Paper 017-2012

SAS® IOM AND YOUR .NET APPLICATION MADE EASY

Karine Désilets, Statistics Canada, Ottawa, Ontario, Canada

ABSTRACT

At Statistics Canada, many statistical systems are implemented in a client-server development context allowing maximum use of SAS tools, software and solutions. This article focuses on development of Microsoft .Net client applications using SAS Integrated Object Model (IOM) to take advantage of the processing, analysis, reporting and data storage power of SAS.

This article covers a number of best practices such as the different communication modes between Microsoft .Net and SAS, types of SAS code execution, parameter management, libref and fileref management, error management with raised event, data acquisition via Ado.Net, automated analysis of SAS log files and management of customized return codes between SAS and Microsoft .Net. Finally, a few broader topics associated with this work and future research projects are addressed.

INTRODUCTION

This article is intended for programmers who want to build applications in a Microsoft .Net environment that utilize SAS technology via SAS IOM, a component of SAS Integration Technologies. This avenue has already been explored in articles (2), (5), (6), (7) and (12), which deal with specific tasks.

The IOM's full potential is exploited here by defining a class, called *SasEasyIom*, which manages the main elements needed to build an application. This article's unique contribution lies in the integrative impact of the *SasEasyIom* class and the best practices associated with its use. It provides developers with everything they need to integrate SAS with a client technology such as Microsoft .Net.

SASEASYIOM CLASS OVERVIEW

The *SasEasyIom* class is written in C# in a Microsoft .Net environment (the code is presented in Appendix A). It contains all the properties, methods and events required to manage the SAS connection.

For convenience, *mySasCon* object, which is an instance of the *SasEasyIom* class, is referenced throughout the examples presented in this article. This unique object contains the IOM's main functionalities. Using the object makes the IOM's functionalities even more transparent and much simpler. The *SasEasyIom* class offers programmers the following functionalities:

- Open and close a SAS session in local mode, client/server mode and client/server mode by logical name (openWS and closeWS)
- Submit SAS code (submitSASCode)
- Submit stored processes (SubmitStoredProcess)
- Assign and deassign librefs and filerefs (using the FileService and DataService interfaces)
- Acquire data using Ado.Net (getDS, closeDS)
- Analyze the log automatically (log, analyzeLog)
- Return customized return codes between a SAS program and a Microsoft .Net environment (getErrCode)
- Manage errors by events: *StepError*, *SubmitComplete*, *ProcStart*, *ProcComplete*, *DatastepStart* and D*atastepComplete*

COMMUNICATION BETWEEN MICROSOFT .NET AND SAS

The foundation for communications between a Microsoft .Net client and SAS can be divided into three modes: local, client/server by server name and client/server by logical name via a metadata server. Of course, it is preferable to use the power of the SAS Metadata Server since that technology makes it possible to connect transparently to a server without knowing its exact address on the network. A change in a server's address has no impact on the .Net application code. Finally, the server security is managed through the SAS Metadata Server.

