IBM WebSphere Commerce Suite Product Advisor

by J. J. Rofrano

This paper describes the Product Advisor component of the IBM WebSphere™ Commerce Suite. Product Advisor consists of a set of tools to be used by marketing personnel to generate, without programming help, Web shopping applications involving a product catalog. The user interfaces generated by these tools cover an extensible set of shopping paradigms including parametric search, a knowledge-based "virtual salesperson," and side-by-side comparison shopping. The paper also describes the Java™ infrastructure that supports the Product Advisor tools: servlets, JavaBeans™ and JavaServer Pages™ in the server, and applets in the client. Furthermore, it describes the implementation techniques used for rendering dynamic data: a data container hierarchy, a data type hierarchy, and presentation beans.

When trying to sell products over the World Wide Web, companies face a unique problem. How do they sell to customers who are unfamiliar with their products? How do customers who know the product features they want, find the particular product that will meet their needs among the hundreds offered? In a physical storefront, a salesperson is available to help both novice and knowledgeable customers, but on the Web, there are no salespersons; only knowledgeable customers have a real chance of finding the right product. WebSphere* Commerce Suite[1] Product Advisor provides tools that e-commerce sites can use to support "feature-relevant" searches and a "virtual salesperson" to help customers find the products that meet their needs.

Another problem, often overlooked, is "How do merchants maintain their dynamic Web sites?" As Web technology advances, its complexity increases. JavaServer Pages** (JSP**),[2] used for rendering dynamic output to client browsers, requires some programming skills. But the decision to place data on a page is often made by marketing people trying to elicit a particular customer behavior. The problem is to implement new technology in a way that will allow marketing people, with no programming skills, to take full advantage of the technology. There are many tools for programmers and many tools for Web page designers, but very few tools for marketing people. This paper describes how Product Advisor provides a set of tools for marketing people to record "meta-data" about what is to be displayed on a page, and a run-time environment that interprets the meta-data and dynamically renders the correct content.

The IBM WebSphere Commerce Suite, a member of the WebSphere family of products, is an application built on top of the WebSphere Application Server Advanced Edition. It provides the tools used to build e-commerce Web sites and is based on server-side Java**[3] and JSP technologies. JSP pages are HyperText Markup Language (HTML) pages that contain special HTML tags for declaring and using JavaBeans**.[4] They can also contain scripting elements based on the Java programming language, which will be executed at the server before rendering the page on the client browser. Server-side Java programming allows Java code to run at the server

where it has greater bandwidth for access to back-end resources such as databases.

Owners of e-commerce Web sites are constantly looking for ways to make their Web sites more dynamic and compelling and, at the same time, easier for customers to navigate. Navigating a Web site by following hyperlinks can be tedious and may often lead to "dead ends." Search mechanisms, such as keyword search, often lead either to zero hits or hundreds of hits with varying degrees of relevance. In the case of many hits, it is up to the customer to sort out these results, which can be unproductive and frustrating at times.

Product Advisor is a suite of tools within the IBM WebSphere Commerce Suite that allows merchants to create e-commerce Web sites with advanced product search and display capabilities. There are currently three tools in the suite:

1. *Product Exploration Builder* provides parametric search capabilities that allow customers to specify the product features and the corresponding values they are interested in, so they can quickly narrow down the product search space. Product Exploration Builder keeps a nonempty set of products in the search space and thus never yields zero hits.
2. *Sales Assistance Builder* is a knowledge-based tool that uses the salesperson's knowledge to create a "virtual salesperson." The resulting user interface guides customers to the products that are right for them through a question-and-answer session.
3. *Product Comparison Builder* is used to create a user interface that provides dynamic side-by-side product comparisons. It also serves as a complementary component for displaying the results of the other two search techniques.

Writing server-side Java code requires programming skills. JSP pages are usually created by Web designers, in much the same way as HTML pages are created. In addition, Web designers can place special bean tags on the JSP page that can be used to render dynamic data using JavaBeans. Here dynamic data are any data that are extracted from a database and used to fill an area on a Web page. The Web designer may need to also add some Java scripting code to the page in order to iterate over the data returned by the JavaBean. While there are JSP layout tools like WebSphere Studio that will automatically generate much of this code, Web designers still need to understand the structure and type of data being returned in order to iterate over it and format it properly for presentation. Product Advisor bridges the gap between the skills needed for laying out Web pages and those needed for Java scripting.

The remainder of this paper is organized as follows. The next section describes the tools that enable a marketing person to create the user interface for parametric search and question-driven product selection. The following section describes the Java infrastructure and the implementation techniques that enable Product Advisor to work without the need to include custom-developed Java code. The paper ends with a conclusion.

