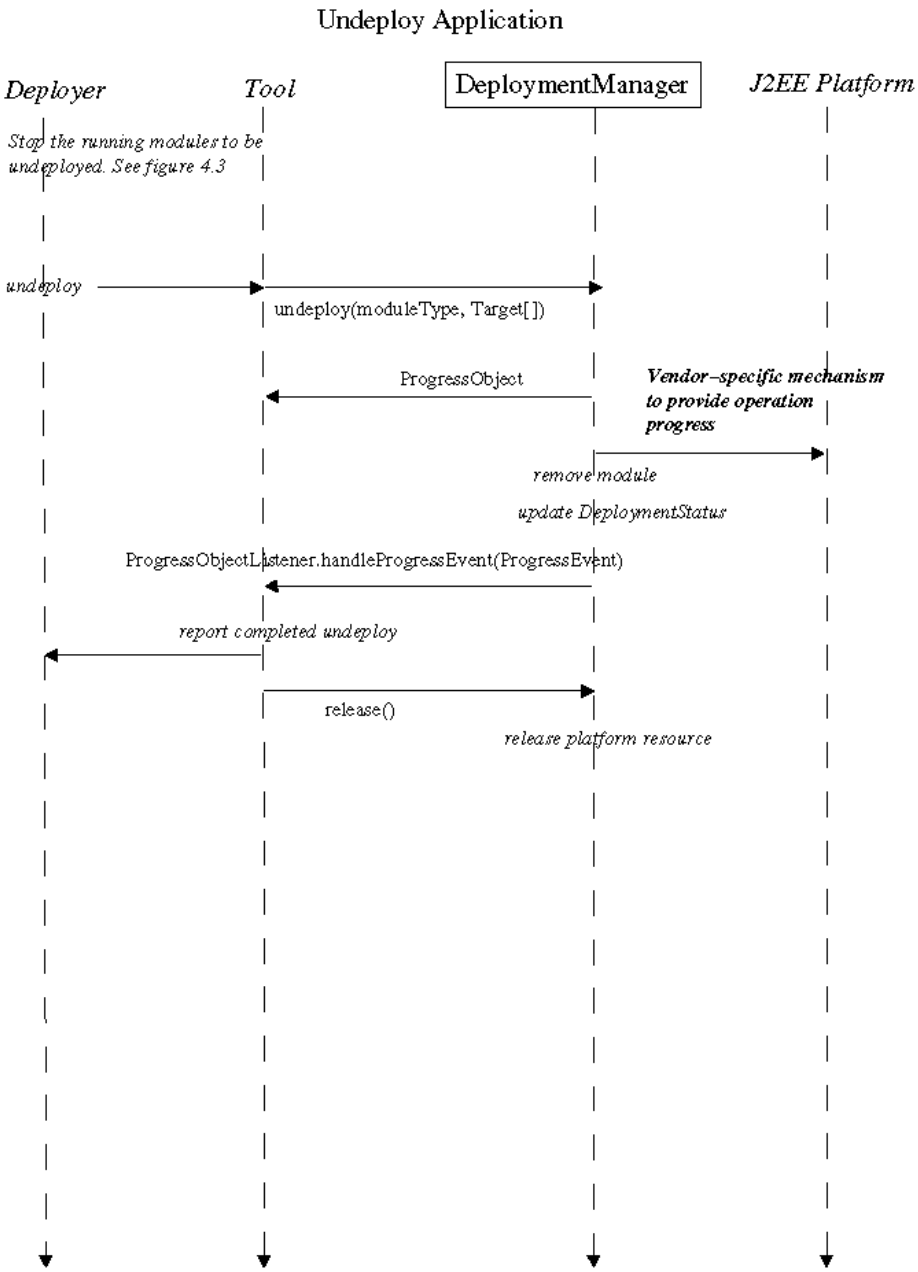


Figure 7.2 Target Examples

7.3 Target and the J2EE Management Specification (JSR 77)

There is no managed object in the management model that translates directly to a Target object. There is a J2EEServer object. This represents a single J2EE server. A Target can represent a single server, but it can also represent a collection of servers. It is left to the Java EE Product Provider to provide a translation of Target object to J2EEServer objects for their product. See section 3.3, "J2EEServer extends J2EEManagedObject" in the Java 2 Enterprise Edition Management Specification.

8 TargetModuleID

The `TargetModuleID` object contains a target-module ID, which is a unique identifier associated with a distributed module. The identifying information consists of the target name on which the module is distributed and a unique identifier assigned to the module. The module identifier must be unique within the Java EE product. The identifier remains the same for the life of the module on the product. The target-module ID of each deployed module must be accessible to the `DeploymentManager`.

The `TargetModuleID` also maintains a reference to its parent and its children. If the parent reference is null, the `TargetModuleID` is the root of the deployed application. A `TargetModuleID` for a stand-alone module will have no parent and no children references.

The `TargetModuleID` is the mechanism by which the Deployer identifies to the Java EE product through the `DeploymentManager` the deployed application or module on which to perform a deployment operation, such as starting a module or undeploying a module.

8.1 TargetModuleID Methods

- **getModuleID** returns a string containing the module name for the deployed module.
- **getTarget** returns the `Target` object for the module.
- **toString** returns a string containing the unique identifier, consisting of the target name and module name, that represents the deployed module.

