

MODELS OF BEANS?

JAVA AND UML WORKING TOGETHER...

John Jay King, King Training Resources

Java and UML

Java is the de-facto programming standard of the programming world. For years, the Object-Oriented (OO) world bickered over standards for analysis, design, and diagramming. Unified Modeling Language (UML) represents the joining of the "Three-Amigos" Rumbaugh, Booch, and Jacobsen and has quickly become an international standard. In addition, the Unified Process (UP) has become the process of choice for creating systems using UML. Since Java is an Object-Oriented language, most of its objects map directly to UML types. In this session attendees will see how UML is used to model various aspects of Java programming and deployment. We will also investigate the support for UML provided by Oracle's JDeveloper product (9i & 10g).

Introduction to UML

The Unified Modeling Language (UML) was first specified in 1994 and adopted as the standard for Object-Oriented Analysis and Design diagramming by the Object Management Group (OMG) in 1997 (<http://www.omg.org>). UML was built upon the groundbreaking work of Grady Booch, Ivar Jacobsen, and Jim Rumbaugh. For years the works of the "Three-Amigos" were competing standards in the Object-Oriented Analysis and Design World. Since the three got together and crafted UML, UML has quickly become accepted as the standard modeling language for Object-Oriented Analysis and Design. UML uses diagrams referred to as "artifacts" to describe the design. UML artifacts may be created manually using paper and pencil (or just about any tool), however, most people prefer to use either automated drawing tools or Integrated Development Environments (IDEs). It is important to understand that UML is neither a process nor a method, it is a diagramming standard. The Rational Unified Process (RUP) is one technique developed to use UML in the modeling of systems.

OMG

The Object Management Group (OMG) is a non-profit organization created by a consortium of software companies to provide a neutral source for standardized components and component definitions (<http://www.omg.org>). The complete UML standard may be found at the www.omg.org website. UML is one of the building blocks for the OMG's Model Driven Architecture (MDA).

Concepts

Object-Oriented Analysis and Design (OOA&D) provides the tools to analyze, design, and build using Object-Oriented concepts. Object-Oriented concepts include: Abstraction, Encapsulation, Inheritance, and Polymorphism.

Abstraction allows a developer to ignore details in order to concentrate on essential characteristics. The most common type of abstraction is Procedural Abstraction where processing details are performed by routines making code more clear and simple. Data Abstraction allows us to ignore the details of how a data type is represented internally and concentrate on how it is used. Encapsulation allow access to an object's data only via an object's methods. Non-object systems might use "black box" subroutines in a similar fashion.

Inheritance allows defining a superclass/subclass (base class/derived class) relationship where the generalization (superclass) shares data and methods with any specialized classes (subclasses) that extend the superclass.

Polymorphism is the ability to process different methods based upon the data in the message passed to the method. This is sometimes called "Dynamic Binding" when the choice is made at run time. Overloading is another, simpler, form of polymorphism; methods with the same name are selected by the compiler depending upon the arguments passed to them.

Modeling is used in many industries to reduce the costs of creating things. For instance, it is usually much cheaper for an aircraft manufacturer to discover that a model does not fly than it is to build an actual prototype first. Likewise, it is much cheaper for an architect to create a model to share with construction people than to find out that parts of the design are not optimal during the actual construction. To reiterate, it is far cheaper to throw away a blueprint than it is to start over once construction has begun. The question is, is it cheaper for a software to be modeled first? In many cases software models become the focal point of the process rather than the actual software being created, thus, the models add extra cost while providing little benefit. It is important to keep the design at the forefront and not get too caught up in the specifics of modeling.

Modeling software recognizes the reality that most software will go through a lifetime of iterations. Models of software provide useful information to people who need to create the next iteration of the software; this is where the cost savings (if any) occur. It is too often the case that software developers get caught up in the modeling and diagramming details rather than concentrating on the actual design, please stay focused... Few software systems spring into existence all at once, instead software is rolled-out in iterations or phases. The often-cited "80-20 Rule" implies that in most endeavors: 80% of the value takes about 20% of the work and 20% of the value consumes about 80% of the work.

