Code review
Custom Documentum application code review & best practices

EMC Proven Professional Knowledge Sharing 2009

Christopher Harper,
Senior Consultant
harper_christopher@emc.com
29 October 2009
Code review

Custom Documentum application code review & best practices

Christopher Harper
Senior Consultant
harper_christopher@emc.com
29 October 2009
# Table of Contents

DFC......................................................................................................................................... 5
Queries........................................................................................................................................ 5
  Collections.......................................................................................................................... 5
  Recursion.......................................................................................................................... 7
Query types ......................................................................................................................... 9
DQL queries....................................................................................................................... 10
Direct SQL ....................................................................................................................... 20
Get object.......................................................................................................................... 20
  Computed attributes ...................................................................................................... 20
  Convenience vs. performance ....................................................................................... 21
  Justifications ................................................................................................................ 21
  Null checks ................................................................................................................... 22
Piggybacking session ........................................................................................................ 22
BOF ....................................................................................................................................... 23
  TBO ............................................................................................................................... 23
  SBO ............................................................................................................................... 24
Methods............................................................................................................................... 24
  Method framework ....................................................................................................... 25
Utility classes..................................................................................................................... 28
  Example........................................................................................................................ 30
WDK....................................................................................................................................... 32
  Preconditions .................................................................................................................. 32
  Docbase object configuration ...................................................................................... 34
General ............................................................................................................................... 36
  DBA ............................................................................................................................... 36
Language ............................................................................................................................ 36
  Natural .......................................................................................................................... 36
Programming ..................................................................................................................... 38
Exception handling ............................................................................................................ 39
  Catching......................................................................................................................... 39
  Pass-through problem ................................................................................................. 39
  Method signatures ......................................................................................................... 40
  Unreadable code ........................................................................................................... 40
  Documentation burden ............................................................................................... 40
  Ideal situation................................................................................................................ 40
  Utilisation of constants ............................................................................................... 40
  Data caching ................................................................................................................... 41
    Candidates.................................................................................................................... 42
    Implementation ............................................................................................................ 42
  Custom object types ..................................................................................................... 44
  Design principles .......................................................................................................... 44
  Persistent object inheritance ......................................................................................... 46
  Relation vs. attribute ..................................................................................................... 47
  Aspects ............................................................................................................................ 48
  Value assistance ........................................................................................................... 49
  ACL Design ................................................................................................................... 49
  Duplicate identifier ........................................................................................................ 50
  Group membership ....................................................................................................... 51
Summary............................................................................................................................... 52
Biography............................................................................................................................ 53

Disclaimer: The views, processes or methodologies published in this compilation are those of the authors. They do not necessarily reflect EMC Corporation's views, processes, or methodologies.
A typical task of a technical Documentum consultant is to be that last point of contact when things are going or have already gone bad. Fire fighting is the more commonly used name for this at times daunting task that occasionally falls into our lap. Why do we even do fire fighting? Normally it is necessitated by a solution developed by someone who has no or limited knowledge of the system on which they have implemented the solution. Without fail this will cause issues both in the design of the application and the way that it is implemented. The task of fire fighting typically presents itself as heaps of documentation, one or more Documentum Foundation Classes (DFC) / Web Development Kit (WDK) projects containing source code and a time boxed schedule that prevents a full review of documentation and code.

How should one approach the task of reviewing code written by a third party that does not necessarily conform to the standards one is accustomed to, or to any standard for that matter, in a limited amount of time?

The aim of this document is to give some basic principles on what to look for and to explain some of the commonly encountered ways of misuse of our systems that will lead to poor performance or other such issues. A technical reason why not to use a particular approach is given and corrective measures to each problem encountered are also discussed. All of the cases examined in detail come from erroneous practices that have been encountered “in the wild” and the proven technical solutions that have been implemented instead.

The primary focus of the document will be on the different programming approaches that are valid in the sense that they are part of our Application Programming Interface (API), yet using them in the wrong context will cause issues either when the volume of the system users or the objects stored in the repository grow.
DFC
DFC is the low level API to access the Documentum repository which allows developers to perform just about all possible tasks in the repository. There are multiple ways of achieving most of these goals and the following tries to outline the most commonly found misuses of the API.

Queries
Documentum is primarily a database application and review efforts should commence by looking at how the repository database is queried.

The way the locations where the repository is queried are identified is to do searches to the whole code base with search strings: `IDfCollection`, `IDfQuery` and maybe `select`. This will lead one to the locations where queries are executed and by doing some investigation one can find out whether the usage has been valid or not.

Collections
One of the first things to be checked is how collections are handled. Collections are database cursors opened by a Documentum Query Language (DQL) query and each repository session has by default ten (10) collections available. If a session tries to open up the eleventh collection it will receive an exception which has an error message indicating that the maximum amount of collections has been exceeded. In the more recent DFC versions this problem is not so pronounced since JVM garbage collection (GC) closes any open collections. This however is not a reliable way of closing collections since GC is not a predictable process. To be absolutely sure that a session does not run out of collections it is the burden of the developer to ensure that all collections are closed as soon as they have been handled.

Another compelling reason for closing open collections is to release any open database resources as swiftly as possible. A third reason for looking at the `IDfCollection` instances is that it will give one a general feel of the quality of the application as a whole.

If collections are not diligently closed, the application may completely stop working for a given session over a period of time forcing the user to re-login to be able to work with the application.
**Try finally paradigm**

All collections opened by the application must follow the approach below:

```java
final IDfCollection queryResults = query.execute(session, IDfQuery.READ_QUERY);
try {
    while(queryResults.next()) {
        /* Application logic. */
    }
} finally {
    queryResults.close();
}
```

Closing the collection in a `finally` block ensures that the opened results are always closed, regardless of how the rest of the application logic performs.

**Utility class**

A problem that arises with this is that the `IDfCollection.close();` method throws an exception that in the majority of cases is swallowed. If this potential exception is handled inline it causes bloated code that will be upwards of five lines of repeated code for each `IDfCollection.close();` call. This approach leads to code that is harder to maintain and clutters readability.

**NOTE:** The reason for this approach may be that the development team is measured on Source lines of code (SLOC).

The closing of collections should be placed in a utility class as a static method and called from the `finally` block of the code executing the query. Example of close method implementation:

```java
/**
 * Close a collection and log the possible (unlikely) exception thrown from close.
 * Created: 2 Dec 2006 13:09:53
 * @since 1.0.0.0
 * @author Christopher Harper
 * @param caller the calling class used to log a possible failure message.
 * @param results the result set to close.
 */
public static void close(final Object caller, final IDfCollection results) {
    try {
        if ((results != null) && (IDfCollection.DF_CLOSED_STATE != results.getState())) {
            results.close();
        }
    } catch(final DfException dexSwallow) {
        DfLogger.warn(caller, "Failed to close a collection.", null, dexSwallow); //$NON-NLS-1$
    }
}
```
Recursion

A typical location where one comes across the problem of running out of collections is recursion through a folder structure. What makes this an elusive problem is that it only appears with deep folder structures which typically are not present at testing time and the problem is only discovered in production. A simplified example of this problem would be:

```java
/**
 * Recursively print the folder paths of all the folders in the repository.
 * Created: 12 Jan 2009 14:56:51 Author: Christopher Harper
 * @since 1.0.0.0
 * @param parentFolderId the id of the folder whose containing folders paths are printed.
 * @throws DfException if the query fails.
 */
public void printFolderPaths(final IDfId parentFolderId) throws DfException
{
    final StringBuilder dql = new StringBuilder()
    .append("select r_object_id, r_folder_path from"); //NON-NLS-1$
    if (parentFolderId == null)
    {  
        dql.append(" dm_cabinet"); //NON-NLS-1$
    } else
    {
        dql.append(" dm_folder where any i_folder_id = \\
        ").append( //NON-NLS-1$  
            parentFolderId.getId()).append('"');  //NON-NLS-1$
    }
    final IDfCollection folders = new DfQuery(dql.toString()).execute(
        getSession(), IDfQuery.READ_QUERY);
    try
    {
        while (folders.next())
        {
            System.out.println(folders.getAllRepeatingStrings("r_folder_path", \\
                ""); //NON-NLS-1$ //NON-NLS-2$  
            printFolderPaths(folders.getId("r_object_id")); //NON-NLS-1$
        }
    } finally
    {
        Documentum.close(this, folders);
    }
}
```

As soon as the folder hierarchy is deeper than ten levels this code will break and can be fixed with one of two approaches. One is to modify the collection count in the `dfc.properties` file and the other – which is preferred – is to change the program to work like:

```java
final List<IDfId> folderIds = new Vector<IDfId>();
final IDfCollection folders = new DfQuery(dql.toString()).execute(
    getSession(), IDfQuery.READ_QUERY);
try
{
    while (folders.next())
    {
        System.out.println(folders.getAllRepeatingStrings("r_folder_path", \\
                "); //NON-NLS-1$ //NON-NLS-2$  
        folderIds.add(folders.getId("r_object_id")); //NON-NLS-1$
    }
} finally
{
    Documentum.close(this, folders);
}
for (final IDfId folderId: folderIds)
{
    printFolderPaths(folderId);
}
```
Ways of working

The above example serves well to illustrate a point that one must always be aware of – this is the distinction between “can do” and “should do”. Especially for the inexperienced developer DFC provides multiple ways of doing things, of which the example above falls into the category of “can do”. Not losing the bigger picture, though, one can and is supposed to give a “should do” approach to the problem instead. The current solution will execute one query for each `dm_folder` or its subtype in the repository. Given that the aim is to print all the folder paths available in the system the code should read as:

```java
/**
 * Print the folder paths of all the folders in the repository. Created: 12 Jan 2009 14:56:51
 * @since 1.0.0.0
 * @throws DfException if the query fails.
 */
public void printFolderPaths() throws DfException {
    final IDfCollection folders = new DfQuery("select distinct r_folder_path from dm_folder")
        .execute(getSession(), IDfQuery.READ_QUERY);
    try {
        while (folders.next()) {
            System.out.println(folders.getString("r_folder_path"));
        }
    } finally {
        Documentum.close(this, folders);
    }
}
```

This approach will reduce the amount of queries from (amount of folders + 1) to just one.