- **getParentTargetModuleID** returns a `TargetModuleID` object that references the parent of this object. A *null* value means that this is the root object of the deployed application.

- **getChildTargetModuleID** returns a list of all the children of this object.

- **getWebURL** returns the URL of a web module if this ID represents a web module. A *null* value means this ID does not represent a web module.

8.2 TargetModuleID and the J2EE Management Specification (JSR 77)

In the management model the class J2EEObjectName is used to identify a managed object. It is a value object that uniquely identifies a managed object within a management domain. The object name consists of two parts, a domain name and a set of key properties. The key property list enables the assignment of unique names to managed objects of a given domain. It is recommended that the moduleID be used as one of the key properties of the managed object name. See section 7.3, "J2EEObjectName Class" in the Java 2 Enterprise Edition Management Specification.

9 ProgressObject

A `ProgressObject` object tracks and reports the progress of potentially long-lived deployment operations, such as those represented by the distribute, start, stop, and undeploy methods. It also provides an means to retrieve, configure and run an application client. The ProgressObject class has been defined such that a tool can either poll it for status or provide a callback.

The Java EE Product Provider may provide a cancel method or stop method for the running operation. These are *optional* operations in the API. A tool can check for support of the cancel operation by calling the isCancelSupported method, and support of the stop operation by calling isStopSupported. An unsupported cancel or stop operation must throw an `UnsupportedOperationException`.

A cancel request on an in-process operation stops all further processing of the operation and returns the environment to its original state before the operation was executed. An operation that has run to completion cannot be cancelled.

A stop request on an in-process operation allows the operation on the current `TargetModuleID` to run to completion but does not process any of the remaining unprocessed `TargetModuleID` objects. The processed `TargetModuleID` objects must be returned by the method `getResultTargetModuleIDs`.

A ClientConfiguration object is returned by the ProgressObject for each application client distributed to the Java EE product. The ClientConfiguration object is a JavaBean that installs, configures and executes an application client. A ClientExecuteException is thrown if the configuration is incomplete.

9.1 ProgressObject Methods

- **getDeploymentStatus** returns the `DeploymentStatus` object that contains the current status details.
- **getResultTargetModuleIDs** returns a list of `TargetModuleIDs` that completed the associated `DeploymentManager` operation successfully.
- **getClientConfiguration** returns a ClientConfiguration object that installs, configures, and executes an application client.
- **isCancelSupported** indicates whether this product provider has implemented a cancel operation for the associated `DeploymentManager` operation.
- **cancel** stops all further processing of the operation and returns the environment to its original state before the operation was executed. This is an *optional* method for vendor implementation.
- **isStopSupported** indicates whether this product provider has implemented a stop operation for the associated `DeploymentManager` operation.
- **stop** allows the operation on the current TargetModuleID to run to completion but does not process any of the remaining unprocessed TargetModuleID objects. This is an *optional* method for vendor implementation.

9.2 DeploymentStatus Interface

The `DeploymentStatus` object contains information about the progress status of deployment actions.

9.2.1 Deployment Command Enumeration Objects

- **CommandType.DISTRIBUTE** indicates that the object represents status information for a distribute command.
- **CommandType.START** indicates that the object represents status information for a start command.
- **CommandType.STOP** indicates that the object represents status information for a stop command.
- **CommandType.UNDEPLOY** indicates that the object represents status information for an undeploy command.
- **CommandType.REDEPLOY** indicates that the object represents status information for a redeploy operation.

9.2.2 Deployment Status Enumeration Objects

- **StateType.COMPLETED** indicates that the deployment operation has completed normally.
- **StateType.FAILED** indicates the deployment operation has failed.
- **StateType.RUNNING** indicates that the deployment operation is running normally.
- **StateType.RELEASED** indicates that the `DeploymentManager` started running in a *disconnected* mode while this ProgressObject was still active.

9.2.3 Progress Action Enumeration Objects

- **ActionType.CANCEL** indicates that a cancel operation is being performed on the original deployment operation.
- **ActionType.STOP** indicates that a stop operation is being performed on the original deployment operation.
- **ActionType.EXECUTE** indicates that the initial deployment operation is being performed.

9.2.4 Deployment Status Message

Additional information about the object's deployment status can be provided in a text string.

9.2.5 DeploymentStatus Methods

- **getState** returns the current status value.
- **getCommand** returns the DeploymentManager's command value.
- **getAction** returns the current action value.
- **getMessage** returns information text provided about the status.
- **isCompleted** returns *true* if the command has completed successfully.
- **isFailed** returns *true* if the command has failed.
- **isRunning** returns *true* if the command is currently running.

9.3 ClientConfiguration Methods

- **execute** installs, configures and executes the application client .

Note that the Serializable nature of the ClientConfiguration object is limited across VM activations of the application client container.

9.4 Object Interaction Diagrams for a ProgressObject

This section contains object interaction diagrams (OID) that illustrate the interaction of a `ProgressObject` with the tool and `DeploymentManager`.

The diagrams illustrate two hypothetical sessions. Figure 8.1 shows the use of polling to get operation status. Figure 8.2 shows the use of a callback to get operation status.

