Software through Pictures®

Millennium Edition

Release 8

Generating Enterprise JavaBeans™ with StP/UML 8.x

StP.README-10145/001

Software through Pictures

Generating Enterprise JavaBeans with StP/UML 8.x
Millennium Edition (NT and Windows 2000)
April 2001

Aonix® *reserves the right to make changes in the specifications and other information contained in this publication without prior notice. In case of doubt, the reader should consult Aonix to determine whether any such changes have been made. The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.*

Copyright © 2001 by Aonix® Corporation. All rights reserved.

This publication is protected by Federal Copyright Law, with all rights reserved. Unless you are a licensed user, no part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, by any means, without prior written permission from Aonix. **Licensed users may make copies of this document as needed solely for their internal use—as long as this copyright notice is also reproduced.**

Trademarks

Aonix and its logo, Software through Pictures, and StP are registered trademarks of Aonix Corporation. ACD, Architecture Component Development, and ObjectAda are trademarks of Aonix. All rights reserved.

Windows NT and Windows 2000 are trademarks and Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and other countries. Adobe, Acrobat, the Acrobat logo, and PostScript are trademarks of Adobe Systems, Inc. Sybase, the Sybase logo, and Sybase products are either trademarks or registered trademarks of Sybase, Inc. DocEXPRESS and DOORS registered trademark of Telelogic. ClearCase is a registered trademark of Rational Software Corporation. Continuus and Continuus products are either trademarks or registered trademarks of Continuus Software Corporation. SNiFF+ and SNiFF products are either trademarks or registered trademarks of Wind River Systems, Inc. Segue is a registered trademark of Segue Software, Inc. WebLogic Server is a registered trademark of BEA Systems, Inc. Java and J2EE are registered trademarks of Sun Microsystems, Inc.

All other product and company names are either trademarks or registered trademarks of their respective companies

Table of contents

PREFACE **4**

INTENDED AUDIENCE 4
REQUIREMENTS 4
SUGGESTED READING 4

INTRODUCTION **5**

HISTORY 5
EJBS AT A GLANCE 5
THE TYPES OF ENTERPRISE BEANS 7

MODELING EJBS WITH STP **8**

HOW EJBS SHOULD BE MODELED 8
Package structure *8*
Classes *8*
Attributes *9*
Operations *9*
Exceptions *9*
Associations *9*
Inheritance *9*
WHAT ADDITIONAL INFORMATION IS REQUIRED 9
Classes *9*
Attributes *10*
WHAT IS GENERATED (AND WHERE) 10
EJB classes *10*
Non-EJB classes *11*
DeploymentDescriptor *11*
Build files *11*
Source file location *11*
COMMON MISTAKES TO BE AVOIDED 12
Classes *12*
Attributes *12*
Operations *12*

GENERATING AND COMPILING CODE **12**

REQUIREMENTS 13
INVOKING THE CODE GENERATOR 13
CREATING AND DEPLOYING JAR FILES 13
The Buildfile *13*
Compiling and Archiving *14*
Verifying *14*
The Deployment Process *14*

APPENDIX A: AN OVERVIEW OF USED STEREOTYPES AND TAGGEDVALUES **15**

APPENDIX B: GLOBAL VARIABLES IN TDL SCRIPTS AND MAKEFILES **16**

GENERIC VARIABLES FOR CODE GENERATION 16
SERVER-SPECIFIC VARIABLES 17
WebLogic Server *17*

APPENDIX C: AN EXAMPLE INSTALLATION AND DEPLOYMENT **18**

DOWNLOADING AND INSTALLING REQUIRED SOFTWARE 18
CODE GENERATION AND COMPILATION 18
PREPARING WEBLOGIC 19
DEPLOYING THE BEANS 20

Preface

This document describes the process of modeling Enterprise JavaBeans with StP/UML. Enterprise JavaBeans, or „EJB„ for short, is a new architectural framework for building distributed applications, much in the fashion of OMG's CORBA architecture.

This document shows how users can model the various types of EJBs with StP/UML and how they should augment their models with additional information needed for code generation.

Furthermore, we'll discuss the outcome of the generation, the generated interfaces, classes, buildfiles, and the DeploymentDescriptor, an XML file with descriptive information about EJBs. We'll have a short look at how the generated EJBs can be verified and deployed into an EJB application server.

This document does **not** describe

- the EJB architecture in epic length (it gives, however, a short overview of the architectural foundations),
- the development of client applications using EJBs,
- the deployment of EJBs into an EJB server,
- the integration with 3rd party products like Java IDEs.

Intended Audience

The audience for this user guide includes both modeling experts and enterprise developers with some experience in UML. A working knowledge of the EJB framework is needed if client applications using EJBs are to be written.

Requirements

In order to take full advantage of generating EJBs with StP/UML, you need the following:

- A Java Development Kit (JDK), version 1.2.2 or higher, downloadable from Sun Microsystems. Code generated by the templates has been successfully compiled with both JDK 1.2.2 and 1.3.0.
- The Java2 Enterprise Edition Software Development Kit (J2EE), version 1.2.1 or higher, downloadable from Sun.
- For testing purposes: an EJB server (currently Sun's EJB server, which is included in the J2EE reference implementation, and BEA's WebLogic Server 5.10 are supported). A working knowledge of the EJB architecture is assumed.