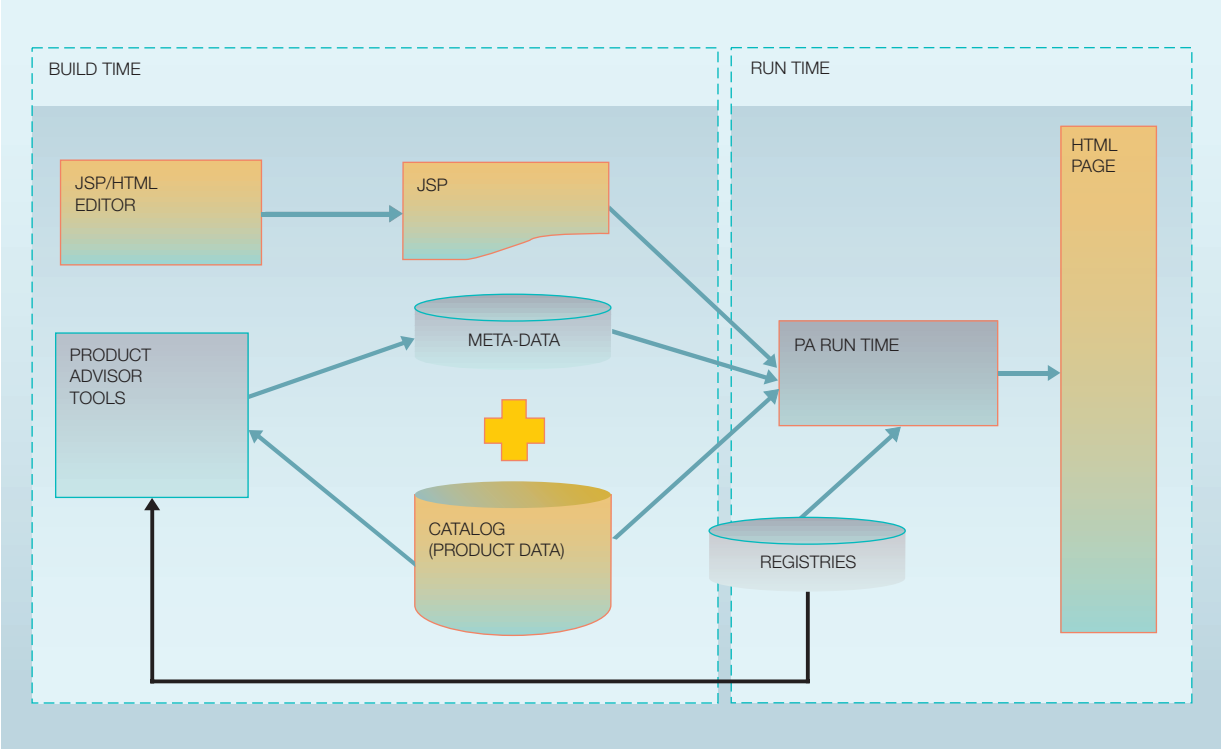
Tools for building e-commerce Web sites

The way customers shop depends on the specific need they're trying to satisfy and their knowledge of a product. Customers familiar with a product are usually looking for particular features, while others need sales advice to help them select the right product. To attract and satisfy the broadest range of customers possible, e-commerce Web sites need to support different shopping methods for different types of shoppers. Product Advisor provides tools that combine the marketing knowledge of a merchant's salesperson, the technical knowledge of the product specialist, and the rich set of product features that are necessary for a customer to make an informed buying decision.

When developing an e-commerce Web site, persons of varying skills are needed. Among these are operations people—the traditional IT (information technology) staff that maintains the site and the product data in the database. There are Web design people—the traditional graphics design staff whose role is to design the Web pages. There are also marketing people who understand the products and how to sell them. Product Advisor takes these different roles into account. It provides a set of tools for marketing people to use as a "marketing workbench," and a set of JavaBeans for Web designers to include dynamic content on Web pages. Traditionally, the person designing a Web catalog would need to work with a marketing person or product specialist to understand what content needs to appear on each page, then work with a programmer who helps implement the dynamic content rendering to the client.

Because the WebSphere Commerce Suite is a Web application, the client is always a browser. Java ap-

Figure 1 Product Advisor high-level data flow



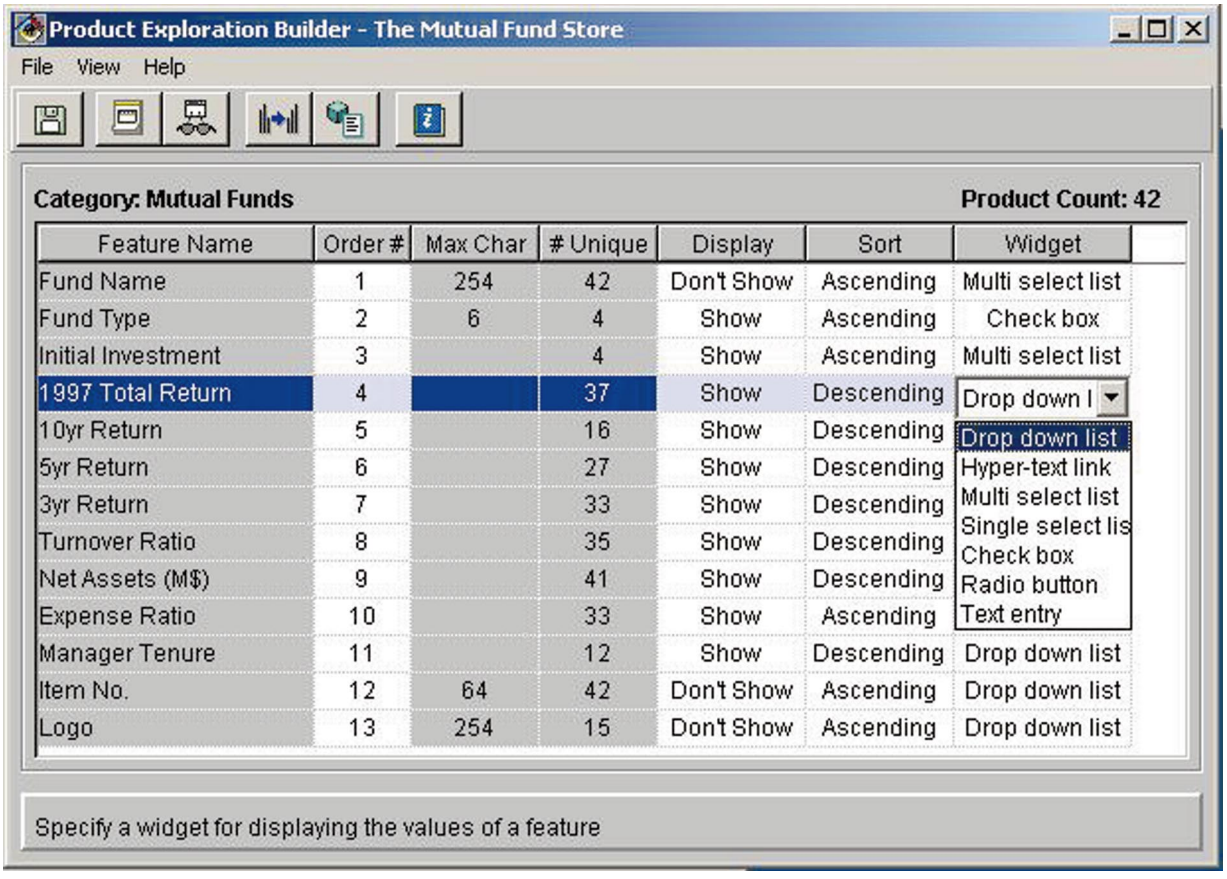
plets technology was selected to create the user interface for Product Advisor tools. Applets allow code versioning to be controlled from the server, avoiding client installation headaches. They also provide a much richer graphical interface than just HTML forms. These applets have the "look and feel" of the windowing system for their respective platform and allow the application to be accessed from any workstation with just a Web browser. The applets communicate with the server component, which is based on Java servlets. Java Database Connectivity (JDBC**) is used to manipulate relational data in the commerce database.

