SAS® GLOBAL FORUM 2021

Paper 1079-2021 ◄ preprint, version 2021.02.05 ►

My First SAS® Package - a How To

Bartosz Jabłoński Warsaw University of Technology / Citibank Europe PLC Poland

ABSTRACT

A **SAS® package**[1] is an automatically generated, single, stand alone `zip` file containing organised and ordered code structures, created by the developer and extended with additional automatically generated "driving" files (i.e. description, metadata, load, unload, and help files).

The purpose of a package is to be a simple, and easy to access, code sharing medium, which allows: on the one hand, to separate the code complex dependencies created by the developer from the user experience with the final product and, on the other hand, reduce developers and users unnecessary frustration related to a remote deployment process.

The **SAS Packages Framework** is a "pack" of macros, which allows to *use* and to *develop* SAS packages.

To create a package the developer executes a few simple steps which, in general, are:

- prepare the code (package content files) and a description file,
- "fit" them into a structured form (see next sections for details),
- download the `SPFinit.sas` (the SAS Packages Framework) file,
- and execute the `%generatePackage()` macro.

In this article we are presenting a step-by-step tutorial which allows to develop a SAS package.

INTRODUCTION and CONTEXT

In the world of programmers, software developers, or data analysts the concept of a package, as a practical and natural medium to share you code with other Users, is well known and common. To give an evidence of this statement let us consider a few very popular examples: Python, TEX, and R software. As an endorsement of this fact it is enough to visit the following web pages:

```
https://pypi.org/
https://www.ctan.org/
https://cran.r-project.org/
```

to see the number of available packages. As of January 1st, 2021 there were over 5,962 packages available on the Comprehensive TEX Archive Network, and over 16,851 packages on the Comprehensive R Archive Network! More than a dozen of TEX packages were used while writing this article.

With this said, the fundamental question of a new or a seasoned SAS User should be: "Why there is no such thing like packages in SAS?" To be clear, there are "packages" in the SAS ecosystem (see the footnote) but they do not offer such functionality as the one mention above. For example the SAS/IML offers (limited) functionality similar to the concept of a package in programming languages mentioned above but such functionality is not available in the Base SAS.

The general overwiew of the SAS Packages Framework was introduced in [**01**]. The main goal of this article is to describe, explain, and discuss the process of creating SAS packages and to allow to learn how to easily share code with others.

[1] The idea presented in this article should not be confused with other occurrences of "package" concept which could be found in the SAS ecosystem, e.g. Proc DS2 packages, SAS/IML packages, SAS ODS packages, SAS Integration Technologies Publishing Framework packages, or a `*.egp` file.

Note. Though using or building a package is very straightforward process the creation of a package content (i.e. macros, functions, formats, etc.) requires some experience. That is why the intended readers for the following text are *at least* intermediate SAS programmers. The good knowledge of the Base SAS and practice in macro programming will be an advantage. Such a background knowledge can be found for example in [**02**].

THE FRAMEWORK

The SAS Packages Framework is a set of macros which allow to interact with SAS packages. Several of these macros are used to work with a package when it is already created. But from the package generation point of view the `%generatePackage()` macro is the one which is crucial. Though all of the remaining macros are also used during the process, without the `%generatePackage()` nothing can be done. Below is the description how to enable the framework in the SAS session.

Location. The SAS Packages Framework macros are all stored in the `SPFinit.sas` file, which is available at the following GitHub repository:

https://github.com/yabwon/SAS_PACKAGES

The GitHub repository is the *main* repository of the framework. An alternative/emergency location which also contains the file is:

`https://pages.mini.pw.edu.pl/~jablonskib/SASpublic/SAS_PACKAGES/`

(previously: `https://mini.pw.edu.pl/~bjablons/SASpublic/SAS_PACKAGES/`)

Instalation. The file can be installed (i.e. downloaded) either manually or programmatically. But the very first step (step zero) is to create a folder where the framework will be installed. Let's assume that from now on for the Windows OS it will be the `C:\saspackages` folder and for the Linux/UNIX OS it will be the `~/saspackages` directory.

To download the file by hand the following steps have to be executed:

- go to repository mentioned above,
- open the `SPF` subdirectory,
- select the `SPFinit.sas` file,
- click on the "Raw" button on the left side,
- click right-mouse-button on the opened text and select `save-as` from the menu
- navigate to the directory created in the step zero
- save the `SPFinit.sas` file

To enable the framework in the SAS session the following snippet has to be run:

```
filename packages "<directory/from/step/zero/>";
%include packages(SPFinit.sas);
```

Side Note: Another way to get the framework file would be to just pull the GitHub repository, but with this way all additional files from the repository are also downloaded.

