Variables Limit Reusability

Allen Wirfs-Brock
allenw%spt.tek.com@relay.cs.net
(503) 627-6195

Brian Wilkerson
brianw%spt.tek.com@relay.cs.net
(503) 627-3294

P.O. Box 500, Mail Sta. 50-470
Tektronix, Inc.
Beaverton, OR 97077

ABSTRACT

One of the primary benefits of the object-oriented paradigm is the ability to increase programmer productivity by enabling the reuse of existing code. One way to reuse code is to refine or specialize it to create a new subclass. We believe that the direct reference of variables can be an impediment to the refinement of code. This in turn decreases programmer productivity. We first present some examples of when and why this occurs. We then present some programming conventions that will eliminate such problems and discuss the advantages and disadvantages of using these conventions.

Topic Area: Software Engineering

Keywords: Reusability, instance variable, object-oriented programming, Smalltalk, refinement, modularity.

Word Count: Approximately 4000

Variables Limit Reusability

Introduction

The object-oriented paradigm has proven to be a powerful programming methodology that can significantly contribute to programmer productivity. Much of this productivity improvement can be attributed to the fact that the methodology supports development of reusable software components. Two of the main principles of the paradigm that support reusability are modularity and refinement [Meye87].

Modularity

The principle of modularity states that problems should be broken into pieces. The unit of modularity in object-oriented programming languages is usually called a class. Modularity is enforced by:

- using a message-based interface to perform operations on objects, and
- hiding implementation details within a class.

Implementation details are hidden by encapsulating state within objects.

Refinement

The second principle, refinement, states that classes (the units of modularity) should be built in such a way that they can be incrementally modified for reuse. This incremental refinement, and the ability to reuse code, allows programmers to be far more productive than they might be in a conventional programming environment.

Designs are refined, and code is reused, chiefly through the mechanism of inheritance using subclassing. Subclasses inherit methods from their superclasses, and add methods of their own. If an inherited method adds behavior inappropriate to the subclass, it may be overridden. Indeed, it is a simple matter to change any design decision represented by a method.

Design Decisions

Modularity and refinement cannot be separated when you define a class. Modularity determines which classes you define. In defining a class, you must specify its private implementation and its public interface. Its private implementation may, in turn, have a significant effect on the ease with which the class may be refined.

The design decision that concerns us here is how to represent the state of an object. In Smalltalk-80™ [Gold83], you might choose to represent some state with a variable. The state represented by a variable may be accessed in either of two ways from within a method.

- The variable may be referred to by name. This is the most fundamental way to access the value stored in that variable.
- The object can send a message to itself, asking for the value stored in the variable. This message invokes a method, which accesses the variable by name and returns the value. This way is less direct than the first.

A subclass might need to change the design decision represented by a variable. In order to do so, the subclass must override all methods in which the variable has been referred to by name. Direct references to variables cannot be modified in any other way by subclasses. It may not necessarily be a simple matter to change any design decision represented by a variable if the variable has been referred to by name.

On the other hand, if a message has been sent to access the variable, the subclass need only override the accessing method. Subclassing and inheritance becomes much simpler. Indeed, in Smalltalk-80, instance, class, pool, and global variables all represent the state of an object, and therefore sending a message to access them is good object-oriented style.

These ideas can apply to any object-oriented programming language, but the following discussion uses examples written in Smalltalk-80.

Instance Variables

Assume that Rectangle is a class representing a rectangular area, having two instance variables (origin and extent), both of which are points. This class might have a method called center defined as:

```
center
    "Return the center point of the receiver."
    ↑origin + (extent // 2)
```

This method computes the center point of a rectangular region. Now imagine an application which needs to test the center point of a large number of rectangles. Performance analysis of this application might show that a significant amount of time is being spent performing the center computation. The designer of this application might decide that the program could be optimized by explicitly saving the center point as part of the internal state of objects representing rectangle. This could be accomplished by creating a new subclass of Rectangle, called, perhaps, CenteredRectangle, which has an extra instance variable (called center) to hold the value of the center point. The method center would then have to be overridden to be something like:

```
center
    "Return the center point of the receiver."
    ↑center
```

