Encapsulation as a First Principle of Object-Oriented Design

by Scott L. Bain
Senior Consultant
Net Objectives

Introduction: First Principles

Object Orientation (OO) addresses as it's primary concern those things which influence the rate of success for the developer or team of developers: how easy is it to understand and implement a design, how extensible (and understandable) an existing code set is, how much pain one has to go through to find and fix a bug, add a new feature, change an existing feature, and so forth. Beyond simple "buzzword compliance", most end users and stakeholders are not concerned with whether or not a system is designed in an OO language or using good OO techniques. They are concerned with the end result of the process – it is the development team that enjoys the direct benefits that come from using OO.

This should not surprise us, since OO is rooted in those best-practice principles that arose from the wise dons of procedural programming. The three pillars of "good code", namely *strong cohesion*, *loose coupling*, and the *elimination of redundancies*, were not discovered by the inventors of OO, but were rather inherited by them (no pun intended).

But presaging even fundamental principles like these is the "First Principle" underlying the OO paradigm. A First Principle is that which implies (or leads one to) other principles.

If one can discover the First Principle that underlies a given set of principles, and fully understand all its implication, then entire hierarchies of wisdom can be retained in one cognitive place. It's a bit like a fractal seed that explodes into a complex and beautiful pattern through the simple amplification of its nature.

An example of this from the non-technical world is "The Golden Rule". Parents discover that it is often difficult to teach their children all the detailed rules of good citizenship, but if one can get a child to embrace and practice this First Principle –

treat others as you would like to be treated – then most of the legal and moral standards we are held to will follow in due course. Most legal systems are based on The Golden Rule.

One way to understand OO well (beyond the superficial syntactic understanding that is often mistaken for real understanding) is to discover and explore the First Principles that inform OO at every level.

The First Principle we are concerned with here is:

What you hide you can change.

OO theorists use the term "encapsulation" to represent this principle, and while every OO analyst, designer, and developer believes (s)he knows what encapsulation means, we often fail to investigate it fully enough to empower it as a First Principle. There is much more to Encapsulation than simply making data members private ("data hiding"), and yet this is often how it is understood.

The purpose of this article is to investigate encapsulation more fully, to determine its value as a First Principle, and to discover the ways it which it implies, or leads to other OO principles and concepts. Along the way we'll find that a rigorous understanding of encapsulation, in and of itself, can be the key to grasping the wisdom of OO in general, and that concepts like polymorphism, abstracting concepts, and designing to interfaces are actually all implied by the search for comprehensive encapsulation.

Un-Encapsulated Code: The Evil of the Global Variable

Here 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 dependant 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 `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 is thwarted.

Global variables are evil 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 this:

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

But the fact 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`. There are a number of techniques to 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 like Python and Ruby 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

While 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 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. Put 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 the fact that this particular implementation of `Bar`'s `process()` method would consistently produce a zero, note that `Bar` is not only 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 the fact 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`, or 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.

To encapsulate 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, nor that it's actually stored in `Foo`, nor that it's stored anywhere at all (it could be a random number). Now the developers are free to change `Foo` without effecting `Bar`, nor 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() {
                return Integer.parseInt(x);     // convert when needed
        }
        public void setX(int anInt){
                x = new Integer(anInt).toString(); // convert back
        }
}


public class Bar {
        private Foo mFoo = new Foo();
        public int process(){
                int intialValue = myFoo.getX(); // none the wiser
                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, extensions, and to fix bugs is made much easier when one encapsulates as much and as often possible.

Self-Encapsulating Members

While 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()`, and so even though `x` is private, it can be accessed directly in the method.

However, 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, or serialized to the disk, or 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 re-written 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.

One has 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 above is the ability to make a member of a class 'read only'. If we simply remove the `setX()` method in `Foo` above, then the value is readable, but not changeable. This eliminates the potential that `Foo` may serve to couple two other classes, since while one class may depend of the return value of `getX()`, no other class may change this value. Alternately, we could eliminate the `getX()` method, but leave the `setX()`, meaning that another other class could change the state of a `Foo` instance, but none could depend on it.

