Pools: An Attractive Nuisance

Juanita J. Ewing
Instantiations, Inc.

Derived from Smalltalk Report

Pools are a Smalltalk language construct for sharing data between class. Classes that share data using pools are not required to be hierarchically related. At first blush, pools sound attractive: pools allow functionally–related classes to connect by sharing data.

However, pools are not without problems. We will see that they are poorly supported in most Smalltalk implementations and limit reusability. Hence, they are labeled an attractive nuisance.

What are Pools?
Pools are dictionaries of variables. The variables in a pool are called pool variables. Each variable has a value which is often a constant, but there is no language constraint that the variables be constants.

How do you use Pools?
When you define a class, you can instruct the Smalltalk compiler to use pool variable names when compiling instance and class methods. In some Smalltalk implementations, pool access is inherited by subclasses.

Show me.
Suppose you have a class, called Stream, that is used for reading and writing data. You also have a pool called CharacterConstants, containing variables that describe commonly used characters. It would define variables such as Cr, Lf and Tab. You could use the CharacterConstants pool to implement methods for writing formatted data to the stream.

First, let's examine the Stream class definition:

```
Object
  subclass: #Stream
  instanceVariableNames: 'collection position size'
  classVariableNames: ''
  poolDictionaries: 'CharacterConstants'
```

In addition to instance variables and class variables, developers can customize the set of pools used by the class. Stream uses one pool, CharacterConstants.

Next, let's examine the method nextLine that uses variables from the pool CharacterConstants. Both Cr and Lf are pool variables from CharacterConstants.

```
Stream
nextLine
    "Answer a String consisting of the characters of the receiver up to the next line delimiter."

    | answer |
    answer := self upTo: Cr.
    self peekFor: Lf.
    ^answer
```

What's wrong with that?
Nothing's wrong so far. Now let's try some important operations like defining a new pool. The traditional way to define a new pool in Smalltalk is to create a global variable whose value must be a dictionary. Then the user must populate the dictionary with keys that will be interpreted as variable names when the dictionary is used as a pool. Some implementations of Smalltalk require the keys in the dictionary to be Strings, others require Symbols. The code to create a pool looks like this:

```
"Declare the variable"
Smalltalk at: #MyUIConstants put: nil.

"Create the dictionary."
MyUIConstants := Dictionary new.

"Populate"
MyUIConstants at: 'BackgroundColor' put: Color paleYellow.
MyUIConstants at: 'ForegroundColor' put: Color blue.
MyUIConstants at: 'TextColor' put: Color black.
MyUIConstants at: 'TextHighlightColor' put: Color darkYellow.
```

How do you use that pool?
In this example, we created a pool called MyUIConstants, and filled it with color values. Now we can use this pool in the definition of the class TextWidget:

```
Widget
  subclass: #TextWidget
  instanceVariableNames: 'contents'
  classVariableNames: ''
  poolDictionaries: 'MyUIConstants'
```

Why did we use a dictionary to define a pool?

You'll notice that creating a pool didn't involve any expressions of the form Smalltalk createPoolNamed: #MyUIConstants. Instead, we create a global variable and set its value to a dictionary.

Historically, pools were never formally defined as first-class elements of the Smalltalk language. There is no syntax for defining pools or pool variables. Instead, the exact implementation of the pool language construct is known and relied on by developers. This isn't a good idea because it prevents vendors from improving the implementation of pools– future versions of Smalltalk may not even use dictionaries to implement pools. It also makes it difficult for developers to move their code to different Smalltalk platforms that have a different implementation of pools. The worst thing, though, is that developers write code that treat pools as dictionaries.

What problems result from treating pools as dictionaries?
Because pools are globals and available from every method, and their implementation is known, developers are very tempted to write code like this:

MyUIConstants at: 'TextColor' ifAbsent: [^nil]

The problem with this code is that the compiler does not detect it as a pool variable reference. It is just a message send to a global variable. Thus, the Smalltalk programming environment cannot reason about this expression as a pool variable reference.

This type of reasoning would be important if you were redesigning your program, and were considering eliminating the pool variable TextColor. If you asked the programming environment to search for all references to the pool variable TextColor, it would not find this one.

Another problem related to the public implementation of pools and the availability of a pool in the global name scope, is inappropriate access of pool variables. Pool variables can be accessed in any method, not just methods in classes that define usage of the pool. If the pool is treated like a dictionary, you can send it a message to access its contents, which are pool variables. For example, the expression ColorConstants at: 'ClrBlue' provides access to the pool variable ClrBlue.

How do I store pools?
A source form of a Smalltalk application is more than just a rarely used archiving artifact. It is a necessity for serious developers. For a more complete discussion on the benefits of storing source, see my Smalltalk Report column on *How to Manage Source Without Tools*, Volume 2 Number 3.

The typical way developers create a source form of their application is by filing classes out of an image. When developers file classes in and out of an image, they

encounter another problem with pools. A class that references a pool can file out without a problem, but its pools are not filed out. Because pools are shared between classes, it would be inappropriate to file them out with any single class. Instead, pools should be independently stored in source form.

In a standard Smalltalk development environment, there is no way to store pools in source form. Most Smalltalk environments don't even define a source form.[1]

[1] Digitalk's Team/V does provide formal support for pools.

The pools must be present, however, when you file your class back in.