To get and enable the framework file "by coding", from within a SAS session, an execution of the `filename` statement with the URL device has to be possible in the session. If this is the case the following code snippet should be run:

```
filename packages "<directory/from/step/zero/>";
filename SPFinit url
  "https://raw.githubusercontent.com/yabwon/SAS_PACKAGES/main/SPF/SPFinit.sas";
%include SPFinit;
%installPackage(SPFinit)
```

to enable and install the framework. In all subsequent SAS sessions only this code:

```
filename packages "<directory/from/step/zero/>";
%include packages(SPFinit.sas);
```

should be run to enable the framework.

Side Note: Adding those two lines of code to the `autoexec.sas` will enable the framework for every new SAS session.

THE PACKAGE CONTENT

When the SAS Packages Framework is installed and ready the next step is to prepare the package files. From now on, without loss of generality and for the sake of simplicity, let's assume that the package will be named `myPackage`.

The Example. The process of creating `myPackage` will be presented with a little help of the following example code. The code is one file containing the code of:

- one library `myLib` created in the (same named) subderectory of the `WORK` directory [lines 1:13],
- two FCMP functions: `F1` and `F2` [lines 15:25],
- one numeric format `fmtNum` and informat `infNum` [lines 27:42],
- one exemplary small dataset `myLib.smallDataset` [lines 44:50],
- one exemplary bigger dataset `myLib.biggerDataset` [lines 52:59], and
- two macros: `mcrOne` and `mcrTwo` [lines 61:78].

The example code is just a "shell" but it is "complex" enough to perform a fully valid presentation.

the example

```
1  /*one library "myLib" created in the (same named) subderectory of the "WORK" directory.*/
2  data _null_;
3    length rc0 $ 32767 rc1 rc2 8;
4    lib = "myLib";
5    rc0 = DCREATE(lib, "%sysfunc(pathname(work))/");
6    put rc0 = ;
7    rc1 = LIBNAME(lib, "%sysfunc(pathname(work))/" !! lib, "BASE");
8    rc2 = LIBREF (lib);
9    if rc2 NE 0 then
10     rc1 = LIBNAME(lib, "%sysfunc(pathname(work))", "BASE");
11 run;
12
13 libname myLib LIST;
14
15 /*two FCMP functions: "F1" and "F2", */
16 proc FCMP outlib = work.f.p;
17   function F1(n);
18     return (n+1);
19   endsub;
20
21   function F2(n);
22     return (n+2);
23   endsub;
24 run;
25 options cmplib = work.f;
26
27 /*one numeric format "fmtNum" and informat "infNum", */
28 proc FORMAT;
29   value fmtNum
30   low -< 0         = "negative"
31          0         = "zero"
32          0 <- high = "positive"
33          other     = "missing"
34   ;
35   invalue infNum
36     "negative" = -1
37     "zero"     =  0
38     "positive" =  1
39     "missing"  =  .
40     other      = 42
41   ;
42 run;
43
44 /*one exemplary small dataset "myLib.smallDataset", and*/
```

```
data myLib.smallDataset;
  do n = ., -1, 0, 1;
    m = put(n, fmtNum.);
    output;
  end;
run;

/*one exemplary bigger dataset "myLib.biggerDataset".*/
data myLib.biggerDataset;
  do i = ., -1e6 to 1e6;
    j = put(i, fmtNum.);
    k = ranuni(17);
    output;
  end;
run;

/*two macros: "mcrOne" and "mcrTwo" */
%macro mcrOne();
  %put **Hi! This is macro &sysmacroname.**;
  data _null_;
    set myLib.smallDataset;
    p = f1(n);
    p + f2(n);
    put (n p) (= fmtNum.);
  run;
%mend mcrOne;

%macro mcrTwo(m=mcrOne);
  %put **This is macro &sysmacroname.**;
  %put **and I am calling the &m.**;
  %&m.()

  %put The answer is: %sysfunc(inputn("I don't know...", infNum.));
%mend mcrTwo;

/* %mcrTwo() */
```

When the code facilitating the package is ready the next step is to split it into separate files, according to so called *types*, arrange them in a package structure and provide the help information. The upcoming section describes the process.