The order of the interactions listed should be considered illustrative of an implementation rather than prescriptive.

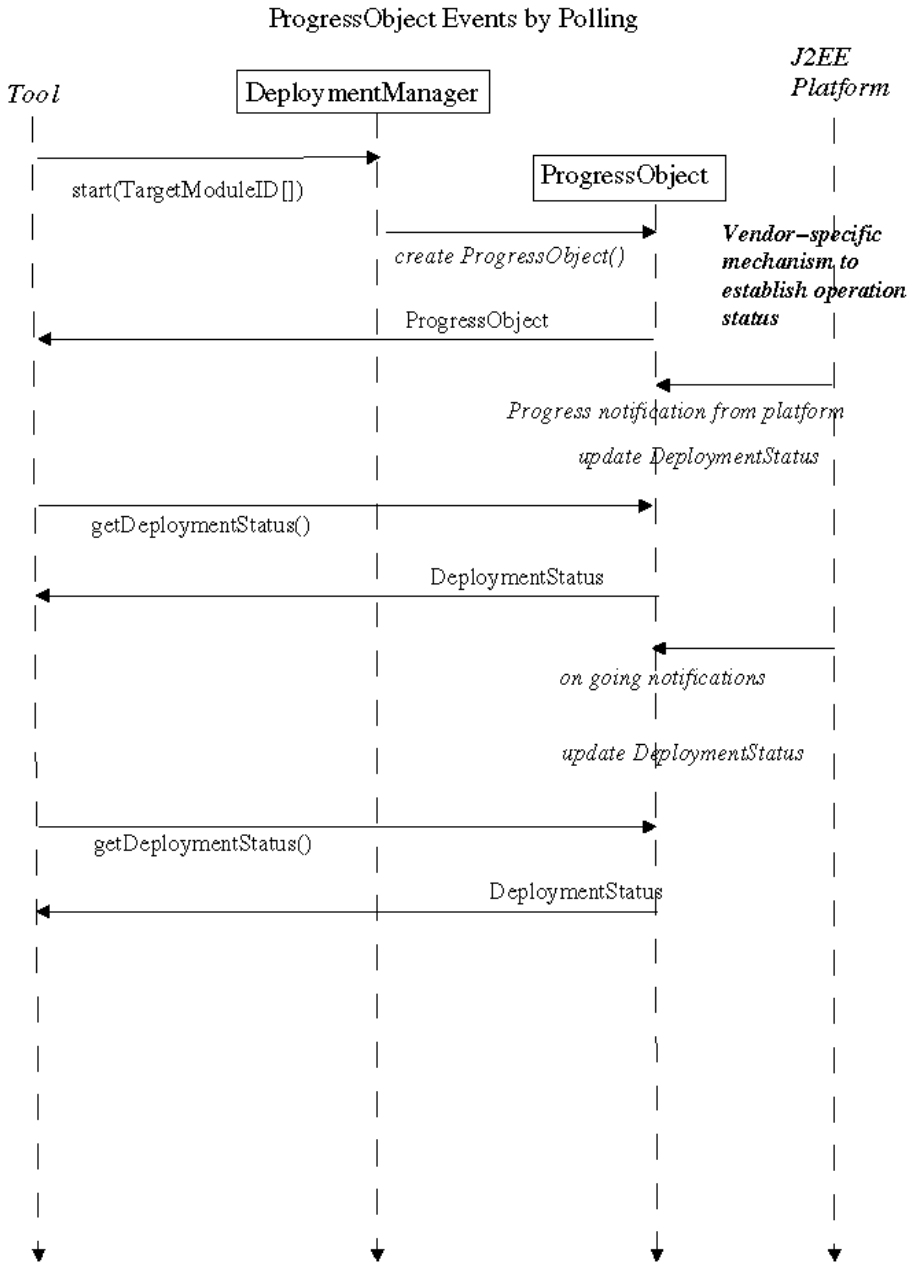


Figure 9.1 ProgressObject Events by Polling

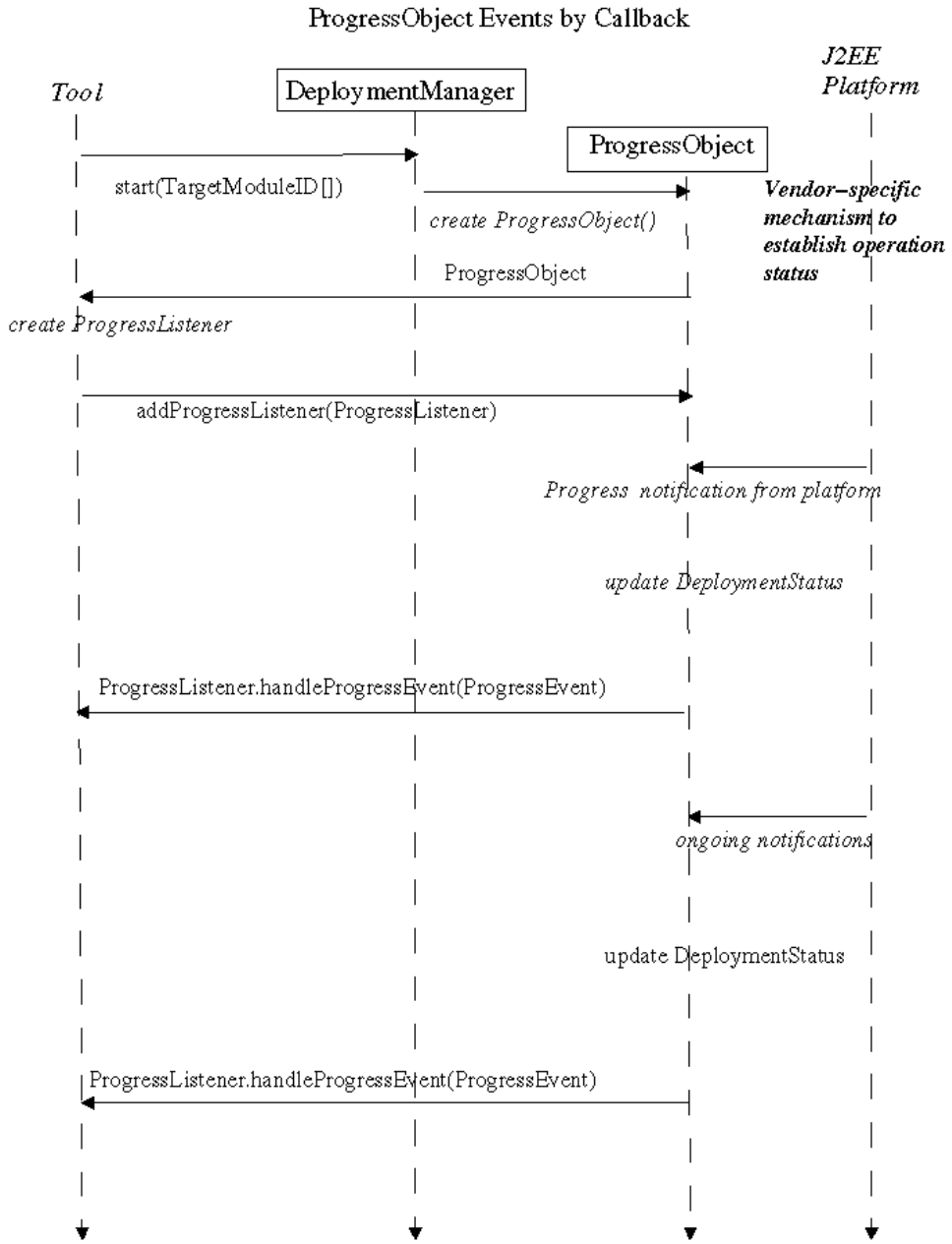


Figure 9.2 ProgressObject Events by Callback

9.5 ProgressObject and the J2EE Management Specification (JSR 77)

The management model provides a facility for event notification by managed objects. This is an optional feature. If a managed object supports event notification and the ProgressObject wishes to receive the events, it must register an event listener object that implements the J2EEManagementEventListener interface. See section 7.7.3, "Event Listener Requirements", and section 5.1, "StateManageable" in the Java 2 Enterprise Edition Management Specification.

10 DeploymentManager Discovery

The `DeploymentManager` is a service that helps the Deployer configure and deploy an application to a Java EE product. Every Java EE product provides a `DeploymentManager`. A deployment tool must acquire a reference to the Java EE product's `DeploymentManager` through a `DeploymentFactory` object.

10.1 DeploymentFactory

A `DeploymentFactory` object is a deployment driver for a Java EE product. It returns a `DeploymentManager` object.

Each Java EE product provider must provide at least one implementation of this class with its product. The class implementing this interface must have a constructor that takes no arguments, and must be stateless (that is two instances of the class must always behave the same). It must be able to return a *connected* or *disconnected* DeploymentManager object.

10.1.1 DeploymentFactory Methods

- **handlesURI** is the method that inspects the Uniform Resource Indicator (URI) provided and returns *true* if it can provide a deployment factory for the URI, and *false* if it can not.
