A strategy for using instance variables

Bobby Woolf

I HAVE BEEN DEVELOPING a strategy for using instance variables that you might find helpful. This strategy provides guidance for several common programming tasks, such as properly initializing instance variables and providing accessors to use them. It shows how to implement equality methods and helps guide the initial decisions in making an object persistent. Finally, it explains why the instance variables in various application layers tend to behave differently.

Although this strategy probably isn't perfect, it is one that I find useful. The strategy doesn't consist of hard and fast rules you should always obey, just suggestions you should consider and trends you can look for. I can't guarantee that following these guidelines will make you a better programmer, but they should help.

TYPES OF INSTANCE VARIABLES

I've noticed that not all instance variables are created equal. Some seem to be more important than others. When using instances of a particular class, I notice that I'm constantly inspecting certain instance variables to make sure their values look reasonable, yet I consistently ignore other instance variables. So I've been trying to figure out how to distinguish the important ones from the unimportant ones.

In looking at how I use instance variables, I've found that there are three types, which I call identity, status, and cache. When looking at a new class, I try to distinguish these types to help figure out how the class works. When one of my own classes doesn't work well, I look at how I'm using these types; often I find inconsistencies; when I clean those up, the class works better. As I help other people develop their classes, I look for these types. If possible, I encourage the developers to identify each instance variable's type and use it "correctly."

I describe the three types in the following subsections.

Identity variables

Identity variables are how you distinguish two instances of a class. If both objects have the same identity values, they represent the same entity. Once an identity value is set, it usually doesn't change. After all, if you recognize an object because it has a certain identifier, and that identifier changes, how will you recognize it again next time? An object's identity values must be set for the objects' state to be valid. Also, there are usually no good default values for identity variables. Multiple objects with the same default values would be indistinguishable. Examples of typical identity variables include uniqueID, name, and a tree node's parent.

Status variables

When developers talk about instance variables—the variables that maintain an object's state and are accessed through getter and setter methods—they're usually talking about what I call "status variables." *Status variables* maintain an object's internal state and its relationships to other objects. These relationships may be aggregate or associative. Whereas identity values don't change, status values change constantly to reflect the object's changing state. Like identity variables, status variables must be set in order for the object's state to be valid; otherwise its internal state is undefined and inconsistent. If a status value is lost (set to an invalid value such as nil), the object's state cannot be recovered. Finally, status variables have suitable default values. (If nothing else, nil can be used as the default value, but that's often not a very good one. See my previous discussion on the Null Object pattern.[1]) Taken together, these default values describe the object's initial state. Examples of status variables include address, employer, and a tree node's children, as well as the various settings represented on a GUI using check boxes, radio buttons, etc.

Cache variables

Cache variables cache the results of expensive calculations. Their values are derived from the values of identity and status variables. When those values change, the

cache values must be recalculated. So cache values change as frequently as the values they are based on change. Cache values are optional; the object's state is still valid without them. If a cache value is lost, it can easily be recalculated. A cache variable's default value is usually *uncalculated*, a flag indicating that the value hasn't been calculated yet. The most common flag for uncalculated is nil, but there can be other such flags. For an example of a cache variable in VisualWorks, see CompositePart>>preferredBounds. A composite calculates its preferred bounds by merging those of its components; it caches the result for efficiency.

RAMIFICATIONS

These definitions are comforting, but they alone don't make your code any better. Yet you can improve your code by recognizing these types and writing your code accordingly.

Initialization

There are three approaches to initialize a variable:

1. Let a collaborator set its value explicitly.
2. Set its value to a default constant.
3. Set its value to the result of a calculation.

Each of these approaches is used to initialize a different type of instance variable:

1. *Identity initialization*—Initializes the identity variables.
2. *Creation initialization*—Initializes the status variables.
3. *Lazy initialization*—Initializes the cache variables.

Identity variables are initialized by the collaborator which creates the object. The collaborator should accomplish this via an instance creation method on the class side. Two examples of instance creation methods in VisualWorks—besides the standard ones like new, basicNew, and new:—are Point class>>x:y: and Dependent Partclass>>model:. An instance creation method on the class side should be implemented via a corresponding identity initialization method on the instance side. For example, Point class>>x:y: uses the identity initialization method Point>>setX:setY: to create the new instance:

```
Point class>>x: xInteger y: yInteger
    ^self basicNew setX: xInteger setY: yInteger

Point>>setX: xPoint setY: yPoint
    x := xPoint.
    y := yPoint
```

The instance creation methods in Circle and Interval are implemented the same way. I prefer to name this identity initialization method init..., so the name I would have used for Point>>setX:setY: would have been initX:y:. I put these methods in the "initialize-release" protocol.

Status variables should be initialized to their default values when the new instance is created. The standard name for the method that performs creation initialization is "initialize". VisualWorks has tons of examples of this, such as SortedCollection>>initialize. Another example is OrderedCollection>>setIndices; it isn't called "initialize" but it should be because it serves the same purpose.