THE STRUCTURE

The first step of setting up a structure of a package is to create a package folder. It could be any folder and it does not have to be "related" to the framework folder. But, from the practice, since the folder for the framework is already available it can be used. The package will be named `myPackage` so let's assume that for the Windows OS the package folder will be the `C:\saspackages\myPackage` and for the Linux/UNIX OS it will be the `~/saspackages/myPackage` directory.

Types and Files. Inside the package folder the developer creates subfolders for package content. All of them have to be *in lower case letters* similarly as all files inside.

The code of the package has to be divided in separate files according to the following rules:

- *One-file-one-object*, e.g. macro `%abc()` definition has to be contained in a single file without any definitions of other objects. The only exception are formats/informats, in this case one file has to contain all definitions of formats/informats *sharing the same name*, e.g. numeric format `abc.`, character format `$abc.`, numeric informat `abc.`, and character informat `$abc.` all have to be kept in one file.
- *An object name is a file name*, e.g. a definition of a macro named `%abc()` has to be contained in a file named `abc.sas`.

According to these rules the Example code has to be divided into nine parts, one for each object: `mylib.sas`, `f1.sas`, `f2.sas`, `fmtnum.sas`, `infnum.sas`, `mylib.smalldataset.sas`, `mylib.biggerdataset.sas`, `mcrone.sas`, and `mcrtwo.sas` .

The files have to be placed in subfolders indicating their *types*, and the subfolders have to be created according to the specific rules:

- subfolder name contains only lower case letters, digits, and an underscore ("_")
- subfolder name is composed of *two parts* separated by an underscore, i.e.
  - *the first* part is a series of digits (with leading zeros, e.g. `001`, `002`, ..., `123`, `124`, ...) to keep an execution sequence in case the code must be ordered to run properly. For example if a code creating a library has to be execute before a code creating a dataset then subfolders could have the following form: `001_libname` and `002_data`.
  - *the second* part, called folder *type*, indicates subfolder's content. The *type* has to be one of the following:
    - `libname` (for libraries assignments),
    - `macro` (for macros),
    - `function` (for `proc fcmp` function),
    - `functions` (mind the "s" at the end, for `proc fcmp` functions),
    - `format` (for format and informat),
    - `formats` (mind the "s" at the end, for formats and informats),
    - `imlmodule` (for IML modules),
    - `proto` (for `proc proto` external C functions),
    - `data` (for the code generating datasets),
    - `lazydata` (for the code generating datasets loaded on demand, so called "lazy datasets"),
    - `exec` (for so called "free code"),
    - `clean` (for the code cleaning up the session after `exec`s) or
    - `test` (for developer code with package tests, not included into the package).

According to these rules the Example code will be structured as presented in Figure 1. The *Notes* below provides additional justification for such structure.

Note 1. Types `functions` and `formats` behaves differently than `function` and `format`. The difference is related to the way how the code has to be structured inside the file (see [**01**] for the details). Work with the "S" is more convenient, and no additional options are needed, that is why types `functions` and `formats` will be used.

Note 2. The code of the dataset `smallDataset` refers to the `myLib` library and uses the `fmtNum.` format that is why their code has to be generated first and, as a consequence, folders for their code will have smaller ordering numbers to ensure proper code execution. So when the package will be loaded into the User's SAS session as the first the code from the `001_libname` folder will be executed, as the second the code from the `002_functions` and so on.

Note 3. The code of the dataset `biggerDataset` generates a dataset of approximately 50 megabytes whence it will be included in the package as a *lazydata* type. This means that it will not be generated automatically during the package loading but it will be generated "on demand".