The time during which the Product Advisor tools are used to build the Web site is known as the *build time*. The time when the customer visits a site is the *run time*. A high-level data flow for Product Advisor can be seen in Figure 1. WebSphere Commerce Studio Page Designer tool (seen at the top left of the figure, labeled JSP/HTML Editor) is used to create JSP and HTML pages. The registries are a collection of database tables used both at build time and run time.

There are three registries defined in the Product Advisor:

1. *Data Type Registry*—Holds the list of extensible data types. These data types are an extension of the Java wrapper classes for the primitive data types. These data types are needed for the purpose of rendering the Web page and are discussed in the next section. New data types can be added to this registry as needed.
2. *Data Entry Widget Registry*—Holds the list of GUI (graphical user interface) controls that know how to render values for data entry on a JSP page. These include the default controls provided by HTML tags such as radio buttons, check boxes, drop-down lists, etc., as well as new controls implemented as Java applets, such as sliders and dials.
3. *Tool Registry*—Holds the list of tools installed in the suite. The tools that are registered by default are Product Exploration Builder, Sales Assistance Builder, and Product Comparison Builder. Developers can extend the product by adding new

Figure 2 Product Exploration Builder parametric search setup tool



tools and registering them in this table. They will then appear in the tools launch window.

The importance of the registry design is that it allows Product Advisor to be extended by adding new data types that know how to render themselves, new GUI controls for data entry, and new tools that create new shopping paradigms.

The Product Advisor tools are now illustrated in a case study taken from the mutual funds industry. The user interface of the Product Exploration Builder for building parametric search is shown in Figure 2. The tabular view has one row for each product attribute, the first column containing the attribute name. The columns represent meta-data.

The second column controls the display order of the product attributes. This allows the tool user to move the most important attributes to the top where customers will see them first. The columns in the middle are informational. The *# Unique* column shows how many unique values there are for each attribute. *Max Char* shows the number of characters needed for display.

The *Display* column turns the display of an attribute on and off. This controls what attributes are actually displayed on the parametric search page. Note that the upper right-hand corner of Figure 2 indicates there are 42 products in this category and the unique value column indicates there are 42 unique values for *Fund Name*. That means selecting Fund Name will narrow the search to one product. This may, or may not, be desirable. One might argue that if one knew the name of the mutual fund one wanted, there would be no need for a search at all. More importantly, if there were 10000 funds in this category, it may not be desirable to display 10000 fund names in a list on the JSP page. It is for these reasons that

this attribute has been turned off by selecting "Don't Show" as its display value. It will be shown later that a dynamic table will be generated from these values. Adding and removing columns in this table is as easy as turning these values from "Show" to "Don't Show." No programmer intervention is required.

The *Sort* column determines how the values are sorted for each attribute. The *Total Return* and *Net Assets* attributes are set to sort in descending order because "more is better" for these attributes. *Initial Investment* is set to be sorted in ascending order because "less is better" for this attribute. Finally, the *Widget* column controls which graphical control element will be used to display the actual data: multi-select list, single-select list, drop-down list, radio button, hypertext link, or check box, etc. These values, which were read from the data entry widget registry, can be extended by creating additional GUI controls and adding corresponding entries in the registry.

The parametric search user interface is shown in Figure 3. The product count (42) above the table represents the number of products in the current search space. The customer first selects an attribute value, say, the value *20%* for the *10yr Return* row. The customer then clicks on the ≥ button in the same row. Since all the buttons to the right of the attribute value fields are of the type SUBMIT, clicking any of these buttons causes the HTML form to be sent to the server for processing. The result of this processing is a new page containing the same form with values that reflect the new, and now reduced, search space. Thus, the product count drops to, say, 20.

The parametric search user interface is so designed that the product count stays always larger than zero. This is done in order to avoid unproductive searches, which may frustrate the customer. In the example above, assume that among the 20 products that make up the new search space there are no funds of type *Growth*. Then, on the page returned by the server there would be no check box marked *Growth* in the row for *Fund Type*, since a search for growth funds would then result in zero funds.

The Product Exploration Builder tool allows the marketing person to experiment with changes in the GUI controls. An interactive session results, in which there is immediate feedback as to what the parametric search page will look like to a customer. The tool lets merchandisers determine what is important and how the user interface is to be customized. The resulting data are part of the meta-data to be used at run time. It is thus possible to use a minimal set of JSP pages and generate from these a broad range of pages for each category of products.

Figure 3 Product Exploration parametric search customer view

What about customers who need more help in selecting a mutual fund? The Sales Assistance Builder tool in Figure 4 constructs a knowledge tree based on a set of questions and answers, as shown in the left window in the figure. A salesperson would start by adding the leading question to the knowledge tree. This is simply the first question asked if a customer says, "I am looking for a mutual fund." This would be followed by entering the possible answers that a customer could give to this question. Each answer is formulated as a product constraint. The pop-up window on the right side of the figure shows the dialog box that is presented to the salesperson using the tool when a product constraint is specified. These constraints will be added to the search criteria and applied to the product in a way analogous to the parametric search. The difference is that a salesperson is specifying the constraint based on the customer's answering a question. Each answer can have a follow-up question. This question will have a set of possible answers which, in turn, can add new product