The iterative process that is part of the Unified Process produces software in measured "chunks" yielding systems more quickly, that more accurately reflect the needs of the user. The iterative process provides incremental improvements in a cycle that gradually focuses on full functionality. The iterative process is generally broken into phases: Inception, Elaboration, Construction, and Transition.

Inception is where the approximate vision of the need is investigated. Inception often includes a business case, scoping, very vague estimates, and rudimentary prototypes. In short, just enough planning is done to determine whether or not to go on. Elaboration is where the vision is refined. During Elaboration core elements are implemented in iterations. Also any high-risk issues are also resolved. When Elaboration is complete, requirements and scope are fully understood and an executable prototype is complete. Estimates for the remaining work may be clarified once Elaboration is complete.

Construction is where the remaining elements and lower-risk issues are implemented in iterations. The final product is prepared for deployment including: user documentation, release notes, etc...

In the Transition beta-tests are performed and the rollout is planned. Finally, full deployment may occur.

UML uses several artifacts as part of the design process including: Use-cases, Use-case diagrams, Class diagrams, Sequence (and/or Collaboration) diagrams, Interaction diagrams, State diagrams, and Package diagrams.

Use-cases are used to obtain requirements from users in meaningful and manageable "chunks." Use-cases provide the mechanism for evaluating the success and/or failure of the design and construction in later phases. Class diagrams illustrate the structure of data to support the use-cases. Classes become the basis for software components. Sequence (and/or Collaboration) diagrams (interaction diagrams) illustrate the interaction between objects in a single use-case. State diagrams illustrate how one object's state changes over multiple use-cases. Package diagrams groups classes and illustrate dependencies between them.

Use-Cases and Use-Case Diagrams

A Use Case (or Scenario) is the mapping of actors to events and specification of how the event is considered to be a success or failure. The Unified Process defines Use Cases as: "A set of use case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor." An "actor" is something or somebody outside the system with behavior that involves the system. Actors are typically illustrated with a stick-figure in UML diagrams, even though the actor might not be human (clocks, devices, or an organization). Object-Oriented designers attempt to state Use Cases in a "black-box" form recording the responsibility of the system (the what) rather than how the system will accomplish something (the how). Use Cases reflect the functional requirements for the system. Use Cases might be stated at a very high-level (informal/casual), or, they might very rigorous defining exact actions and reactions required. Use Cases are goal-oriented representing what business tasks the system will accomplish. While most Use Cases are oriented to business needs, it is likely some Use Case scenarios will be driven by system requirements for security, auditing, backup/recovery, and so on.

Class Diagrams

Classes represent the "things" of interest to a system. Creation of Class Diagrams occurs several times in the Unified Process. Class diagrams might be completed with varying levels of detail: Conceptual, at a high-level; Specification, more code-oriented, describing Interfaces or Associations between classes without providing implementation details; Implementation, where each association is reflected by methods and attributes showing navigability paths.

Experienced teams might start listing expected Classes based upon the application involved. An easy way to start is to simply list each of the nouns found in the system Use Cases. Some organizations create CRC (Classes, Responsibilities, Collaborations) descriptions through role-playing. Classes have attributes (structure) and operations (behavior).

Class names are important because they occur repeatedly in the design and documentation, also, most will be represented by code eventually. Name classes using single tense (Customer rather than Customers) and concise business terms (Customer Order rather than Transaction). For those used to modeling databases, Classes seem very much like Database Entities. However, Database Entities describe the Attributes (structural) aspects but do not illustrate object behavior. Classes provide both the Attributes and Behavior of an object.

Classes are often categorized into loose groups called "stereotypes." Three commonly-used class stereotypes are: Domain Entity, Boundary, and Control.

