ESSENTIAL SKILLS FOR THE AGILE DEVELOPER

A Guide to Better Programming and Design

ALAN SHALLOWAY
SCOTT BAIN
KEN PUGH
AMIR KOLSKY

NetObjectives
Lean-Agile Series

Praise for *Essential Skills for the Agile Developer*

"I tell teams that the lean and agile practices should be treated like a buffet: Don't try and take everything, or it will make you ill—try the things that make sense for your project. In this book the authors have succinctly described the 'why' and the 'how' of some of the most effective practices, enabling all software engineers to write quality code for short iterations in an efficient manner."

—Kay Johnson
Software Development Effectiveness Consultant, IBM

"Successful agile development requires much more than simply mastering a computer language. It requires a deeper understanding of agile development methodologies and best practices. *Essential Skills for the Agile Developer* provides the perfect foundation for not only learning but truly understanding the methods and motivations behind agile development."

—R.L. Bogetti
www.RLBogetti.com,
Lead System Designer, Baxter Healthcare

"*Essential Skills for the Agile Developer* is an excellent resource filled with practical coding examples that demonstrate key agile practices."

—Dave Hendricksen
Software Architect, Thomson Reuters

CHAPTER 5

Encapsulate That!

"Encapsulation" is a word that's been with us in software development for a long time, but if you asked people what it means, many would say something like "hiding data." In fact, there are quite a few books and websites that would use that as the definition of the word. However, we have found that an examination of the true meaning of encapsulation can be enormously beneficial and can make many other aspects of object-oriented (OO) design (design patterns, for instance) easier to understand and to use.

We begin simply, by showing encapsulation in its most obvious and straightforward forms, and then expand these concepts into the patterns and all the qualities of code that make it fundamentally easier to maintain, debug, enhance, and scale. What you will see is that encapsulation, given its more useful definition, is a fundamental, first principle of OO.

Unencapsulated Code: The Sabotage of the Global Variable

The following is the lowest degree of encapsulation possible in a pure-OO language like Java or C#.

```
public class Foo {
  public static int x;
}
```

Any class in the system can access `x`, either to use its value in a calculation or other process (and thus become dependent upon it) or to alter its value (and thus cause a side effect in those classes that depend on it). `Foo.x` might as well be thought of as `Global.x` (and in fact there are many developers who have done just this), and in one fell swoop

the efforts of the Java and C# creators to prevent global variables are thwarted.

Global variables are ineffective because they create tight coupling. They are rather like a doorknob that everyone in the household touches and, thus, during the cold and flu season becomes the vector for sharing germs. If any class in the system can depend on `Foo.x` and if any other class can change it, then in theory every class is potentially coupled to every other class, and unless your memory is perfect, you're likely to forget some of these links when maintaining the code. The errors that creep in over time will often be devilishly difficult to find. We'd like to prevent things that carry such obvious potential for pain.

What most OO developers would naturally think of as the lowest degree of encapsulation is the following:

```
public class Foo{
  public int x;
}
```

That `x` in this case is an instance variable is, in fact, a kind of encapsulation. Now, for any class in the system to depend on `x`, it must have a reference to an instance of `Foo`, and for two classes to be coupled to each other through `x`, they must both have a reference to the *same* instance of `Foo`.

A number of techniques can prevent this from happening, so whereas a public static variable cannot be encapsulated, here we have at least a chance of preventing unintended side effects. There are weakly typed languages that posit this level of encapsulation to be enough in and of itself.

Also note that `Foo` is a public class. Another encapsulating action would be to remove the `public` keyword, which would mean that only classes in the same package (Java) or assembly (C#) would be able to access `Foo` in any way at all.

Encapsulation of Member Identity

Although putting `x` into an instance does create some degree of encapsulation, it fails to create an encapsulation of identity.

Identity is the principle of existence. Identity coupling is usually thought of in terms of class A "knowing" that class B exists (usually by having a member of its type, taking a parameter of its type, or returning its type from a method), but instance variables have identity, too.

The following puts it another way:

```
public class Foo {
  public int x;
}

public class Bar {
   private Foo myFoo = new Foo();
   public int process(){
    int intialValue = myFoo.x;
    return initialValue * 4;
  }
}
```

Ignoring that this particular implementation of `Bar`'s `process()` method would consistently produce a zero, note that not only is `Bar` coupled to the value of `x` in the instance pointed to by `myFoo`, but it is also coupled to the fact that `x` is an integer (or, at minimum, a type that can be implicitly cast to one) and that it is an instance member of the `Foo` class. It is coupled to `x`'s nature.

If a future revision of `Foo` requires that `x` be stored as, for instance, a String, that it be obtained from a database or remote connection dynamically at runtime, or that it be calculated from other values whenever it is asked for, then `Bar`'s method of accessing it will have to change. `Bar` will have to change because `Foo` changed (and so will every other class that accesses `x` in a `Foo` instance). This is unnecessary coupling.

