documentum

Documentum Business Objects Framework

March 2003

Table of Content

1.0 Overview

The Documentum Business Objects Framework (BOF) provides an environment for efficient development of business applications by abstracting content management functionality into the business layer. It provides an object oriented framework that supports extensibility and reusability of server side business logic components, leveraging standard J2EE and .NET environments. The interfaces are defined at a high level for easy application development with reusable & extensible business objects providing abstraction layer between applications and the Documentum repository. Analysis of applications shows that most customers can reuse application components when writing content rich applications.

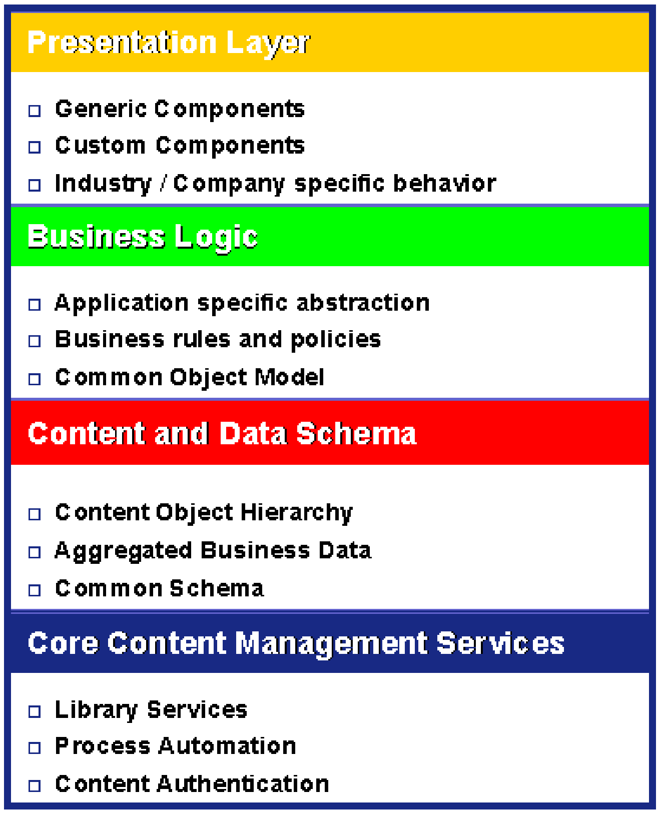
BOF has been designed to provide developers with an infrastructure for building such content rich business applications like Catalog, Strategic sourcing, Contract management, etc. and application components. It allows implementers and providers to assemble interoperable applications quickly by using Business Objects (BO) and Service Objects (SO) metaphor. The main benefits of such a framework are:

- Development partners can take advantage of BOF to quickly integrate their offerings in a predictable, uniform and maintainable fashion thereby reducing barriers to partnership with best of breed vendors in the market.
- Consulting partners and customers can leverage such BO as best practice object schemas, utilities, and applied security models to reduce cost of implementation and ownership to a network of subscribers like resellers, retailers, and distributors.
- Enterprises can connect the business processes for efficiency with the interoperable applications.

This white paper describes the BOF principles and shows how Documentum Business Objects can be used.

1.1 Makeup of a typical content-rich application

A typical *content rich* application has four major logical areas and they can be identified as shown below:



The *presentation layer* is used to display content and structured information in the web browser or end user application. In a Web environment, Documentum WDK would be the choice to present the layout and also manage the flow and sequence of user interactions. The presentation layer consists of generic and custom UI components that allow information to be displayed in different ways depending on such factors as the user role or the privileges. This layer is, for most implementations, highly customized and has to be very flexible.

The *business logic layer* is completely independent of visualization and of the toolkit used to display information. It provides the application specific abstraction layer and implements the customer specific business rules and policies. It is based on a common object model that allows the use of components in different application areas. In most cases, implementation specific business logic is tied to the given company's policies and procedures and it not very likely to change a lot. On the other hand, if a procedure does change, the implementer can just modify the corresponding business logic component and all applications are automatically changed to the new policy.

The *content and data schema layer* defines the content object hierarchy and defines the schema used to store the aggregated business data. It is hidden from the presentation layer by the business logic layer, which fully depends on how data and content is organized and is therefore tightly coupled to the Docbase object model. There are common themes within application categories and similar requirements across customers and industries. Therefore, there is a need to reuse complete schemas or in a slightly modified form.

Finally, there is the *core content management services layer* that provides, for example, the library services, process automation and content authentication to the business objects layer. This layer is implemented by the Documentum eContent Server. It provides the basic building blocks to build higher-level business logic components.