Domain Entity (or, Entity) classes provide the ability to persist data, they will typically end up being Java Beans or Entity EJBs; they may or may not end up being translated into Relational Entities. Domain Entity classes will usually be realized as Java Beans or Entity EJBs. Typical Domain Entity classes might be: Customer, Order, or Order Line. Boundary classes provide the interface between external Actors and the Domain Entity classes, they will typically end up being user interface components like JSPs/Servlets. In diagrams, Boundary classes represent the interface between an external actor (e.g. Sales Clerk) and the Domain Entity (e.g. Order). Boundary Classes frequently provide a Graphical User Interface (GUI) via JSPs/Servlets, Applets, or Applications; however, if the external actor is another system or some kind of equipment the Boundary Class might provide interface behavior. Typical Boundary classes might be: CreateOrderPanel, MaintainOrderPanel, or CreditCheckInterface. Control classes coordinate activity providing control structures and behavior that does not fit as Domain Entity or Boundary classes, they will typically end up being Java Beans or Session EJBs. Control classes provide most of the implementation of Use Case pathways and the interaction between Boundary classes and Domain Entity classes. Control classes do not normally persist, they exist only as long as it takes to accomplish the goal of the Use Case. Usually at least one Controller class will exist for each Use Case. Control classes are frequently implemented as Java Beans or Session EJBs. Typical Control classes might be: ProcessOrders or MaintainOrders. Classes are related to other classes because it is the collaboration of objects that makes Object-Oriented systems work. Relationships describe the message mechanism between classes as well as object state and integration. Three types of relationships are normally considered: Association, Generalization, and Dependency. An Association relationship associates an object of one class to one or more objects of another class (Sales Clerk creates Orders). Generalization describes the inheritance framework (Superclass-Subclass) shared by classes; all non-private attributes and behavior of a superclass object will be available to any subclass object. Dependencies are Associations where a change in the state of one class object impacts the behavior of another class. Dependencies are more often modeled in Package diagrams than in Class diagrams.

Inheritance/Generalization

The Object-Oriented concept of Inheritance is documented in Class Diagrams using generalization. Generalization is illustrated using an open arrow at the end of the association pointing to the general class (superclass). Class diagrams illustrate multiplicity by listing the limits on number of objects that may be related using the following notation: "n" = Specific number of occurrences (most often 1); "n..m" = Range of occurrences from a minimum of n to a maximum of m; "n..*" = Unbounded maximum.

Composition and Aggregation

Composition means just what it sounds like, one class is made up of other class objects; the second class object has no meaning without the composition. Aggregation means that a whole is made up of parts like a Composition, but, the parts may exist independently from the whole. Aggregation is much less common than Composition (when in doubt, leave it out!).

Attributes (fields)

Attributes are logical data values of an object. If a Use Case implies the need to remember information about a class, that information should be an Attribute. Attributes have data types since they reflect real values. The most common attribute types are: String, Number, Date, Boolean, and Time. Other common attribute types include: UPC (Universal Product Code), EAN (European Article Number), SKU (Stock Keeping Unit), Address, Phone Number, and Zip Code/Postal Code. Attribute names should start with lowercase letters and use Uppercase letters to show each new subPart of a name. Attributes are listed with their data types in the middle compartment of the class (data type optional) formatted as:

```
+ varName : datatype = 'defaultvalue'      (+ = public, - = private, # = protected).
```

Operations (Methods)

Operations define the behavior of a class. Two questions may be asked to help identify operations: What does the class know how to do? What are the class's responsibilities? An Order class might use a getOrderValue operation to determine the value of an order. Here is the format used to specify the operation:

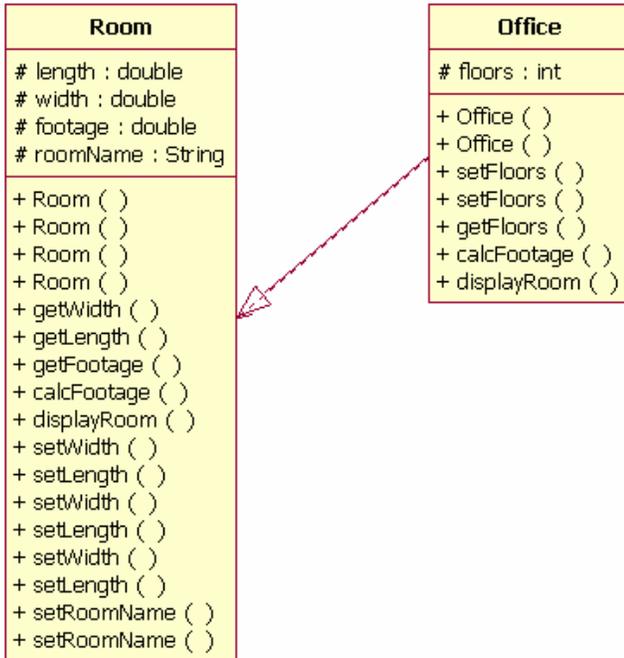
```
+ calcVarName(inputVar1:datatype,inputVar2:datatype) : returnDatatype
(+ = public, - = private, # = protected)
```

Interfaces

Interfaces are defined specifically in Java and C#, they may be simulated in C++ using a pure virtual class and in Visual Basic. Interface definitions look like class definitions. UML provides diagram features specifically for Interfaces called "Realization" that combines the inheritance and dependency symbols from classes.

Java and Class Diagrams

Here is an example Class diagram and the supporting Java code:



Room.java

```

package MyClasses;
import java.io.*;
/**
 * Sample Room Class
 *
 * @version 1.04 04/19/2000
 * @author John King
 */
public class Room {
    protected double length;
    protected double width;
    protected double footage;
    protected String roomName;
    /** Room constructor with no parameters
     * @return returns nothing
     */
    public Room() {
        this.width = 10.0;
        this.length = 10.0;
        this.roomName = "Generic";
        calcFootage();
    }
    /** Room constructor with parameters
     * @param width, length, roomName
     * @return returns nothing
     */
    public Room(double width, double length, String roomName) {
        this.width = width;
        this.length = length;
        this.roomName = roomName;
        calcFootage();
    }
    public Room(int width, int length, String roomName) {
        this.width = (double) width;
        this.length = (double) length;
        this.roomName = roomName;
        calcFootage();
    }
    public Room(float width, float length, String roomName) {
    
```

```

this.width = width;
this.length = length;
this.roomName = roomName;
calcFootage();
}
public double getWidth() {
return this.width;
}
public double getLength() {
return this.length;
}
public double getFootage() {
return this.footage;
}
public void calcFootage() {
this.footage = this.width * this.length;
}
public void displayRoom() {
System.out.println("Room width = " + this.width);
System.out.println("Room length = " + this.length);
System.out.println("Room name = " + this.roomName);
calcFootage();
System.out.println("Square footage = " + this.footage);
}
public void setWidth(double width) {
this.width = width;
}
public void setLength(double length) {
this.length = length;
}
public void setWidth(int width) {
this.width = (double) width;
}
public void setLength(int length) {
this.length = (double) length;
}
public void setWidth(float width) {
this.width = (double) width;
}
public void setLength(float length) {
this.length = (double) length;
}
public void setRoomName(String roomName) {
this.roomName = roomName;
}
public void setRoomName() {
try {
System.out.print("Enter room name: ");
InputStreamReader inStream = new InputStreamReader(System.in);
BufferedReader inBuf = new BufferedReader(inStream);
this.roomName = inBuf.readLine();
}
catch(IOException e) {
System.out.println("I/O error");
}
}
}
}

```

Office.java

```

package MyClasses;
import java.io.*;
/**
 * Sample Office Class
 * @version 1.04 04/19/2000
 * @author John King
 */
public class Office extends Room {
protected int floors;
public Office() {
this.floors = 1;
}
}

