Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Data Access Routines

Martin Fowler

One of the most important things about good design is modularity—dividing a system into separate pieces so that you can modify one module without the changes rippling all over the system. Early on, David Parnas observed that modules should be arranged around system secrets, each module hiding its secret from the other modules. Then if the secret thing changes, you avoid a ripple effect.

One of the most common secrets to hide these days is data structures. An axiom of object-oriented design is that data should always be private, but the idea of hiding data goes far beyond objects. Here I'm going to talk about guidelines for basic data hiding. My examples all use objects (I am after all an object bigot), but the arguments apply just as well to non-OO modules.

When thinking about data access routines, I find two major cases: encapsulating either a single value (such as a person's name) or a collection (such as the line items on an order). The two cases are a bit different, with the collection case being more complicated, so I'll start with a single value.

Accessing single values

The simplest data encapsulation approach that a language can take is to make data inaccessible from outside the module and to provide accessor methods that clients can use to get that data. So in Java, we find code such as in Figure 1. We can then manipulate our data with code such as

```
aPerson.setName("Martin Fowler");
```

The value of encapsulating something in this way appears when we want to add behavior to the accessor functions. Suppose we change our class so that it stores first and last names separately but still provides a single name string when required. We can do that with the code in Figure 2 while still preserving the interface in Figure 1. This ability to alter internal data structures without changing clients is the essence of encapsulating data. (Actually, the interface isn't quite preserved. The old code would accept names that didn't have two words, whereas the new one rejects them. I'll leave dealing with that as an exercise for you.)

Although this is a common approach, it's actually quite inconvenient to work with. It's much better for encapsulated data to look the same as public data, so that we can write code such as

```
aCustomer.name = "Martin Fowler".
```

We can't do this with some languages, but we can through others that use the notion of properties. So, for example, in C# we can write simple accessors such as in Figure 3. If we then need to add behavior, we can do so using code as in Figure 4. The clients still access the property in the same way (or almost—it does make a difference in .NET if you use reflection). The result feels much more natural, or at least it does to me.

Accessing a collection of values

While encapsulating single values is usually pretty well understood, that same understanding doesn't seem to quite make it to multiple values. There are a number of subtleties in-

0740-7459/03/$17.00 © 2003 IEEE

```
class Person {
  private String name;
  public String getName() {return name;}
  public void setName(String arg)
    {name = arg;}
}
```

Figure 1. Encapsulating a single-valued field with accessors (in Java).

```
class Person {

private String lastName;
private String firstName;
public String getName() {
  return firstName + " " + lastName;
}
public void setName(String arg) {
  String[] words = arg.split(" ");
  if (words.length != 2)
    throw new IllegalArgumentException
      ("name must have two words");
  firstName = words[0];
  lastName = words[1];
}
```

Figure 2. Altering the data structure in Figure 1 while (almost) preserving the interface (in Java).

```
class Person {
  public String Name;

  public static void Main() {
    Person martin = new Person();
    martin.Name = "Martin Fowler";
    System.Console.WriteLine(martin.Name);
  }
}
```

Figure 3. Encapsulating a single-valued field with a property (in C#).

```
class Person {
 public String Name {
  get {
   return _firstName + " " + _lastName;
  }
  set {
   String[] words = value.Split(' ');
   if (words.Length != 2)
    throw new Exception ("name can only
      have two words");
   _firstName = words[0];
   _lastName = words[1];
  }
 }
 private String _firstName;
 private String _lastName;
```

Figure 4. Altering the data structure in Figure 3 while (almost) preserving the interface (in C#).

```
class Album {
  private List tracks =new ArrayList();
  public List getTracks() {
    return tracks;
   }
}
```

Figure 5. A class that doesn't fully encapsulate its collection field (in Java).

```
class Album
 def initialize
  @tracks = []
 end
 def tracks
  return @tracks.clone
 end
end
```

Figure 6. Preserving collection encapsulation by copying (in Ruby).

volved, and language support tends to be rather less than that for single values.

The most common thing beginners miss is to use the same interface style for collections as for single values—and they fail to realize that this often breaks encapsulation. In the code in Figure 5, the tracks aren't fully encapsulated: clients can access the actual list of tracks and add or remove tracks to that list without the album knowing about it. When we encapsulate access to a collection, we usually want to know when someone adds or removes items, so we must avoid passing out the naked data structure.

