How to Manage Source without Tools

Juanita J. Ewing

Instantiations, Inc.

Copyright 1994, Juanita J. Ewing

Derived from Smalltalk Report

Many Smalltalk programmers develop significant applications without any source management tools. Although it takes a certain amount of discipline, small to medium sized applications can be developed without additional tools. This column will describe several sound practices for the successful management of application source.

The code in this column is for versions of Smalltalk/V under Windows and OS/2. The ideas are applicable to other versions of Smalltalk/V and to Objectworks\Smalltalk.

Concepts

There is one concept that is critical for successful management of application source:

- Never view your image as a permanent entity.

And there are two corollaries:

- Don't depend on your image as the only form of your application.
- Store your application in source form and rebuild your image frequently.

Viewing the image as a non-permanent entity doesn't necessarily imply that vendors are selling unreliable software. There are several ways an image can become non-functional, other than a serious Smalltalk bug or a disk crash.

An image can become unusable because of some simple mistake on the part of a developer, such as accidentally removing a class that is relevant to the application under development. If the image is the only form of an application, recovering sources for application class can be difficult and tedious. Another common mistake is the accidental deletion of the change log or changes file. All the source for all the changes you've made to an image is stored in this file.

Not all motivation for storing your application outside of a image is because of mistakes. When your vendor releases a new version, migration to the new version may be necessary to take advantage of new features or to continue the highest level of technical support.

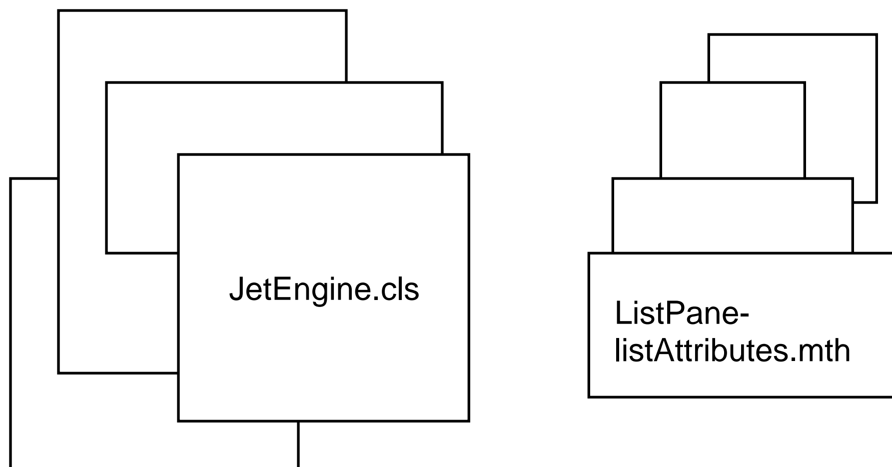
Practice

What is your application? In Smalltalk, this is not always a straightforward answer. Images contain large class libraries, and applications are developed by adding to and modifying the class libraries. There is no clear distinction between system and application code. Because of this, it is very difficult to extract all parts of an application from an image, especially after the development is completed. It is better to extract or list the parts of your application as you develop it, when short term memory can help you decide if the modification you made was necessary for your application, or a temporary modification for debugging. One of the most common errors is to leave out a critical piece of your application.

I will discuss two techniques for extracting your application code as you develop. The first technique uses the browser to file out code right after it is developed. Most application code will be located in new classes, which can be filed out as a unit. Other application components are extensions to system classes, which can be filed out at the method level. The result of this technique is many small files.

There are dependencies among the classes defined in these files. For example, a subclass depends on its superclass. I use a script to reassemble all these files in the correct order, rather than try to remember what the dependencies are. It is possible to create the script for reassembly at the same time the parts of an application are filed out.

Figure 1 contains a script for installing multiple files. The script consists of a list of file names, which is enumerated over to install each file into the image.




```
"Read and file in application files."

#(
'ExtendedListPane.cls'
'AviationGraphPane.cls'
```

```
'JetEngine.cls'
'PropEngine.cls'
'RudderMechanics.cls'
'ListPane-class-supportedEvents.mth'
'ListPane-listAttributes.mth'
'ListPane-listAttributes:.mth'
'GraphicsMedium-bezierCurve:.mth'
)
    do:
        [:fileName |
        (Disk file: fileName) fileIn]
```

Figure 1. Example of reconstructing an application using multiple files.

Another technique is to make a list of all the relevant application pieces as they are developed. The list can be maintained in order of reassembly, and used to extract all the components of an application on demand. The result of extraction is a single file. Reconstruction of the application is a simple matter of installing one file. The source can be partitioned into several files if necessary.