```

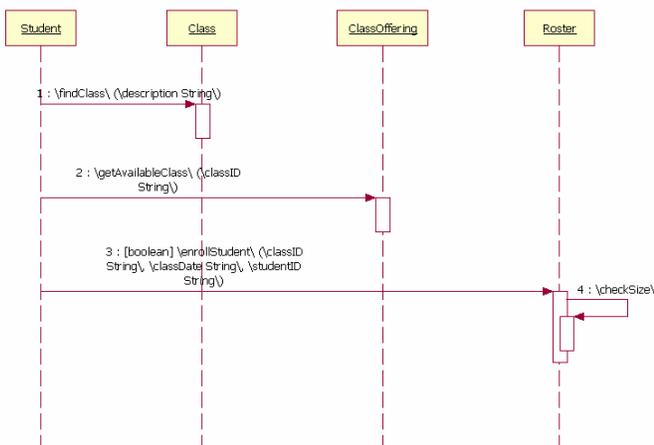
```

calcFootage();
}
public Office(int floors,double width, double length, String roomName) {
super(width,length,roomName);
this.floors = floors;
calcFootage();
}
public void setFloors(int floors) {
this.floors = floors;
}
public void setFloors(double floors) {
this.floors = (int) floors;
}
public int getFloors() {
return this.floors;
}
public void calcFootage() {
this.footage = this.width * this.length * this.floors;
}
public void displayRoom() {
System.out.println("# of floors = " + this.floors);
super.displayRoom();
}
}
    
```

Sequence (Interaction) Diagrams

Interaction diagrams are models that show how groups of object collaborate in some behavior and the flow of control between them. Usually, one Interaction Diagram describes the behavior of a single Use Case including: Names of objects involved in behavior, Messages flowing between objects, and the Sequence of messaging. Two kinds of Interaction Diagrams are used, they show the same information in different forms: Sequence Diagrams (most common) emphasize the sequence of messages and Collaboration Diagrams (less-frequently used) emphasize the relationships between objects.

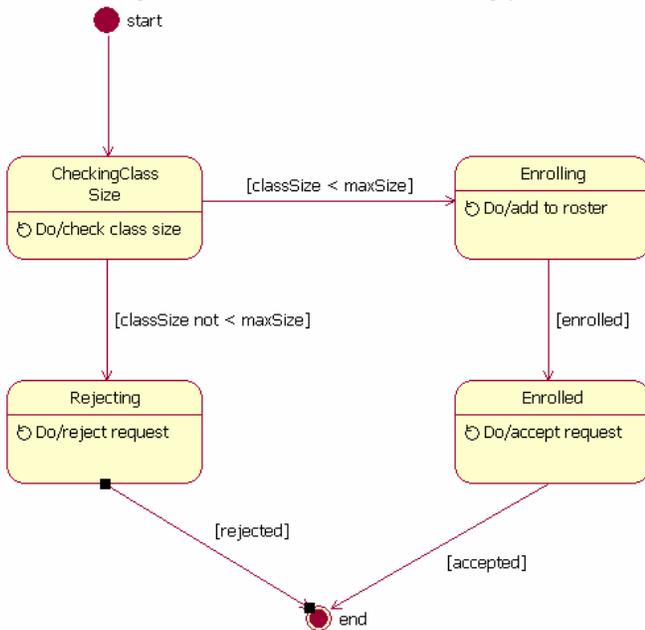
The Sequence Diagram is one of the most heavily used UML artifacts. Sequence Diagrams show the flow of messages from one class/actor to another, these will be represented by Java methods. Each object in the Sequence Diagram represents a class (probably an Entity Class or Boundary Class). Each message in the Sequence Diagram represents a method call and signature. Collaboration Diagrams offer another view (usually exclusive of Sequence diagrams). Collaboration Diagrams contain basically the same information as Sequence Diagrams and are frequently done without. Other than appearance, the significant difference between Sequence Diagrams and Collaboration Diagrams is the emphasis: Sequence Diagrams emphasize the sequence of messages (and thus, the operational sequence of the application); Collaboration Diagrams emphasize relationships rather than sequence.



