How to Create Smalltalk Scripts

Juanita J. Ewing
Instantiations, Inc.

Derived from Smalltalk Report

In my last column I discussed a critical concept in Smalltalk application development: never view your image as a permanent entity. Most of the last column was centered on extracting application source from an image and rebuilding the image by recreating classes and methods from source. But in complex applications it is not enough to recreate classes and methods. Classes, pools and global variables must be defined and initialized, usually with a script.

This column tells you how to construct Smalltalk scripts. It includes a discussion of some useful expressions for scripts, and describes some useful structuring mechanisms for scripts.

Format

A Smalltalk script is really just Smalltalk code stored in a file, usually in file–out format[1]. This is the format with the !'s. Most people are used to seeing class and method definitions in this format. The most common way to create a file in this format is to use a *file–out* menu item in a browser.

A class definition consists of the definition string followed by a single exclamation point.

```
Point subclass: Object
      instanceVariableNames: 'x y'
      classVariablesNames: ' '
      pools: ' '!
```

Method definitions are more complicated. There is a header portion indicating the class to which the methods belong. The header portion begins and end with exclamation points. The header is followed by one or more method definitions, each ending with a single exclamation point. After all method definitions, another exclamation point is appended, indicating that the series of method definitions is over.

---

[1]Described in Chapter 3 , The Smalltalk-80 Code File Format, of Smalltalk-80 : Bits of History, Words of Advice

```
! Point methods !

x
        "Return the x component of the receiver"

        ^x !

= aPoint
        "Return true if both the x and y coordinates are equal."

        ^self x = aPoint x and: [self y = aPoint y]! !
```

Most implementations are very picky about a separating character between the last two exclamation points. This is because exclamation points in the definition source are doubled when written to a file. If there is no separator between the exclamations, the system will interpret it as a single exclamation in the method source.

The header portion of method definitions varies from implementation to implementation. The example in figure 2 uses a Smalltalk/V style header. In Smalltalk-80 derived implementations, the header also identifies a protocol.

```
!Point methodsFor: 'accessing'!

x
        "Return the x component of the receiver"

        ^x! !
```

Other interesting pieces of code that people want to put in scripts are really just do–its. It turns out that class definitions are also do–its, so we already know the proper format. It is code followed by a single exclamation point.

Useful Script Expressions

I usually start my file–in scripts with a comment, describing the contents of the script and any relevant assumptions. Filing in a comment has no effect on your image, but it is a handy way to document the contents of a file. Just like any other do–it, an exclamation must follow the expression.

```
"This file contains the script to load the drawing application.
This load procedure has been tested with version 1.4" !
```

Another common expression in a file is a class initialization. In this example the message initialize is sent to the DrawingApplication class. The appropriate initialization method will vary from class to class.

```
DrawingApplication initialize !
```

It might be appropriate to query the user for the location of relevant files before proceeding. Note the use of a temporary variable in this expression.

```
| directory |
directory := Prompter prompt: 'Where is the archive
directory?'.
directory isEmpty
        ifFalse: [(Disk file: directory, '\archive') fileIn] !
```

A dialog with the user might be appropriate during the file–in process, particularly if the expression is destructive. In this example, a global name is going to be removed from the system. Place interaction with the user at the beginning of the script to allow automated builds.

```
| confirm |
confirm := Prompter confirm: 'The next step is irreversible.
Continue?'.
confirm ifTrue: [Smalltalk removeKey: #Vector] !
```

Some applications make use of global variables. Global variables can be declared and initialized in scripts. Current ways to declare global variables reveal some implementation details of the global name space in Smalltalk implementations. Global variables are stored as symbols in Smalltalk, which is a dictionary. Don't forget the # mark which creates a symbol literal in the expression. In our example we create two global variables, the first with an initial value of nil. There is no convenient way to declare a global variable without an initial value, so we use the value nil. Nil is used as the initial value of variables in other places in the Smalltalk system.

```
Smalltalk at: #DrawingMode put: nil.
Smalltalk at: #DefaultColor put: ClrBlack !
```

It is also possible to test if a particular global name has been defined. This can be useful when combining segments of an application in a mix and match style. In the first expression the existence of the global name Vector is tested for, and if it is not defined, then the file containing its definition is loaded. This type of expression is really ad hoc configuration management.

