Developing J2EE Applications with the UML and Rational Rose

Khawar Ahmed

Rational Software White Paper

Rational®
the e-development company™

Table of Contents

Introduction

As businesses try to develop software faster and more predictably with higher quality, the Java 2 Platform, Enterprise Edition (J2EE) is emerging as a popular standard for building enterprise applications. Not only does J2EE provide a comprehensive paradigm for developing enterprise applications, it also helps bring together a previously disjoint set of diverse technologies.

Quite unsurprisingly, the keys to success with J2EE are the same as with any complex software platform: effectively communicating requirements, making the right analysis and design decisions, and identifying the best implementation choices.

Organizations that follow the industry accepted best practice of visual modeling are able to develop their software faster and build better quality systems. The Unified Modeling Language (UML) is the software industry's standard for such visual modeling.

In this white paper, we explore how you can effectively leverage the UML and Rational Rose 2001a, the leading UML based software modeling and development tool for developing J2EE based enterprise applications.

What is the Unified Modeling Language (UML)?

The Unified Modeling Language (UML), an OMG standard since late 1997, is a graphical language for the modeling and development of software systems. It provides modeling and visualization support for all phases of software development, from requirements analysis to specification, to construction and deployment.

The central idea behind using the UML for visual modeling is to capture the significant details about a system such that the requirements for the project are clearly understood, solution architecture is developed, and a chosen implementation is clearly identified and constructed. A rich notation for visually modeling software systems is needed to accomplish this. The UML not only provides the notation for the basic building blocks, it also provides for ways to express complex relationships among the basic building blocks. Such relationships are captured in the form of UML diagrams.

Let's look at how the UML and Rational Rose can help in understanding, designing and implementing J2EE applications.

Understanding requirements

Projects often fail because the requirements were not well understood or communicated. This is not too surprising in light of the fact that language, whether written or oral, is imprecise by nature and ambiguous.

You can apply UML use case modeling to develop a precise model of what is required of the system, and then utilize the use cases as the basis for driving other aspects of your enterprise system development. In effect, a use case acts as the string that binds the pearls of a necklace together. Use cases bridge the gap between the end user and the requirements of the system. They can be used to establish traceability between functional requirements and the system implementation itself. The use cases also serve as a connection point to the use case document where the details of the requirements are captured.

Figure 1 shows a partial use case diagram for an online CD store, derived by distilling the written and verbal requirements of the expected functionality into use cases. In this case, it is immediately obvious that the shopper (represented by the stick figure known as an actor in the UML) can use the system in one of four ways (each shown via the ellipse referred to as a Use Case in the UML).

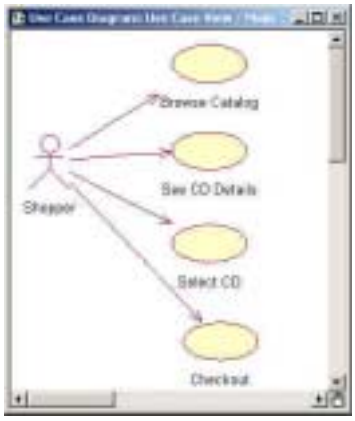

Figure 1: A simple use case diagram

Each use case, in turn, is elaborated via one or more scenarios typically via sequence diagrams. Of course, in this early phase of requirements capture and analysis, the sequence diagrams, by necessity, are relatively simple and may be incomplete. An example of such a sequence diagram is shown in Figure 2. In Rational Rose, you create such sequence diagrams that are tightly related to specific use cases by selecting the use case in the browser and selecting New>Sequence Diagram from the use case context menu.

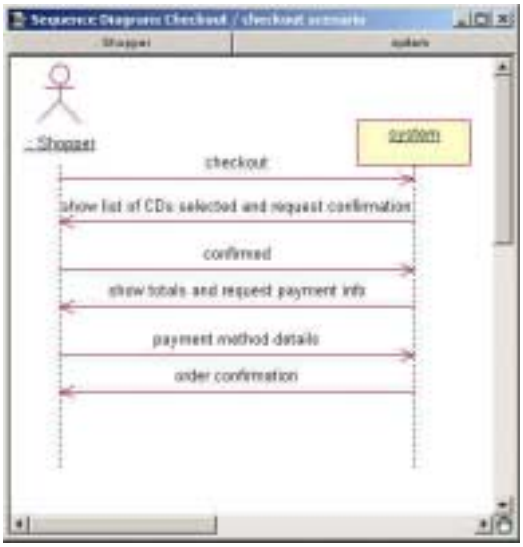

Figure 2: A sequence diagram illustrating checkout use case

Designing a solution

The next stage, Use Case Analysis, provides an initial, high-level definition of how internal elements interact in order to satisfy the system's functional requirements, and how they are related to each other statically. This activity can involve much trial and error before satisfactory solutions are created. "Analysis classes", for which behaviors often are described abstractly using natural language, are a useful tool to use during this analysis. Analysis classes are usually not implemented in software, although they can be. Rather, the analysis classes are refined later in the overall design process into precisely defined design classes and subsystems.