Would any other methods have to be overridden by this new class? It would have to initialize the new instance variable when a new CenteredRectangle is created, and update the instance variable if the position or size of the rectangle changes. Presume that the methods origin: and extent: are defined by class Rectangle as follows.

```
origin: aPoint
    "Change the origin of the receiver."
    origin ← aPoint

extent: aPoint
    "Change the size of the receiver."
    extent ← aPoint
```

These methods would have to be overridden by CenteredRectangle as follows in order to maintain the center value.

```
origin: aPoint
    "Change the origin of the receiver."
    origin ← aPoint.
    self computeCenter

extent: aPoint
    "Change the size of the receiver."
    extent ← aPoint.
    self computeCenter
```

A new method would have to be added to compute the center point.

```
computeCenter
    "Update the cached center point value."
    center ← self origin + (self extent // 2)
```

In this example, computeCenter is a new private method called whenever the origin or extent was modified to recompute the saved center point value. Would any methods other than origin: and extent: have to be overridden by CenteredRectangle? That depends upon whether there are any other methods which directly modify the instance variables origin and extent. If all other methods modify these instance variables by sends of origin: or extent: to self then no other methods would have to be modified to support CenteredRectangle. However, any method which directly modified those variables would also have to be overridden. For example, suppose the method width: had been defined in class Rectangle as shown below.

```
width: anInteger
    "Change the width of the receiver to be that specified by the argument."
    extent ← anInteger @ extent y
```

It would then have to be overridden, perhaps as follows.

```
width: anInteger
    "Change the width of the receiver to be that specified by the argument."
    "Remember to update center state."
    extent ← anInteger @ extent y.
    self computeCenter
```

Because the new definition of extent: automatically updates the center state, it would be preferable to override the method as follows.

```
width: anInteger
    "Change the width of the receiver to be that specified by the argument."
    self extent: anInteger @ extent y
```

In the Smalltalk-80 Version 2 virtual image there are fifteen methods in class Rectangle which directly modify either or both of its two instance variables. Because the instance variables are modified directly, each of these methods would have to be overridden in order to create a class like CenteredRectangle. If all of these methods had used accessing methods such as origin: and extent:, only those two methods would have had to be overridden to create the new subclass.

Is it only direct modifications of instance variables which cause problems? Is there any reason not to directly access the values of instance variables? Consider another example, also using class Rectangle.

Within the Tektronix Smalltalk implementation, an instance of class Rectangle occupies 20 bytes of storage. Instances of class Point also occupy 20 bytes. Since a rectangle references two Point instances, a total of 60 bytes of storage is needed for each rectangle.

If an application needed to use an extremely large number of rectangles, it might try to optimize its storage requirements by changing the representation of rectangles. If instances of class Rectangle directly stored the *x* and *y* values of the two points as instance variables, then a rectangle could be represented with 28 bytes of storage instead of 60 bytes. This could be accomplished by defining a new subclass of Rectangle which had two extra instance variables, originY and extentY. We might call this new class CompactRectangle. CompactRectangle would override methods as follows.

```
origin: aPoint
    "Change the origin of the receiver."

    origin ← aPoint x.    "Store the x coordinate in the old origin instance variable."
    originY ← aPoint y    "Store the y coordinate in a new instance variable."


extent: aPoint
    "Change the size of the receiver."

    extent ← aPoint x.    "Store the x coordinate in the old extent instance variable."
    extentY ← aPoint y    "Store the y coordinate in a new instance variable."


origin
    "Return the origin point of the receiver."

    ↑Point x: origin y: originY


extent
    "Return the extent point of the receiver."

    ↑Point x: extent y: extentY
```

Because the argument and result specifications of these methods are the same as for the original class Rectangle, any uses of Rectangle could be transparently replaced by CompactRectangle.

