BEA WebLogic Commerce Server Components Developer's Guide

BEA WebLogic Commerce Server 2.0
Document Edition 2.0
April 2000

BEA WebLogic Commerce Server Components Developer’s Guide

Document Edition	Date	Software Version
1.0	January 2000	WebLogic Commerce Server 1.7
1.1	February 2000	WebLogic Commerce Server 1.7.1
2.0	April 2000	WebLogic Commerce Server 2.0

Contents

About This Document

This document explains how to use the BEA WebLogic Commerce Server Components to extend or modify an e-Commerce Web site.

This document covers the following topics:

- Chapter 1, “Overview of WebLogic Commerce Server Components.”
- Chapter 2, “Components Catalog.”
- Chapter 3, “Development Process.”
- Chapter 4, “Deploying Your Application.”
- Chapter 5, “Component Examples.”

What You Need to Know

This document is intended for Enterprise JavaBeans (EJB) and Java developers involved in working with EJB components for an eCommerce site using BEA WebLogic Commerce Server. It assumes a familiarity with the WebLogic Commerce Server platform, WebLogic Application Server, EJB, Java, and related Web technologies as described below. The topics in this document are organized primarily around development goals and the tasks needed to accomplish them.

Generally, the topics in this document speak particularly to the Java developer and requires the basic knowledge with regard to the technology focus of that role:

- *Java developer* extend or modifies the Enterprise Java Bean (EJB) components that make up the Commerce Server engine, if that level of customization is needed.

The Java developer working with the EJB components will also interact with other development team members or may take on other roles as well:

- *HTML author* uses the Java Server Page (JSP) tags provide in the JSP tag library, thereby leveraging the power of personalization without having to know Java.
- *Java Server Page (JSP) developer* creates JSPs using the tags provided or by creating custom tags as needed.
- *Application assembler*, *system analyst*, or *systems integrator* writes rules, writes, schemas, and monitors usage.
- *System administrator* installs, configures, deploys, and monitors the Web application server

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at http://e-docs.bea.com.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the WLCS documentation Home page at http://e-docs.bea.com/wlcs/. A PDF version of this document is also available on your local system if you installed the separate WLCS documentation kit. In the installed WLCS directory, the documentation's default starting location is:

```
\server\public_html\docs\index.htm
```

You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Commerce Server documentation Home page, click the PDF files button and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at http://www.adobe.com/.

Related Information

For more information about the Java 2 Enterprise Edition (J2EE) APIs, see the Sun Microsystems, Inc. Web site at http://java.sun.com/j2ee/.

Contact Us!

Your feedback on the BEA WebLogic Commerce Server documentation is important to us. Send us e-mail at **docsupport@beasys.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Commerce Server documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Commerce Server 2.0 release.

If you have any questions about this version of BEA WebLogic Commerce Server, or if you have problems installing and running BEA WebLogic Commerce Server, contact BEA Customer Support through BEA WebSupport at **www.beasys.com**. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address

- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Example:
	`public interface Item extends ConfigurableEntity`
	`{`
	`public ItemValue getItemByValue() throws`
	`RemoteException;`
	`public void setItemByValue(ItemValue value) throws`
	`RemoteException;`
	`//...`
	`}`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*

Convention	Item
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

`...`	Indicates one of the following in a command line:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

1 Overview of WebLogic Commerce Server Components

This section contains the following topics:

What are Commerce Server components?

At the heart of the BEA WebLogic Commerce Server (WLCS) are the Commerce Server *components*. Commerce Server components are software building blocks for eBusiness that can be selected and snapped together to create a robust eCommerce Web presence. You can use any component as is, or customize or extend it to fit particularly unique aspects of your business scenario. This family of Enterprise Java Beans helps you bring new e-business services to your customers quickly and easily, while allowing you to focus precious resources on your unique competitive requirements.

The WLCS components family is structured as packages. Each package contains a set of components that plays a specific role in assembling an enterprise application. WLCS provides *industry specific component packages* for financial services, telecommunications, and Internet retail. Today WLCS includes component packages required to build **e-business** applications, with an emphasis on Internet e-commerce and customer self-service.

Using novel design patterns, we have modeled, designed, built, and tested our reusable server-side components to work seamlessly in conjunction with your commercial EJB application server.

A Quick Look at a Few Key Components

Following are descriptions of only a few of the key eBusiness components (order, invoice, customer, session, and shoppingAdvisor). WLCS includes more than 80 out-of-the-box, pluggable Java components designed to provide most of the functionality of essential e-business. For a more comprehensive and detailed look at the components, you can refer to Chapter 2, “Components Catalog,” or the complete API in Javadoc.

Customer and Session

The fundamental entities for any business are *customers* and the *products* sold to them. The Customer is an extension of the Axiom.Person. It provides the ability to store contact, profile, and billing information for your customers. The Session components are used to manage the process of allowing customers to access the system as guests and then to register when they are ready to make a purchase. They also bind customers to the orders that they build.

ShoppingAdvisor and Items

The **Item** is the interface to the products that you are selling. It stores basic product identification and description, and provides a mechanism for pricing, including runtime pluggable pricing policies. The pricing mechanism is designed to allow you to take into account a specific customer's profile. This allows the application of special merchandising discounts and incentives. The **ShoppingAdvisor** is the means by which you organize your products and make them searchable. Its additional features include learning about customer preferences over time and recommending products based on the resulting profile.

Order Fulfillment

The **Order** acts both as a shopping cart and the basis for order fulfillment. It is the mechanism by which a customer keeps track of items that they want to purchase. The list is manipulated through business methods so that overloading can enforce the business rules associated with building an order. There is also an order cost calculation method that can be used to take into account discounts across multiple individual orders.

When an order is completed it is bound to a **PackingList** so that shipping cost can be calculated. The next step is the creation of an **Invoice**, so that the order can be billed. Finally, the **Inventory** is updated.

The **TroubleTicket** components provide customer service issue tracking. These components provide you the ability to accept and track issues submitted by your customers.

Features at a Glance

- Customizable Enterprise JavaBeans built from the ground up
- Plug-and-play components that allow you either to use our out-of-the-box solution, or to integrate with your legacy applications
- Easy-to-use component APIs that are fully customizable and extensible using technologies like pluggable methods and dynamic runtime configuration
- Implemented using established design and analysis patterns for ease of use and re-use
- WLCS architecture ensures that applications built on the components model run in a scaleable, high-performance, enterprise-class fashion
- Components work with other EJBs, including third-party and custom-built components
- High-performance features such as pass-by-value (PBV)
- Works with leading EJB Application Servers
- Works with leading databases (Oracle, Sybase, DB2, Cloudscape, etc.)
- Integrate with legacy systems (CICS, IMS, legacy databases)
- You don’t need to be an EJB expert to use and customize our pre-built EJBs!

Specifications

- 100% Pure Java
- Components are Enterprise JavaBeans 1.1
- Support for Java 2
- Support for the advanced Java 2 Collections API

- Support for Enterprise Java APIs including EJB, RMI, JNDI, JTS, and JDBC
- Support for modeling using UML, with roundtrip engineering from Rational Rose
- Component can be invoked from Java Clients, Java Servlets, Java Server Pages (JSP/JHTML), CORBA Clients and Servers,
- Support for ActiveX/COM, and other clients (Visual Basic, PowerBuilder)
- Full support for BEA WebLogic Server and related features, including clustering and JDBC connection pooling. Support for other leading application servers coming soon.

eCommerce brings tremendous opportunity and new challenges.

Application server and EJB technologies present a tremendous opportunity for enterprise information systems. Businesses can gain competitive advantages by rapidly deploying applications that address today's sophisticated requirements of integration, networking and scalability. However, building these systems from the ground up, without using specialized tools and prebuilt components, demands a great effort and expense.

A software development team building an eBusiness application using EJBs faces challenges that are compound by the increasingly short delivery times. To be successful a development team must be able to perform the following tasks seamlessly in record time frames:

- Master changing and complex technologies
- Model the business process accurately using object-oriented methodologies
- Design an application architecture that takes advantage of the infrastructure
- Implement, test, and deploy all business functions as Enterprise JavaBeans

Developers can take advantage of application server features and advanced tools to accelerate development but regardless of infrastructure and tools, they still must build all the business objects that make up their application. This task involves a tremendous amount of risk and effort.

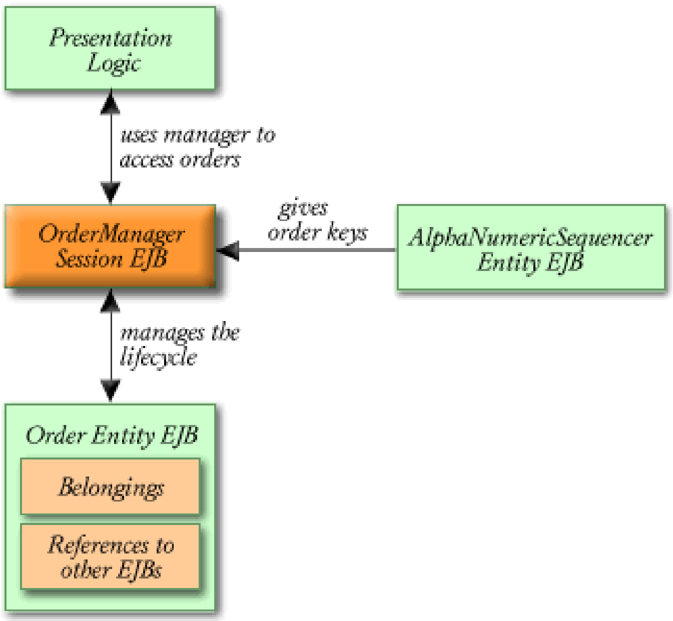
Build versus Buy: WLCS components offer the best of both solutions.

In the process of planning eBusiness applications, corporations are faced with a build or buy decision. Building an application of this kind from the ground up would consume considerable time and resources. On the other hand, an off-the-shelf application does not meet the company's unique needs. The best solution is to use *components*. Components are packages of pre-built business functions that jump-start the development of an eBusiness application. Components allow developers to customize and snap together enterprise applications quickly, while tailoring them to specific business needs. The result is a complete solution that takes advantage of EJB technology without its time-consuming complexities.

The BEA WebLogic Commerce Server provides a complete family of EJB components for eBusiness. Developers using WebLogic Commerce Server components do not need to start from scratch or master EJB complexities. By using WLCS components, developers can build eBusiness applications customized for their company's unique business needs in record time frames.

How do WLCS components work?

WebLogic Commerce Server components can interact with each other as well as interact with other EJBs outside the component family. They have been modeled, designed, built, and tested to work together as a family and in combination with third party and customer-built EJBs. Applications built using WLCS components take the maximum advantage of EJB and application server technology.



Component Interaction Example

Applications built with WLCS components leverage a scaleable, high-performance Architecture.

WLCS components are designed to work together at run-time in a distributed, highly interactive environment. Once snapped together and deployed, these components can form an efficient, robust eCommerce application or *engine*, automatically leveraging powerful object oriented design patterns and taking full advantage of EJB 1.1 features. The WLCS component architecture provides the framework for scaleable, high-performance, enterprise-class eCommerce applications.

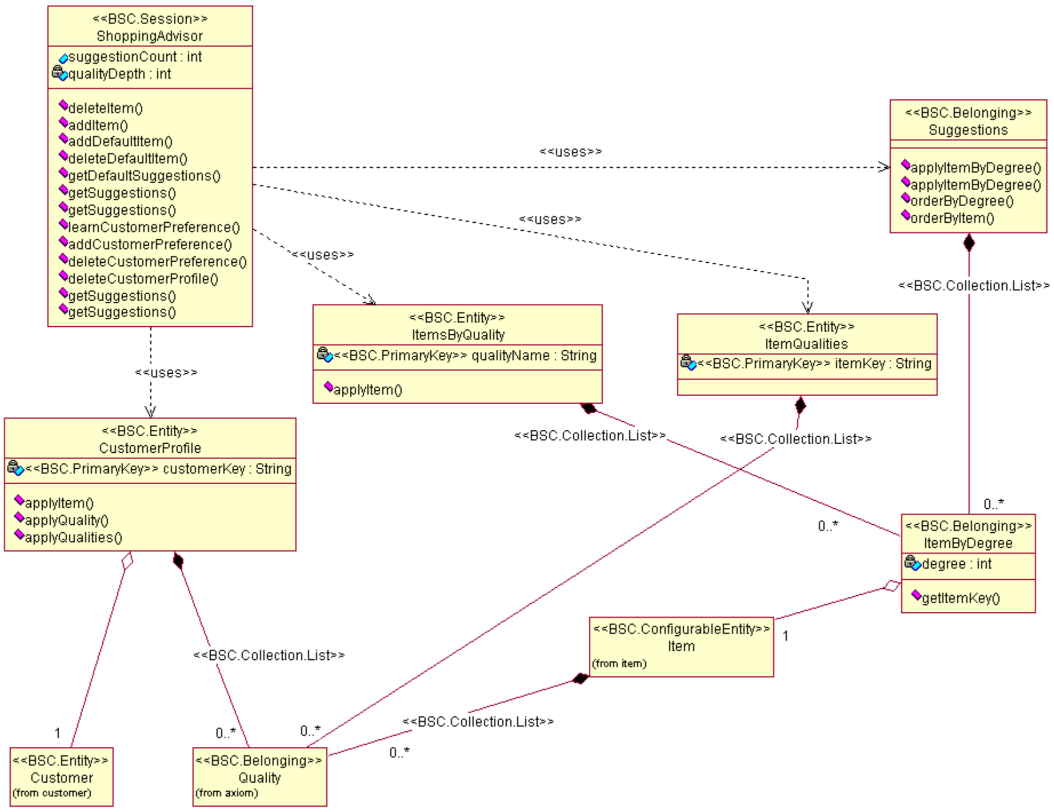
BEA also provides the complete WLCS component *object model* in Unified Modeling Language (UML) diagrams. Developers can either select existing pluggable components from the model or extend and customize components, using a WLCS tool to generate EJB source code based on the model. Either way, the integrity of the object model and its design advantages are ensured in the application development process. Developers can focus on writing the business logic in their applications, and rely on the WLCS architecture to supply all the details of a well-designed EJB application, including transaction processing, messaging, proven design patterns and business policies, efficient database access across the network, and so on. All the good stuff you get by using an object oriented development methodology is built in to the components.

For a live example of this powerful EJB component architecture put to work as a Web presence, see the "Getting Started" topic in the *WebLogic Commerce Server Components MyBuyBeans Tour.*.

Components are easy to use and customize.

WLCS components are designed with usability in mind. Their ease of use allows developers to rapidly customize components and snap together applications with minimal training. Components are easily customizable and can be extended to make new components specific to a particular business. WLCS components include customization tools that integrate with leading modeling tools and Java IDEs. To customize or extend a component, developers simply customize its associated object model, and WLCS automated tools generate the customized code.

Base your eCommerce applications on our smart models and generated EJBs.



WebLogic Commerce Server components make it easy to *model* an application. Using industry-standard Unified Modeling Language (UML), analysts can graphically model your company’s business process, selecting WLCS components from a repository.