Cache variables do not need to be initialized until they are used. In fact, initializing them is usually expensive and should be avoided until you know the values are needed. The easiest way to do this is to build lazy initialization into their accessors. VisualWorks doesn't use this technique much, but two examples are CompositePart>>preferredBounds and SliderView>>marker. You might implement Circle with radius as an identity variable and diameter and area as cache variables:

```
Circle>>radius
    ^radius

Circle>>diameter
    diameter isNil ifTrue: [self computeProperties].
    ^diameter

Circle>>area
    area isNil ifTrue: [self computeProperties].
    ^area

Circle>>computeProperties
    | r |
    diameter := radius * 2.
    r := self radius asLimitedPrecisionReal.
    area := r class pi * r * r
```

Developers often use lazy initialization with variables that are not caches, but I avoid this. Although caches are expensive to initialize, other variables usually aren't, so I see no compelling advantage in using lazy initialization on those other variables.

Often status variables are initialized in terms of identity variables, which means that an identity initialization method (in the form of initA:b:...z:) has to be run before the creation initialization method. Here's a hypothetical example of an instance creation method that will do this:

```
Example class>>x: newX y: newY
    ^(self basicNew initX: newX y: newY) initialize
```

HelpBrowser class>>on: is implemented this way because HelpBrowser>>initialize ends-up using the value of on:'s parameter.

Accessing

Developers often automatically create getter and setter methods for all of their instance variables and put them in a public protocol like "accessing." I prefer to be a little more selective and only create accessors for certain types of instance variables.

Identity variables need getters but no setters. The getters may be public or private. Setters are usually not necessary because the identity variables' values typically don't change. The only "setter" that is required is the identity initialization method (initA:b:...z:). Any setters you do

provide should definitely be private. Status variables use getters and setters in the conventional manner. These methods can be public or private.

Cache variables have getters but no setters. The getters, which can be public or private, contain lazy initialization. I prefer to implement the lazy initialization via a compute... method, as shown earlier in Circle>>computeProperties. If the calculations for one cache variable calculate others in the process, group the initialization for all of those variables together in one compute... method. Don't implement setters; they could be used to set the caches to values that are inconsistent with the object's state. Instead of setters, I implement flush... methods which reset the variables back to their uncalculated state (usually nil). If one change invalidates a number of caches, I flush them all in one method.

I see no compelling advantage in using lazy initialization.

For example, let's say that the Circle described earlier caches both diameter and area and that radius can change. Some more of the code would be

```
Circle>>radius: newRadius
    radius := newRadius.
    self flushProperties

Circle>>flushProperties
    diameter := nil.
    area := nil
```

The compute... and flush... methods are private ones. The cache getter methods with the lazy initialization send the compute... methods (see Circle>>diameter). The setter methods for the status (and identity) variables send the flush... methods (like Circle>>radius:). A particular setter does not need to flush all of the object's cache variables, only the ones that were calculated from it.

Equality versus identity

In my previous article, I talked about the difference between object identity and object equality. Object identity is very clear cut. If two variables contain identical objects, they are double-equal, which means that they both point to the same address in memory. Thus the two variables actually contain the same object.

Object equality is not so straightforward. If two variables' values are equal but not identical, they contain separate objects that are equivalent. The question is: What makes objects equivalent? In theory, they represent the same value. In practice, for Smalltalk, it means that a Set considers them to be duplicates.

I contend that two objects are duplicates if their identity variables are equal; their status and cache values are irrelevant. Because identity values rarely/never change, this means that two objects that are sometimes equal are always equal. Changes in their status don't affect their equalness. Thus if one object is a duplicate of another, it will be so through its entire lifetime, which is how it should be.

Just as implementors of equal (=) use identity variables, so do implementors of hash. If two objects are equal, their hash values need to be the same. So the same variables which are used for determining equality are also used for calculating hash values.

Persistence

When an object needs to store itself persistently, it shouldn't necessarily store all of its instance variable values the same way. Some instance variable types are persistent, others are not.

When storing an object in a relational database, its identity values belong in the database table's key columns. Just as identity variables should uniquely identify an object, a row's key column values should be unique from other rows. Status variables that represent state have simple values that are stored directly in table columns. Those maintaining relationships to other objects become database joins. There is generally no need to store cache values persistently. Rather than consume database space, just recalculate them after reading the object out of the database.

The storage issues for an object database are similar to those of a relational one. An object's identity values serve as its keys for retrieving it from the database. Status values are simply stored with the object. And cache values do not need to be stored at all, although they can be for completeness.

Database proxies also make use of instance variable types. A proxy must contain the identity values for its real object. That way it will be able to load the real object out of the database. Because a proxy is supposed to be lightweight, it shouldn't contain status or cache variables. Ideally, as much of the proxy's behavior as possible will be implemented just using the identity values. This will help maximize the amount of work the proxy can perform and minimize the number of real objects that need to be read from the database.