Figure 4 Sales Assistance Builder tool

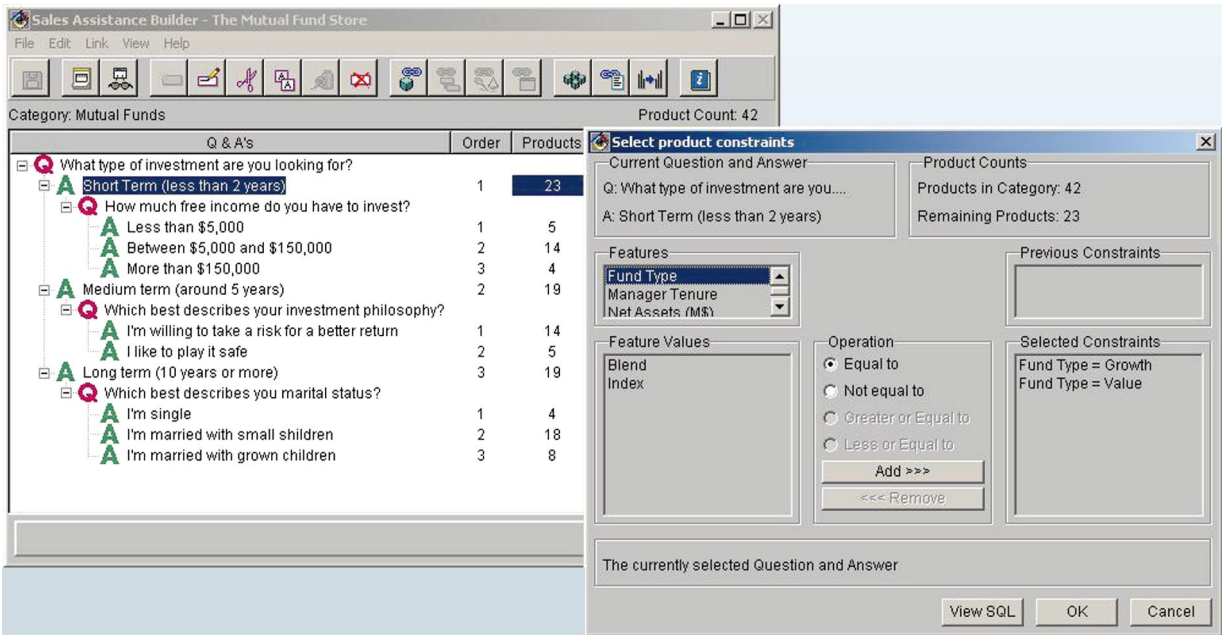
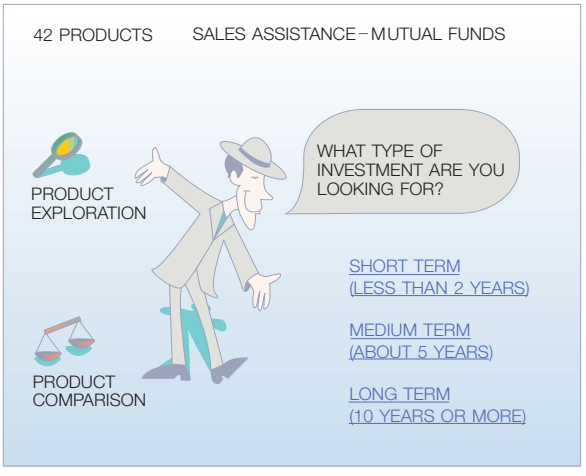


Figure 5 Sales Assistance "virtual salesperson" customer view



constraints and more questions. This builds a knowledge tree that enables a user interface in which customers are led to select a product via a question-and-answer session. The resulting user interface is shown in Figure 5.

Implementation techniques for rendering dynamic data

Figure 6 illustrates a typical HTML page to be rendered in a client browser for a Web application that includes a product catalog. As shown in Figure 6, the displayed page often contains an image of the product, a description, and possibly a table summarizing the main product attributes and their values.

Consider now a design that uses a single JSP page to display all products in a catalog. Moreover, consider a catalog that contains both refrigerators and washing machines. Because refrigerators have different specifications than washing machines, the pages to be displayed will differ in the number of attributes and their data type. As shown below, it is often advantageous to use a single JSP page for both products.

The Web designer laying out the page shown in Figure 6 would usually need to know the type of data. String data can simply be displayed, but numeric data have to be formatted correctly. If an image is involved, the designer would have to know and create an image link instead of displaying the image name. The designer would also need to know how to format the price data correctly. If the Web site is mul-

ticultural and allows customers to shop in several currencies, the Web designer would need to have information available on various currencies and how these are formatted.

The Web designer would also need to know something about the product specifications, such as the multiple name/value pairs that must be formatted into a table. The display of such a table would require Java scripting in order to iterate over the data.

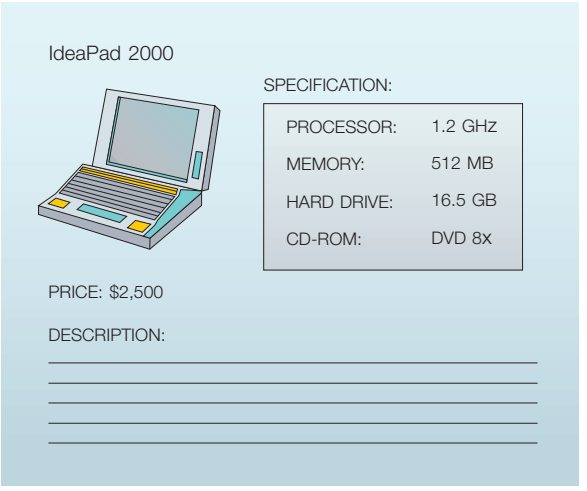
The Web designer could make certain assumptions and "hard code" some of these decisions in order to get the page to display properly. The page will work for all products, in a given category, that have identical product attribute specifications. Now let us suppose there are 200 categories of products in the catalog. Making 200 different pages to display the various products may not be desirable.

The Product Advisor implementation incorporates a solution that requires a single JSP page for displaying all products in the catalog. It provides a tool for tagging the data, a common set of data containers, presentation beans, and data types that understand how to render themselves properly. The tool for tagging the data can be seen in Figure 7. This tool allows the user to specify the data type for each product attribute. The attribute *Logo*, for example, is assigned a data type of *Image*. This information is stored as meta-data, and it is used at run time to determine how to display these data. When retrieved from the data, the information is really string data, but by using the meta-data, it will be determined that the string is actually an image name, and it will be assigned a data type that knows how to display it as an HTML image tag. By using this system of tagging data, the run time can correctly render any type of data by looking the tag up in the Data Type Registry and finding the data bean that represents the data type.