State Diagrams

State diagrams (sometimes called State Chart Diagrams) describe the behaviour of a system, illustrating the different states an object might experience and an object's reaction to different events. States are the condition at a moment in time, the time between events such as "Waiting for credit authorization." State Diagram events are significant activities in the system, for

instance: “A credit purchase is authorized.” Transitions are relationships between two states that indicate that an event has occurred for instance: “When the authorized event happens, the purchase transitions from pending to complete.” State diagrams may be applied to classes, use cases, and any other UML document where it is desirable to map out the flow. In fact, state diagrams resemble flow charts strongly.



Other Diagrams

UML has numerous other diagram types including Package Diagrams, Use-Case Diagrams, and others.

UML and the J2EE world

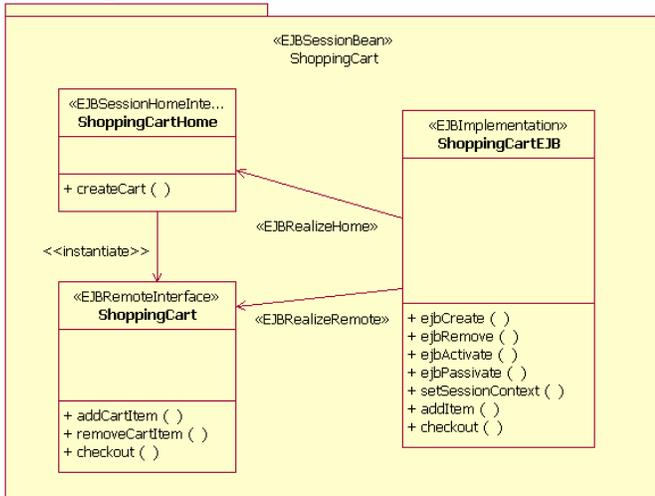
J2EE is an architecture defining how applications and technologies should interact. J2EE is a family of Application Programming Interfaces (APIs) and services designed for the complexity of modern enterprise computing. The real idea behind J2EE is that it probably makes more sense to buy complex components than to build them. Common components that can be difficult to build include: Transactions, Security, Communications, Fault Tolerance, Database Connectivity, Monitoring, and Control. J2EE Containers provide runtime capability for: Transaction management, Security checks Resource pooling, and State management. Containers provide a collective view of the J2EE APIs (Application Programming Interfaces) used by specific application components.

J2EE user interaction is separated into three tiers: Client Tier, Web Tier, and Business Tier. The Client Tier provides the user interface usually including Applet Containers and Application Containers. The Web Tier generates the Client Tier presentation and processes the user response usually with JSPs and Servlets. The Business Tier provides the business logic of the system in the EJB (Enterprise Java Bean) Container.

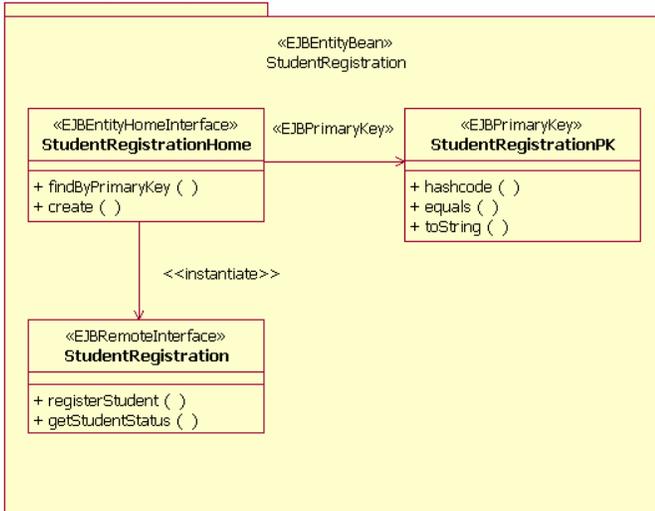
EJBs are not the end-all in software development, in fact, there may very well be simpler choices (e.g. servlets) that do not require their robustness; however, when needed EJBs are extremely useful. EJBs should be considered when: the application is required to support multiple clients or client types, multiple concurrent updates and accesses, when the application requires control of the database transaction, when complex security requirements exist within the system, or when large-scale scalability is required.

