Abstract

This white paper provides a general overview of the Business Object Framework (BOF). It broadly outlines the fundamentals of BOF, types of BOF modules, the BOF module cache, and the process of packaging and deploying BOF modules. It describes common problems that developers encounter while using BOF modules and provides suggestions to avoid such problems. It also describes the process of implementing the Java Server methods as BOF modules.

July 2011
# Table of Contents

**Executive summary**........................................................................................................ 4

**Introduction to Business Object Framework (BOF)** ................................................ 4

**BOF modules and module types**.................................................................................. 5

- Modules, module registry, and module types ............................................................. 5
- Type-based Business Objects (TBOs)........................................................................ 6
  - Dynamic inheritance................................................................................................. 8
- Aspects............................................................................................................................ 9
  - DFC Aspect Model .................................................................................................. 9
- Simple Modules ........................................................................................................... 11
  - Implementing a Java Server method as a BOF module .......................................... 11
  - Service-based Business Objects (SBOs)................................................................. 12

**BOF Module Caching** .................................................................................................. 13

- Maintaining BOF module cache consistency............................................................ 14

**Packaging and Deploying BOF modules** .................................................................... 15

- BOF classloader hierarchy........................................................................................ 15
- Sandboxing................................................................................................................... 16

**Common BOF-related problems and resolutions** ..................................................... 17

**Debugging and Troubleshooting BOF Problems** ..................................................... 19

**Conclusion**.................................................................................................................. 20

**References**................................................................................................................ 20
Executive summary

This white paper provides a general overview of the Business Object Framework (BOF). It briefly describes the fundamental capabilities of BOF. It provides a basic understanding of BOF modules and the module registry. It also explains module types such as Type-based Business Objects (TBOs), Service-based Business Objects (SBOs), Aspects, and Simple Modules with useful tips to implement them. The paper also describes the process of implementing and deploying a Java Server method as a BOF module.

In addition, this paper describes the caching BOF modules in client systems and the mechanism that maintains the BOF module cache consistency. It also explains the sandboxing of BOF modules with some useful tips for packaging and deploying BOF modules. It describes common problems that developers encounter while using BOF modules, and their resolutions. It also provides some useful tips to debug and troubleshoot these common problems.

The developer must be familiar with the classloader hierarchy introduced by BOF as explained in this paper, to use the BOF functionality effectively. Developers can implement better customized solutions with fewer production issues if they know and understand Java classloading and BOF classloading concepts.

Audience

This white paper is intended for developers, architects, and production support and technical support engineers.

Introduction to Business Object Framework (BOF)

Documentum Business Object Framework (BOF) is an object-oriented framework for building, deploying, testing, and executing reusable business logic components known as Business Objects. Developers can use BOF that is built into Documentum Foundation Classes (DFC), to develop custom applications.

BOF offers the following capabilities:

- Customization of Documentum functionality in one of the following ways:
  - Extend core Documentum functionality
  - Add or modify existing Documentum types and their behavior
  - Add new attributes and behavior dynamically to individual object instances

- Centralization of Documentum functionality
  - Client independence

  BOF allows developers to install BOF modules containing business logic solutions directly on the repository, to allow DFC-based client applications to access them without explicitly adding the module JAR files locally. The underlying DFC framework ensures that the module JAR files are dynamically and transparently downloaded to the client system from the repository through
the dynamic delivery mechanism. This relieves the end user from explicitly copying custom solution JAR files to each client machine or the application before accessing them.

- **Hot deployment**
  This feature allows developers to upgrade the business logic implementation by installing the updated module directly into the repository without restarting the application server or the client application. The underlying DFC framework automatically detects the modules that have changed, and updates the client cache transparently, relieving the end-user from explicitly updating classes and JARs on the client systems.

- **Reusability**
  Developers can write the business logic implementation once, and any DFC-based application can reuse it. This feature reduces development time and effort.

- **Abstraction of the business logic from the presentation**
  - This feature allows two different teams to work in parallel. While one team implements the business logic, the second team works on the presentation, simultaneously. This promotes separation of concerns and allows the developer to build the solution quickly.
  - Developers can change the business logic and redeploy it without changing the presentation layer implementations.