Suggested Reading

Users not comfortable with the EJB architecture should read:

- Vlad Matena, Mark Harpner, *Enterprise JavaBeans™ Specification, v1.1*, Sun Microsystems, 2000
- Richard Monson-Haefel, *Enterprise JavaBeans*, 2nd Ed., O'Reilly & Associates, 2000

Introduction

History

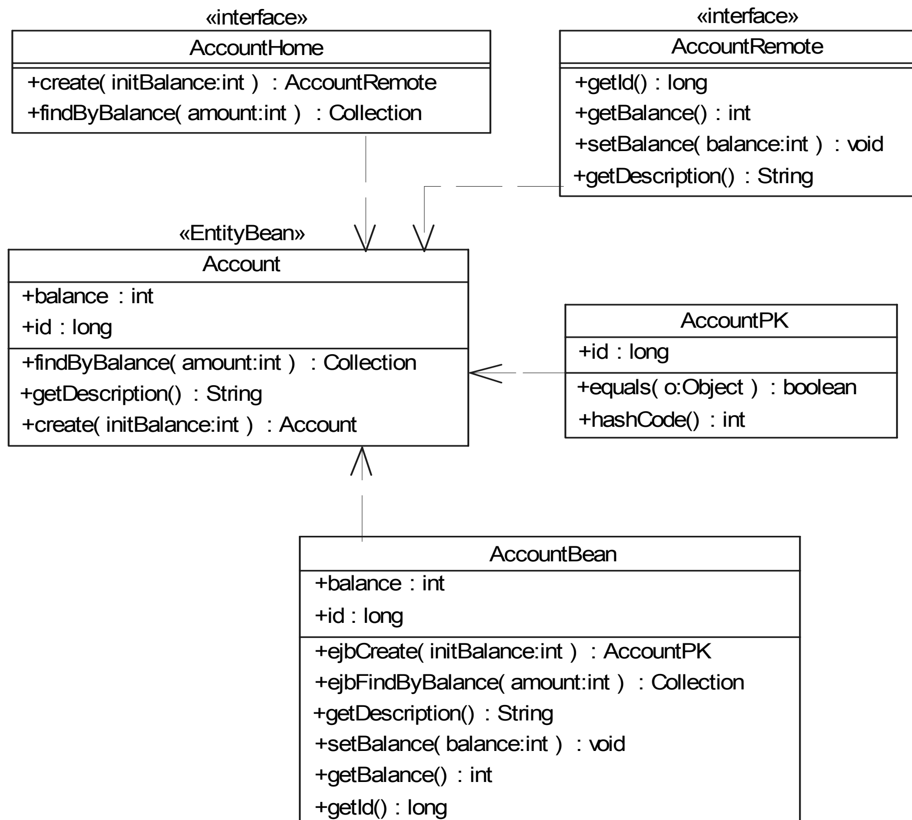
Since the inception of the programming language Java in 1995, its inventor, Sun Microsystems, looked for ways to extend the reach of this language into the enterprise market. In 1997, Sun introduced *Enterprise JavaBeans 1.0*, both a specification and a reference implementation of a framework for development of distributed, transaction-aware, and multi-user secure enterprise applications. In fall 1999 Sun published EJB 1.1, mostly a clarification of the first specification with some enhancements regarding portability. Before we discuss modeling EJBs, we'll give a brief overview of the underlying architecture.

EJBs at a glance

Conceptually, EJB is a marriage between the Java component architecture called *JavaBeans* and the technology for distributed objects known as *Remote Method Invocation* (RMI). Like their siblings on the desktop, EJB consists of operations, attributes, and properties. Properties are attributes for which *getter* and *setter* methods exist. Unlike its cousins, EJB can reside transparently for the client on a remote server and can be distributed.

Physically, an Enterprise Bean is actually a conglomerate of different interfaces and classes; only one of them is usually called a *Bean* class. As a whole, they form an *Enterprise Bean*. There is some form of work-sharing between the different pieces of code. Methods for creating and finding an EJB reside in the so-called *HomeInterface*. The methods that a client can invoke are declared in the *RemoteInterface*. The actual work is done in the Bean class itself. This class implements all the methods in the HomeInterface and RemoteInterface. Look at Fig. 1 to see how a Bean translates to code artifacts.

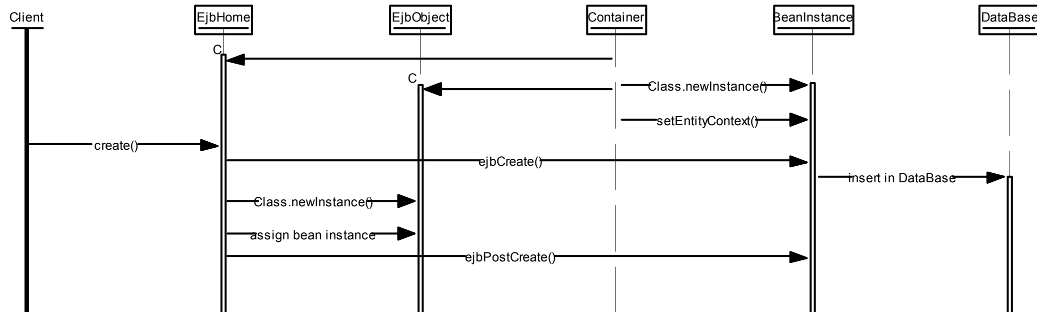
Fig. 1: Translating the model into code



When all the code artifacts of an EJB are created, they are compiled and then bundled into a JAR archive. An additional configuration file, the DeploymentDescriptor, must be created to declaratively code information not known at compile time. This includes security information and transaction attributes for the various methods. Once bundled, the archive is ready for deployment into an EJB server. The server is a container that creates all the necessary objects and manages the services and resources that are needed to support the functionality of an EJB.

If a client wants to invoke a method on a bean, it first needs a remote reference to the bean. Fig. 2 gives us a brief glimpse about what's going on.