This approach requires a single JSP page in order to display products from any number of categories correctly, because the data type of their attributes can be easily determined and rendered correctly. The rendering scheme is composed of three components:

1. A data container hierarchy in which the data containers are self-describing. This includes both having a well-known interface, as well as optional meta-data that describe the contents of the container.

Figure 6 Typical product Web page

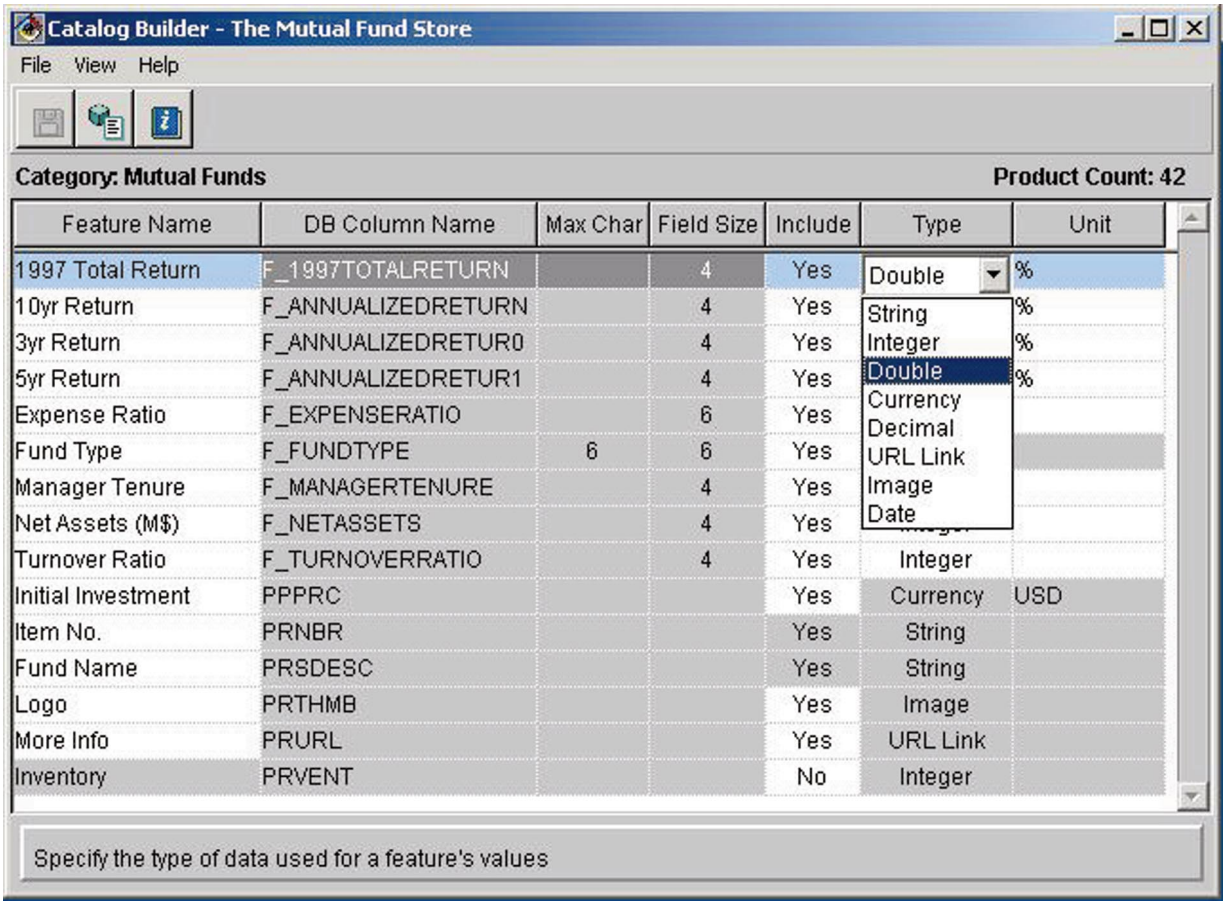


2. A data-typing hierarchy that extends the Java wrapper classes, such as Integer, String, Double, etc., and carries enough information so that the data beans can be rendered correctly. This information includes items such as locale, unit of measure, etc.
3. Presentation beans that understand how to interpret the data containers and require the typed data beans to render themselves.

This design emulates the javax.swing ListModel/JList, TableModel/JTable pairs where a data container (e.g., ListModel or TableModel) has a well-defined interface, and a presentation object (e.g., JList or JTable) understands how to manipulate the data container.

Data container hierarchy. The data container is a hierarchy of classes as shown in the Unified Modeling Language[5] (UML**) diagram in Figure 8. At the root of the hierarchy is an abstract class called DynamicDataBean from which all other data container subclasses are derived. These beans can have new data elements dynamically added to them. The ItemDataBean represents a single data item, whereas the ListDataBean, ColumnDataBean, and TableDataBean represent a collection of ItemDataBeans. Likewise, the NodeDataBean represents a single data node, and a TreeDataBean represents a collection of nodes. Notice also that a ListDataBean contains one or more ItemDataBeans, a TableDataBean contains a collection of either List-