**BOF modules and module types**

This section provides a basic understanding of BOF modules, module registry, and module types.

**Modules, module registry, and module types**

Customized business logic solutions built using the Business Objects Framework are packaged as modules and represented as dmc_module type (a subtype of dm_folder) objects in the repository.

Every repository contains the **Modules** folder under the **System** cabinet where the following subfolders, corresponding to the different module types are located:

- `/System/Modules/SBO` – contains Service-based Business Objects (SBOs)
- `/System/Modules/TBO` – contains Type-based Business Objects (TBOs)
- `/System/Modules/Aspect` – contains Aspects
- Subfolders under the `/System/Modules` folder can also contain other types of Simple Modules

The hierarchy of folders located in the `/System/Modules/` folder constitutes the module registry of the repository. **Figure 1** illustrates the repository organization.
When a new module is deployed in the repository, a folder of type dmc_module is created to represent the module. The implementation and interface JAR files of the module are linked directly to the dmc_module folder. The interface JAR files, on which the module depends, are linked to the External Interfaces subfolder. Additional documentation is available in the Miscellaneous subfolder of the dmc_module folder. Developers must ensure that the interface and implementation classes are placed in the respective JAR files.

BOF modules are of 4 types:

- Type-based Business Objects (TBOs)
- Service-based Business Objects (SBOs)
- Aspects
- Simple Module

The following sections describe each of the BOF module types.

**Type-based Business Objects (TBOs)**

A Type-based Business Object (TBO) is a type of BOF module used to add, modify, and extend the behavior of a persistent custom type.

For example, if a developer wants to add new attributes or change the behavior of an inbuilt dm_document type, the developer must perform the following tasks:

- Create a subtype (my_custom_type) of dm_document type and add new attributes to it.
- Create a module of TBO type to add or change the behavior of my_custom_type.
- The subtype and TBO module must have the same name.

- The TBO implementation class must implement the IDfDynamicInheritance interface and extend the DfDocument class.

- Optionally, although it is recommended, the TBO implementation class must implement the IDfBusinessObject interface (directly or by implementing the TBO interface that extends the IDfBusinessObject interface).

After a TBO is installed, DFC internally creates an instance of TBO implementation class every time clients request to access or create an object of my_custom_type. If DFC runtime fails to find a TBO with the same name as my_custom_type, DFC instantiates the inbuilt DfDocument class.

**Note:**
- Use the interface type to typecast the returned instance instead of using the implementation class.
- Change the behavior of an inbuilt type. However, if the business need requires that the default behavior of the inbuilt type must be overridden, the developer must ensure that the TBO implementation class created to override the default behavior of the inbuilt class does not implement the IDfDynamicInheritance interface.

**Tips for implementing a TBO**

- TBOs are created only for persistent object types to modify existing behavior or add new behavior.

- Developers can use a TBO to change the behavior of an inbuilt Documentum type, although it is not recommended.

- It is a good practice to define additional behavior using an interface rather than using concrete implementation classes.

- The TBO implementation class must extend the DFC type class associated with the super type of the custom repository type for which the TBO is created. For example, if the custom type extends dm_document, the corresponding TBO implementation class must extend the DfDocument class.

- Developers can add the custom business logic to the TBO implementation class by adding new methods and overriding parent class methods to perform some pre- or post-processing. However, developers must ensure that the parent class method is invoked after pre-processing or before post-processing in the overridden methods using the `super` keyword. If a developer fails to call the super method, the core functionality will be lost. Consequently, the user will experience unexpected behavior.

- Developers must always override methods that are prefixed by “do” such as `doSave()`, `doCheckin()`, `doCheckout()`, `doAddESignature()`, and so on.

- Note that TBOs are repository-specific.
**Dynamic inheritance**

Dynamic inheritance is a BOF mechanism that modifies the class inheritance of a TBO dynamically at runtime, driven by the hierarchical relationship of the associated repository objects. This mechanism enforces consistency between repository object hierarchy and the associated class hierarchy. It also allows developers to design polymorphic TBOs that inherit from different superclasses depending on runtime dynamic resolution of the class hierarchy. This functionality is achieved by ensuring that the TBO implementation class implements the IDfDynamicInheritance marker interface.