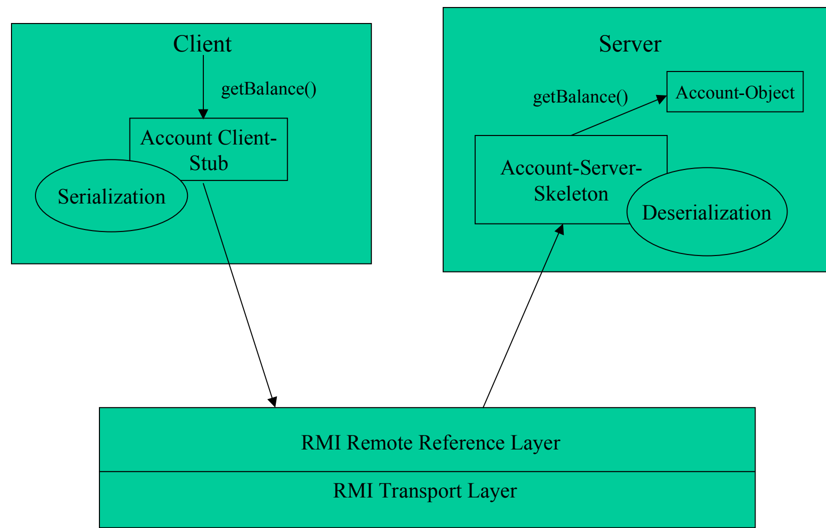
Fig. 2: Retrieving a remote reference



- At deployment time the EJB server creates implementation classes from HomeInterface and RemoteInterface called *HomeObject* and *RemoteObject,* and remote stubs. The classes are server specific and not known to the client.
- The EJB server creates instances of Beans in advance and holds them in an *Object Pool*. At creation time a Session/Entity context is associated with the bean. The context holds configuration information that changes dynamically through the various stages of a bean's life. At this time the bean instance has no identity.
- A client first retrieves a reference to the home object through a JNDI (Java Naming and Directory Service) lookup. This requires that the names for the lookup must be coded in the DeploymentDescriptor; the server binds the HomeObject to that name. The client then retrieves a remote stub class of the RemoteObject that implements this particular HomeInterface.
- The client invokes one of the `create` methods of the HomeInterface.
- The server creates a RemoteObject and associates it with an appropriate bean. Once tied to a RemoteObject, the bean has an identity.
- The server eventually initiates the bean with the parameters of the `create()` method. In case of an entity bean, the server writes the bean's state to a database.
- After successful initialization, the server returns a remote reference through an object stub.

After retrieving a remote reference to a bean, the client invokes one of the methods in the remote interface. Fig.3 describes the process.

Fig. 3: Method invocation over the network



- The client invokes a method on the remote interface stub, which is created by the *rmic* compiler of the EJB server.
- The method and all its parameters are serialized and sent over the network.
- On the server side the message is deserialized by the RemoteObject.
- The method call is then delegated to the bean associated with the remote object.
- If the message has a return value, the process of serializing/deserializing depends on the value (this time the other way around) and the value will be retrieved by the client.

Unusual for the Java architecture and a little bit confusing at first is that the Bean class implements the methods of RemoteInterface and HomeInterface only by convention. It hasn't any „`implements`„ relationship to these interfaces! The main reason is that the Bean would otherwise be required to implement many additional and unnecessary methods.

The types of Enterprise Beans

There are two main types of Enterprise Beans serving different purposes: *EntityBeans and SessionBeans*.

- EntityBeans provide an object oriented view of entities (tables) in a database. EntityBeans should be used for data that are persistent (should be stored in database), that are shared (can be accessed from more than one client), and that model an independent concept.
- SessionBeans are used to model a workflow (several methods working together) and therefore for transactions (units of work that should be indivisible).

The main difference programming-wise between SessionBeans and EntityBeans is persistence: The data of an EntityBean survives even a crash of the EJB server or the database storing it. This is not true for SessionBeans. Also they can have instance variables and therefore a state, this state is never been saved into a database.

Persistence comes in two flavors: *container-managed* and *bean-managed* persistence:

- In Container-Managed Persistence (CMP), the EJB server/container regulates all aspects of storing and retrieving data. The attributes that are to be treated by CMP are declared in the DeploymentDescriptor.
- In Bean-Managed Persistence (BMP), the Bean provider has the sole responsibility for persistence of the attributes. He must implement additional container callbacks, which create and remove data and are used for synchronizing the Bean's state with the underlying database.

SessionBeans have variations too: They can either be *Stateless* or *Stateful*.

- Stateless SessionBeans maintain no conversational state. This means that they can serve the requests of different EJBObjects. In fact, they can be associated with a different client after *each* method invocation. Stateless SessionBeans normally are created in advance and dynamically associated with a client.
- Stateful SessionBeans are, once created, always associated with the same EJBObject. They are therefore created and destroyed when the corresponding EJBObject is created and destroyed.

Modeling EJBs with StP

In StP/UML 8.x you generate code with *ACD*, the **A**rchitecture **C**omponent **D**evelopment facility. ACD consists of an implementation of relevant parts of the UML meta model and a flexible language to navigate through said meta model. The latter is called *TDL*, the **T**ransformation **D**escription **L**anguage.

Modeling EJBs with StP/UML instead of programming them by hand or using a Java IDE with some EJB capabilities, results in several significant advantages:

- It helps the application designer to concentrate on the business logic instead of technical details,
- It avoids a 1:1 mapping between UML classes and generated Java classes,
- It can generate much more code than other code generators,
- Code generation can be customized to the needs of the designer,
- Code generation provides much more consistency than coding from hand,
- There is a real chance that the EJB model can be used for other middleware technologies.

The second point here is the most important one; al other are derived from it. Because there is no 1:1 mapping between model and code, code generation with StP/UML avoids the problem of some „advanced„ tools: they provide an EJB implementation, even an adaptable one, but the resulting model is filled with uninteresting technical clutter. StP/UML has the ability to generate 10 or more classes from one modeled class (not the standard implementation presented here, though). Implementing new features or changing old ones is easy because of the flexible navigation through the resulting model with TDL.

The last point in the „advantage„ section should be taken with a little grain of salt: Although we have no technical details anymore in our models, they are still EJB designs. Changing the infrastructure completely (say: from EJB to OMG CORBA) probably always enforces a restructuring of the model. But there is much less to do than in traditional approaches. In fact, StP/UML models are much more analytical models than design or implementation models!

How EJBs should be modeled

Package structure

It's not absolutely necessary, but you should begin your EJB design with a high level partitioning of your architecture into packages. It‘s probably best to provide an overview diagram of each package with all the classes in it. By using this practice you avoid intermixing classes and packages in diagrams with more detailed class relationships.