**NOTE:** The solution for this problem is refined further in the section Registered tables.
Query types

A common problem found in the applications reviewed is that the developers have not familiarised themselves with the provided API. A typical example of this problem would be:

```java
query.execute(session, 0);
```

And even if it would be written as:

```java
query.execute(session, IDfQuery.READ_QUERY);
```

There is little understanding of what the integer switch passed as the second argument to the `IDfQuery.execute()` method does. This ignorance typically demonstrates itself with the following problem description: “I know my query returns several objects, but the solution only updates 20 or so of them…”.

If the following information would have been looked up from either “Server Fundamentals” or “Documentum Foundation Classes API Specification” this sometimes hard to spot issue could have been avoided or at least deciphered more easily.

The issue outlined above is caused by a read query being closed if any data manipulation in the repository is performed on the same session whilst processing the query results. Following are the descriptions for each switch that can be passed to the `IDfQuery.execute()` method.

**Read query (IDfQuery.READ_QUERY)**

The query must be less than or equal to 255 characters in length if one is using Dynamic Data Exchange (DDE) as the communications protocol between the external application and Content Server.

Read query is used when you want to execute a select statement whose results will be processed without any database changes occurring during the processing. For such select statements, read-query provides better performance than the query method.

**NOTE:** Making changes in the repository while processing the results of a read query execution automatically closes the collection returned by the select. Consequently, if you want to make changes in the repository while processing query results use the query method rather than read-query.

One can execute non-select statements with read query also. However, there are no performance benefits to doing so.
If one wants to send a query that is greater than 255 characters and you are using DDE, use the exec query method instead of read query.

**Query (IDfQuery.QUERY)**
Whenever one executes a DQL statement using the query method, the results are returned as a collection. This feature lets you write generic code that can process any DQL statement whether it is a select statement or not.

**Cache query (IDfQuery.CACHE_QUERY)**
One uses the cache query method when executing a query whose results are generally static. For example, one might use cache query to return the users in the repository. Query cache files are maintained within and across sessions.

The results of the select statement executed by cache query are stored in a file in the client’s local area. The collection identifier returned by the cache query method points to this file.

The cache query method is only effective if query caching is enabled in the user’s repository and environment.

**Execute query (IDfQuery.EXEC_QUERY)**
One uses the exec query method when you want to send a long query to Content Server. A long query is defined as a query containing more than 255 characters.

**Execute read query (IDfQuery.EXECREAD_QUERY)**
One uses the read query flag for better performance when you execute a long select statement whose results will be processed without any database changes occurring during the processing.

**Apply (IDfQuery.APPLY)**
Used to execute non select Structured Query Language (SQL) statements.

**DQL queries**
Commonly a custom application contains DQL statements that make the job of the database server if not impossible at least unfeasible. Following are the most common encountered erroneous practices that need to be scrutinised with the development team and sometimes pushed back to the business whilst explaining to them why a given requirement is impractical.

**DQL scalar functions**
A common business requirement is to make a “Google” like search. This to an extent is possible with full-text searches that have the following syntax:

```
search document contains '<condition>'
```
As it stands the full-text also contains the metadata values of objects. This approach however is often not taken and the following type of DQL is implemented instead:

```
select *
from dm_sysobject
where upper(object_name) = '<upper_condition>'
  or upper(title) = '<upper_condition>'
```

In this example the scalar function `upper` is used which will disable the usage of database indexes when performing a query.

**NOTE:** Some database flavours however enable you to create function indexes on given columns.

The scalar functions provide in DQL are `upper`, `lower` & `substr` and if used in the where portion of the query they will degrade performance.

**Like**

Often the problem outlined in the previous section is made if possible worse by the usage of `like` in the following manner:

```
select *
from dm_sysobject
where object_name like '<condition>%'
  or title like '%<condition>'
  or upper(subject) like '%<upper_condition>%'
```

In this example the query has three different `like` conditions, more commonly known as:

- Starts with
- Ends with
- Contains

Of these three "Starts with" is the only one that can utilize database indexes effectively where the other two would have to rely on some more exotic form of database indexing to perform reasonably.

**Naming conventions**

It is commonplace and good practice for a project to have a naming convention where items have a suffix or a prefix added to the name to identify them as belonging to a given project or a customer. If queries are to be performed using the naming convention as a condition, the only way to go is suffixes given the way in which database indexes are built.
Spanning tables

In the previous two examples all the where condition columns resided in the same table, which made it possible to create an index to support these example queries. A typical custom application query does not conform to this rule and is of the form:

```sql
select *
from dm_sysobject
where owner_name = 'John Doe'
  and any keywords = 'Approved'
```

This too is an effective way to disable the usage of database indexes. The example above is created for standard Documentum object type and the column owner_name is in the table dm_sysobject_s and the column keywords is in the table dm_sysobject_r.

This issue often becomes more pronounced in custom applications where more concern is placed on the logical data model than the consideration of how the data will be utilized.

**NOTE:** See “Custom object types” for more details.

'*' queries

Queries with 'select *' are in surprisingly common use and they should be removed from the solution. In some cases these queries are used against registered tables where the harm is contained and the exes work the database needs to do is limited, to the contrary of cases where the type being queried is a custom type inherited from a low layer Documentum type.

How Documentum builds its type hierarchy is through several different tables that are collected into a view doing table joins. In a case where a * query is issued against a Documentum type the following rules apply:

1. Some single values are returned.
   a. DM object attributes are: r_object_id, object_name, title, subject, resolution_label, owner_name, owner_permit, group_name, group_permit, world_permit, log_entry, acl_domain, acl_name, language_code, r_object_type, r_creation_date, r_modify_date, a_content_type

2. Practically all custom single value attributes are returned.

Regardless of the situation only those columns actually required for the functioning of the application should be returned and even in the case where it would be all returned columns (more typical for registered tables) they should be listed to promote the readability of the code.
Local evaluation

In a few rare cases one has stumbled upon a clear lack of understanding of why and what a relational database is there for. Consider the following – real life – function:

```java
/**
 * Check whether a case is open. Created: 13 Jan 2009 10:04:25 Author:
 * Christopher Harper
 * @since 1.0.0.0
 * @param caseNumber the id of the case.
 * @return true if the case is open.
 * @throws DfException if the query fails.
 */
protected boolean isCaseOpen(final String caseNumber) throws DfException {
    /*
     * select
     * c_closed
     * from
     * c_case
     * where
     * c_case_number = 'YYYY-nnnnnnnn'
     */
    final IDfCollection results = new DfQuery(new StringBuilder(70).append("select c_closed from c_case where c_case_number = '").append(//$NON-NLS-1$ caseNumber).substring(')').toString()).execute(getSession(), IDfQuery.READ_QUERY);
    try {
        while (results.next()) {
            if (!results.getBoolean("c_closed")) //$NON-NLS-1$
                return true;
        }
    } finally {
        Documentum.close(this, results);
    }
    return false;
}
```

At first glance this might seem an all right way to find out whether a case is open, but it again falls into the category of "can do". What are the issues in this fairly benevolent looking method and how should one then remedy the situation to turn it into a "should do"?

First the query is not complete. The goal of the method is to find a single `c_case` with a given `c_case_number` whose `c_closed` is set to false. The `c_closed` condition should be added to the DQL query instead of evaluating it locally in Java. Secondly the query should not return all the rows that meet the condition since one just wants to know whether one or more case is open.
How should this be written to perform more efficiently? Here is a stab at creating a better performing function:

```java
/*
 * select count(r_object_id) as cnt
 * from c_case
 * where c_case_number = 'YYYY-nnnnnnnn'
 * and c_closed = 0
 */
final IDfCollection results = new DFQuery(new StringBuilder(110)
  .append("select count(r_object_id) as cnt from c_case 
    where c_case_number = \\
    " + caseNumber + \\n    " and c_closed = 0").toString()).execute(getSession(),
  IDfQuery.READ_QUERY);
```

In the above solution there are two modifications to make the function perform better.

1. A count database function instead of returning all rows.
2. Added c_closed = 0 condition to the where portion of the query.

Complex queries

In some rare cases customers have a requirement that their select query contain more than the ten (10) source tables that are allowed in DQL. This perceived problem and the issue of complex queries are typically addressed using database views that are registered into Documentum as registered tables. These views take care of the complexity of a given query so that the application DQL will be as straightforward as possible.