For example, assume that the repository contains two custom types, doctype_1 and doctype_2, where doctype_2 is the subtype of doctype_1, and doctype_1 is the subtype of the inbuilt dm_document type. In addition, assume that doctype_1 and doctype_2 have TBOs with the implementation classes called MyTBO_1 and MyTBO_2, respectively. If the TBO implementation classes implement the IDfDynamicInheritance interface, then both classes can extend the DfDocument class at design time, and DFC will dynamically build the MyTBO_2 implementation class such that it extends MyTBO_1 at runtime, instead of DfDocument. For more information, see the Documentum Foundation Classes Development Guide.

![Figure 2. Design time hierarchy](image)

For example, assume that the repository contains two custom types, doctype_1 and doctype_2, where doctype_2 is the subtype of doctype_1, and doctype_1 is the subtype of the inbuilt dm_document type. In addition, assume that doctype_1 and doctype_2 have TBOs with the implementation classes called MyTBO_1 and MyTBO_2, respectively. If the TBO implementation classes implement the IDfDynamicInheritance interface, then both classes can extend the DfDocument class at design time, and DFC will dynamically build the MyTBO_2 implementation class such that it extends MyTBO_1 at runtime, instead of DfDocument. For more information, see the Documentum Foundation Classes Development Guide.
Aspects

An aspect is a type of BOF module that allows the developer to add new attributes and behavior dynamically to individual instances, not to repository object types. The dmc_aspect_type object, a subtype of dmc_module, represents an aspect in the repository. An aspect is instance-based, and not type-based. In addition, it is late-bound, and not an early-bound object. Developers can attach or detach aspects from an instance dynamically, at runtime. A persistent object instance can have multiple uniquely named aspects. Different object instances of the same persistent type can have different sets of aspects attached to them. An aspect can be attached to multiple repository-object types. Aspect attributes can be single or repeating and of any supported data type. Fetching an object retrieves the basic and the aspect attributes. By default, aspects are enabled for dm_sysobject and its sub-types.

DFC Aspect Model

Assume that a repository contains a TBO with ICustomType and CustomType as interface and implementation classes respectively, for a custom type of dm_document called my_custom_type. An instance of the CustomType implementation class that extends DfDocument class and implements ICustomType interface is created when the my_custom_type object is accessed at runtime. Assume that the repository also contains two aspects, Aspect1 and Aspect2, each with the relevant defining interface and implementation classes. When these aspects are attached to the my_custom_type instance, its underlying implementation class is rebuilt dynamically with all the attached aspects that extend each other resulting in a compound object as illustrated in Figure 4.
Figure 4. Rebuilt implementation class with all attached aspects

Tips for implementing an aspect

- Define the interface and implementation classes for the aspect. Define new behaviors with getters and setters for aspect attributes.
- Attach or detach an aspect to a persistent object in the client program by first typecasting the object to IDfAspects type, and then invoking the attachAspect() or detachAspect() method.
- Create a new class that implements the IDfAttachAspectCallback or IDfDetachAspectCallback interfaces, and define the implementations for the doPostAttach() or doPostDetach() methods of the callback interfaces to use the aspects in a TBO or other aspects. Pass the instance of the new class as a parameter while attaching or detaching the aspect using the attachAspect() or detachAspect() method.
- Access aspect attributes using their fully qualified names, such as `aspect_name.attribute_name`, in all getters and setters. This ensures that conflicts do not occur even if multiple aspects (attached to an instance) define attributes with the same name.
- Maintain uniqueness in method names to avoid potential conflicts when two different aspects defining two identical methods or two methods with the same name and the same set of parameters but with different return values. You can achieve this by not using generic method names such as getName, getId, and including the aspect abbreviation in each method name.
- Avoid attaching an aspect more than once to an instance that can cause a DfDuplicateAspectException.
- A developer can define a default aspect for a particular type that is applicable to all instances of that type. Use the following command to attach a default aspect to a specific type, if the type has the r_aspect_name attribute and is not a lightweight object type:

```sql
ALTER TYPE type_name [SET|ADD | REMOVE] DEFAULT ASPECTS aspect_list
```
• Ensure that the referenced aspects are available in the target repository during object replication so that aspect attributes are replicated in the target repository like other basic attributes.

