SDN Community Contribution

(This is not an official SAP document.)

Disclaimer & Liability Notice

Exception Handling In the ESA World

Applies To:

WebAS 6.40 SP10, Java stack.

Summary

This is the first in a series of articles focusing on some of the specific design issues we faced during the development of our Java-based Custom Composite Application (CCA). These articles are rather tightly focused and are not meant to address the bigger issues of ESA and how to architect a CCA. This article addresses how to design exception handling in an ESA world.

By: Richard Andrulis

Company: SAP America

Date: 23 Sep 2004

Exception Handling

Here we are dealing with how to support exception handling in an ESA compliant manner where we need to provide both human-user friendly messages as well as more technical information (for use in logs, etc). Java has a very robust exception handling functionality, but most of the messages raised in a Java exception are of the type:

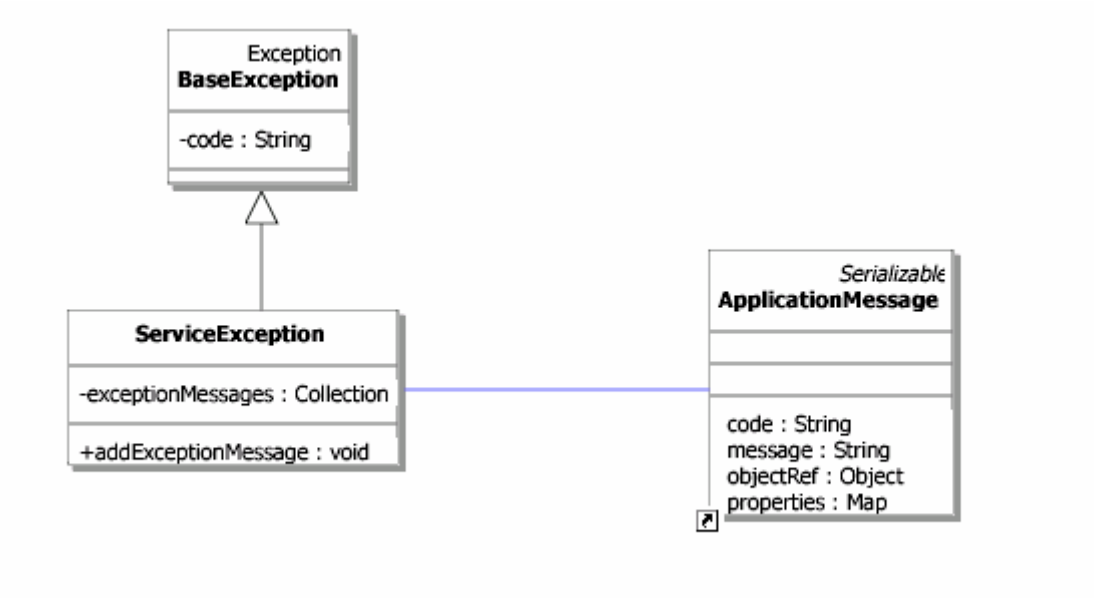
"Exception occurred in class, com.sap…., method, validate. Invalid value for x."

Which can be really helpful for the developer when debugging, but makes an end user go,

"What the heck does that mean?"

So, how do you provide a more human-user friendly message while not sacrificing the technical details that a developer/administrator needs? This is part of a bigger question of how to handle common presentation logic. By all principles of good design, presentation logic should be in the client application, but you don't want to have to repeat yourself for multiple clients -- you really want to only write the code once. So what to do? Either way you violate a design principle.

Our solution, at least for exception handling, was to provide a flexible way to provide all the information the client application may need, but not try to dictate to the client what they had to do. Our basic class design was something like the following:



We have a base exception which called *BaseException* and various children, an example of which is a *ServiceException*. Each exception contains an error code which is a String and a collection of *ApplicationMessage* objects. For example:

```
catch (NamingException ne)
{
      String key = MessageConstants.UTILS_NAMING_EXCEPTION_ERROR;
      ApplicationMessage appMsg = new ErrorMessage(key);
      throw new ServiceException(key, appMsg , ne);
}
```

Or, a more detailed example below. (Here we try to look up the roles for a user in the UME based on a specified userName). We pass the user Display Name and Id so they are available for variable replacement if desired by the client.

```
catch (Exception exception)
{
      // Perform Logging
      logger.errorT(exception.toString());

      if (user != null)
      {
```

```
            message = new ErrorMessage(
            ErrorConstants.UME_GET_USER_ROLES, null, new
            Object[]{user.getDisplayName(), user.getUniqueID()} );
      }
      else
      {
            message = new ErrorMessage(
                  MessageConstants.UME_GET_USER_ROLES, null,new
                  Object[]{new String(""), userName } );
      }
      // Create a new UME Exception
      throw new ServiceException( MessageConstants.UME_GET_USER_ROLES,
                              message, exception);
}
```

This would display a message like:

Problems while retrieving the roles of user, Richard Andrulis (andrulis). Please contact System Administrator.

The *ApplicationMessage* class is for more specific errors or problems. For example, if the user wants to change five items and on three of the items we receive exceptions, we will create one *ServiceException* and three *ApplicationMessage* objects. On the *ServiceException*, we will set the general code and the exception that we had. On each *ApplicationMessage* object we will set the error code, a message that describes the error, the name of the object that caused the error (the actual item), and, if needed, we can add some additional properties. The file *MessageConstants* maintains every module error code for reference.

The client then has the choice of using the exception message as is (usually a highly technical description of which class/method caused the error) or using the error code to look up a more human-user friendly message. These messages could even be used to support multiple languages using either the WebDynpro's message pools or a standard java property file, for example, MessageTexts_en.properties.

Note some limitations with this approach. This does not handle *RuntimeExceptions* or any Checked Exceptions that are not caught and repackaged as a *ServiceException*. There is also the issue of how to handle ABAP errors returned by a BAPI call. In this case we could easily repackage the BAPI return structure in our *ServiceException*. One could ask why bother given the well-developed capabilities of the ABAP runtime to provide formatted and localized messages, but we may still want the ability to pass multiple messages per exception, so we just need to add the message text to the *ApplicationMessage*.

Exception handling usually ranks as a pretty high priority for many experienced designers, but in a new ESA world, you have to be more flexible in your design because you don't know who may be working with the results you provide.

Author Bio

The author is a Development Architect for SAP Custom Development. He has worked for SAP for over 7 years starting in the IS-Aerospace and Defense group.