Classes

To differentiate EJBs from other class types, which might also generated or referenced, you provide the stereotype `«EntityBean»` and `«SessionBean»` for EntityBeans and SessionBeans respectively. For classes that represent JDBC data sources, there is a stereotype `«DataSource»` (more on that later). Besides that, you model EJB classes as every other UML model, which means you draw the classes in the class diagram editor and provide attributes and operations in the class table editor. For user defined exceptions, the stereotype `«Exception»` should be attached to a class. These exceptions can then be used in the `throws` clause of a method (see below).

Attributes

Attributes are probably the most important part of your model, because they can be used in a very flexible manner throughout EJB code generation. One thing you must keep in mind is that attributes, whose values should be used by the client in some way, must be `public`. In case they are persistent, they can neither be `static` nor `final`. „EJB attributes„ **must** therefore be given the visibility `public`. This may seem inconvenient, but there is no other good way to tell them apart from attributes, which are used only in Bean code.

Operations

Operations that should be part of a publicly available Enterprise Beans method should be modeled as `public` and should not be `static` or `final`. By convention, all methods that go into the HomeInterface must begin with `create` or `find`. They do not appear in the Bean class or RemoteInterface. All other business methods are modeled as described above. A Bean class can have private methods. You should give them a visibility other than `public`.

Exceptions

According to the EJB Spec, a business method can have arbitrary exceptions. You provide them in the section „analyses items„ in the class table. Besides that, some methods (e.g., the lifecycle method) have normally predefined exceptions, which you don't model. More on that later.

Associations

You model associations as you would do it in every StP/UML model, by drawing an association arc between Enterprise Beans or other classes. For classes marked as `«DataSource»`, no Java code is generated, but references to them appear in the DeploymentDescriptor (see below).

Inheritance

Inheritance relationships are currently not generated by the EJB code generation. This probably makes sense only for EntityBeans anyway. Often customers have developed their own framework approach, by which the Enterprise Beans are automatically subclasses of some EJB superclass. This approach lies outside this user guide and could be subject of an additional paper about customizing the EJB templates.

What additional information is required

This section deals with additional information the user must provide in order to control and modify various aspects of code generation.

Classes

SessionBeans are further divided into `Stateless` and `Stateful` by giving them the tagged value `ejb_SessionType=<Type>`, `<Type>` being one of `Stateless` or `Stateful`. If you want (stateful) SessionBeans to implement the interface `SessionSynchronization`, which provides additional container callbacks for data synchronization, you give it the tagged value `ejb_TransactionAware=True`. In SessionBeans you can manage the transaction context for methods either by the EJB container or by yourself. In either case you can provide the tagged value `ejb_TransactionType=Type`, `Type` being either `Container` or `Bean`. `Container` is the default here.

EntityBeans can have container-managed or bean-managed persistence. You differentiate between the two types with the tagged value `ejb_Persistence=<Type>`, `Type` being either `Container` or `Bean`. EntityBeans can be generated as *reentrant*, which means they can call each other in a circular way. For this feature you must add the tagged value `ejb_Reentrant=True` (`False` being the default). Additionally EntityBeans can have *BulkAccessors* (sometimes also called ValueObjects or PropertyObjects). These are classes that contain all or part of the instance variables of the Bean. They are designed to avoid the EJB/RMI overhead, which arises in individual accessor methods. They are used to set or get the bean's state in one method call. You model this by adding the tagged value `ejb_BulkAccessors=True` to the bean's tagged values (`False` is the default).

Attributes

For Attributes you provide additional information solely through the use of tagged values.
Normally for all attributes, accessor methods are generated, which means methods that simply retrieve or set the value of this field. The only condition is that they are not static or final. If you want only a `get()` method you must set the attribute to `readonly` with a tagged value. Fields used as attributes of a PrimaryKey are automatically set to read-only. If you want avoid accessor methods altogether, you must set the tagged value `ejb_access=False`.

In EntityBeans with CMP, all attributes that are public, not static or final or transient, are included automatically in the CMP mechanism. If you don't want CMP for a specific attribute, you should set the tagged value `ejb_CMP=False`.

In EntityBeans one ore more attributes must be designated as the fields of a PrimaryKey class. Alternatively you can designate exactly one attribute as the PrimaryKey class. In the first case, you assign the tagged value `ejb_PKField=True` to the attributes. In the latter case, you give one attribute the tagged value `ejb_PKClass=True`. According to EJB spec 1.1, such a simple PrimaryKey class must either be of type `java.lang.String` or a wrapper class of one of the primitive types such as `java.lang.Integer`. The PK class or fields must be a subset of the container-managed fields. It is therefore an error if such an attribute has the tagged value `ejb_CMP=False`.

What is generated (and where)

EJB classes

For every class marked as `«SessionBean»` or `«EntityBean»`, a RemoteInterface, a HomeInterface, and a Bean class are generated. For every EntityBean that has attributes marked with `ejb_PKField=True`, a PrimaryKey class that contains those attributes is generated. As described above, the creation of an extra PrimaryKey class is suppressed if one attribute has the tagged value `ejb_PKClass=True`.

In the generated HomeInterface and Remote-Interface all methods throw at least the exception `RemoteException` (besides possible others). The RemoteInterface contains all business methods declarations with their modeled exceptions, and all accessor method declarations for (valid) attributes that are not specified as *Bean Private* or with `ejb_access=False`. This includes attributes originating from associations. The HomeInterface contains all `create()` and `find()` methods. The `create()` methods have at least the additional exception `CreateException` and the `find()` methods have the additional exception `FinderException`. All other exceptions come from the model; their type is properly imported in case they come from a different package than the actual class.