The only thing to keep in mind is that these views should not bypass the standard Documentum security. This is accomplished by returning just the object ID of the objects that have an Access Control List (ACL) attached to them (anything inheriting from dm_sysobject object type). Then the view and the dm_sysobject query are joined to create the final result set where the dm_sysobject query provides the security and the view contains the complexity of the query. A simplified example of a structure like this would be:
View SQL:

create view subscribers as select  
s.r_object_id  
, r.child_label  
, r.description  
, r.order_no  
, u.user_name  
from  
  dm_sysobject_s s  
, dm_relation_s r  
, dm_user_s u  
where  
  s.r_object_id = r.parent_id and  
  u.r_object_id = r.child_id and  
  r.relation_name = 'dm_subscription' and (  
    r.effective_date < sysdate or  
    TO_CHAR(r.effective_date,'DD-MM-YYYY HH:MI:SS') = '01-01-0001 12:00:00')  
and (  
    r.expiration_date > sysdate or  
    TO_CHAR(r.expiration_date,'DD-MM-YYYY HH:MI:SS') = '01-01-0001 12:00:00');

Register DQL:

register table dm_dbo.subscriptions (  
  r_object_id char(16)  
, r_child_label char(32)  
, description char(255)  
, r_order_no int  
, user_name char(32)  
)

Application DQL:

select  
  s.object_name  
, u.user_name  
, u.description  
from  
  dm_sysobject s  
, dm_dbo.subscriptions u  
where  
  s.r_object_id = u.r_object_id

Query hints

The code under review is typically void of query hints that instruct the database to process the results in a certain way depending on the hints passed in. It is recommended to develop queries so that they leverage these hints where applicable.

How to judge whether to use a hint or not is based on the comparison results of the query execution with the hint (different values) and without the hint. In addition to the obvious comparison of the execution time the resulting SQL of the DQL query should be examined to see which will produce the best result in the long run.
Hints that can be passed through using the enable block at the end of the query are:

- **SQL_DEF_RESULT_SET N**
  Most useful used against the SQL server where it directs the server to use a result set instead of a cursor. On other databases this functions in the same way as RETURN_TOP.

- **FORCE_ORDER**
  Controls the order in which tables are joined.

- **RETURN_TOP N**
  This hint limits the amount of rows returned by the query and is recommended to be used in conjunction with OPTIMIZE_TOP.

  **NOTE**: Oracle and Sybase do not handle this hint on the database level and it is left for the content server to handle it.

- **OPTIMIZE_TOP N**
  Instructs the database to return the first rows of the result quickly and the rest of the rows at a normal speed. If one is using sorting or the keyword ‘distinct’ in a query the effectiveness of this hint is reduced.

- **FETCH_ALL_RESULTS N**
  Instructs the database to return all the results from the database and close the cursor immediately. The hint doesn’t affect the execution plan but frees up database results more quickly.

- **OPTIMIZATION_LEVEL level_1 level_2**
  One uses the OPTIMIZATION_LEVEL hint against a DB2 database when you want to change the optimization level for a particular query.

- **UNCOMMITTED_READ**
  One uses the UNCOMMITTED_READ hint in read only queries, to ensure that the query returns quickly even if another session is holding locks on the tables queried by the read only query.

  This hint is useful only on SQL Server, DB2, and Sybase databases.
ROW_BASED

Normal behaviour of the content server is to bundle repeating attribute values into a "single" row with multiple values for the repeating value as follows:

<table>
<thead>
<tr>
<th>Row number</th>
<th>r_object_id</th>
<th>keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>090034f480005c6f</td>
<td>review, report, document</td>
</tr>
<tr>
<td>2</td>
<td>090034f480005c70</td>
<td>sop, draft</td>
</tr>
</tbody>
</table>

If the **ROW_BASED** keyword is provided the same result ends up like:

<table>
<thead>
<tr>
<th>Row number</th>
<th>r_object_id</th>
<th>keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>090034f480005c6f</td>
<td>review</td>
</tr>
<tr>
<td>2</td>
<td>090034f480005c6f</td>
<td>report</td>
</tr>
<tr>
<td>3</td>
<td>090034f480005c6f</td>
<td>document</td>
</tr>
<tr>
<td>4</td>
<td>090034f480005c70</td>
<td>sop</td>
</tr>
<tr>
<td>5</td>
<td>090034f480005c70</td>
<td>draft</td>
</tr>
</tbody>
</table>

Pass-through hints

These are hints that do not affect the content server but are passed directly to the underlying database. Use a database keyword to identify the database whose hints are in question. Valid values are: oracle, sql_server, sybase, and db2. Use the following syntax to pass these hints:

```sql
select d.object_name , u.user_address
from   dm_document d , dm_user u
where   d.r_creator_name = u.user_name
enable (oracle('RULE', 'PARALLEL'),
         sybase('AT ISOLATION READ UNCOMMITTED'),
         sql_server('LOOPJOIN', 'FAST1'))
```

Escaping

Projects often tend to be lax in how they scrutinize values coming from the client. An example of this would be DQL strings that are not checked for characters that need to be escaped as follows:

```sql
select r_object_id
from   dm_user
where   user_name = 'Conan O'Brien'
```

This query will fail since the single quote should have been escaped to produce DQL:

```sql
select r_object_id
from   dm_user
where   user_name = 'Conan O''Brien'
```
Values concatenated into a DQL query must be rigorously checked for invalid characters. Typically this is solved with a utility class that has a static method for escaping string parameters before they are added to a query.

**Registered tables**

The Content Server fundamentals manual defines registered tables as “Registered tables are RDBMS tables that are not part of the repository but are known to Content Server. They are created by the DQL REGISTER statement and automatically linked to the System cabinet in the repository. They are represented in the repository by objects of type dm_registered.

After an RDBMS table is registered with the server, you can use DQL statements to query the information in the table or to add information to the table.”

In addition to this super users can select from any underlying RDBMS table regardless of whether it is registered or not.

In the light of this consider if the example given in chapter [Recursion](#) where the best provided example executes the query:

```sql
select distinct r_folder_path from dm_folder
```

which the Content Server (CS) turns into the following SQL:

```sql
select distinct dm_repeating.r_folder_path
from
  dm_folder_sp dm_folder,
  dm_folder_rp dm_repeating
where (dm_folder.i_has_folder = 1
  and dm_folder.i_is_deleted = 0)
and dm_repeating.r_object_id = dm_folder.r_object_id
```

**NOTE:** This is the result of a query run as a super user which omits the security portion of the query since super users have read privileges to all repository objects. If a normal user would execute this query it would be substantially more complex.
Looks fairly straightforward—right? Well let's look behind the `dm_folder_rp` view which has the SQL query that looks like this:

```sql
SELECT
    LK_.r_object_id,
    LK_.i_position,
    LK_.i_partition,
    LK_.authors,
    LK_.keywords,
    LK_.i_folder_id,
    LK_.r_composite_id,
    LK_.r_composite_label,
    LK_.r_component_label,
    LK_.r_order_no,
    LK_.r_version_label,
    LK_.a_effective_date,
    LK_.a_expiration_date,
    LK_.a_publish_formats,
    LK_.a_effective_label,
    LK_.a_effective_flag,
    LK_.a_extended_properties,
    LK_.r_aspect_name,
    LK_.i_retainer_id,
    ULB_.r_folder_path,
    ULB_.i_ancestor_id,
FROM test.dm_sysobject_r LK_, test.dm_folder_r ULB_
WHERE (LK_.r_object_id = ULB_.r_object_id
    AND LK_.i_position = ULB_.i_position)
```

and the SQL query behind the `dm_folder_sp` view which looks like this:

```sql
SELECT
    JK_.r_object_id,
    JK_.object_name,
    JK_.r_object_type,
    JK_.title,
    JK_.subject,
    JK_.a_application_type,
    JK_.a_status,
    JK_.r_creation_date,
    JK_.r_modify_date,
    JK_.r_access_date,
    JK_.a_is_hidden,
    JK_.i_is_deleted,
    JK_.a_retention_date,
    JK_.r_links_cnt,
    JK_.i_links_high_cnt,
    JK_.r_assembled_from_id,
    JK_.r_frzn_assembly_cnt,
    JK_.i_reference_cnt,
    JK_.i_has_folder,
    JK_.r_link_cnt,
    JK_.r_link_high_cnt,
    JK_.r_assembled_from_id,
    JK_.r_frzn_assembly_cnt,
    JK_.i_reference_cnt,
    JK_.i_has_folder,
    JK_.r_link_cnt,
    JK_.r_link_high_cnt,
    JK_.r_assembled_from_id,
    JK_.r_frzn_assembly_cnt,
FROM test.dm_sysobject_s JK_, test.dm_folder_s SLB_
WHERE JK_.r_object_id = SLB_.r_object_id
```

Not so simple any more? Consider what a similar view for a type four levels beneath `dm_document` will look like.

**NOTE:** Each created type has a single value view that has the name: `<type_name>_sp` and a repeating value view that has the name: `<type_name>_rp`.

In comparison let's look at the query performed against the registered table. First the DQL looks like this:

```sql
select distinct r_folder_path from dm_folder_r
```

and the generated SQL is:

```sql
select distinct r_folder_path from <repository_name>.dm_folder_r
```

Which do you think will perform better and would the “ultimate” solution?
This section is not to say that registered tables are the be-all and end-all solution to performance issues, quite the contrary as outlined in the section Complex queries. However, one must say that in some cases – like method environment run as super user – they merit careful consideration.

Direct SQL
Given that the majority of developers have knowledge of working with databases but few have actually got their feet wet working with Documentum, it is not uncommon to see developers use direct Java Database Connectivity (JDBC) connections to the database to get the job done.

This is not a supported way of querying – let alone updating – the repository table space. The reason for it not being supported is that direct SQL access completely bypasses the inbuilt security model of Documentum and it can compromise the integrity of the data.