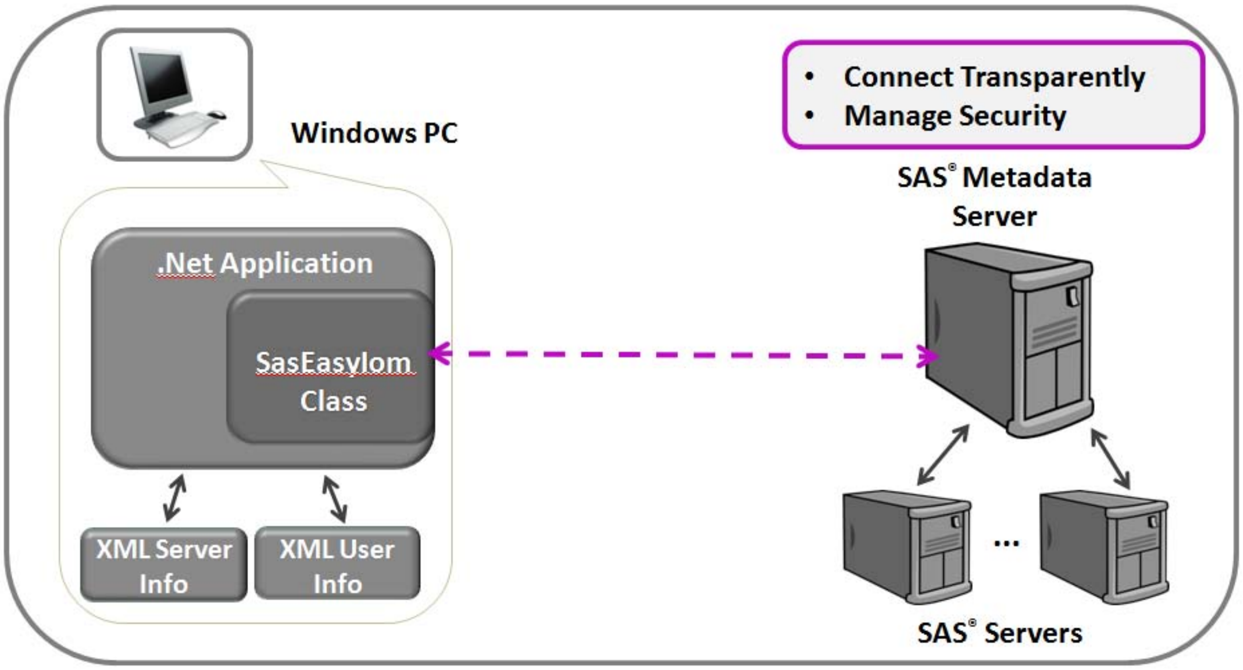

Figure 1. Communication between Microsoft .Net application and SAS in the context of SAS Metadata Server

CONNECTION BY LOGICAL NAME

Connection by logical name requires two XML configuration files containing the connection metadata. The XML files are read by the *SetMetadataFile* method. The system configuration file contains the following information: the port, the server name, the connection type and the communication protocols. It may also contain the username and the password, but if a user configuration file exists, it will take precedence over the system configuration file. The user's configuration file contains information about the user, the password and the domain.

Those configuration files can be created by either running the SAS Integration Technologies Configuration Wizard (*Itconfig2.exe*) or by editing the two XML files directly. The first method provides special interfaces that allow the user to change the connection configuration settings and save the results in XML files. It is recommended that the wizard be used to test the connection with the XML files. It is also possible to automate the creation of the two XML files directly in the .Net code. For more information about this technique, see (11). The two following examples show two different instances of XML files, one containing information about the server and the other about the user.

XML file containing information about the server

```
<?xml version="1.0" encoding="UTF-8" ?>
<Redirect>
  <LogicalServer Name="Open Metadata Server"
                ClassIdentifier="0217E202-B560-11DB-AD91-001083FF6836">
    <UsingComponents>
      <ServerComponent Name="Open Metadata Server"
                      ClassIdentifier="0217E202-B560-11DB-AD91-001083FF6836">
        <SourceConnections>
          <TCPIPConnection Name="Open Metadata Server" Port="8561"
                          HostName="disneysasmeta.statcan.ca"
                          ApplicationProtocol="Bridge" CommunicationProtocol="TCP">
          </TCPIPConnection>
        </SourceConnections>
      </ServerComponent>
    </UsingComponents>
  </LogicalServer>
</Redirect>
```

XML file containing information about the user

```
<?xml version="1.0" encoding="UTF-8" ?>
  <AuthenticationDomain Name="DefaultAuth">
    <Logins>
      <Login Name="statcan\mickey" UserID="statcan\mickey"
             Password="{base64}SnVzdGluTmF="></Login>
    </Logins>
  </AuthenticationDomain>
```

The *SasEasyIom* class uses these two XML files to connect. Below is an example of how it connects from the class interface to the *DISNEY - PROD - Logical Workspace Server*:

```
SasEasyIom mySasCon = new SasEasyIom();
bool myConnect = false;

mySasCon.logicalName = "DISNEY - PROD - Logical Workspace Server";
mySasCon.ServerXmlInfo =
                        "C:\\ProgramData\\SAS\\MetadataServer\\oms_serverinfo.xml";
mySasCon.UserXmlInfo =
    "C:\\Users\\mickey\\AppData\\Roaming\\SAS\\MetadataServer\\oms_userinfo.xml";

myConnect = mySasCon.openWS(2);
if (myConnect == true)
{
       // program implementation
}
```

The C# code that establishes the connection uses the two metadata files (MyserverFileName and myUserFileName) and the server's logical name (MyLogicalName in the example below). That code is extracted from the *OpenWS* method of the *SasEasyIom* class. The class also offers the option of connecting in local mode and client/server mode by server name. Here is the code for a logical name connection in client/server mode:

```
obObjectFactory.SetMetadataFile(MyserverFileName, myUserFileName, false);
mySAS = (SAS.Workspace)obObjectFactory.CreateObjectByLogicalName(MyLogicalName,
         "");
Con.ConnectionString = "provider=sas.iomprovider.1;sas workspace ID=" +
                       mySAS.UniqueIdentifier;
Con.Open();
```

SAS CODE EXECUTION

SAS code is executed through Microsoft .Net by calling the *SubmitSASCode* method of the *SasEasyIom* class. This method allows running previously created SAS code. There are several ways of creating SAS code.