In all Bean classes, whether they belong to a SessionBean or an EntityBean, all modeled attributes (besides those required by the EJB specification) are written as differentiating between static and non-static attributes. For all attributes that are neither `static` nor `final` and that are not marked with `ejb_access=False`, at least a `get()`method is provided. For those that are not read-only, an additional `set()`method is generated. Associations with other Beans are automatically translated to attributes, with the following defaults: they are `public`, not `final` nor `static` nor read-only. Besides that, they behave the same as real attributes. All business methods are written with the same signature as in the RemoteInterface, excluding `RemoteException`. They have at least a default return value that is the default null value of the proper type. All methods contain a user modifiable section, which can be edited by the Bean provider. All method parameters whose type is a modeled class are included in the `import` section of the class. All `create()` methods from the HomeInterface are translated to corresponding `ejbCreate()` and `ejbPostCreate()` methods with the same signature minus `RemoteException`. In all Beans, the corresponding container callbacks are at least implemented with empty bodies.

In stateless SessionBeans, an empty `ejbCreate()`method is automatically generated. In SessionBeans that implement the `SessionSynchronization` interface, the additional container callbacks are implemented with empty method bodies.

In EntityBeans with Bean-Managed Persistence, some of the callbacks and the ejbCreate() methods are implemented much more elaborately; however, in some cases they must be probably adapted by the Bean provider.

In EntityBeans with Bean-Managed Persistence, all find() methods from the HomeInterface are translated to corresponding ejbFind() methods in the Bean class minus the RemoteException. They have a default implementation, which, however, must probably be modified by the Bean provider. The mandatory ejbFindByPrimaryKey() method is also implemented.

In EntityBeans that have the TaggedValue ejb_BulkAccessor set to True, BulkAccessor classes are generated. These classes contain all the instance variables of the Bean. Into the Bean class, set() and get() methods are written, which generate instances of the BulkAccessor or set all instance field from the BulkAccessor instance.

Non-EJB classes

Classes marked with the stereotype «Exception» are transformed into Java exceptions, which means „classes derived from java.lang.Exception„. Two constructors are created for every modeled exception. Exceptions can be used in the throws clause of any method. If an exception is located in a package different from the classes that use it, it is correctly imported from that package.

DeploymentDescriptor

For all top-level packages, which means packages that are not nested inside another, a DeploymentDescriptor is generated. This DeploymentDescriptor contains the static structure information about all SessionBeans and EntityBeans. For all Beans it specifies the RemoteInterface, the HomeInterface, and the BeanClass. Furthermore, for all Beans that have associations with other Beans, a reference section is written to the DeploymentDescriptor. This reference contains a string that is suitable for an JNDI lookup by a client. Likewise, a reference section is generated for all associations to a class with stereotype «DataSource». For all EntityBeans the PrimaryKey class and eventually the PrimaryKey field are specified. For EntityBeans with Container-Managed Persistence it specifies also the attributes that fall under CMP management. In the assembly-descriptor section there are dummy entries for the methods transactional context, which can be later on modified by the deployer.

Depending on the requirements of your EJB server, additional DeploymentDescriptors may be created. E.g., WebLogic Server needs an additional descriptor for clustering and caching declarations and, for EntityBeans with CMP, a second one is needed for declaring mappings from CMP attributes to database columns and defining finder queries, if necessary. Other EJB servers may require similar additional descriptors.

Build files

For all top-level packages there is also a build file, by which the package's classes can be compiled and (together with the DeploymentDescriptor) bundled into an JAR archive. These build files exist for both WinNT and UNIX systems. The build files are controlled by a central makefile that is generated once for the whole model (likewise for both WinNT and UNIX systems).

Source file location

All files are generated below the directory <PROJ_DIR>/<SYSTEM>/src_files/. The Java files are generated below src (relative to the base directory) in a directory structure that reflects their package containment. The makefiles are in the directory make. The DeploymentDescriptors are located below build/<Toplevel-Package>/classes/META-INF. For compiled class files and JAR archives, see the section *Generating and compiling code*.

Common mistakes to be avoided

This section helps the user to avoid unnecessary or contradictory information in the model. In general it is not necessary to provide information if you are only interested in the default value. For a detailed description of the various stereotypes and tagged values including their default values, please refer to Appendix A, An Overview of Used Stereotypes and TaggedValues.

Classes

There is one contradictory situation regarding SessionBeans: If you model a stateless SessionBean, making it transaction-aware with `ejb_TransactionAware=True` is useless. For EntityBeans with BMP it makes no sense to mark an attribute with ejb_CMP: With BMP, no attribute is included in the `cmp-field` section of the DeploymentDescriptor.

Attributes

As noted above it makes no sense to give the tagged values `ejb_PKField` or `ejb_PKClass` to attributes that are `private`. Such information is simply ignored. Marking an attribute with either `ejb_PKField` or `ejb_PKClass` and at the time with `ejb_CMP=False` is a mistake. Do not assign the tagged value `readonly` to an attribute if you have tagged it already with `ejb_access=False`; this would be redundant.

Operations

EJB Spec 1.1 does not support references to other beans with regard to persistence (concerning JNDI lookups, it does). This means that a reference to other beans cannot be a container-managed field. This has consequences for finder methods. Some servers do not support finder methods that use lookups for references to other beans. You must avoid them currently. Use bean-managed persistence instead for that class(es). The upcoming EJB Spec 2.0 will handle EJB references in a portable manner.

Generating and compiling code

The previous section showed us how things should be modeled and what is generated. In the following section we'll discuss generation and deployment of the class files and EJB JAR files and their deployment into an EJB server. Currently the Sun reference server and WebLogic Server 5.10 from BEA Systems are supported.

Creating „generic" code (J2EE EJB in the Code generation menu) means that all source files and deployment descriptors meet the EJB 1.1 generic specification. StP's implementation of Enterprise JavaBeans gives you all the flexibility of this specification. The code generated in this mode however only contains information that all EJB 1.1 compliant servers **must** understand.

The templates which generate code for WebLogic on the other hand are a specialization of the generic solution in order to meet the vendor specific demands of that application server. This implementation can also be seen as an example how to adapt the generic J2EE templates to generate code for a particular application server. The JAR files generated by the J2EE templates can be deployed into other servers but their deployment utilities will eventually generate missing information.

