Direct Variable Access

You need to access Common State as readably as possible.

How do you get and set the value of an instance variable?

Accessing state is a topic much like initialization. There are two good answers. One is more readable. One is more flexible. Also like initialization, you will find dogmatic adherents of both approaches.

The simple, readable way to get and set the values of instance variables is to use the variables directly in all the methods that need their values. The alternative requires that you send a message every time you need to use or change an instance variable's value.

When I first started programming in Smalltalk, at Tektronix in the mid-80s, the debate about accessing state was hot. There were factions on both sides making self-important statements (don't you just love research labs?)

The Indirect Variable Access crowd has won the hearts and mind of the Smalltalking public, mostly I think because most of the high volume training companies teach indirect access. The issue is nowhere near as simple as "direct access bad, indirect access good".

I tried to reopen the debate in my Smalltalk Report column a few years back. A little brush fire started that eventually fizzled out. I was wondering if I was making too big a deal of the merits of direct access. Maybe I should have left well enough alone.

Then I spent several months working for a client that insisted on indirect access. A professional programmer can take on any of a number of styles at will, so I was a good soldier and wrote my getters and setters.

After I was done with that client, I wrote some code for myself. I was amazed at how much smoother it read than what I had been working on previously. The only difference in style was direct versus indirect access.

What I noticed was that every time I read:

```
...self x...
```

I paused for a moment to remind myself that the message "x" was just fetching an instance variable. When I read:

```
...x...
```

I just kept reading.

I told Ward of my experience. He had another good explanation. When you write classes that only have a handful of methods, adding a getting and a setting method can easily double the number of methods in your class. Twice as many methods to buy you flexibility that you may never use.

On the other hand, it's awful frustrating to get a class from someone and see the opportunity to quickly subclass it, only to discover that they used Direct Variable Access so there is no way to make the changes you want without changing most of the code in the superclass.

Access and set variables directly.

Indirect Variable Access

You need to access Common State as flexibly as possible.

How do you get and set an instance variable's value?

Now I have to display my true schizophrenia. Having convinced you in Direct Variable Access that just using variables is good enough, I'm going to ask you to ignore that crazy bastard and listen to me talk some sense here.

When you use Direct Variable Access, many methods make the assumption that a variable's value is valid. For some code this is a reasonable assumption. However, if you want to introduce Lazy Initialization, to stop storing and start computing the value, or if you want to change assumptions in a new subclass, you will be disappointed that the assumption of validity is wide spread.

The solution is to always use Getting Methods and Setting Methods to access variables. Thus, instead of:

```
Point>>+ aPoint
    ^x + aPoint x @ (y + aPoint y)
```

you would see:

```
Point>>+ aPoint
    ^self x + aPoint x @ (self y @ aPoint y)
```

If you browse the instance variable references of a class that uses Indirect Variable Access you will only see two references to each variable, the Getting Method and the Setting Method.

What you give up with Indirect Variable Access is simplicity and readability. You have to define all those Getting Methods and Setting Methods. Even if you have your system do it for you, you have that many more methods to manage and document. As explained above, code using Direct Variable Access reads smoothly, because you are not forever reminding yourself "Oh yeah, that's just a Getting Method."

Access and change an instance variable's value only through a Getting Method and Setting Method.

If you need to code for inheritance, use Indirect Variable Access. Future generations will thank you.

You will need to define a Getting Method and a Setting Method for each variable. For variables holding collections, consider implementing Collection Accessor Methods and an Enumeration Method.

Getting Method

You are using Lazy Initialization or Indirect Variable Access.

How do you provide access to an instance variable?

Once you have decided to use Indirect Variable Access, you are committed to providing a message-based protocol for getting and setting variable values. The only real questions are how you use it and what you call it.

Here's the real secret of writing good Getting Methods- ***make them private at first***. I cannot stress this enough. You will be fine if the same object invokes and implements the Getting Method. Another way of saying this is you always send messages invoking Getting Methods to "self".