WLCS provides you with a basic, EJB 1.1 compliant, eBusiness UML model. Analysts and developers simply extend and modify the base eBusiness model, using the WLCS components as needed.

Once the UML class diagram is completed, all components and all object relationships are automatically generated using the WLCS Smart Generator. The Smart Generator is a tool that transforms a UML representation of your business process into EJB components. All the source code, object definitions, object relationships, documentation, and EJB-required files are automatically created by the Smart Generator. Adding a relationship between two objects is as easy as drawing a line between them. Using visual modeling and roundtrip engineering, changes to the business model can be rapidly translated into changes in the application, greatly reducing maintenance cost and boosting application reliability.

For more information about the development process, see Chapter 3, "Development Process."

Components use industry-standard Design and Analysis Patterns.

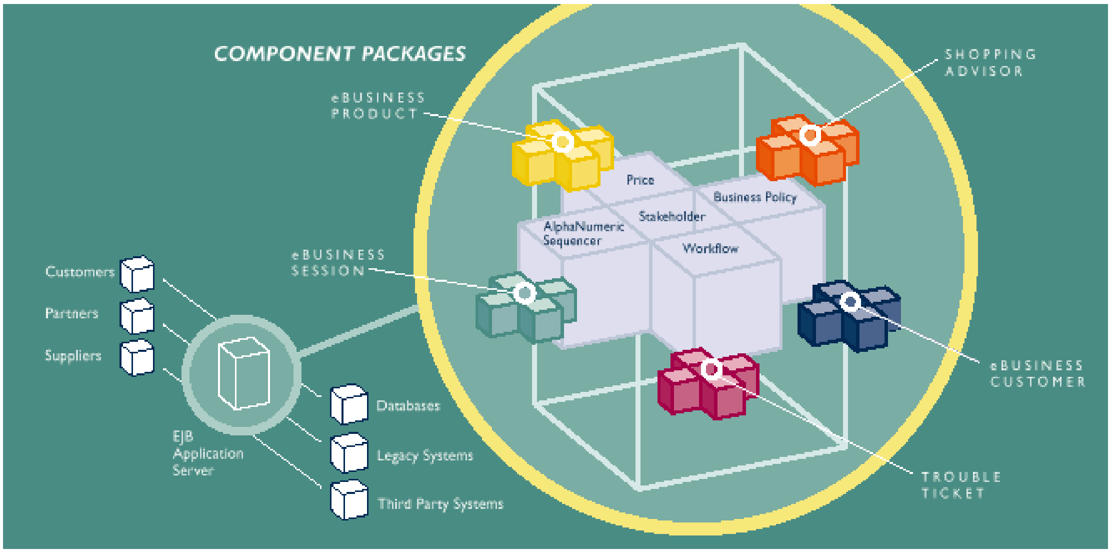
WebLogic Commerce Server components interact with each other using industry standard Design Patterns to solve a wide range of business problems. The use of proven design patterns results in a better business model, lowered cost of development and maintenance, and faster time-to-market.

Industry Standard Design Patterns	WLCS Component Examples	Description	Benefits
Strategy, Policy, and Chain of Responsibility Patterns Individual Instance Method analysis pattern	BusinessPolicy	BusinessPolicy is a set of Behavioral patterns. It lets you interchange business policies. BusinessPolicy pattern works with organizational business hierarchies to manage the company's business policies.	Provides pluggable methods that can be changed at developent time or at runtime. Allows for fast adaptation of new/customized business policies with minimum cost. These new business policies can be tailored for specific clients or organizations.

Industry Standard Design Patterns	WLCS Component Examples	Description	Benefits
Command and Action Patterns	Task	A Behavioral pattern that encapsulates a business process that provides isolation between business logic and business objects.	Allows the separation of business logic from business objects. Results in lower maintenance costs.
Abstract Factory Pattern	SmartHome, BelongingHome	A Creational pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes.	Provides a consistent programming model for creation of objects. Results in lower maintenance costs.
Aggregation with Life Cycle Pattern	Singleton and collection aggregation of business objects by value, and by reference.	A combination of Creational and Behavioral patterns that provide flexible ways to control the life cycle of aggregate objects directly from its owner.	Provides a complete set of powerful and flexible APIs to maximize programming efficiency. Results in a better business model design and lower application development costs
Proxy Pattern	SmartHandle, Collection of remote objects.	A Structural pattern that provides a surrogate or placeholder for another object to control access to it. Allows for lazy evaluation of large collections of remote objects.	Allows for a natural modeling of business relationships without compromising performance

Components are neatly organized in Component Packages.

The WebLogic Commerce Server components family is structured as packages. Each package contains a set of components that plays a specific role in assembling an enterprise application.



- At the lowest level is the **Foundation package**. This package provides an interface to the application server and enhances EJBs with WebLogic Commerce Server design patterns. It allows all WLCS components to take advantage of both EJB technology and our implementation of industry-standard design patterns.
- For commonly used or *core* business functions, WLCS provides the **Axiom package**. This collection of components is designed to provide common business functionality that is used across applications. These components are typically lightweight and can be combined to create new powerful, industry-specific application components.
- WLCS provides *industry specific component packages* for financial services, telecommunications, and Internet retail. Today WLCS includes component packages required to build e-business applications, with an emphasis on Internet e-commerce and customer self-service. Taken as a whole, these industry specific component packages are referred to as the **eBusiness package** within WLCS.

WLCS includes more than 80 out-of-the-box, pluggable Java components designed to provide most of the functionality of essential e-business. For a comprehensive and detailed look at the components, see Chapter 2, “Components Catalog,” or the complete API in Javadoc.

MyBuyBeans.com Example

If you would like to see a working example of how to snap together the WebLogic Commerce Server components to form a robust, high-performance, retail Web site, see the *WebLogic Commerce Server Components MyBuyBeans Tour.*. The tour steps you through the development and deployment process using the imaginary MyBuyBeans.com retailer as a focus.

CHAPTER

2 Components Catalog

The following table lists the Enterprise Java Beans (EJBs) in the BEA WebLogic Commerce Server (WLCS) Component kit and provides links into the complete Javadoc API.

Package/Description	Components	Type
theory.smart.**axiom.accounting** Stores lists of transaction entries (a transaction history) and balances. Used to describe anything from a cash value to an inventory of items.	Account	ConfigurableEntity
	AccountEntry	Entity EJB
	PostingRule	Business Policy
	DefaultPostingRule	Business Policy
theory.smart.**axiom.contact** Customer Information package. Stores personal data used for billing, shipping, marketing and any other services requiring customer contact	Address	Belonging
	Stakeholder	Entity EJB
	PhoneNumber	Belonging
	Email	Belonging
	Person	Entity EJB
	Url	Belonging
	CreditCard	Belonging
	PostalCode	Belonging
theory.smart.**axiom.messaging** Provides messaging between two parties. Has a mailbox along with search and retrieval capabilities.	PostOffice	Session EJB
	MailBox	Entity EJB
	Message	Belonging

Package/Description	Components	Type
theory.smart.**axiom.units** Standard solution to common unit conversion problems. Conversion of one unit of measurement to any other in the same classification. **Note:** The price/money component provides multi-currency support.	UnitPrice	Belonging
	UnitConverter	Session EJB
	UnitConversion	ConfigurableEntity
	UnitList	Entity EJB
	UnitCategories	Belonging
	ConversionFunction	Business Policy
	Unit	Belonging
	Quantity	Belonging
	Quality	Belonging
	Price	Belonging
	DefaultConversionFunction	Business Policy
theory.smart.**axiom.util** Generic utilities package.	AlphaNumericSequencer	Entity EJB
theory.smart.**axiom.workflow** Objects to create a state machine with transitions. Can be used within other components.	StateMachine	Workflow
	TransitionPolicy	Business Policy
	Transition	Belonging
	State	Belonging
theory.smart.**ebusiness.customer** Customer interaction and profile management package. Can be seamlessly mapped to your existing customer database.	Customer	Entity EJB
	CustomerManager	SessionEJB
theory.smart.**ebusiness.giftregistry**	GiftRegistryManager	SessionEJB
	GiftRegistry	EntityEJB
	PurchasedOrderLine	Belonging

Package/Description	Components	Type
theory.smart.**ebusiness.inventory** Distributed interface to your existing inventory system. Interfaces with legacy apps and existing databases	InventoryManager	SessionEJB
	ItemInventory	EntityEJB
	InventoryRecord	ConfigurableEntity
	Locator	Belonging
theory.smart.**ebusiness.invoicing** Distributed interface to your existing invoicing/billing system. Interfaces with legacy apps and existing databases	InvoiceManager	SessionEJB
	Invoice	Entity EJB
theory.smart.**ebusiness.item** Flexible management and access to catalogs of products and services, with dynamic policy-based pricing	Item	ConfigurableEntity
	ItemPriceCalculationPolicy	Business Policy
	DefaultItemPriceCalculationPolicy	Business Policy
theory.smart.**ebusiness.order** Online order entry, order management, and shopping cart functionality	Order	Entity EJB
	OrderManager	SessionEJB
	OrderLine	Belonging
	OrderWorkflow	Workflow
theory.smart.**ebusiness.session** Complete online user session management, including guest, authenticated login and multiple login functions. Sessions are stored transactionally for trouble-free web interaction	ebusinessSessionManager	SessionEJB
	ebusinessSession	Entity EJB
	ebusinessSessionWorkflow	Workflow
theory.smart.**ebusiness.shipping** Distributed interface to your existing shipping/order fulfillment system. Interfaces with legacy apps and existing databases	ShippingManager	SessionEJB
	ShippingMethod	Entity EJB
	PackingList	Belonging
	ShippingCostCalculationPolicy	Business Policy
	DefaultShippingCostCalculationPolicy	Business Policy

Package/Description	Components	Type
theory.smart.**ebusiness.shoppingAdvisor** Personalizes customer's shopping experience. Suggests products and services based on customer profiles and buying patterns. Learns customer profiles. Allows for accurate targeting of offerings to consumers.	ShoppingAdvisor	Session EJB
	CustomerProfile	Entity EJB
	ItemsByQuality	Entity EJB
	ItemsQualities	Entity EJB
	ItemsByDegree	Belonging
	Suggestion	Belonging
theory.smart.**ebusiness.troubleticket** Complete customer support system. Includes ticket entry and response management, with robust transactional workflow	TroubleTicket	Entity EJB
	TroubleTicketWorkflow	Workflow
	TroubleTicketManager	Session EJB
	JournalEntry	Belonging

3 Development Process

WebLogic Commerce Server (WLCS) provides prebuilt Enterprise JavaBean (EJB) components that you can use in your e-commerce Web applications. In the Java implementation source code that WLCS generates, you can add your business logic between specially provided code markers. If needed, you can also extend the WLCS components to add new components that match your specific business requirements.

The first section in this chapter outlines the overall development process. Subsequent sections provide details about each step. The following topics are presented:

- What is the overall development process?
- Before You Begin: Copy the Model
- Step 1: Export the WLCS model in Rational Rose
- Step 2: Run the WLCS Smart Generator
- Step 3: Add Your Business Logic: Edit the Java files and Compile Them
- Step 4: Run the EJB Compiler
- Step 5: Deploy your application, and start the server
- Step 6: If desired, change the model, and iterate

What is the overall development process?

You can create EJB components by modeling them using the Unified Modeling Language(UML) and then generating Java source code. This technique utilizes a UML drawing tool, in this case Rational Rose™, and creates an intermediate file that describes that model. That file is transformed by a WLCS application called the Smart Generator into the Java classes that make up one or more EJBs.

Code generation from UML has long been recognized as a promising technology. This technique is powerful because it allows the designer to model the components in a natural way without being concerned with implementation-specific details.

Despite its promise, this technique has not been adopted widely for a number of reasons:

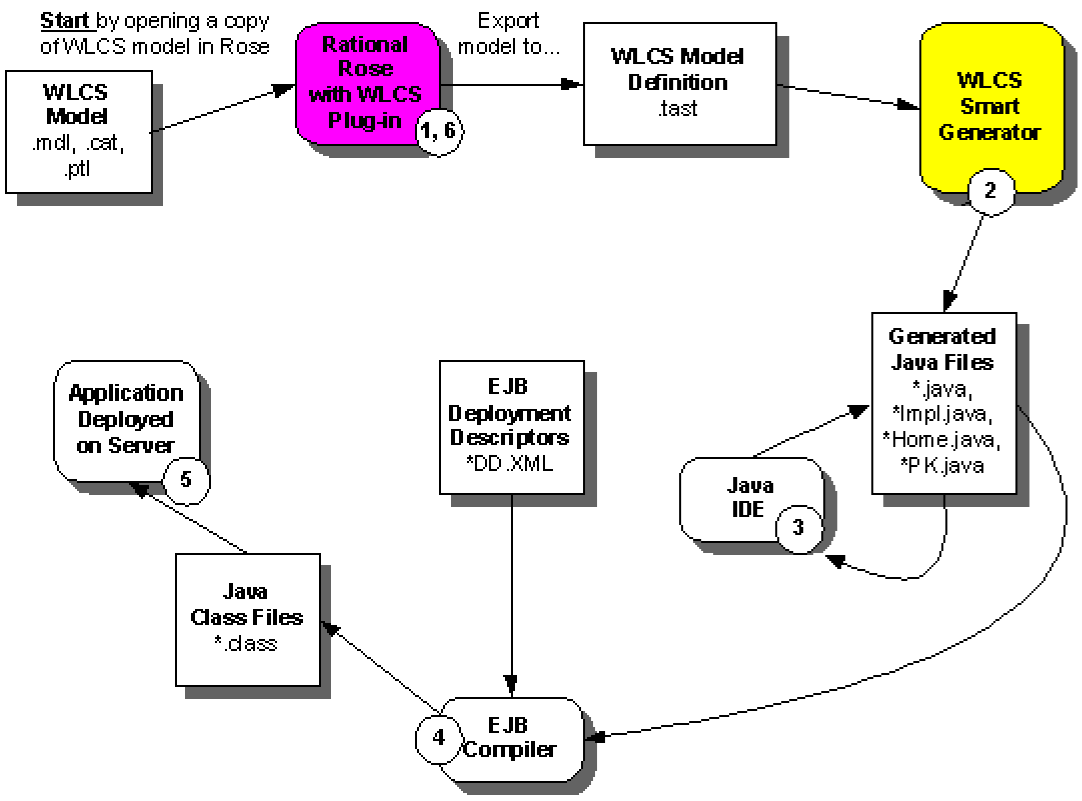
- The generated code was often thought to be inferior, and there was no easy way to generate the implementation of the business logic
- Lack of an iterative development cycle, which meant that most tools could only be used to generate the first attempt at the classes; and an associated problem: often times the model and the code became unsynchronized and much of the model's value was lost.

The WLCS utilities solve these problems by going a step further. The utilities do not assume a direct mapping from the model to the underlying language constructs. The user models the business objects and the Smart Generator creates a set of classes that implements these objects with reference to the Enterprise JavaBeans 1.1 Specification. Many of the laborious tasks of creating access methods and handling containment of references is automatically handled.

