On the Use and Misuse of Accessor Methods in OO Programming

Should we avoid or embrace the use of accessor methods in OO programming?

© Martin Hunter, June 2006

Executive Summary

Accessor methods have become a popular technique in Object Oriented programming. In this paper we discuss what accessor methods are and whether or not we should be using them. We conclude that accessor methods should, in general, be avoided. However, under specific conditions and constraints accessor methods may be accepted as a pragmatic solution.

What is an Accessor Method?

If you've been working in OO development for a while then you probably will have come across the term "accessor method" at some point. Accessor methods are publicly declared class level operations whose express purpose is to facilitate access to the privately declared (local) member variables within a class. Often, a class will declare two accessor methods for each local member variable declared (one for reading or *getting* the local variable value, and one for writing or *setting* the local variable value) though this is not conventionally required.

The Java fraternity refer to accessor methods colloquially as "getters" and "setters" because they are named according to a conventional form involving "get" and "set" as prefixes to the method names. For example:

Java
```
public class Television
{
        // local member variable
        private boolean isOn = false;

        // The getter for "isOn"
        public boolean getIsOn()
        {
                return isOn;
        }

        // The setter for "isOn"
        public void setIsOn(boolean on)
        {
                isOn = on;
        }
}
```

The Java accessor method naming convention originally came about due to a requirement of early J2EE specifications for bean classes to allow the J2EE container to gain access to local variables, and therefore to follow a public naming pattern that the J2EE container could understand. In more recent editions of Enterprise Java, mechanisms have been established to obtain the values of privately declared member variables without the need for getters or setters.

For some time, Microsoft have expressly defined accessor methods in their language meta models. Microsoft defined *get* and *set property procedures* as method types for their Visual Basic 6 programming language in the early 1990s. More recently many of the .NET languages formalise the idea of accessor methods in to *properties*, including C# as follows:

C#
```
public class Television
{
        private bool isOn = false;

        public IsOn
        {
                get { return isOn; }
                set { isOn = value; }
        }
}
```

Microsoft's formalisation of accessor methods as properties within the programming language model can be seen as a tidier and more mature solution to a problem that Java did not expressly address. However, for reasons that shall be explained, the incorporation of properties into the language model is far from helpful. In fact it flies directly in the face of a number of the core tenets of Object Orientation that such languages are said to support, and the use of accessor methods in OO programming results in an unnecessarily high degree of coupling between objects and potential for errors.

What's the problem with Accessor Methods?

To understand what problems lie around accessor methods, we must go back to basic OO principles. In particular, we must look at one of the foundation tenets: *Encapsulation*.

Encapsulation is about the scoping of responsibility. In particular, encapsulation states that an object is wholly responsible for its internal state (data) and behaviour (operations). Therefore, no other object should directly access or modify the state or behaviour of an object. To do so will:

1. Violate the ability of the object to function correctly, increasing the likelihood of error.

2. Increase the reliance of one object on another to function (increasing the coupling between objects).

3. Increase the reliance of the system at large on the *interaction* between two objects due to the distributed nature of logic.

Accessor methods break the encapsulation principle in that they facilitate external manipulation of the internal state of an object. Proponents of the use of accessor methods in OO development take pains to point out that accessor methods are just that: Methods. As such, it is argued that it is the object that is manipulating its own internal state when the accessor is called, not the caller of the accessor method.

Though this is technically correct, when we broaden our line of thinking to look at the intent behind the call, we see that encapsulation is still broken logically. When an accessor method is called, the caller is implicitly expecting the object to modify its internal state as a result. If the object did not behave in this way, either the caller would suffer because the object wouldn't be adhering to expected behaviour or the method wouldn't be an accessor. In calling an accessor method, therefore, the caller must understand something of the internal state of the object, the consequences that modifying state has for the object and the behaviour of the object's accessor methods. The internal implementation details of the object are therefore exposed. These factors break encapsulation because the object is not wholly responsible for its state or behaviour.

What should we do instead?

As mentioned earlier, the J2EE specification introduced accessor methods as a means for J2EE Containers to obtain the internal state of certain types of Java objects. The purpose of this was to allow the abstraction of the persistence of data within such objects to the Container, so that state of the objects could be persisted centrally and without the need for such objects to know how to persist themselves. In early editions of the J2EE specification, public accessor methods were required for the mechanism to work. Similar schemes exist for other managed object persistence environments, such as Object-Relational-Mapper (ORM) tools such as Hibernate.

The decision to use accessor methods should be a pragmatic one. If there is a significant advantage, such as object state persistence being managed centrally, then accessor methods can be accepted as a necessary evil. However, usage should be considered and constrained. Accessor methods should be avoided in general programming practice and should certainly not be used when an object's state needs to be modified for the purposes of implementing business logic. If you find yourself doing this, then I would suggest that, at best, there is a better way and, at worst, your classes, their interfaces and behaviour are poorly defined.

The Memento Pattern: a pattern for persisting state

One of the most effective OO design patterns for persistence of state without violating encapsulation is the Gang of Four's *memento* pattern (or *token* as it is sometimes known). The purpose of the pattern is to allow the capture and externalisation of an object's internal state so that it can be restored to that state later. A brief synopsis of this pattern follows[1].

Structure

The memento pattern is structured as follows:
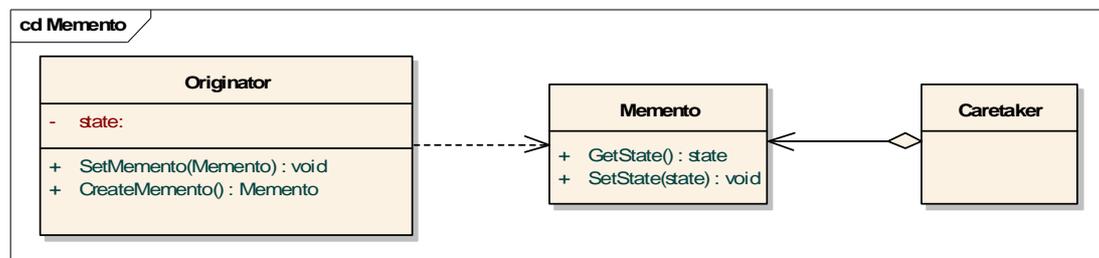
Participants

Instances of the **Originator** class are objects that require state to be persisted. When required, the originator puts its state into a **Memento**. The Memento is then passed to a **Caretaker** for safe keeping, allowing the Originator to be destroyed without loosing its state. At some later stage, a new Originator is created, and the Caretaker passes the stored Memento to it to reinitialise itself with the old state.

---

[1] Refer to "Design Patterns: Elements of Resuable Object-Oriented Software", Gamma et. al. 1995, Addison-Wesley, for further information.