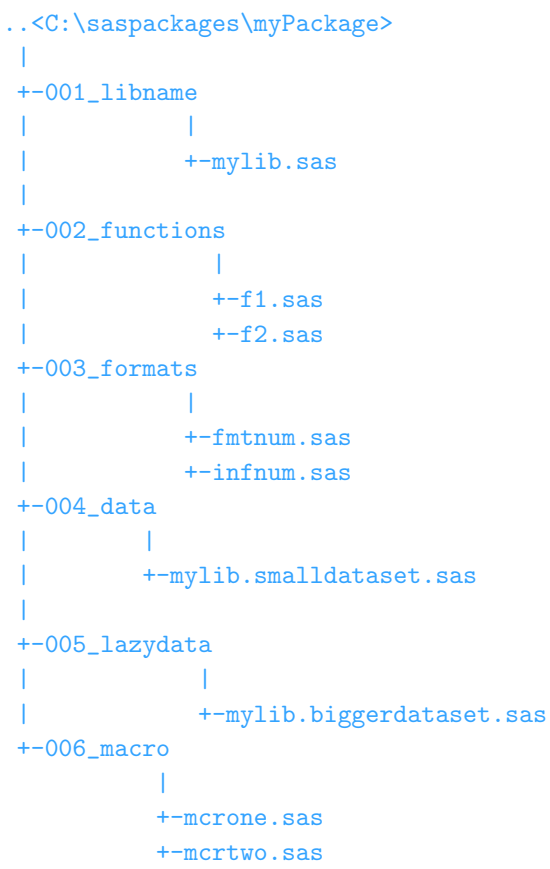

Figure 1: Structure of the `myPackage` package subfolders.

The Code. When the files and folders structure is set up the subsequent step is to prepare the code. For some types the code can be copy-pasted "as is", for other some small "cosmetic" changes are required. Everything is very straightforward.

The code of the `libname` type can be copy-pasted so the `mylib.sas` file contains the following content:

mylib.sas

```
data _null_;
  length rc0 $ 32767 rc1 rc2 8;
  lib = "myLib";
  rc0 = DCREATE(lib, "%sysfunc(pathname(work))/");
  put rc0 = ;
  rc1 = LIBNAME(lib, "%sysfunc(pathname(work))/" !! lib, "BASE");
  rc2 = LIBREF (lib);
  if rc2 NE 0 then
    rc1 = LIBNAME(lib, "%sysfunc(pathname(work))", "BASE");
run;

libname myLib LIST;
```

Since the `functions` type is used for the FCMP functions only the plain code of the function has to be copy-pasted, without the `Proc FCMP` header and the `run` footer. The header and the footer will be generated automatically. Whence the `f1.sas` file contains the following content:

f1.sas

```
  function F1(n);
    return (n+1);
  endsub;
```

and the `f2.sas` file contains the following content:

f2.sas

```
function F2(n);
  return (n+2);
endsub;
```

Similarly, to the `functions` type, works the `formats` type. Only the plain code of the format/informat has to be copy-pasted into the code file. Also without the `Proc FORMAT` header and the `run` footer which will be generated automatically. The `fmtnum.sas` file contains the following content:

fmtnum.sas

```
value fmtNum
low -< 0         = "negative"
       0         = "zero"
       0 <- high = "positive"
       other     = "missing"
;
```

and the `infnum.sas` file contains the following content:

infnum.sas

```
invalue infNum
  "negative" = -1
  "zero"     =  0
  "positive" =  1
  "missing"  =  .
  other      = 42
;
```

The code for both datasets (including lazy data) and macros can be copy-pasted "as is" so the files `mylib.smalldataset.sas`, `mylib.biggerdataset.sas`, `mcrone.sas`, and `mcrtwo.sas` will contain, respectively, the following content:

mylib.smalldataset.sas

```
data myLib.smallDataset;
  do n = ., -1, 0, 1;
    m = put(n, fmtNum.);
    output;
  end;
run;
```

mylib.biggerdataset.sas

```
data myLib.biggerDataset;
  do i = ., -1e6 to 1e6;
    j = put(i, fmtNum.);
    k = ranuni(17);
    output;
  end;
run;
```

mcrone.sas

```
%macro mcrOne();
  %put **Hi! This is macro &sysmacroname.**;
  data _null_;
    set myLib.smallDataset;
    p = f1(n);
    p + f2(n);
    put (n p) (= fmtNum.);
  run;
%mend mcrOne;
```

and

mcrtwo.sas

```
%macro mcrTwo(m=mcrOne);
  %put **This is macro &sysmacroname.**;
  %put **and I am calling the &m.**;
  %&m.()
```

```
5
6   %put The answer is: %sysfunc(inputn("I don't know...", infNum.));
7 %mend mcrTwo;
```

Help. The final step of preparing the package content is providing a valuable-help-information. Every part of the code or a comment in the package file surrounded by the "/*** HELP START ***/" and the "/*** HELP END ***/" tags will be treated as a part of the help information extracted for the User by the %helpPackage() macro. These tags have to be placed in separate line and left adjusted. If the following text:

"The '%mcrTwo()' macro is the main macro of the package.
It has one key-value parameter 'm' with default value 'mcrOne'.
Example 1. Basic use-case: %mcrTwo(m=mcrOne)"

is to be used as a comment in the mcrtwo.sas file it could be placed the following way:

mcrtwo.sas with the help information

```
1  /*** HELP START ***/
2  /*
3  ## General Info: ##
4
5  The '%mcrTwo()' macro is the main macro of the package.
6
7  It has one key-value parameter 'm' with default value 'mcrOne'.
8  */
9  /*** HELP END ***/
10
11 %macro mcrTwo(m=mcrOne);
12   %put **This is macro &sysmacroname.**;
13   %put **and I am calling the &m.**;
14   %&m.()
15
16   %put The answer is: %sysfunc(inputn("I don't know...", infNum.));
17 %mend mcrTwo;
18
19 /*** HELP START ***/
20 /*
21 ## Examples: ##
22
23 Example 1. Basic use-case:
24 ~~~~~~~~~~~~~~~~~~~~~~~~~
25
26 %mcrTwo(m=mcrOne)
27
28 ~~~~~~~~~~~~~~~~~~~~~~~~~
29 */
30 /*** HELP END ***/
```

Attentive reader will notice at once that the comment was provided in a markdown format (see [**04**]). Though it is the author's favourite way of writing help info (and also very practical, since it is "readable" both as a plain text and as interpreted one) it is not the obligatory way of doing so. In fact, the form of the help text is one-hundred percent the Developer's initiative.

Note. If the help information is not provided, i.e. there are no help tags in the file, the package generation process will rise a warning in the SAS log indicating this fact. This means that help tags should be provided also to other files in the package not only to the mcrtwo.sas file.

THE DESCRIPTION FILE

When the content is ready, and the files and folders are structured there is only one more thing to do. It is the creation of the desctiption.sas file and (optional) the licence.sas file. Both files have to be

placed in the package folder alongside with the subfolders with the code, so for the Example the folder structure would be extended like in Figure 2.

```
..<C:\saspackages\myPackage>
 |
 +-001_libname
 |           |
 [...]       +-mylib.sas
 |
 +-006_macro
 |          |
 |         +-mcrone.sas
 |         +-mcrtwo.sas
 |
 +-description.sas
 |
 +-<license.sas>
```

Figure 2: The `description.sas` and (optional) `license.sas` files in the package folder.

The `description.sas` file is *mandatory*, it has simple structure, and contains the package metadata and (short) description (compare [**03**]). An example of the `description.sas` file for a package can be seen in Figure 3. The structure will be explained in the next paragraphs (see [**01**] for the details).

```
/*** HEADER ***/ ①
Type: Package                                                    ❶
Package: ThePackageName                                          ❷
Title: Short description, single sentence.                       ❸
Version: x.y                                                     ❹
Author: Fname1 Lname1 (xxx1@yyy.zz), Fname2 Lname2 (xxx2@yyy.zz) ❺
Maintainer: Fname3 Lname3 (xxx3@yyy.zz)                          ❻
License: XYZ17                                                   ❼
Encoding: UTF8                                                   ❽

Required: "Base SAS Software", "SAS/XYZ", "SAS/ACCESS Interface to ..." ③
ReqPackages: "piPackage (3.14)", "ultimatePackage (42)"                 ④

/*** DESCRIPTION ***/ ②
DESCRIPTION START: ❾
Lorem ipsum dolor sit amet, ThePackageName consec tetur
adipis cingelit. Nullamdapibus lacus a elit congue
elementum.

Suspendisse iaculis ipsum ThePackageName nec ante
luctus volutpat. Donec iaculis laoreet tristique:
~~~~~~~~~~~~~~
  data _null_;
    x = 17;
  run;
~~~~~~~~~~~~~~
DESCRIPTION END: ❿
```

Figure 3: General structure of a SAS package description file.

The meaning of entries inside the `description.sas` file are the following (the dark bullet with values from 1 to 10 marks elements which are *mandatory*):

(①) `/***HEADER***/` - marks the header start (it is optional comment). Each of the following lines is a `key:value` pair and such a pair must be a *single line of text*. The colon (`:`) is a field separator and is restricted in lines of the header.

(❶) is a constant value, always `Type:Package`.

(❷) is the package name. It is required, not null, *up to 24 characters* long, and shares SAS dataset naming restrictions.