The WLCS Smart Generator also uses intelligent algorithms to generate sensible naming for collections and methods. In addition, it generates documentation for these classes using the same intelligent naming scheme. Because the Smart Generator embeds code markers, it is possible for developers to add the business logic and then resynchronize those changes with the model.

Figure 3-1 introduces the development process when you use WLCS components.

Figure 3-1 WLCS Components Development Process


Start by opening a copy of WLCS model in Rose
WLCS Model .mdl, .cat, .ptl
Rational Rose with WLCS Plug-in
1, 6
Export model to...
WLCS Model Definition .tast
WLCS Smart Generator
2
Generated Java Files *.java, *Impl.java, *Home.java, *PK.java
Java IDE
3
EJB Deployment Descriptors *DD.XML
EJB Compiler
4
Java Class Files *.class
Application Deployed on Server
5

The overall steps are as follows:

Before you begin: copy the installed WLCS model

Go to the “`model\BEA WeblogicCommerce\`” directory found under the WLCS installation directory. Create a separate work directory and copy the `BEA WeblogicCommerce.mdl` model file to your work directory.

Step 1: Export the WLCS model in Rational Rose

Start Rational Rose™, a graphical UML modeling product. Open your copy of the WLCS Components model, and use the WLCS plug-in to export the model to an intermediate file (`*.tast`).

Step 2: Run the WLCS Smart Generator

From the Rose menu, or the Windows Start menu, or a command prompt, run the WLCS Smart Generator. The Smart Generator is a Java application that reads the `*.tast` model definition file and generates the Java source files and EJB Deployment Descriptors.

Step 3: Edit the Java files: Add Your Business Logic

Edit the generated `*Impl.java` source files to add your business logic between the provided code markers. Because the Smart Generator embeds code markers, it is possible for you to add the business logic and then resynchronize those changes with the model.

Step 4: Run the EJB Compiler

Run the EJB compiler to generate the Java class files for your EJBs.

Step 5: Deploy your application, and start the server

Deploy the application using either Bean-Managed Persistence (BMP) or Conainer-Managed Persistence (CMP) to the host system or systems. Then start the WLCS Server on each machine that hosts the application.

Step 6: If desired, change your copy of the model, and iterate

If you want, you can change your copy of the WLCS model, adding new components or business policies that extend the ones provided by WLCS. You can then iterate through the development process (starting again at step 1). Smart Generator preserves the changes you made to the `*Impl.java` source files by locating your additions within the code markers.

Subsequent sections in this chapter provide details on the steps in the development process.

Before You Begin: Copy the Model

This prerequisite step is simple.

1. Go to the "`model\BEA WeblogicCommerce\`" directory found under the WLCS installation directory. The `BEA WeblogicCommerce.mdl` file in that directory is the model.
2. Create a subdirectory that will contain your copy of the model. In the WLCS Components Tour, readers are instructed to create a \model\tour\ subdirectory. However, you can create your work directory in a location that is separate from the installed WLCS folder hierarchy, such as `d:\myWebApps\work\`.
3. Copy the WLCS Component model file, `BEA WeblogicCommerce.mdl`, to the work directory that you created.

Step 1: Export the WLCS model in Rational Rose

Start Rational Rose™, a graphical UML modeling product. If you installed the WLCS software after you installed Rational Rose, as described in the WLCS Installation Guide, the WLCS plug-in to Rational Rose is already in place.

Open your copy of the WLCS Components model, and export the model to an intermediate definition file (`*.tast`). If at this time you are not extending the model and generating new classes based on existing WLCS classes, the procedure in this section is simple.

(See the section "Step 6: If desired, change the model, and iterate" for more advanced considerations if you are extending the WLCS classes.)

The steps during your initial cycle through the development process are as follows:

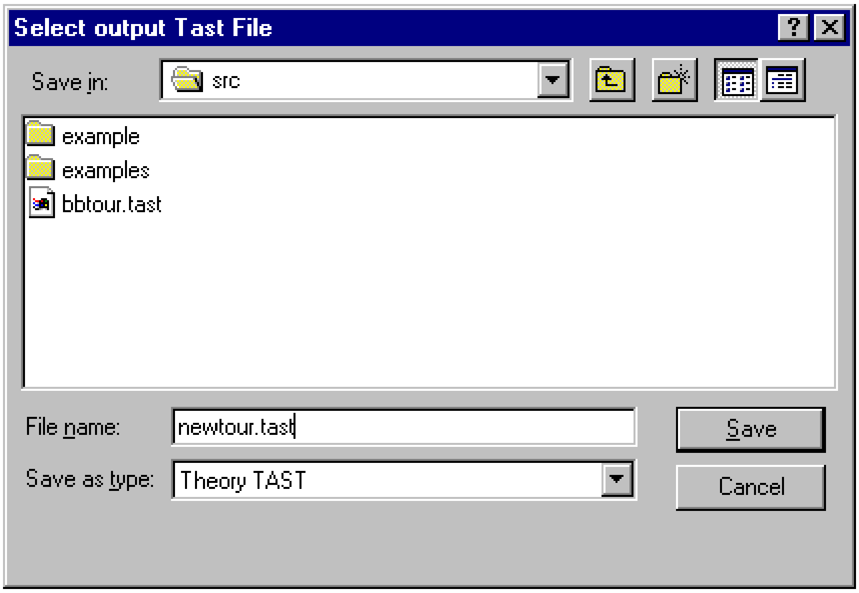
1. Start Rational Rose. From the Windows Start menu, select Start → Programs → Rational Rose...

2. From the Rational Rose top-level menu, click File → Open, and browse to the directory where you put your **copy** of the model file, `BEA WeblogicCommerce.mdl`.

3. Double click on the `BEA WeblogicCommerce.mdl` file. Rational Rose opens the model.

4. From the Rational Rose top-level menu, click Tools → WeblogicCommerce→ Export Model As...

5. The WLCS plug-in to Rational Rose displays the following screen:


Select output Tast File

Save in: src

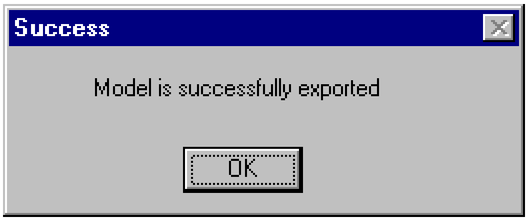
example
examples
bbtour.tast

File name: newtour.tast

Save as type: Theory TAST

6. Enter a file name for the model definition file. The default file type is `.tast`.

7. The WLCS plug-in to Rational Rose displays a confirmation message when the export operation completes successfully:



Model is successfully exported

Step 2: Run the WLCS Smart Generator

The WLCS Smart Generator reads the `*.tast` model definition file and generates the Java source files based on an industry standard modeling language.

Advantages

Smart Generator provides the most comprehensive EJB code generation in the industry. The EJB code that is generated by the WLCS Smart Generator is optimized for the high performance demands of interactive e-commerce Web applications. The code is based on input from industry-leading persistence experts and systems integrators. The advantages of using the WLCS Smart Generator include:

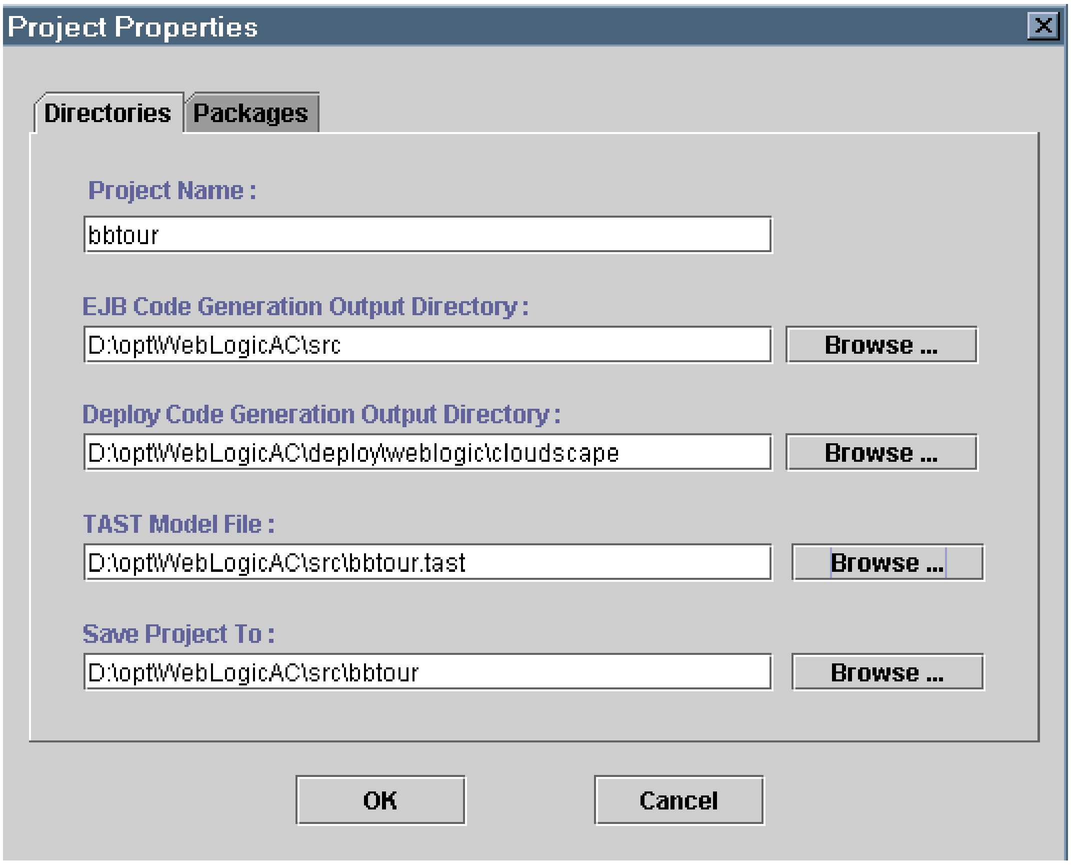
- It allows you to focus on building your company's business logic
- It enables reusability and customization of prebuilt EJB components
- You do not have to know the details of EJB or track changes in the EJB Specification from Sun
- You do not have to know the details of implementing EJB code for high performance
- Preservation of your investment: your business logic can be regenerated to conform to revisions in the Sun Microsystems EJB Specification

After you update the generated Java implementation files, where you add your business logic (Step 3 in the development process), the WLCS Smart Generator reads in your changes the next time it runs, preserves your changes, and reflects the changes back in your copy of the model. Special code markers are provided in the generated `*Impl.java` files that enable the Smart Generator to synchronize the model with your business logic code.

Define a New Project

To define a new Smart Generator project, follow these steps. You can also see these steps with sample values in the WLCS Components Tour. The tour includes a great walk-through of extending an Item component and extending a pricing business policy.

1. Start the Smart Generator. Use one of the following options:
   - From the Rational Rose top-level menu, select Tools → WeblogicCommerce→ Smart Generator.
   - From the Windows Start menu, select Start → Programs → BEA WebLogic Commerce Server → Smart Generator.
   - From a system prompt, run the script that starts the Smart Generator, which is implemented as a Java application. On Windows systems, the `smart-generator.bat` file is in `\bin\win32\` in the installed WLCS directory. On Solaris systems, the `smart-generator.sh` file is in `/bin/solaris2/` in the installed WLCS directory.
2. On the initial Smart Generator screen, click the New button.
3. Smart Generator displays its Project Properties screen. For example:



Project Name : bbtour

EJB Code Generation Output Directory : D:\opt\WebLogicAC\src

Deploy Code Generation Output Directory : D:\opt\WebLogicAC\deploy\weblogic\cloudscape

TAST Model File : D:\opt\WebLogicAC\src\bbtour.tast

Save Project To : D:\opt\WebLogicAC\src\bbtour

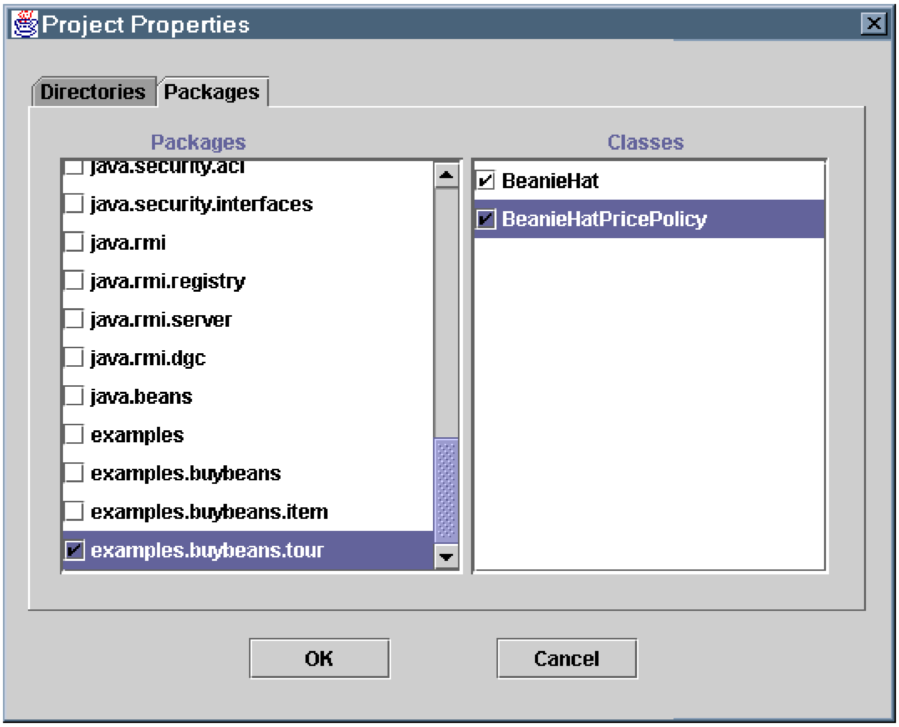
4. In the Project Name field, enter a descriptive project name.
5. In the EJB Code Generation Output Directory, enter the location for the generated Java source files. This is the location where Smart Generator will place the Java interfaces, business logic, and other core EJB code.
6. In the Deploy Code Generation Output Directory, enter the location for the generated deployment code source files. This is the location where Smart Generator will place deployment descriptors and files containing JDBC instructions to persist Entity Beans using a specified database map.
7. In the TAST Model File field, enter the name of the `*.tast` file you exported from your copy of the WLCS model (in Step 1).

