Appendix 5 - Style recommendations

Overview

This appendix contains a small selection of Smalltalk conventions, idioms, and patterns. The stylistic part contains a few simple and natural guidelines for naming variables and methods, writing comments, naming method protocols, and similar topics. The idioms and patterns part gives examples of some of the techniques used by experienced Smalltalk programmers.

In some details, stylistic guidelines and preferred patterns vary from one programmer to another and some of our suggestions thus differ from those recommended by some other authors. In the end, it is not important whether you follow our recommendations or somebody else's because what matters is to have a stable style rather than follow one particular style.

A 5.1 Introduction to Smalltalk style guidelines

Having a set of rules for writing Smalltalk programs makes programs easier to read and understand and this is very important for readability and understandability. If you are a part of a team or a student in a course, setting a common set of stylistic guidelines is essential. Most of the following guidelines have been extracted from a much larger set listed and justified in [Skublics, et al.] and [Beck]. We recommend these two highly respected books as additional reading.

A 5.2 Naming

Guideline N.1: *Uppercase and lowercase first letter.*
Start names of shared objects with upper case letter (classes, global variables, class variables, pools), start names of methods, keywords, parameters, and local names (instance, temporary, and block variables) with lower case.
Examples: Shared objects - Number, Smalltalk, Pi, TextConstants; methods and local objects - factorial, between:and:, width.

Guideline N.2: *Multiword names.*
When a name combines several words, start each word with a capital letter.
Examples: SmallInteger, studentNumber.

Guideline N.3: *Choice of name.*
Use short descriptive names and avoid abbreviations except where standard.
Examples: Number, today, factorial, between:and: , BTree (balanced tree)

Guideline N.4: *Names of methods.*
Method names should suggest the kind of object being returned.
Examples: isHungry, name, between:and:

Guideline N.5: *Names of keywords.*
Use names describing the role of parameters.
Examples: between:and:, name:address:

Guideline N.6: *Variable names versus argument names.*
Variable names should indicate the role of the variable, argument names should indicate the kind of argument expected.
Examples: Variable names - width, seed; argument names - name: aString, + aNumber

Guideline N.7: A*rgument names when type is repeated.*

When several arguments are of the same type, distinguish them by indicating their role.
Example: origin: originPoint corner: cornerPoint

Guideline N.8: *Names of block arguments*.
Name block arguments according to their role.
Example: employees do: [:employee| ...]

Guideline N.9: *Names of accessing methods.*
Name accessing methods by names of accessed variables.
Examples: Use width, width: to get or set variable width.

Guideline N.10: *Names of state-related test methods.*
Use a state-related phrase for tests and access of Boolean variables.
Examples: isEmpty, hasBorder, isOn

Guideline N.11: *Names of action methods.*
For methods that perform an action, use imperative form of verbs describing effect.
Examples: display:, enable, turnOn, turnOff

Guideline N.12: *Names of widget properties.*
Use widget label to name *Action*, *Selection*, and *Aspect* properties.
Examples: Use action method save for action button named *Save*, use selection property red for check box named *Red*, use aspect named name for input field with label *Name*.

Exercise

1. Browse the class library and find ten well selected names and five poorly selected names.

A 5.3 Comments

Guideline C.1: Use meaningful, short, and grammatically correct comments using correct spelling, full sentences, and standard English punctuation. Use active voice.
Example: "Create and return a Point at the given radius and angle."

Guideline C.2: Follow the Browser method template: Start all methods except accessing and very short self-explanatory methods with a comment explaining some or all of the following: the purpose of the method, its principle, its prerequisites, its effect, its use, the role of its arguments, the returned object.
Example: "Include all elements of aCollection as the receiver's elements. Answer aCollection."

Guideline C.3: Minimize the number of comments in method body - code should be self-explanatory.

Guideline C.4: For class comment, use the template displayed in the browser for class comment. Use active voice. If you wish to use first person, use it consistently.
Example: See comment of class Array, Float, Date for examples.

Exercise

1. Browse the class library and find five good and bad method and class comments.

A 5.4 Formatting

Guideline F.1: Use automatic formatting. Since the VisualWorks formatter exhibits some undesirable behavior, such as misplacing comments, we recommend using the formatter written by Randy Giffen and included in our software package. If your formatter allows several formatting styles, use one consistently.

Exercise

1. Write a method without any regard for formatting and format it. Compare the readability of the two versions.

A 5.5 Names of common protocols

Wherever appropriate, use the following protocol names common in the class library:

Class protocols

class initialization
examples
instance creation
interface spec
resources

Instance protocols

accessing
adding
arithmetic
converting
comparing
controller accessing
copying
events
displaying
enumerating
menu commands
printing
private
private - menu messages
removing
testing
updating

Exercises

1. Browse the class library and find the proportion of the above names among protocol names.
2. Browse the class library and find ten non-standard protocol names.