But what about the rest of the inherited methods from class Rectangle? Do they need to be overridden? Any method which directly accessed the instance variables origin and extent would have to be modified to use the accessing methods of the same name. Even if we assume that all the of the methods which directly modify the instance variables had already been changed as suggested in the previous example, the Smalltalk-80 Version 2 definition of class Rectangle

would still have 28 different methods which would have to be overridden because they directly access the instance variables. None of these methods would have to be overridden if they had been written using accessing methods instead of direct instance variable references.

If a few simple coding conventions were followed, the subclassing of methods would be greatly simplified.

- ☐ For each variable defined by a class, define two accessing methods: one to set the value of the variable, and one to retrieve the value of the variable.
- ☐ Variables should only be accessed or modified using message sends which invoke the accessing methods.
- ☐ An accessing method should only store or retrieve the value of its associated variable. It should perform no other computations.
- ☐ Because variable accessing methods reflect the internal representation of the object, they should be considered private methods.

We propose that variables never be referred to by name within methods. Instead, variables should be referred to in only two stylized ways, using accessing or modifying protocol as shown below.

Accessing Protocol

<variable name>
  ↑<variable name>

Modifying Protocol

<variable name>: argument
  <variable name> ← argument

In order to simplify it, the CompactRectangle example did not fully follow these rules. Specifically, it did not use accessing methods for the new instance variables. The preferred definition and implementation of class CompactRectangle would be as shown in the boxes below.

Class Definition

```
Rectangle subclass: #CompactRectangle
    instanceVariableNames:  'originY extentY'
    classVariableNames: ''
    poolVariableNames: ''
```

Accessing Methods

origin: aPoint

"Change the origin of the receiver."

```
self originX:  aPoint x.
self originY: aPoint y
```

extent: aPoint

"Change the size of the receiver."

```
self extentX: aPoint x.
self extentY: aPoint y
```

origin

"Return the origin point of the receiver."

```
↑Point x: self originX y: self originY
```

extent

"Return the extent point of the receiver."

```
Point x: self extentX y: self extentY
```

Private State Representation

originX: aNumber
"Save the state of the x coordinate of the origin."
origin ← aNumber *"Use inherited origin variable to store originX state."*

originY: aNumber
"Save the state of the y coordinate of the origin."
originY ← aNumber *"Use originY instance variable to store originY state."*

extentX: aNumber
"Save the state of the x coordinate of the extent."
extent ← aNumber *"Use inherited extent variable to store extentX state."*

extentY: aNumber
"Save the state of the y coordinate of the origin."
extentY ← aNumber *"Use extentY instance variable to store extentY state."*

originX
"Return the current state of the x coordinate of the origin."
↑*origin* *"Use inherited origin variable to store originX state."*

originY
"Return the current state of the y coordinate of the origin."
↑originY *"Use originY instance variable to store originY state."*

extentX
"Return the current state of the x coordinate of the extent."
↑extent *"Use inherited extent variable to store extentX state."*

extentY
"Return the current state of the y coordinate of the origin."
↑extentY *"Use extentY instance variable to store extentY state."*

Class Variables

So far, we have discussed only instance variables. But the same stylistic considerations apply to other kinds of variables as well. To illustrate this, here is an example showing how the above

guidelines may be applied to class variables.

What are the characteristics of a class variable? A class variable stores one value which is shared by a class and all its instances, as well as all of its subclasses and all *their* instances.

It is common in Smalltalk-80 for all windows of the same type to share a menu. For example, all workspaces share the same middle button menu. In order for them to do so, the menu is stored in one place. This place is typically a class variable. When the menu must be displayed, the methods to do so access the class variable.

Suppose now that an application wishes to use a window of a similar type, adding new items to its menu. The application dare not modify the class variable, because that will change the menu of all existing windows of the original type. Instead, the natural way to add a menu item is to subclass the class which stores the menu as its class variable. The subclass then replaces it with a new class variable which contains the new menu. This new subclass, however, must override every method that refers to the original class variable by name, replacing the direct reference to specify the new class variable instead. Not only is this a lot of work, but it creates a lot of unnecessary duplication as well.