What lessons have developers learned?
If developers have defined and used pools before, they have learned to save the code they used to create the pool, and execute it again to recreate pools when rebuilding their development environment. This is typically some workspace code, and it is usually saved in a file.

To rebuild their development environment, developers must manually track which classes require which pools, and rebuild their development environment with a combination of source code to re-create classes and executable code to recreate global and pools.

Developers have also learned to write their pool definition code carefully because pool definition code is fragile. If you overwrite an existing pool by creating a new dictionary, any existing code using pool variables will be disconnected from the pool. Changes to the pool will not be tracked by the compiled code.

Suppose you have a class, TextWidget, that uses the pool MyUIConstants. The method initialize uses two pool variables, TextColor and TextHighlightColor.

TextWidget
initialize
```
"Initialize the receiver for standard look and feel."

self setTextColor: TextColor.
self setHighlightColor: TextHighlightColor
```

You can redefine the pool with this expression:

```
Smalltalk at: #MyUIConstants put: Dictionary new.
```

But, the redefinition breaks the connection between the pool and existing references from methods. This is because you have create a new pool that happens to have the same name as the old pool. You are not redefining the old pool. Even if you populate the new dictionary with identical variables you cannot re-establish the connections:

```
MyUIConstants at: 'BackgroundColor' put: Color black.
MyUIConstants at: 'ForegroundColor' put: Color blue.
MyUIConstants at: 'TextColor' put: Color white.
MyUIConstants at: 'TextHighlightColor' put: Color green.
```

The original value of TextColor was black. In the new pool MyUIConstants, its value is white. The initialize method still contains a reference to the old pool variable TextColor, and initializes TextWidgets to have the a black text color.

Examining the source of the method gives no clue about the current state of the compiled code. Recompiling the method will allows the compiler to rebind the reference.

To avoid redefining existing pools, developers usually place conditionals around pool creation expressions (requiring further assumptions about the implementation of pools):

```
(Smalltalk includesKey: #MyUIConstants)
        ifFalse: [Smalltalk at: #MyUIConstants put: Dictionary new]
```

Accidental pool redefinition is another reason why it is dangerous to allow the implementation of pools to be known.

What impact do pools have on reusability?
Pools have a negative impact on the sacred cow of Smalltalk, reusability. Let's examine our definition of pools again: a construct for sharing **data** between classes. In other words, pools contain data and do not define behavior. The two main mechanisms for reuse in Smalltalk are inheritance and polymorphism. Both are focused on behavior. They rely on behavior functioning on encapsulated data–exactly the opposite of what pools provide.

Let's look at an example with the pool MyUIConstants. The class TextWidget uses the pool to access user interface constants. The method initialize is implemented as follows:

TextWidget
initialize

```
        "Initialize the receiver for standard look and feel."

        self setTextColor: TextColor.
        self setHighlightColor: TextHighlightColor
```

Suppose we make a variation of TextWidget that has a different highlight color. We don't want to change the original class, so we create a subclass of TextWidget that uses the MyUIConstants pool. And, we add a new pool variable to represent the new highlight color, called MarkupTextHighlightColor.

```
MyUIConstants at: 'MarkupTextHighlightColor' put: Color darkYellow.
```

Because the inherited initialize method contains a direct reference to the pool variable, we are forced to override the entire initialize method instead of just overriding the color specification for text highlight. Here is the new initialize method for the subclass:

MarkupTextWidget

initialize
"Initialize the receiver for mark up look and feel."

self setTextColor: TextColor.
self setHighlightColor: MarkupTextHighlightColor

A better way to write this code is to isolate and encapsulate references to the constants in this method. Then subclasses can override the encapsulating methods if necessary. Remember though, that these constants are used in several classes. It may be better, depending on the usage, it the constants are encapsulated in methods from a stand-alone class.

A new class, WidgetUIConstants, could function as the encapsulator for all user interface constants. It would respond to messages like foregroundColor and textHighlightColor. Straightforward use of this class would look like this:

TextWidget
widgetConstantClass
"Return the class containing user interface constants."

^WidgetUIConstants

initialize
"Initialize the receiver for mark up look and feel."

self setTextColor: self widgetConstantClass textColor.
self setHighlightColor: self widgetConstantClass textHighlightColor

A new subclass of WidgetUIConstants could contain the variations appropriate for the subclass MarkupTextWidget. MarkupTextWidget now needs to override the specification of the user interface constants class, but does not need to override the initialize method.

MarkupTextWidget
widgetConstantClass
"Return the class containing user interface constants."

^MarkupWidgetUIConstants

This example illustrates that it is not straightforward to override references to pool variable in a subclass. The override usually results in multiple methods that specify the same constants, which leads to maintenance problems. This is typically of the extensibility problems found in cases of direct variable references.

The added benefit of a stand-alone class alternative is that the class can be stored in source form and managed by ordinary Smalltalk tools. It can also be subclasses

to provide variations of the constants. Pools have neither of these capabilities.

Bottom Line
Behavior is better than data. Smalltalk reuse mechanisms work on behavior.

Because pools are data, avoid pools whenever possible. Instead, create a class that encapsulates the data in the pool and replace existing pool variable references with messages. Your code reuse and ability to store your application in source form will improve. Send your feedback on this discussion to juanita@digitalk.com.