The EJB creation and deployment process requires normally 5 steps, most of them are handled by StP/ACD code generation:

1. Generating the source code and deployment descriptor(s) - all code generated by templates
2. Compilation of the generated Java source files into Java classes. - build scripts generated by templates
3. Creation of an JAR - build scripts generated by templates
4. Creation of server specific deployment descriptors - build script for EJB compiler generated
5. Deployment of the JAR file into a server

The first step is always performed by StP if you invoke EJB code generation. As we'll see in the next section you can execute the following steps as well with StP/ACD. The fourth step may be necessary for most EJB servers with specific needs. This is currently handled only for WebLogic Server. For all other servers, you must use their proprietary deployment tools.

Requirements

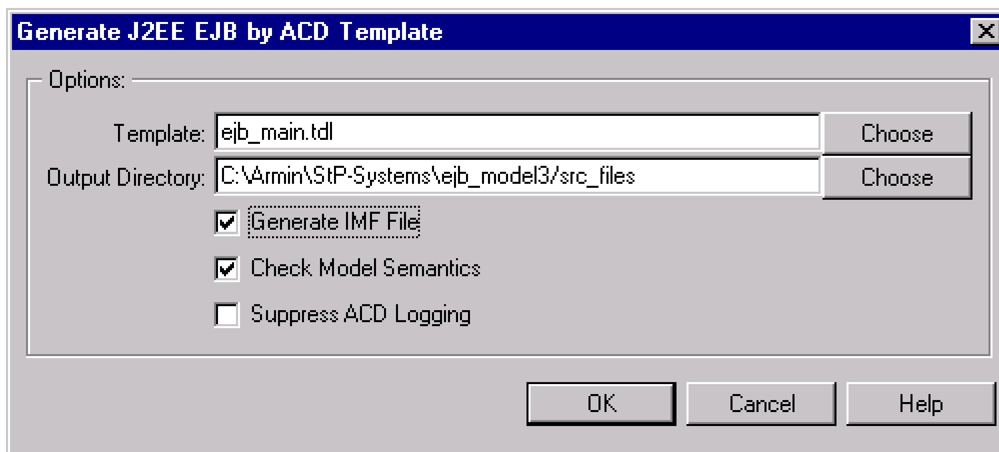
As described in Section 1 (Introduction), for compilation and deployment you need certain prerequisites such as a Java Development Kit (JDK) or a compatible Java Development Environment like JBuilder with a command-line Java compiler and an J2EE environment for the necessary EJB class libraries. Of course, if you actually want to deploy the generated JAR files, you need an EJB server. In the Sun J2EE Kit, there is already a server included. However, Sun's reference implementation has limited functionality.

Invoking the code generator

The first step in the EJB creation process is always generating the source code. After you are satisfied with your model, invoke the EJB code generation within the StP/UML Desktop. At this time we support both generic EJB code which fits to Sun's J2EE reference implementation as well as code for BEA's WebLogic server, choose the appropriate menu command in **Code->EJB**. Fig. 4 shows us the dialog for both templates.

The „Options„ group in the top half of the dialog presents the standard controls of ACD code generation. You can choose a different EJB main template, if present. Additionally you can choose the directory where the source files should go. The checkbox „Generate IMF file„ is necessary if you compile for the first time or if your model has changed.

Fig. 4: The EJB code generation dialog



Creating and Deploying JAR files

The Buildfile

As noted above, for every system, one makefile for NT/W2K systems and one makefile for UNIX systems is created. Actually these files are simple shell scripts, not makefiles in the usual sense. Their primary task is to set necessary environment variables. The actual work is done by buildfiles, which are created for every top-level package. The makefiles simply call the appropriate buildfiles. If you want to adapt the make process to your needs, you should edit the makefiles or (better still) directly edit the TDL scripts that generate them. The global variables that influence code generation are located in `ejb_globals.tdl` and in the globals files of the respective EJB server, e.g., for the WebLogic server in `wls_globals.tdl`. For an overview of these variables, please refer to Appendix B. Note that no build and makefiles are generated if you generate EJB code selectively (i.e. for a class, package or diagram), deployment descriptor files are generated/updated, though.

Compiling and Archiving

Simply invoke `makefile_nt.bat` or `makefile_unix` in the directory `<PROJ_DIR>/<SYSTEM>/src_files/make`. On Unix you probably need to change the makefile's permissions to allow execution. The classes in <src_files> will compiled into to corresponding directories below `<BUILD_DIR>/<Toplevel-Package>/classes`. The DeploymentDescriptor for the top-level packages (see above) is created in `<BUILD_DIR>/<Top-level-Package>/classes/META-INF`.

After the compile stage, the generated classes and the DeploymentDescriptor are bundled into a JAR archive that can be found afterward under `<BUILD_DIR>/<Toplevel-Package>/lib`.

Specific server:
For the WebLogic server, additional DeploymentDescriptors are generated and bundled into the JAR file. If your target is WebLogic, the makefile generates not only these files, it also invokes WebLogic‘s EJB compiler. This compiler generates all internal implementation classes and runs an RMI compiler that creates additional stub and skeleton classes. All these classes are then bundled into an new archive with a suffix appended to the name of the old JAR file (normally `_rfd`, „ready for deployment„). You can directly deploy this JAR files into the WebLogic Server. If you use EntityBeans (you will, when you use the model), you must generate a connection pool that matches the variable `[wls_db_pool]` in `wls_globals.tdl` and all necessary database tables.

Verifying

A JAR file (in particular, one with all beans in it) must conform to the specification. Although a correctly modeled system should result in deployable JAR files, errors are possible at every stage of the development process. In order to avoid fruitless deployment attempts and for debugging purposes, JAR files can be tested for correctness. A good tool for that is the *verifier* tool located under `<J2EE_HOME>/bin`. Its results are normally found under `<TMP_DIR>/Results.txt`.

The Deployment Process

Every EJB server normally has a GUI-based tool for deploying beans, where you add missing information like security roles, method permissions, or environment entries. There is no good way currently to model all the additional information an EJB server may need in StP/UML. We have to wait here for a more complete ACD meta model to capture this information.

