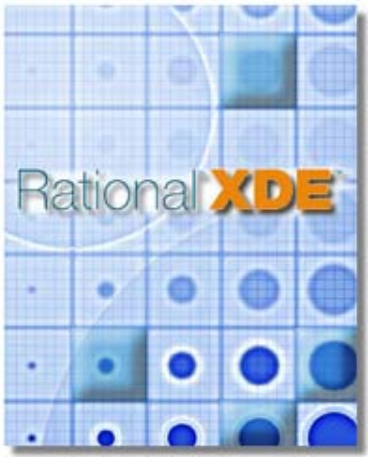
Accelerating J2EE Development with Rational XDE

by **Khawar Z. Ahmed**
Rational XDE Technical Marketing

Rational® XDE™ provides a truly seamless integrated visual modeling environment within the most popular Integrated Development Environments (IDEs). But there is a lot more to Rational XDE's capabilities than just automated code generation and effortless model/code synchronization. To get full benefit from Rational XDE, you need to understand its powerful internal J2EE design patterns and code templates, which you can either use as is or customize. This article illustrates how to use these Rational XDE capabilities by walking you through a simple J2EE project.



Sample Application Overview

To illustrate how to use Rational XDE's unique capabilities, we will implement portions of a simple Web-based loan management application that allows loan owners and loan administrators to perform a few simple tasks:

Loan Owners

- Make payments online.
- View amortization schedules via the Web.

Loan Administrators

- Set up one or more loans for a client.
- Obtain information about upcoming loan payments due.

We'll assume that we must leverage the existing infrastructure: an IBM WebSphere application server and several databases.

First, we can use Rational XDE's free-form modeling capabilities to communicate broad ideas about the direction of our solution, as shown in Figure 1. Such free-form diagrams offer a distinct advantage: They allow you to continue working within one toolset and to easily integrate planning documents into the project.

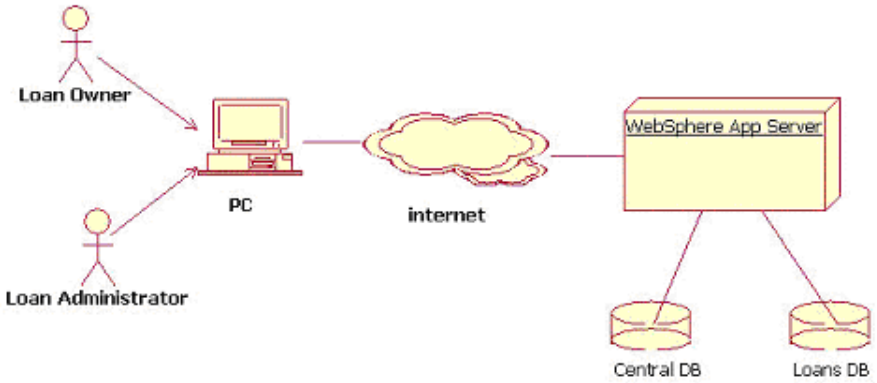


Figure 1: Free-Form Diagram Illustrating Broad Outlines of the Solution

Automating Analysis and Design Activities

Once the preliminary planning is done, we can move into analysis and design. Although Rational XDE runs within a software development IDE, it provides all capabilities needed to accomplish traditional analysis and design activities -- use-case modeling, sequence diagrams, and so forth -- right inside the IDE. In fact, Rational XDE comes with the Rational Unified Process® Configuration for Java Developers (RCJD), which provides proven guidelines for Java software development.

Figure 2 shows the initial use-case model of the simple loan management system, arrived at via further elaboration and discussion with the customer.

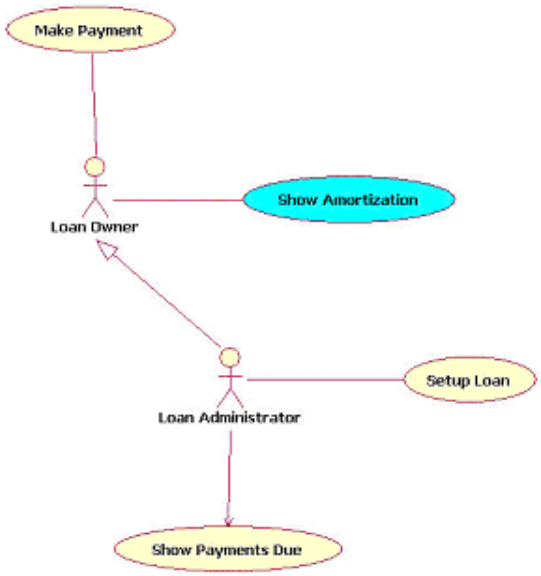


Figure 2: Use-Case Model for the Simple Loan Management System

For the sake of brevity, we'll follow the progress of just one use case -- Show Amortization -- to illustrate how Rational XDE capabilities come into play during development. The initial sequence diagram, elaborating the basic use-case scenario, is shown in Figure 3.