8. In the Save Project To field, designate a directory that will contain the project definition file.

9. Click the OK button.

Configure the Project

1. Click on the **Packages** tab.



Packages:
- [ ] java.security.acl
- [ ] java.security.interfaces
- [ ] java.rmi
- [ ] java.rmi.registry
- [ ] java.rmi.server
- [ ] java.rmi.dgc
- [ ] java.beans
- [ ] examples
- [ ] examples.buybeans
- [ ] examples.buybeans.item
- [✓] examples.buybeans.tour

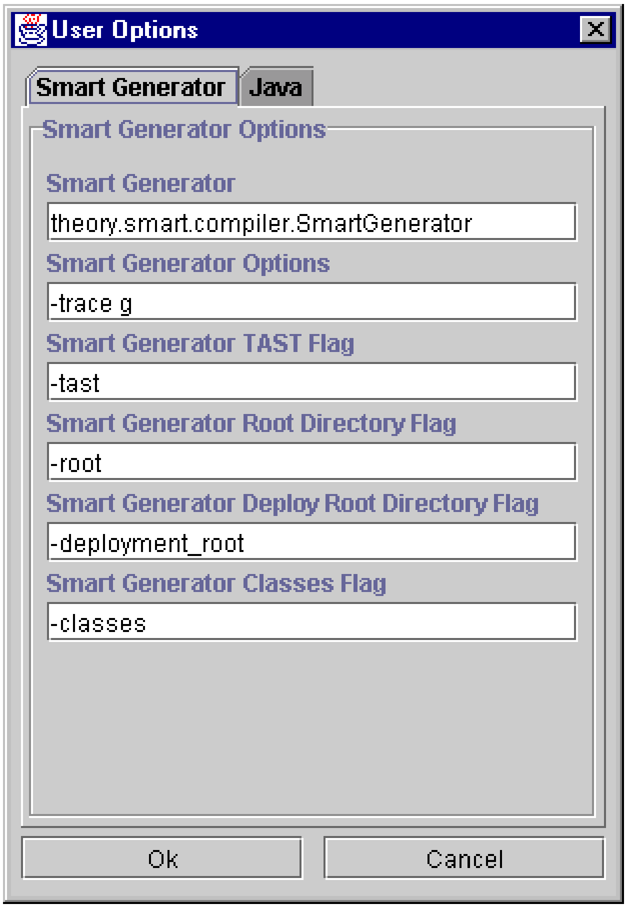
Classes:
- [✓] BeanieHat
- [✓] BeanieHatPricePolicy

2. In the **Packages** pane, click on the package(s) you want to implement.

3. In the **Classes** pane, double-click on the boxes next to the classes you want to implement. Double-clicking adds checkmarks next to selected classes.

4. Click **OK**.

5. Select Configuration→ Options from the Smart Generator menu. The User Options screen is displayed:



Smart Generator Options

Smart Generator: theory.smart.compiler.SmartGenerator

Smart Generator Options: -trace g

Smart Generator TAST Flag: -tast

Smart Generator Root Directory Flag: -root

Smart Generator Deploy Root Directory Flag: -deployment_root

Smart Generator Classes Flag: -classes

6. If needed, you can enter the following options on this Smart Generator screen:

`-trace`

Sets the trace switch. The default value is g, which enables a code Generator trace. Other options: `-trace c` (compiler trace), `-trace t` (Tast processor trace), and `-trace +` (trace all).

`-tast`

Sets the metadata exchange format (default and only option in the current release).

`-root`

Sets the location of the root directory for the `.java` source files to be generated.

`-deployment_root`

Sets the root directory for output of deployment-specific code.

`-classes (c1, c2, c3)`

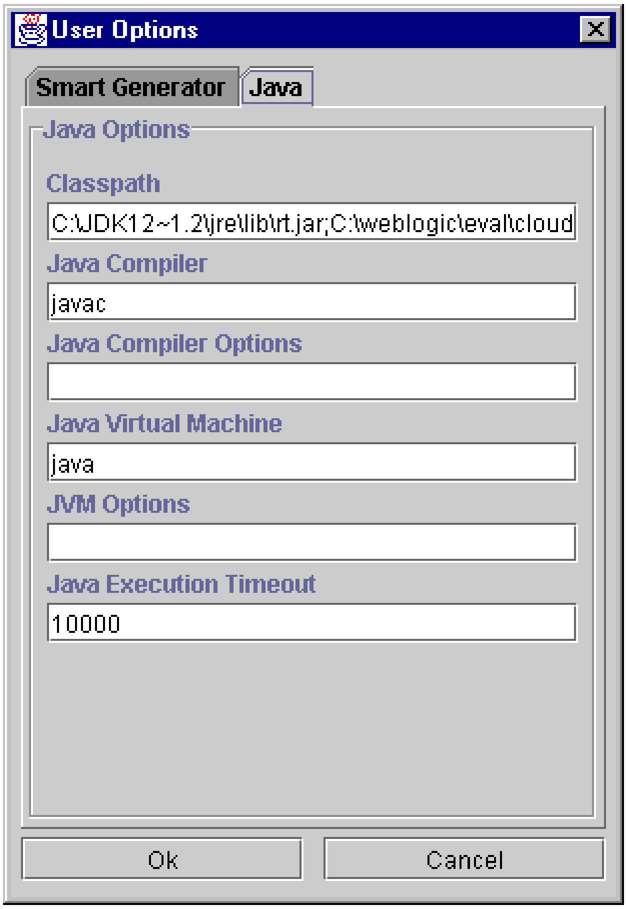
Allows you to specify a subset of classes to be processed by the next Generate operation in the WLCS Smart Generator. Enclose multiple entries in parentheses, separated by a comma. There is no default.

`-bmp`

Switch to specify that you *do not* want to use Bean-Managed persistence. In other words, sets bean-managed persistence to *off*. Do this if you want to use container-managed persistence instead.

The default is to deploy with this switch *on* using bean-managed persistence on a relational database such as Oracle. In the default *on* mode, the Smart Generator references a database mapping properties file to generate the appropriate database-related code.

7. On the User Options screen, you can also click the Java tab if you want to specify Java compiler options and related options. The WLCS Smart Generator displays a screen similar to the following:



Classpath: C:\JDK12~1.2\jre\lib\rt.jar;C:\weblogic\eval\cloud

Java Compiler: javac

Java Compiler Options:

Java Virtual Machine: java

JVM Options:

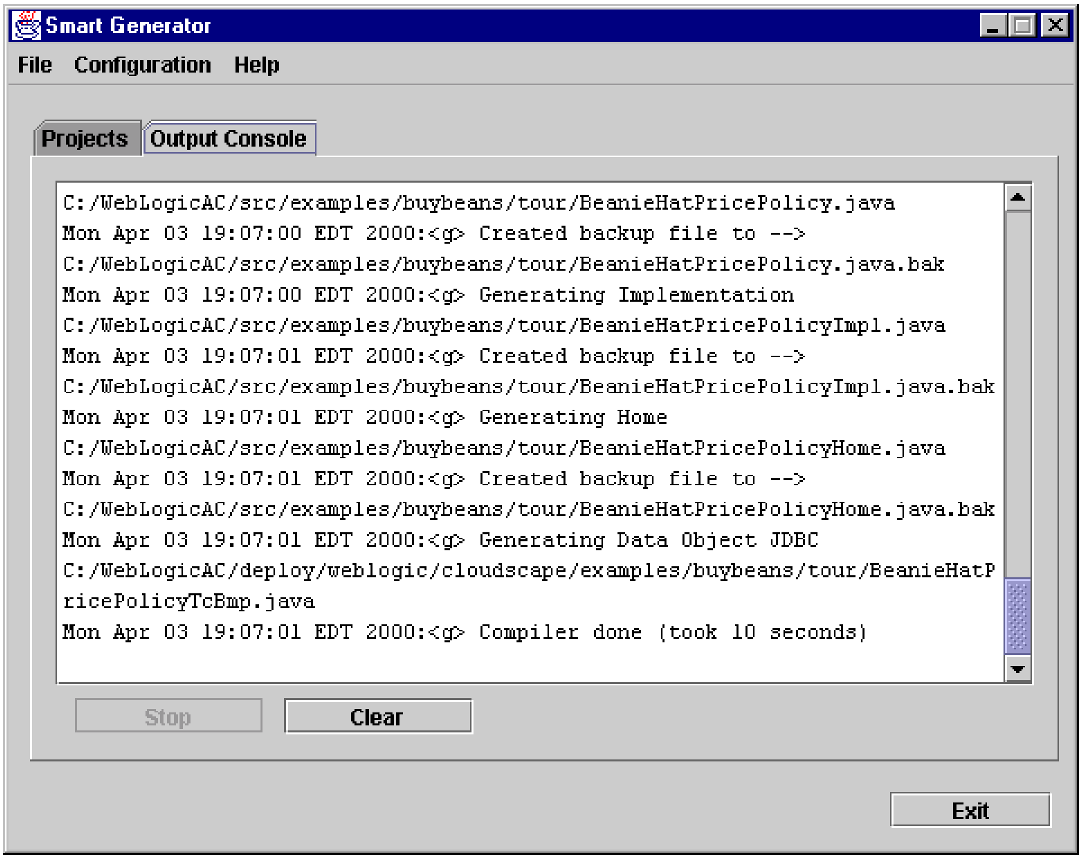
Java Execution Timeout: 10000

8. Click the Ok button after you enter any compiler options.

Generate the Java Sources

When you are ready to generate the Java sources, click the Generate button on the main WLCS Smart Generator screen. Wait for the Smart Generator to complete its work. You can click the Output Console tab to view messages recorded during the generation step. Look for the "Compiler done" message at the end of the console output.

For example:



```
C:/WebLogicAC/src/examples/buybeans/tour/BeanieHatPricePolicy.java
Mon Apr 03 19:07:00 EDT 2000:<g> Created backup file to -->
C:/WebLogicAC/src/examples/buybeans/tour/BeanieHatPricePolicy.java.bak
Mon Apr 03 19:07:00 EDT 2000:<g> Generating Implementation
C:/WebLogicAC/src/examples/buybeans/tour/BeanieHatPricePolicyImpl.java
Mon Apr 03 19:07:01 EDT 2000:<g> Created backup file to -->
C:/WebLogicAC/src/examples/buybeans/tour/BeanieHatPricePolicyImpl.java.bak
Mon Apr 03 19:07:01 EDT 2000:<g> Generating Home
C:/WebLogicAC/src/examples/buybeans/tour/BeanieHatPricePolicyHome.java
Mon Apr 03 19:07:01 EDT 2000:<g> Created backup file to -->
C:/WebLogicAC/src/examples/buybeans/tour/BeanieHatPricePolicyHome.java.bak
Mon Apr 03 19:07:01 EDT 2000:<g> Generating Data Object JDBC
C:/WebLogicAC/deploy/weblogic/cloudscape/examples/buybeans/tour/BeanieHatP
ricePolicyTcBmp.java
Mon Apr 03 19:07:01 EDT 2000:<g> Compiler done (took 10 seconds)
```

Click the Exit button to close the WLCS Smart Generator.

Step 3: Add Your Business Logic: Edit the Java files and Compile Them

1. Edit the Java files to add your business logic.

   Edit the generated `*Impl.java` source files to add your business logic between the provided code markers. Because the WLCS Smart Generator embeds code markers, it is possible for you to add the business logic and then resynchronize those changes with the model.

   When you run the Smart Generator again, it recognizes the changes you have made in your *Impl.java sources and reflects those changes in the model.

   The code fragment in Listing 3-1 shows the type of code markers that are provided by the WLCS Smart Generator. The sample generated implementation file is `BeanieHatPricePolicy.java`, which you can create by running the WLCS Components Technical Tour in the online documentation. A **bold** typeface is used in the listing to highlight the markers.

   Note: In all cases, the end tag, such as $_End, must appear **before** the closing brace of the additional method.

Listing 3-1 Code Markers Indicating Where to Insert Your Business Logic

```
package examples.buybeans.tour;

import theory.smart.foundation.*;
import theory.smart.util.*;

//$Import$_Begin ------------ CUSTOM CODE ---------------
// Place additional import statements here
//$Import$_End   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

   .
   .
   .

class BeanieHatPricePolicyImpl implements BeanieHatPricePolicy
//$Implements$_Begin ------------ CUSTOM CODE ---------------
// Add interfaces that are implemented here
//$Implements$_End   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
    .
    .
    .

//$AdditionalAttributeDeclarations$_Begin ------------ CUSTOM CODE
---------------
// Add additional attribute declarations here
//$AdditionalAttributeDeclarations$_End
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    .
    .
    .

protected BeanieHatPricePolicyImpl()
{
  super();
//$Constructor$_Begin ------------ CUSTOM CODE ---------------
// Add constructor code here
//$Constructor$_End   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
}

    .
    .
    .

//$MethodException theory.smart.axiom.units.Price
calculatePrice(theory.smart.ebusiness.item.Item item,
theory.smart.axiom.units.Quantity qty,
theory.smart.ebusiness.customer.Customer customer)$_Begin
------------ CUSTOM CODE ---------------

// Add additional exceptions here

//$MethodException theory.smart.axiom.units.Price
calculatePrice(theory.smart.ebusiness.item.Item item,
theory.smart.axiom.units.Quantity qty,
theory.smart.ebusiness.customer.Customer customer)$_End
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

{
//$Method theory.smart.axiom.units.Price
calculatePrice(theory.smart.ebusiness.item.Item item,
theory.smart.axiom.units.Quantity qty,
theory.smart.ebusiness.customer.Customer customer)$_Begin
------------ CUSTOM CODE ---------------

  return null; //in Components Tour, custom pricing policy
               code in WLCS documentation is inserted here...
//$Method theory.smart.axiom.units.Price
calculatePrice(theory.smart.ebusiness.item.Item item,
theory.smart.axiom.units.Quantity qty,
theory.smart.ebusiness.customer.Customer customer)$_End
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
}

    .
    .
    .
```

Note: If you change the signature of a method it will not be properly managed by the WLCS Smart Generator. This happens because the round-trip engineering feature works by matching the exact signature of the method and the parameters. If a generated method is no longer present in the model it will simply be deleted, along with the associated implementation. To avoid this situation, you must make matching changes in the model and the source code. As a consequence it is extremely important to consider the parameters to methods up front so as to avoid this problem.

2. Compile the `*.java` source files with any supported Java compiler. For example:

   ```
   javac surfcity.java sunhat.java
   ```

Step 4: Run the EJB Compiler

The EJB compiler (`ejbc`) generates container classes according to the deployment properties you have specified in your deployment files. For more information on deploying EJBS, see the topics Deploying EJBs in WebLogic Server and Deploying EJBs with DeployerTool in the *BEA WebLogic Server Enterprise JavaBeans documentation*.

To run the EJB compiler against your files:

1. Create the files `ejb-jar.xml` and `weblogic-jar.xml` according to the Weblogic EJB deployment guide. For container-managed persistence (CMP) deployments, the `beanname-CMP-RDBMS.xml` will also be needed.

2. Create a deployment jar file that contains the xml files and the Home, Remote and Impl class files. To do this, use the syntax `jar cvf <jar-file> <files...>` For example:

   ```
   jar cvf morehats.jar propeller.class button.class
   ```

   Note: For help, type `jar` at the command prompt, and it will show the usage parameters.

3. Run `ejbc` on the deployment jar. The server and client stubs will be generated inside `that jar.`

4. Finally, copy your application JAR files to the deployment system that will host your application.

You can automate running the EJB compiler tasks for subsequent builds by creating a `*.bat` (NT) or `*.sh` (Solaris) build script file. For an example of a build script, see the `tour-build.bat` script in `bin\win32` under the WLCS installed directory. In addition to generating the *.class bytecodes for your application, the script should create or update the Java Archive (JAR) file for your application.

Note: `tour-build.bat` will compile the tour files, `tour-deploy.bat` will jar them up, and run `ejbc` on that jar.