For example, suppose that the class ApplicationController has a class variable called SizeMenu. This menu is initialized as follows:

Class method

```
initialize
    "Initialize the menu."
    SizeMenu ← PopUpMenu labels: 'small\medium\large' withCRs
```

When we access SizeMenu, we refer to it by name, as in the following method.

Instance method

```
changeSize
    "Pop up a menu to allow the user to change the size."
    self setSize: SizeMenu startUp
```

In order to subclass this class, you must override the class method initialize and declare a new class variable. You must also override the instance method changeSize (and any other method that references the variable by name), as shown below.

```
ApplicationController subclass: #MyApplicationController
    instanceVariableNames: ''
    classVariableNames: 'MySizeMenu'
    poolVariableNames: ''
```

Class method

```
initialize
    "Initialize the menu."

    MySizeMenu ← PopUpMenu labels: 'petite\small\medium\large\giant' withCRs
```

Instance method

```
changeSize
    "Pop up a menu to allow the user to change the size."

    self setSize: MySizeMenu startUp
```

A better way to implement ApplicationController would be to set up a class variable called SizeMenu, and initialize it, as before, but to access it through the accessing protocol sizeMenu and sizeMenu:. Subclasses would have to override the class variable again, and the class methods initialize, sizeMenu, and sizeMenu:, but only class protocols change. All instance methods would use these accessing methods.

It is very much to the point of the subclass that these accessing methods be overridden. Another programmer, browsing the subclass later, would not be forced to browse through many methods containing one slight modification irrelevant to the purpose of the method.

The superclass would define these methods as shown below.

Class methods

initialize
"Initialize the menu."
self sizeMenu: (PopUpMenu labels: 'small\medium\large' withCRs)

sizeMenu
"Answer the menu used to select a new size."
↑SizeMenu

sizeMenu: aMenu
"Set the menu used to select a new size to be the argument, aMenu."
SizeMenu ← aMenu

Instance method

changeSize
"Pop up a menu to allow the user to change the size."
self setSize: self class sizeMenu startUp

Then, in order to subclass this class, you need only override the class methods as follows.

Class methods

initialize
"Initialize the menu."
self sizeMenu: (PopUpMenu labels: 'petite\small\medium\large\giant' withCRs)

sizeMenu
"Answer the menu used to select a new size."
↑MySizeMenu

sizeMenu: aMenu
"Set the menu used to select a new size to be the argument, aMenu."
MySizeMenu ← aMenu

But that way, you still had to introduce a new class variable in the subclass. You could, however, use a class instance variable (that is, an instance variable of the class) in the superclass. That way, the subclass need only override *one* class method, initialize. Overriding initialize now does not change superclass menus. However, for the same reasons discussed above, the class instance variable should be referred to only with accessing methods.

```
ApplicationController class
    instanceVariableNames: 'sizeMenu'.
```

Class methods

```
initialize
    "Initialize the menu."
    self sizeMenu: (PopUpMenu labels: 'small\medium\large' withCRs)

sizeMenu
    "Answer the menu used to select a new size."
    ↑sizeMenu

sizeMenu: aMenu
    "Set the menu used to select a new size to be the argument, aMenu."
    sizeMenu ← aMenu
```

Instance method

```
changeSize
    "Pop up a menu to allow the user to change the size."
    self setSize: self class sizeMenu startUp
```

The subclass now need only override initialize as shown below.

Class method

```
initialize
    "Initialize the menu."
    self sizeMenu: (PopUpMenu labels: 'petite\small\medium\large\giant' withCRs)
```

Other Variables

Indeed, the four conventions recommended above should be applied to variables of *any* kind.

Pool Variables

For example, pool variables are variables which may be shared among classes outside of the inheritance hierarchy. Because the value stored in a pool variable may be shared by *any* class defined to do so, without even the control imposed by inheritance, direct references to pool variables can be even more difficult for the programmer to manage. Clearly, the conventions discussed above would simplify this state of affairs.

For example, in the Smalltalk-80 system, the class Paragraph refers to the pool variable CaretForm (from the pool TextConstants) directly. If a subclass wished to change the appearance of the insertion point cursor it would have to override all methods that directly refer to this variable. If those methods had sent an accessing message to the class instead, then only one method would need to be overridden.

