PhUSE 2011

Paper AD10

A Modern Software Design Principle Applied To SAS Macro Programming: The Inversion Of Control Concept

Patrick René Warnat, HMS Analytical Software GmbH, Heidelberg, Germany

ABSTRACT

The inversion of control (IoC) architectural concept is a programming principle commonly used in object oriented programming. The basic concept is to implement high-level (in relation to the level of abstraction) source code more reusable by making it independent from low-level, problem-specific code. Program flow and thus control is inverted by using a callback approach which means that code on a lower abstraction layer calls code defined at a higher abstraction layer. This is an inversion of the program flow direction used in traditional procedural programming. The big advantage of IoC is that you get more reusable code, that you decouple your code and thus have less risk of side-effects when you change your code. The usage of the IoC principle is not limited to object oriented programming. This article points out how to apply IoC to SAS macro programming with an actual example and how to benefit from its advantages.

INTRODUCTION

The term inversion of control describes a programming principle commonly used in object oriented programming. Many software frameworks are built upon this concept: A function of an application is registered at a standard library and is called by the library at a later time. Instead of only using code of the standard library, the application program hands over control of the execution of subprograms to the framework. This is also called the Hollywood principle: "don't call us, we call you" (following what actors participating at a film casting are supposedly being told by the movie production companies at the end of the casting). In object oriented programming, an example of how to implement such an inversion of control are Listeners (accordant with the Observer pattern).

The underlying general principle is to implement high-level (in relation to the level of abstraction) source code more reusable by making it independent from low-level, problem-specific code. Program flow and/or dependency relations and thus control is inverted by using a callback approach which means that code on a lower abstraction layer calls code defined at a higher abstraction layer. This is an inversion of the program flow direction used in traditional procedural programming. The big advantage of IoC is that you get more reusable code, that you decouple your code and thus have less risk of side-effects when you change your code.

Let us look at an example, let us assume we have to create a program that solves the following task:
Given a directory path, search the directory and all subdirectories for data files of a certain type, and create a new report file for each file found. Analyzing these requirements, (at least) two subtasks are easily identified. First: traverse the directory tree and find files, second: check whether the file is a data file, read in the data of the data file and generate a report file. You could easily imagine to implement a subprogram for the task to read in a single data file and to generate a report out of it. Then you write a main program that traverses the directory tree, and for every file it finds, it calls the subprogram. That will work very well. But what if you need to have a program that does something else for every single file in a subdirectory structure, for example convert them or simply delete them? Well, of course you could copy the program you already implemented, give it another name and simply call another subprogram instead of the report generating subprogram. That will work, but once you want to change something in your directory traversing code, you already have to change code at two places, and that's not optimal. A better solution is to decouple the code for traversing a directory structure from specific tasks, in order to be able to maintain only a single implementation of it. This can be done by implementing the traversing program not containing a call of a specific subprogram, but only to a general interface of subprograms appropriate to the task to perform an action with a single file. Thus, the dependency relation is inverted, as the traversing program is not dependent on a specific subprogram, and can be used with different types of subprograms that share the same interface.

The IoC principle is not limited to object oriented programming, other programming languages also support it through mechanisms to define callback functions, e.g. the C programming language offers so called function pointers. It is also possible to apply the concept to SAS macro programming, which is demonstrated in the next paragraph.

INVERSION OF CONTROL APPLIED TO SAS MACRO PROGRAMMING

Let us assume you have two SAS macros with a simple relation, one (sub-) macro is called by the other (caller-) macro. Let us further assume the caller-macro implements a sequence of actions that should be applied at several places in your application and in combination with several subtasks. Here, the IoC principle can be applied: implement the caller-macro independently of a specific sub-macro call. So, how can we implement this in SAS? Let us look at a first example. The macro callTwice takes a String as an parameter and calls the sub-macro simplePut two times:

```
%MACRO callTwice(aString);
   %DO i=1 %TO 2;
      %simplePut(&aString.);
   %END;
%MEND callTwice;

%MACRO simplePut(stringToOutput);
   %PUT(&stringToOutput.);
%MEND simplePut;
```