This starts by elaborating the sequence diagrams such that they reveal the internal workings of the system and instead of showing the interaction between actors and a monolithic system, the system is split into analysis level objects. The responsibilities of the system are divided among the analysis level objects to achieve a finer grained sequence diagram. Three kinds of analysis objects are used:

- Boundary Objects

  Boundary objects represent all interactions between the system's inner workings and its surroundings. These include interaction with a user via graphical user interface, interactions with other actors (such as those representing other

systems), communications with devices, and so on. Boundary objects serve to isolate and shield the rest of the system from external concerns. Generally speaking, each actor-use case interaction pair maps in a boundary object.

- Entity Objects

Entity objects represent information of significance to the system. They are usually persistent and exist for an extended duration. Their primary purpose is to represent and manage information within the system. Key concepts within a system manifest themselves as entity objects in the model.

- Control Objects

Control objects are used to model behavior within the system. Control objects do not necessarily implement the behavior, but may instead work with other objects to achieve the behavior of the use case. The idea is to separate the behavior from the underlying information associated with the model, making it easier to deal independently with changes in either later on

The UML provides the notion of a stereotype, represented as text enclosed in double angle brackets, to distinguish between different types of classes. In Rational Rose, you can easily create analysis classes by changing the class stereotype field to <<boundary>>, <<entity>>, and <<control>> respectively. These can then be used as the basis for creating analysis level diagrams.

An updated version of the sequence diagram for the checkout use case, this time with the system decomposed into analysis objects, is shown below in Figure 3. The figure uses iconic representation for the boundary, control and entity objects (circle with a T, circle with an arrow, and circle with a tangential line, respectively).

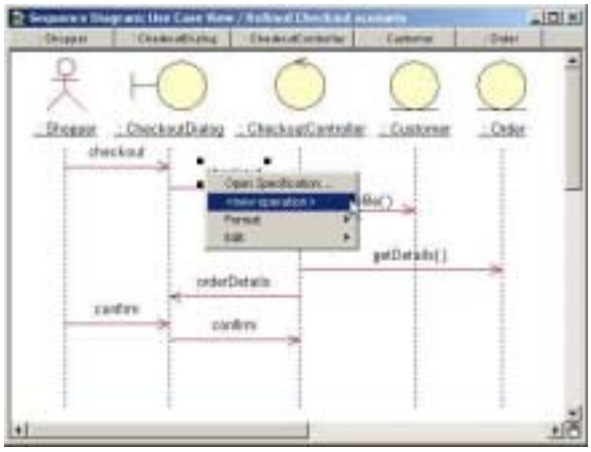

Figure 3: Refined sequence diagram with analysis objects

Of course, classes often participate in several use cases and it is equally important to understand their static relationships to ensure consistency across the system. The UML class diagram is useful for capturing the static relationships between different structural elements.

As a first step, we identify and place all the classes that participate in the use case on a class diagram. We have already distributed the behavior of the use case to the objects so it is a relatively simple exercise to create analysis operations for the responsibilities assign to each. It is important to note that these are analysis operations, meaning that these operations will most likely need to evolve as we continue with our analysis and design efforts.

Rational Rose allows you to easily define new operations on the analysis class from sequence diagrams by selecting the existing message and choosing <new operation> from the context menu (as shown in Figure 3). If you have already defined operations on a class, you can simply select the existing operation from the list.

This is typical of the approach used in Rational Rose to improve user productivity and ensure consistency, and therefore quality, across your entire model. Other similarly useful capabilities include the ability to query the model on which classes and messages are unresolved (i.e. unmapped to actual classes or operations in the model).

Another aspect of fleshing out each individual class is to identify attributes for the class. Attributes represent information that may be requested of the class by others or that may be required by the class itself to fulfill its responsibilities. At this stage in the analysis, it is appropriate to identify attributes as generic types such as number, string, etc.

Identifying the relationships between the classes completes the class diagram for the use case. The relationships we are specifically interested in at this stage are association, dependency, and inheritance.