One must understand that the final step of EJB generation, deployment into an actual server, is highly tool specific and cannot be discussed here. Some servers provide only a GUI tool for this task; on some servers, you can invoke a command-line tool. Please refer to the documentation of your server manufacturer here. In Appendix C, we describe an example installation and deployment into WebLogic Server. You can derive your own deployment process from the example.

Some servers, like Sun’s reference implementation, may unable to handle bundled JAR files or even the generated DeploymentDescriptors. In this case, you must build these EJB libraries from “scratch,” which means from the compiled class files. All the missing declarative information must then be coded manually by the user. In Appendix C, we provide an example deployment for the model that is part of this implementation.

Appendix A: An Overview of used Stereotypes and TaggedValues

The following table gives an overview of the various Stereotypes and TaggedValues that a model element can be marked with. For every element (only *Class* and *Attribute* need special information) all possible extensions are listed. Of particular interest is the column *Values*. The bold entries denote the default value.

Modelelement	Extension	Type	Values	Avoid
Class	`SessionBean`	Stereotype		
	`EntityBean`	Stereotype		
	`ejb_SessionType`	TaggedValue	**Stateless**/Stateful	
	`ejb_TransactionType`	TaggedValue	**Container**/Bean	
	`ejb_TransactionAware`	TaggedValue	True/**False**	Not with stateless SessionBeans
	`ejb_Persistence`	TaggedValue	**Container**/Bean	
	`ejb_Reentrant`	TaggedValue	True/**False**	
	`ejb_BulkAccessors`	TaggedValue	True/**False**	
Attribute	`Readonly`	TaggedValue	True/**False**	
	`ejb_access`	TaggedValue	True/**False**	
	`ejb_CMP`	TaggedValue	True/**False**	Not with ejb_Persistence= Bean
	`ejb_PKField`	TaggedValue	True/**False**	Not with `ejb_PKClass` or `ejb_CMP=False`
	`ejb_PKClass`	TaggedValue	True/**False**	Not with ejb_PKClass or ejb_CMP=False

Appendix B: Global Variables in TDL scripts and makefiles

Generic Variables for Code Generation

The following standard environment variables should be set in `ejb_globals`:

`[java_home_nt]`	the base directory of the version of JDK you want to use (Windows)
`[java_home_unix]`	the base directory of the version of JDK you want to use (Unix).
`[j2ee_home_nt]`	the base directory of your J2EE version (Windows).
`[j2ee_home_unix]`	the base directory of your J2EE version (Unix).
`[javac_nt]`	the used Java compiler. `<JAVA_HOME>/bin/javac` is the (slow) default. You can use more speedier compiler such as *Jikes* (from IBM DeveloperWorks).
`[javac_unix]`	`same on Unix`
`[jar_nt]`	the jar tool for archiving. Normally `<JAVA_HOME>/bin/jar` is used (Windows).
`[jar_nt]`	the jar tool for archiving. Normally `<JAVA_HOME>/bin/jar` is used (Unix).

`[display_name_suffix]`	NameSuffix for the EJB's display name
`[pkg_display_name_sf]`	NameSuffix for display name of a particular JAR
`[home_name_suffix]`	name suffix for the Home Interface
`[remote_name_suffix]`	name suffix for the Remote Interface
`[bean_name_suffix]`	name suffix for the Bean class itself
`[ejbjar_name_suffix]`	Bean-Name in DeploymentDescriptor
`[pk_name_suffix]`	name suffix for the PrimaryKey class
`[vo_name_suffix]`	name suffix for the ValueObject class (e.g. „Data„ or „VO„)
`[jndi_name_suffix]`	name suffix for a JNDI lookup
`[entity_bean_stereo]`	Stereotype, an EntityBean should have
`[session_bean_stereo]`	Stereotype, a SessionBean should have
`[datasource_stereo]`	Stereotype, a DataSource should have
`[def_coll_class]`	Default class for collections (could be List etc.)

The following standard environment variables should be set in `makefile_nt.bat` or `makefile.unix`: So You have a chance to change compilation options after code generation (e.g., to moved the source directory to a different place).

`BASE_DIR`	Base directory of the generated files. The default is the value you provided at generation time in the property sheet of the code generation menu (normally `<PROJ_DIR>/<SYSTEM>/src_files`). You can set it if have moved the directory after generation.
`SRC_DIR`	Top-level directory of the generated Java source files.
`BUILD_DIR`	Top-level directory for compiling and archiving of classes and deployment descriptors. For every top-level package in the model, one subdirectory is created.
`CLASSPATH`	Specify additional values here if you need to. Do not delete entries however as this will probably prohibit compiling.

Server-Specific Variables

WebLogic Server

For the WebLogic server, the following variables can be customized in `wls_globals.tdl`:

`[wls_home_nt]`	Base directory of your version of WebLogic Server (Windows).
`[wls_home_unix]`	Base directory of your version of WebLogic Server (Unix).
`[ejbc_nt]`	The used EJB compiler. The EJB compiler takes a JAR file compliant with EJB Spec 1.1 and WebLogic Server and generates all the necessary support classes (with the help of a Java compiler). The name of this program has changed in the past, so we're providing an extra variable.
`[ejbc_unix]`	`Same on Unix.`
`[wls_type_version]`	Version of your WebLogic Server. Only version 5.1.0 is currently supported.
`[wls_type_identifier]`	Version of WebLogic Persistence Service for EntityBeans with container-managed persistence.
`[wls_jar_suffix]`	Suffix for completely compiled JAR files for WebLogic Server. “rfd” stands for “Ready for deployment.„
`[wls_db_pool]`	Name of the used WebLogic Connection Pool. For container-managed persistence to work, the user must declare a connection pool in `weblogic.properties` with exactly the this name.

Appendix C: An Example Installation and Deployment

In this section we present an example utilization of our current EJB implementation. It shows what is currently possible with StP/UML and ACD, and – equally important – what is not. To make things more interesting, we don't use Sun's J2EE as an EJB server; instead we use WebLogic Server, both more complex and feature-rich.