In Figure 2, the script has three lists, one for classes, one for instance methods and one for class methods. The classes listed in the first script are written to the stream, then the methods in the second list are written to the stream. The file out code makes use of ClassReader, which knows about Smalltalk source file format.

```
| sourceStream reader |

"Create file stream for storing sources."

sourceStream := Disk file: 'AviationSource.st'.

"Write application classes."

#(
ExtendedListPane
AviationGraphPane
JetEngine
PropEngine
RudderMechanics)
    do:
        [:className |
        reader := ClassReader forClass: (Smalltalk at:
className).
        reader fileOutClassOn:  sourceStream].

"Write standalone instance methods"

#(
(ListPane listAttributes)
```

```
(ListPane listAttributes:)
(GraphicsMedium bezierCurve:)
)
    do:
        [:classNameAndSelector |
        reader := ClassReader forClass: (Smalltalk at:
(classNameAndSelector at: 1)).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on: sourceStream].

"Write standalone class methods"

#(
(ListPane supportedEvents)
)
    do:
        [:classNameAndSelector |
        reader := ClassReader forClass: (Smalltalk at:
(classNameAndSelector at: 1) class).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on: sourceStream].
sourceStream close
```

Figure 2. Example of creating a single file for application reconstruction.

This script makes use of a new method, fileOutClassOn:, defined in Figure 3. The new method, which writes a class definition and its methods on a stream, takes an instance of FileStream as an argument. It is similar to an existing method, fileOut:, that takes a file name as an argument, creates the file and writes a class and its methods to the file.

```
ClassReader

instance method

fileOutClassOn: aFileStream
    "Write the source for the class (including the class
    definition, instance methods, and class methods) in
    chunk file format  to aFileStream."

    class isNil ifTrue: [^self].
    CursorManager execute change.
    aFileStream lineDelimiter: Cr.
    class fileOutOn: aFileStream.
    aFileStream nextChunkPut: String new.
    (ClassReader forClass: class class) fileOutOn:
aFileStream.
```

```
self fileOutOn: aFileStream.
CursorManager normal change
```

Figure 3. Supporting code in ClassReader for filing out a class onto a stream.

The script in Figure 2 works in the simplest cases, in which there are no forward references to classes. For example, if code in the class JetEngine refers to the class PropEngine, the file in will not proceed properly. This problem can be avoided by defining all classes before any methods. as in the script in Figure 4. This script also has two lists, but the first list is enumerated over twice. A supporting method is defined in Figure 5.

```
| sourceStream classList reader |

“Create file stream for storing sources.”

sourceStream := Disk file: ‘AviationSource.st’.

“Classes in the application”

classList := #(
ExtendedListPane
AviationGraphPane
JetEngine
PropEngine
RudderMechanics).

“Write application class definitions.”

classList
    do:
        [:className |
        reader := ClassReader forClass: (Smalltalk at:
className).
        reader fileOutClassDefinitionOn:  sourceStream].

“Write the methods for the application class”

classList
    do:
        [:className |
        reader := ClassReader forClass: (Smalltalk at:
className).
        reader fileOutOn:  sourceStream].

“Write standalone instance methods”

#(
(ListPane listAttributes)
```

```
(ListPane listAttributes:)
(GraphicsMedium bezierCurve:)
)
    do:
        [:classNameAndSelector |
        reader := ClassReader forClass: (Smalltalk at:
(classNameAndSelector at: 1)).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on: sourceStream].



"Write standalone class methods"

#(
(ListPane supportedEvents)
)
    do:
        [:classNameAndSelector |
        reader := ClassReader forClass: (Smalltalk at:
(classNameAndSelector at: 1) class).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on: sourceStream].
sourceStream close.
```

Figure 4. Example of creating a single file for application reconstruction.

```
fileOutClassDefinitionOn: aFileStream

    "Write the source for the class (but not for the instance
      methods and class methods) in chunk file format
    to aFileStream."

    class isNil ifTrue: [^self].
    CursorManager execute change.
    aFileStream lineDelimiter: Cr.
    class fileOutOn: aFileStream.
    aFileStream nextChunkPut: String new.
    CursorManager normal change
```

Figure 5. Supporting code in ClassReader for filing out a class definition without methods.

Initialization

Applications consist of more than classes and methods. Instances of windows, panes and domain-specific classes are part of an application. Application reconstruction, therefore,

must consists of more than filing in class and methods. The expressions executed in a workspace or inspector to set up the state of your application, such as initializing classes and creating new objects, need to be re-executed when your application is reconstructed. Save these expressions by collecting them in a file and executing them after reconstructing your application. In a future column I will discuss these types of expressions, and ways to execute them as part of a script.

Errors

The most error-prone portion of these techniques is recording pieces of the application as it is developed. That's why source management tools are so valuable: they record this information automatically. Because the pieces of the application are recorded by hand, it is also common practice to search back through the change log to make sure no pieces of the application have been forgotten. This activity is usually performed in a regular fashion, such as before each snapshot.

Another common error is to rebuild an application on top of an image that has been used for development. This is not a good idea because the state of the image is unknown. There may be unwanted side effects from objects in the image. It is imperative that the application is reconstructed from a clean, pristine image.

Frequency

How often should the application be rebuilt? Early in development, when many classes are being created, the scripts are being modified rapidly. It valuable to rebuild often to test the scripts. If the scripts are too far out of synch with the application source, it can be difficult to debug the reconstruction process. In the middle stages of development the scripts are not in so much flux, and the application doesn't need to be rebuilt so often to test them out. Other considerations may force application reconstruction, such as the redesign of parts of an application. As the product is nearing completion, the development team may want to reconstruct the application often to confirm that the build process is bug free.