

PhUSE 2009

Paper AD04

Effective Scripting in JMP

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

ABSTRACT

The JMP software for statistical analysis offers a highly interactive and easy-to-handle graphical user interface. It is a less well known fact that it also offers a mature scripting language (JMP Scripting Language, JSL) for automation and implementation of new functionality.

After a short general introduction to JSL, this paper focuses on providing recommendations for effective software development using JSL. Structuring, reuse and deployment of JSL code and writing unit-tests in JSL is covered as well as tips for creation of custom reports.

The usage of JSL scripting offers the opportunity to effectively customize JMP for your needs and your business processes.

INTRODUCTION

JMP® is a software application for statistical analysis which offers an interactive and easy-to-handle user interface. Thus, the software is very well suited for interactive data exploration and analysis. In addition, JMP comprises a scripting facility that enables users to write and execute scripts within JMP using a JMP-specific scripting language (JMP Scripting Language, JSL). With JSL, almost everything that you can achieve by clicking through the graphical user interface is also achievable by writing and executing a JSL script. Moreover, JSL is general enough to be used to implement new functionality. Therefore, JSL both can be used for automation of a sequence of task JMP offers out of the box and for implementation of your own algorithms within JMP.

The details of the JSL language are very well documented in the JMP scripting guide shipped with JMP. The examples therein cover everything that is needed to create small scripts. However, JSL is suitable to implement complex business logic that covers complete business processes. This paper presents suggestions that are especially useful when you plan to implement more than a simple automation script, but instead a larger piece of JSL-Code. The paper focuses on general strategies how to work effectively with JSL, in particular in large projects. The remainder of this paper is structured as follows: At First, a short overview on the JSL scripting language is given. The main part of the paper provides recommendations how to work effectively with JSL. Finally, the paper contains a conclusion section, as well as sections providing information about the versions of the software used, references and contact information.

JSL OVERVIEW

The basic concept behind the JMP scripting language is a mix of a “functional” programming approach (like in e. g. Haskell) intermixed with some “object orientation”. The basic structure of a JSL script is the so called Expression. Most expressions are a function call (e. g. `FUNCTION(PARAMETER1, PARAMETER2)`), even basic operations like Assignment or Addition can be expressed as function calls. In JSL, the expression

```
R = 5 + 6
```

is equivalent to the expression

```
ASSIGN(R, ADD(5, 6))
```

and both expressions can be used in a JSL script.

Thus, a JSL script is a sequence of nested functions, program flow control (e.g. `FOR()`, `IF()`) is also expressed by use of functions. Even the semicolon (;), that is used in many programming languages as a statement terminator is equivalent to the JSL function `GLUE()`. In JSL, a semicolon means that the expressions before and after the semicolon are executed one after another and the value of the expression after the semicolon is returned.

The above mentioned influence of the object orientation-paradigm on JSL is that JMP Environment components (e.g. data tables) are represented as objects to which messages can be send (e.g. to read out the value of the data table cell of a certain row and column). However, no own “classes” of data structures can be defined by the programmer, only general data structures (single value variables, lists, maps and matrices) are available for usage in JSL scripts. For are more detailed introduction to JSL please refer to the JMP scripting guide shipped with JMP, containing a good introduction chapter alongside with the full language documentation of JSL.

PhUSE 2009

RECOMMENDATIONS FOR EFFECTIVE SCRIPTING IN JMP

The recommendations given below are grouped by topics and render a personal list of useful tips resulting out of lessons learned in several projects utilizing JSL scripting.

STRUCTURING AND REUSE

A very effective way to structure JSL source code is to use the possibility to define own JSL functions. With the function

```
FUNCTION( {<ARGUMENTS>}, SCRIPT)
```

a function can be defined that has a certain list of ARGUMENTS, which can be referenced as variables in the source code given in SCRIPT. The defined function returns the last value generated during the execution of SCRIPT.

Example:

```
SQUAREFIGURE = FUNCTION( {AFigure}, AFigure*AFigure );  
SQUAREFIGURE(3); //RESULTS IN A VALUE OF 9
```

When a function is defined, default values can be given for its arguments. Arguments with default values are optional when a function is called, if they are omitted, the default values will be used.

In JSL, user defined variables are always global variables. That means that variables defined within functions are also global variables, which could interfere with variables outside of the context of your function. Thus, I recommend to use the build-in function LOCAL(). With LOCAL() you can define variables that exist only in a certain scope. Variables used as temporary storage in your function definition should always be declared as local variables. Using the function TRY() it is possible to “catch” errors that occur during execution of JSL code that is enclosed by the TRY() function. Thus, it is possible to implement your own error handling in a JSL function, for example to perform “clean-up” actions like closing opened data tables or to generate error messages customized to specific requirements.