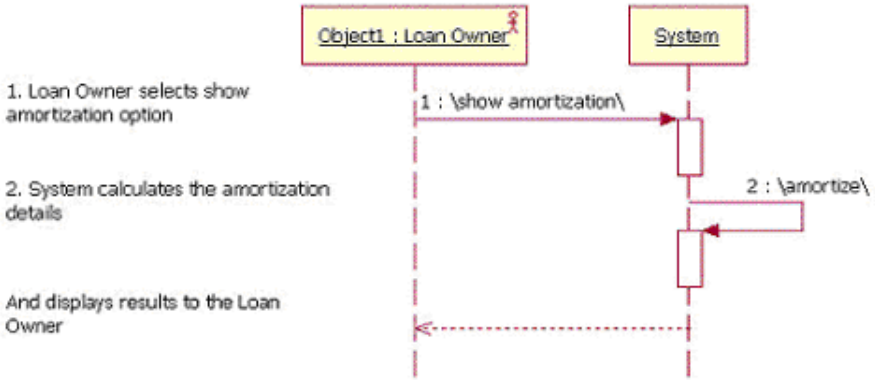


Figure 3: Sequence Diagram to Elaborate Show Amortization Use Case Basic Flow

Obviously, this is a very high-level view of the interaction, covering none of the system's internal workings. The Rational Unified Process suggests using analysis level classes to provide further detail, as shown in Figure 4.

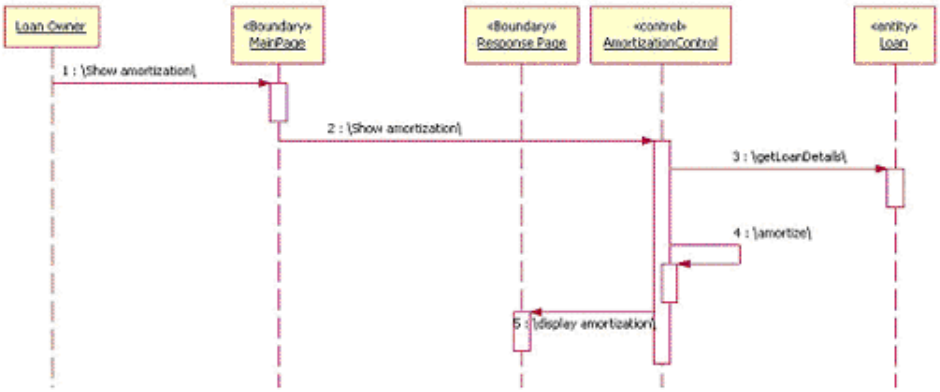


Figure 4: Refined Main Flow for the Show Amortization Use Case

The exact details of how to arrive at this diagram are beyond the scope of this document, but you can consult the first three **References** listed below for further information. Note that in this case, we've chosen to split the boundary into *two pages*. A simpler approach might depict all boundary interactions via a single element.

Creating Models with a Custom Pattern

You are probably familiar with design patterns such as Gang of Four patterns and Core J2EE Patterns. These pre-defined patterns are supported by Rational XDE, and we will explore usage of some of the Core J2EE patterns later on in this article. Rational XDE also goes a step farther, however, by providing the capability to define your own custom patterns easily via the UML. Rational XDE's custom patterns capability allows you not only to implement custom designs, but also to automate some tedious aspects of creating models, such as generating multiple use-case flow diagrams. Use cases typically have more than a single flow of events. For example, the Show Amortization use case has at least two flows of events: one for the successful scenario documented above and another for failures that might occur while trying to obtain loan details. Figure 5 shows a custom pattern (UCArtifacts) for generating diagrams for the main (successful) flow of events and two failure flows.

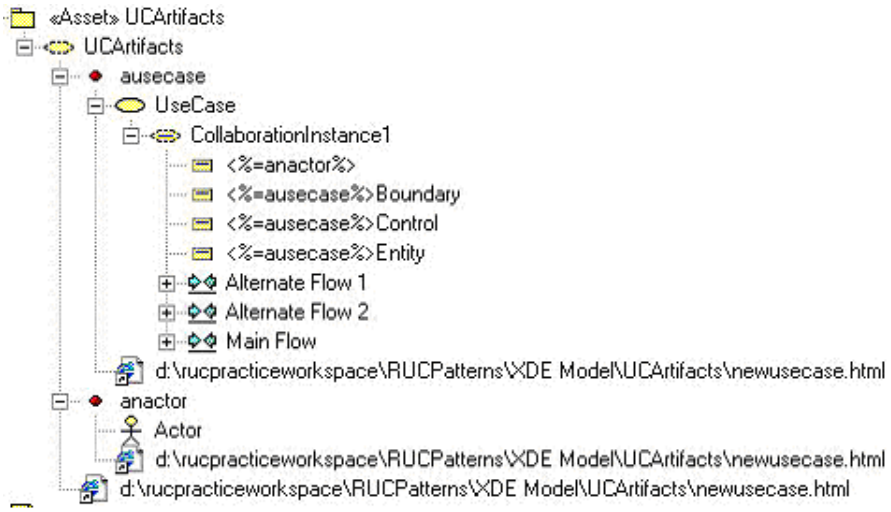


Figure 5: Custom Pattern (UCArtifacts) that Automates the Creation of Use-Case Flow Diagrams