I see three main ways to support the ability to read values while retaining encapsulation: copying, protection proxy, and using an iterator.

The simplest of the three is to pass out a copy of the underlying data structure, as in Figure 6. A protection proxy passes out a wrapper to the collection that prevents updates (Figure 7). Both of these techniques give the client a collec-

```
class Album {
 private List tracks = new ArrayList();
 public List getTracks() {
  return Collections.unmodifiableList
    (tracks);
 }
}
```

Figure 7. Using a protection proxy to encapsulate a collection (in Java).

```
class Album {
 private IList tracks = new ArrayList();
 public IEnumerator TrackEnumerator {
  get {return tracks.GetEnumerator();}
 }
```

Figure 8. Using an iterator to encapsulate a collection (in C#).

tion. With a copy, the client may change the copy, but this has no effect on the album's storage. With protection proxy, you get a runtime error if you try to change it. (C++'s `const` keyword gives you a similar capability.)

Often clients want to iterate through a collection and do something with the elements. So, another good tack is to provide an iterator (see Figure 8). The most common kind is an external iterator, with methods that let users advance it, test for the end of the collection, and access the current object.

Each of the three approaches has its pros and cons. I prefer protection proxy because it clearly signals what's going on. If I can't easily do that, I go with a copy. Most people are concerned about the performance of frequent copies, but in practice, that's a much rarer problem than people think (after all, you are only copying references, not the referred objects). The iterator only lets you iterate—it doesn't give you all the other methods that a collection can (`size`, `contains`, and so on). Although I've shown a C# iterator, I wouldn't choose that approach in C# because it prevents me from using the `foreach` statement.

Often you'll find that people don't protect the collection at all but rely on convention to avoid modifying the underlying collection. As I said earlier, this isn't really encapsulation and can lead to awkward bugs if people alter the collection directly. In some cases, however, the alternatives become too awkward to be worthwhile.

Don't worry about any calls made to the members once they are being iteraterated over. Enacapsulating a collection doesn't mean that you prevent changes to the members in the collection but that you prevent changes to the membership of the collection.

In contrast, the `modify` methods (usually an `add` and a `remove` method) are easy to support. You might also see an `addAll` method. Assignment is rare; supporting it is fine during object creation or if there's no behavior in the adds and removes. If there is behavior there, assignment becomes more complicated.

One sticky area for this kind of collection encapsulation is where you use a framework that expects direct access. The most common case of this is a GUI framework with data binding to a collection interface. Here you are faced with two unpleasant alternatives: either you retain the encapsulation and give up the convenient binding, or you use the binding. If you use the binding, you must break the owning class's encapsulation or use messy techniques such as collection-specific subclasses or events. It's difficult to make generalizations here because the trade-offs tend to vary with the GUI platform and tooling. I want to explore this further; if I do, I'll post my conclusions to http://martinfowler.com.

Self-encapsulation

A common issue is whether you should use accessors within a class. The rationale for doing so is that any extra behavior is used consistently within a class. It also can make matters easier if you subclass and want to override the accessor behavior. On the negative side, it can make the code more cluttered. I don't have a strong opinion either way.

Object construction

People often ask whether it's better to create an empty object and use setters to add its data, or to use a multiargument constructor. In general, I prefer to create fully formed objects, so I usually prefer the multiargument constructor. It's particularly handy with immutable data, which you can then clearly signal by not including a setter. Having said that, an empty-object approach can be better in some cases, so it's not an iron-clad rule.

I usually find that life is much easier if you initialize a collection field to an empty collection during object construction. That way, you don't have to deal with null checks in any of the later code.

Be wary of accessors

Finally, I must point out an important caveat: Only provide accessors if you really need to. Accessors often lead to code where one object pulls data out of another and then does something that the original owner should have done directly. A module that only has accessors is usually a bad thing. Modules are better if they really hide their data and don't provide accessors at all. Often this isn't possible, particularly in very data-oriented applications. But whenever you call an accessor, ask yourself if that doesn't suggest moving some behavior to the module with the data. A general rule of thumb is to question any situation where one object makes multiple calls to another's accessors.

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. Contact him at fowler@acm.org.