**Note:** If multiple aspects are attached to an object instance, the order of execution is random. Hence, developers must ensure that the logic used does not depend on the aspect order. However, if there is a business need to attach two aspects to one instance and execute their code in a sequence, the developer can achieve this by defining a dm_aspect_relation type instance.

**Simple Modules**

Simple Modules are a type of BOF modules that are similar to SBOs but are not repository-agnostic. Simple Modules have the following general characteristics:

• A Simple Module must be installed in each repository before it is accessed
• When Simple Modules are deployed in a repository, the corresponding dmc_module folders are created under the `/System/Modules` folder in the repository
• A Module implementation class must implement the IDfModule marker interface
• A Module is initialized by calling the newModule() method of the IDfClient interface

**Note:** The developer can use Simple Modules to implement repository methods such as those associated with workflows, document lifecycles, and Java Server methods.

**Implementing a Java Server method as a BOF module**

Developers are recommended to develop and deploy new Java Server methods as BOF modules. Java Server method implemented as a BOF module has the following advantages over traditional Java Server methods:

• Developers can package and deploy the Java Server method implementation in the same DAR or docapp as the dm_method definition.
• A Java method is self-contained. As a result, the client is not required to add the JAR file in the classpath or the dba or java_methods directory. It helps to avoid JAR hell by avoiding the `java_methods` directory.
• It enables the hot deployment of a method implementation, and allows access to the new implementation without restarting the Java Method Server.

**Tips to deploy the Java Server method as a BOF module**

• Define a module by implementing the IDfModule marker interface, directly.
• Ensure that the module implements the IDfMethod interface and not the IDmMethod interface.
• Insert the module name in the command_verb of the dm_method object, instead of class name.
• Ensure that the module is self-contained. As there is no visibility to classes in the java_methods directory, all the required material must be available in one or more BOF modules.

**Service-based Business Objects (SBOs)**

A Service-based Business Object (SBO) is a reusable business logic component containing methods that can be invoked directly by a program, rather than being invoked during a type-based action such as checkin or save. SBOs are used to create high-level, content management-related business objects that can be exposed as services to any application. SBOs are used to expose common functionality to several TBOs or applications. Developers can extract common functionality into a separate SBO. SBOs provide an additional advantage by allowing implementations of common functionalities to be introduced only once in an SBO, rather than in multiple TBOs.

SBO modules are installed on the global repository. They offer dynamic distribution from a single repository to multiple clients. DFC connects to the global repository using the properties defined in the dfc.properties file.

SBOs can operate on multiple object types. They can retrieve and process objects that are not related to Documentum objects, such as external email messages. An SBO can be converted into a web service. SBOs can be designed to be stateful as well as stateless. When an SBO is designed to be stateful, it can maintain the state between calls. However, owing to the ease of deployment on multithreaded and other environments, stateless SBOs are preferred.

**Tips for implementing an SBO**

• The SBO interface must always be coherent by offering only specific functionality that the SBO intends to provide.
• The SBO interface must extend the IDfService interface.
• The SBO implementation class must implement its defining interface and extend the DfService class.
• SBOs must override some of the methods of the DfService class such as supportsFeature(), getVendorString(), and so on.
• The getSession() method of the DfService class in an SBO method can be used to obtain a shared repository session. The newSession() method must be invoked on the SessionManager instance obtained using the getSessionManager method of the DfService class, to request a new private session.
• The session obtained from an SBO must always be released after usage.
• SBOs must be instantiated using the newService method of the IDfClient interface that takes a session manager instance as a parameter. The newService method searches the global registry for the SBO and instantiates the associated Java class.
• Ensure that the SBO name is unique to the global registry.
• Do not assign the SBO instance to a class variable. Instead, create an SBO instance every time it is required.
• Rely on DFC to cache the repository data rather than implementing a separate cache as part of the service implementation.
• If the SBO does not begin a transaction, do not commit or abort the transaction in the SBO method.
• Avoid using session-based transactions in an SBO method. If an SBO method contains session-based transactions, DFC will throw an exception. Instead, use the session manager-based transactions.