(❸) is the short text, i.e. one phrase, describing the package.

(❹) is the package version, has to be a positive number (i.e. > 0). *Note.* The `%loadPackage()` macro contains a dedicated macrovariable `requiredVersion` to test if the provided version of a package is *greater or equal* then the required version - a very important, in line with the "*SAS way*", assumption must be highlighted! The assumption is: *packages are assumed to be backward compatible!*

(❺) and (❻) are comma separated lists of package author(s) and maintainer(s).

(❼) indicates the license under which the package is distributed. The license text itself should be inserted into the `license.sas` file (see further paragraphs for details).

(❽) points the information about SAS session encoding the package files were created in. It is technical parameter and possible values have to satisfy requirements for the `encoding=` option of the `filename` statement.

(②) `/*** DESCRIPTION ***/` - marks the description start (it is optional comment).

(❾) and (❿) The package description is the last required part. It is a free text bounded between the "`DESCRIPTION START:`" and the "`DESCRIPTION END:`" tags. It can be multi-line. It should elaborate about the package and its components (e.g. macros, functions, datasets, etc.). Text outside the tags is ignored.

There are two additional, optional, elements of the header.

(③) `Required` - is a *quoted and comma separated* list of licensed SAS products required for the package to work. Values on the list should be the same as the `proc setinit` output in the SAS log.

(④) `ReqPackages` - is a *quoted and comma separated* list of names and versions (in parentheses) of other SAS packages required for the package to work.

The `description.sas` file for the Example discussed can look like in Figure 4.

```
Type: Package
Package: myPackage
Title: My first SAS package.
Version: 1.0
Author: John Smith (john.smith@mail.com)
Maintainer: Jane Smith (jane.smith@mail.com)
License: MIT
Encoding: UTF8

Required: "Base SAS Software"

DESCRIPTION START:
## The myPackage ##

The ‘myPackage‘ is my first SAS package and
for sure it won’t be the last package!
It was created during SAS Global Forum 2021
virtual event.

It helps me to share my code with other SAS users!
DESCRIPTION END:
```

Figure 4: The description file for `myPackage`.

License. A separate discussion is the `license.sas` file. If the file is not provided by the Developer the `%generatePackage()` macro automatically generates the MIT license file based on generation date and the `Author:` tag in the description file.

If the Developer want to distribute the package under different conditions then the MIT license then the `license.sas` file, containing required license text, has to be provided.

In the Example that is discussed the MIT license will be generated automatically.

PACKAGE GENERATION

At this point everything is ready, the SAS Packages Framework is downloaded, the description file is prepared, and the package content is structured and "sprinkled" with the help information. There is nothing left to do but to generate the package!

The `%generatePackage()` macro. To generate the package the `%generatePackage()` macro has to be executed. According to the Example's setup the code to be run has to be the following:

code to generate the package under Windows

```
filename packages "C:\saspackages";
%include packages(SPFinit.sas);

%generatePackage(C:\saspackages\myPackage)
```

or the following:

code to generate the package under Linux/UNIX

```
filename packages "~/saspackages";
%include packages(SPFinit.sas);

%generatePackage(~/saspackages/myPackage)
```

Summary. When the above code is executed the `mypackage.zip` file is created inside the package folder. When the process is finished the SAS Packages Framework prints out a summary, both in the log and as a set of tables in the output window.

The best practice is to start with investigating the log. In the log there are information about results of creating the package content. This section indicates if a given file was copied into the `zip` file and if it was successful. It also prints a warning message if the help tags were not found in some of files or an error message if the tags were mismatch (e.g. overlapping). Package SHA256 checksum is printed in the log too. The final part of the log provides an information about the testing process, if one was executed.

In the output window a general information about the package is printed out. It contains:

- package folder location,
- login of a user who was running the session,
- time-stamp of the execution,
- version of SAS used for the process
- information about both package and session encoding

The next part lists packages components and metadata information such as: name, version, license type, MD5 hash of the package low cased name (used as a fileref), and a list of required SAS components and SAS packages (if they were provided). Package SHA256 checksum is printed out again.

The last part is a table with the summary of tests executed. It contains, in subsequent observations, a name of the test and values of `systask` return code, `sysrc` return code, the number of errors and the number of warnings for that particular test. Also a path to the location of logs and listings from the tests is printed out.