- **getDeploymentManager** returns a *connected* `DeploymentManager` object. A `DeploymentManager` that runs connected to the Java EE product can provide access to Java EE resources.

- **getDisconnectedDeploymentManager** returns a *disconnected* `DeploymentManager` object. A `DeploymentManager` that runs disconnected only provides module deployment configuration support.

10.1.2 DeploymentFactory Discovery

The fully qualified class name of every `DeploymentFactory` implementation provided in a plugin by the Java EE product provider must be listed in the `J2EE-DeploymentFactory-Implementation-Class` attribute of the containing JAR's manifest file.

A tool vendor must introspect each JAR manifest file extract this information in order to a create instance of each implementation class.

An example manifest file identifying two `DeploymentFactory` implementation classes:

```
Manifest-Version: 1.0
Specification-Title: J2EE Specification
Specification-Vendor: Sun Microsystems, Inc.
Created-By: 1.3.0 (Sun Microsystems Inc.)
Implementation-Vendor: Sun Microsystems, Inc.
Specification-Version: 1.3
Implementation-Version: 1.3beta
J2EE-DeploymentFactory-Implementation-Class:
com.sun.enterprise.deployapi.spi.RIDeploymentFactoryAlpha
com.sun.enterprise.deployapi.spi.RIDeploymentFactoryBeta
```

10.2 DeploymentFactoryManager

The `DeploymentFactoryManager` represents a central registry of `DeploymentFactory` connections. The deployment API provides an implementation of the `DeploymentFactoryManager`. A tool creates an instance of the `DeploymentFactoryManager`. The tool creates an instance of a `DeploymentFactory` object and registers it with the `DeploymentFactoryManager`.