**Significance of the global registry in the context of SBOs**

Unlike a TBO, an aspect, or a simple module that is specific to its repository, an SBO is always registered in the global registry. The global registry refers the module registry of a single repository from where BOF delivers the SBO module JAR files to the clients. DFC uses the following properties in dfc.properties to connect to the global registry from DFC clients:

- dfc.globalregistry.repository
- dfc.globalregistry.username
- dfc.globalregistry.password

**BOF Module Caching**

A client machine refers to the machine running an application server with Documentum Webtop, Documentum Administrator (DA), or any DFC-based client application. DFC maintains a BOF cache where the interface and implementation JAR files of TBOs, SBOs, and other modules are downloaded and cached on the client machine. The location of the BOF cache on the client machine can be specified by setting the `dfc.cache.dir` property in the `dfc.properties` file. Its default value is the cache subdirectory of the directory indicated by the `dfc.data.dir` property in the `dfc.properties` file. The BOF cache is shared among all the applications that use the same DFC installation.

**Note:** A unique DFC installation is defined by a unique `dfc.properties` file. If two applications on the same machine use two different `dfc.properties` files but the same `dfc.jar` file, the two applications will be considered as separate DFC installations.
Figure 5. Structure of the BOF client cache (module cache)

The cached interface and implementation JAR files have the same names as the actual object IDs of the associated JAR files in the repository. The user can define the `dfc.bof.cache.append_name` property in the `dfc.properties` file to append the JAR name after the object ID while caching the module JAR files. These JAR files are transparently downloaded by DFC into a folder named after the repository. The same folder also contains the `content.xml` and `content.lck` files that are created by DFC internally. The `content.xml` file contains metadata information.

**Note:** DFC creates and maintains the `content.xml` and `content.lck` files. Do not modify these files manually. Delete the directory containing JAR files, and the `content.xml` and `content.lck` files to clean the BOF cache.

### Maintaining BOF module cache consistency

When a DFC client application accesses a BOF module, DFC downloads the corresponding module JAR files to the client machine, transparently. DFC does not download the JAR files until the module is updated in the repository. In addition, DFC does not verify cache consistency with the repository for each module request. Instead, DFC verifies cache consistency based on the value set for the `dfc.bof.cache.currency_check_interval` property defined in the `dfc.properties` file and the version stamp of the `_BOF_MasterChangeRecord` sysobject.

**Note:** Do not create, modify, or delete the `_BOF_MasterChangeRecord` sysobject because it is created and managed by DFC internally to optimize updates to the cached module JAR files.
The default value of the `dfc.bof.cache.currency_check_interval` property is 60 seconds. The user can modify the value based on the performance requirement. If DFC detects that a module or a JAR file of a module has been removed from the repository, it removes the relevant JAR files from the cache to ensure that there are no obsolete JAR files in the cache.

SBOs are installed in the global registry and DFC connects to the global registry based on global registry properties defined in the `dfc.properties` file. If DFC is unable to connect successfully to the global registry, it waits for an interval indicated by `dfc.globalregistry.connect_attempt_interval` in the `dfc.properties` file before attempting to connect again. The default value set for `dfc.globalregistry.connect_attempt_interval` is 60 seconds. This setting helps to control the access to the global registry and improves performance.

**Packaging and Deploying BOF modules**

A basic understanding of fundamental BOF concepts, such as BOF classloader hierarchy and sandboxing is essential to understand the process of packaging and deploying BOF modules.

**BOF classloader hierarchy**

Each BOF module (TBO, SBO, Aspect, or Simple Module) has its own module-specific classloader that loads the implementation JAR files contained in the module. Classes loaded in different module-specific classloaders cannot reference each other. Owing to the parent-last nature of the module-specific classloader, classes within the classloader take precedence over the same classes in their parent classloaders.

*Figure 6. BOF classloader hierarchy*

The shared BOF classloader is the immediate parent of all module-specific classloaders, and is shared across a DFC instance. All interface JAR files are loaded into the shared BOF classloader that allows published interfaces to be shared across
different modules. Due to the parent-first nature of shared BOF classloader, the classes loaded by its parent classloader take precedence over its own classes. The application/Webapp/system classloader is the immediate parent of the shared BOF classloader.