C# has instituted an alternative way of accomplishing this, called a property:

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

This is a bit of syntactic sugar that allows the programmer to embed the gets and sets as part of the data-member definition. Bar, however, would still access it as it would any public variable:

```
// C#
public class Bar {
        private Foo mFoo = new Foo();
        public int process(){
                int intialValue = myFoo.x; // actually calls the get
                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`, one could eliminate the `set{}` code as 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.

There are arguments to be made for and against this. We'll not engage them here, but the good news is that, in C#, you can use either technique easily. Note that self-encapsulating a data-member is a simpler issue in C#, because the syntax for referencing the member locally would not have to be changed when the member became a property.

The difficulty of Encapsulating Reference Objects

One common practice in OO is the use of constructors to guarantee the proper creation of compositional objects. Consider this code:

```
public Foo{
        private int x;
        Foo (int anInt){    // Constructor requires an int be passed in
                x = anInt;  // ensuring proper instantiation of Foo.
        }
        public int getX() {
                return x;
        }
        public void setX(int newInt) {
                x = newInt;
        }
}

public Bar{
        private Foo myFoo;
        public Bar(Foo aFoo){// Constructor requires an instance of Foo
                myFoo = aFoo;// be passed in, ensuring the proper
        }                    // instantiation of Bar.
        public int process(){
                int intialValue = myFoo.getX();
                return initialValue * 4;
        }
}

public class Client{
        public static void main(String[] args){
                int x = 5;          // Make needed int for Foo instance
                Foo f = new Foo(x); // Made Foo instance using x
                Bar b = new Bar(f); // Make Bar instance using f
                int i = b.process();// Use Bar instance
        }
}
```

Here `Client` creates an instance of `Foo` (initializing the value of `x` by passing an `int` into the constructor), and then hands this instance to the `Bar` constructor, which the instance of `Bar` will now hold as a private member.

Is `myFoo` in `Bar` encapsulated? It's private. It has no `set()` method to allow changes to it (it does not even have a `get()` method). Isn't the concept of a private member with no accesors the very definition of member encapsulation? The assumption most developers would make is that `myFoo` is fully encapsulated, and this could be a disastrous assumption.

Consider this version of the `Client` class:

```
public class Client{
        public static void main(String[] args){
                int x = 5;
                Foo f = new Foo(x);
                Bar b = new Bar(f);
                int i = b.process();
                f.setX(10);         // Still holding the Foo reference!
                i = b.process();
        }
}
```

`b.process()` will produce an entirely different value in the second call, because `Client` still holds a reference to `f` (the instance of `Foo` it created to hand over to

Bar's constructor), and thus can still change its state. Martin Fowler calls this effect 'aliasing'. Since Bar's behavior depends upon the state of this Foo reference, Client can break the encapsulation if it retains this Foo reference for its own purposes, and Bar cannot prevent it from doing so. If Client passes this Foo reference to another class, then it too will be able to break the encapsulation of myFoo in Bar.

This is because myFoo is a reference to an object on the heap. When Client calls new Bar(f), f is indeed passed by copy, but it's a copy of a reference and it therefore points to the same instance that the original did, and so the copying neither hides nor protects it from future manipulation.

Contrast this to the call to new Foo(x). Since x is a value, not a reference, the copying of x completely removes any possibility that Client can change it by changing the original. In this code fragment:

```
int x = 5;
Foo f = new Foo(x);
x = 10;
```

...the code 'x = 10' will have no effect on the state of f, because the integer it holds is a different integer than the one Client has. It's a copy.

So, this problem of altering-after-the-fact is unique to references (at least in Java), and can be tricky to deal with. One way of solving the problem is by having Bar make a defensive copy of the Foo reference it takes in its constructor, assuring that it holds a different reference of Foo (with the same state) than any other object holds. This requires that Foo be cloneable, or that it expose its state so that Bar can make another Foo with the same state. Let's examine two versions of Bar's constructor that could ensure good encapsulation of its myFoo member:

```
public Bar(Foo aFoo){
        myFoo = new Foo(aFoo.getX());
}
```

This will work as the code is written because aFoo allows Bar, an outside class, to access its x member, and so Bar can make a new, private, separate Foo for its own use. Now Client can manipulate the original Foo it all it wants, and will not effect Bar.

If Foo did not allow this access (if it had no getX() method), then Foo could be made cloneable:

```
public class Foo{
        private int x;
        public Foo(int anInt){
                x = anInt;
        }
        public Foo clone(){
                return new Foo(x);
        }
```

```
}
```

...and so Bar's constructor could make its defensive copy this way:

```
public Bar(Foo aFoo){
        myFoo = aFoo.clone();
}
```

Either way, `Bar`'s `myFoo` reference will point to a different object on the heap than any other class in the system, and thus `myFoo` is once again completely encapsulated.

Breaking Encapsulation with Get()

Making members private and then providing `get()` methods but no `set()` methods is often thought to be strong encapsulation.

We've seen above how this is not the case with reference objects. However, if we take the step of making a defensive copy of any reference held by a particular class, then can we still say that `set()` methods break encapsulation but `get()` methods do not? A `set()` allows an outside class to make a change, but a `get()` does not, right?

Not with references. Consider:

```
public Bar{
        private Foo myFoo;
        public Bar(Foo aFoo){
                myFoo = aFoo.clone();  // Make a private instance
        }
        public Foo getFoo(){
                return myFoo;
        }
        public int process(){
                int intialValue = myFoo.getX();
                return initialValue * 4;
        }
}

public class Client{
        public static void main(String[] args){
                int x = 5;
                Foo f = new Foo(x);
                Bar b = new Bar(f);   // Bar will make a defensive copy
                int i = b.process();      // Use Bar instance
                Foo fFromBar = b.getFoo();// Client gets Bars new Foo.
                fFromBar.setX(10);        // ...and changes it
                i = b.process();          // effecting Bar again.
        }
}
```

Here `Client` makes a `Foo`, and `Bar` clones it to defend against subsequent manipulation by `Client`. However, `Client` is able to get `Bar`'s newly-made, private `Foo` reference by calling the `getFoo()` method provided, and so encapsulation is broken again. The only way around this is to have `Bar` clone `myFoo` again, and then return this object from its `getFoo()` method:

```
public Bar{
        private Foo myFoo;
        public Bar(Foo aFoo){
                myFoo = aFoo.clone(); // Make a private instance
        }
        public Foo getFoo(){
                return myFoo.clone(); // Return a clone, not the member
        }
        public int process(){
                int intialValue = myFoo.getX();
                return initialValue * 4;
        }
}
```

If a `setFoo()` method is provided, it would have to work like the constructor does, to ensure the encapsulation is maintained.

So, the game is completely different when dealing with value objects (which are themselves passed by copy) and reference objects (which are references passed by copy).

Encapsulation of Value and Reference Types

Activity	Value Object Encap?	Reference Object Encap?
State passed into constructor	Yes	No, unless a defensive copy is made.
Get() method provided	Yes	No, unless a defensive copy is returned.
Set() method provided	No	No, unless a defensive copy is made.

C# makes this issue both simpler and more complex.

For instance, while the only value objects in Java are the primitives (`int`, `float`, `boolean`, and the like), in C# it is possible to create value objects with both complex state and functional members (methods), called structs. Structs work very much like classes do (you `can` instantiate them, they can implement interfaces), but they are passed by making complete copies of the object, not by copying a reference to an instance, and so these particular encapsulation issues can be largely alleviated.

However, C# also makes value objects passable by reference, using `ref` and `out` keywords in method signatures, and so it's possible to break encapsulation on any member, value or reference, if they are used. Therefore, `ref` and `out` should be used very carefully.

A pragmatic point: with all these issues, the main danger comes when one breaks encapsulation unknowingly. There are many approaches to design that might take advantage of the fact that a reference passed into a constructor could subsequently be used to change state on a contained object, or that a value object in C# could be externalized through a `ref` or an `out` parameter. If this is intentional, and well-documented, then the danger is minimal.

However, these subtle issues can easily escape notice, and create situations where members seem to be well-encapsulated, and yet are not. Understanding how and why this happens is the best defense.

Encapsulation of Type

Encapsulation is often thought of as data-hiding. There are even references that define it as precisely that. However, encapsulation is a broader notion – as we've already seen, it's meaningful to think of encapsulating the identity of a data member, not just the value that it holds.

Taken further, it's possible to think of the hiding of entire types as encapsulation as well. For example:

```
public abstract class Calculator{
        public abstract int calc(int x, int y);
}


public class Adder extends Calculator{
        public int calc(int x, int y) {
                return x + y;
        }
}

public class Multiplier extends Calculator{
        public int calc(int x, int y){
                return x * y;
        }
}
```

Here, `Adder` and `Multiplier` extend the abstract base class `Calculator`. Because of this, any instance of `Adder` or `Multiplier` can be held by a reference of `Calculator` type (this is called an *implicit cast*), and the class that holds the reference need not "know" what is "really" being held. For instance:

```
public class CalcUser{
        private Calculator myCalculator;
        public CalcUser(Calculator aCalculator){
                myCalculator = aCalculator;
        }
        public void process(){
                int i1 = 4;
                int i2 = 5;
                int r = myCalculator.calc(i1, i2);
        }
}
```

Note that `CalcUser` contains no mention whatsoever of `Adder` or `Multiplier`, yet `Calculator` itself is abstract (cannot be instantiated), so whatever instance is passed into the constructor will have to be either an instance of `Adder` or an instance of `Multiplier` (these are the only classes that can be cast to `Calculator`, so the type-checking in the compiler will only allow these instances to be passed in).
Since they share a common interface, `CalcUser` can use either `Adder` or `Multiplier` instances in exactly the same way (this is an example of Polymorphism) without any knowledge of which subclass it has, or even what subclasses are possible, or even that `Calculator` is abstract in the first place.

This is the encapsulation of type. `Calculator`, an abstract base class (although this is equally true if an interface is used) encapsulates its subclasses if other classes hold their references to them as an upcast to the base type. Many/most of the popular design patterns make use of this fact, and it is just what the "Gang of Four" had in mind when they made the recommendation that good designers should "design to interfaces".

It's a tad more complicated in C# because, unlike Java, methods in C# are not automatically virtual (late-bound). What this means is that a method called on a subclass reference that was cast back to the superclass type may revert to the superclass method unless the override keyword is used in the subclass method:

```
public abstract class Calculator{
        public abstract int calc(int x, int y); // abstract methods are
        }                                       // inherently virtual
}

public class Adder : Calculator{
        public override int calc(int x, int y) {
        return x + y;
}

public class Multiplier : Calculator{
        public override int calc(int x, int y){
                return x * y;
        }
}
```

Note that `calc()` is abstract in `Calculator`, and so it is inherently/automatically virtual. Sometimes base classes provide default implementations of methods, and if these are present they must be declared virtual, or they may not be overridden like this:

```
public abstract class Calculator{
        public virtual int calc(int x, int y){  //default implemetation
                return System.Math.Max(x, y);   //must be 'virtual'
        }                                       //to be overridden
}
```

Encapsulation of Design

In the previous example, the Calculator abstraction made it possible to hide the specific kinds of calculator subclasses that exist from other parts of the application. The virtue of this is maintainability and flexibility – we can add new types of calculators to the system without changing the client objects that use them.

However, whereas we can hide the specific classes Adder and Multiplier from most of the rest of the system, certainly we cannot hide them entirely. Something, somewhere, will have to contain the code `new Adder()` and `new Multiplier()`, in order for these classes to be instantiated. And, something will have to make the decision as to which one to build under a given circumstance. If the client objects that use these classes are given this responsibility, then we must break the encapsulation of type, and we lose the modularity that seemed so attractive.

The obvious answer is to use another object to build the `Calculator` subclasses. Such an object is usually called an "object factory":