Figure 7 Catalog Builder tool for tagging data

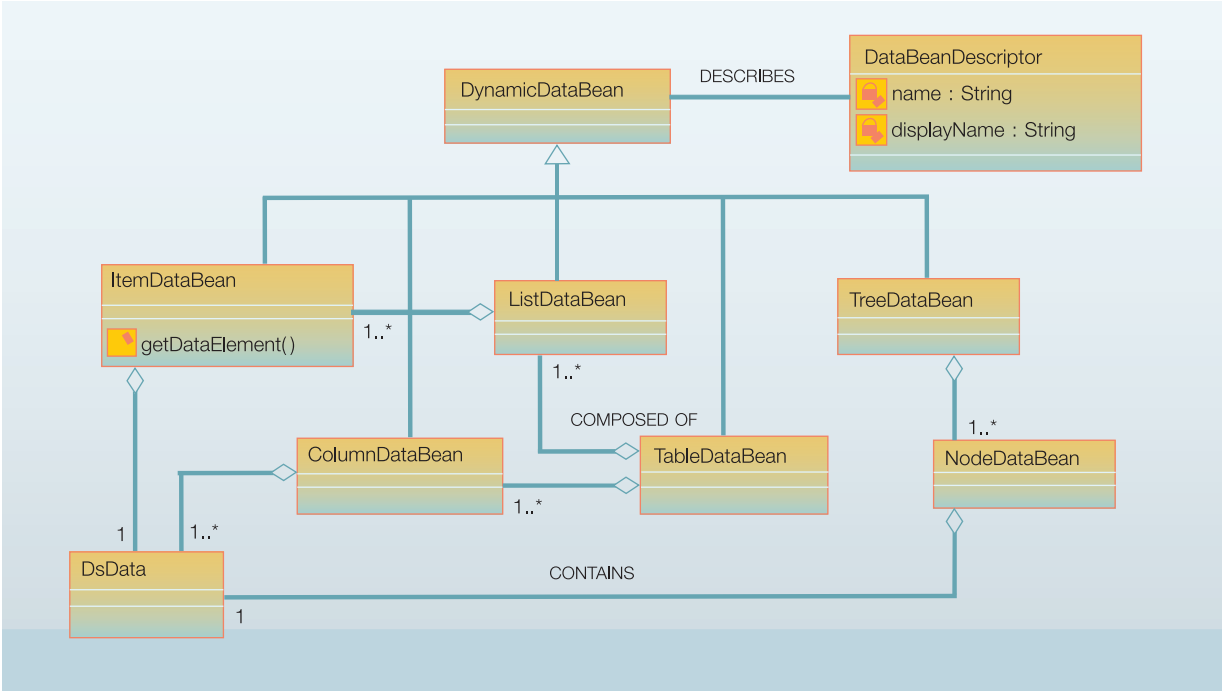


DataBeans or ColumnDataBeans, and a TreeDataBean contains a collection of NodeDataBeans.

Each of these containers has appropriate getXXX( ) methods to return the other containers or data objects they are composed of. For example, TableDataBean has getColumnAt(i), getColumn("name"), and getRowAt(i) methods, while ListDataBean has a getItemAt(i) method and ItemDataBean has a getDataElement( ) method to get the data. Only the ItemDataBean, ColumnDataBean, and NodeDataBean contain actual data objects. The data object they contain is an object derived from DsData. The DsData class hierarchy is a model of data types that know how to format themselves for rendering. The DsData class also has a getRawData( ) method to get at the actual unformatted data if this is needed.

The DynamicDataBean classes also contain a DataBeanDescriptor that is derived from the java.beans.FeatureDescriptor class. This allows the server-side code to store more information about the data that must be carried to the client. The typical use of the DataBeanDescriptor is to store the *name* and *displayName* for the data. The name is a token that will not change across locales, while the displayName may be in a language that matches the client's locale. The ItemDataBean, along with the DataBeanDescriptor, make up a name/value pair to be used in rendering. The advantage is that in addition to numerical formatting of the data, a name or label can be printed with it as well. As stated earlier, this is particularly helpful when making a table of product specifications because the names and values are all contained in the ItemDataBean object.

Figure 8 DynamicDataBean container hierarchy



The set of well-defined containers allows the presentation beans to understand the structure of the data much like a JTable understands a TableModel in Swing. The advantage of using containers over just sending a two-dimensional array of data to represent a table, is that the Web page designer would need to know which dimension represents the rows and which dimension represents the columns. A simple two-dimensional array would not carry any metadata about what the column names should be. Perhaps this would need to be sent separately or an agreement would be made between the client and server code that the first row contains the column names. Finally, if only a subset of columns is to be displayed, the Web designer would need to know the index of each column rather than addressing the columns by name (with getColumn("name")) as this model allows.

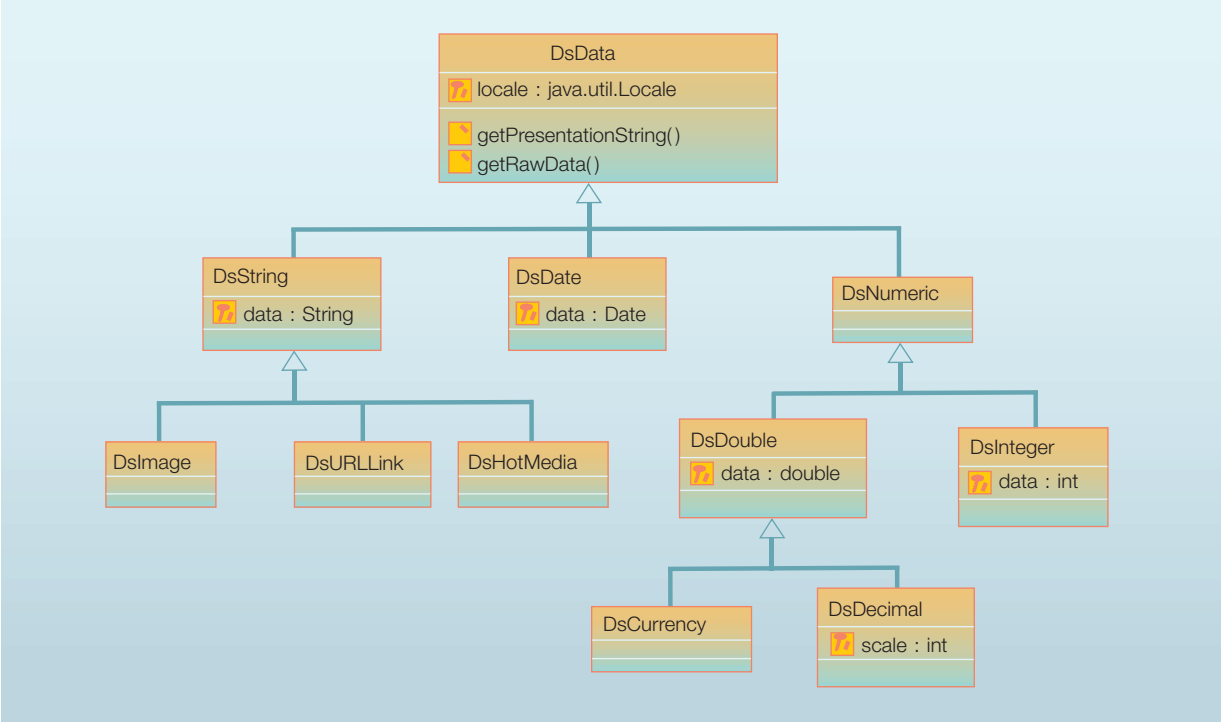
Data type hierarchy. The next component of the model deals with the data themselves. This is the data type hierarchy shown in Figure 9.