This simple pattern consists of two input parameters; *ausecase* takes a use-case model element as input, and *anactor* takes an actor model element as input. The pattern uses the names of these input parameters to create objects by appending the word *Boundary*, *Control*, or *Entity* to the use-case name. These objects are then used to define a skeleton sequence diagram for the main flow diagram and two alternate flow diagrams, based on standard UML diagrams supplied in XDE. It is as simple as creating sequence diagrams for the use case and dragging/dropping the associated objects on to the diagram from the Model Explorer. Each of the two alternate flow diagrams the pattern creates has the supplied Actor, and one each of the boundary, control, and entity objects. We can complete the diagram simply by drawing the required messages between the different objects and deleting any analysis objects that do not participate in the flow.

Arriving at the Design

The initial class diagram for the Show Amortization sequence diagram is shown in Figure 6.

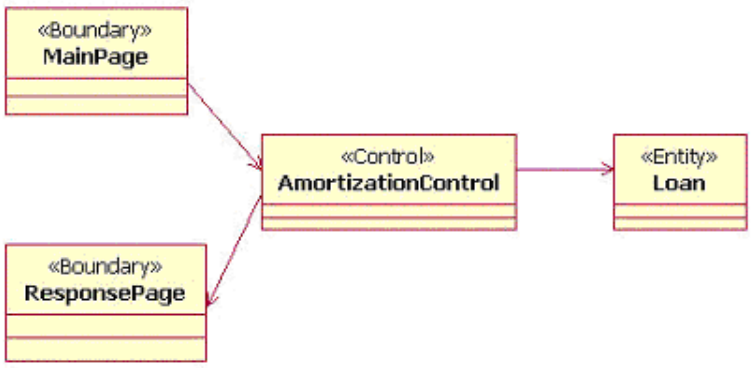


Figure 6: Initial Class Diagram for the Show Amortization Use Case

This diagram shows the basic roles needed to fulfill key requirements but does not consider classes that participate in implementing other use cases. To arrive at a unified analysis model, we would need to analyze those other use cases in a similar manner and then combine the results. At this stage, however, our primary goal is to identify and eliminate duplicate classes and merge the ones with similar or overlapping functionality. For instance, we could merge the Amortization Control object with the `MakePaymentControl`object to arrive at a single Controller object that can handle multiple use cases. As we are focusing on the Show Amortization use case, we will not

show all of these operations here, but you can consult the first three **References** listed below to learn more.

Implementing the Solution in J2EE

So far this discussion has been mostly implementation-technology independent. Now, let's see how the capabilities described above relate to J2EE development. In general, the analysis classes we discussed above map to J2EE technology as follows:

Boundary

- HTML
- JavaServer Page (JSP)

Control

- Servlets
- Session Beans
- Message Driven Beans

Entity

- Entity Beans
- JavaBean

This is a starting point for identifying the right J2EE technology to use in developing your application. Because different technologies can be used to implement similar solutions, the best choice will depend on your implementation details and requirements.

Core J2EE Patterns

Regardless of the problem you are facing, if you are implementing a J2EE solution, I strongly encourage you to learn and use the core J2EE patterns as part of your everyday development efforts. And of course, you can create your own custom patterns that either utilize the core J2EE patterns or customize them to make them more suitable to your needs. A collection of best practices and design strategies assembled by the Sun Java Center, these patterns are based on proven techniques identified in real-world J2EE projects. Rational XDE provides a robust implementation of all fifteen patterns (see Figure 7). [1]