Having analyzed all the use cases and having created the class diagrams for each use case, it is time to coalesce the various analysis classes to arrive at a unified analysis model. This is an important activity, as we want to arrive at a minimal set of classes and avoid unnecessary redundancy in the final analysis model.

The key task at this stage revolves around identifying classes that may be duplicated across use cases or masquerading in slight variations. For example, control classes that have similar behavior or represent the same concept across use cases should be merged. Entity classes that have the same attributes should also be merged, and their behavior combined into a single class.

Figure 4 shows a preliminary analysis level class diagram for the use cases identified in Figure 1. As we're primarily interested in the relationships between the classes, we've used Rational Rose's display filtering capabilities to filter out the details of the individual classes by un-checking Format>Show all attributes and Format>Show all operations.

Figure 4: Preliminary analysis level class diagram

Implementing the software

While the analysis model can help you develop a solid foundation for solving a problem, it is still far from an implementation. During design, you must take into account additional restrictions and requirements imposed upon your application by virtue of the underlying technology, and try to map the solution to the optimal implementation. For instance, if you are building an online application, which requires a client application, the technology you use may be different from the technology choices for a functionally identical web based application.

For our example, let's assume we are building a web-based application. Implementing such an application requires a well-thought out approach. The analysis model helps here as it provides the starting point for determining how the different J2EE technologies map to the solution. As it turns out, the <<control>> classes map nicely to Java Servlets or to Enterprise JavaBean (EJB) Session Beans. This approach lines up well with the J2EE tiered implementation model and the Sun "Model 2" Reference Architecture.

Rational Rose provides a simplified interface for the development of servlets as well as EJB Session Beans. Figure 5 shows the dialog for creating a servlet.

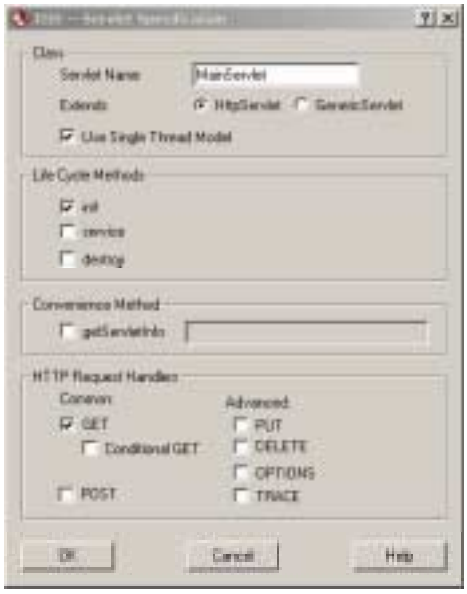

Figure 5: Servlet definition dialog

Rational Rose provides a convenient interface for creating EJBs, even though an EJB consists of multiple interfaces and classes. The dialog for EJB creation is shown in Figure 6. In this specific case, the dialog shows the setting required to create a stateless Session Bean.

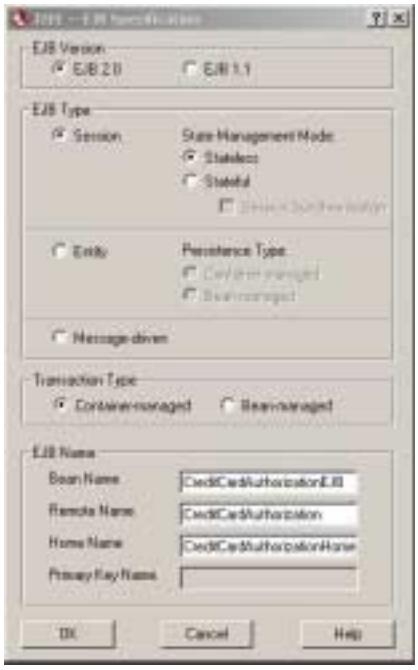

Figure 6: Creating an EJB Session Bean

The resulting Session EJB is shown in Figure 7. This is based on the UML modeling for EJB profile being developed in JSR-26 under the Sun Java Community Process. This shows relationships between the various elements that make up an EJB, specifically the home and remote interfaces and the EJB implementation class. Since the EJB does not actually implement the home and remote interfaces (these are implemented by objects auto-generated by the deployment utility), the relationship is not a realization but <<EJBRealizeHome>> and <<EJBRealizeRemote>> respectively. The dependency between the home and remote interfaces shows that the home interface instantiates the remote interface.