Get object
In an earlier chapter the overhead caused by '*' queries was discussed. This issue becomes even more pronounced with the usage of the different flavours of the IDfSession.getObject methods. Where the '*' queries return a limited number of columns, the getObject methods get every attribute value associated with the object and it calculates all applicable computed attributes. This can cause multiple table joins and several rows to be returned from the repeated value tables.

Computed attributes
A list of all computed attributes of which applicable ones are calculated:

```markdown
_accessor_app_perm, _accessor_name, _accessor_perm, _accessor_perm_type, _accessor_xpermit, 
_accessor_xpermit_names, _acl_ref_valid, _alias_set, _all_users_names, _allow_change_location, 
_allow_change_permit, _allow_change_state, _allow_execute_proc, _allow_change_owner, 
_attribute_list_values, _cached, _changed, _componentID, _containID, _content_buffer, _content_state, 
_current_state, _docbase_id, _dump, _has_config_audit, _has_create_type, _has_create_group, 
_has_create_cabinet, _has_purge_audit, _has_superuser, _has_sysadmin, _has_view_audit, _id, 
is_restricted_session, _isdeadlocked, _isnew, _isreplica, _istransactionopen, _lengths, _masterdocbase, 
_names, _permit, _policy_name, _repeating, _resume_state, _sign_data, _status, _type_id, _type_name, 
types, _typestring, _values, _xpermit, _xpermit_list, and _xpermit_names
```

Lifecycle related computed attributes:

```markdown
_alias_sets, _entry_criteria, _included_types, _next_state, _previous_state, _state_extension_obj, 
_state_type
```

From this two things can be determined. Firstly to utilise the available computed attributes if the object reference is available instead of doing additional queries, and secondly to avoid the getObject methods as much as possible.
Convenience vs. performance

The DfPersistentObject and its inherited classes are a convenient way of accomplishing tasks, but come with the price of added overhead. To get the best performance out of a Documentum solution it is more often better to go with DQL queries.

For example, obtaining an object name from a CURRENT dm_document using the getObject method will be written in less lines of code and will most likely be implemented quicker:

```java
final IDfPersistentObject object = session.getObject(new DfId(objectId));
object.getString("object_name"); //NON-NLS-1$
```

This, however, will cause a performance hit and should be written as:

```java
/*-
 * select
 * object_name
 * from
 * dm_document
 * where
 * r_object_id = 'nnnnnnnnnnnnnnnn'
 * enable (fetch_all_results 1, return_top 1, optimize_top 1)
*/
IDfCollection result = new DfQuery(new StringBuilder(150).append("select object_name from dm_document where ").append("r_object_id = "').append(objectId).append(" enable (fetch_all_results 1, return_top 1, optimize_top 1)").toString()).execute(getSession(), IDfQuery.READ_QUERY);
try {
    if (result.next()) {
        result.getString("object_name"); //NON-NLS-1$
    }
} finally {
    Documentum.close(this, result);
}
```

Justifications

There are only a few cases where the usage of getObject can be justified and they are:

- Check in / checkout
- Content file access (set/get)
- Single object updates
  - Even then all update operations can be performed through DQL. As an example:
    ```java
    update dm_document
    object set object_name = 'New name'
    where r_object_id = 'xxxxxxxxxxxxxxxx'
    ```
- Requirement to read a substantial portion of the object's metadata.
- Grant / revoke
- Operations that would cause unsubstantiated amount of code/work to complete.
Null checks
The final thing that needs to be said about the different flavours of `getObject` methods is that a null check **MUST** be performed on the returned value, since the methods return a null value if they do not succeed in returning the object specified in the argument(s). If during the review it is detected that the null check is not diligently performed a recommendation should be given to comb the entire codebase to add a null check to all locations where `getObject` methods are used.

**NOTE**: Using `getObject` methods in **WDK Preconditions** is the grossest abuse of this method encountered.

Piggybacking session
A common practice found in the reviewed solutions was to use a piggybacked `IDfSession` of an object passed to a function. This is **NOT** a recommended way of obtaining a session. To list the main reasons for not doing this:

- The calling code needs to have intimate knowledge of the implementation details of the method it is calling. When calling a method that uses this mechanism of obtaining sessions the calling code cannot release the session at will but is forced to hold onto the reference until the called method is finished.
- The session manager does not perform to its full potential when sessions are held indefinitely.
- If the session used to acquire the object is released back to the manager the object may after some time become unusable.

If for some reason one needs to hold onto an object reference for a prolonged period of time, the recommended approach to ensure the proper functioning of the underlying session pool is:

```java
final IDfDocument document;
final IDfSession session = sessionManager.getSession(docbaseName);
try {
    document = (IDfDocument) session.getObject(new DfId(documentId));
    document.setSessionManager(sessionManager);
} finally {
    sessionManager.release(session);
}

Continue one's work with the object.
```
BOF

Business Object Framework (BOF) is a convenient way of delivering custom functionality to a variety of client applications.

TBO

Type Based Objects (TBO) are a convenient way of hooking custom functionality to standard operations performed by the various clients that access the repository. How do people most commonly violate them?

NOTE: Since a TBO can be attached to one’s custom types the functionality and the complexity that is to be deployed with the TBO may slightly affect one’s type design.

Convenience methods

One thing that developers typically do is to create convenience methods for all of their custom attributes. On single value attributes this may – in some rare cases – be arguably the right approach, especially if the TBO is utilized in a custom application where the convenience methods can actually be utilized – standard clients obviously do not know of them – but even then it is highly questionable. When the same approach is used for repeating attributes the result is nothing short of disastrous. With this approach each repeating attribute gets eleven (11) convenience methods:

1. appendXXX(value);
2. findXXX(value);
3. getAllXXX(separator);
4. getXXX(index);
5. getXXX(); /* index zero */
6. insertXXX(index, value);
7. removeXXX(index);
8. removeAllXXX();
9. truncateXXX(index);
10. setXXX(index, value);
11. setXXX(value); /* index zero */

Currently IDFSysObject carries 350+ methods and those combined with the often ridiculous amount of convenience methods make the TBOs actually less intuitive and even counterproductive.

Application logic

Another common practice is to embed application logic into the TBO which in turn commonly causes unnecessary getObject calls just to get the application logic triggered, thus creating a situation that promotes an unfavoured programming approach.
Utility classes
One tends to think the TBOs themselves should be as lightweight as possible without any convenience methods with all of the application logic encapsulated into utility classes that can be used from a variety of contexts, not just within the TBO class.

SBO
Service Based Objects (SBO) provides functionality that is not specific to a particular object type or repository and is installed into the global repository. The only issue sometimes observed in the context of SBOs is that people go “technology happy” and fail to ask the question “how is the task the easiest to implement” thus missing the obvious answer: Plain Old Java Object (POJO).

Methods
It often seems to be the case when doing reviews on methods implemented to run on Java Method Server (JMS) that everything learnt about Object-oriented programming (OOP) is suddenly thrown out of the window and there is a return to the “stone ages” writing procedural programming. Typically this throwing away is demonstrated in a project where there are multiple methods that all start somewhat similarly:

```java
/**
* @since 2.0.0.0
* @param arguments the method arguments
* @param report the log stream
* @return the method return value
* @throws Exception
* @see com.documentum.fc.methodserver.IDfMethod#execute(java.util.Map,
* java.io.PrintWriter)
*/
@SuppressWarnings("unchecked")
@Override
public int execute(final Map arguments, final PrintWriter report) throws Exception {
    final IDfLoginInfo login = new DfLoginInfo(((String[]) arguments .get("user"))[0], null); //NON-NLS-1$
    final IDfSessionManager manager = DfClient.getLocalClientEx().newSessionManager();
    manager.setIdentity("*", login); //NON-NLS-1$
    final IDfSession session = manager.getSession(((String[]) arguments .get("docbase_name"))[0]); //NON-NLS-1$
    try {
        /* The method logic. */
    } finally {
        manager.release(session);
    }
    return 0;
}
```

Above is a tidied up version of what a typical repeated method implementation looks like.
One of the primary goals of any software project should be maintainability, which should also be a key focus of the review. What if something changes in one of the method implementation details? One is in a situation where each method needs to be re-written to meet the changed details. There are other adverse effects with this approach, the chief amongst them being the tendency to implement the same thing over and over again with each method having its own implementation of a particular task.

**Method framework**

What if some thought were put into how methods, besides reasonable performance, were best executed and best maintained?
The following Unified Modelling Language (UML) class diagram gives you an idea of what the result of such thinking might be:
This in no way aims to be an all-encompassing approach to executing methods, but it goes miles beyond the approach given in the code example. The main ideas of the classes in the above diagram are:

- **DCTMMethod** – abstract class that takes care of all the basic functionality of a server method. Gathers all the arguments and loads the method property file. When all preparations are done calls the abstract method `performWork()`.
  - **Logger** – creates a standardised log file for each method.
  - **Accessors** – utility class to check whether an accessor exist in the repository.
  - **Arguments** – class holding the arguments passed to the method as arguments or from the method/job object.
  - **Sessions** – a container for open sessions where they are cached.
  - **Settings** – method specific settings file handler (for each method class a resource file with the same package and name is loaded if present).
  - **General** – a class containing general constants and utility method.
  - **MethodArguments** – parameter names for the methods.
  - **IDfMethod** – interface that a method must implement (interchangeable with `IDmMethod`).
  - **IDmMethod** – interface that a method must implement (interchangeable with `IDfMethod`).