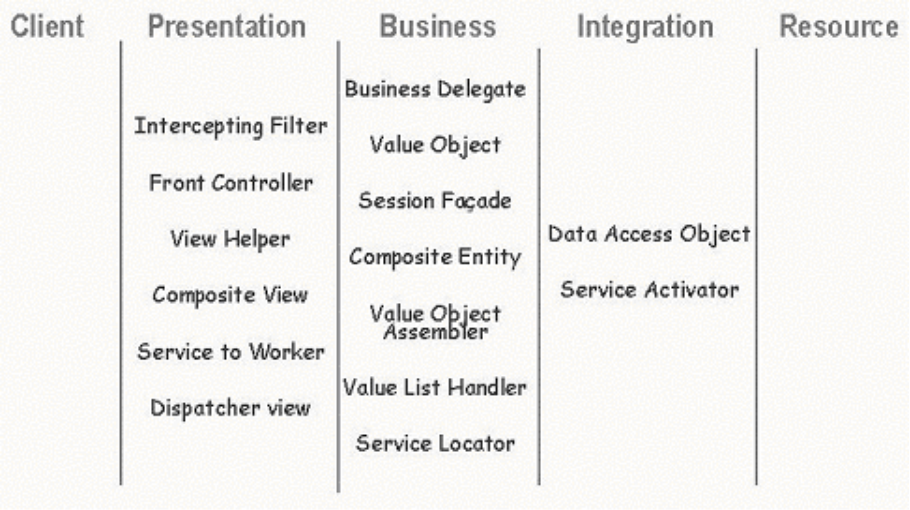


Figure 7: Core J2EE Patterns

Now, let's see how you can leverage two of these core J2EE patterns -- Service Locator and Front Controller -- in Rational XDE.

Leveraging the Service Locator Pattern

J2EE Components use the JNDI (Java Naming and Directory Interface) to look up and create EJB and JMS components. A problem arises when many types of clients repeatedly use the JNDI service, and the JNDI code unnecessarily appears multiple times across these clients.

To address this problem, you can use the Service Locator pattern, which uses the Service Locator object to abstract all JNDI usage, thereby hiding all the complexities of initial context creation, EJB home object lookup, and EJB object re-creation. Clients simply reuse the Service Locator object to reduce code complexity and provide a single point of control.

Figure 8 shows how the Service Locator pattern is used in the context of a client (e.g., a servlet or JavaBean) and a service provider (e.g., an EJB).

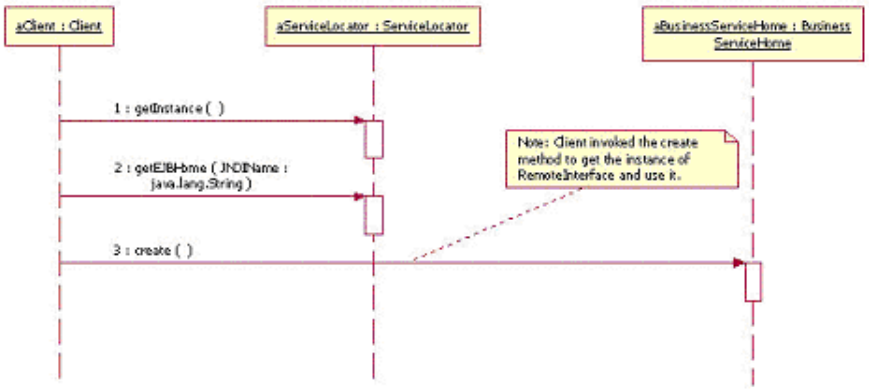


Figure 8: Service Locator Pattern Usage

Below is the simplified code for locating an EJB when using the Service Locator pattern instead of directly looking up the EJB via JNDI lookup.

```
// locate the amortization EJB
AmortizationHome theHome;
try {
```

```
theHome =(AmortizationHome)sl.getEJBHome("Amort/AmortizationHome");
Amortization theRemote = theHome.create();
// call the remote method on the session bean
float totalInterest = theRemote.amortize(…….);
```

Leveraging the Front Controller Pattern

In a multi-tier system, if a user accesses the Presentation View directly without going through a centralized mechanism, several problems might occur:

- There is often duplicate code because the same resources are accessed by multiple clients.
- Content and view navigation might become intermingled.
- Distributed control might be more difficult to maintain because changes need to be made in numerous places.

For example, as shown in Figure 9, you could use an implementation approach that uses a servlet to process incoming requests, obtain the necessary data from EJBs or databases, and then build the appropriate dynamic response pages. Although this approach is simple, it does present the challenge of coupling between business logic, formatting, and control. It also becomes harder to understand and maintain such an implementation, as each iteration introduces more changes and greater complexity to support the increased functionality.

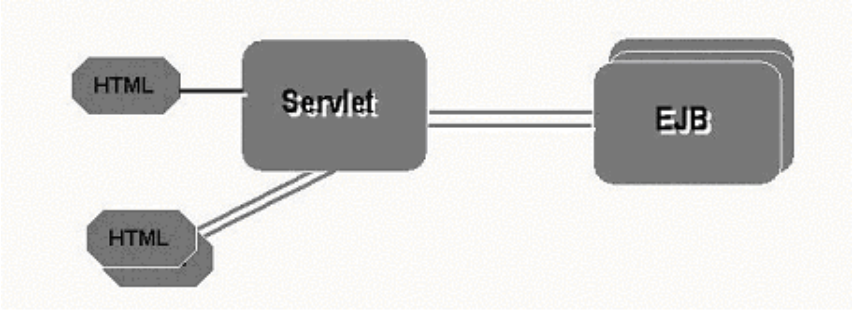


Figure 9: An Implementation Without the Front Controller Pattern

The Front Controller pattern addresses these issues by managing the request handling, including:

- Invoking security services (such as authentication and authorization).
- Delegating business processing.
- Managing the choice of an appropriate view, handling errors, and managing the selection of contact creation strategies.