Step 5: Deploy your application, and start the server

Deploy your application using either Bean-Managed Persistence (BMP) or Conainer-Managed Persistence (CMP) to the server. Then start the WLCS Server on each machine that hosts the application.

This section provides a brief overview of deployment. For more details, see Chapter 4, “Deploying Your Application.”

Before You Start the WebLogic Application Server

Before you start the WebLogic application server, you must do the following:

- Set the environment variable `DEPLOYMENT_SET` to either `BMP` for bean-managed persistence, or `CMP` for container-managed persistence.
- In the directory where you installed WLCS, rename the appropriate `weblogic-`**`XXX`**`.properties` file to `weblogic.properties`, where **`XXX`** is either **`BMP`** or **`CMP`**. The renamed file should reside in the top-level WLCS installed directory, such as `c:\webLogicCommerce`.
- In the directory where you installed WLCS, rename the `weblogiccommerce-`**`XXX`**`.properties` file to `weblogiccommerce.properties`, where **`XXX`** is either **`BMP`** or **`CMP`**. The renamed file should reside in the top-level WLCS installed directory, such as `c:\webLogicCommerce`.

The WLCS EJB source code is independent of the persistence method, whether it is container-managed or any implementation of bean-managed persistence.

The source code that uses JDBC instructions to persist Entity Beans is generated according to one of the BEA reference implementations of bean-managed persistence. If you use the reference data model, the source code may be used as generated. The more likely case is that the JDBC source code may be used as templates to persist EJBs to a legacy data model.

One of the advantages of the WLCS approach is that business logic and other core EJB source code is independent of the persistence implementation. This isolates business application development from database development and schema changes.

Starting the Server

On the deployment system that will host your application, you must start the WebLogic Server.

1. Edit the `weblogic.properties` file so that the jar created in the last step is part of the `weblogic.ejb.deploy` field.

2. Make sure that the class path set in `sethome.sh` or `sethome.bat` points to the class files that are needed by your bean. Also, make sure that the JAR file that you created is *not* in the classpath.

3. Start the server using either `StartCommerce.sh` for UNIX systems or `StartCommerce.bat` for Windows NT systems.

Step 6: If desired, change the model, and iterate

If desired, you can change your copy of the WLCS model, adding new components or business policies that extend the ones provided by WLCS. You can then iterate through the development process (starting again at step 1).

When you extend the WLCS model, Smart Generator preserves any changes you made to the `*Impl.java` source files by locating your additions in between the code markers.

Changing your copy of the model requires a basic understanding of:

- A simple subset of UML notations
- The WLCS model's Foundation packages
- The Smart Generator rules

To master these concepts, we have provided the following topics in this section:

- Do I have to be a Rational Rose or UML Expert?
- Understanding the Foundation Package and Stereotypes
- Understanding the Basic UML Modeling Notations
- WLCS Smart Generator Rules: Factors that Influence the Generated Java Files
- Design Decisions

Do I have to be a Rational Rose or UML Expert?

It is not necessary to be an expert in UML concepts or Rational Rose to model your EJBs. The remainder of this chapter takes a step-by-step approach to explain our technique to creating EJBs from UML. This discussion does not assume a familiarity with either of these topics and provides introductory explanations of the key elements of each. While knowledge of these specific technologies is not assumed, familiarity with the underlying concepts of object-oriented design, distributed objects, and transaction services is required.

There are a number of references in this document to various "Design Patterns" and "Analysis Patterns". There is a welcome trend towards documenting these axiomatic solutions to common computer science problems.

Understanding the Foundation Package and Stereotypes

The `theory.smart.foundation` package is a set of classes from which the WebLogic Commerce Server components are built. These classes provide the building blocks for the value added features of our components. Most of the classes that are generated from the model are derived from classes in the Foundation package.

For example, the `theory.smart.ebusiness` package contains classes that are built on the Foundation package.

To simplify the complexity of the UML diagrams, the Foundation package relationships are described through class stereotypes rather than inheritance. Each of these stereotypes is used to model certain behaviors and implies the presence of additional methods. This section discusses the Foundation package from a conceptual viewpoint. If you need to extend the functionality of theory.smart.ebusiness classes, it is helpful to first understand the theory.smart.Foundation package.

With that goal in mind, this section describes the following concepts used in the Foundation package:

- Belongings
- Sessions
- Entity

- Configurable Entity
- Business Policy
- Workflow
- Smart Features

Belongings

A Belonging is the simplest form of a WLCS Component. A Belonging is a lightweight, local object that can be serialized. A Belonging gets its name because it must "belong" to, or be acquired from, another object, typically a Session or Entity. It must be serializable so that it can be persisted with the class to which it belongs and passed remotely as a parameter.

One of the key characteristics of a Belonging is that it must be implemented using the Abstract Factory pattern. This means that for each belonging there is a home class, an interface, and at least one implementation of that interface. Because access to the object is through an interface, there is a guaranteed level of abstraction. This provides a great deal of flexibility because it means that you can substitute implementations. You could, for instance, make the object remote without changing the code that uses it. Alternatively, you might substitute different business logic at runtime by changing the implementation returned by the home class.

Implementing all these classes by hand is lot of work. The WebLogic Commerce Server development tools simplify the process by generating all of the necessary classes automatically. You can fully concentrate on modeling the attributes and methods so that they fit the needs of your business.

Sessions

Session components, implemented as Session EJBeans, are used to model service-oriented objects. The key concept is that a Session is an object that provides access to a service implemented in itself or somewhere else on the network. Attributes of a session are used only to configure it for use during the lifecycle of that session. It is important to note that the attributes of a Session are not persistent. The business methods are the most important part of a Session.

Sessions provide a way of remotely implementing business logic, thus extending the reach of your client application. For instance, when you need to perform an extended set of operations on a collection of remote objects it often makes sense to create a "Manager". The Manager object can be co-located with the objects it will be operating on. This will reduce the network overhead and latency.

Sessions are also commonly used to provide an interface to a legacy system or to a service that is pinned to a specific piece of hardware. The remote interface allows the client software to access the remote device as if it were local.

Finally, by wrapping a subsystem and factoring out the functions common to similar systems it is possible to provide a level of redundancy. An example of this would be the case where there are multiple providers of credit card validation services. These systems would likely have similar function but different implementations. By creating a common interface to use the different implementations, it is possible to load balance between them or substitute one for the other.

Entity

An Entity, implemented as an Entity EJB, is an object with staying power. Persistence is the key aspect of an Entity object. In its simplest form, an instance of an entity could be the equivalent of a single row in a relational database. This is an over-simplification because each Entity may include collections of attributes and implement business methods.

Entities are representative of the attributes of which they are composed. This is what distinguishes them from Sessions, which represent a collection of services. As a general rule Entities do not implement sophisticated business logic, instead, they are the components that are acted upon.

Configurable Entity

In addition to the standard qualities associated with an EJB Entity, the WLCS Component software provides dynamic configuration. Dynamic configuration is the ability to add properties and methods at runtime and is provided by the Configurable Entity. The Configurable interface allows the programmer to associate a named value with the Entity. These values are persisted separately so that they are permanently associated with the object without affecting the underlying schema.

When the value stored in a Configurable Entity is a method, the result is the ability to exchange the implementation of a method dynamically or a “Pluggable Method” which is the implementation of the “Strategy” pattern.

Business Policy

Configurable Entities can be arranged in a hierarchy of successors. When this type of hierarchy is in place, a request to retrieve a value from a Configurable Entity triggers an upward search through the hierarchy of successors until a matching value is found or the top of the hierarchy is reached. This is the implementation of the "Chain of Responsibility" design pattern.

The combination of “Pluggable Methods” and the hierarchy of succession is called a Business Policy.

Workflow

For many business applications a simple mechanism to maintain internal state is all that is required to achieve a basic level of workflow. The WLCS software provides such a capability for defining and verifying the states and events that describe a business process. What this means to the developer is that they can represent this process as a state diagram and then verify the legitimacy of business method invocations with a single method call to ask for a transition. Adding a step is as simple as adding a new state. The engine will then enforce the rule that this step must be taken without changes to existing code.

Smart Features

The WLCS Components software implements built-in advanced features that considerably improve the ease of use and efficiency of the final system.

SmartKey

The EJB specification requires that for each Entity there is a class that represents the attributes of the primary key of that class. This Primary Key class is used to find and test the equality of instances of Entity objects. To accomplish these simple goals the EJB specification only requires that the Primary Key class must be serializable.

The SmartKey interface extends this functionality and requires the implementation of the Comparable interface from the java collection API. This is so that SmartKeys can be easily compared and stored in ordered lists. The result is that it is easy to model relationships that require the ordering of Entities.

The `toString` method of a SmartKey simplifies the implementation of profiling and debugging code.

SmartHandle

The EJB specification provides for the passing of lightweight references to Enterprise Java Beans through the use of Handles. A handle in EJB is an opaque type that can be converted to and from an EJB Object. A handle is required to implement a test for equality such that given two handles it is possible to determine if they refer to the same Session or Entity object.

For a WLCS Entity component it is possible to create a SmartHandle that includes the object’s associated SmartKey. Because the SmartKey implements the Comparable interface it is possible to order a list of smart handles without accessing the remote objects that they refer to. This simple mechanism greatly improves performance.

SmartValue

Each Entity is composed of the attributes that describe it. In order to encapsulate the remote objects all attributes must be read and written through accessor methods, typically named `get<Attr>` and `set<Attr>`. This has the negative consequence that retrieving the attributes of an entity may result in many remote method invocations. To alleviate this problem the WLCS software provides a convenience class, derived from SmartValue, that contains a copy of all the top-level attributes.

Understanding the Basic UML Modeling Notations

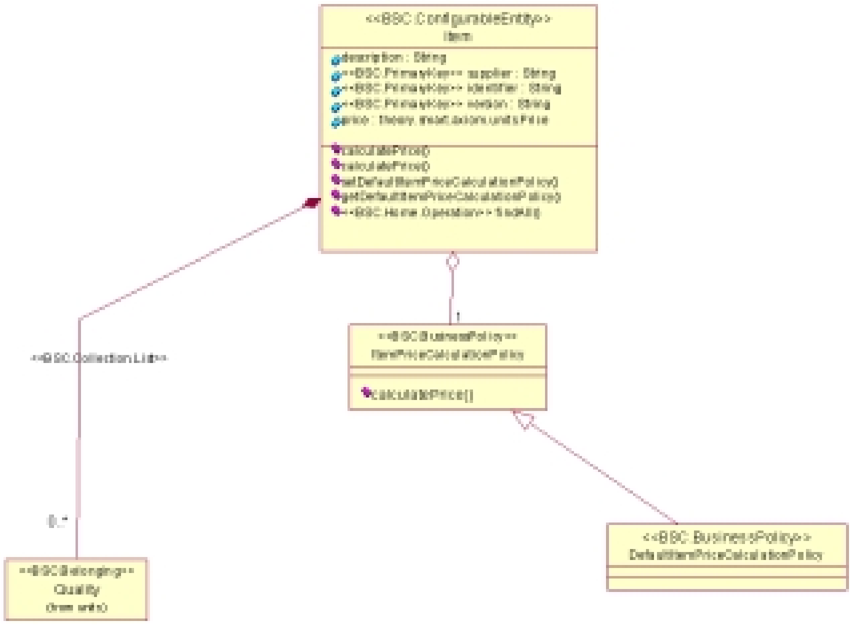
You only need to know a small subset of the UML notations to use the WLCS components model. This section explains the UML notations for:

- Classes and Stereotypes
- Inheritance
- Aggregation and Multiplicity

- Packages

UML describes objects and their relationships graphically. The WLCS Components software uses UML as a mechanism for simplifying the design and implementation of EJBs. Before we discuss the details, let’s review some of the UML notation from a higher level perspective. In this section we focus on the aspects of the notation that are of particular interest to the WLCS Smart Generator, analyzing portions of the sample UML diagram in Figure 3-2.

Figure 3-2 Sample UML Diagram

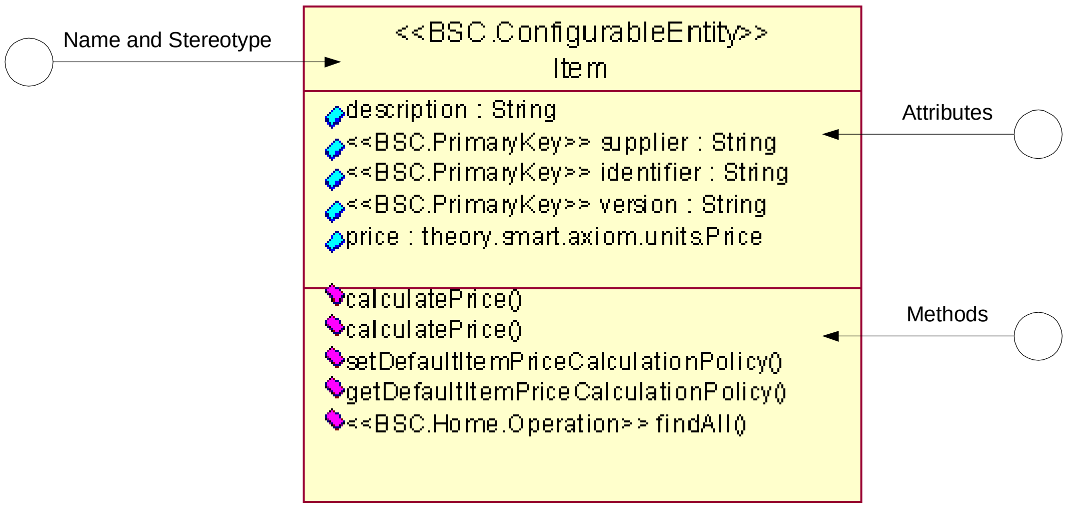


<<BSC ConfigurableEntity>>
Item
description : String
<<BSC PrimaryKey>> supplier : String
<<BSC PrimaryKey>> identifier : String
<<BSC PrimaryKey>> version : String
price : theory.smart.axiom.units.Price
calculatePrice()
calculatePrice()
setDefaultItemPriceCalculationPolicy()
getDefaultItemPriceCalculationPolicy()
<<BSC Home Operation>> findAll()
1
<<BSC Collection List>>
0..*
<<BSC Belonging>>
Quality
(from units)
<<BSC BusinessPolicy>>
ItemPriceCalculationPolicy
calculatePrice()
<<BSC BusinessPolicy>>
DefaultItemPriceCalculationPolicy

Classes and Stereotypes

Let’s start by focusing on the Java classess and stereotypes in Figure 3-3.

Figure 3-3 Compartments in each Class Box


Name and Stereotype
<<BSC.ConfigurableEntity>>
Item
description : String
<<BSC.PrimaryKey>> supplier : String
<<BSC.PrimaryKey>> identifier : String
<<BSC.PrimaryKey>> version : String
price : theory.smart.axiom.units.Price
calculatePrice()
calculatePrice()
setDefaultItemPriceCalculationPolicy()
getDefaultItemPriceCalculationPolicy()
<<BSC.Home.Operation>> findAll()
Attributes
Methods

Each of the rectangles in a diagram is a representation of a class in UML. There are generally three compartments in each class box. A compartment may be left out if it is empty or if the details of the contents are not pertinent to a particular diagram. The latter is often the case when an object from another package is referenced.