At the root of this data type hierarchy is the DsData class. This is an abstract base class from which all data objects are derived. It contains a java.util.Locale object that is inherited by all other classes. This is especially useful in a multicultural environment. At the client browser, only the client's locale is known. There may be times when it is necessary to know the locale at the location where the data were stored in order to format the data properly. Formatting currencies falls into this category. It is not enough to just send a decimal number from the server to the client.

Each DsData object has a method called getPresentationString( ) that will format the data correctly and return the data as a string for displaying. The model supports all of the primitive data types found in Java and some, like DsCurrency, that are not. This model can easily be extended by creating new subclasses for new data types.

One could argue that if all data were formatted correctly as strings when extracted from the database, this would be all that is required. While it does solve the display problem, this approach has a drawback. JSP pages can contain Java code. If there is a need to perform math on the data in the JSP page—for

Figure 9 Data type hierarchy



example, to determine if the price is less than a certain value—this would require parsing and stripping all the formatting from the string in order to extract the numeric value within it. It is not easy to parse numbers in a multicultural environment, where the comma and decimal point are used differently, without knowing the original locale of the data. It is better to leave the data in their original form and use getRawData( ) to get the actual data or getPresentationString( ) to format the data only when needed for display. This gives the Web designer the most flexibility.

Presentation beans. The data beans can be manipulated manually with Java code, but it was one of our design goals to have the data rendered automatically just by dropping a data bean and a presentation bean on a page. The classes that enable the automatic rendering are shown in the UML diagram in Figure 10.

The presentation beans are in a class hierarchy that mirrors the data container hierarchy of DynamicDataBeans. For each dynamic data bean there is a presentation bean that knows how to iterate over the data in that bean and render it correctly. A DynamicTable knows how to render a TableDataBean, a DynamicList bean knows how to render a DynamicListBean, etc. The advantage of the dynamic presentation framework is that the Java code to iterate over the data in a DynamicTableBean is written and tested in the DynamicTable presentation bean. Web designers can reuse this code by using the DynamicTable presentation bean anywhere they need a table of dynamic data to be rendered. It will set up the table headers by asking each ColumnDataBean for its displayName property. Then it will iterate over the rows and columns, creating the correct HTML table tags, asking each DsData object it encounters for its formatted string, and placing it in the appropriate cell of the table.

Putting it all together. All of the design elements discussed above are utilized by the JSP sample code in Figure 11. This figure actually shows two examples, one that does not use a presentation bean and

Figure 10 Presentation beans hierarchy

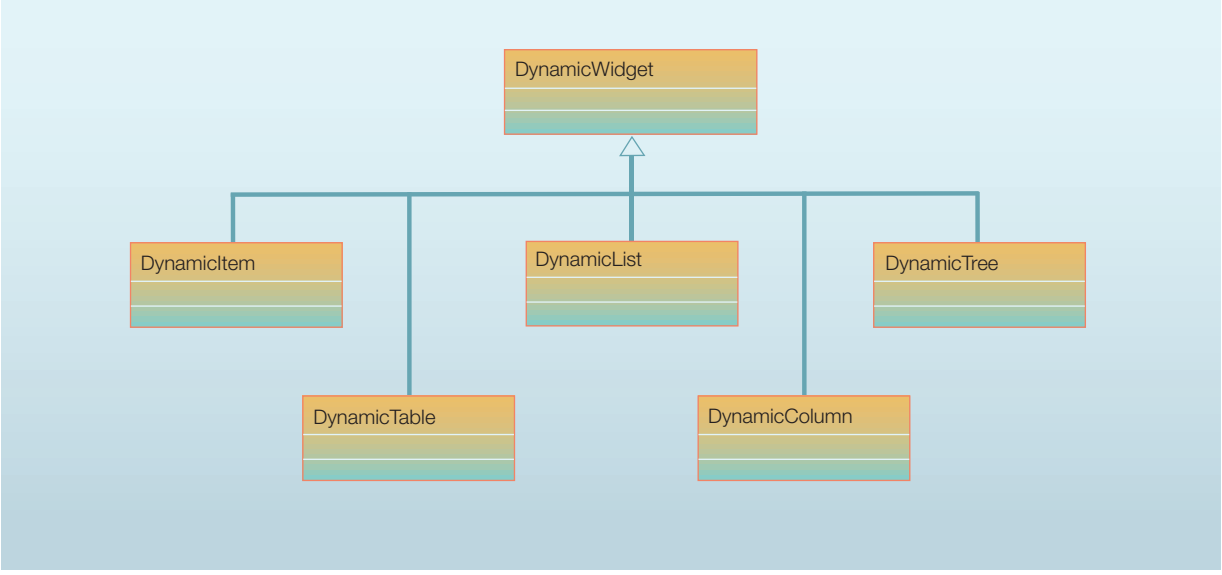


Figure 11 JSP sample code

```
<HTML>
<BODY>
<!-- Example One -->
<jsp:useBean id="productCount" class="com.ibm.commerce.beans.ProductCountDataBean"/>
<%
    com.ibm.commerce.beans.DataBeanManager.activate(productCount, request);
%>
<p><b>Funds: <%= productCount.getDataElement().getPresentationString() %></b>

<!-- Example Two -->
<jsp:useBean id="pcDS" class="com.ibm.commerce.beans.ProductCompareDataBean">
<jsp:setProperty  name="pcDS" property="*"/>
</jsp:useBean>

<TABLE CELLPADDING=5 CELLSPACING=2 BORDER=0>
<jsp:useBean id="pcTable" class="com.ibm.commerce.widget.DynamicTable">
<jsp:setProperty name="pcTable" property="dataBeanName" value="pcDS"/>
<jsp:setProperty name="pcTable" property="orientation" value="HORIZONTAL"/>
</jsp:useBean>
<%
    pcTable.execute(request, response, out);
%>
</TABLE>
</BODY>

</HTML>
```