The pattern introduces a few additional classes but makes the system easier to manage and evolve over the long term. In the Rational XDE implementation of the Front Controller pattern, a servlet that acts as the controller is responsible for receiving and initial processing of all incoming requests. As it receives the requests, the controller determines what use case the request corresponds to. Then, based on that information, it requests the HelperFactory to instantiate the helper bean associated with that use case and passes on the request to that bean.

Each helper bean is responsible for handling all aspects of a specific use case. The loan management application, for example, has helper beans for the ShowAmortization use

case, MakePayment use case, and so on.

Each helper bean implements the IHelper interface and also the `processRequest()` method, which forwards requests to the helper beans. When the `processRequest()` method is invoked on a helper bean, it can work with other entities in the system, such as EJBs, to collect the required data and then initiate the appropriate view handler (e.g., JSP) to format and present the results to the user.

Figure 10 shows a static view of the Front Controller pattern implementation. Note the different helper beans instantiated by the HelperFactory.

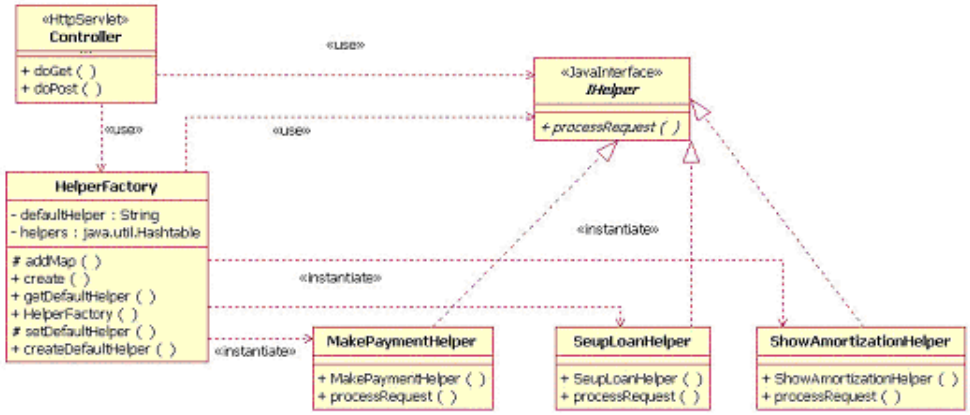


Figure 10: Front Controller Pattern Applied to the Loan Management Application

Implementation Details

Of course, the coolest part about the Rational XDE J2EE patterns is that the implementation for each pattern is quite complete; all you need to provide is the business logic specific to your application. In this section, we will look at the specifics you would need for the Rational XDE Front Controller Pattern.

Establish the Relationship Between an Incoming Request and a Use Case. An incoming request (e.g., a form submission) must be easily identified with a specific use case. In the Front Controller pattern implementation we have been discussing, this information would be communicated via a hidden field in the form that is filled with the use-case name at form construction time. When the form is submitted, the controller simply parses the parameters to determine which use case is involved. A default use case function is usually set up to handle requests that do not explicitly identify a use case (see below).

```
<FORM Name="Form" METHOD="GET" ACTION="Controller">
        <p align="left">
          <b><i><u>Show Amortization</u></i></b>
        </p>
        <p>
          Enter the following information about the loan:
        </p>
        <p>
          Principal
          <input type="text" name="principal" size="8" value="250000">
          dollars
        </p>
        <p>
          Rate
          <input type="text" name="rate" size="8" value="10.0"> %
        </p>
```

```
        <p>
          Duration
          <input type="text" name="termInYears" size="3" value="30">
          years
        </p>
          <input type="hidden" name="UseCase" value="ShowAmortization">
        <p>
          <input ID="IDCancel" Name="Cancel" Type="reset" Value="Cancel">
          <input ID="IDOK" Name="OK" Type="submit" Value="OK">
        </p>
</FORM>
```

Request Handling by the Controller. There is no implementation work for the controller, as the pattern implementation creates the logic required for extracting the hidden parameter, requesting the appropriate associated helper bean and delegating the processing to the helper bean. Figure 11 shows the static relationship between the pages and the controller. The complete, auto-generated request handling code is shown below Figure 11.