A 5.6 Introduction to idioms and patterns

Over the more than 20 years of Smalltalk existence, Smalltalk programmers invented and tested various ideas and evolved a body of rules that help writing readable, easily extendible, and efficient code. Why not take advantage of this body of knowledge and save yourself months of frustrating trial and error?

Most of the following guidelines have been borrowed from [Beck] and [Skublics, et al.]. Our list is limited to idioms and patterns used in this book and we recommend highly that you read the recommended references for more details and many more guidelines.

Some of the following guidelines can be classified as patterns, meaning that their validity extends beyond Smalltalk. Other guidelines can be classified as idioms because they are specific to Smalltalk and don't apply to other languages.

A 5.7 General patterns

Pattern G.1: *Don't Say Anything Twice*
When the same piece of code appears in several places in one class, convert it to a method. When the same object is calculated more than once in one method, cache it in an instance variable. When the same object is needed in several methods in the same class, create an instance variable to hold it.

Pattern G.2: *Lots of Little Pieces*
Shorter methods, and objects with fewer components are better. They are easier to understand and more reusable. Like all smaller things, they don't 'break' so easily to become useless.

Pattern G.3: *Think of Alternatives*
When designing a system or a class think of extensions and modifications. Can they be done by adding a few new classes or will some classes have to be rewritten?

Pattern G.4: *Think of Specialization*
If you anticipate subclassing, separate code into methods that will be inherited in their entirety from methods that will be redefined in subclasses.

A 5.8 Methods

Idiom M.1: *Short Method*
Keep methods to a few lines. Most methods should fit into the Browser window.

Idiom M.2: *Self-Contained Method*
A method should perform one identifiable task.

Idiom M.3: *Composed Method*
When a method gets too long, divide it into self-contained methods, all at the same level of abstraction.

Idiom M.4: *Constructor Method*
Creation methods should create ready-to-use objects. When appropriate, provide methods that create instances with initialized instance variables.
Example: Rectangle class >> origin: originPoint corner: cornerPoint rather than just Rectangle new.

Idiom M.5: *Debugging Method*
To support debugging, write a printOn: method early in the coding stage of a new class.

Idiom M.6: *Display Method*
To support display of selected aspects of an object in a widget such as a list, write a displayString method.

Idiom M.7: *Selection Avoiding Method - Polymorphism*
To avoid complicating code with multiple ifTrue:ifFalse: messages, use polymorphism - define the same selector with different implementations in all classes whose instances may be accessed in the same way.

Example: Geometric objects define methods displayStrokedOn: and displayFilledOn: and may thus be used interchangeably in code including enumeration. This avoids the need for testing what kind of geometric objects is being displayed, speeding up operation, making code more readable and more easily extendible.

Idiom M.8: *Double Dispatch Method*
To avoid complicating code with multiple ifTrue:ifFalse: messages when the combination of the receiver and the argument determine the operation, send a message to the argument.
Example: Implementation of + aNumber uses aNumber sumFromInteger: self in SmallInteger, aNumber sumFromFloat: self in class Float, and aNumber sumFromFraction: self in class Fraction

Idiom M.9: *Simple Delegation*
When an object needs work involving one of its components, delegate the message to the component.
Example: The printOn: method in Point sends printOn: to the x and y components. Note: A stricter definition of delegation classifies this principle as forwarding.

Idiom M.10: *Self Delegation*
When the delegate needs to know about the delegating object, send message with a for: keyword and self as the argument.
Example: A dialog window may need access to the instance variables of the application model that created it. If the dialog's model is different from the application model, open the modal dialog with openFor: self from the main application, thus providing access to the main application's state.

A 5.9 Behaviors

Idiom B.1: *Instance Variable Enumerator*
To enumerate over selected instance variables, create accessing methods for each variable, and use an array of symbols representing the variables' accessing methods and enumerate using perform:.
Example: Assume that we have variables and accessing methods called part1, part2, part3, part4, part5. To calculate the total of part values, use

```
#(#part1 #part2 #part3 #part4 #part5) inject: 0 into: [:part :total| (total + self perform: part) value]
```

Idiom B.2: *Pluggable Behavior*
When the method to be executed is not known beforehand, for example when the desired method depends on context, use one of the forms of perform: with a symbol representing the method.
Examples: Operation of widgets is based on using their aspect accessing method via perform: because the method needed to execute the method is not known until the user defines it as the widget's property. Similarly, execution of pop up menu commands is based on perform:.