As a general template for function definitions I recommend to combine the above mentioned JSL procedures as follows:

```
<FUNCTION NAME> = FUNCTION( {<ARGUMENTS>},  
    LOCAL( {<LOCAL VARIABLES>},  
        TRY(  
            <FUNCTION BODY SCRIPT CODE>  
            , //CATCH  
            <ERROR HANDLING SCRIPT CODE>  
        ) ; //END TRY  
    ) ; //END LOCAL  
); //END FUNCTION
```

However, as the number of lines of code grows in your project, it is a good idea to distribute your function definitions to several files or even to define only one function per file. The INCLUDE() function gives you the opportunity to call JSL code contained in another script file.

JSL does not provide any mechanism to group your functions into modules or packages. Nevertheless, it is a good idea to organize your functions into logical groups. To organize your functions, I recommend to include a “package” name into the name of your functions like “util_includeAllScriptFilesInDirectory”. I recommend to put the package name at the beginning of the function name and to place every function in a separate file which has the same name as the function contained. This way, it is easier to keep track of all the JSL functions of a project.

Every JSL function can return only one value. Whenever you want to return several values (even of different types), you can use a list object. However, instead of using a list, I recommend to use an Associative Array, a data type concept also known as “Map” in other programming languages. An Associative Array is a container for key-value pairs. As a return value of a function, a map could be used as an entity-object-like container. For example, if your function returns values describing one person (Name, Age, Sex), do not use a list, where you have to rely that the name is always at first position. It is better to use a map where you could assign an “attribute” name to every value. ASSOCIATIVE ARRAY({ { "NAME", "JOHN SMITH"}, { "AGE", 44}, { "SEX", "MALE"} });

UNIT-TESTS

In order to implement JSL code that runs reliable, there is no avoiding testing. I recommend to implement your tests as JSL scripts, one “test-script” file for every JSL function to be tested. This not only results in a kind of executable documentation of what you have tested, but furthermore enables you to repeat execution of your tests easily, which is very important in order to check whether later modifications on your functions have unintended side effects. This approach of unit-testing is a key to effective software development and can be used for JSL scripting, too.

For JSL, a unit-test framework named JSL-Unit is available from the JMP website (navigate to www.jmp.com; there to “user community” -> “file exchange”, look for JSL-Unit).

Once downloaded and executed within JMP, it provides a dialog to select files containing test-scripts (Figure 1, left). A test-script contains tests of a function, each of which consists of code that prepares a certain test-setup (e.g. loading of a data table with test data), a call to the function to be tested and one or more assertion-statements (special functions provided by JSL-Unit) that assess whether the test produced the expected results. The unit-test framework executes one or more test scripts and generates a simple summary report with the test results in the JMP log (Figure1, right).

This simple unit test framework enables you to apply the powerful paradigm of test-driven software development to JSL programming.