```
public class CalculatorFactory {
        public Calculator getCalculator() {
                if(someDecision()) {
                        return new Adder();
                } else {
                        return new Multiplier();
                }
        }
}
```

If all other objects that require a `Calculator` implementation use this factory to get it, then we've got one single place to maintain when a new class, say `Subtractor`, comes into being.

In Design Patterns Explained, Elements of Reusable Object-Oriented Software, the authors[1] illustrate the encapsulation of type in many of their patterns, such as the Strategy Pattern:

```
public class Client {
        public static void main(String[] args) {
                Context c = new context();
                Strategy s = StrategyFactory.getStrategy();
                c.takeAction(s);
        }
}

public class Context {
        public void takeAction(Strategy myStrategy) {
                // Whatever else
                myStrategy.varyingAction();
                // Whatever else
        }
}

pubic abstract class Strategy {
        public abstract void varyingAction;
}
```

`StrategyFactory` (not shown) clearly makes an implementing subclass (also not shown) of `Strategy`, and provides this to `Client`. `Client` then hands this to the `Context` object to use. `Context` is "unaware" of which actual `Strategy` implementation it has, and so is `Client`, since it obtained it from the `StrategyFactory` blindly.

But there is still an opportunity for encapsulation here that we're not taking advantage of. The `Strategy` implementations are hidden from everything except `StrategyFactory`, but `Client` does have "awareness" of the fact that the `Context` object requires a `Strategy` implementation to be "handed in" in order to function properly. What is not encapsulated, therefore, is the Strategy Pattern itself, the

[1] Gamma, Helms, Johnson, and Vlissides

design we're using to handle the variation. If at some point in the future we decide to change this design, to one where no such delegation takes place, or where the delegation is accomplished in some other way, then we'll have to change the way `Client` interacts with `Context`. The design is not encapsulated.

Sometimes this is necessary. If, for instance, we need a high degree of dynamism in the way `Context` operates (perhaps every time it is used we need to be able to give it a different `Strategy` implementation) then this implementation of the pattern would be appropriate. Where it is not necessary, however, it's better to hide the design.

How? The most common technique it to either:

a) Have the `Context` object request the `Strategy` implementation from `StrategyFactory`, instead of having `Client` do so. Thereby, `Client` can simply call the method on `Context` without handing anything in.
b) Have a factory build the `Context` object in the first place, and while doing so hand in the proper `Strategy` object via its constructor.

…or both. Let a factory establish the initial `Strategy` to use, but let the `Context` object (or the `Client`) change this when/as needed.

Either way, the `Client` object would now have no coupling to the fact that the Strategy Pattern was in use, which would make it much easier to change this design when/if the issues involved became more complex, and this design was no longer adequate.

Encapsulation on all Levels

We should strive to encapsulate everything that can be encapsulated, because it simplifies maintenance every time we do so. Also, it is always easy to break encapsulation after the fact, and extremely difficult to encapsulate something later on. When something is encapsulated, and a need arises that requires the hidden thing to be revealed, we simply provide access. When something was not encapsulated, and now needs to be hidden, then every part of the application that has become coupled to it must be re-worked.

To achieve maximum encapsulation, one must:

1. Encapsulate by policy, reveal by need. When in doubt, hide it, then reveal it when this becomes necessary.
2. Broaden your view of what can be hidden. Using an object factory, for instance, encapsulates the construction issue, and can even encapsulate an entire design, if all the players in the solution are build by the factory. We don't normally think of this as "encapsulation", but it is, and brings the same value that other types of encapsulation bring: ease of change later on.

Conclusions

Much of the power of OO comes from the circumspect use of its features. Extensible and maintainable designs are more likely when one takes optimal advantage of delegation, abstraction, polymorphism, and when one hides everything one can.
Encapsulation, properly understood, is a First Principle that can lead us in the right direction if we constantly ask ourselves:

- Should I hide **this**?
- Is **this** really hidden?
- What entities is **this** hidden from?
- How could encapsulation be broken **here**?
- When should encapsulation be broken **here**?
- What prepares **this** best for possible future changes?

The point of all of this is to make development easier, less frustrating, more sustainable, more economically viable, and ultimately more successful.

-Scott Bain-
April 2004