Visibility of Attributes and Methods

Stephen McHenry

An discussion of encapsulation in object systems.

This document describes the various types of attribute and method visibilities that are found in the object environment and discusses the advantages and disadvantages of using each.

Introduction to Visibility

What is Visibility?

Visibility simply refers to which other objects are allowed to see (and consequently, reference) an attribute or method that is contained in an object. The visibility of each attribute/method is defined at the class level, and each instance of that class (object) has the same visibility attached to the attributes/methods defined by that class.

Public

Public attributes/methods can be seen by all other classes. This means, for methods, that any other method of any other class can call the method which is defined as public. For attributes, this means that the attribute can be accessed from anywhere - from within the class that defines it, or any other class anywhere in the system.

Private

Private attributes/methods can only be seen within the class where they are defined. They cannot be seen from within any other class in the system, including subclasses. This means that any attributes/methods that are defined as private may only be used ny the class defining them.

Protected

Protected is a special visibility hybrid used with inheritance. Protected attributes/methods can be seen from within the class that defines them, and all subclasses of the pro-

tected class. If a class that defines protected attributes/methods has no subclasses, the effect is the same as if the attributes/methods had been defined as private.

Visibility for Methods

There are reasons to use each of the different types of visibility for methods, depending upon what you are trying to accomplish.

Public Methods

Public methods are used to provide the interface to a class for all other classes that need its services. Since these methods can be called by anyone in the system, there should be no assumptions about how the caller will use the methods. If events need to be posted prior to leaving (to indicate a state change in the object) this must be done prior to returning, since there is no guaranty that the caller will ever return to the object via another method call.

Public methods may be accessors (methods that return the value of an attribute) or mutators (methods that change the value of one or more attributes) or more complex functions that perform significant processing.

Protected Methods

Protected methods are used to provide visibility of a method to subclasses, but not to the "outside world". This is useful for (super)classes that implement an abstract interface and need to provide services to their subclasses. Like public methods, these services may be accessors (for attributes defined in the superclass) or mutators (to change attributes defined in the superclass) or more complex functions that perform significant processing, with the exception that they may only be invoked by subclasses of the class in which they are defined.

Private Methods

Private methods are used to define methods that may not be called outside of the class in which they are defined. These are quite useful for providing "atomic" functionality - doing one thing in one place. These atomic functions can then be called from other places in the object to provide more complex functionality needed by the public and protected methods that define the interface of the class. As long as all places in the object understand the rules for calling the private methods, there is no requirement that each private method leave the object in a consistent state before returning, since they cannot be called from outside of the object itself. (It must be clear, however, that the caller, at some level, is responsible for leaving it in a consistent state before exiting from the object altogether.)

Visibility for Attributes

In this section, the issue of visibility switches to attributes. Unfortunately, the issues having to do with attribute visibility are significantly less understood than those surrounding method visibility. Furthermore, many practitioners (particularly those new to the object paradigm and the benefits of good encapsulation) attempt to think of attribute

visibility in terms of the issues surrounding method visibility. Orthogonal to this is also the issue of performance. Unfortunately, all of this tends to get jumbled together, resulting in a significant lack of clarity followed by recommendations that don't make any sense with a deeper understanding of the issues. (It's a bit like saying "If you drive slow on snow and ice, you could skid and have an accident, so you'd better drive fast!")

Therefore, this section attempts to describe first what the various visibility issues are with respect to attributes, and where each mode is appropriate. Following this understanding, an attempt is made to consider these issues in the expanded area of "the real world".

Public Attributes

An attribute that is defined as *public* can be seen by any method of any class anywhere in the system. Unlike a public method, which must be called by other objects to do work, having a public attribute is exactly the same as having a globally accessible data structure in the days preceding the adoption of the object paradigm. In fact, if public attributes are allowed, then the whole idea of objects and methods can go away too. After all, we don't need methods defined on our objects if we can just access the attributes of the object directly! So, let's roll back the clock and get all the benefits of having globally accessible data structures that we had in the mid 1970s.