PhUSE 2009

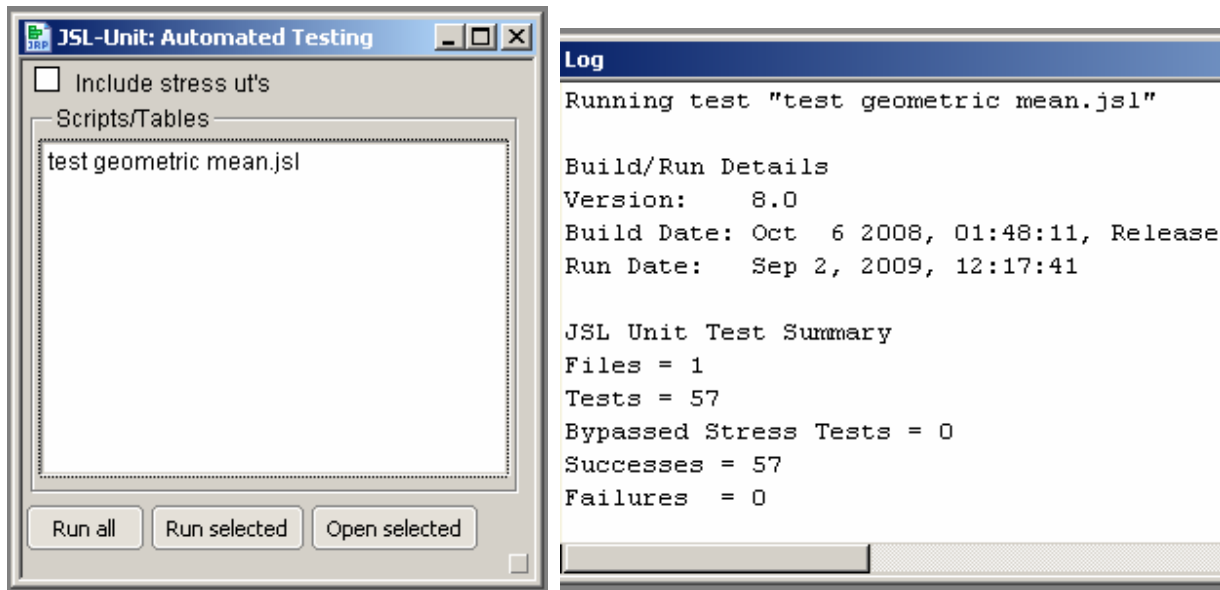


Figure 1:
Left: JSL-Unit dialog window; Right: JSL-Unit summary report in JMP log window

DEPLOYMENT

Having implemented and tested your JSL code, you often want to provide your code to colleagues or customers. In the simplest case, you copy them your JSL script files which they then can open and manually start in their local JMP installation.

You could relieve them of the latter task by adding the characters "!!" (without quotes) at the beginning of a script file. With default settings in JMP, a script file starting with these characters is not opened in JMP as a script window but executed directly. Thus, a script can directly generate a GUI when opened within JMP and no script window is shown to the user. Of course the contents of such a script file can still be read by opening it in an external editor or by changing preferences in JMP. If you want to prevent others from seeing the contents of your scripts, JMP offers the opportunity to encrypt scripts with a password (Menu Edit -> Encrypt Script). Once encrypted, it can only be decrypted or even executed if you know the correct password.

If you want to incorporate certain scripts permanently in your JMP environment, it is possible to extend the JMP menu and to link a self defined menu to the script file. Other "deployment containers" for scripts are JMP journal and project files. Both of them can contain links to script files which can be executed by a user, when he has opened the Journal or Project file. It is also possible to incorporate links to script files stored in a central location on a write-protected network drive within the network of your company. Using a combination of this approach with file access rights set appropriately to your needs, you could avoid that users are able to modify your script files.

CUSTOM REPORTS

When you are developing scripts that generate custom JMP reports, a very useful approach is to reuse components of reports generated by build-in JMP methods. If you want to access components of an existing report by a script (e.g. a certain column of a table in the report), a very useful tool is to view the display tree of the existing report. To get the display tree for a given report window within JMP, hold down the Control and Shift keys while right-clicking on a blue triangle in the report; select Edit > Show Tree Structure from the menu that appears. In the display tree, you see the type and id number of every component arranged into a hierarchical tree structure representation of the report. Holding a reference to the report window in a variable, it is possible to get single components of the report by functions using the id num and the type of a report component.

```
REPORT[BOXTYPE(N)]; //FINDS THE NTH DISPLAY BOX OF TYPE BOXTYPE IN REPORT VARIABLE  
//REPORT
```

However, I recommend to use mainly queries with component names (which is possible with report elements of type outlinebox and columnbox) instead of id numbers, because if you use references to ids and the report is changed later by adding or deleting components, all id numbers of components of the same type can change, as the ids are simply enumerations of all components of one type in a report. Using names, you do not get an error if the report changes due to modification of scripts executed for generating the report.

EXTERNAL TOOLS

Last, but not least, I recommend to use external software tools to support your JSL scripting. First and most

PhUSE 2009

important, a version control system (e.g. Subversion; <http://subversion.tigris.org/>) should be used, most importantly (among other good reasons) because it takes away the fear of changing a running script and messing up your source code irreversibly.

Next, I recommend to use a source code documentation system like Doxygen (<http://www.doxygen.org>), because it enables you to generate (at least a part) of the technical documentation of your source code.

Additionally, in large projects where you have to deal with many JSL files, you might consider to use a build tool (e.g. Apache Ant; <http://ant.apache.org/>) to automate some tasks (e.g. copy actual version of your source code files and archive them into a zip-file).

CONCLUSION

In conclusion, JMP offers a mighty scripting language for automation or enhancement of JMP functionality. Following the recommendations in this paper will hopefully help you implementing even large projects effectively with JSL.

VERSIONS OF USED SOFTWARE COMPONENTS

Software	Version
JMP	8

REFERENCES

JMP Scripting Guide:

SAS Institute Inc. 2008. JMP Scripting Guide. Cary, NC: SAS Press.

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
Email: patrick.warnat@analytical-software.de
Web: <http://www.analytical-software.de>

Brand and product names are trademarks of their respective companies.