UML does not have specific modeling for EJBs. UML class definitions are too fine-grained to fully support EJBs, but, they can be combined with packages to make things workable. UML does have a component object which is similar to the notion of an EJB component, however, UML components are not typically modeled. A third option is to model the EJB as a subsystem using a combination of class and package attributes allowing a model that matches up nicely with EJBs.

Modeling Session EJB



Modeling Entity Beans



Java and JDeveloper

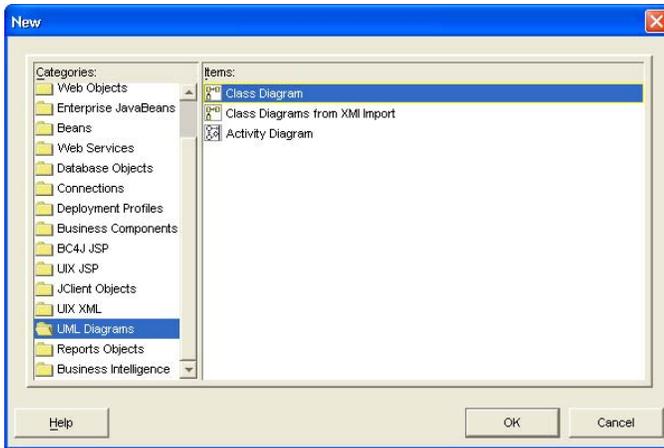
JDeveloper is a full-featured Integrated Development Environment (IDE) used to create Java programs including J2EE applications. JDeveloper provides features tailored to the Oracle environment.

JDeveloper Support for UML

JDeveloper allow you to create new class and activity diagrams. All of the objects are available from any project in the Navigator. JDeveloper gives you three alternatives: Class Diagrams you create, Activity Diagrams you create, and Class Diagrams imported from XMI definitions created by UML modeling tools.

Class diagrams allow you to model both Java classes and Oracle Business Components as part of a JDeveloper project.

Activity diagrams allow you to model activities, flows and states. Activity diagrams may also be used to generate code for Oracle Workflow and Oracle9i Advanced Queuing.



UML future directions

Just as it is possible to move from UML to Java code, the return trip can be made; moving from Java to UML is referred to as Reverse-Engineering. Modeling with UML, generating Java, manipulating the Java, and then Reverse-Engineering the changes back into the model is referred to as Round-Trip Engineering. Modern tools are beginning to offer this round-trip capability with varying degrees of success. While there is a good match between standard Java and UML, the same is not true of EJBs. Most tools that reverse-engineer Java into UML cannot properly handle EJBs or other J2EE constructs (they are not in the UML specification). A few tools provide full round-trip engineering; the success of the reverse-engineering depends upon the complexity of the application.

A Java Service Request (JSR) is underway (JSR-000026 UML/EJB Mapping) in the Java Community Process (JCP) to utilize the UML profile as a tool for mapping EJBs (See the following URL for more <http://jcp.org/en/jsr/detail?id=026>).

About the Author

John King is a Partner in King Training Resources, a firm providing instructor-led training since 1988 across the United States and Internationally. John specializes in application development software on a variety of platforms including Web, Unix, Linux, IBM mainframe, and personal computers. John has worked with Oracle products and the database since Version 4 and has been providing training to Oracle application developers since Oracle Version 5. John develops and presents customized courses in a variety of topics including Oracle, DB2, UDB, Java, XML, C#, and various programming languages. He has presented papers at many industry events including: IOUG-A Live!, UKOUG Conference, EOUG Conference, AUSOUG Conferences, RMOUG Training Days, MAOP-AOTC, NYOUG, and the ODTUG conference.

John Jay King
King Training Resources
6341 South Williams Street
Littleton, CO 80121-2627
U.S.A.
Phone: 1.303.798.5727 1.800.252.0652 (within the U.S.)
Fax: 1.303.730.8542
Email: john@kingtraining.com
Website: www.kingtraining.com