Documentum Business Objects are designed to provide modular business logic to the presentation layer by hiding the underlying Content Repository (Docbase) schema and by using the core services of the Documentum Content Server.

1.2 Example Business Scenarios

Following are some examples how BOF will enables e-businesses meet the demand:

1. **RFQ Generation**

   Business Environment: Customer needs to develop a sourcing application to manage internally generated RFQs and include trading partners in the RFQ generation processes.

   Drivers: Increase efficiency and profits by giving partners the capability to build their application logic into a client-independent layer that will allow them to reuse the business logic by exposing it through several web-based client applications. They will not be faced with the issue of customizations not migrating between versions of the client, reducing costs by removing gross inefficiencies.

2. **Standard Service**

   Business Environment: Systems integrator would like to build a set of reusable application components that would form the core of a service offering. They currently develop some of the same logic and cutomizations repeatedly for different clients. They would like to build this set of components in the MS development environments that they are most familiar with: VB and C++.

   Drivers: Increase efficiencies by growing this set of components over time and utilizing them as the business logic layer for standard Documentum clients and custom clients. Save significant development time to get started with a framework, examples and documentation that would make developing these components easy.

3. **Contracts Management**

Business Environment: Independent software vendor would like to integrate to the Documentum Content Repository (Docbase) for use with their contract management application. They have realized that there is significant additional business logic associated with this integration that should reside in a layer between their application and the Documentum server. They would like to develop this logic in Java and have it reside in the middle tier, and run on a J2EE application server.

Drivers: Save significant development time to get started with a framework, examples and documentation that would make developing these components easy.

4. **Collaborative Services**

   Business Environment: Documentum customers would like to customize WebTop in which a user can initiate or participate in a discussion thread that represents a single discussion topic. This should also enable discussion of any object inside the Content Repository (Docbase).

   Drivers: Increase efficiencies by quickly building custom application utilizing the business logic layer that is primarily type based. This service stack provides functionality that can be applied to any type of document and, therefore, should be interoperable with other stacks. For example a discussion can be launched for contracts.

5. **Catalog Management**

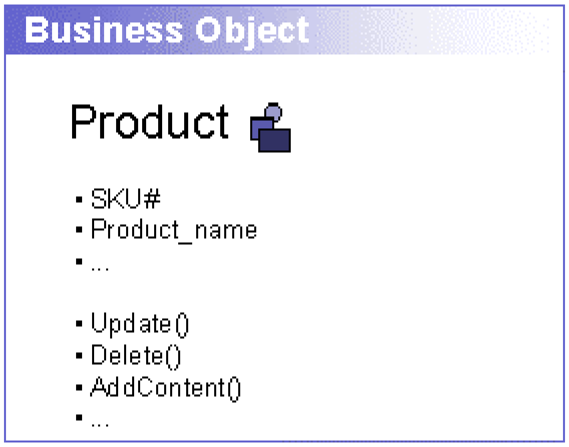
   Business Environment: Company wants to develop a web-based catalog application to represent product categories, products and product contents. The application should be able to manipulate the Business Object (Catalog) by adding, modifying, deleting, linking, and creating a collection of objects.

   Drivers: Increase efficiency and accuracy by utilizing already defined business objects and services.

2.0 Business Object Fundamentals

2.1 Business Objects

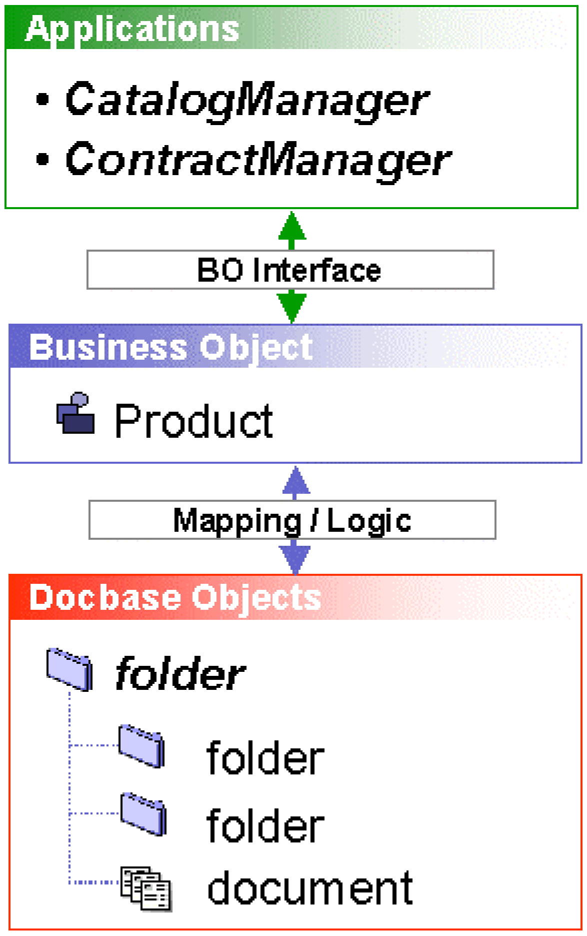
Business objects are abstract entities that have a type or a class and properties (attributes), and are able to have unstructured content associated to them. For example, a catalog application would likely include a business object of type “product”.