When a tool requests a `DeploymentManager` and provides a URI, the `DeploymentFactoryManager` is responsible for finding a `DeploymentFactory` that recognizes the URI and for using it to return the corresponding `DeploymentManager`. If the `DeploymentFactory` understands the URI, it will return a `DeploymentManager` object; otherwise it returns null.

The `DeploymentFactory` class provides a method, `handlesURI`, which the `DeploymentFactoryManager` can use to determine which of its registered drivers it should use for a given URI.

10.2.1 DeploymentFactoryManager Methods

- **registerDeploymentFactory** adds a `DeploymentFactory` object to the set of available factories.
- **getDeploymentManager** acquires a *connected* `DeploymentManager` instance.
- **getDisconnectedDeploymentManager** acquires a *disconnected* `DeploymentManager` instance.
- **getDeploymentFactories** returns a list of currently registered deployment factories.

10.2.2 URI

A Uniform Resource Identifier (URI) can be used to identify a DeploymentManager.

A URL could be used as in the following example to identify a connected DeploymentManager.

For example, if the Acme company provided a server product named AcmeServerPlus, its URL could be:

```
example:  deployer:AcmeServerPlus:myserver:9999
```

The following code example shows how a tool obtains a connected `DeploymentManager` object.

```
DeploymentManager manager;
String url = "deployer:AcmeServerPlus:myserver:9999";
String user = "admin";
String password = "pw";
manager = DeploymentFactoryManager.getDeploymentManager(url, user, password);
if(manager != null) {
        ... // Deploy an application
}
```

10.3 Object Interaction Diagrams for DeploymentManager Discovery

This section contains object interaction diagrams (OID) that illustrate how a `DeploymentManager` can be retrieved.

The diagrams shows two methods of acquiring a `DeploymentManager` object. Figure 9.1 shows acquiring a disconnected `DeploymentManager` and figure 9.2 shows acquiring a connected `DeploymentManager`. Where possible the corresponding method calls and data types are used. A general description of the interaction is provided for those actions that are implementation-specific.

The order of the interactions listed should be considered illustrative of an implementation rather than prescriptive.