Executing dynamic SAS code directly in C# code

The first technique is to create the code dynamically in the .Net environment. This technique is advantageous for short calls; however, maintenance of the code can quickly become cumbersome:

```
string SasCode = null;
SasCode = "data bestCharacters;  name = 'Mickey';  rank = 1; run;";
mySasCon.SubmitSASCode(SasCode);
```

Executing SAS or macro code from an external (.sas) file

Another way is to include the code in an external .sas file. This makes it possible to add longer programs that are independent of the .Net code. The presentation layer is separated from the processing layer. Furthermore, the code may contain partial or complete programs or macros.

```
string SasCode = null;
SasCode  = "%include  'C:\\Orlando\\programs\\helloMickey.sas';";
mySasCon.SubmitSASCode(SasCode);
```

Executing stored compiled macro

Calling stored compiled macros has a speed advantage over calling external SAS (.sas) files since they are already compiled.

```
string SasCode = null;
SasCode  = "libname disney 'C:\\Orlando\\programs';  " +
           "options mstored " + " sasmstore = disney; " + " %helloPluto; ";
mySasCon.SubmitSASCode(SasCode);
```

In general, the use of macros and stored compiled macros helps generate complex, dynamic, reusable and easily maintainable code.

Executing stored process

The fourth way is to call stored processes. A stored process is a server-based program that is parameterized and transparent to the client. As well, like calling compiled macros, calling stored processes separates the SAS processing layer from the client layer. The key feature of stored processes is that they are called without using SAS language (10):

```
mySasCon.SubmitStoredProcess("C:\\spRepository", "createDisneyGraph", "")
```

The above `SubmitStoredProcess` method takes three parameters. The first one is the directory where the stored process is located, the second one is the stored process filename and the third one is the parameter's name and his associated value (ex: "param1 = 1").

INPUT MANAGEMENT

In programming, it is essential to have techniques for parameterizing programs so that they are reused under various circumstances. In a Microsoft .Net environment, a number of techniques are used to parameterize the SAS code. One technique is to define SAS variables dynamically in the C# code. Again, this technique is for short calls and should be used sparingly as maintenance of the code quickly becomes cumbersome.

Other methods include using the SYSPARM option, creating external files containing the variables, calling macros (stored in .sas files or catalogues) and calling parameterized stored processes. Again, the last two techniques are preferred because they separate the client layers from the processing layers.

LIBREF MANAGEMENT

In SAS, librefs are particularly useful because they serve as labels that are temporarily assigned to folders. For example, the storage location, in the form of a file path, can be determined from the libref. Two forms of path assignment are available. One creates the libref directly by submitting SAS code (9):

```
string SasCode = null;
SasCode = "libname disney 'C:\\Orlando\\programs';";
mySasCon.SubmitSASCode(SasCode);
```

This method is simple and direct. However, it does not provide access to the various configurations available with the *DataService* class. Another technique uses the *AssignLibref* method available in the *DataService* class of a *Workspace*:

```
mySasCon.obDataServ.AssignLibref("disney", "", "C:\\Orlando\\programs", "");
```

This approach is much more complete in the fact that it takes advantage of the classes available with the *Workspace*, because it can check whether a libref has been correctly assigned and it can make use of the other options available in that class. The interface is also transparent to the SAS code.

Writing the contents of a Microsoft Excel file in a SAS data set

This example is a typical use case in which the *DataService* class can be frequently used:

```
mySasCon.obDataServ.AssignLibref("xlsData", "excel", "C:\\Orlando\\charac.xls","");
mySasCon.SubmitSASCode("data bestCharacters; set xlsData.'best$'n; run;");
mySasCon.obDataServ.DeassignLibref("xlsData");
```

Reading a Microsoft Excel file through the SAS Excel libname engine is straightforward. The workbook (charac.xls) acts as a library and the spreadsheet (best) acts as a data set. Once the Excel file is read (and not needed for other purposes), the reference to the libref xlsData can be deassigned and the Excel file closed.

Assigning a macro catalogue and executing a stored compiled macro

As previously showed, macro catalogues are useful and have a speed advantage. The following example shows how to use them correctly with the *DataService* class:

```
mySasCon.obDataServ.AssignLibref("disney", "", "C:\\Orlando\\programs", "");
SasCode = "options mstored sasmstore = disney; %helloPluto()";
mySasCon.submitSASCode(SasCode);
```

FILEREF MANAGEMENT

File handling and management are important steps in programming. In the IOM, there is a *FileService* class for the management of files and filerefs. Each SAS *Workspace* that is created has its own *FileService* object. In fact, in this interface, when direct references to the server are avoided, it is possible to create applications that will run properly on a number of platforms (8).

Managing a temporary fileref and a permanent fileref

The TEMP and DISK devices are used in the following example. TEMP creates a temporary file, stored in the same folder as the WORK library. The file exists only as long as it is assigned in a SAS session, which is useful when the file’s location is unimportant and has to be temporary. The DISK device refers to a permanent location.

```
string fileRef1 = null;
string fileRef 2= null;

mySasCon.obFileServ.AssignFileref("park", "DISK", "C:\\Orlando\\data\\park.txt",
                                  "", out fileRef1);
mySasCon.obFileServ.AssignFileref("tmpPark", "TEMP", "", "", out fileRef2);

SasCode = "data wdwPark; infile park; input ParkId rank; run; ";
mySasCon.submitSASCode(SasCode);
SasCode = "data _null_; set wdwPark; file tmpPark;  put parkId rank; run; ";
mySasCon.submitSASCode(SasCode);

mySasCon.obFileServ.DeassignFileref("park");
mySasCon.obFileServ.DeassignFileref("tmpPark");
```