Encapsulating the identity of `x` requires that we create a method or methods that encapsulate `x`'s nature.

```
public class Foo {
  private int x;
  public int getX() {
    return x;
  }
  public void setX(int anInt){
    x = anInt;
  }
}

public class Bar {
  private Foo mFoo = new Foo();
  public int process(){
    int intialValue = myFoo.getX();
    return initialValue * 4;
  }
}
```

The new way of accessing Foo's x in Bar (highlighted in bold) now creates an encapsulation of x's identity, or nature. Bar is coupled only to the fact that getX() in Foo takes no parameters and returns an integer, not that x is actually stored as a integer, or that it's actually stored in Foo, or that it's stored anywhere at all (it could be a random number).

Now the developers are free to change Foo without affecting Bar, or any other class that calls getX(), so long as they don't change the method signature.

```
public class Foo {
  public String x = "0"; // x no longer an int
  public int getX() {
    // convert when needed
    return Integer.parseInt(x);
  }
  public void setX(int anInt){
    // convert back
    x = new Integer(anInt).toString();
  }
}

public class Bar {
  private Foo mFoo = new Foo();
  public int process(){
    // none the wiser
    int intialValue = myFoo.getX();
    return initialValue * 4;
  }
}
```

Here x is now stored as a String (though this is just one of any number of changes that could be made to the way x is maintained in Foo), but Bar need not be touched at all.

Why?

What you hide, you can change. The fact that x was an integer in Foo was hidden, so it could be changed. Envisioning this over and over again throughout a system leads to the conclusion that the power to make changes, to make extensions, and to fix bugs is made much easier when you encapsulate as much and as often as possible.

Self-Encapsulating Members

Although many developers might find it quite natural to encapsulate a data member behind a set of accessor methods (another word for get()

and `set()` methods), the standard practice for accessing a data member from within the class itself is generally to refer to it directly.

```
public class Foo{
  private int x;
  public int getX(){
    return x;
  }
  public void setX(int anInt){
    x = anInt;
  }
  public boolean isPrime(){
    boolean rval = true;
    for(int i=2; i<(x/2); i++){
      if(Math.mod(i, x) == 0)
        rval = false;
    }
      return rval;
    }
}
```

Here, `isPrime()` calculates a true/false condition on `x`, which is local to `Foo()`, so even though `x` is private, it can be accessed directly in the method.

For the most part it's a matter of convenience, but consider the earlier scenario where `Foo` was changed to the effect that `x` is no longer stored as an int or is no longer stored as a local data member at all (perhaps it's stored in a database, serialized to the disk, obtained from some other class, or calculated from other values). Now `isPrime()`, and any other local method that directly refers to `x`, will have to be rewritten to account for the new situation. In fact, the new code in local methods that converts whatever `x` has become into the integer it used to be will likely be highly redundant. And we know we don't want redundancy.

However, for the most part it's a matter of convenience, but if the possibility of `x` changing in this way seems likely (what Bruce Eckel calls "the anticipated vector of change"), then using `getX()` even within `Foo`'s own methods would reduce the maintenance headaches considerably when the change is made.

You have to weigh the syntactic inconvenience of writing `getX()` instead of `x` in these algorithms against the need to make extensive changes when and if the nature of `x` needs to change. Generally, it's found to be worth the extra typing.

Preventing Changes

Another advantage of the accessor methods shown earlier is the capability to make a member of a class "read-only." If we simply remove the `setX()` method in `Foo` earlier, then the value is readable but not changeable. This eliminates the potential that `Foo` may serve to couple two other classes, since although one class may depend on the return value of `getX()`, no other class may change this value.

Alternately, we could eliminate the `getX()` method but leave `setX()`, meaning that another class could change the state of a `Foo` instance, but none could depend on it.

.NET has instituted an alternative way of accomplishing this, called a property:

```
// C#
public class Foo {
  private int myX;
  public int x {
    get { return myX; }
    // 'value' is an implicit variable
    set { myX = value; }
  }
}
```

This is a bit of syntactic sugar that allows the programmer to embed the `get`s and `set`s as part of the data-member definition. `Bar`, however, would still access it like it would any public variable.

```
// C#
public class Bar {
  private Foo mFoo = new Foo();
  public int process(){
    // actually calls the get
    int intialValue = myFoo.x;
    return initialValue * 4;
  }
}
```

The theory here is that `x` could have begun its life as a public member and then later could be changed to a property without changing `Bar`. In `Foo`, you could eliminate the `set{}` code like before to make the property read-only (or the `get{}` to make it write-only) or could write them to fetch/send/calculate `x` and thus gain similar beneficial encapsulation as was possible with the `getX()` and `setX()` methods.