LEGEND:
HTML TAG
JSP TAG
JAVA SCRIPTLET

Figure 12 Results of JSP sample code

Funds: 3

Item No.	27000-2	31000-3	32000-4
Fund Type	Value	Growth	Blend
Initial Investment	$1,000.00	$1,000.00	$1,000.00
1997 Total Return	36.1%	32.1%	50%
10yr Return		12.8%	18.5%
5yr Return	21%	15.8%	22.8%
3yr Return	32.6%	22.3%	32.5%
Expense Ratio	0.63	1.08	1.02
Manager Tenure	1	8	6
Net Assets (M$)	703	118	534
Turnover Ratio	131	146	125

one that does. The first JSP tag (i.e., jsp:useBean) in the figure is a ProductCountDataBean, which is derived from an ItemDataBean. It returns a single data element that represents the number of products in a category. The Java scriptlet call to DataBeanManager.activate( ) fills the bean with data. The next Java scriptlet calls the getPresentationString( ) method on the DsData object returned by productCount.getDataElement( ), and the resulting data are displayed.

We now look at how this rendering model simplified the Web designer's job. If the product count was 1000, this sample would correctly display 1,000 or 1.000, depending on the locale of the data. The Web designer did not need to do any additional work. To illustrate the advantage of this code over simply returning strings, consider the case when the Web designer wants to display a special message that suggests to the user to use the search page if there are more than 1000 products. Then a call to the productCount.getRawData( ) method returns an integer object and it is easy to determine if the count was over 1000 without having to parse already formatted data.

The second example in Figure 11 shows a data bean and a presentation bean working together. The second JSP tag in the figure contains a data bean called ProductCompareDataBean, which is derived from a TableDataBean. It returns a table of products and their attributes so that a side-by-side comparison can be displayed. The third JSP tag in the figure is a presentation bean called DynamicTable. The first property that is set is "dataBeanName" and its value is the identifier of the previous ProductCompareDataBean "pcDS." The second property that is set is "orientation" and its value is set to "HORIZONTAL." Finally, a Java scriptlet is added to tell the dynamic table to render itself with a call to pcTable.execute( ). The result is a table of formatted data complete with column headings taken from the *displayName* property in the DataBeanDescriptor of the ColumnDataBeans, which can be seen in Figure 12.

If the Web designer wants to flip the table on its side, this requires only a change in the "orientation" property of the DynamicTable from "horizontal" to "vertical," and the presentation bean would render rows as columns and columns as rows, thus turning the table on its side. This would have taken much longer using Java code because inner and outer loops would have to be reversed (and this assumes the Web designer would even recognize what a Java looping construct does). The DynamicTable also has the ability to paginate data by setting its *pageSize* property, which can save the Web designer considerable time when only a subset of a large data set is to be displayed.

Conclusion

The Product Advisor component of the IBM WebSphere Commerce Suite provides a set of tools for the product specialist or marketing person that allows control of the Web site dynamic content without the need for programming skills. At the same time, these tools provide advanced search capability, such as parametric search or question-and-answer session, with a "virtual salesperson." The implementation techniques that support this functionality include a Swing-inspired set of container-type object hierarchy, together with a set of HTML-aware presentation beans. An extension to the Java wrapper classes for primitive data types leads to multicultural data beans and thus enables the application to operate in a multiple-locale, multicultural environment.

Product Advisor has been successfully deployed in several customer Web sites. One such customer is a chain of retail stores in the United States that is attempting to consolidate an emerging e-business presence.

The Product Advisor team participated in the installation of the product by working with the company-designated Web designers. The newly designed Web

pages were obtained, and the Product Advisor team, with the cooperation of the customer product specialists, created and installed corresponding JSP pages following the process described earlier in this paper.

The launch was successful and preliminary data show a significant increase in Web site activity. It was learned, through informal discussions with the customer team, that the new Web site also had an indirect impact on sales with some customers who visited a store and made a purchase after having browsed the Web site. It appears that those customers used the Web site to educate themselves and make early buying decisions, which in turn benefited the retailer, because these customers spent less time in the store with sales personnel.

Acknowledgments

A product is only as good as the team that helped transform the concept into a reality. I would like to acknowledge the team that worked with me on Product Advisor. In alphabetical order they are: Peter Becker, Arthur Greef, Darko Hrelic, Hasib Jamal, Deborah Kalantari, Mohammad Khan, Galina Kofman, Agnes Krechko, Ian Kupfer, Alan Lampert, Jianren Li, Martin Maldonado, Joel Mumper, Lorraine Remza, Tom Schields, Ning Yan, and Kevin Zhang. A special acknowledgment and thanks to Bob Zitelli, without whom I could not have held the whole thing together.

Cited references

1. IBM WebSphere Commerce Suite fundamentals, IBM Corporation, http://www.ibm.com/software/webservers/commerce/wcs_pro/lit-tech-general.html.
2. JavaServer Pages Specification Version 1.0, May 28, 1999, Sun Microsystems, http://java.sun.com/products/jsp/download.html.
3. Java 2 Platform, Standard Edition Documentation, Version 1.3, Sun Microsystems, http://java.sun.com/j2se/index.html.
4. JavaBeans Specification Version 1.1, April 1999, Sun Microsystems, http://java.sun.com/products/javabeans/software/index.html.
5. Unified Modeling Language, Version 1.1, Sept. 1, 1997, Rational Software Corporation, http://www.rational.com/uml/resources/documentation/index.jsp.

Accepted for publication September 22, 2000.

John J. Rofrano *IBM Software Group, 17 Skyline Drive, Hawthorne, New York 10532 (electronic mail: rofrano@us.ibm.com).*

Mr. Rofrano is a Senior Technical Staff Member and one of the senior architects for the IBM WebSphere Commerce Suite product. He received his B.S. degree in computer science from Mercy College, New York, in 1984. That same year he joined IBM, where he has held various management and software development positions. Prior to his current position, he was the chief architect for Net.Commerce Product Advisor, and the chief architect for IBM Visual Warehouse.