Obs	testName	systask	sysrc	error	warning
1	loading	0	0	0	0
2	my test	0	0	0	0

Table 1: Expected result for properly executed test for loading package and a Developer's test.

TESTS

Process of testing is composed of two parts. The first one is automatic and mandatory, the second one is up to developer's will. But they both are dependent on the `XCMD` option and are executed only if it is enabled. This is because tests are executed in separate SAS sessions run by the `systask` statement. As a default the `!SASROOT` directory is used to search for SAS binary and config file but it can be altered by `sasexe=` and `sascfgfile=` parameters.

Test for Loading. Assuming the `XCMD` is on, the first test is generated automatically by the framework and checks some fundamental functionalities like: package loading, printing package help info, printing package code, checking if macros (if there are any) were compiled, and eventually checks package unloading. The test result is printed out in the summary table as the number one under the name "loading". If everything went well and the test passed all result values for it should be zeros (see Table 1). If there were any errors or warning in the loading test log they will be also printed out in the main SAS session log with lines numbers.

Developer's tests. If the Developer want to perform additional tests the `test` type subfolder has to be created in the package folder. Each file with code inside this folder will be executed in a separate SAS session the same way as the loading test. The file with test code does not have to contain code loading the package, it is automatically added in dedicated `autoexec.sas` file. For example trivial test executing only one line of code:

my test.sas

```
1 %mcrTwo(m=mcrOne)
```

will be executed in a separate SAS session and, similarly to the loading test, if everything went well and the test passed all result values for it should be zeros (see Table 1).

No XCMD. Testing can be suspended by setting the `testPackage=` parameter to "N". But when the `XCMD` option is disabled the testing cannot be executed automatically, in such case the framework prints out text of code which should be executed by hand by the Developer to verify loading process.

CONCLUSION

This article presented the process of generating SAS package with use of the SAS Packages Framework. Fundamental concepts of types, code structuring, providing help information and testing were discussed. Everything was executed with a little help of a very simple Example. Author hopes that knowledge presented here will help and engage Developers to contribute and create more practical and useful SAS packages in the (near) future!

REFERENCES

[01] Bartosz Jabłoński, "SAS Packages: The Way to Share (a How To)",
SAS Global Forum 2020 Proceedings, 4725-2020
https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2020/4725-2020.pdf
extended version available at: https://github.com/yabwon/SAS_PACKAGES/blob/main/SPF/Documentation

[02] Art Carpenter, "Carpenter's Guide to Innovative SAS Techniques", SAS Press, 2012

[03] Hadley Wickham, "R Packages: Organize, Test, Document, and Share Your Code",
O'Reilly Media, 2015, http://r-pkgs.had.co.nz/description.html

[04] Markdown, https://en.wikipedia.org/wiki/Markdown as of January 2021

ACKNOWLEDGMENTS

Author would like to acknowledge **Filip Kulon** and **Krzysztof Socki** for their contribution and effort to make this paper looks and feel as it should!

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at one of the following e-mail address:

yabwon✉gmail.com

bartosz1.jablonski✉citi.com

or via the following LinkedIn profile: www.linkedin.com/in/yabwon

Appendix A - Other Types

A few of SAS package file types were not discussed in the Example presented in the main part of the article. They will be discussed in this Appendix section.

A definition of an `imlmodule` has to be plain code of the `Proc IML` module, for example:

imlmodule

```
start aModule(x);
  return (x#x + 42);
finish aModule;
```

All modules defined in the `..._imlmodule` subfolder are compiled with one `Proc IML` execution and are stored in the `work.&packageName.iml` catalog.

A definition of a `proto` external C function has to be plain code of the `PROTO` C function. All functions defined in the `..._proto` subfolder are compiled with one `Proc Proto` execution and are stored in the `work.&packageName.proto` dataset. Function code has to contain the header, and the body of the function between `externc` and `externcend`, for example:

proto

```
int doublePlusOne(int x);

externc doublePlusOne;
  int doublePlusOne(int x)
  {
    return (2*x + 1);
  }
externcend;
```

The `exec` folders are for so-called "free code", i.e. if a package, to be ready and usable, requires some additional code to be run, and the code does not fit provided *types*, this code can be inserted into a file inside the `exec` subfolder.

The `clean` folders are design for cleaning after `exec`s, i.e. if a code from one of `exec` folders creates some object (e.g. a catalog, a macro, or a dataset) the appropriate code inside a `clean` subfolder shoud be developed to remove that created object.