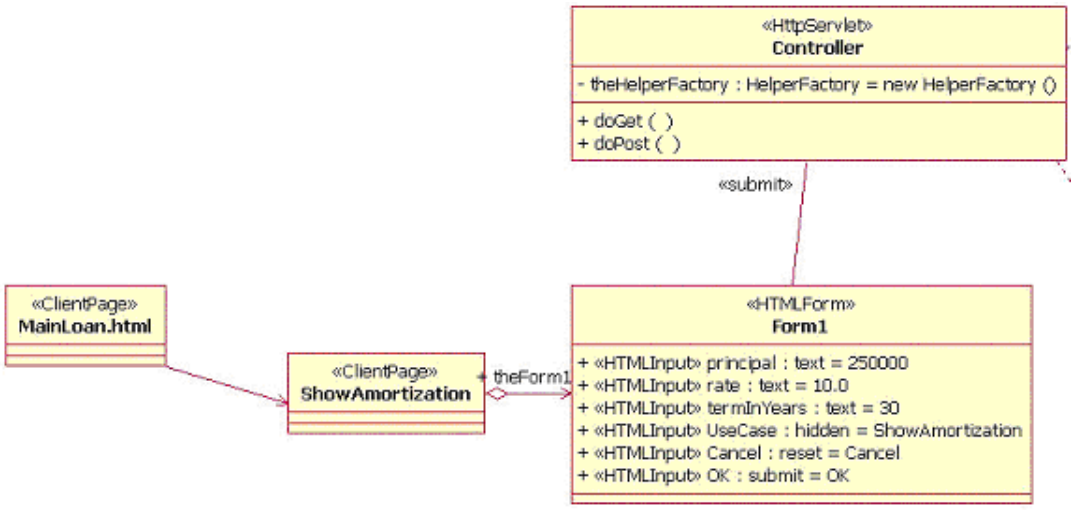


Figure 11: Relationship Between a Form and the Controller

```
// Check if we have a use case name passed on the URL.
  String useCase = httpRequest.getParameter("UseCase");
  IHelper ihelper = null;
  if (useCase != null)
  {
    // Dynamically instantiate a helper and call it.
    ihelper = (IHelper)theHelperFactory.create(useCase);
    if (ihelper == null) ihelper =
          (IHelper)theHelperFactory.createDefaultHelper();
  }
  else
  {
    ihelper = (IHelper)theHelperFactory.createDefaultHelper();
  }
  ihelper.processRequest(httpRequest, httpResponse);
```

Establishing Use-Case-to-Helper-Bean Mapping. You can establish use-case-to-helper-bean mapping by adding mapping information to the HelperFactory constructor at design time. Sample mapping for the ShowAmortization use case and the SetupLoan

use case is shown in the text box below. This code just needs to be placed in appropriate, pre-defined locations in the HelperFactory constructor at design time.

```
// establish ShowAmortization use case mapping to its helper class
usecasename = "ShowAmortization";
classname = "ShowAmortizationHelper";
addMap(usecasename, classname);

// establish SetupLoan use case mapping to its respective helper class
usecasename = "SetupLoan";
classname = "SetupLoanHelper";
addMap(usecasename, classname);
```

Helper Bean Implementation. Clearly, helper bean implementation will vary from one use case to another in this scenario. Generally speaking, however, a helper bean usually needs to extract the appropriate form parameters from the incoming request, coordinate additional information gathering or processing with other participants (e.g. session or entity beans), and then, finally, make the results of the processing available to others (e.g., a JSP) for display. A partial example of this process is shown below. Note that the XDE patterns engine eliminates the need to manually create each helper class. Simply select the "Generated Value" option in the wizard and enter the helper class names in the list. Figure 12 shows a snapshot of this wizard screen.

```
// extract the info from the request parameter
String sPrincipal = httpRequest.getParameter("principal");
…..
// locate the amortization EJB using the service locator
…....
// call the remote method on the session bean
float totalInterest = theRemote.amortize(principal, rate, termInYears);

// put data in the AmortValueBean
AmortValueBean avb = new AmortValueBean();
avb.setPrincipal(principal);
…..
// set the session attribute
httpRequest.getSession().setAttribute("avb", avb);
```

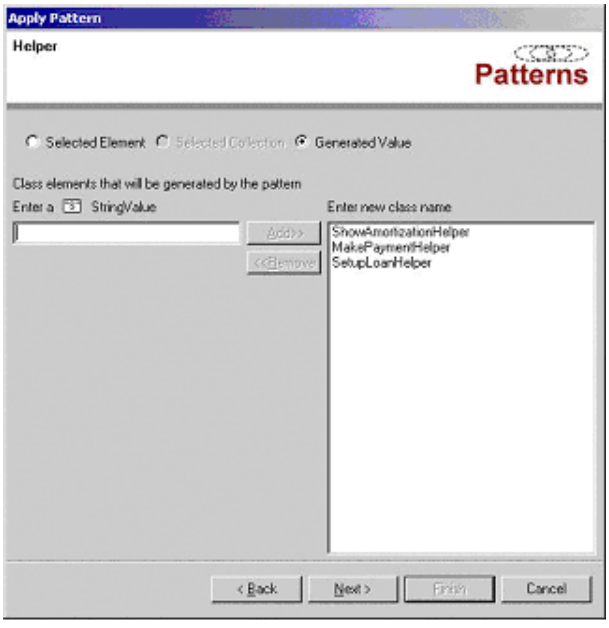


Figure 12: Automatically Creating the Basic Helper Beans

Displaying the Results. Again, various approaches will work for this. Typically, the data is made available to a JSP that is then responsible for formatting and presenting the information. Figure 13 shows the static relationships involved in one such setup. JSP source code for such an approach is shown below Figure 13.