Global Variables

The conventions outlined above defer a design decision. They make the code less direct, so that the decision can be changed by refinement. But sooner or later, the decision must be made.

Global variables in Smalltalk-80 are typically class names. As such, programmers tend to think of them as constant values. Global variables, then, might seem a safe and natural place to fix the design decision. But are they, really?

Let's discuss two examples, one simple, and one a bit less so, to see what effect the proposed conventions would have.

For the first example, let us presume a method, such as copy, that returns an instance of its own class. For example, suppose that the method for copying an instance of MyClass is defined as MyClass new. This method must be explicitly overridden in any subclass of MyClass, or else copying an instance of the subclass will return an instance of the wrong class. The copy method should instead say self class new. (The accessing method class is predefined.)

For the second example, consider the Smalltalk compiler. The compiler is represented by approximately 20 classes, including a class for each type of node in the parse tree. The compiler refers to these classes directly.

If you wish to modify the compiler, you must subclass the parts of the compiler you wish to change. Subclassing protects you from breaking the compiler in a way that makes it impossible to compile the code to fix the compiler.

For example, do you wish to store more information in the class LiteralNode? You might create the subclass TrulyLiteralNode to do so. However, all methods in other classes that must create a LiteralNode say LiteralNode new. If these methods instead said self class literalNodeClass new it would save you a lot of trouble, because as things stand now you are going to have to override all these methods. In the process, you will create almost as many new subclasses as there are methods to override. This is a lot of work and overhead for one *substantive* new class. It no longer seems simple to make changes to the system, which is one of the explicit goals of the inheritance and refinement mechanisms. Instead, the technique, which requires all this

unnecessary duplication, is time-consuming and, especially, error-prone.

The Smalltalk-80 user interface classes implicitly recognize the value of these conventions, and use the proposed technique at least partially. All views understand the message defaultControllerClass. Therefore, modifying a controller for a given view is easy — only one method in the view must change.

It must be admitted, however, that sometimes it is not practical to do other than to refer to global variables directly by name. The designer must therefore decide what is conceivable for a subclass to need to change. Such items should be parameterized. But the designer can assume that certain global variables are, indeed, functionally constants. These design decisions, if made with a sensitivity to future refinement, can break the otherwise endless chain of regression.

Temporary Variables

Earlier, we stated that all variables should be accessed using accessing protocol only, and argued the case for instance, class, pool, and global variables. But are these *all* the kinds of variables? What about temporary variables, that is, variables within methods?

The variables used within methods do not represent encapsulated state. They do not persist; they typically represent intermediate results, and vanish after the method has finished executing. They cannot be accessed from outside the method. Instead, they are a way of factoring code for readability and efficiency. For example, consider the method below, which uses temporary variables.

```
| width height |
width ← self origin x - self corner x.
height ← self origin y - self corner y.
↑(width squared + height squared) sqrt
```

Compare it with the following method, which uses no temporary variables.

```
↑((self origin x - self corner x) squared + (self origin y - self corner y) squared) sqrt
```

Clearly, the two methods are equivalent as far as the results are concerned, but the one using the temporary variables is easier to read.

Because temporary variables are not state variables, and because they cannot be accessed from outside a method, the conventions recommended above need not, and indeed *cannot*, apply to them.

Objections

Are there any disadvantages to using the proposed accessing conventions? Three objections can be raised to these conventions:

- they make applications less efficient,
- they make code less readable, and
- they depend on programmers respecting an essentially arbitrary convention: that of "private" methods.

Let's discuss these objections one at a time.

Efficiency

Sending a message to access a variable is less direct, and presumably less efficient, than simply referring to the variable by name. But the loss of execution efficiency must be balanced against the increased programmer productivity. By simplifying the process of refinement, programmers are able to create new programs and maintain existing ones with far less time and toil.

Furthermore, it is not inherently less efficient. One can imagine future implementations of Smalltalk in which accessing methods are as efficient as direct variable references.