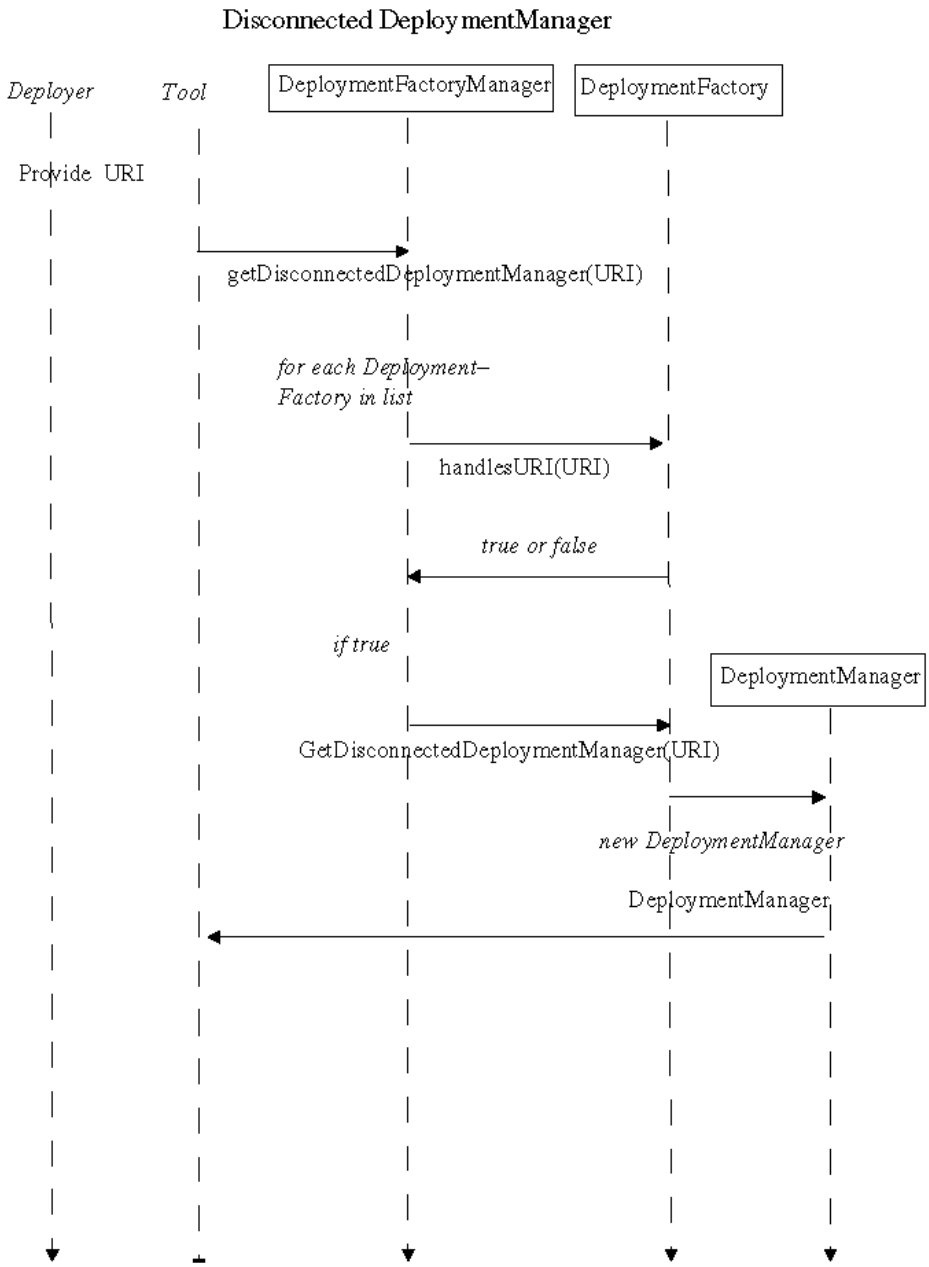


Figure 10.1 Acquiring a Disconnected DeploymentManager

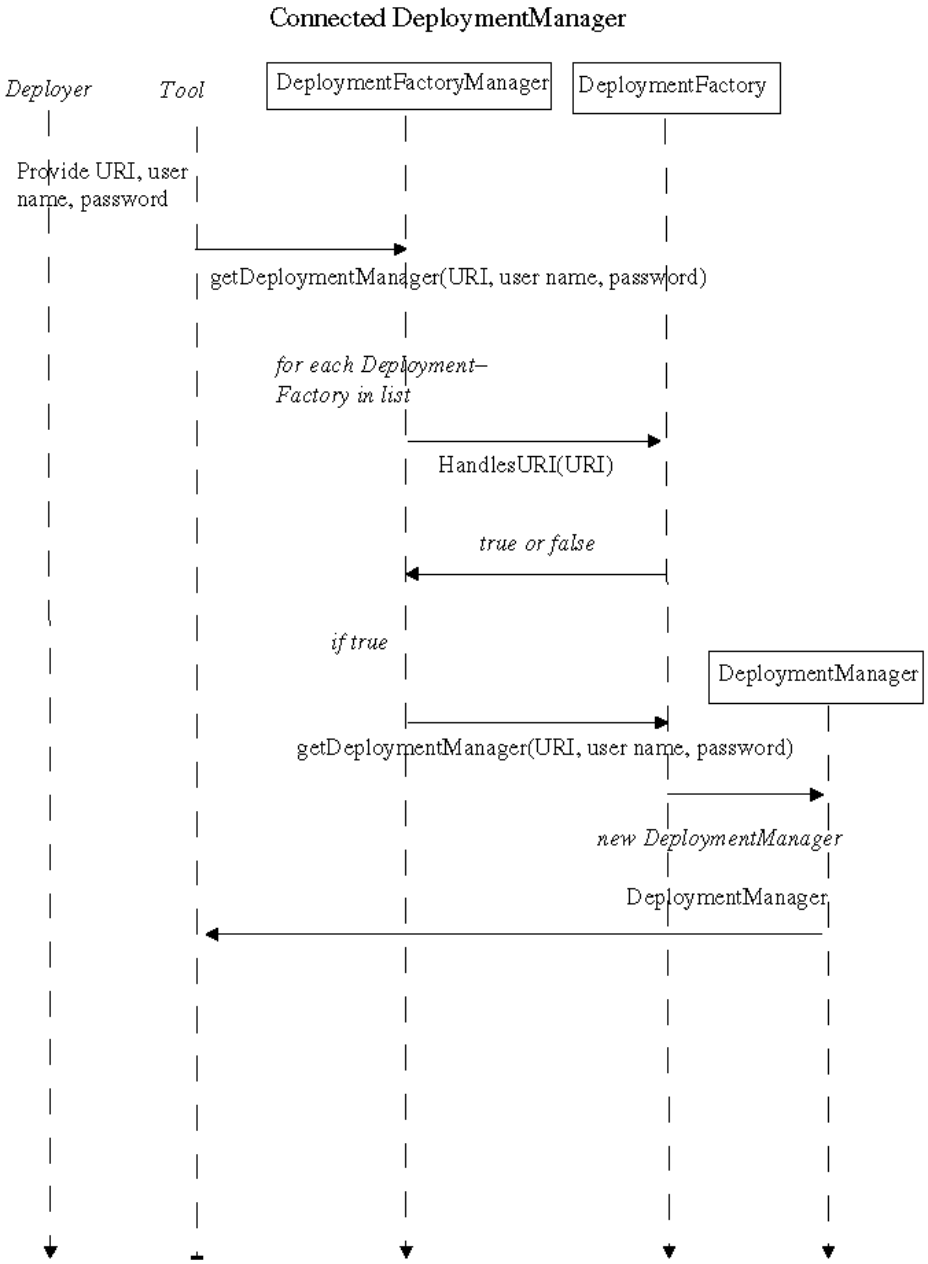


Figure 10.2 Acquiring a Connected DeploymentManager

11 Exceptions

This chapter describes the exceptions used in the Deployment API.