Sandboxing

Sandboxing enables different versions of the same JAR files to be concurrently used by different modules without any conflict. However, this benefit comes at the cost of increased memory requirements. The developer must define a java_library object to achieve the sandboxing feature and enable the sandbox flag. The java_library object allows developers to combine multiple third-party dependent JAR files into a single unit. If the sandboxing feature is not implemented, and if a custom application calls multiple modules that depend on different versions of third-party JARs, then the version of the third-party JARs that will be used depends on the runtime module request sequence.

DFC achieves sandboxing using two separate classloaders, one to load the interface and third-party JAR files (shared BOF classloader), and the other to load module-specific implementation classes. When a java_library object is sandboxed, all JAR files defined in the java_library object are loaded into the corresponding module-specific classloader and not in to the shared BOF classloader. Since the classes loaded in different module-specific classloaders cannot reference each another, different BOF modules can concurrently use different versions of the third-party JAR files without conflict.

Tips for packaging and deploying a BOF module

- Package the implementation and published interface classes in separate JAR files. It is a good practice to create implementation and published interface source files in separate directories to make the development easy and avoid any errors during the packaging process.
- Implementation JAR files consist of implementation classes containing the custom business logic of the application under development. These JAR files may also contain non-published interfaces such as interfaces that neither the client nor any other BOF module require.
- The interface JAR files contain all published interfaces that the clients need.
- Exception classes or factory classes that appear in the interfaces must be packaged in the interface JAR files. Ensure that the interfaces packaged in the interface JAR files do not transitively expose any classes that are not packaged with the interface JAR file, to avoid encountering the NoClassDefFoundError.
- Do not use mixed JAR files for any future development.
Common BOF-related problems and resolutions

ClassCastException

The ClassCastException is thrown by the Java Virtual Machine (JVM) when it tries to typecast an object to a different type. In Java, two classes are considered the same if they have the same fully qualified name and are loaded by the same classloader.

Incorrect packaging is one of the main reasons that cause the ClassCastException to occur during runtime usage of BOF modules.

For example, assume that the developer has incorrectly packaged the published interface with the implementation JAR file. As a result, the interface class is loaded in the module-specific classloader because of its parent-last nature. When the client application creates an instance of the BOF module and tries to cast it to the interface type, it uses the interface loaded by the client application classloader. In spite of having the same fully qualified name, the two interface classes cannot be considered same, because they are loaded by two different classloaders. In addition, the parent classloader cannot identify the classes loaded by its children classloaders. Hence, there is no way for the client application to refer to the interface class loaded by the module-specific classloader. Consequently, the ClassCastException is thrown.

Another scenario, in which a developer can encounter the ClassCastException, is when a developer does not set the Thread Context classloader to module classloader before making the request to third-party libraries such as jaxb, jax-ws, and so on, that rely on the Thread Context classloader to load their classes. The developer must set the Thread Context classloader to module classloader before making the request to any third-party library and perform the following steps to reset it:

1. Save the current ThreadContext Classloader:

   ```java
   ClassLoader oldContextClassLoader = Thread.currentThread().getContextClassLoader();
   ```

2. Use the try block to set the contextClassLoader as current classloader:

   ```java
   try {
       ClassLoader classClassLoader = this.getClass().getClassLoader();
       Thread.currentThread().setContextClassLoader(classClassLoader);
       call dfs service/jaxb
   }
   ```

3. Use the finally block to reset the classloader to the original contextClassLoader:

   ```java
   finally {
       Thread.currentThread().setContextClassLoader(oldContextClassLoader);
   }
   ```
Guidelines to avoid ClassCastException

- While preparing separate JAR files for the module interfaces and implementation classes, ensure that all published interfaces have been removed from the implementation JAR.
- It is not recommended to sandbox a library that contains an interface that is part of the method signature of a module, since it may result in a ClassCastException.
- Java libraries can contain interfaces, implementations, or both.
- Use the try, catch, and finally block to access third-party JAR files, if third-party classes use the Thread Context classloader to load classes.

ClassNotFoundException

The JVM throws ClassNotFoundException when it is unable to find the class in the classpath. The exception occurs because JVM is unable to find the class file in any JAR file associated with the BOF module.