In this example the Product business object has a number of attributes such as product name, SKU (stock keeping unit) etc., and a number of methods specialized for the handling of product objects. For example we can update a product using method "Update". This method aware of the business rules specific to Product. For example it knows where this object is located in the Content Repository (Docbase), if the product name is unique, and if a link is required to the manufacturer of the product. Similarly to remove a product, the "delete" method is used. This product specific method first checks if the product is no longer used in an active catalog and then removes all the contents associated with this product.

2.2 Abstraction Layers

Let's assume we are going to write a custom catalog manager application. If we start from scratch, we would first build a number of business objects. For example, we would create a business object of type "Product".



This product object is bound to a Documentum object type “cm_product” and is therefore type bound. “cm_product” is a sub-type of dm_folder in the Content Repository (Docbase) and contains all components of a product that can include documents (pictures, sound, specifications, pricing etc.) and other folders that serve as links to other business objects like “vendor” or “manufacturer.”

For a developer using the product business object, all details except methods such as ‘create’, are hidden. The Content Repository (Docbase) storage schema of the product is governed internally by the BO and not exposed.

The business objects provide a clearly defined high level interface at a level the developer of the catalog application can understand. For example a developer of a catalog application knows what a product is and how it is used in a catalog. It is not really necessary to understand the implementation details such as how the data is represented in the Docbase.

Once the product business object is created it may also be used in other applications such as a contract management application or it can even be incorporated into other high level business objects.

2.3 Type based vs. service based

The previous example of a product business object is called a type based business object. A type business object is tightly linked to an object type in the Docbase. TBOs are essentially extensions of basic DFC types such as IDfPersitentObject, IDfSysObject, IDfDocument, IDfFolder, or any other class that inherently extends IDfPersistentObject. They allow a developer to encapsulate some business logic that is specific to a particular object type. There are two main reasons for creating TBO:

1. Providing a new behavior for existing or new object types. For instance, a developer may decide to add new get and set methods for object specific attributes or add new methods like addProduct() to a Catalog TBO.

2. Customizing low level operations to enforce data validations, referential integrity and object specific business rules. For instance, a developer may decide to override the save() and checkin() methods to be able to validate data before saving them in the persistent storage.

Service based Business Objects (SBO) are functional components that are not bound or related to a specific Docbase object type. They are components that may operate on multiple object types, retrieve different objects types unrelated to Documentum objects (e.g. external email message), and typically perform some processing.

An example of a service-based object is a Documentum IInbox object that retrieves items from a user inbox and performs several operations, such as removing and forwarding items. Such a business object is not tied to a specific Docbase object type, therefore it may operate on different types of objects.

For readers familiar with Enterprise Java Beans (EJB), a Service based Business Object is similar in concept to a Session Bean whereas a Type based Busines Object is more imilar to an Entity Bean.

3.0 Business Object Framework

3.1 Overview

A framework has been developed by Documentum in order to implement Documentum business objects. This framework facilitates the development and provides a runtime environment for such a metaphor. The Documentum business object registry (DBOR) is provided for enabling developers to both, browse a catalog of existing business objects and to extend, modify or replace existing business logic to meet specific customer requirements. The framework also provides factory mechanism that instantiates these business objects using information stored in the DBOR for Java, EJB or .NET environments.

3.2 DBOR

Design time environment

DBOR (Documentum business object registry) is a registry that stores all information required to manage Documentum business objects. This information is stored in a platform neutral format and is managed in a central repository during design time.

The information stored in the DBOR includes:

Business object name or type: This is a unique identifier for every type or class.

Docbase type binding: For type based business objects a Documentum object type has to be specified.