The upper most box holds the class name and its stereotype. A stereotype is a "sub-classification" of an element in the model. It is represented as the name of the stereotype enclosed in guillemets, as in `<<stereotype>>`. In UML, anything can be tagged with a stereotype. In the previous diagram, the Item class is stereotyped as a Configurable Entity. This means that it would have the qualities of one as described in the section Entity.

Attributes are listed in the second compartment. In UML the name of the attribute is specified first followed by its type. The name and the type are separated by a colon. It is notable because it is different from the Java language. It works well for object oriented modeling which is generally an iterative process. Often times a designer will list the attributes of class without specifying types the first time through. The same techinique holds true when specifying the arguments to a method. Note that as already mentioned, attributes can be decorated with a stereotype. The stereotype precedes the attribute and is embedded in guillemets as before.

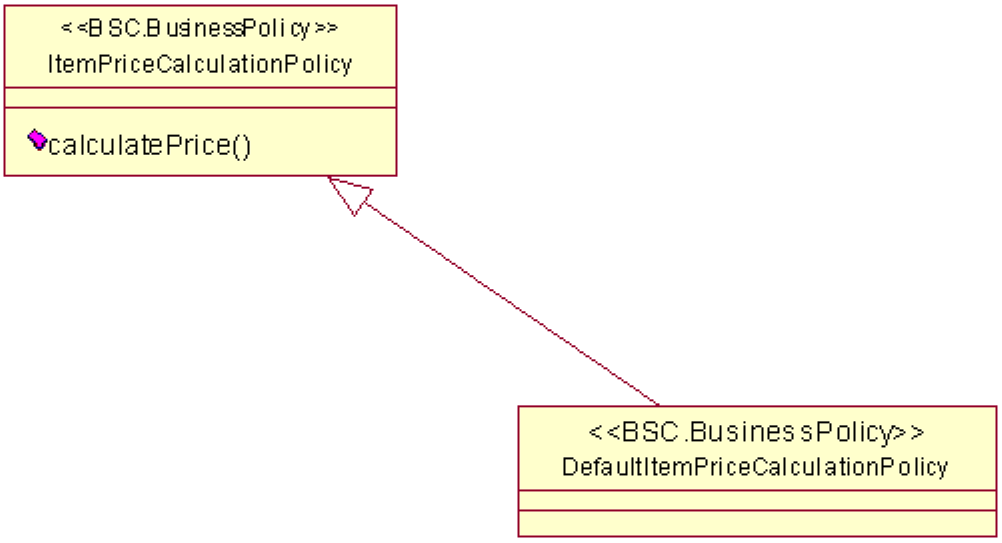
The third and final compartment lists the methods. The return type is listed after the closing parentheses and is separated from the class definition with a colon. Often the display of the parameters and the return value are supressed on the diagram because they consume a great deal of space.

When specifying attributes and methods it is possible in the UML to indicate whether or not they are private, protected, or public. The "tilted brick" icon to the left will have slight variations depending on this.

Inheritance

Figure 3-4 focuses on the UML notation for inheritance.

Figure 3-4 UML Notation for Inheritance

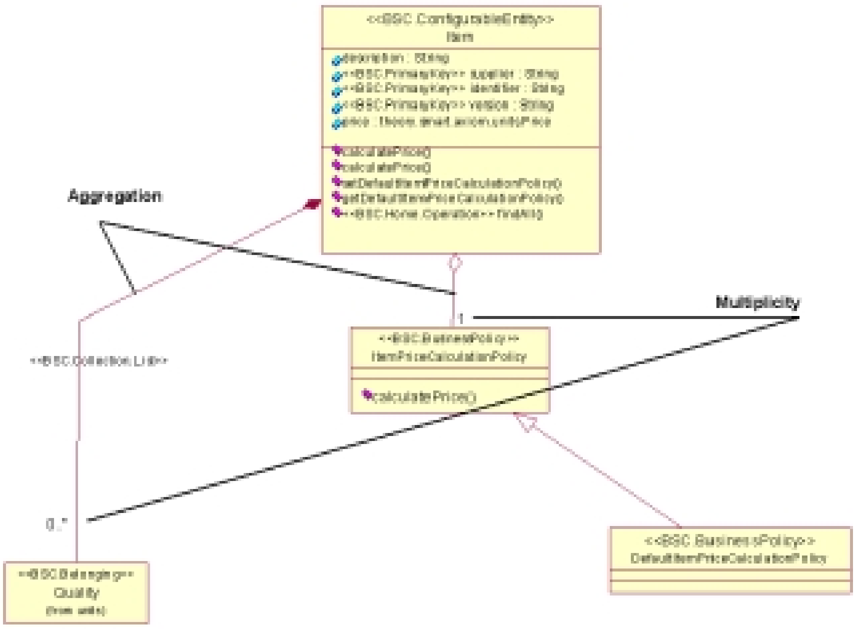


In a UML diagram, inheritance is depicted an unfilled arrow that points from the subclass towards its parent. In this case the `ItemPriceCalculationPolicy` will have a `calculatePrice` method through inheritance. The subclass will share all of the properties and attributes of its parent.

Aggregation and Multiplicity

Figure 3-5 illustrates the UML notations for concepts called aggregation and multiplicity.

Figure 3-5 UML Notation for Aggregation and Multiplicity



<<BSC ConfigurableEntity>>
Item
description : String
<<BSC PrimaryKey>> supplier : String
<<BSC PrimaryKey>> identifier : String
<<BSC PrimaryKey>> version : String
price : theorem.smart.axiom.units.Price
calculatePrice()
calculatePrice()
setDefaultItemPriceCalculationPolicy()
getDefaultItemPriceCalculationPolicy()
<<BSC Home Operation>> findAll()
Aggregation
Multiplicity
1
<<BSC BusinessPolicy>>
ItemPriceCalculationPolicy
calculatePrice()
<<BSC Collection List>>
0..*
<<BSC Belonging>>
Quality
(from units)
<<BSC BusinessPolicy>>
DefaultItemPriceCalculationPolicy

Aggregation is used to describe a containment relationship between classes. This is an alternative to simply defining an attribute with the type of the class. In UML this means that the contained object shares a life cycle with the containing object. That is, the containing object holds the only reference to it and is responsible for removing the object upon when it, itself, is removed.

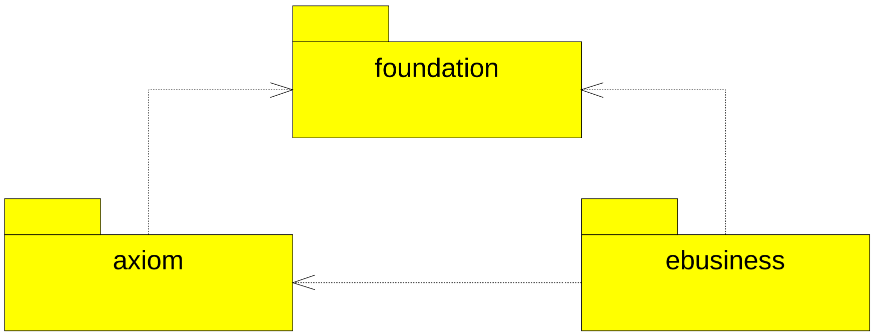
Aggregation is depicted in UML with a line that extends from the containing to the contained item. The line begins with an oblong diamond that specifies a category of containment. A hollow diamond is used to show that the object is being contained by reference. A solid diamond specifies that the object is contained by value.

It is also possible to specify a **multiplicity** for the object being contained. Options are 1 (one to one), 0..1 (optionally null for references), or 0..* (zero to many). As with all other elements of the UML it is possible to stereotype the relationship. It is also possible to name an aggregation, although there is no example of this in the above diagram.

Packages

Figure 3-6 illustrates the UML notation for the relationship between packages.

Figure 3-6 UML Notation for Packages



Packages are used to group classes and other packages in to a hierarchy. Each package will contains classes and/or other packages. When the classes of one package use the classes of another this is depicted as a dotted line with an arrow in the appropriate directions. This same "uses" notation can be applied to classes as well.

WLCS Smart Generator Rules: Factors that Influence the Generated Java Files

This section explains how the WLCS Smart Generator transforms a UML diagram into EJB components. We will describe the Java code that will be generated as the result of making specific notations in a UML diagram.

Classes

Only classes in the model that are stereotyped as `eBusiness Smart Component (eBSC)` will result in the generation of Java classes. There is not a one-to-one mapping between each class in the UML model and Java. In particular, all eBSCs are implemented using the Abstract Factory pattern. This means that there will be at least one interface and two Java classes generated for each eBSC that is modeled in UML. In addition, each Entity eBSC will have an associated Primary Key and Value class that is generated as well.

The following table describes the mapping of classes based on the class stereotype.

Stereotype	**Class Only**	**Interface**	**Home**	**Impl**	**PK**	**Value**
BSC Belonging		[x]	[x]	[x]		
BSC Session		[x]	[x]	[x]		
BSC Entity		[x]	[x]	[x]	[x]	[x]
BSC Workflow	[x]					
BSC Business Policy	[x]					

The naming convention for the generated classes is a follows:

- Class Only

  The class will implement the respective interface and will be given the same name as the class in the model.

- Interface

  The interface will be given the same name as the class in the model.

- Home

  The Home interface/class will be the class name with the word “Home” appended. For example, ItemHome. For the Session and Entity objects this will be an interface that is used by the EJB Compiler to generate the home implementation.

- Implementation

The Implementation class will be the class name with the letters "Impl" appended. For example, ItemImpl. **You will add your business logic to each generated *Impl.java file**.

- Primary Key

  The Primary Key class will be the class name with the letters "Pk" appended. for example, ItemPk.

- Value

  The Value class will be the class name with the letters "Value" appended. For example, ItemValue.

Primary Key and Value

For Entity Components there are two special classes that are generated. The Primary Key class is a Java class with public members for each of the attributes that are stereotyped as `<<BSC.PrimaryKey>>`. The primary key class is used by the create and `findByPrimaryKey` methods of the generated home class.

Listing 3-2 demonstrates the usage of the `PrimaryKey` class.

Listing 3-2 Use of the PrimaryKey Class

```
public class OrderPk extends SmartKey implements java.io.Serializable
{
public String key;

public OrderPk(
 {
       super();
 }

   … more code here

}

public interface OrderHome extends SmartEJBHome
{
       public Order create(theory.smart.ebusiness.order.OrderPk orderPk )
              throws CreateException, RemoteException;
       Order findByPrimaryKey(theory.smart.ebusiness.order.OrderPk orderPk)
              throws RemoteException, FinderException;
}
```

The `Value` class is a Java class with public members for each of the attributes of the associated Entity. This includes attributes that are specified through aggregation. This class is used by the generated Value accessor methods. The purpose of these method is to simplify the retrieval of multiple attributes and reduce the overhead associated with remote method invocation.

Listing 3-3 shows the use of Value objects in the code generated by the WLCS Smart Generator. Use caution when you use the `setByValue` method because there is no built-in Entity locking. When using the `setByValue` on an Entity object it is important to realize that the attributes which are members of the primary key cannot and will not be updated. This is because as part of the identity of the Entity they are immutable.

Listing 3-3 Use of Value Objects

```
public class ItemValue extends SmartValue
{
       public String version;
       public String identifier;
       public String supplier;
       public String description;
       public theory.smart.axiom.units.Price price;
      public LinkedList qualities;

 protected ItemValue()
 {
   super();
 }
}

public interface Item extends ConfigurableEntity
{
public ItemValue getItemByValue() throws RemoteException;
public void setItemByValue(ItemValue value) throws RemoteException;
//...
}
```

Interfaces, Homes, and Implementations

The Abstract Factory pattern requires that objects be accessed only through their interfaces and that the classes that implement those interfaces be acquired only through a factory class. The factory class in the case of EJB is referred to as a Home. This has slightly different implications for EJB components and Belongings.

When dealing with Session and Entity objects, run the EJB compiler to create the appropriate proxies stubs and skeletons. At deployment time the application server will be responsible for registering the home interface with the Java Naming and Directory Interface(JNDI) so that users of the EJBs will be able to create and find them.

For Belongings, the home, interface, and implementation will reside wherever they are instantiated. Belongings are always passed by value. When a belonging is used as the parameter to a method of a Session or Entity it will be serialized and then reinstantiated on the server. To make this happen the Java class associated with the belonging must be available in the class path on the server.

The deployment implication is that the release of these classes must be coordinated between the client and the server.

The Home interface is where finder methods reside. A finder method is one that locates one or more preexisting entities. The WLCS Smart Generator will automatically generate a finder method based on the primaryKey, as shown in Listing 3-2.

It is often necessary to create finders that search for entities based on the values of some other attributes. Adding an operation to the main class and stereotyping it as `<<BSC.Home.Operation>>` will accomplish this. The resulting method will be generated into the associated home class.

Attributes and Accessor Methods

For each attribute that is specified in the WLCS Components model a pair of accessor methods are generated. The `get<AttributeName>` method will retrieve the value of the attribute from the remote object and return it to the client. The `set<AttributeName>` method will pass the attribute to the remote object where it will be updated. In the case of an Entity the entire object will be marked as dirty such that the application server will know that the changed values need to be persisted in the database. (The “isDirty” attribute is specific to BEA WebLogic Server.) This is true of Sessions to a lesser degree in that many application servers perform a serialization of Session beans for the purpose of optimizing the caching of Sessions.

The following listing shows the generated accessors.

Listing 3-4 Generated Accessors

```
public interface Item extends ConfigurableEntity
{
  public String getSupplier() throws RemoteException;
  public String getIdentifier() throws RemoteException;
  public String getVersion() throws RemoteException;

  public String getDescription() throws RemoteException;
  public void setDescription(String description) throws
RemoteException;
  public theory.smart.axiom.units.Price getPrice() throws
RemoteException;
  public void setPrice(theory.smart.axiom.units.Price price) throws
RemoteException;
}

public class ItemImpl extends ConfigurableEntityImpl
{
  public String version;
  public String identifier;
  public String supplier;

  public String description;
  public theory.smart.axiom.units.Price price;

 public String getDescription()
 {
   return (String) description;
 }
 public void setDescription(String description)
 {
   isDirty = true;
   this.description = (String) description;
 }
 public String getSupplier()
 {
   return supplier;
 }
 public String getIdentifier()
 {
   return identifier;
 }
 public String getVersion()
 {
   return version;
 }
 public theory.smart.axiom.units.Price getPrice()
 {
   return (theory.smart.axiom.units.Price) price.value();
 }
 public void setPrice(theory.smart.axiom.units.Price price)
 {
   isDirty = true;
   this.price = (theory.smart.axiom.units.Price) price.value();
 }
}
```

One omission in the previous sample code is that there are no methods for *setting* attributes that are stereotyped as part of the PrimaryKey for an entity. This is because those attributes are part of the identity of the object and as such they are immutable, cannot be changed.

Accessors are generated for belongings as well. The call to an accessor of a belonging is a direct call to the implementation object.

All of the attributes must be serializable. This also ensures that they can be persisted.