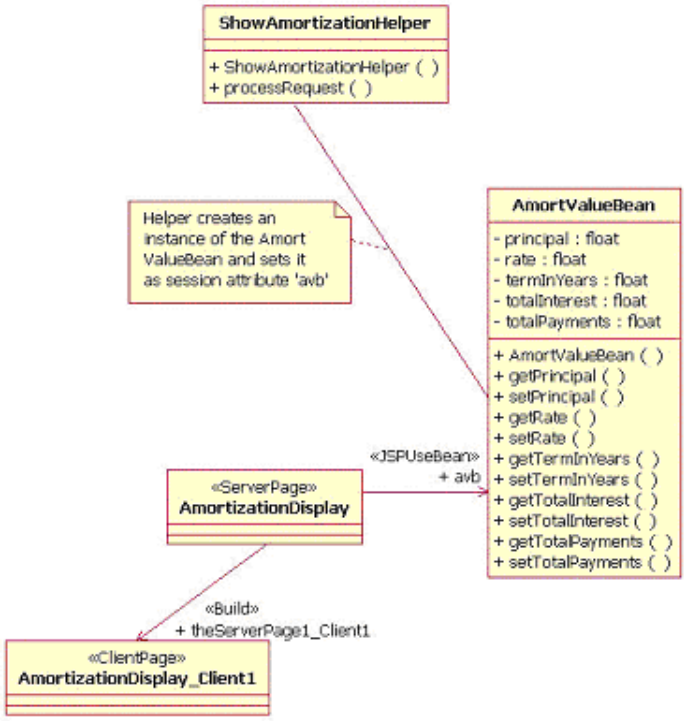


Figure 13: JSP Relationship with Participants Involved in Format/Display of Data

```
extracts data from the 'avb' bean instance and formats/displays it
<jsp:useBean class="AmortValueBean" id="avb" scope="session"/>
<b>
Results of your loan amortization request:
<br>
Pricipal entered: <%= avb.getPrincipal() %>
<br>
ýý.
Total Payments: <%= avb.getTotalPayments() %>
<br>
</b>
<p><a HREF="MainLoan.html">Back to Main Page</a>
```

Code Templates

Code templates are another Rational XDE capability that developers can leverage to greatly simplify their lives. They provide an efficient mechanism for:

- Reusing code snippets and algorithms.
- Sharing code-based solutions to specific problems to achieve uniformity.
- Creating standard headers/comments.

Rational XDE provides substantial support for code templates. In addition to an extensive, built-in infrastructure, it provides the ability to:

- Bind one or more code templates to a method.
- Unbind a code template as needed.
- Reorder the binding order to affect the code generated.
- Parameterize the code template.

In the context of our example, you could create a code template to establish the use-case-to-helper-class mapping. You could then bind the code template multiple times to the constructor and simply provide a different default value for the code template binding to customize the code generation. An example of this is shown in Figure 14.

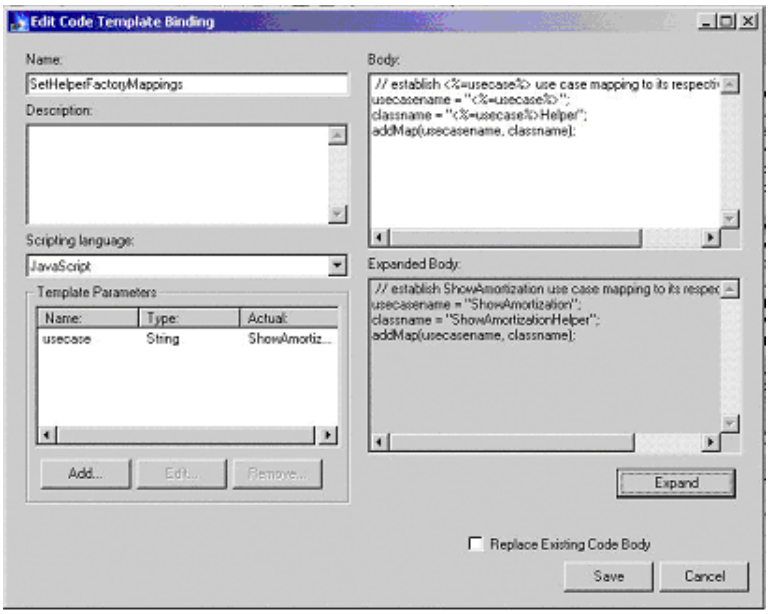


Figure 14: Code Template for Establishing Use-Case-to-Helper-Bean Mapping

Code templates can be used as part of a custom pattern, so you can create extensive application frameworks with Rational XDE. In fact, Rational XDE internally uses the code templates and patterns capabilities to deliver numerous product capabilities. For example, the code generation associated with getters and setters, design pattern implementations, and so on, all use the code templates capability!