- **AuthenticatedMethod** – Authenticates that the calling user is valid and only then calls the abstract method `performAuthenticatedWork()`.

- **Workflow** – Loads and acquires a work item and then calls the abstract method `handlePackage(final IDFPersistentObject object)` for each package object and then calls the abstract method `handleFinalising()` to complete the work item.

- **Job** – Checks whether a job is in its allocated time window and if it is calls the abstract method `performJob()`. If the job is not in its time window a new execution time is calculated and saved in the job, but the job itself is not run.

**Utility classes**

To avoid writing code to solve the same problem over and over again it must be placed into a location where it can be called from multiple locations.
Above is an UML class diagram from a past project where all re-usable functionality outside of the method first requiring that functionality is written as sub-class of DCTMMethodUtil. The chief design goal was to easily provide to the utility the same session(s), log, parameters and settings that the method has. This is achieved by constructing the utility class with the running method as an argument. From this instance all the required information can be obtained.

**NOTE:** As the utility is constructed with a concrete class, the utilities are bound to the method server environment. A future development effort would be to extract an interface from DCTMMethod class that would be used as the parameter in the utility class constructor. The created interface could be implemented by any functionality de-coupling the method from the utility.
Example

How would a method written on top of such a framework differ from methods implemented with a more procedural approach? Let’s take a look at a demo method written using the real life classes displayed in the above UML diagrams.

```java
public final class Export
    extends AuthenticatedMethod
{
    /**
     * Perform the export, zipping and moving. Created: 26 Nov 2008 12:51:25
     * Author: Christopher Harper
     * @throws DfException if the method fails.
     * @since 1.0.0.0
     * @see com.emc.dctm.method.AuthenticatedMethod#performsAuthenticatedWork()
     */
    @Override
    protected void performAuthenticatedWork() throws DfException
    {
        final IDfId docId = getArguments().getDocumentId();
        if (docId == null)
        {
            throw new DfException("An invalid value passed for the argument \%. A valid ID is required!", //$NON-NLS-1$
                new Object[] {MethodArgumentNames.DOCUMENT_ID});
        }
        final IDfSession session = getSession();
        try
        {
            final IDfDocument document = (IDfDocument) session.getObject(docId);
            final ZipFile zip = new ZipFile(this, new File(document.getFile(null)));
            final File zip = File.createTempFile("demo_", ".zip"); //$NON-NLS-1$ //NON-NLS-2$
            zipper.zip(zip.getParentFile(), zip.getName());
            final Copy copy = new Copy(this);
            final File target = new File(getSettings().getString(this,
                "DEPLOYMENT_DIR"), zip.getName()); //$NON-NLS-1$
            copy.copy(zip, target, true);
            setReturnValue(Return.SUCCESS, String.format("Exported document with id \%s to \%s.", new Object[] {docId, zip})); //$NON-NLS-1$
        }
        catch(final IOException ioex)
        {
            throw new DfException("Failed to export document with id '\%0'\.", //$NON-NLS-1$
                new Object[] {docId, ioex});
        }
        finally
        {
            release(session);
        }
    }
}
```

A fairly small fingerprint for a fully functioning JMS method that performs the following tasks:

- Parses the arguments from the method call and the method object.
- Authenticates that the calling user can log into the repository.
- Maintains a time stamped log file like this:

```plaintext
[BROKER 2 host: 192.168.131.128, protocol: rpc_static, port: 1489, timeout: 0]
[BASE 1 name: onyx, id: 12345, desc: Demo repository]
[DCF 6.8.0.11581]
[INFO 10:12:00:644][main] Using debug user dmadmin information to connect to repository onyx.
[INFO 10:12:00:644][main] Launch argument debug_password=***************
[INFO 10:12:00:644][main] Launch argument level=0
[INFO 10:12:00:644][main] Launch argument docbase_name=onyx
[INFO 10:12:00:644][main] Launch argument debug_user=dmadmin
[INFO 10:12:00:644][main] Launch argument start_user_ticket=***************
[INFO 10:12:00:644][main] Launch argument objectId=00031038000000600
[INFO 10:12:00:644][main] Default method description that should be overridden in each method properties file.
[INFO 10:12:00:937][main] Copied 45 KB from C:\Temp\dcm_pdf.pdf to /tmp/ZipTest123456.pdf.
[END] 09 January 2009 10:12:01 EET
[DURATION 00:00:03:748]
```
**NOTE**: Exceptions thrown will also be logged with their stack trace.

- Handles and caches sessions.
- Uses two utility classes:
  - ZipFile: Class that zips either a single file or a whole directory to a target zip file.
  - Copy: Functionality to copy files efficiently either locally or over a network.

If the same functionality were to be implemented in a procedural manner, the code would be hundreds or even thousands of lines instead of the current 40 or so lines.
WDK

WDK is the Documentum tool kit for building content enabled web applications.

Preconditions

Most of us are familiar with the term: “premature optimization is the root of all evil” coined by Donald Knuth. In WDK there is at least one location where this statement is questionable. Given the volume of hits a precondition may receive there is no measure too drastic to try and improve their performance.

Consider the following partial – real life – precondition in the light of what was discussed under the chapter “Get object”:

```java
public boolean queryExecute(String s, IConfigElement iconelement, ArgumentList argumentlist,
                            Context context, Component component)
{
    String basketId = argumentlist.get("objectId");
    try
    {
        IDfSysObject basket = (IDfSysObject)component.getDfSession()
                                   .getObject(new DfId(basketId));
        if(basket.getOwnerName().equals(component.getCurrentLoginUsername()))
        {
            return true;
        }
    }
    catch(Exception d)
    {
        Trace.println("error getting the basket owner" + d.getMessage());
    }
    return false;
}
```

What is so appalling about this, one might ask? First of all, when opening a folder in Webtop classic view the precondition is executed once for each object that is in the scope of the action and the same precondition may be defined for multiple actions. This precondition at its very best will cause multiple unnecessary get object calls, and if the same approach is taken in several of the custom application preconditions, the whole application will grind to a halt.

How should this be remedied then? When the content of the folder is rendered, each data grid row has a set of arguments passed to it. This set can be located by finding out the jsp page that is used to render the list of items. The following argument list is taken from the file %webtop%\webcomponent\navigation\doclist\doclist_body.jsp:
Above we see the highlighted value `<dmf:argument name='ownerName' datafield='owner_name'/>` which can be passed to the action precondition. To make this value available to the precondition we need to modify the action definition xml as follows:

```xml
<params>
  <param name="objectId" required="true" />  
  <param name="ownerName" required="true" />  
</params>
```

And if the – often the way not to go – argument is marked `required="true"` one needs to modify the method `getRequiredParams()` from the action precondition class as follows:

```java
public String[] getRequiredParams()
{
    return new String[] {"objectId", "ownerName"};  //$NON-NLS-1$  //NON-NLS-2$ 
}
```

After this the precondition can be written as:
/**
  * Check whether the current user owns the item. Created: 11 Jan 2009
  * 10:53:05 Author: Christopher Harper
  * @since 1.0.0.0
  * @param actionName name of the action.
  * @param config configuration xml settings.
  * @param arguments action arguments.
  * @param context the action context.
  * @param component the caller component.
  * @return true if the logged in user is the owner of the item.
  * @see com.documentum.web.formext.action.IActionPrecondition#queryExecute(java.lang.String,
  * com.documentum.web.formext.config.IConfigElement, com.documentum.web.common.ArgumentList,
  */
public boolean queryExecute(final String actionName, final IConfigElement config,
  final ArgumentList arguments, final Context context, final Component component)
{
    return SessionManagerHttpBinding.getUsername().equals(arguments.get("ownerName")); //NON-NLS-1$
}

This will just perform a memory lookup instead of the expensive database lookup.

NOTE: The argument list above is the standard one and by using the standard WDK technique of
overriding the objectlist component and creating a new jsp page a custom set of arguments
can be passed to action preconditions.

Now looking at the example, maybe this one falls more into the category of “right and wrong”
instead of optimization as stated at the beginning of the chapter, but the main point still stands
in relation to preconditions. Make them as fast as possible!

Docbase object configuration

This is the only one of the sections that does not focus on the spring chickens amongst the
community that implements Documentum applications. The old ways of doing things seem to be
hard to shake – sometimes the newcomers learn the correct way of doing stuff right off the bat.

For those of us who have been around for a while while the modification of the standard attributes
component in WDK was synonymous with some customization work which still seems to be the
case for a large portion of us. Grief was especially directed at the tag <dmfx:docbaseattributelist>
which could not easily be configured or customized. All sorts of innovative approaches were
taken to customise the attributes dialog, which often were not upgrade friendly.
Nowadays, however, there is a configuration file in the directory %WEBAPP%/webcomponent/config/library/ with the name pattern docbaseobjectconfiguration_dm_???.xml that makes light work of the toil previously experienced when modifying the attributes component. By extending this configuration one can define for each attribute the following:

- Value handler
- Value formatter
- Tag class
- Label tag class
- Value tag class
- Edit component

And the same goes for different data types also. If one encounters one of these elaborate attributes component configurations in the wild it is probably due just to a lack of knowledge that things can be done more easily.
General

This section lists issues that are general hindrances to a Documentum project – regardless of the technology used – and its life in production.