But, one of the biggest problems we had with the coding styles of the 1970s was exactly that - everything was accessible from everywhere. So, if a data structure was defined and several hundred places around a large system operated on the contents of that data structure, and then a change was made to that data structure, the programmer making the change had to locate and understand every last place that the data structure was used and make the corresponding modifications to that piece of code. The likelihood that every place would be found was, for some systems, near zero. This resulted in numerous undocumented features (sometimes known as *bugs*). In many cases, it wasn't even possible to track down where a data structure was being referenced, because the address was a (known) offset from some other data structure and the field that was used was also computed as an offset from that, leaving the programmer mystified as to why a particular bit of code had stopped working. Anyone who has not experienced this has not been programming very long.

Now, for a bit of history with respect to the object paradigm. One of the things that made the object paradigm attractive in the early days was the concept of *encapsulation*. This was the idea that the data for an object was wrapped up in a "bag" (object) with the code that operated on that data and no one outside of that bag (object) could operate on the data. Thus, when it came time to make a change, only code inside the bag needed to be inspected to see if it needed to be changed.

Now, let's consider - in detail - the ramifications of violating encapsulation.

Suppose there is a class in the system that stores a temperature (and perhaps other stuff). It stores this temperature in Fahrenheit. However, different parts of the system need to fetch the temperature in Fahrenheit and Centigrade (Celsius). So, there is a mutator that will accept a value, in Fahrenheit, and store it in a private attribute in the object. There are two accessors that fetch temperature, one that fetches the Fahrenheit value, and another that converts it to Centigrade prior to returning it.

```
class measurement
private
     integer: temp;
public
     method setFtemp (val: integer)
          temp = val;
          return;
     method getFtemp (): integer
          return temp;
     method getCtemp (): integer
          return (temp-32)*5/9;
end class;
```

Now, after running the system for a while, it was determined that there were many fetches of both the Fahrenheit and Centigrade temperature for each time that it was stored. So, the code was changed to add a second private attribute that contained the Centigrade temperature and the conversion was done when the attribute was stored, rather than when it was fetched. This saves the conversion for every fetch of the Centigrade value.

```
class measurement
private
     integer: ftemp;
     integer: ctemp;
public
     method setFtemp (val: integer)
          ftemp = val;
          ctemp = (val-32)*5/9;
          return;
     method getFtemp (): integer
          return ftemp;
     method getCtemp (): integer
          return ctemp;
end class;
```

Now suppose that some clever programmer decided that even higher performance could be achieved by making the ctemp and ftemp attributes public, thereby eliminating the accessor call. So, now the code looks like this:

```
class measurement
public
     integer: ftemp;
     integer: ctemp;
public
     method setFtemp (val: integer)
          ftemp = val;
          ctemp = (val-32)*5/9;
          return;
     method getFtemp (): integer
          return ftemp;
     method getCtemp (): integer
```

```
            return ctemp;
    end class;
```

On top of this, another person, not realizing that ftemp and ctemp really represent the same value, stores a new ftemp value without updating the ctemp value. (Note: this could not have happened if the attributes had been private and everyone were forced to use the mutator.) The system is now wrong. The defect must be located, the erroneous code must be corrected, and the system must be retested. All of this translates to additional time and money. Since we're always complaining that we never have enough of either, it is surprising that we would be so willing to do something that steals from what little we have.

One of the original attractions of the object paradigm to very complex systems was that it had the ability to change this. If ftemp and ctemp remained private attributes, there was no way they could ever become inconsistent.

Now, let's take that a step further. Suppose that, instead of just a temperature, that the two values that needed to be consistent were aircraft position, one in (latitude, longitude, altitude) and the other in (compass angle, elevation angle, distance). The issue is the same; the consequences are more dire.

Now, some people will argue that these situations can be prevented by education and rules governing usage. This author will not dispute that this is theoretically possible. However, we have had years to perfect the "education and rules" approach with little increase in the overall effect on our systems. Human nature being what it is, we all tend to rely on the expedient approach when we're behind schedule. And, as systems get more and more complex, that just doesn't cut it anymore.

Protected Attributes

Protected attributes are attributes that are only visible to subclasses of a superclass in which they are defined. Obviously, with protected attributes, the visibility is confined to a much more limited set of classes than public attributes. Furthermore, there are numerous occasions where a subclass needs access to the values of attributes from the superclass. So, (it is argued) protected attributes provide a way for subclasses to access the values in a superclass which they need to access without opening up encapsulation. Unfortunately, there is a lack of foresight in this argument.