So, what if you see that you need additional variants of the way a string is being output twice, e.g. an “upcase” variant, a “reverse” variant, and so on? You could work with a selection parameter and with blocks of IF-statements in your code, but then, you always have change the macro code of callTwice every time you need a new variant of the output macro. In addition, with many variants, the source code of callTwice would get unnecessarily large. Thus, a better way for the implementation would be:

```
%MACRO callTwice(aString, outputVariant);
   %DO i=1 %TO 2;
      %&outputVariant(&aString.);
   %END;
%MEND callTwice;

%MACRO simplePut(stringToOutput);
   %PUT(&stringToOutput.);
%MEND simplePut;
%MACRO upcasePut(stringToOutput);
   %PUT(%UPCASE(&stringToOutput.));
%MEND upcasePut;
```

Examples for the call of the macro are:

```
%callTwice(testString, simplePut)
%callTwice(testString, upcasePut)
```

This first simple example of the application of the IoC principle to SAS programming assumes that the caller-macro only needs to call sub-macros with a single parameter, thus it is dependent to a class of submacros with a specific interface. You even could implement this more flexible by using a flexible number of parameters for the submacros:

```
%MACRO callTwice2(mNameToCall, mParameters);
   %DO i=1 %TO 2;
      %&mNameToCall(p0=a, %UNQUOTE(&mParameters));
   %END;
%MEND callTwice2;

%MACRO m1(p0, p1);
   %PUT outputM1: &p0 &p1;
%MEND m1;

%MACRO m2(p0, p1, p2);
   %PUT outputM2: &p0 &p1 &p2;
%MEND m2;

%callTwice2(mNameToCall=m1, mParameters=%STR(p1=first Call))
%callTwice2(mNameToCall=m2, mParameters=%STR(p1=second, p2=Call))
```

Instead of quoting mechanisms you could also work with string lists that you explicitly process with %SCAN, or, if your macros are not restricted to only use macro code, you could work with parameter tables consisting of name-value pairs.

After having demonstrated the way to apply the IoC principle to SAS programming, let us look at a more realistic example than the examples with the callTwice macros. Assume we want to traverse a directory tree and perform some action for all SAS files found. So first lets look at the actions we need electively to perform on every single SAS file in a directory tree. Let us assume we have two distinct tasks: to convert the SAS file to a CSV file, to generate a

short pdf report showing its contents. So, our Implementation (for SAS running on Microsoft Windows) could look like this:

```
%MACRO convertSasToCsv (pathToSasFile=, libnameOfSasFile=, sasFileName=);
   PROC EXPORT data=&libnameOfSasFile..&sasFileName.
    outfile="&pathToSasFile.\&sasFileName..csv"
    dbms=csv
    replace;
   RUN;
%MEND convertSasToCsv;
%MACRO getSimplePdfReportOfSasFile (pathToSasFile=, libnameOfSasFile=,
sasFileName=);
   ODS PDF FILE = "&pathToSasFile.\&sasFileName..pdf" NOTOC;
      PROC PRINT data=&libnameOfSasFile..&sasFileName.;
      RUN;
   ODS PDF CLOSE;
%MEND getSimplePdfReportOfSasFile;
```

For simplicity, both makros assume that a SAS libname has been set for the directory with path “&pathToSasFile” containing a SAS file with name (without file extension) “&sasFileName”.

An example implementation of the macro traversing a directory tree for all SAS files could look like this:

```
%MACRO traverseDirForSasFilesAndDoTask(dir=, taskMacro=);
   %local fileref rc did dnum dmem memname len dsn didc aSasFileName;
   %let rc=%sysfunc(filename(fileref,&dir));
   %let did=%sysfunc(dopen(&fileref));
   %if &did=0 %then %do;
      %put ERROR: Directory does not exist;
      %put ERROR- The macro will terminate.;
      %return;
   %end;
   %let dnum=%sysfunc(dnum(&did));
   %do dmem=1 %to &dnum;
      %let memname=%sysfunc(dread(&did,&dmem));
      %if %upcase(%scan(&memname,-1,.)) = SAS7BDAT %then
         %do;
            %let aSasFileName=%scan(&memname,1,.);
            %PUT &aSasFileName;
            libname cDir "&dir";
            %&taskMacro.(
                        pathToSasFile=&dir,
                        libnameOfSasFile=cDir,
                        sasFileName=&aSasFileName)
            libname cDir clear;
         %end;
      %else %if %scan(&memname,2,.)=  %then %do;
         %traverseDirForSasFilesAndDoTask(
                        dir=&dir\&memname,
                        taskMacro=&taskMacro)
      %end;
   %end;
   %let didc=%sysfunc(dclose(&did));
   %let rc=%sysfunc(filename(fileref));
%MEND traverseDirForSasFilesAndDoTask;
```

