Constants, Defaults and Reusability

Juanita J. Ewing
Instantiations, Inc.

Derived from Smalltalk Report

This column is focused on two aspects of reusability, subclassing and client usage, and how they relate to constants and defaults. Many classes have constants and defaults to represent commonly used values. Some of the values represented as constants may not really be constants, such as heuristically determined values. These values are often hard-coded and embedded into methods. Though expedient in the prototyping stage, most constants should evolve into defaults as classes are refined. Developers of reusable software need to create reasonable defaults and include a mechanism to override them.

This column will show you how to use constants and defaults and still maintain a high level of reusability. We will examine several classes and methods from the Windows and OS/2 versions of Smalltalk/V that contain defaults. We will also revise some existing image code that has embedded constants and improve its reusability.

Constants

Many initialization methods contain constants, and their values are often Smalltalk literals. In the class EntryField, the initialize method contains four constants: a string, an integer, a point and a boolean. An initialization method is an appropriate place for constants. Subclasses typically override the initialize method to customize initial values.

```
initialize
    "Private - Initialize the receiver."

    value := ''.
    maxSize := 32.
    selection := 1@1.
    modified := false.
    ^super initialize
```

A less appropriate location for constants is embedded in arbitrary methods. A method should have one purpose. It might be to define a default, or to perform some

computation, but it shouldn't be both. With an embedded constant, reusability is impacted because it is difficult to

- find and modify the constant
- override the constant in a subclass.

The method file: in DiskBrowser has a constant that controls file contents display based on the size of a file. This constant is a size limit used to determine whether to display the entire file or a portion of the file. If the file size exceeds this limit, it takes an extra action to see the entire contents. The main purpose of the file: method is to display the file contents. It should not contain the definition of the size limit also.

file: filePane

"Private - Set the selected file to the selected one in filePane. Display the file contents in the text pane."

```
| aFileStream |
CursorManager execute change.
self changed: #directorySort:.
selectedFile := filePane selectedItem.
self switchToFilePane.
aFileStream := selectedDirectory fileReadOnly: selectedFile.
wholeFileRequest := aFileStream size < 10000.
aFileStream close.
wholeFileRequest
        ifTrue: [self fileContents: contentsPane]
        ifFalse: [self showPartialFile]
```

Another DiskBrowser method, showPartialFile, also contains this constant. Having the same embedded constant in two method can lead to maintenance problems.

showPartialFile

"Private - Display the head and tail of the selected file in the text pane."

```
| aFileStream fileHead fileTail startMessage endMessage cr |
CursorManager execute change.
contentsPane modified: false.
aFileStream := selectedDirectory fileReadOnly: selectedFile.
 cr := String with: Cr with: Lf.
startMessage := 'File size is greater than 10000 bytes, ', cr, 'first 1000 bytes are
...', cr.
endMessage := cr, '**********', cr, 'last 9000 bytes are ...', cr.
fileHead := aFileStream copyFrom: 1 to: 1000.
fileTail := aFileStream
        copyFrom: aFileStream size - 9000
        to: aFileStream size.
aFileStream close.
```

```
contentsPane
        fileInFrom: (ReadStream on: (startMessage, fileHead, endMessage,
fileTail));
        forceSelectionOntoDisplay.
(self menuWindow menuTitled: '&Files') enableItem: #loadEntireFile.
(self menuWindow menuTitled: '&File') disableItem: #accept.
CursorManager normal change
```

Defaults

Developers should not embed constants in arbitrary methods. Instead, each constant should be defined in a separate method, allowing it to be easily identified and overridden. Once isolated, we call these values defaults because subclasses can easily override the defining method, increasing the reusability of the class.

The method initWindowSize, from the class WindowDialog, specifies the initial size of a dialog. Because this value is isolated in a method, we consider it a default—subclasses can easily override the default initial window size.

```
initWindowSize
        "Private-Answer the default window size."

        ^150 @ 100
```

Another example from the image involves the application framework class ViewManager. The class ViewManager has a method that specifies the class of the top pane in the view structure. Subclasses can easily override this method to specify another top pane class, giving subclasses the critical ability to override the creation of collaborators.

```
topPaneClass
        "Private-Answer the default top pane class."

        ^TopPane
```