DBA

As stated in the section Queries, Documentum is first and foremost a database application. Any Documentum application is crippled in the long run if it does not have someone to monitor and work on its database to maintain acceptable performance. Utilizing a database administrator (DBA) as a magic bullet when the custom solution is already grinding to a halt rarely has the desired effect. To get the most out of a DBA, they should be a part of the application creation project participating in designing the application queries and the object model. The rest of the development team normally finishes its work when the application is deployed into production, which is not the case for the DBA whose job of monitoring the repository database continues indefinitely.

Language

Natural

That English is the language to use when working with just about any type of technical discipline related to computers would be the understanding of most of us who work with them. This however is not always the case when working with customers in non-English-speaking countries and especially with public sector projects that have the requirement of native language documentation.

It is understandable that end user facing documentation be created in a language other than English, but often the requirement is that all documentation be produced in the native language and in the extreme cases even the code and comments are not written in English.

Concern raised

What is the concern here and why does it merit its own chapter in a code review document?

From a technical standpoint it makes performing technical reviews a lot harder – if not completely impossible – since one cannot read the business cases or the design specifications for the application. In the extreme cases where code and comments are not English it becomes hard to decipher what the code is supposed to do and it violates the idea that one should be able to read it with ease. Consider the following simplified code as an example:
/**
 * Luo kansio ja sijoita se kohdekansioon. Created: 12 Jan 2009 11:06:43
 * Author: Christopher Harper
 * @since 1.0.0.0
 * @param kohdekansioId
 *   mihin uusi kansio sijoitetaan.
 * @param arvot
 *   uuden kansion viitetiedot.
 * @return uusi kansio
 * @throws DfException
 *   jos kansion luonti epäonnistuu
 */
public IDfFolder luoKansio(final IDfId kohdeKansioId, final Map<String, List<String>> arvot)
   throws DfException
{
    final IDfFolder uusiKansio = (IDfFolder) getSession().newObject("dm_folder"); //NON-NLS-1$
    for (final String nimi: arvot.keySet())
    {  
       for (final String arvo: arvot.get(nimi))
        {  
           uusiKansio appendString(nimi, arvo);
        }
    }
    uusiKansio.link(kohdeKansioId.getId());
    uusiKansio.save();
    return uusiKansio;
}

It is valid code, but reading and understanding it is hard – yes, even for a native Finnish speaker – compared to the same thing written in English:

/**
 * Create a folder and link it to the target folder. Created: 12 Jan 2009 11:06:43
 * Author: Christopher Harper
 * @since 1.0.0.0
 * @param targetFolderId
 *   where to link the new folder.
 * @param values
 *   metadata values for the new folder
 * @return the new folder
 * @throws DfException
 *   if the folder creation fails.
 */
public IDfFolder createFolder(final IDfId targetFolderId, final Map<String, List<String>> values)
   throws DfException
{
    final IDfFolder newFolder = (IDfFolder) getSession().newObject("dm_folder"); //NON-NLS-1$
    for (final String name: values.keySet())
    {  
       for (final String value: values.get(name))
        {  
           newFolder appendString(name, value);
        }
    }
    newFolder.link(targetFolderId.getId());
    newFolder.save();
    return newFolder;
}

In code written in a foreign language one needs to carefully read each line of code – a bit like reading code that does not use descriptive variable names – to understand what is going on instead of just glancing at the comment and the method signature to gauge what is going on.
Another unpleasant feature that is raised especially with WDK applications is that the User Interface (UI) becomes either completely foreign or a hybrid with two languages. The rule should be that the application is first wholly developed in English and then translated into the other language.

**Resource problem**

Why not just let people implement their solution in the way that they choose, instead of placing demands on the way they work? The answer is simple: knowledgeable Documentum resources are sparse to come by and all of them regardless of their nationality have a sufficient command of the English language. If the project is in a non-English language, the customer might be unable to find help when they most need it.

As an example, one did a technical review of a project that was done wholly in French, and they were having considerable problems finding capable resources that understood French. This is a good point illustrating the fact that it is not just the small languages that need to pay attention to the language issue. Reviews are often performed at different stages of the development cycle of the application. As soon as it is revealed that English is not the working language of the technical portion of the project it needs to be flagged as a huge risk for the success of the project.

It seems – like writing cumbersome code – that writing documentation in a non-English language has often more to do with job protection than the true benefit of the customer.

**Programming**

When working either in the capacity of doing a technical review or a technical lead one is commonly faced with the question: “What standards should we adhere to when doing Documentum development?” During the years one has seen several different “standards” floating around and trumpeted as the way to do Documentum development, but all of them only seem to cause confusion to the teams where they are utilized.

**Java**

In one’s experience referencing anything but the standard “[Code Conventions for the Java Programming Language](https://docs.oracle.com/javase/8/docs/technotes/guides/language/standards-ejbcoding-conventions.html)” created by Sun invites the amusement of the development team one is working with and underminds one’s credibility.


**Others**

Given that our repository is exposed through Documentum Foundation Services (DFS) to practically all possible programming environments the standards question may arise in any of these. Like with Java, one would always fall back to the standard provided by the vendor of the given language / technology and only if there was none available, would one hesitantly make suggestions.

**Exception handling**

Swallowing of exceptions is one of the most common violations that developers do. The rule is that exceptions should never be swallowed. However, there are rare circumstances where this is arguably the right way to go as shown in the [collection closing example](#), but it must be crystal clear that the swallowing is intentional.

**Catching**

The catch statement should be for a specific exception type. So if the try block contains statements that throw `DfServiceExceptions` a `DfException` should not be caught. The base exception types `Throwable` and `Exception` should never be caught.

**Pass-through problem**

The usage of checked exceptions in projects may cause issues, because they inappropriately expose the method implementation details. A large amount of exceptions being thrown from a single method is normally caused by a developer who tallies up all the exceptions thrown from a methods initial implementation and adds them to the throws clause (many Integrated Development Environments (IDE) help you to do this.). One of the problems in the pass-through approach is that it does not conform to the Joshua Bloch's Item 43 “Throw exceptions appropriate to the abstraction” from the book “Effective Java”.

This approach commonly leads to a situation where the caller does not know what went wrong and does not know what to do with the thrown exception. It may also lead to a situation where instead of catching a specific exception a lowest common denominator is caught. This common denominator is often `java.lang.Exception` that should never be caught. By using exception chaining exceptions that are more appropriate can be thrown without throwing away exception details such as the stack trace of the underlying problem.
Method signatures
The previous pass-through problem often leads to unstable method signatures, since every time the implementing method is changed the fingerprint of the method may change and that can cause a domino effect that goes through the whole code. Managing this type of fragile method signatures becomes expensive especially after the class has been deployed. This problem goes back to Bloch's Item 43 that states that a methods exception should reflect what the method does, not how it does it.

If developers get caught in a situation where the method signatures constantly change, they commonly stop using exception handling by just declaring that their methods throw java.lang.Exception. This approach, needles to say, is not a good exception handling strategy.

Unreadable code
If a large number of methods throw more than one exception, the ratio of actual code that does something compared to the code that is in place just to handle exceptions can be very high. The design principle behind exceptions is to make the code smaller by centralizing exception handling. Methods with ten different exceptions may cause a situation where a simple method with a couple lines of code has easily over forty lines of exception handling.

Documentation burden
While un-checked exceptions help us to get rid of the main design problems, it introduces a new one. Checked exceptions are a hundred percent clear to developers because they need to react to them by either catching them or re-throwing them. Un-checked exceptions require that each method that throws any type of exception document them exhaustively. Using uncaught exceptions may require that try { } finally { } blocks are more commonly implemented to make sure that resources such as collections and sessions are properly cleared out.

Ideal situation
The ideal situation for exceptions would be that each different situation would have its own exception type. This however is not feasible, and a middle ground between exception clarity and a reasonable amount of different exceptions should be agreed on.

Utilisation of constants
Typical observation while reviewing custom code reading is that constants provided in DFC are not being utilised as they are supposed to, making the code harder to read. It may be clear to an experienced Documentum developer that the integer value 3 means READ privilege in the
context of an ACL, but using the constant `IDFACL.DF_PERMIT_READ` makes it abundantly clear to all what the privilege in question is.