Some people try to ensure this by prefixing "my" to the name of the method. Thus, instead of:

```
x
    ^x
```

you have:

```
myX
    ^x
```

This makes sure that code like:

```
self bounds origin myX
```

looks stupid. I don't feel this is necessary. I'd rather give programmers (myself included) the benefit of the doubt. If some other object absolutely has to send my private Getting Method, then they should be able to do so in as readable a manner as possible.

Provide a method that returns the value of the variable. Give it the same name as the variable.

There are cases where you will publish the existence of Getting Methods for use in the outside world. You should make a conscious decision to do this after considering all the alternatives. It is much preferable to give an object more responsibility rather than have it act like a data structure.

Setting Method

You are using Indirect Variable Access.

How you change the value of an instance variable?

Everything I said once about Getting Methods I'd like to say twice about Setting Methods. Setting Methods should be even more private. It is one thing for another object to tear out your state, it is quite another for it to bash in new state. The possibilities for the code in the two objects to get out of sync and break in confusing ways are legion.

Revisiting naming, I don't feel it is necessary to prepend "my" to the names of Setting Methods. It might provide a little more protection from unauthorized use, but I don't think the extra difficulty reading is worth it.

Provide a method with the same name as the variable that takes a single parameter, the value to be set.

Even if I use Indirect Variable Access, if I have a variable that is only set at instance creation time, I would not provide a Setting Method. I'd use the Creation Parameter Method to set the all the values at once. Once you have a Setting Method, though, you should use it for all changes to the variable.

Set boolean properties with a Boolean Property Setting Method.

Collection Accessor Method

You are using Indirect Variable Access.

How do you provide access to an instance variable that holds a collection?

The simplest solution is just to publish a Getting Method for the variable. That way, any client that wants can add to, delete from, iterate over, or otherwise use the collection.

The problem with this approach is that it opens up too much of the implementation of an object to the outside world. If the object decides to change the implementation, say by using a different kind of collection, the client code may well break.

The other problem with just offering up your private collections for public viewing is that it is hard to keep related state current when someone else is changing the collection without notifying you. Here's a department that use a Caching Instance Variable to speed access to its total salary:

```
Department
    superclass: Object
    instance variables: employees totalSalary
totalSalary
    totalSalary isNil ifTrue: [totalSalary := self computeTotalSalary].
    ^totalSalary
computeTotalSalary
    ^employees
        inject: 0
        into: [:sum :each | sum + each salary]
clearTotalSalary
    totalSalary := nil
```

What happens if client code deletes an employee without notifying the department?

```
...aDepartment employees remove: anEmployee...
```

The totalSalary cache never gets cleared. It now contains a number inconsistent with the value returned by computeTotalSalary.

The solution to this is not to let other objects have your collections. If you use Indirect Variable Access, make sure Getting Methods for collections are private. Instead, give clients restricted access to operations on the collection through messages that you implement. This gives you a chance to do whatever other processing you need to.

The downside of this approach is that you have to actually implement all of these methods. Giving access to a collection takes one method. You might need four or five methods to provide all the necessary protected access to the collection. In the long run, it's worth it though, because your code will read better and be easier to change.

Implement methods that are implement with Delegation to the collection. To name the methods, add the name of the collection (in singular form) to the collection messages.

If Department wanted to let others add and delete employees, it would implement Collection Accessor Methods:

```
addEmployee: anEmployee
    self clearTotalSalary.
    employees add: anEmployee
removeEmployee: anEmployee
    self clearTotalSalary.
    employees remove: anEmployee
```

Don't just blindly name a Collection Accessor Method after the collection message it delegates. See if you can find a word from the domain that makes more sense. For example, I prefer:

```
employs: anEmployee
    ^employees includes: anEmployee
```

to:

```
includesEmployee: anEmployee
    ^employees includes: anEmployee
```

Implement an Enumeration Method for safe and efficient general collection access.