In addition, the developer will encounter this exception in the following scenario:

If the developer has two modules that depend on a specific JAR file, however, while packaging the modules, the developer fails to add the JAR file to the second module. Consequently, the JAR file is found in the BOF cache, which is downloaded as part of the first module definition. The downloaded JAR file is not part of the second module, because it was not defined while packaging, although the JAR file is available in the BOF cache.

NoClassDefFoundError

Incorrect packaging can cause the NoClassDefFoundError to occur during runtime usage of BOF modules. This error can also occur in the following scenarios:

- The NoClassDefFoundError occurs when the developer transitively exposes a class that a classloader cannot find. This is relevant when the developer incorrectly packages an implementation class in an interface JAR file, and the implementation class references another class in an implementation JAR file. In this case, the class that is loaded in the shared BOF classloader cannot refer to the classes loaded by the module-specific classloader. As a result, the NoClassDefFoundError occurs.
- When an interface depends on an exception class included in the implementation JAR.
- When factory classes are defined in the interface JAR and the dependent interface is incorrectly included in the implementation JAR.

Problems related to hot deployment

Hot deployment allows users to access the new implementation without restarting the client JVM or application server. The BOF framework automatically detects the updated module, and transparently downloads the updated JAR files to the BOF cache.
**Note:** Hot deployment is applicable only to implementation classes and the associated JAR files. If a developer changes the interface classes by modifying a method signature or adding new methods, hot deployment is not supported. The client JVM must be restarted for such changes to take effect.

The shared BOF classloader that is shared across the DFC instance, loads the interface JAR files. Therefore, the shared BOF classloader must be destroyed and rebuilt and the JVM must be restarted for any updates in the interface JAR files to take effect. Alternatively, when the developer deploys new implementations for a BOF module, the module-specific classloader is destroyed and rebuilt by its parent classloader, which is the shared BOF classloader. Hence, hot deployment will work successfully only when the implementation JAR files are updated in the repository.

**Problems encountered while using aspects**

A developer can encounter a DfDuplicateAspectException while attaching an aspect to an object with which it is already associated. Developers are recommended to verify the aspects currently attached to the object, and then attach a new aspect using the following code:

```java
IDfList aspects = (IDfAspects) <obj_reference>).getAspects();
If (!aspect.findString("aspect_name"))
{
    (IDfAspects) <obj_reference>).attachAspect("aspect_name");
}
```

Attaching an aspect rebuilds the object’s implementation class automatically, to reflect the changes even before the changes are saved in the server by invoking the save() method on that object. The DFC session persistent cache is updated accordingly. If the user flushes the session cache without saving the object, the attached aspect is lost and this can cause unexpected behavior.

**Note:** Developers are recommended to save the object after attaching the aspect. If it is not feasible to save the object frequently due to performance considerations, developers must ensure that the persistent object cache maintained by DFC is not flushed as long as the aspect information associated with the object is required.

**Debugging and Troubleshooting BOF Problems**

Developers are highly recommended to understand the Java classloader functionality before developing and troubleshooting any BOF-related problem. After gaining the required knowledge about the java classloader, the developers must understand the working of the BOF framework. They must also understand how classloading strategies are employed for different JAR files associated with the BOF module. It is also very important to understand the BOF module packaging to identify the relevant classloader that is responsible for loading each individual class.
Set the following properties in the *dfc.properties* file to obtain more information for debugging and troubleshooting BOF issues:

```
dfc.tracing.enable=true
dfc.tracing.verbose=true
dfc.tracing.log.category[0]=com.documentum.fc.client.impl.bof
dfc.tracing.log.level[0]=DEBUG
dfc.bof.cache.append_name=true
```

## Conclusion

This paper has attempted to provide information about the effective usage of the BOF functionality. The information in the paper calls for an understanding of the basic Java class loading functionality. Developers must also understand the classloader hierarchy introduced by BOF as explained in this document. Knowledge of Java class loading and BOF class loading will help developers to develop customized solutions that have fewer production issues. For more information, see the *Documentum Foundation Classes Development Guide*.

## References

- EMC Documentum Foundation Classes Development Guide
- EMC Documentum Content Server Fundamentals