Reading an XML file

The example below shows that it is sometimes necessary to use a combination of *filerefs* and *librefs* to access certain files. Among the latter are the XML files:

```
mySasCon.obFileServ.AssignFileref("xmlData", "DISK",
                             "C:\\Orlando\\data\\characters.xml", "", out MyName);
mySasCon.obFileServ.AssignFileref("xsd", "DISK",
                             "C:\\Orlando\\data\\schemaChar.xsd", "", out MyName);
```

```
mySasCon.obDataServ.AssignLibref("xmlBest", "xml", "",
                      "xmlfileref = xmlData  xmlschema = xsd xmltype = generic " +
                      "ODSCHARSET = 'utf-8' xmlmeta = schemadata");

SasCode = "data bestCharacters; set xmlBest.MICROVARIABLE_RECORD; run;";
mySasCon.submitSASCode(SasCode);
```

With the use of the SAS XML libname engine, an XML document can be imported as a SAS data set or a SAS data set can be exported as an XML document. In order to achieve that, the filerefs are used to assign the XML document and its associated schema file and next, through the SAS XML libname engine the XML document has the possibility to be imported or exported.

Methods for managing and manipulating files

In this interface, there are several methods for managing and handling files in the .Net environment.

```
string result1 = null;
string result2 = null;
string fullName = null;
string mypath = null;
```

Making a directory

```
mypath = mySasCon.obFileServ.MakeDirectory("C:\\", "Orlando");
```

The directory "C:\\Orlando" is created.

Splitting a directory name

```
mySasCon.obFileServ.SplitName("C:\\Orlando", out result1, out result2);
Debug.Print(result1 + result2);
```

result1 = Orlando and result2 = C:\\

Splitting a file name

```
mySasCon.obFileServ.SplitName("C:\\Orlando\\hollywoodStd.sas", out result1,
                              out result2);
```

result1 = hollywoodStd.sas and result2 = C:\\Orlando

Renaming a file

```
mySasCon.obFileServ.RenameFile("C:\\Orlando\\magicK.sas7bdat",
                              "C:\\Orlando\\mKingdom.sas7bdat");
```

The final file is now called "C:\\Orlando\\mKingdom.sas7bdat".

Deleting a file

```
mySasCon.obFileServ.DeleteFile("C:\\Orlando\\epcot.sas7bdat");
```

The file "C:\\Orlando\\epcot.sas7bdat" has been deleted.

Creating a full name file

```
fullName = mySasCon.obFileServ.FullName("animalKingdom.sas7bdat", "C:\\Orlando");
```

The full name file "C:\\Orlando\\animalKingdom.sas7bdat" has been created.

ERROR MANAGEMENT WITH RAISED EVENTS

When a SAS program is run from a Microsoft .Net environment, events may be captured by the .Net application. Errors arising in SAS programs are managed with the events *StepError*, *DatastepStart*, *DatastepComplete, ProcStart*, *ProcComplete* and *SubmitComplete*, all available in the *SasEasyIom* class. They detect when an error occurs in the code, when a DATA or PROC step begins or ends, and when submission of the program is completed. They are very useful for validating proper execution of the code or detecting errors. They can be managed in various ways in the .Net environment and depend on the application's context.

When an error occurs in a SAS program, the *reset* or *cancel* methods can be used to manage the state of the *LanguageService* (1) in the .Net environment. The *reset* method puts *LanguageService* back in its initial state. It is useful for extricating *LanguageService* of an error associated with the execution of invalid syntax or an incomplete program (1). The *cancel* method interrupts execution of the program submitted, and *LanguageService* executes the *reset* method.

ACQUISITION OF DATA VIA ADO.NET

Running SAS programs also involves processing data files. The latter are stored permanently or temporarily in libraries. They generally contain data in the form of *data sets* or SAS views. The data can be accessed in a Microsoft .Net environment by connecting to the OLE DB data source via Ado.Net.

In this case, the connection to the data source is created with the opening of the *Workspace* and remains associated with the *Workspace* through a unique identifier. With this unique identifier workspace property, the connection can then be reused to acquire data, in the form of a *data set* object, with the *getDS* method. Please note that SQL commands must be used to acquire the data when the *getDS* method is called.

Acquiring SAS data from the sashelp.class data set

```
SasCode = "select name from sashelp.class; ";
ds = mySASconnection.getDS(SasCode);
foreach (DataRow row in ds.Tables[0].Rows) // Loop over the rows.
{
    foreach (var name in row.ItemArray) // Loop over the items.
    {
        Debug.Print("Item: " + name); // Print label.
    }
}
```

It is important to keep in mind that when a *Workspace* is created in the Microsoft .Net environment, the SAS session created is equivalent to one in batch mode. Consequently, the data available is in the W*ork*, S*ashelp* or *Sasuser* libraries or some other assigned library.