Evolving Constants into Defaults

In the section above, we saw two DiskBrowser methods containing an embedded constant, 10000. Next we see the two original methods rewritten, plus one other method that isolates the file size limit for automatic reading. The isolated constant is now a default because it can easily be overridden by subclasses. With a default, maintainers can more easily locate the limit, and are less likely to create inconsistent methods caused by modifying one reference to the constant but not the other reference.

```
autoReadLimit
        "Return the file size limit that determines whether the entire contents of a
        file will be automatically displayed."
```

```
^10000
```

file: filePane

"Private - Set the selected file to the selected one in filePane. Display the file contents in the text pane."

```
| aFileStream |
CursorManager execute change.
self changed: #directorySort:.
selectedFile := filePane selectedItem.
self switchToFilePane.
aFileStream := selectedDirectory fileReadOnly: selectedFile.
wholeFileRequest := aFileStream size < self autoReadLimit.
aFileStream close.
wholeFileRequest
        ifTrue: [self fileContents: contentsPane]
        ifFalse: [self showPartialFile]
```

showPartialFile

"Private - Display the head and tail of the selected file in the text pane."

```
| aFileStream fileHead fileTail startMessage endMessage cr limit initial
final |
CursorManager execute change.
limit := self autoReadLimit.
initial := limit // 10 roundTo: 1000.
final := limit - initial.
contentsPane modified: false.
aFileStream := selectedDirectory fileReadOnly: selectedFile.
cr := String with: Cr with: Lf.
startMessage :=
        'File size is greater than ', limit printString, ' bytes, ', cr,
        'first ', initial printString, ' bytes are ...', cr.
endMessage :=
        cr, '**********************', cr,
        'last ', final printString, ' bytes are ...', cr.
fileHead := aFileStream copyFrom: 1 to: initial.
fileTail := aFileStream
        copyFrom: aFileStream size - final
        to: aFileStream size.
aFileStream close.
contentsPane
        fileInFrom: (ReadStream on: (startMessage, fileHead, endMessage,
fileTail));
        forceSelectionOntoDisplay.
```

```
(self menuWindow menuTitled: '&Files') enableItem: #loadEntireFile.
(self menuWindow menuTitled: '&File') disableItem: #accept.
CursorManager normal change
```

Instances Modify Defaults

In addition to allowing subclasses to override defaults, developers can structure code so that instances can modify the default, improving client reuse. In this scenario, the class provides

- storage for the default value, usually an instance variable,
- accessing method for setting the default,
- accessing method for retrieving the default (optional).

The class EntryField has a default for the maximum number of characters in an instance of EntryField. In addition to the initialize method we saw above and an instance variable to hold the value, there is one other method that accesses the default maxSize. The accessing method maxSize: allows instances to customize the maximum number of characters that can be typed in an EntryField.

```
maxSize: anInteger
    "Set the maximum number of characters in the receiver to anInteger."

  maxSize := anInteger.
  handle = NullHandle
    ifFalse: [ self setTextLimit ]
```

There are several ways to provide an initial value for a default. In the initialize method for EntryField, maxSize is set to 32. An alternative design, shown below, has an accessing method that provides a default. The initialize method no longer sets the value of maxSize. In this case, the initial default value is only used if the default has not been otherwise set.

```
maxSize
    "Return the maximum number of characters that can be entered in the
receiver. If no other value has been set, use the initial max size value and
remember it."

    maxSize == nil
        ifTrue: [maxSize := self initialMaxSize].
    ^maxSize

initialMaxSize
    "Return the initial maximum size for text entry."

    ^32

initialize
    "Private - Initialize the receiver."
```

```
value := ''.
selection := 1@1.
modified := false.
^super initialize
```

Defaults Replace Arguments

Defaults can also be used to diminish interaction complexity. Commonly used values do not need to be passed as parameters. Instead, they can become defaults. Developers need to provide a way to override the default values, and still provide for the most common situations in which the defaults are an applicable value.

The typical way for developers to provide default arguments is with additional methods that leave out keywords. The method fill:rule:, from GraphicsTool, calls fill:rule:color: with the fill color set to the foreground color. The foreground color is a default. To override the default, the message fill:color:rule: can be sent.

```
fill: aRectangle rule: aRopConstant
        "Fill aRectangle in the receiver medium with foreColor using
aRopConstant. "

        self fill: aRectangle rule: aRopConstant color: foreColor
```

Conclusion

The important difference between constants and defaults is their affect on reusability. Defaults, isolated in a method, are easily overridden by subclasses. Default values can be modified by instances if developers add enough support or can be used to eliminate arguments and reduce interaction complexity. Developers should always strive to evolve constants into defaults in order to make their classes more reusable.