11.1 jaxax.enterprise.deploy.spi.exceptions package

- **BeanNotFoundException** is thrown when the bean is not a child of the parent bean .
- **DeploymentManagerCreationException** is thrown when the `DeploymentFactory` is unable to create a `DeploymentManager` object.
- **DConfigBeanVersionUnsupportedException** is thrown when the DDBeans for a particular Java EE platform verions can not be provided by the tool.
- **InvalidModuleException** is thrown when an invalid module type is detected by the `DeploymentManager`.
- **TargetException** is thrown when an invalid `Target` object is detected by the `DeploymentManager`.
- **java.lang.UnsupportedOperationException** is thrown when an unsupported operation is called.
- **ConfigurationException** is thrown when a `ConfigBean` cannot be created.
- **ClientExecuteException** is thrown when the application client run environment could not be setup properly.
- **java.lang.IllegalStateException** is thrown when a method has been invoked at an illegal or inappropriate time.

11.2 javax.enterprise.deploy.model.exceptions package

- **DDBeanCreateException** is thrown when a DDBeanRoot object can not be created for a specified XML instance document.

A Appendix

A.1 DConfigBean Design Scenarios

This section shows several ways of designing a deployment configuration bean.

This example shows three ways a `DConfigBean` could be designed to extract the `res-ref-name` data from the deployment descriptor fragment . Note these examples assume that an XML parser call retrieves the Xpath data from the deployment descriptor file.

```
<ejb-jar>
 <display-name>Ejb1</display-name>
 <enterprise-beans>
  <session>
   <display-name>com_sun_cts_harness_vehicle_ejb_EJBVehicle</display-name>
   <ejb-name>com_sun_cts_harness_vehicle_ejb_EJBVehicle</ejb-name>
   <resource-ref>
    <res-ref-name>eis/whitebox-tx</res-ref-name>
   </resource-ref>
   <resource-ref>
    <res-ref-name>eis/whitebox-notx</res-ref-name>
   </resource-ref>
   <resource-ref>
    <res-ref-name>eis/whitebox-xa</res-ref-name>
   </resource-ref>
   <resource-ref>
    <res-ref-name>eis/whitebox-tx-param</res-ref-name>
   </resource-ref>
   <resource-ref>
```

```
      <res-ref-name>eis/whitebox-notx-param</res-ref-name>
    </resource-ref>
    <resource-ref>
      <res-ref-name>eis/whitebox-xa-param</res-ref-name>
    </resource-ref>
  </session>
 </enterprise-beans>
</ejb-jar>
```

A.1.1 Scenario one

In the first scenario, a `DConfigBean` returns a child `DConfigBean` for each `res-ref-name` element provided by the tool.

`Session_DConfigBean`, the config bean for a deployment descriptor `session` definition, is implemented by the Java EE Product Provider.

1. `Session_DConfigBean` requests the tool to return all of the `res-ref-name` data by providing the relative Xpath, "`resource-ref/res-ref-name`".
2. The tool calls the XML parser to retrieve the XML elements and creates a `DDBean` for each XML element that is returned.
3. Each `DDBean` is passed to the `Session_DConfigBean`.
4. The `Session_DConfigBean` returns a corresponding `ResRefName_DConfigBean` to the tool.
5. The `ResRefName_DConfigBean` returns a null Xpath, because it does not have any child data to be collected.

```
/* Code provided by the Java EE Product Provider */
public Class Session_DConfigBean implements DConfigBean {
   public String [] getXpaths() {
      String [] str = {"resource-ref/res-ref-name"};
      return str;
   }

   public DConfigBean getDConfigBean(DDBean bean) {
      return new ResRefName_DConfigBean(bean);
   }
```

```
}

public Class ResRefName_DConfigBean implements DConfigBean {
   public String [] getXpaths() {
      return null;
   }
}

/* Code provided by the Tool Provider */
public class Tool {
   DConfigBean parentCfgBean;  // a Session_DConfigBean
   DConfigBean childCfgBean;   // a ResRefName_DConfigBean

   :
   :
   :
   // parentCfgBean was passed to the tool earlier
   :

   String [] xpaths = parentCfgBean.getXpaths();
   for (int i = 0; i < xpaths.length; i++) {
      /* Have the XML parser return the matching Xpath objects */
      NodeList nList = XpathAPI.selectNodeList(xmlDoc, xpaths[i]);

      /* Get a new child config bean for each DDBean presented */
      for (int j = 0; j < nList.getLength(); j++) {
         bean = new DDBean(nList[j]);
         DConfigBean childCfgBean =
         parentCfgBean.getDConfigBean(bean);
      }
   }
}
```

A.1.2 Scenario two

In the second scenario, the `DConfigBean` builds a internal table of `res-ref-name` data that will be used to display to the user. No child `DConfigBeans` are returned.

The `Session_DConfigBean` in this example returns a null value for method `getXpaths`, therefore method `getChildBean` will never be called by the tool.