AUTOMATED ANALYSIS OF SAS LOG FILES

In a Microsoft .Net environment, the SAS log file is read using the *FlushLog* and *FlushLogLines* methods available with the IOM's APIs. *FlushLog* returns the log in the form of character strings, while *FlushLogLines* returns the log's components, or line types (including *LineTypeError* messages and *LineTypeWarning* messages), so that each line of the log can be analyzed.

In the *SasEasyIom* class, two methods that implement these interfaces were developed. One of them, the *log* function, reads the log and returns it in the form of character strings. The other, *AnalyzeLog*, is a procedure that analyzes the SAS log and quickly identifies errors and warnings. It produces three output files: the log file, the error file and the warning file for efficient debugging. This method is very useful after a SAS program is executed.

Calling the log and AnalyzeLog methods

```
logFile = "C:\\Orlando\\logs\\mylog.log";
warnFile = "C:\\Orlando\\logs\\warning.log";
errFile = "C:\\Orlando\\logs\\error.log";

mySasCon.ServerName = "localhost";
```

```
SasCode = "data bestCharacters;  name = 'Mickey';  rank = 1 ; run; ";
myConnect = mySasCon.openWS(0);
if (myConnect == true)
{
    mySasCon.submitSASCode(SasCode);
    string saslog = (string) mySasCon.log();
    Debug.Print(saslog);
    SasCode = "data bestCharacters;  set minnie;  rank = 2 ; run; ";
    // Error if data set called minnie doesn't exist
    mySasCon.submitSASCode(SasCode);
    mySasCon.analyzeLog(logFile, warnFile, errFile);
    mySasCon.closeWS();
}
```

The *log* and *analyzeLog* methods could be used in a completely different context. For more details on the interfaces provided by the *LanguageService* object and the possibilities offered by the *FlushLog* and *FlushLogLines* methods, see the documentation available under Integration Technologies (1).

CUSTOMIZED RETURN CODE FROM SAS TO MICROSOFT .NET

Using events to manage errors as described in section 6 has a finite set of use cases. It is here that the addition of a return code between SAS and .Net becomes appropriate: it provides the SAS programmer with the flexibility to create customized error codes. This approach requires the addition of the code in pink below in order to capture an error in a SAS macro.

Template of a macro that permits customized return codes

```
%macro testErrCode(param=) ;
    %let procerr=;
    %if %substr(&param,1,1) ne _  %then %do;
        %let procerr= 1;
        %put *********************ERROR******************************************;
        %put ERROR : Parameters name must begin with an underscore(_ParameterName);
        %put *******************************************************************;
    %end;
    %if (&procerr ne ) %then %goto exit;
    %goto end;
    %EXIT: %abort cancel file &procerr;
    %END:  %put "end testErrCode.sas";
%mend  testErrCode;
```

The label *%EXIT* with *%abort* halts execution of the macro, the DATA step, the SAS program or the SAS session. The addition of the *Cancel* option halts the elements that have just been submitted while the addition of the *File* option prevents only the contents of the *autoexec* file or the *%INCLUDE* file from being erased by the *%ABORT* element. The addition of the *&procerr* transfers the variable's value to the automatic macro variable *SYSINFO*. At this point, the client program can read the value of *SYSINFO*, which is stored in the *sashelp.vmacro* table. This technique requires the SAS programmer to add error codes throughout the program and associate them with the *procerr* variable. This approach makes it possible to create customized error types and transfer them to the client.

In order to avoid the addition of error codes throughout the program, the client program can also read the contents of the automatic macro variable *SYSERRORTEXT*, which is also stored in the *sashelp.vmacro* table. This variable, which is updated automatically, contains the text of the last error in the SAS program.

Finally, effective error management involves the use of several strategies, including raised events, automated log analysis and customized error codes.

Calling the testErrCode macro from the Microsoft .Net environment

```
mySASCode = "%include 'C:\\Orlando\\programs\\testErrCode.sas'; ";
mySASconnection.SubmitSASCode(mySASCode);
mySASCode = "%testErrCode(param=_WaltDisney);";
mySASconnection.SubmitSASCode(mySASCode);
errCode = mySASconnection.GetErrCode();
System.Windows.Forms.MessageBox.Show("ERROR CODE VALUE :" + errCode);
//return 0 and no ls_StepError is raised
mySASconnection.closeWS();
```

CONCLUSION

In concrete terms, we have laid the foundation for the construction of client/server infrastructures and applications that combine the power of the SAS IOM with Microsoft .Net and SAS technologies. A developer interested in developing applications that link Microsoft .Net and SAS now has all the tools required to effectively manage those technologies.

The work done so far with the *SasEasyIom* class has been in an R&D environment with the aim of validating the methodologies. Other elements or variants may be added to what has already been developed. For example, the class may need to be adjusted to reflect specific characteristics associated with the application's domain. For that reason, the documentation available in (1), (3), (8), (9) and (11) is very important and can be consulted to ensure that the available classes can be judiciously exploited.