At issue here is, again, how many places (different classes) need to be inspected and changed if the meaning or definition of an attribute changes. If a superclass has only a single subclass, then protected attributes only force the programmer to look in one additional place when making the modifications - twice the work, but not unmanageable. However, it could be argued that a superclass with only a single subclass is not the best modeled super/subclass relationship.

In most cases, superclasses are used to capture abstract interfaces to their corresponding subclasses. When used in this way, one of the measures of a "successful" superclass is how many subclasses it has. So, let's consider the case where a well formed superclass has 50 subclasses. If protected attributes are used in the superclass, then each subclass (51 places) must now be examined for potential impact. The damage is certainly less than with public attributes, but it is still there, nonetheless.

Private Attributes

The only visibility that any attribute of an object should have is private. It should never be visible outside of the class that defined the attribute.

All attributes that need to be fetched from outside of a class (even by subclasses) should be fetched by using accessors that return their values.

Lifecycle Considerations

Before discussing the pragmatic considerations, it is important to realize that the most tightly encapsulated that a system will be is on the day of its initial release. Pragmatically speaking, encapsulation is one of those things that seems to be controlled by a one way "ratchet" - you can open it up, but you can never close it down again. So, it is a naive developer/architect/manager that figures "We'll just make everything public for now and later we'll come back and tighten up the encapsulation." It just won't happen.

The reason for this is that there will be too many places to track down and change, for each attribute that is public. And, there is never time to do this, once a system has been deployed. Focus is always on getting to the next version, with the requisite enhancements. So, while you may choose to relax some of these rules for reasons to be discussed in a moment, recognize that this is decision you will live with forever.

Pragmatic Considerations

The preceding discussion focused only on the aspect of what is the best way, from an overall software engineering perspective, to engineer a system using visibility. This section adds a number of pragmatic considerations that have arisen over the years to challenge those techniques.

Performance

One of the oldest battles against tight encapsulation has been performance. Depending upon the environment, this can be an issue. In many others, it represents a lack of understanding of the benefits of encapsulation and the facilities available in the language to get both good performance and good encapsulation.

In C++, for example, good performance can be obtained from accessors by specifying them as "inline" functions. That is, instead of actually performing a procedure call to get the value, the code from the function is (behind the scenes) expanded in line, right where the accessor was called. Hence, no procedure call. Other OO languages have features which provide similar results (the keyword *final*, for example, in Java).

If the language you are using does not support some way to make the use of accessors fast, rather than arguing against accessors, the author's suggestion would be to exert pressure on the language vendor to add performance enhancements to the language to get the full benefits of the object paradigm.

(A special note about the Forte environment here: There are several handy facilities of the TOOL language that cannot be used if the attributes of an object are defined as private. These include 1) the ability to use SQL to select directly into the attributes of the object, and 2) the ability to map the attributes directly to fields on the screen. If these facilities are required, then there is little choice but to have them be public attributes. So, you may choose to have them as public attributes for these reasons. However, be aware that, once they are made public, there is little likelihood that they will ever become private again. So, use caution!)

Accessors Just Get In The Way

"I just want to get at the attributes. Accessors just get in my way!" - This person is a hacker (with all of the resulting attention to long term software engineering considerations that you get from this type of individual). If you want the type of system this person will build, you don't need this article.

Concluding Remarks

In the author's experience, no one who has ever used tight encapsulation to build a system, and enjoyed the benefits to be derived therefrom, would ever consider attempting to build another system without it. As systems evolve, changes must be made. It is a powerful capability to locate the place they must be made, and then not spend the next week (or six) dealing with the tidal waves that ripple throughout the system. Instead, the changes that are required are localized to a particular object. The next six weeks is spent not finding all of the other places impacted by this change, but going on to do more useful work. The author finds it interesting that virtually all of the people arguing why strong encapsulation isn't necessary are people who have never actually built a system using it. Those that have had it never want to go back. Just ask one of us.

Good encapsulation provides the following benefit: changes that used to produce tidal waves through your system that affect every last corner of it, now are little ripples that travel up to the nearest encapsulation boundary and stop. If this is a benefit you want from the object paradigm, the way to get it is good encapsulation.