Downloading and installing required software

In the following, we assume that you have downloaded all needed pieces of software described in the *requirements* section. Suppose you have these software installed in the following places:

Software	Installed under
JDK 1.3	D:\JDK13
J2EE1.2.1	D:\J2EESDK121
StP/UML 8.x	D:\StP\StP_V8.x
ejb_model	D:\StP\StP_V8.x\examples
BEA WebLogic Server 5.10	D:\bea\wls510

Make sure that all software you installed is working flawlessly. You also need to recover the `ejb_model` repository located in the `examples` directory in StP if you haven't done so yet, by using the StP desktop command **Repository->Maintain Systems->Recover System Repository.**

Then, adapt `java_home_nt` and `j2ee_home_nt` in `ejb_globals.tdl` (located in `<StP_DIR>\templates\uml\qrl\code_gen\ejb`) to the location of your JDK and J2EE environment as well as the paths in `wls_globals.tdl`, located in the same directory.

Code generation and compilation

Now execute **Code->EJB->Generate WebLogic EJB by ACD Template** from the StP/UML desktop. ACD will create Java source files under `<StP_DIR>\examples\ejb_model\src_files\`.

Next, open a command window in the directory `<StP_DIR>\examples\ejb_model\src_files\make\` and run the `command makefile_nt`, which has also been generated by ACD. The compiled classes go under `<StP_DIR>\examples\ejb_model\src_files\build\<ToplevelPackage>\classes`, with `ToplevelPackage` replaced by `credit` and `com`. The JAR files (without server container classes) can be found under `<StP_DIR>\examples\ejb_model\src_files\build\<ToplevelPackage>\lib`. If WebLogic is installed properly (don't forget to apply the latest service packs!), the EJB compiler should have generated the deployable JAR files `credit_rfd.jar` and `com_rfd.jar` under the respective `lib`- directory.

Check any `*.err` files located `in build\<ToplevelPackage>\` for potential errors being generated by the Java or EJB compiler (they should have a size of 0 bytes if everything ran successfully).

Preparing WebLogic

Before we can deploy these two JAR files we must prepare WebLogic for use of EntityBeans, in particular for the `CustomerBean`, which uses container-managed persistence. In fact, if you only deploy the `credit_rfd.jar` file, you can skip the following two steps and invoke the WebLogic deploy tool. Otherwise you need to configure the database and the connection pool.

In our example we use the Java database Cloudscape, which comes with WebLogic. First, create a database and a table for `CustomerBean`. WebLogic comes with the utility *Schema,* a Java command line tool for uploading and execution of SQL scripts. To create the CUSTOMER table in database WLS_POOL, open and command line shell in windows, and use a command or batch file like we provide in
`<StP_DIR>\templates\uml\qrl\code_gen\ejb\misc\create_table.cmd`
Adjust the paths in that file as needed before you run the command.

The SQL/DDL file used as input to create the table (located in the same directory) looks like this:

```
DROP TABLE CUSTOMER;
CREATE TABLE CUSTOMER
(
	ID		INTEGER		PRIMARY KEY,
	FIRSTNAME	VARCHAR(20) NOT NULL,
	LASTNAME	VARCHAR(30) NOT NULL
);
```

Obviously, this only provides a very basic 1:1 mapping between CustomerBean and the Customer table. Although persistence mapping could be generated by ACD as well, we chose not to implement it in this release since there is no “common practice” yet and EJB 2.0 is expected to provide additional guidance in that area, too.

Make sure that the above command runs without producing any error messages (other than a possible error due to the fact that the table CUSTOMER cannot be dropped when you run the command for the first time).

Next, open your weblogic.properties file and add a new connection pool. You can search for the demoPool section and copy the values to a new connection pool statement. The name of the connection pool must match the value of the variable [wls_db_pool] in wls_globals.tdl. After editing, your new statement should look like:

```
weblogic.jdbc.connectionPool.WLS_POOL=\
       url=jdbc:cloudscape:WLS_POOL,\
       driver=COM.cloudscape.core.JDBCDriver,\
       initialCapacity=1,\
       maxCapacity=2,\
       capacityIncrement=1,\
       props=user=none;password=none;server=none
# Add a TXDataSource for the connection pool:
weblogic.jdbc.TXDataSource.weblogic.WLS_POOL=WLS_POOL
#
# Add an ACL for the connection pool:
weblogic.allow.reserve.weblogic.jdbc.connectionPool.WLS_POOL=everyone
```

A connection pool is a group of pregenerated connections to a specific database. Creating database connections always costs time; creating them in advance and reusing them can accelerate database queries and updates significantly.

Deploying the Beans

Now we are ready to deploy the created JAR files. From the Windows Start Menu, invoke the WebLogic server. You should see the following lines in the message output of the server if your connection pool is set up correctly

```
<JDBC Pool> Creating connection pool WLS_POOL with:
poolName=WLS_POOL maxCapacity=2 props=user=none;server=nonedriver=COM.cloudscape.core.JDBCDriver
aclName=weblogic.jdbc.connectionPool.WLS_POOL capacityIncrement=1 initialCapacity=1
url=jdbc:cloudscape:WLS_POOL
<JDBC Pool> Connection for pool „WLS_POOL„ created.
<JDBC Init> Creating Tx DataSource named weblogic.WLS_POOL for WLS_POOL pool
```

After the server has completed its boot process, start the WebLogic deploy tool *EJB Deployer* from the WebLogic program group in Windows. Change to Deployer Projects. If not done yet, create a new entry for the server, into which the JAR files are to be deployed. Its name is probably *myserver* if you have not defined another server entry in weblogic properties. Load the credit_rfd.jar and com_rfd.jar from the build\com\lib and build\credit\lib-directory. Click on the JAR and JAR-1 entries and invoke **Tools->Deploy To->myserver.** This concludes our deployment example for WebLogic 5.10.