Idiom V.3: *Redefining Equality*
If a new class cannot reuse the inherited equality, redefine = and hash. Both should use equality and hash of those instance variables that are relevant for comparison. The hash method frequently uses xor: or + on the components used for =. Remember that *two objects that are equal must have the same hash value* (but not necessarily the other way around).
Examples: Equality in class ColorValue is defined as

```
= aColor
        ^aColor class == self class and:
                [aColor scaledRed = red and:
                        [aColor scaledGreen = green and:
                                [aColor scaledBlue = blue]]]
```

and because it uses colors, hash is redefined as

```
hash
        ^red hash + green hash + blue hash
```

Class Rectangle redefines = in terms of equality of corners and origins, and hash is thus defined as

```
hash
        ^origin hash bitXor: corner hash
```

Idiom B.4: *Inclusion Test Reversal*
Testing whether an object has one of several possible values can often be performed more easily by testing whether the collection of all possible values includes the object.
Examples: The first impulse for testing whether a Character is a vowel is

```
isVowel
| lowercase |
        lowercase := self asLowercase.
        ^lowercase = $a | lowercase = $e | lowercase = $i | lowercase = $o | lowercase = $u
```

but a better solution is

```
isVowel
        ^#($a $e $i $o $u) includes: self asLowercase
```

or

```
isVowel
        ^'aeiou' includes: self asLowercase
```

A 5.10 Variables

Idiom V.1: *Explicit Default Initialization*
To initialize selected variables to default values, create a new method including initialize.
Example: If variable accounts is an OrderedCollection, include
accounts := OrderedCollection new
or a similar statement in the initialization method.

Idiom V.2: *Lazy Initialization*
If the value of a variable is not immediately required or may never be required, don't initialize it when the object is created. Instead, define a getter method that returns the value of the variable if the variable is not nil and assigns and returns the default value if the variable is nil.
Example: Accessing methods for widget aspects created with *Define* use lazy initialization.

Idiom V.3: *Default Value Method*
To make initialization more flexible, create a default method that calculates and returns the default value.
Examples: Abstract class Filename uses message defaultClass to determine which of its concrete subclasses will handle messages addressed to Filename objects. Class String uses the same principle. Changing the default then requires changing only the getter method.

Idiom V.4: *Constant Method*
To provide access to significant class values or to values shared across all instances, create a class variable or a class method that returns the value.
Examples: Class Float has method Pi that returns the value of π. Class ColorValue has class variables representing the most frequently used color values and class methods that return them.

Idiom V.5: *Direct Variable Access*
For better readability, access variables directly. Advantages: faster and protects encapsulation.
Examples: name := aString instead of self name: aString and width printString instead of self width printString.

Idiom V.6: *Indirect Variable Access*
For greater flexibility, access variables by accessing methods. Advantages: Provides a central place where change of variable occurs (easier to monitor), makes it easier to change internal class implementation. Disadvantage: Breaks encapsulation (other objects can access variables), negligible message send overhead.
Example: Lazy initialization (Idiom V.2) requires that the variable be always accessed by an accessing method.
Note: The opinion on the use of *Direct* versus *Indirect* access varies from one expert to another.

Idiom V.7: *Safe Collection Access*
When an object contains a collection, access its elements via enumeration rather than by accessing the collection directly.
Example: Assume that class Account contains variable transactions. Method transactionsDo: aBlock allows flexible access to all transactions without allowing the accessor to change the value of individual transactions which is what would happen if you provided accessor method transactions.

Idiom V.8: *Caching Temporary Variable*
Use temporary variable to preserve a calculated object used repeatedly in a method or a block. If the calculation is lengthy or repeated many times, this will improve execution speed.
Example: Replace
```
department employeesDo:
                    [:employee| Transcript show: 'Department ', department name, ' Employee ',
                                        employee name; cr]
```
with
```
| label |
label := 'Department, department name, 'Employee '.
department employeesDo: [:employee| Transcript show: label, employee name; cr]
```

Idiom V.9: *Explaining Temporary Variable*
When a calculation requires a complex expression, save its parts in well-named temporary variables to make the code more readable.
Example: Replace
```
tax :=    ((incomes inject: 0 into: [:income :sum| sum + income]) -
          (expenditures inject: 0 into: [:expenditure :sum| sum + expenditure]) -
          (donations max: (donations - minDonations) * donationsRate)) * taxRate
```
with
```
| totalIncome totalExpenditures deductibleDonations |
totalIncome := incomes inject: 0 into: [:income :sum| sum + income].
totalExpenditures := expenditures inject: 0 into: [:expenditure :sum| sum + expenditure].
deductibleDonations := donations max: (donations - minDonations) * donationsRate.
tax := (totalIncome - totalExpenditures - deductibleDonations) * taxRate
```

Idiom V.10: *Object Sharing Instance Variable*
When several methods in the same class need the same object, save it in an instance variable instead of passing it as method argument.

Idiom V.11: *Concatenating Stream*
When concatenating several collections (usually String objects) use a WriteStream and its contents method.
Example: The printString: method defined in Object creates a WriteStream, passes it as an argument to printOn:, and returns the contents.

Idiom V.12: *Cascading*
When sending a series of messages to the same receiver, use cascading. Emphasize the receiver by indentation to improve readability.
Example:
```
self builder componentAt: #save
            lookPreferences foregroundColor: color;
            enable;
            beVisible
```

<u>Idiom V.13:</u> *Adding, removing, and* yourself
Adding and removing methods in collection and stream classes return the object being added or removed rather than the modified receiver. Use message yourself to return the modified receiver.
Example: `newCollection := oldCollection add: 13; yourself`