In the second expression the existence of Vector is tested for and a message displayed if the name is already defined. Since the user doesn't furnish any meaningful input, a better alternative is to write messages to the Transcript instead of putting dialogs in the middle of a script.

```
Smalltalk
        at: #Vector
        ifAbsent: [(Disk file: 'Vector.st') fileIn] !

(Smalltalk includesKey: #Vector)
        ifTrue: [MessageBox message: 'About to redefine Vector'] !
```

Another type of global that needs to be declared is a pool. Current ways to declared pools also reveal some implementation details. Pools are dictionaries, and the keys are available in the methods of classes using the pool. Note that the declaration of the pool is a separate expression from the subsequent references to it. Each expression is independently compiled. The first expression is compiled and executed, which declares the pool if neccessary. We avoid redefining the pool if it already exists because that would orphan existing references to its variables. After the pool has been declared, subseqent do–its can reference it by name. The second expression defines three pool variables. This example is appropriate for Smalltalk/V systems. In Smalltalk–80 derived systems, the keys should be symbols.

```
Smalltalk
        at: #TypesettingConstants
        ifAbsent: [Smalltalk at: #TypesettingConstants put: Dictionary
new] !

TypesettingConstants at: 'Bold' put: '.B'.
TypesettingConstants at: 'Italic' put: '.I'.
TypesettingConstants at: 'Underline' put: '.U' !
```

Structuring Scripts

Do–its in a workspace or file are executed, logged in the changes file, and never referenced again by the system. Typical Smalltalk source control mechanisms don't capture do–its, and are therefore do–its are difficult to maintain. To overcome this problem in scripts, which typically have many do–its, developers should, whenever possible, turn do–its into methods. Methods are maintained by the Smalltalk system, and can be browsed and filed–out. They don't disappear after execution. An expression to initialize a class variable, for example, can be turned into a class method.

Files are the basis of another structuring mechanism. Application source can be composed of multiple files based on functionality. Several files based on

functionality are more reusable than a single large application file. It is easier to distribute and use a piece of functionality if it is separated from the rest of an application. Because extracting a unit of functionality from a large application source file is very challenging, interesting functionality will not be reused if it is not separated.

Even though application source in separated into multiple files, the application can be reconstructed quite easily. Scripts often load a series of files in a particular order. In this expression three files are loaded into an image.

```
(Disk file: 'enhancements.st') fileIn.
(Disk file: 'classes.st') fileIn.
(Disk file 'initialization.st') fileIn !
```

An alternative equivalent expression, which is easier to extend, is

```
#(
'enhancements.st'
'classes.st'
'initialization.st')
    do:
        [:each | (Disk file: each) fileIn ] !
```

The final structuring mechanism to discuss is based on a class. In this mechanism, we devote an entire class to rebuilding an application. This class probably also has functionality to store the source for an application. Do–it expressions that are not related to a class should be incorporated into methods in the rebuilding class. For example, an expression that creates and initializes a global variable should become a method. Then, all methods that create global variables should be called from a controlling method.

```
initializeGlobals
        "Define and initialize global variables."

        self initializeDrawingGlobal.
        self initializeLabelGlobal.
        self initializeDrawingLocationGlobal
```

Developers should create a similar set of methods for defining and initializing pools. The entire rebuilding class can now be maintained by the Smalltalk system, instead equivalent code maintained by the developer in script files. The source for the rebuilding class also needs to be archived in the same manner as the source for the rest of the application.

Conclusion

The ability to declare globals and pools, and initialize classes in a non-interactive mode is important in rebuilding complex Smalltalk applications. Understanding the file–in format and having a few examples can go a long way toward creating effective scripts, but script code should be turned into methods and classes whenever possible. Be wary of scripts and initialization methods that are too complicated, because they are difficult to debug and maintain.