Example calls:

```
%traverseDirForSasFilesAndDoTask(dir=C:\Temp\TestIoC, taskMacro=convertSasToCsv)
%traverseDirForSasFilesAndDoTask(dir=C:\Temp\TestIoC,
taskMacro=getSimplePdfReportOfSasFile)
```

Using this implementation, you could add many new macros implementing actions on a single SAS file, that can readily used in the traverse macro, without having to change the traverse macro implementation (see Figure 1).

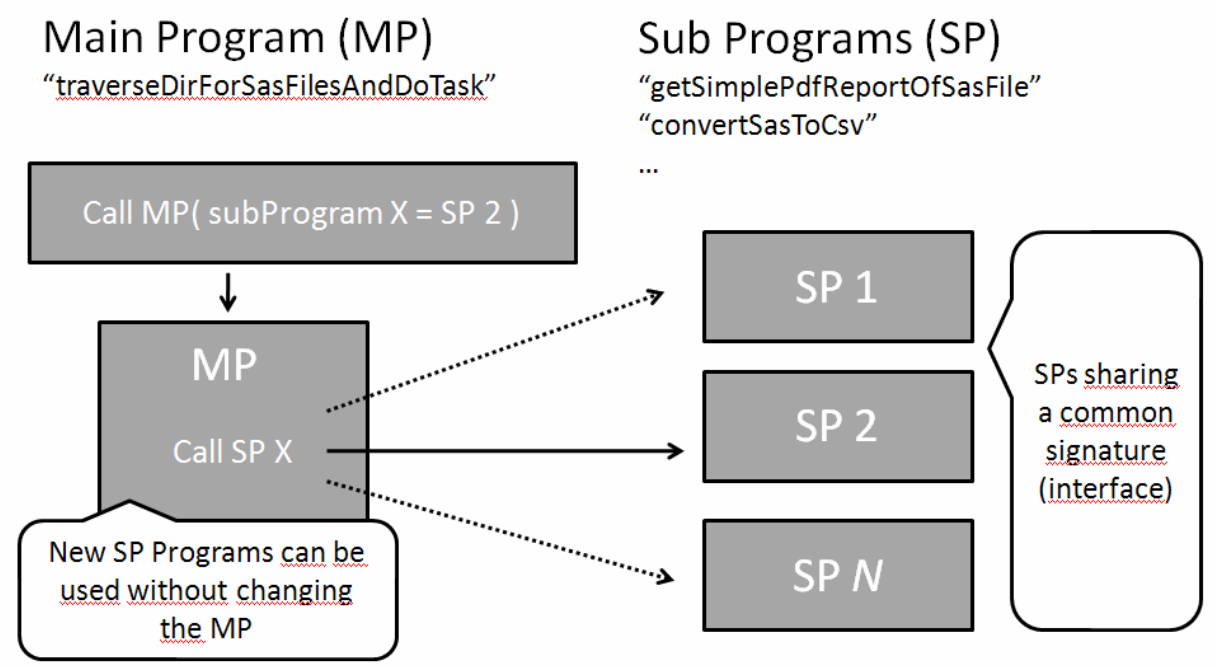

Figure 1: Inversion of Control applied to the example task of a directory search as described in the text

CONCLUSION

The inversion of control principle can be easily applied to SAS programming, resulting in the benefit of more reusable code that is decoupled from specific sub-macros. Your macros only rely on a certain interface describing a group (or class) of sub-macros. Thus, you have less risk of side-effects when you change your code. In addition, caller-macros that are calling sub-macros do not have to be changed if you need to call a new variant of the sub-macro, saving work and potential sources of coding errors.

RECOMMENDED READING

1. Wikipedia page on Inversion of control: http://en.wikipedia.org/wiki/Inversion_of_control
2. Article about the The Dependency Inversion principle by Robert C. Martin : http://www.objectmentor.com/resources/articles/dip.pdf

CONTACT INFORMATION

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

Dr. Patrick R Warnat
HMS Analytical Software GmbH
Rohrbacher Str. 26
69115 Heidelberg
Germany
Fax: +49 6221 60 51 - 0