Dictionaries, Smalltalk objects that act somewhat like simple databases, also make use of instance variable types. Each element is stored in a Dictionary by a key that must be unique. That key is often an identity variable. That variable's value must not change while the element is stored in the Dictionary. Thus an identity variable makes a much better key than a status variable does.

Application layering

A Smalltalk program contains four main layers: view, application (mediator), domain, and infrastructure.[2] Most of the variables in application models and view objects are status variables. Identity variables are concentrated in domain objects. Infrastructure objects tend not to contain much state at all; they mostly point to domain

objects in some way (which can be an identity or status relationship).

Exceptions

These guidelines are not rules that are engraved in stone. Identity values *can* change during an object's lifetime. It's *sometimes* helpful for an instance creation method to initialize some status variables. A proxy *may* want to contain certain status values because they're used so often. However, I try to stick to these guidelines when possible. When I make an exception, I like to have a good reason.

Here are some interesting exceptions to these guidelines that I've found in VisualWorks.

Object identity is very clear cut. Object equality is not so straightforward.

Set's tally variable—Its behavior is a cache. If its value were ever lost, it could easily be recalculated. However, it's implemented as a status variable. That is because its value only changes by ±1 each time, a simple and well-defined transformation on the old value. For a large Set, it is much easier to add or subtract 1 than to flush the value and recalculate it from scratch.

Model's dependents variable—It's behavior is a typical status variable. However, when storing a Model persistently, this variable must be treated specially. Dependents are usually transient and thus are not stored when their parent is.

Point's x and y variables—Are these identity variables or status? Once a Point is created, can its x and y values change? Generally, changing their values is a bad idea, but there are plenty of examples where it works just fine. The same goes for the instance variables in Rectangle, Circle, Date, etc.

OBJECTIONS

As I discuss these ideas with other developers, I hear certain objections repeatedly. Here are some of them and my replies:

"Initialize is expensive"—Not if it's used properly. I use it to initialize status variables, ones which have readily available default values. If an implementor of initialize is expensive, it's probably doing more than just initialization. Which leads to…

"This status variable is expensive to initialize"—Then it's a cache variable. Cache variables require calculation to initialize; that's why they're lazy initialized. Status variables are initialized with simple default values that need no calculation.

"This status variable is hardly ever used"—Then get it out of that object! Every time you instantiate an instance of that class, you're sucking up memory for variables that probably won't be used. If there are a number of these variables, you're wasting a lot of memory. Refactor the class into two or more classes that separate the variables that are usually used from those that usually aren't. By the way, each of the pointers to these optional separate objects is a status variable, but it can be implemented as a cache.

"Lazy initialization is more efficient"—Not for identity and status variables. They're only initialized once. Why have the getters check every time to make sure they're initialized? They already have been. Lazy initialization is fine for cache variables because they get flushed periodically. But for identity and status variables, you always use them, so initialize them once and get it over with.

A WELL-DESIGNED OBJECT

Let's take a look at how you would use these guidelines to design a class. First of all, we assume that the class' implementation requires a number of instance variables.

- Some of their values are computed from the values of others. These are cache variables.
- Some are required as part of the object's state and have suitable default values. These are status variables.
- Some others are also required but do not have good default values. The object's collaborators must set these values when they create the object. These are identity variables.

Once you've established these designations for your variables, follow the other guidelines to help implement the class properly. The identity values should not change. They should be used in implementors of equals and hash and as database keys. The cache variables should have lazy getters as well as flush and compute methods. The status variables should be used to maintain the object's current state.

CONCLUSIONS

Here are the main points in this article:

- There are three types of instance variables: identity, status, and cache.
- Identity values don't change, status do, and cache are calculated from identity and status.
- Each type is initialized differently: identity initialization from collaborators, creation initialization, and lazy initialization.
- Identity variables are used for =, hash, and as dictionary and database keys.
- Status variables store an object's state and relationships to other objects.
- Cache variables require flush and compute methods.
- These are guidelines only; there are exceptions.

In my next article, I'll talk about how to display an object as a String. It turns out that identity variables are very helpful for doing this.

References

1. Woolf, B. "A Hierarchy that Acts Like a Class," *The Smalltalk Report* 5(4), Jan. 1996: 4–10.
2. Brown, K. "Remembrance of things past: Layered architectures in Smalltalk applications." *The Smalltalk Report* 4(9), July–Aug. 1995: 4–7.

Bobby Woolf is a Member of Technical Staff at Knowledge Systems Corp. in Cary, NC. He mentors Smalltalk developers in the use of VisualWorks, ENVY, and Design Patterns. Comments are welcome at woolf@acm.org, or at http://www.ksccary.com.