Rules for Aggregation Notations in the UML Diagram

Aggregation allows for the definition of an attribute of a class by drawing a line between it and another class which will be a included as a member. The following rules describe the allowable notations:

- A Belonging may only be contained by value (solid diamond).
- An Entity may only be contained by reference (hollow diamond). In such cases the attribute is stored as a SmartHandle.
- A Workflow is similar to a Belonging and is always contained by value. A Workflow is persisted using a WorkflowContext.
- A BusinessPolicy is similar to a Belonging and is always contained by value. The accessors for the BusinessPolicy must be explicitly specified as business methods.
- If an aggregation is named, that name will be used by the WLCS Smart Generator when it creates the accessors for that attribute. This is necessary so that multiple relationships to the same class can be modeled.

- If an aggregation is not named, the accessors will be created by the WLCS Smart Generator based on the name of the class that is being contained.
- Multiplicity may be defined as described in the section on "Collections."

Collections

One of the most challenging issues when designing distributed object systems is implementing one-to-many relationships between objects. When modeling eBSC in UML such relationships are described by stereotyping either an attribute or an aggregation with a multiplicity of zero or more.

When an aggregation relationship is stereotyped as a particular collection type, the internal attribute reflects that choice and the appropriate accessors are generated. The table below describes the options, a brief description of their usage, and the Java 2 SDK class upon which the implementation is based. See the Java 2 SDK documentation at http://java.sun.com/products/jdk/1.2/docs/index.html for more details about the features of each collection type.

Table 3-1 Collection Stereotype Mappings

Stereotype Name	Purpose	Collection Type
BSC.Collection.Set	A collection that contains no duplicates and in which there is no implied ordering.	java.util.Collection.TreeSet
BSC.Collection.Array	An ordered collection that is stored as contiguous elements. This allows for optimal random access so that operations like re-sorting can be executed quickly.	java.util.Collection.ArrayList
BSC.Collection.List	An ordered list that optimizes insertions at the ends.	java.util.Collection.LinkedList

Table 3-1 Collection Stereotype Mappings

Stereotype Name	Purpose	Collection Type
BSC.Collection.Map	A collection that is indexed by string and optimized for quick lookup. Iteration will be in ascending order according to the natural sort method.	java.util.Collection.TreeMap

The accessors for collections are generated for each stereotype as described in Table 3-2. The table uses a shorthand syntax to convey which accessors are generated when a given stereotype is chosen. The token <Attribute> is replaced by the name of the attribute or aggregation as specified in the model. In the case of methods that accept or return a collection, the type is stereotype specific as defined in Table 3-1. The details of the parameters and return values are implied so that the table itself can be concise. While there is no true inheritance relationship, it should be considered that Set serves as a basis for Array, which is a basis for List. Map is different in that it supports lookup by key.

In the case where an aggregation to an entity is specified by value, an additional group of methods is generated. These methods simplify the maintenance of the ownership relationship by ensuring that the underlying Entity is removed from its home in conjunction with the removal of its reference from the list. The converse, add by value, is not supported because it would require that the containing entity be aware of the home of the entity to be added.

The Set provides methods for adding and removing attributes from a collection, it provides a "bag" type collection mechanism. The Array provides random access methods and is optimized for random access by integral position, for this reason it is especially useful when multiple sort orders are required. The List provides random access but is optimized for adding at the ends; this makes it good candidate for use when stacks or queues are needed.

The Map makes it possible to index a collection by a String.

Table 3-2 Generated Accessors by Stereotype

	Accessors	Iterator Methods	Entity by Value
Set	add<Attribute> add<Attributes> (<CollectionType>) contains<Attribute> is<Attributes>Empty removeAll<Attributes> get<Attributes>() :<CollectionType>	create<Attribute>Iterator hasNext<Attribute> getNext<Attribute> remove<Attribute>At	remove<Attribute>ByValue remove<Attributes>ByValueAt removeAll<Attribute>ByValue
Array	**<All of Set > +** add<Attribute>(int position,…) set<Attribute>(int position, …) get<Attribute>(int position) get<Attributes>(int from, int to) remove<Attribute>(int position) indexOf<Attribute> lastIndexOf<Attribute>	**<All of Set> +** add<Attribute>At set<Attribute>At getNext<Attribute> getPrevious<Attribute> getNext<Attribute>Index getPrevious<Attribute> Index	**<All of Set> +** remove<Attribute>ByValue(int)
List	**<All of Array> +** addFirst<Attribute> addLast<Attribute> getFirst<Attribute> getLast<Attribute> removeFirst<Attribute> removeLast<Attribute>	**<All of Array>**	**<All of Array> +** removeFirst<Attribute>ByValue removeLast<Attribute>ByValue
Map	put<Attribute>(String key) put<Attributes>(TreeMap) get<Attribute>ByKey get<Attributes>(String) contains<Attribute>Key contains<Attribute>Value remove<Attribute>ByKey removeAll<Attributes>	create<Attribute>Iterator hasNext<Attribute> getNext<Attribute> remove<Attribute>At	**<Accessors are defined using WithKey instead of ByKey>** put<Attribute>ByValue remove<Attribute> ByValueWithKey removeAll<Attributes>ByValue

Design Decisions

Now that we have covered the basics, let's discuss some of the choices that you will need to make during the design process. While it would be nice to allow the modeler to design without consideration for implementation details, the reality is that truly good designs take into account deployment-time issues.

Use of Entities versus Sessions

One of the most common issues when modeling EJB is related to legacy systems. These systems very typically provide an API or message-based protocol to allow external systems to access their functionality. The tendency in such cases is to simply model access to such systems as a Session component where each function in the API is a method of the Session bean. In the case of legacy systems that store complex business data and relationships, this is a mistake. In such cases it is best to model the internal objects as Entities where appropriate. This will provide for a more understandable system definition that takes advantage of the important caching and transaction services features of the EJB specification.

Implementing Business Logic in an Entity

In general, Session beans provide a sensible mechanism for implementing "workflow" related business logic. Workflow in this case is logic that coordinates the usage of any number of Entity beans. This has the performance-improving effect of reducing the network overhead associated with executing extended operations remotely. In the case where an Entity bean needs to perform complex business logic on classes that it references, it is best to implement that logic as a method of the Entity bean. This places the business logic where it belongs, with the data that it is manipulating.

Modeling from a Message Specification

There is a strong trend in the industry to translate message specifications, particularly XML DTDs, directly into business objects. While this may be convenient, it may not result in a clean description of the business objects. This is similar to attempt to model a system based solely on the API. A better approach is to consider a message specification as providing insight into a single users perspective of the system. One approach is to consider the messages as method invocations to one or more underlying business objects. The contents of the message can then be modeled as attributes of various underlying objects.

Changing Method Signatures

If you change the signature of a method it will not be properly managed by the WLCS Smart Generator. This happens because the round trip engineering feature works by matching the exact signature of the method and the parameters. If a generated method is no longer present in the model it will simply be deleted, along with the associated implementation. To avoid this situation, you must make matching changes in the model and the source code. As a consequence it is extremely important to consider the parameters to methods up front so as to avoid this problem.

CHAPTER

4 Deploying Your Application

This chapter explains how to deploy your application. The following topics are presented:

- Defining the Persistence Type for your Deployment
- Using Bean-Managed Persistence
- The Oracle Reference Implementations
- Deployment Sets Overview
- Deploying on Windows NT
- Deploying on Solaris
- Considerations in Bean-Managed Persistence

Defining the Persistence Type for your Deployment

Before you start the WebLogic application server, you must do the following:

- Set the environment variable `DEPLOYMENT_SET` to either `BMP` for Bean-Managed Persistence, or `CMP` for container-managed persistence.

- In the directory where you installed WLCS, rename the appropriate `weblogic-`**`XXX`**`.properties` file to `weblogic.properties`, where **`XXX`** is either **`BMP`** or **`CMP`**. The renamed file should reside in the top-level WLCS installed directory, such as `c:\WebLogicCommerce`.
- In the directory where you installed WLCS, rename the `weblogiccommerce-`**`XXX`**`.properties` file to `weblogiccommerce.properties`, where **`XXX`** is either **`BMP`** or **`CMP`**. The renamed file should reside in the top-level WLCS installed directory, such as `c:\WebLogicCommerce`.

The WLCS EJB source code is independent of the persistence method, whether it is container-managed or any implementation of Bean-Managed Persistence.

The source code that uses JDBC instructions to persist Entity Beans is generated according to one of the BEA reference implementations of Bean-Managed Persistence. If you use the reference data model, the source code may be used as generated. The more likely case is that the JDBC source code may be used as templates to persist EJBs to a legacy data model.

One of the advantages of the WLCS approach is that business logic and other core EJB source code is independent of the persistence implementation. This isolates business application development from database development and schema changes.

Using Bean-Managed Persistence

You can deploy the WebLogic Commerce Server examples using Bean-Managed Persistence.

This section discusses the following topics:

- Introduction
- The Oracle Reference Implementations
- Deployment Sets Overview
- Deploying on Windows NT
- Deploying on Solaris

Introduction

You can map WLCS to any database available through JDBC. BEA provides **a reference implementation** (*deployment set*) for Bean-Managed Persistence. This is available as a deployment options in the WebLogic Commerce Server release.

Here, we describe how to deploy the WebLogic Commerce Server software in the following environments:

- Deploying on Windows NT
- Deploying on Solaris

The example Oracle deployment set allows you to map the **MyBuyBeans.com** components to Oracle 8.0.5 and Oracle 8.1.5.

The WLCS Smart Generator has basic object/relational mapping features that generate up to 100% of the mapping from beans to a relational model using serialization. For complex databases the generated code serves as a starting point for reliable and scalable Bean-Managed Persistence that you can modify and fine-tune.

Note: For specific directions on deploying the **MyBuyBeans.com** example portal on Oracle, see the **Installation Guide**.

The Oracle Reference Implementations

The Oracle reference implementation uses Oracle's object-relational features including database object types and nested tables. WebLogic Commerce Server are stored with a maximum transparency. You can query on every field in the example BuyBeans object model. This reference implementation is made available to help gauge the effort and complexity of more-detailed object-relational persistence.

If you have any comments, questions, or need help with a WebLogic-Oracle 805 deployment, please contact support@theorycenter.com.

Oracle OCI Driver. The WebLogic-Oracle 805 deployment uses the Oracle Thin JDBC driver. If you want to use the Oracle OCI driver instead, you do not have to re-run the EJB compiler. You need to modify the `weblogic.properties` file to use the OCI Driver instead of the Thin driver, then re-start the server. All our beans use TRANSACTION_READ_COMMITED as the isolation-level, but if you change the isolation level then you need to re-run the EJB compiler and redeploy all the beans.

Additional Requirements

- MyBuyBeans.com examples successfully deployed using default container-managed persistence on the Windows NT or the Solaris operating systems.
- Oracle Thin driver for JDBC. These drivers can be downloaded from the Oracle Web site at http://www.oracle.com/.

Deployment Sets Overview

Deployment Sets give you the freedom to develop your business application independently from the application server or database vendors.

This separates business logic from the development environment and gives you the freedom to choose to develop in one environment and deploy in another. Each Deployment Set pertains to a particular combination of an application server and database.

There is a deployment set available for each of BEA's certified implementations on an application server and database. Deployment sets are available for:

- **WebLogic**
  - Using Container-Managed Persistence to a Cloudscape database
  - Using Bean-Managed Persistence to an Oracle 805 Database

The following table illustrates the matrix of servers and databases.

Table 4-1 Servers and Databases

Databases	**Application Servers**	
Databases	WebLogic	NAS

Table 4-1 Servers and Databases

Databases	**Application Servers**	
Cloudscape	CMP	n/a
Oracle 8.0.5	BMP	under development

Deploying on Windows NT

This reference example assumes you have installed the WebLogic Commerce Server software under `c:\WebLogicCommerce`.

1. Edit `c:\WebLogicCommerce\bin\win32\sethome.bat`
   - Change DEPLOYMENT_SET to BMP
   - Set ORACLE_HOME to the Oracle installation directory.
2. Create the *BuyBeans schema* in your Oracle 8.0.5 or higher database.
   - Use `c:\WebLogicCommerce\db\oracle\create-p13n-oracle-nt.sql`
3. Change the **`weblogic.properties`** file.
   - Copy the file `c:\WebLogicCommerce\weblogic-bmp.properties` to `c:\WebLogicCommerce\weblogic.properties`
4. Edit the file `c:\WebLogicCommerce\weblogic.properties`.
   - Search for the properties named `weblogic.jdbc.connectionPool.CommercePool` These are examples for using either the OCI or Thin driver. Modify the connection pools to use your database and password. See http://www.weblogic.com/docs/classdocs/API_jdbct3.html#connpools for further help.
5. **Start the WebLogic server** using `c:\WebLogicCommerce\StartCommerce.bat`.
6. Load the **database**

- Run the `c:\WebLogicCommerce\bin\win32\DataLoader.bat`

7. **Start your web browser**. Enter the URL with the name and port of the machine where you deployed the WebLogic server. You will have to register a new user with **username** "*cool*" and **password** "*bean*".
8. You are now using BEA's Bean-Managed Persistence!

Deploying on Solaris

This example assumes you have installed the WebLogic Commerce Server software in `/WebLogicCommerce`.

1. Edit `/WebLogicCommerce/bin/solaris2/sethome.sh`
   - Change DEPLOYMENT_SET to BMP
   - Set ORACLE_HOME to the Oracle installation directory.
2. **Create the BuyBeans schema** in your Oracle 8.0.5 or higher database.
   - Use `/WebLogicCommerce/db/oracle/misc/create-p13n-oracle-unix.sql`
3. Change the **`weblogic.properties`** file.
   - Copy the file `/WebLogicCommerce/weblogic-bmp.properties` to `/WebLogicCommerce/weblogic.properties`
4. Edit the file `/WebLogicCommerce/weblogic.properties`.
   - Search for the properties named `weblogic.jdbc.connectionPool.CommercePool` These are examples for using either the OCI or Thin driver. Modify the connection pools to use your database and password. See http://www.weblogic.com/docs/classdocs/API_jdbct3.html#connpools for further help.
5. **Start the WebLogic server** using `/WebLogicCommerce/StartCommerce.sh`.
6. Load the **database**
   - Run the `/WebLogicCommerce/bin/solaris2/DataLoader.sh`

7. **Start your web browser**. Enter the url with the name and port of the machine where you deployed the WebLogic server. You will have to register a new user with **username** "*cool*" and **password** "*bean*".

8. You are now using BEA's Bean-Managed Persistence.

Considerations in Bean-Managed Persistence

The section discusses the design and implementation of Bean-Managed Persistence.

Container-Managed Persistence Versus Bean-Managed Persistence

In Container-Managed Persistence (CMP), you use the deployment descriptor to tell the container which attributes of the entity bean to persist. Flexibility is therefore governed by the vendors of the container and database.

BMP gives you explicit control of the management of a bean instance's state.

The advantages of using Bean-Managed persistence may include:

- Performance advantages
- The ability to express complex relationships among data
- An interface to complex legacy SQL databases via JDBC
- An interface to other enterprise data sources such as CICS and MQ-Series

Some disadvantages for developers when you use BMP:

- You must explicitly code the `ejbCreate(), ejbLoad()`, and other EJB callback methods.
- You must explicitly code the finders methods in the home implementation.