Even so, it is always possible to eliminate performance bottlenecks. If analysis reveals that accessing variables in this way is slowing your application, you can use direct variable references instead. You then know you are optimizing your application for performance, and not for future refinement. Comment your code accordingly.

Readability

Another objection is that writing programs in this way contributes to syntactic clutter, making methods less concise and harder to read. These conventions may obscure the fact that you are using a variable to do something.

They do obscure that fact, which is part of the point. The principle of modularity says that the implementation of a class should not be important to clients of that class. The refinements of a class are also its clients [Snyd86]. Message-based state accessing hides the implementation of a class from its subclasses. Although code may be slightly harder to read until a programmer gets used to the conventions, they allow subclassing and refinement in a quicker and less error-prone manner. The benefits far outweigh this relatively minor disadvantage.

The Private Method Convention

It may also be objected that Smalltalk-80 doesn't have true private methods. Private methods are just a convention telling the programmer that these methods should not be sent from outside the class. Therefore the only way to really protect state in Smalltalk-80 is to have a variable with no accessing method. But the proposed conventions for the use of variables say that all variables should have accessing methods.

This is not a valid objection because even variables without accessors aren't really safe from external access. Methods defined in the class Object (instVarAt: and instVarAt: put:) allow any instance variable to be accessed or modified. It is already only a convention that a programmer normally shouldn't do this. This convention is no more or less binding than the "private" method convention. Nevertheless, future implementations of Smalltalk should probably support true private methods.

Eliminating Variables

This discussion has introduced a set of conventions. But the problem with all conventions is that programmers must learn and adhere to them. If the convention is intrinsic to the programming paradigm of a language, it should instead be formalized in the syntax and semantics of that language. In that way, the language supports you in doing things correctly.

One could certainly design a new language that follows these principles. For example, Dave Unger and Randy Smith have designed a language called Self which, among other things, has no variables at all, but uses accessing messages exclusively to access state [Unge87].

But it is not necessary to be quite so extreme. One can imagine a relatively minor extension to Smalltalk-80 that formalizes these stated conventions. All that is required is the syntactic mechanism to specify that a pair of message selectors, taken together, represent an instance variable. With such a mechanism, the variable name itself is no longer necessary.

Many such mechanisms are possible. One such syntax might be, for example:

```
Rectangle subclass: #CompactRectangle
   instanceStateMethods:  '(originY originY:) (extentY extentY:)'
```

Using this syntax, the message selectors originY and originY:, for example, represent the same piece of internal object state as that named by the variable originY. Using this syntax, the variable name itself becomes superfluous. This syntax also enables the creation of state method pairs that are not otherwise lexically related, such as getX and setX:.

Conclusion

Object-oriented programming languages, through their support of refinable and reusable classes, offer the potential of greatly increased programmer productivity. However, this potential can be realized only if users of these languages design for reusability. Direct references to variables severely limit the ability of programmers to refine existing classes. The programming conventions described here structure the use of variables to promote reusable designs. We encourage users of all object-oriented languages to follow these conventions. Additionally, we strongly urge designers of object-oriented languages to consider the effects of unrestricted variable references on reusability.

References

[Born87] Borning, Alan, and Tim O'Shea, "Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language," *ECOOP Proceedings*, 1987.

[Gold83] Goldberg, Adele, and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading Massachusetts, 1983.

[Meye87] Meyer, Bertrand, "Eiffel: Programming for Reusability and Extendibility," Technical Report TR-EI-3/GI, Interactive Software Engineering, Inc., Goleta, CA, January, 1987.

[Snyd86] Snyder, Alan, "Encapsulation and Inheritance in Object-Oriented Programming Langauges," *OOPSLA '86 Conference Proceedings*, pp. 38-45. Also published as *SIGPLAN Notices*, vol. 21 no. 11, November 1986, pp. 38-45.

[Unge87] Unger, David and Randall B. Smith, "Self: The Power of Simplicity," *OOPSLA '87 Conference Proceedings*, pp. 227-242. Also published as *SIGPLAN Notices*, vol. 22 no. 12, December 1987, pp. 227-242.