Java class binding: For every business object, Java class has to be specified that implements the appropriate methods. This allows implementation specific modifications while maintaining the high-level method interface. The runtime environment then uses this information when Java classes are instantiated.

Runtime environment

When a business object package is deployed, all the information required to generate the package is extracted from the DBOR. For example when this is deployed in an EJB environment, all the required EJB interfaces and descriptor files are generated, and all the Java classes are packed into an .ear file. A runtime version of the DBOR is also generated that can be accessed through JNDI.

If this is deployed in a web services environment, all the Java classes are packed and the WSDL interface descriptions are generated. A runtime version of the DBOR is then deployed into a UDDI server.

The runtime version of the DBOR is generated from the information stored in the development DBOR and is deployed into the native environment in which the business objects will run. For example, DBOR is deployed in JNDI for an EJB environment. For debugging purposes a development time DBOR is provided as a service registry.

3.3 Business objects and DFC

The Business Object Framework is comprised of the DBOR and new DFC classes as described above. The new DFC classes include additional or enhanced factory classes that instantiate the business objects for runtime.

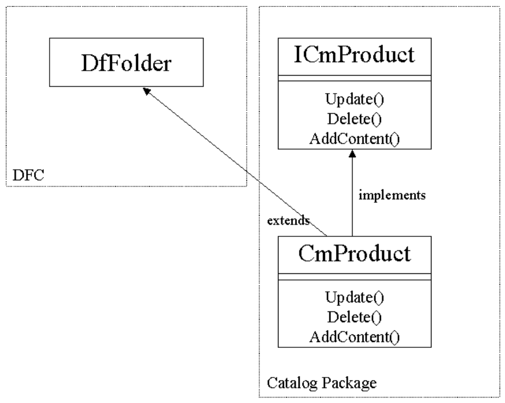
Developers use DFC to implement new business objects. As such, developers familiar with DFC should be able to implement new business objects without a steep learning curve. Also, the objects built using this framework are compatible with existing DFC based applications.

3.4 Type based business objects

Type based business objects are associated with a Documentum object type. One business object is associated with one Docbase type and vice versa. This dependency is configured in the DBOR. The Documentum object type can be a core document type (e.g. “dm_document”) or a specific type that is explicitly created for a specific business object.

To implement the business object, two Java classes are created. One class is an interface that defines all the methods that this business object implements. The second class implements the interface and extends the standard DFC Java class “DfPersistentObject” or sub-types of this class. Extending this DFC object provides backward compatibility with existing clients.

The following example shows how “product” example is implemented:



First, create or generate an interface ”ICmProduct” that defines the methods “update,” “delete” and “AddContent.” Then create or generate a Java class “CmProduct” which implements the interface and also extends “DfFolder”. Also, create dm_product as a sub-type of dm_folder in the Docbase.

An Example of Java code for the interface class “ICMProduct” is as follows:

```
public interface ICmProduct
{
      public void update(String name,String sku,…);
      public void delete();
      public void addContent(ICmContent content);
}
```

The update method takes all mandatory attributes such a name, sku as an argument. It saves this product information in the Docbase. Method delete just removes the current instance from the Docbase whereas addContent adds another business object of type “IcmContent” to the product.

An example of Java code from the implementation of the product business object “CmProduct” is as follows:

```
public class CmProduct extends DfFolder implements ICmProduct
{
      public void update(String name,String sku,…)
      {
          …
      }
      public void delete()
      {
          …
      }
      public void addContent(ICmContent content)
      {
          …
      }
}
```

Factory classes, as they exist in DFC, are invoked to instantiate a new “product” object. For example, to create a new instance of a product, DFC method “newObject” in IDfSession is used with business object type name as parameter. Sample code for a client using the product objects looks like this:

```
ICmProduct product =
(ICmProduct)session.newObject(“catalog_product”);
product.update(”foobar product”,”4711”,…);
product.addContent(picture1);
product.addContent(picture2);
```