Furthermore, these technologies are interconnected, and when they combine and interact, the result is very powerful. Depending on the domain, many other use cases are possible and should be dealt with specifically. We have barely scratched the surface. The creative process associated with these technologies is not over.

REFERENCES

1. *C:\Program Files\SAS\Shared Files\Integration Technologies - fichiers : sasoman.chm, sas.chm, saswman.chm.*
2. *SAS® Integration Technologies, UNIX and Visual Basic .Net Integration Procedure.* **Chevrette, Antoine.** Ottawa, Canada : Sas Global Forum, 2008. paper 011-2008.
3. *SAS® 9.2 Integration Technologies: Windows Client Developer's Guide.* Cary, NC : SAS Institute Inc, 2009. ISBN 978-1-59994-847-8.
4. *SAS® 9.2 Integration Technologies: Overview.* Cary, NC : SAS Institute Inc, 2009. ISBN 978-1-59994-851-5.
5. *Enterprise Integration Technologies What is it and what can it do for me?* **Vodicka, Scott.** Cary, NC : SAS, 2000.
6. *Using IOM and Visual Basic in SAS® Program Development.* **Greg Silva.** Cambridge, MA : Biogen, Inc., 2003.
7. *Access to SAS® Data Using the Integrated Object Model (IOM) in version 9.1.* **Pratter, Frederick.** Eastern Oregon University, La Grande, OR : PharmaSUG 2005 AD05, 2005.
8. Developing Windows Clients - FileService Object. *FileService Object.* [Online] [Cited: 2011-18-08.] http://support.sas.com/rnd/itech/doc/dist-obj/comdoc/fsvca.html.
9. Developing Windows Clients - DataService Object. *DataService Object.* [Online] [Cited: 2011-18-08.] http://support.sas.com/rnd/itech/doc9/dev_guide/dist-obj/comdoc/iyhlca.html.
10. *Creating and Using SAS® Stored Processes.* **Eric Rossland, Kari Richardson.** Philadelphia, Pennsylvania : SAS Institute Inc., 2005. Paper 135-30.

11. Sample 26056: Microsoft Visual Studio 2005 C# Code Snippets. [Online] [Cited: 2011-18-08.] http://support.sas.com/kb/26/056.html.

12. *Through the Looking Glass:Two Windows into SAS®.* **Peter Eberhardt, Richard A. DeVenezia.** Toronto, Canada : SAS Institute Inc., 2005, Vols. SUGI-30. Paper 003-30.

13. *SAS® Integration Technologies - Expanding your choices for integrating SAS Intelligence.* [Online] 2011. [Cited: 2011-29-06.] http://www.sas.com/resources/factsheet/sas-integration-technologies-factsheet.pdf.

ACKNOWLEDGMENTS

I would like to thank Chantal Marquis and Yves Deguire first for giving me the opportunity to explore this technology and then for encouraging me to write about it. I would also like to thank Antoine Chevrette and Marc Thomson, who strongly encouraged me to write and revise this article. Finally, I would like to thank everyone at Statistics Canada's SAS Technology Centre who works directly or indirectly with me. It is a pleasure to work with all of you.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Karine Désilets
Statistics Canada
R.H. Coats 100 Tunney's Pasture Driveway
Ottawa Ontario, K1A 0T6
613-951-3948
karine.desilets@statcan.gc.ca