End-to-End Scenario

Now that we have covered the design and implementation details for our sample application, let's recap how all the different pieces fit together. The next two sequence diagrams capture the end-to-end interaction between the different objects, including participants from the two pattern implementations.

Figure 15 illustrates the front end activities: The form is submitted to the controller, which then obtains the helper bean and requests it to handle the incoming request.

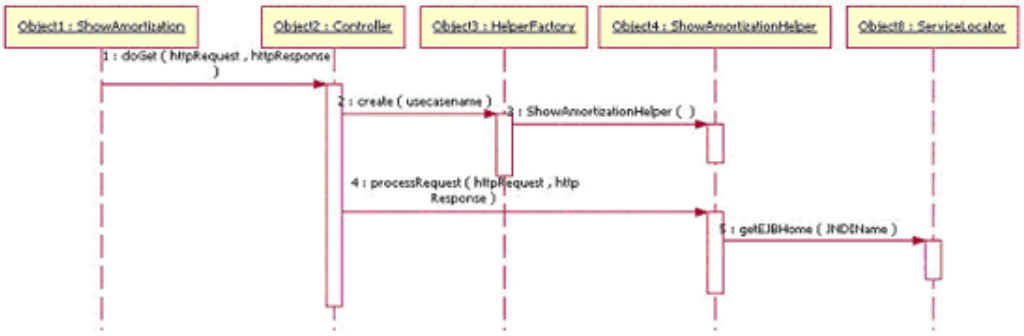


Figure 15: Form Submission and Handling by the Controller

Figure 16 is a continuation of the previous sequence diagram (with some overlap). It shows the various actions initiated by the helper bean, culminating in the forwarding of the request to the JSP responsible for formatting and presenting the computation results to the user. Note the use of the Service Locator pattern in obtaining the reference to the EJB required for processing.

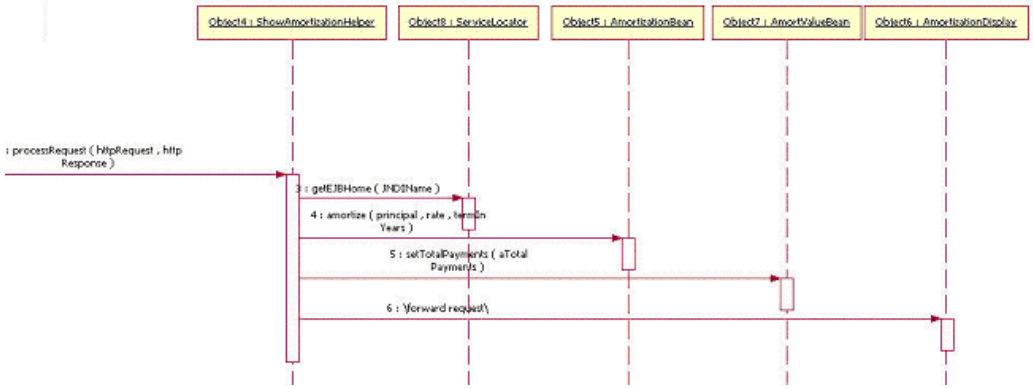


Figure 16: The Helper Bean Performs Processing and Presents Results

Patterns and Templates Simplify and Speed J2EE Development

I hope this discussion has conveyed some of the benefits Rational XDE's advanced capabilities offer to J2EE projects. The core J2EE patterns and code templates within Rational XDE provide an easy way to leverage proven design solutions while speeding up product development. Additional Rational XDE capabilities -- such as auto synchronization and the ability to create custom patterns and code templates -- liberate developers from tedious, repetitive tasks and further accelerate development.

References

Khawar Ahmed and Cary Umrysh, *Developing Enterprise Java Applications With J2EE and UML*. Addison-Wesley, 2001.

Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

The Rational Unified Process, *Guidelines: Analysis Class* section.

Deepak Alur, John Crupi, and Dan Malks, *Core J2EE Patterns*. Prentice-Hall, 2001.

Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

Notes

[1] For more information about these patterns, see Deepak Alur, John Crupi, and Dan Malks, *Core J2EE Patterns*. Prentice-Hall, 2001.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!