Constants that should be utilized can be found from the following classes:

```java
com.documentum.fc.bpm.IdfAttributeValueCondition, com.documentum.fc.bpm.IdfTransitionCondition,
com.documentum.fc.client.DfServiceException, com.documentum.fc.client.IDFACL,
com.documentum.fc.client.IDFActivity, com.documentum.fc.client.IDFAliasSet,
com.documentum.fc.client.IDFCollection, com.documentum.fc.client.IDFPermission,
com.documentum.fc.client>IDFPersistentObject, com.documentum.fc.client.IDFQuery,
com.documentum.fc.client.IDFRelationType, com.documentum.fc.client.IDFRetainerDispositionRule,
com.documentum.fc.client.IDFRetainerEnforcementRule, com.documentum.fc.client.IDFRetainerImmutabilityRule,
com.documentum.fc.client.IDFRetainerRenditionRule, com.documentum.fc.client.IDFRetainerStrategy,
com.documentum.fc.client.IDFRetentionRuleType, com.documentum.fc.client.IDFRetentionStatus,
com.documentum.fc.client.IDFSseekRoot, com.documentum.fc.client.IDFSession,
com.documentum.fc.client.IDFUserManager, com.documentum.fc.client.IDFType,
com.documentum.fc.client.IDFSessionManager, com.documentum.fc.client.IDFSession,
com.documentum.fc.client.IDFSVersionPolicy, com.documentum.fc.client.IDFSWorkflow,
com.documentum.fc.client.IDFSWorkflowBuilder, com.documentum.fc.client.IDFSWorklet,
com.documentum.fc.client.acs.IDFACSConfig, com.documentum.fc.client.search.IDFAttrExpression,
com.documentum.fc.client.search.IDFExpression, com.documentum.fc.client.search.IDFExpressionSet,
com.documentum.fc.client.search.IDFQueryBuilder, com.documentum.fc.client.search.IDFQueryDefinition,
com.documentum.fc.client.search.IDFQueryEvent, com.documentum.fc.client.search.IDFQueryScope,
com.documentum.fc.client.search.IDFQueryStatus, com.documentum.fc.client.search.IDFResultEntry,
com.documentum.fc.client.search.IDFSsearchOperation, com.documentum.fc.client.search.IDFSsearchSource,
com.documentum.fc.client.search.IDFSimpleAttrExpression,
com.documentum.fc.client.search.IDFValueListAttrExpression,
com.documentum.fc.client.search.IDFValueRangeAttrExpression, com.documentum.fc.common.DfId,
com.documentum.fc.common.DfList, com.documentum.fc.common.DfValidationException,
com.documentum.fc.common.IDFAttr, com.documentum.fc.common.IDFException, com.documentum.fc.common.DfId,
com.documentum.fc.common.IDFList, com.documentum.fc.common.ILogInInfo,
com.documentum.fc.common.IDFProperties, com.documentum.fc.common.ILogInInfo,
com.documentum.fc.common.IDFValue, com.documentum.operations.IDFCheckinOperation,
com.documentum.operations.IDFCopyOperation, com.documentum.operations.IDFDeleteOperation,
com.documentum.operations.IDFExportOperation, com.documentum.operations.IDFOperation,
com.documentum.operations.IDFOperationError, com.documentum.operations.IDFOperationMonitor,
com.documentum.operations.IDFVDMPlatformUtils, com.documentum.registry.IDFRegistry,
com.documentum.xml.xdql.IDFXmlQuery
```

**Data caching**

It is not foreign to some reviewed Documentum applications to query for a particular piece of information, use it and then disregard it only to repeat the same in the next method as if the information were just a memory lookup. To add insult to injury this practice is most commonly coupled with the – one hopes notorious by now – `getObject` method.

The saying “database access is the slowest part of an application” may be familiar. With Documentum repositories it could be argued that file operations are even slower. Either way the gist of the point being made in this section is that one does not want to do unnecessary database operation if at all possible.
Candidates
What should one cache then? That varies from application to application, but the rule of thumb to apply is that if it does not have an ACL attached the object/value is a good candidate for caching. This is all types except those that inherit from `dm_sysobject` which makes the pool of candidates fairly substantial. Not caching objects/values that inheriting from `dm_sysobject` is only a general guideline since this cache can readily be made available to all users. However, there are valid situations where `dm_sysobject` objects and its sub-types are cached either with security lookup or in user specific caches. A typical example of a user specific cache would be caching a custom user specific configuration object.

Other considerations that affect the design of possible data caching are the volume of data to be cached, how often the cached data is updated and how frequently it is accessed. The main rules for these three points would be:

- A high volume of data may consume too much memory to be practical for in-memory caching and serializing the data to disk may be just too cumbersome.
- If the data to be cached is constantly changing maintaining an up-to-date instance of the data may become a task that in itself takes either more time to execute or the development time of such a solution takes too long.
- The more frequently the cached data is accessed the more beneficial will the cache be. So data that is accessed once a day – even an hour – is probably not a good candidate for caching.

Implementation
Cache solutions are sometimes encountered where it may seem that the whole solution is nothing but an elaborate caching mechanism and the actual goal of the application is lost. Why not use a standard approach already available from Java?
package com.emc.dctm.proven;

import java.util.LinkedHashMap;
import java.util.Map.Entry;
/**
 * Simple cache of a given size.
 * Created: 20 Jan 2009 10:40:28
 * Description: A simple cache implementation that automatically removes
 * the eldest entry when the cache size grows beyond the given size.
 * @author Christopher Harper
 * @version 1.0.0.0
 * @param <K> key type.
 * @param <V> value type.
 * @since 1.0.0.0
 */
public class SimpleCache<K, V>
        extends LinkedHashMap<K, V>
{
    /**
     * serialVersionUID = -7932495921345790031L;
     * @since 1.0.0.0
     */
    private static final long serialVersionUID = -7932495921345790031L;

    /**
     * How many items should this cache contain?
     * @since 1.0.0.0
     */
    private final int cacheSize;

    /**
     * Sole constructor for the cache. Created: 20 Jan 2009 10:43:45 Author: Christopher Harper
     * @since 1.0.0.0
     * @param theCacheSize how many items to store in the cache.
     */
    public SimpleCache(final int theCacheSize)
    {
        this.cacheSize = theCacheSize;
    }

    /**
     * Check whether the eldest entry should be removed. Created: 20 Jan 2009 10:44:35 Author:
     * Christopher Harper
     * @since 1.0.0.0
     * @param eldest the eldest entry.
     * @return true if the entry should be removed.
     * @see java.util.LinkedHashMap#removeEldestEntry(java.util.Map.Entry)
     */
    @Override
    protected boolean removeEldestEntry(final Entry<K, V> eldest)
    {
        return this.cacheSize <= size();
    }
}

The goal of the elaborate caches can typically be achieved with the previous approach or a slight variation of it.
Custom object types
When a customer wants to describe their information with domain specific metadata, one is probably creating custom object types that are specific to the given customer. So how should this task be approached?

Design principles
First let it be said that the common approach taken here is that there are no “design principles” and it always depends on the customer in question. However, some general guidelines may be given on how to approach the task of creating an object model for a customer.

- Do thorough investigation of the metadata the customer uses to describe their content.

  Depending on the size of the customer and the maturity of the customer’s information architecture this task will probably consume the majority of one’s time when designing the object model.

- Avoid creating deep object type hierarchies. Each type creates at least one extra table into the repository and if the type has repeating attributes two tables are created.

- Avoid repeating attributes.

  Repeating attributes are not the most convenient when doing application queries given the any keyword that needs to be used in the where portion of the DQL statement.

- Use an abstract “master” object to hold all the customer’s common attributes.

- Re-purpose attributes if they are not used on a higher level.

- Consider indicating the type of an object with an attribute instead of a concrete type.

- Prefix your type and attribute names with a customer specific prefix.
Let's look at a simplified custom object model and apply some of the guidelines mentioned above. First one has defined all types for the customer with all their attributes.

From this one can tell that the attribute c_guid is shared across all the custom types and should be moved up the structure. It is also obvious that c_video and c_music share attributes and so do c_project, c_part and c_task. The next iteration of the type hierarchy would look like:

Where two new types c_media and c_technical have been created and c_project and c_task have disappeared to be replaced with the attribute c_type.

The other potential modification to this would be to remove the attribute c_artist and utilise the attribute authors from dm_sysobject by just giving it a custom label ‘Artist’ on the level of c_media.

Once one has decided on the attributes that comprise the customer’s data model one can start to look at where the attributes should be placed. Let’s say that one of the primary queries the application would be performing were something like:
This would compel us to move the attribute `c_part_number` up from `c_part` to the type `c_technical` to be able to create an index that would support the query above. Then – likely since we already have the `c_type` attribute in `c_technical` – we would remove the type `c_part` and use attribute to indicate whether the object was an instance of `c_part`. Then the query would be written as:

```sql
select
  r_object_id,
  object_name,
  subject,
  title
from
  c_technical
where
  c_part_number = '<part_number_been_returned>'
  and c_code = '<code_been_returned>'
  and c_type = 'part'
```

This is just an example of how the custom object structure will evolve depending on the customer's information architecture and the requirements imposed by the application that will utilize this data.

**Persistent object inheritance**

In the section `'*' queries` and `getObject` the amount of attributes that are returned was discussed and the conclusion was reached that often developers fetch much more information than is required for the solution to work. What if one would design the solution so that this would not have such a drastic effect?

Quite commonly custom applications extend the object model from one of the following `dm_user`, `dm_sysobject`, `dm_folder`, `dm_cabinet` or `dm_document` of which all but `dm_sysobject` normally make sense. When extending a type the following considerations should be made:

- Extend `dm_sysobject` or a sub-type if one of the following conditions is met:
  - Need for security.
  - Visible in standard clients.
- Extend `dm_document` or a sub-type if one of the following conditions is met:
  - Need to store content.
  - Need for versions.
• Extend dm_folder or a sub-type if one of the following conditions is met:
  o Need to contain items.
• Extend any of the existing types if additional attributes need to be added to a type e.g. dm_user.

If none of the above conditions were met, a consideration about extending “persistent object” must be done. If there is super user access to the system – which there normally is – types can be created directly under the virtual type “persistent object”. The persistent object type has three properties that it passes to all of its subtypes:

• r_object_id
  The r_object_id property contains a 16-character hexadecimal string that is assigned by the system when an object is created. This value uniquely identifies the object in the repository.

• i_vstamp
  The i_vstamp property contains an integer value that represents the number of committed transactions that have changed an object. This value is used for versioning, as part of the locking mechanism, to ensure that one user does not overwrite the changes made by another.

• i_is_replica
  The i_is_replica property indicates whether the object is a local replica of an object in a remote repository.

Typically this is not done and custom objects that have no place being located in the object hierarchy beneath dm_sysobject end up there just for lack of knowledge. So one needs to at least pose the question of why this approach was taken to the party responsible for the implementation and design of the solution.