APPENDIX A – *SASEASYIOM* CLASS

```
public class SasEasyIom
{
    SAS.Workspace mySAS; // SAS session
    SAS.LanguageService ls; //Submitting SAS language
    SAS.StoredProcessService SASproc;//SAS stored process

    SASObjectManager.ObjectFactoryMulti2  obObjectFactory = new
    SASObjectManager.ObjectFactoryMulti2();
    SASObjectManager.ServerDef obServer = new SASObjectManager.ServerDef();
    SASObjectManager.ObjectKeeper objectkeeper = new
    SASObjectManager.ObjectKeeper();

    public SAS.FileService obFileServ;  // manage filerefs
    public SAS.DataService obDataServ; // manage librefs

    System.Data.OleDb.OleDbConnection Con = new
    System.Data.OleDb.OleDbConnection();
    DataSet ds = new DataSet();
    System.Data.OleDb.OleDbDataAdapter oDA = new
    System.Data.OleDb.OleDbDataAdapter();

    private string MyLogin;  //username
    private string MyPassword;//password
    private int MyServerPort; //port info
    private string MyServer;  //serverName
    private string MyLogicalName; //server logical name
    private string MyserverFileName;  //Xml server filename
    private string myUserFileName; //Xml user filename

    //********** serverName **********//
    public object serverName {
        get { return MyServer; }
        set { MyServer = (string)value; }
    }
    //********** port  **********//
    public object port {
        get { return MyServerPort; }
        set { MyServerPort = (int)value; }
    }
    //********** login  **********//
    public object login {
        get { return MyLogin; }
        set { MyLogin = (string)value; }
    }
    //********** password  **********//
    public object password  {
        get { return MyPassword; }
        set { MyPassword = (string)value; }
    }
    //********** logicalname  **********//
    public object logicalName {
        get { return MyLogicalName; }
        set { MyLogicalName = (string)value; }
    }

    //********* server XML info  *******//
    public object serverXmlInfo {
        get { return MyserverFileName; }
        set { MyserverFileName = (string) value; }
    }
    //********* user XML info  **********//
    public object userXmlInfo  {
        get { return myUserFileName; }
        set { myUserFileName = (string) value; }
    }

    //*************************************
    //open a workspace
    //*************************************
    public bool openWS(int conType)  {
      Boolean connect;
      connect = false;

      if (conType == 0)  //local host
      {
         obServer.MachineDNSName = MyServer;
         obServer.Port = MyServerPort;
         obServer.Protocol = SASObjectManager.Protocols.ProtocolCom;
         mySAS = (SAS.Workspace)obObjectFactory.CreateObjectByServer ("myServer" +
                   conType.ToString(), true, obServer, MyLogin, MyPassword);
      }
      else if (conType == 1)//by serverName
      {
          obServer.MachineDNSName = MyServer;
          obServer.Port = MyServerPort;
          obServer.Protocol =
          SASObjectManager.Protocols.ProtocolBridge;
          mySAS = (SAS.Workspace)obObjectFactory.CreateObjectByServer("myServer" +
                    conType.ToString(), true, obServer, MyLogin, MyPassword);
      }
      else if(conType == 2)//by logicalName
      {
        obObjectFactory.SetMetadataFile(MyserverFileName,
        myUserFileName, false);
        mySAS = (SAS.Workspace)
                  obObjectFactory.CreateObjectByLogicalName(MyLogicalName, "");
        System.Windows.Forms.MessageBox.Show("SAS server running on: " +
        mySAS.Utilities.HostSystem.DNSName);
      }

     configureSASLanguageEvents(mySAS); //Instantiate sasLSevents
     obDataServ = mySAS.DataService;
     obFileServ = mySAS.FileService;
     objectkeeper.AddObject(1, "myServer" + conType.ToString(), mySAS);

      if (MyServer.ToUpper() == (string)"LOCALHOST")
      {
        Con.ConnectionString = "provider=sas.iomprovider.1; sas workspace ID=" +
                               mySAS.UniqueIdentifier + "; Data Source=_LOCAL_";
      }
      else
      {
        Con.ConnectionString = "provider=sas.iomprovider.1; sas workspace ID=" +
                              mySAS.UniqueIdentifier;
      }
      Con.Open();
      //Remove obSAS form the object keeper
      objectkeeper.RemoveObject(mySAS);

      if ((ConnectionState)ConnectionState.Open == Con.State)
      {
       connect = true;
      }
      return connect;
    }
    //*************************************
    // Procedure closeWS
    //*************************************
    public void closeWS() {
      closeDS();
      mySAS.Close();
    }
    //*************************************
    // Procedure submitSASCode
    //*************************************
    public void submitSASCode(string strSASCode) {
      SAS.LanguageService ls = default(SAS.LanguageService);
      string[] arSource = new string[2];
      ls = mySAS.LanguageService;
      arSource[0] = strSASCode;
      System.Array linesVar = arSource;
      ls.SubmitLines(ref linesVar);
    }
    //*************************************
    // Procedure SubmitStoredProcess
    //*************************************
    public void SubmitStoredProcess(string repository, string storedProcess,
                                    string parameter) {
       SASproc = mySAS.LanguageService.StoredProcessService;
       SASproc.Repository = "file:" + repository;
       SASproc.Execute(storedProcess, parameter);
```

```
    }
  //*************************************
  // Procedure closeDS
  //*************************************
  public void closeDS() {
    if (((oDA != null)))
    {
      oDA.Dispose();
      oDA = null;
    }
    if (((Con != null)) && (Con.State != ConnectionState.Closed))
    {
      Con.Close();
      Con = null;
    }
    if (((ds != null)))
    {
      ds.Dispose();
      ds = null;
    }
  }
  //*************************************
  // Function getDS
  //*************************************
  public DataSet getDS(string mySelectQuery) {
    DataSet dst = new DataSet();
    bool r = false;
    try
    {
      r = oDA.ContinueUpdateOnError;
      oDA.SelectCommand = new
         System.Data.OleDb.OleDbCommand(