- You must understand, develop, and maintain a map between bean and database.
- The dependency risk of commiting a bean’s abstract business logic to a specific database type and structure.

Considerations when Persisting an EJB

The data model and access mechanisms have a strong impact on persistence logic. In particular, an entity bean’s primary key, attributes, and contained classes must be considered. Particular attention should be given to other entity beans that are contained by value or reference.

The principles discussed in this section use a framework of mapping EJB Objects to a relational data model using JDBC. These principles may be generalized by the reader and applied to other mechanisms for modeling and accessing enterprise data.

Complexity of the Mapping Implementation

Consider the factors driving the map between EJB Objects and data storage.

A fixed database schema and a fixed object model may increase the complexity of the mapping implementation. Flexibility in either the database schema or the object model may correspondingly decrease this complexity.

Serializing objects or parts of objects may help decrease complexity of data mapping. The tradeoff is that the serialized data is “opaque” and is not easily accessed through third party tools such as report writers.

Dissecting and Persisting an Enterprise Java Bean

Primary Key

The Enterprise JavaBeans Specification requires that each Entity Bean has a class that represent attributes that uniquely identify an instance of that bean. The implication is that these attributes are used in primary key columns or foreign key columns in a relational database. These attributes cannot be serialized because they must be queried against.

Singleton Attributes

Attributes that have a 1:1 relationship with their class are easily mapped to columns or sets of columns in a relation that characterizes the bean.

Primitive Data Types

Attributes that correspond to JDBC primitive types, (for example: `java.sql.Types.LONGVARCHAR`, `java.sql.types.INTEGER`) easily map to columns in a relational table. If these attributes are serialized they cannot be easily be used in SQL reports or queries. Serialization of these attributes may impact the complexity or performance of the object-relational map. References to primitive data types are possible but discouraged.

Compound Data Types

Attributes that are java objects (not EJB's) are easily decomposed into columns that correspond to the JDBC primitive types. When these attributes are contained by value they can go in the table (or relation) where the EJB object is persisted. Care must be taken to ensure they are deleted from the database when the containing EJB object is deleted.

A reference to an attribute can be established with a foreign key to a relation that stores the actual attribute.

It is up to the application designers to determine application specific standards for creating and maintaining these keys. Application logic must ensure dangling references do not occur when the contained object is removed.

Designers should also consider if the referenced data should have a foreign key back to the containing EJB.

The complexity of this issue increases if many EJBs can reference the data.

Entity Beans

Entity Beans can contain other Entity Beans by value or reference. The fields in an entity bean's primary key class comprise a foreign key in the containing bean's relation. In addition, the containing bean needs access to the contained bean's primary key class and home class. This allows the containing bean to call findByPrimaryKey() to locate the contained bean.

Entity beans should be stored in their own relation, regardless of their containment by value or reference.

Application logic must be developed so beans contained by value are not orphaned, and to avoid dangling references to beans contained by reference. Designers should also consider if the target entity bean should have a foreign key back to the containing EJB. The complexity of this issue increases if many EJB's can reference the data.

Collections: Attributes Contained in a Many-to-One Relationship

Entity beans can contain other Java objects in many-to-one relationship. These collections can be stored in a separate table or a nested table if the DBMS supports this. The separate table needs a foreign key to join with the containing entity bean.

Application logic needs to address issues raised by Java's different collection classes. Many of these classes do not have a key to access the data. Some of these classes support ordering which must the persistence logic must manage.

Performance issues arise when persisting collections that have no primary key. When one member of the collection changes, the entire collection must be deleted and updated into persistent storage.

The CRUD operations (create, refresh, update, delete) must be done atomically on all the changing attributes of an Entity bean. This raises issues of transactional integrity and increased resources to support large transactions; for example: transaction logs, open cursors.

Collections can be serialized into a single column if the Java collection class implements the java.io.Serializable interface and the DBMS supports binary data types. This may simplify the persistence logic. In this case, whenever any member of the collection changes, the entire collection must be deleted and updated in the database.

Application logic needs to address orphan and dangling reference issues for objects contained by value and reference. Serialization of collections contained by reference will increase application logic complexity.

Primitive Data Types

Collections of primitive data types raise few issues that have not been previously discussed. Collections of primitive data types by reference are an absurdity, because the only key can be the value of each element in the collection.

Compound Data Types

Collections of compound data types that are contained by value can be serialized or stored in a separate table from the Entity Bean. If they are stored in a separate table, they need a foreign key to join with the containing entity bean.

Collections of compound data types by reference are possible if there is some unique key that identifies each element in the collection. The collection of keys would be serialized or stored in a separate table from the containing Entity Bean. If the keys are stored in a separate table, this table also needs a foreign key to the containing Entity Bean(s). This makes it possible to avoid dangling references when the object is deleted.

The complexity of the object-relational mapping increases when the objects in a collection have collections themselves. Many joins may be required to update and refresh the data. Serialization may reduce this complexity.

Entity Beans

Collections of Entity Beans are simplified because of requirement that each Entity Bean have a primary key class. The collection of primary keys needs to be persisted in serialized or table form.

As previously stated, Entity Beans should be stored in their own relation. The containing class needs to have access to the contained bean's home class and primary key class to invoke findByPrimaryKey() to locate the contained bean.

When entity beans are contained by value, and persisted in a table, this table may not need a foreign key to the containing bean. When they are contained by reference, the table should have a foreign key to the containing Entity Bean(s). This makes it possible to avoid dangling references when the contained entity bean.

CHAPTER

5 Component Examples

The section contains the following topics:

How to Build and Run the Examples

To become better acquainted with the workings of an EJB-based application built using our WebLogic Commerce Server (WLCS), follow these examples.

Note: To build and run any of these examples, you must have the following in your CLASSPATH:

- `theory-smart-generator.jar`, `theory-axiom-foundation.jar`, `theory-ebusiness.jar`, `theory-examples.jar` Each of these jar files can be found in the lib directory under the WebLogic Commerce Server installation directory.
- Application Server classes (default classpath required by WebLogic Server)

The fastest way to run any of the examples is by using the scripts provided in ...`\bin\win32\*.bat or` ..`\bin\solaris2\*.sh` (Found under the WLCS installation directory)

Foundation and Axiom

This example demonstrates the core technology: *Belongings*, *Entity Beans*, *Collections* and *RemoteIterators*.

Package examples.axiom

The Axiom example shows the use of WLCS Axiom package of WebLogic Commerce Server.

Table 5-1 Axiom Package Summary

Class	Description
AxiomExample	Shows how to use the WLCS components Axiom package.

Package examples.axiom Description

The Axiom example shows the use of the WLCS components Axiom package.

This example demonstrates:

- The Abstract Factory Pattern
- How to use EJB WebLogic Commerce Server
- Usage of Belonging WebLogic Commerce Server
- Remote Iterators

Belongings and EJBs

The Axiom package contains light weight components known as belongings, as well as Entity and Session EJB components. Belongings can be aggregated to other components by value. EJBs are used alone or aggregated to other components by reference or value.

The Abstract Factory Pattern

All WebLogic Commerce Server use the abstract factory pattern. The principle is very simple: Don't use `new()` to create an object, instead, you use `Home.create()`. The Abstract factory pattern is implemented as the "Home" for EJBs and as a Java class with static methods for Belongings.

Axiom Example

The example application performs the following steps:

1. Find or create or a Customer component
2. Create belongings
3. Add belongings to the Customer
4. Use a Remote Iterator to iterate through the belongings
5. Remove the Belongings

To get the most out of this example, first read through `AxiomExample.java on our web site`. Then you can build it and run it.

Note: Be sure to set your CLASSPATH as described in "How to Build and Run the Examples" on page 5-2.

Workflow

Workfow, *eBusinessSession*, and *eBusinessSessionManager* components. The *Workflow* maintains state for the session and guides the user through the process.

Package examples.workflow

Shows the use of WLCS Workflow components.

This example demonstrates:

- How to use WLCS components.
- Usage of the Workflow component.
- Usage of `eBusinessSession` and `eBusinessSessionManager`.

Table 5-2 Workflow Package Summary

Class	Description
WorkflowExample	Workflow example.

Package examples.workflow Description

Shows the use of the WLCS Workflow components.

This example demonstrates:

- How to use WLCS components.
- Usage of the Workflow component.
- Usage of `eBusinessSession` and `eBusinessSessionManager`.

Workflow

This example shows the use of a WLCS component that has a workflow associated to it. The workflow states and transitions are modeled with Rational Rose. For this examples, we'll use the `EBusinessSession` Component. This component has a workflow that guides it through the different stages of an online e-business session. If you look at the Rose model file for the ebusiness.session package, you will find that `EBusinessSessionWorkflow` has a state diagram associated to it. The workflow logic can be implemented in any way you want; however, WLCS provides a reference implementation. For the reference implementation, for each component with the BSC.Workflow stereotype, all the states and transitions in the Rose model are generated into a complete state machine by the SmartGenerator, so you can use it immediately, without any hand-coding of the workflow states or transitions.

In this example, we also use the EBusinessSessionManager and show how a "manager" session bean can simplify the usage of an entity bean

Workflow Example

The workflow example application performs the following steps:

1. Create a Guest Session component using the `EBusinessSessionManager`.
2. Try options such as `enroll`, `cancellEnrollment`, `becomeGuest`, and `disableAuthentication`. (You can find these transitions in the `EBusinessSessionWorkflow` state diagram)
3. Register as a new or Login as an existing Customer
4. Perform more options (`authenticate`, and `disableAuthentication`)

To get the most out of this example, first read through `WorkflowExample.java on` our web site then you can build it and run it.

Note: Be sure to set your CLASSPATH as described in “How to Build and Run the Examples” on page 5-2.

BusinessPolicy

Pluggable Methods, Strategy Pattern, or *Individual Instance Method.* No matter what you call it, it is a powerful design tool. This example demonstrates how *ConfigurableEntity* beans and **BusinessPolicy** work together to create very flexible solutions.

Package examples.businesspolicy

BusinessPolicy Example shows the use of WLCS WebLogic BusinessPolicy Components.

Table 5-3 BusinessPolicy Package Summary

Class	Description
AprilFoolsDiscountPolicy	This class is a custom item pricing calculation policy.
BusinessPolicyExample	This example demonstrates the concept of "Pluggable Methods", better known as policies.
SeniorCitizenDiscountPolicy	This class is a custom item pricing calculation policy.

Package examples.businesspolicy Description

The BusinessPolicy example shows the use of the WLCS BusinessPolicy components.

This example demonstrates:

- How to use the WLCS components
- How to add a default Policy to an Item using the WLCS BusinessPolicy components
- How to use a non-default policy to change the price of an item

ItemPriceCalculationPolicy and BusinessPolicy

This example shows the use of `theory.smart.ebusiness.item.ItemPriceCalculationPolicy` which is an extension to `theory.smart.foundation.BusinessPolicy`. A `BusinessPolicy` consists of rules and regulations, specific to your business. These rules can be encapsulated into a component and then added to a Component such as an Item.

This example demonstrates the concept of "Pluggable Methods", better known as policies. When you create your components, you will realize that many times you want to alter the component behaviour based on external conditions that you can not evaluate at development time. Reusability, extensibility and rapid development and enhancement are typical problems that can be solved using policies. `BusinessPolicy` is the WLCS implementation of the Policy and Strategy design patterns. Using this concepts allows you to replace the default policy at runtime. The policy is stored as a property for the item.

In this example we will use an item component. The item component has a pricing policy. The item's price is calculated based on a given quantity and the pricing policy. You can replace the pricing policy to alter the way the price is calculated for the item. This means that you can modify the behaviour of the item by plugging in a method that calculates the price the way you want If you do not provide a pricing policy, a default policy will be used. The example creates an item. It then sets the `SeniorCitizenDiscountPolicy` as the default pricing policy for the item. Then the item's price is calculated using the default policy. Finally, it modifies the item's quantity and once again, and calculates the price by using the `AprilFoolsDiscountPolicy` policy. To better understand this example, first go through the Axiom example first.

The concept is also used in our BuyBeans.com online store where different pricing policies of BuyBeans are used for calculating the prices of `examples.buybeans.item.BeanieBaby`, `examples.buybeans.item.CoffeeBean`, and `examples.buybeans.item.JellyBean` components. They use `BeanieBabyPricePolicy`, `CoffeeBeanPricePolicy`, and `JellyBeanPricePolicy` respectively.

BusinessPolicy Example

The BusinessPolicy example application performs the following steps:

1. Find or create or an Item component
2. Set the Item's `Quantity`.
3. Add the `SeniorCitizenDiscountPolicy` to the Item as the default pricing policy and change the Item's price.
4. Change the Item's `Quantity`.
5. Change the item's price using the `AprilFoolsDiscountPolicy`.

To get the most out of this example, first read through `BusinessPolicyExample.java` on our web site. Then you can build it and run it.

Note: Be sure to set your CLASSPATH as described in "How to Build and Run the Examples" on page 5-2.

PassByValue

Sometimes it is useful to get or set all of the attributes of an object with a single method call. When dealing with remote objects that are persisted in the database, this results in a tremendous performance gain. BEA's WebLogic Commerce Server components give you that flexibility.

Package examples.passbyvalue

Shows the use of BEA WebLogic Commerce Server components' *pass-by-value* feature. This example demonstrates:

- How to use WLCS components.
- How to use *pass-by-value*

Table 5-4 PassByValue Class Summary

Class	Description
Pass-by-value example.	Pass-by-value example.

Package examples.passbyvalue Description

Shows the use of BEA WebLogic Commerce Server' pass-by-value feature.This example demonstrates:

- How to use WLCS components
- How to get and set values for WebLogic Commerce Server components using pass-by-value

Getting and Setting Attributes Using pass-by-value

This example shows how you can get and set the attributes of an Entity Component by value. What this means is that instead of getting/setting one attribute at a time, you can request that a local copy of all attributes be sent to you directly, in one remote call. You can then read and modify this "Value object" or local copy, and send it back in one remote call. This has tremendous performance advantages compared to accessing one attribute at a time It is also important to be able to set many attributes within a single transaction without having to begin/commit a JTS User Transaction from the client. In short, pass-by-value is really handy! Our implementation is tightly-coupled: that means that at compile time, we enforce type consistency for getting/setting attributes in the value objects. This has an advantage over "parameter sets"and "late-binding" implementations where you pass around a set of name/value pairs: with these approaches, if you change the type of an attribute your client will still compile but crash at runtime. This won't happen using WebLogic Commerce Server, since the value object will change accordingly and the client would not compile if an assignment was illegal. In addition, our value objects are generated by BEA SmartGenerator (based on a UML model), so they don't add any maintenance costs.

Pass By Value Example

The PassByValue example application performs the following steps:

- Find or create or a Customer component
- Create a value object and get the CustomerValue.
- Change Customer information
- Set the CustomerValues

To get the most out of this example, first read through `PassByValueExample.java` on our web site**.** Then you can build it and run it.

Note: Be sure to set your CLASSPATH as described in "How to Build and Run the Examples" on page 5-2.

Index