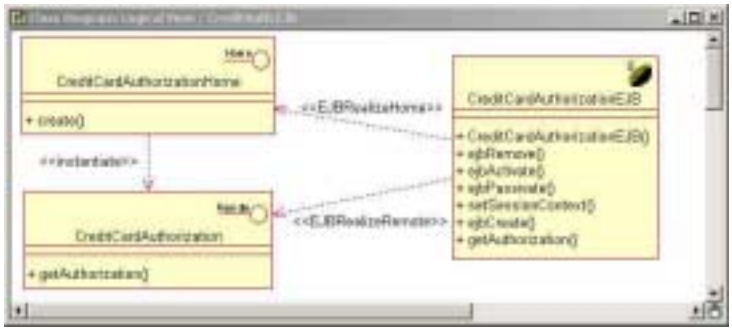

Figure 7: A stateless Session Bean

<<boundary>> classes roughly map to JSPs, HTML pages, and forms, or if you are developing a traditional client based application, to a client application dialog. We use the JSP as the builder of whatever is presented to the entities interacting with the system. Since a JSP really has two aspects to it, namely client presentation and server side behavior, it is modeled as consisting of a Client page and a Server Page, with a special stereotype <<build>> on the relationship. An example is shown in Figure 8.

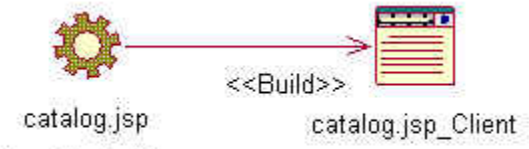

Figure 8: JSP as a server and client page

Creating a JSP is even simpler than an EJB and is done via the Web Modeler>New>Server Page menu off the browser context menu, as shown in Figure 9.

Figure 9: Creating a JavaServer Page in Rational Rose

As it turns out, <<entity>> classes such as Catalog, Order and Customer are good candidates for Entity Beans. These are created using the dialog shown in Figure 6 for the creation of Session Beans.

A common technique utilized in J2EE applications is the use of JavaBeans to pass information between the servlets and the JSPs. This is easily accomplished in Rational Rose by creating attributes on a Java class and setting the attributes to properties via the attribute specification dialog shown in Figure 10 (note that this can also be done on a global basis through a user setting).

Figure 10: Setting an attribute as property

Creating JSPs, servlets, JavaBeans and EJBs is most useful in the context of an overall implementation model. Rational Rose allows you to easily model the relationships and forward engineer the basic details of not just JSPs, servlets, EJBs and JavaBeans, but also HTML pages and Forms involved in the implementation. These can then be handed off to the presentation developers for further refinement, while still ensuring compatibility with your application logic.

Figure 11 shows a partial class diagram showing the implementation with the different technologies involved in the online CD store. The diagram is semi-arranged to show classes and how they lineup with the presentation, business logic and data tiers. Thus, the client pages are on the left, the controller servlet is in the middle, and the Entity Beans are on the right.

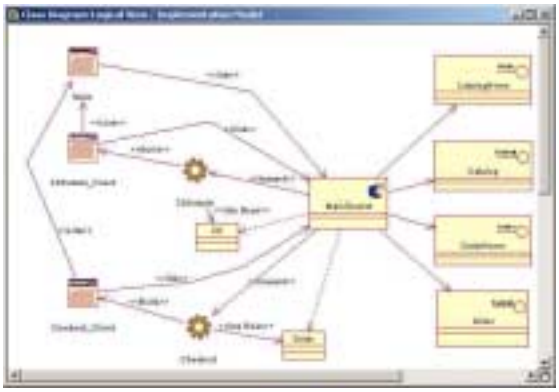

Figure 11: Partial Implementation model for the online CD store

This diagram only shows some of the classes required for the *checkout* and *see CD details* use cases. Let's try to "read" the diagram by going through a simple checkout scenario. Assume that you've just finished browsing and have selected a few CDs you would like to purchase. You select the "checkout" option on the Main page (top left). This invokes the MainServlet controller as shown via the association stereotyped <<link>>. The MainServlet obtains the order details from the Order EJB, constructs a Order JavaBean, sets it as an attribute for the session, and forwards the request, as identified via the <<forward>> stereotype, onto the Checkout JSP. The Checkout JSP uses the Order JavaBean, as shown via the <<Use Bean>> stereotype on the association between the JSP and the Order JavaBean, to construct the Checkout_Client page and presents it to you.

Obviously, we have neglected some details. For example, in a real project, you would probably use a shopping cart to keep track of items; the control responsibilities may be more distributed rather than reside with a single monolithic MainServlet; and so on. The main point here is that the UML is a powerful tool for designing and developing complex J2EE applications and hopefully that has been successfully communicated via the example above.