1. `Session_DConfigBean` retrieves the data it requires by calling its `DDBean` object's `getChildBean` method and passing the Xpath to it.
2. The `DDBean` returns all the matching Xpath elements found. With this data, the `DConfigBean` can build its table.

```
/* Code provided by the Java EE Product Provider */
public Class Session_DConfigBean implements DConfigBean {
   DDBean ddbean;
   DDBean [] childList;

   public Session_DConfigBean(DDBean bean) {
      ddbean = bean;
      childList = ddbean.getChildBean("resource-ref/res-ref-name");
   }

   public String [] getXpaths() {
      return null;
   }

   /** code to create the table **/
      :
      :
}
```

```
/* Code provided by the Java EE Product Provider */
public Class Simple_DDBean implements DDBean {

   public DDBean [] getChildBean(String xpath) {
      /* Have the XML parser return the matching Xpath objects */
      NodeList nList = XpathAPI.selectNodeList(xmlDoc, xpath);

      /* Create a new DDBean for each returned XML element */
      int cnt = nList.getLength();
      DDBean [] childList = new DDBean[cnt];
```

```
        for (int i = 0; i < cnt; i++)
            childList[i] = new DDBean(nList[i]);

        return childList;
    }
}
```

A.1.3 Scenario Three

In the third scenario, the `DConfigBean` builds a internal table of `res-ref-name` data as in the second scenario, but instead of retrieving a list of `DDBeans`, it retrieves the XML data as a list of strings.

In this example the `Session_DConfigBean` object's corresponding `DDBean` `getXpath` method, rather than the `getChildBean` method, is called.

```
/* Code provided by the Java EE Product Provider */
public Class Session_DConfigBean {
    DDBean ddbean;
    String [] childStrList;

    public Session_DConfigBean(DDBean bean) {
        ddbean = bean;
        childStrList = ddbean.getText("resource-ref/res-ref-name");
    }

    public String [] getXpaths() {
        return null;
    }

    /** code to create the table **/
        :
        :
}

/* Code provided by the Java EE Product Provider */
public Class Simple_DDBean {

    public String [] getText(String xpath)
```

```
    {
        /* Have the XML parser return the matching Xpath objects */
        NodeList nList = XpathAPI.selectNodeList(xmlDoc, xpath);

        /* Get the text from the parser node */
        int cnt = nList.getLength();
        String [] childList = new String[cnt];
        for (int i = 0; i < cnt; i++)
            childList[i] = getTextFromNode(nList[i]);

        return childList;
    }

    private String getTextFromNode(Node node) {
        /* extract the string data from the node */
    }
}
```

A.2 EJB Container-managed Persistence

The Java EE platform requires support of the set of Schemas/DTDs for the current Java EE version and for previous J2EE versions. This means that container-managed persistence, CMP 1.1 and CMP 2.0 as defined in the Enterprise JavaBeans ™ (EJB) Specification 1.1 and 2.0, must be supported. This example uses the deployment descriptor version number provided by the `DeployableObject` and an XPath query to demonstrate one possible way to determine the CMP version.

The EJB Specification 2.0 requires backward compatibility for EJB 1.1 entity beans with container-managed persistence. The EJB 2.0 deployment descriptor DTD provides a new element, `cmp-version`, to identify which CMP version to use. This element does not exist in the EJB 1.1 DTD. In an EJB 2.0 component, if the element `cmp-version` is provided in the deployment descriptor file, its value, 1.x or 2.x, identifies the CMP version to be used. If no `cmp-version` element is used in the EJB 2.0 component deployment descriptor file, the default version, 2.x is used.

In this example the class `Entity_DConfigBean` provides support for collecting the runtime configuration information for a EJB entity bean. The class constructor determines the version of the EJB DTD by calling `getModuleDTDVersion` on the `DeployableObject` of the `DDbean`. The constructor assumes the DTD and CMP version will be the same and sets the `cmpVer` accordingly. The constructor sets the list of xpaths based upon the version number for efficiency.

This example could have used the same xpath list for both version 2.0 and 1.1 but since the EJB 1.1 has no such DTD element, `DDbean` would never be returned. Since an EJB 1.1 DTD never provides a `cmp-version` tag method, `getDConfig-Bean` does not need to check the value of `dtdVer`. It only needs to set the value of the `cmp-version` presented.

```
public class Entity_DConfigBean implements DConfigBean {
   String xpathList [];
   DDBean ddbean;
   String dtdVer;
   String cmpVer;

   /**
    * Constructor
    */
   public Entity_DConfigBean (DDBean bean) {
      ddbean = bean;

      /* Get the bean's DeployableObject and its DTD version number.*/
      DeployableObject dObj = ddbean.getRoot().getDeployableObject()
      dtdVer = dObj.getModuleDTDVersion();

      /* Set the xpath list for this entity bean */
      if (dtdVer.startsWith("1")) {
           cmpVer = "1.x";
           xpathList = {"cmp-field"};
      }
      else {
           cmpVer = "2.x";
           xpathList = {"cmp-version", "cmp-field"};
      }
   }
```

```
/**
 * Process XML data provided by the DDBean
 */
public DConfigBean getDConfigBean(DDBean bean) {
    DConfigBean cBean = null;
    String tmpStr = bean.getXpath();

    if (tmpStr.equals("cmp-version")) {
        /* get the version value */
        cmpVer = bean.getText();

        /* ... do other processing */
    :
    :
    :
```