                     mySelectQuery, Con);
      oDA.Fill(dst);
      return dst;
    }
    catch (Exception ex)
    {
    System.Windows.Forms.MessageBox.Show("Error :" + ex);
    return null;
    }
  }
  //*************************************
  // Function getErrorCode
  //*************************************
  public int getErrCode()  {
    int errCode = 0;
    string mystring;
    mystring = "select name, value from sashelp.vmacro where name = 'SYSINFO' " +
               " or name = 'SYSERR' " + " order by name desc; ";
    ds = getDS(mystring);
    submitSASCode("data _null_;run;");
    try
    {

      if (((string)ds.Tables[0].Rows[0][0] == "SYSINFO" &
        (string)ds.Tables[0].Rows[0][1] != "0") &
        ((string)ds.Tables[0].Rows[0][0] == "SYSERR" &
        (string)ds.Tables[0].Rows[0][1] != "0"))
      {
      errCode = Convert.ToInt32(ds.Tables[0].Rows[0][1]);
      }
      return errCode;
    }
    catch (Exception)
    {
      ls.Reset();
      errCode = -2;
      return errCode;
    }
  }
  //*************************************
  // Function log - Parse Sas log
  //*************************************
  public object Log(){
     return (mySAS.LanguageService.FlushLog(10000000));
  }
  //*************************************
  // Procedure analyzeLog
  //*************************************
  public void analyzeLog(string outLog, string outWarning, string outError)  {
    bool bMore = true;
    System.Array CCs = null;
    const int maxLines = 1000000000;
    System.Array lineTypes = null;
    System.Array logLines = null;
    string log = null;
    string errorTxt = null;
    string ParseLineType = null;
    string warningTxt = null;

    warningTxt = "";
    errorTxt = "";
    log = "";

    while (bMore)
    {
         mySAS.LanguageService.FlushLogLines(maxLines, out CCs, out lineTypes,
                                             out logLines);

         for (int i = 0; i <= logLines.Length - 1; i++)
         {
                log += (Convert.ToString(
                logLines.GetValue(i)) + Environment.NewLine);
         }

         for (int k = 0; k <= logLines.Length - 1; k++)
         {
                ParseLineType =     Convert.ToString(lineTypes.GetValue(k));
                if (ParseLineType == "LanguageServiceLineTypeError")
                {
                       errorTxt += ("The line is an error message line:  " +
                                    Convert.ToString(logLines.GetValue(k)) +
                                    Environment.NewLine);
                }
                else if (ParseLineType == "LanguageServiceLineTypeWarning")
                {
                       warningTxt += ("The line is a warning message line:  " +
                                    Convert.ToString(logLines.GetValue(k)) +
                                    Environment.NewLine);
                }
         }

         if (logLines.Length < maxLines)
         {
                bMore = false;
```

```
            }
        }

        //Print errors in a file
        if (errorTxt.ToString().Length != 0)
        {
             using (StreamWriter outfile = new StreamWriter(outError))
             {
                    outfile.Write(errorTxt.ToString());
             }
        }

        // Print warning in a file
        if (warningTxt.ToString().Length != 0)
        {
             using (StreamWriter outfile = new StreamWriter(outWarning))
             {
                    outfile.Write(warningTxt.ToString());
             }
       }

        // Print log in a file
        if (log.ToString().Length != 0)
        {
             using (StreamWriter outfile = new StreamWriter(outLog))
             {
                    outfile.Write(log.ToString());
             }
         }
      }
      //*************************************
      // Procedure configure Events
      //*************************************
      public void configureSASLanguageEvents(SAS.Workspace sasWorkspace)
      {
        ls = mySAS.LanguageService;

        ls.DatastepStart += new
        SAS.CILanguageEvents_DatastepStartEventHandler(ls_DatastepStart);

        ls.DatastepComplete += new
        SAS.CILanguageEvents_DatastepCompleteEventHandler(ls_DatastepComplete);

        ls.ProcStart += new SAS.CILanguageEvents_ProcStartEventHandler(ls_ProcStart);

        ls.ProcComplete += new
        SAS.CILanguageEvents_ProcCompleteEventHandler(ls_ProcComplete);

        ls.StepError += new SAS.CILanguageEvents_StepErrorEventHandler(ls_StepError);

        ls.SubmitComplete += new
        SAS.CILanguageEvents_SubmitCompleteEventHandler(ls_SubmitComplete);
      }
      //*************************************
      // Procedure ls_StepError
      //*************************************
      private void ls_StepError()
      {
        ls.Reset();
        Debug.Print("Error in SAS code !!!! " +
        Environment.NewLine);
      }
```

```
        //************************************
        // Procedure ls_SubmitComplete
        //************************************
        private void ls_SubmitComplete(int Sasrc)
        {
          Debug.Print("Submit Completed " +
          String.Format("{0:0.00}", Sasrc));
        }
        //************************************
        // Procedure ls_ProcStart
        //************************************
        private void ls_ProcStart(string Procname)
        {
           Debug.Print("Proc Started" + Procname);
        }
        //************************************
        // Procedure ls_ProcComplete
        //************************************
        private void ls_ProcComplete(string Procname)
        {
           Debug.Print("Proc Completed" + Procname);
        }
        //************************************
        // Procedure ls_DatastepStart
        //************************************
        private void ls_DatastepStart()
        {
           Debug.Print("DatastepStart");
        }
        //************************************
        // Procedure ls_DatastepComplete
        //************************************
        private void ls_DatastepComplete()
        {
          Debug.Print("DatastepComplete");
        }
    }
```