Working with the implementation

Rational Rose lends a helping hand in all this by allowing you to generate JSP, JavaBean, HTML, Servlet, and EJB code directly from this diagram.

For instance, an <<include>> association between two JSPs (not shown on diagram) automatically results in: *<%@ include file="header.jsp" %>* in the appropriate JSP. Similarly, the <<Use Bean>> stereotype results in: *<jsp:useBean id="cd" class="com.rational.cdshop.util.CD" scope="session"/>* in the JSP using the CD JavaBean.

On the EJB side, aside from generating code for all three types of EJBs identified in EJB2.0 (as well as EJBs compliant to EJB 1.1), Rational Rose provides several features to simplify EJB development. For example, one of the tedious things in developing EJBs is the need to code the methods in both the interface and the implementation class. Rational Rose provides a menu option, which takes care of these details at the click of a button. It also provides a "check and repair" menu option that verifies that the EJB you have defined is in fact legal (for example, the remote methods in the remote interface have matching counterparts in the EJB implementation class) and if not, it offers to fix them for you.

Another powerful EJB related capability offered by Rational Rose is the Rational Quality Architect (RQA) for design testing of EJBs. For instance, you can use RQA for unit testing EJBs. You can also use the sequence diagrams you defined in Rose to drive the testing of multiple EJBs. As well, you can use RQA to generate stubs for those times when you have a dependency on a software component that is not yet ready.

To ensure you can work with the implementation on your terms without worrying about changes to the UML model and vice versa, Rational Rose provides a built in code editor, with user configurable synchronization options. For example, you can choose to keep synchronization on all the time. In this case, the UML model in Rose will be automatically updated whenever you update you source code and press save. Alternately, there may be times when you only want to experiment to see how things might work out, but not impact your model. For such situations, you can turn synchronization off on a global or per-class basis.

Of course, there's no disputing the fact that when you are talking implementation and code, you need to work with an industrial strength IDE such as Sun's Forte for Java or Borland JBuilder. Rational Rose 2001a provides deep integration and auto-synchronization with the leading IDEs so you can continue to use your favorite one while taking full advantage of Rational Rose for UML modeling and development for J2EE applications.

More on the UML and J2EE

We have only scratched the surface of using the UML for modeling and developing J2EE applications. For example, you can use UML activity diagrams to model session management by the various entities involved in the session.

Another challenge is communicating the proper sequence of operation calls expected by a session EJB. A sequence diagram can identify a single scenario but you need a lot of sequence diagrams to convey the entire range of scenarios supported by the component (some try to use a sequence diagram with control and branch statements but that only leads to a complex and convoluted sequence diagram). On the other hand, the UML statechart diagrams provide a powerful capability for modeling and communicating this. You can then validate a sequence diagram against the statechart diagram by "walking" the sequence diagram through it to see whether the component supports the way you are trying to use it.

Summary

There is a fine line between developing software that does the job and developing software that not does the job well today, but is also ready to face the requirements yet to come.

You can significantly improve your chances of developing scalable, easily maintainable, and long lasting software by using the UML for understanding the requirements, doing proper analysis and design, and developing a solution built on proven principles and implementation best practices.

Rational Rose 2001a Enterprise Edition helps you do all these and is the only UML modeling tool that also fully support the modeling and round-trip engineering of J2EE applications using HTML pages, JSPs, Servlets, Java Beans, and EJBs in the context of the Sun Model 2 Reference Architecture.

About the author: *Khawar Ahmed is in Rational Rose eBusiness Products Team, focused on J2EE and web modeling. He is the co-author of "Developing Enterprise Java Applications with the UML", to be published by Addison-Wesley in 2001. He can be reached at kahmed@rational.com*

Rational®
the e-development company™

Corporate Headquarters
18880 Homestead Road
Cupertino, CA 95014
Toll-free: 800-728-1212
Tel: 408-863-9900
Fax: 408-863-4120
E-mail: info@rational.com
Web: www.rational.com

For International Offices: www.rational.com/worldwide

Rational, the Rational logo, Rational the e-development company and Rational Rose are registered trademarks of Rational Software Corporation in the United States and in other countries. Java, J2EE, EJB are trademarks of Sun Microsystems. All other names used for identification purposes only and are trademarks or registered trademarks of their respective companies. ALL RIGHTS RESERVED. Made in the U.S.A.

© Copyright 2001 Rational Software Corporation.

TP197 5/01. Subject to change without notice.