Relation vs. attribute
Relation is a mechanism of relating objects to one another, but sometimes using this design approach to the maximum may have adverse effects. Typically problematic scenarios have been raised in cases of management systems where the whole population of a nation is a possible participant in a case. These users are modelled into the repository as custom objects and different types of relations are created between the case and the user.
Let's take for example a nation whose ministry of interior handles ~100K cases a year with approximately ten (10) different participant types each type having on average three (3) participants in them. The cases have a retention period of approximately fifteen (15) years. Similar relations are required for all the actions and files placed into the case which amount to approximately 25 per each case. From this one can calculate that one is fairly fast approaching the billion item mark on the \texttt{dm\_relation\_s} table. These billion rows will contain the following – in this case – mostly unnecessary information:

- \texttt{r\_object\_id} ID
- \texttt{relation\_name} CHAR(32)
- \texttt{parent\_id} ID
- \texttt{child\_id} ID
- \texttt{child\_label} CHAR(32)
- \texttt{permanent\_link} BOOLEAN
- \texttt{order\_no} INTEGER
- \texttt{effective\_date} TIME
- \texttt{expiration\_date} TIME
- \texttt{description} CHAR(255)
- \texttt{i\_partition} INTEGER
- \texttt{i\_is\_replica} BOOLEAN
- \texttt{i\_vstamp} INTEGER

This leads one to take at least for some of the participants a similar approach that is taken with e.g. the field \texttt{owner\_name} in standard Documentum types. This approach is called \textit{denormalization}.

\textbf{Denormalization}

The standard way of designing a relational database is to do \textit{normalisation} which aims for the first normal form where all values are directly dependent on the primary key and the data is free of repeating groups. In larger systems, however, this often leads to poor performance, and \textit{denormalization} is performed to gain some of it back.

\textbf{Aspects}

Looking at the problem \textit{Relation vs. attribute} described above, logical reasoning would say that aspects could easily solve this issue. Maybe they can, but one has yet to see them being used in anger in repositories of substantial size where the utilization of aspects would be commonplace.

One of the known issues is that the aspect attributes do not display Out Of the Box (OOTB) on client properties pages.
Value assistance

One common requirement that is implemented time and time again in different projects is a metadata management tool for value assistance values. Countless hours are used/wasted in creating bespoke applications to manage a few rows in a database table.

The approach one commonly takes is to use standard objects that can be modified using our standard OOTB clients. When creating types a DQL statement like the following is used:

```dql
create type c_val_ass
(c_val_ass_attr string(128) {
    value assistance is
    qry
        'select object_name
         from dm_document
         where folder('/System/Config/ValAss/c_val_ass/c_val_ass_attr') order by 1'
        qry attr = object_name
        is complete
        , set label_text = 'Value assistance example'
        , set is_required = 0
        , set read_only = 0
        , set not_null = 1
        , set ignore Immutable = 0
        , set is_hidden = 0
    )
} with supertype dm_document
set label_text = 'Value assistance'
publish
go
```

Now the customer only needs to manage `dm_document` objects in the folder `/System/Config/ValAss/c_val_ass/c_val_ass_attr` to have full control of their metadata which can be accomplished with any of our OOTB clients. With this approach another common requirement of value assistance value security is fulfilled.

This obviously is a simplified example of what can be accomplished. The solution can easily be taken further with the use of `@value` in the value assistance query or by creating a structure and defining a `selector` to use.

ACL Design

The `dm_acl` to `dm_sysobject` ratio is one of the key factors in repository performance. One has used the ratio of one to ten (1/10) as an optimal value for a repository. This however is not often the case. Typical situations where the `dm_acl` to `dm_sysobject` ratio deteriorate are lack of poor user training, repository design and case management systems.
The first of these is easily remedied by instructing users on how to properly manage the security of their objects. This however is not often possible since the system removes the privileges to do this by application design and applies its own security settings. If the application security is not carefully designed, it will probably lead to a system which is hard to maintain and has a poor \texttt{dm\_acl} to \texttt{dm\_sysobject} ratio.

Last of the three – case management systems – almost always have a requirement that causes the \texttt{dm\_acl} to \texttt{dm\_sysobject} ratio to be poor. The reason for this is that typically each item in a case management system will have their security inherited from multiple different sources and they often have protective markings applied which complicate the security scenario even further.

\textbf{Duplicate identifier}

If one is stuck with how the application assigns security, it does not mean that one is stuck with a poor \texttt{dm\_acl} to \texttt{dm\_sysobject} ratio. Even if the repository has a one to one (1/1) \texttt{dm\_acl} to \texttt{dm\_sysobject} ratio, it does not mean that no two \texttt{dm\_acl} objects would be the same. In fact quite the contrary, there is a very high probability that more than one \texttt{dm\_acl} objects grant the exact same privileges. If one is in the territory of custom \texttt{dm\_acl} objects (\texttt{dm\_45...}) which should never be reused anyway or if one’s application assumes full control of the security, one is safe to modify the \texttt{dm\_acl} settings so that more than one \texttt{dm\_sysobject} objects point to a single \texttt{dm\_acl} object.

Obviously there are multiple ways of identifying duplicate \texttt{dm\_acl} objects in the repository, but one fairly usable way is to alphabetically organize the accessors (users & groups) that have been granted privileges in the ACL, then to concatenate a string that contains values from attributes \texttt{r\_accessor\_name}, \texttt{r\_accessor\_permit}, \texttt{r\_accessor\_xpermit}, \texttt{r\_is\_group}, \texttt{r\_permit\_type}, \texttt{r\_application\_permit}.

Given that a partial dump of the ACL would look like:

dump,c,4500162e800000100
...
object_name : dm\_4500162e800000100
  r\_accessor\_name : [0]: dm\_world
                        [1]: dm\_owner
                        [2]: docu
  r\_accessor\_permit : [0]: 3
                        [1]: 7
                        [2]: 5
  r\_accessor\_xpermit : [0]: 0
                        [1]: 0
                        [2]: 3
  r\_is\_group : [0]: F
                 [1]: F
                 [2]: T
  r\_permit\_type : [0]: 0
                  [1]: 0
                  [2]: 0
  r\_application\_permit : [0]:
                          [1]:
                          [2]:
The generated string would be: dm_owner70F0dm_world30F0docu53T0. From this generated string a hash code -1529768294 is created which uniquely identifies the security granted by this ACL. The hash code is then stored either in the dm_acl object description field, a separate table with the ACL name, owner and hash code or whatever mechanism is readily available to persistently store this information. Then using the mechanism of choice (job & method is a typical way of accomplishing this) duplicate hash code fields are identified and all the dm_sysobject objects having the same security are pointed to a single ACL object. To make sure that the redundant ACL objects are removed from the system the Dmclean (called dm_DMClean in DA) job should be turned on to remove all orphaned ACL objects.

**Group membership**

One of the approaches to manage security in a repository is to add users into groups and then grant those groups privileges. This is a valid approach, to which there however is a caveat that may rear its ugly head if a single user’s group membership gets out of hand, which it commonly does if the repository security has not been properly designed.

When a query returns dm_sysobject instances, the Content Server (CS) creates an SQL statement that differs depending on the amount of the user's group membership. Tests indicate that if a user belongs to less than 250 groups these group names are added to the generated SQL statement, but if the user belongs to more than 250 groups a sub-query is generated instead.

The poor performance for users that belong to a large number of groups has been investigated by the EMC performance group. As it stands there are no immediate plans to change the way in which the resulting SQL is created.

If this might be an issue the application design should change or at least it should be made sure that proper indexes are in place. To make sure the following API statements (or just the SQL) should be issued:

```sql
execsql,c,drop index idx_dm_group_r_id_name
execsql,c,create index idx_dm_group_r_id_name on dm_group_r(users_names, r_object_id) compute statistics
```
Summary

This document set out to cover erroneous practices found in DFC and WDK programming, but while coming up with the most devastating practices encountered while working with customer implementations it was clear that some of the design choices could not be remedied with any amount of good coding, so some of the more severe ones were discussed. Some general principles of project work were also discussed bearing in mind that wrong practices will have a big impact on the implementation project as a whole, which led to the document in parts being a best practice paper.

This document aims to give a heads up for people either in a technical lead position or doing code/application reviews as to what they should keep an eye out for. It obviously covers only a fraction of the cases where our system is not used to its full potential, but the ones listed are severe and commonplace among the community that develop solutions that use Documentum.
Biography

I have worked with core Documentum products since July of 1998 at a Finnish partner TietoEnator Oy. I first started in support and moved to consultancy work cutting my teeth with a 5 year stint at Nokia. After 7 years at TietoEnator in different technical lead positions in Documentum projects EMC² acquired Documentum. This opened up a realistic possibility to work for the vendor since EMC² had a country office in Finland. This is a venue I pursued and have now successfully worked for over three years at EMC².

Proven Professional certifications

- **Associate**
  - EMC Content Management Foundations

- **Specialist**
  - Content Management Server Programming Specialist Version 5 (EMCApD)
  - Content Management Server Programming Specialist Version 6 (EMCApD)
  - Content Management Web Application Programming Specialist Version 5 (EMCApD)
  - System Administrator, Content Management System Administration Specialist Version 5 (EMCSyA)
  - System Administrator, Content Management System Administration Specialist Version 6 (EMCSyA)
  - Technology Architect, Content Management Systems Architecture Specialist Version 6 (EMCTA) (Pending the exam to go live)