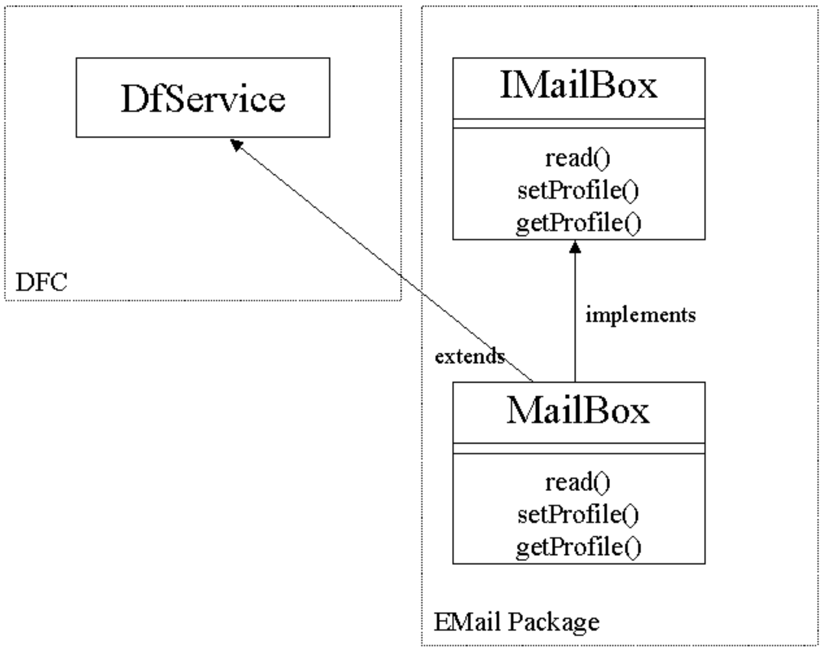
The business object name is used to lookup the implementation for a given type in DBOR. In this case the factory method instantiates “CmProduct” since this is the class configured in DBOR. Then update is used to set the required parameters and add content to the product that will look something like what has been shown in the pictures of the product.

3.5 Service based business objects

Service based business objects are not linked to a specific Documentum object type. They are “free floating” objects that provide a specific service that may operate on different types of business objects. Every service oriented business object is configured in the DBOR as well, but it is not associated with a Documentum object type.

To implement a service based business object, an interface has to be created that defines all methods of the given business object. Then a Java class that implements this interface has to be created or generated and extend a new abstract class in DFC, “DfService” in this case.

The following example shows the implementation of a mailbox service:



This service allows mail messages to be read from a pop3 server and stores the retrieved messages in the Docbase. We can specify a profile that defines the pop3 parameters (host, login etc.). The profile must be specified before the mailbox can be read. The read function returns a list of business objects that can be subtypes of “DfPersistentObject” of DFC. To process the individual objects retrieved from the mailbox, type based business objects will have to be used.

The following is sample Java code for the mailbox interface “IMailBox”:

```
public interface IMailBox
{
      public Iterator read();
      public void setProfile(IProfile profile);
      public IProfile getProfile();
}
```

The first method reads the mailbox and creates mail message objects in the Docbase. These objects could be specific type based business objects such as “IMailMessage” with attributes “recipient”, “subject” etc. The method returns a collection of type based business objects that could be mail messages or simple documents in case mail message has attachments. The objects also contains methods to set and get the profile that defines the pop3 parameters for the mail server.

Example java code for such implementation is as follows:

```
public class MailBox extends DfService implements ImailBox
{
      private IProfile profile = null;

      public Iterator read()
      {
            …
      }
      public void setProfile(IProfile profile)
      {
            this.profile = profile;
      }
      public IProfile getProfile()
      {
            return this.profile ;
      }
}
```

The client of this business object uses the new DFC method of class “IDfClient” to create an instance of the service based business object that implements a mailbox:

```
IMailBox mailbox =
(IMailBox)client.getService("pop3.mailbox",sessionManager);
mailbox.setProfile(myProfile);
Iterator newMsgs = mailbox.read();
while (newMsgs.hasNext())
      DoSomething((IMailMessage)newMsgs.next());
```

The factory method “getService” is part of IDfClient and returns a new instance of “MailBox” that has been configured in the runtime part of DBOR. The service factory method takes the DFC session as an argument. Then profile is set and it reads the messages from the pop3 server. The read method returns a collection of IMailMessage that can then be processed.

3.6 Session Manager

The Business Object Framework also provides a session manager that decouples Services from DFC session handles. This enables running of these services in a pooled DFC session environment without having to hold on to DFC sessions once an HTTP request complete. The DFC session can be used for other requests until the user submits the next request from the web browser.

Session manager also provides support for multiple identities and multiple Docbases. Once identities for specific Docbases are defined (i.e. for each Docbase login parameters are defined) the session manager is able to dispatch request to multiple Docbases transparently.

A principal support is also provided for a J2EE Application server environment. This allows user authentication to the J2EE framework as opposed to Docbase.

The session manager also coordinates transaction management for services. For example if calls to 3 different services have to be within one Docbase transaction the session manager can handle this transparently.