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 design (design patterns, for instance) easier to understand and to use.

We'll 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 we will see is that encapsulation, given its more useful definition, is a fundamental, first principle of OO.

Un-Encapsulated Code: The Sabotage 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 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 is 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 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 on 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.

.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; }
            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 contained 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. We call 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).

Put another way, an entity that appears to be strongly encapsulated, but which contains references that are not encapsulated, really isn't as encapsulated as it appears. We always need to ask ourselves how changes from outside can affect a given class, and see encapsulation as a protection against those changes.

Encapsulation of Value and Reference Types

Activity	Value Object Encapsulation?	Reference Object Encapsulation?
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.

Even when done correctly, getters and setters break encapsulation. So far we've talked about how getters and setters can inadvertently break encapsulation. But there is a subtle way they violate encapsulation even when used properly. This is that they expose that the concept is part of the the containing class. In other words, if there is a getX in Foo, even if it does not expose how 'x' is implemented in Foo, entities will become coupled to the fact that Foo has a concept called 'x' (and how to use). Better to hide the concept entirely if possible.

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 Object-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 object 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 object 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 implementation
            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 {
```

[1] TODO : GOF REF HERE

```
        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 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:

- 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.
- 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 than 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:

- Encapsulate by policy, reveal by need. When in doubt, hide it, then reveal it when this becomes necessary.
- 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.

Encapsulate That!

We are, among other things, consultants and coaches at Net Objectives. Very often, our role is to help people find a good solution to a problem they are having, and often that solution will be one of the many well-known design patterns that have been documented in our industry.

In other words, sometimes it is a matter of *selecting* a good pattern to use, at least as a starting point, to reveal a natural and powerful solution to the problem before us.

What we noticed, over time, was that the interaction between the coach and the customer had a certain repetition to it. It would often go something like this:

- Customer: I have problem so and so, and I'm not sure what to do.
- Coach: Describe the problem, and I'll see if I can help.
- Customer: (long description)
- Coach: Can you boil that down a bit? What's the aspect of this that really is causing you the problem?
- Customer: (shorter description, still hard to see)
- Coach: Focus your description a little more. Try to get to the one big thing that you're concerned about.
- Customer: (one or two sentences that describe the major impediment)
- Coach: Encapsulate that!

Of course, the conversation never literally goes this way, but in one form or another, as a coach, we always seem to be suggesting that something be encapsulated.

How does this help?

Many, many things can be encapsulated, and in each case, the technique used to encapsulate a given thing is almost always a design pattern. In our simple example above about the various calculators, the "thing being encapsulated" is the calculation algorithm. The Strategy Pattern does that, it encapsulates a single varying algorithm.

The other patterns encapsulate things like:

- Sequences
- Relationships
- Multiple varying algorithms
- Dependent varying algorithms
- Creation of complex objects
- Structures of Objects

…and so on. So, if you can figure out what your problem is, where your biggest risks and concerns are, you might not be able to solve them, but you may be able to limit their impact by encapsulating them.

See: http://www.netobjectivesrepository.com/PatternsByEncapsulation for a cross-index at our Design Patterns Repository, listing patterns by what they encapsulate.

Encapsulate That in Practice

Seeking to encapsulate is something that many developers have learned from their experiences, in different ways and under different circumstances. Here is one such experience, as related by Al Shalloway:

I have learned that asking good questions is often a very good design technique. One of my favorite questions when I am designing code is:

"If I knew that however I was going to figure this out now was going to need to change in an unexpected way in the future, how would I design it?

This is a subtle question. I am somewhat acting as if no matter what I do will be wrong. This is not pessimistic as much as it is realistic. I am what I call ***pre-cognitive impaired.*** I don't do pre-cognition. ☺ However, I can take advantage of experience. While I don't know how my requirements are going to change (this would require pre-cognition) I do know that they will change (this is experience). I also know, from experience, that if I design for something to occur, it may not. If I added things to my design that are now not needed, I have added complexity to my code for likely no gain.

I am asking the question of how do we design when we know that we don't yet know all we need to make a good design? My answer would be to encapsulate the concept that we have uncertainty about. This follows our general mantra of encapsulating by policy, reveal by need. Let's see an example of how this works.

I was talking to a client once who was working on a system that had several processors in it. Each of these processors often had events that other processors needed to know about. The question was – how should they communicate with each other? The need for a Messaging class was clear – but how should it be implemented? First in line for consideration was a Messenger class that used TCP/IP. There were other candidates as well. The concern was that performance was going to be critical but at the start of the project, without complete software to measure, it was difficult to tell what the performance problem really was going to be?

I was being asked to solve a problem which required knowledge that wasn't available now. I, of course, could guess, or use my experience, but in fact, my experience told me that this was an approach that hadn't led to a lot of success for me. Stated another way, my experience was that whatever I decided upon now was going to be discovered to be wrong in the future.

The approach to take, of course, is to encapsulate the Messenger class. That is, build a class to do messaging knowing we are going to have to rewrite it after we discover how we should have built it. Now this may seem like a lot of waste, but it actually isn't. If we take the attitude that we are going to re-write this messaging object, it's probable that we can find a really simple one to write at first. It's also possible we can mock the object. If our messaging is merely used to tell other objects on different

processors things, and we are not getting messages back, then a write-only mock works quite well and is trivial to write.

In any event, I would suggest that a real message object would not be that difficult to write. Bear in mind that while we are working on our code and haven't delivered it yet, we can take several short-cuts as long as we have prepared for them. For example, we might write a TCP/IP message handler without full error handling. We can throw exceptions when an error occurs, intending to write the error handling later (if we, indeed, keep TCP/IP at all).

Chapter Summary

In writing their ground-breaking book on Patterns, the "Gang of Four" had several pieces of general design advice for us. Among these was a phrase that is sometimes dismissed as "old thinking" because of the way it is worded:

"Consider what should be variable in your design and encapsulate the concept that varies."

Perhaps because of the word "should", this can be interpreted as a "big design up-front" point of view; indeed, it likely was written to mean that, given that the book was written in 1994 at a time when this was the predominant way of thinking about software design.

However, we have learned to see design as a constant process, and we do not expect to be able to anticipate every design element we will need before we begin to create a product. In other words, we don't trust our ability to know, consistently and reliably, what "should" be variable. So, the advice we would give is very similar, with just a small change of focus:

"Consider what is now, or about to become, varying, and encapsulate it conceptually."

In the past it was seen as dangerous to wait on such decisions, but if we follow good practices such as "programming by intention", "encapsulating construction", and if we define our tests early, we find that the risks of waiting are largely removed, and we can work in a more responsive, reality-based way.

That said, it is good to discover what you can, especially mission-critical issues, as early as you can. This also can help us to avoid/remove the risk of deferring commitment to a specific design. One practice that can help you here is "Commonality-Variability Analysis", which is the subject of our next chapter.

Business-Driven Software Development™

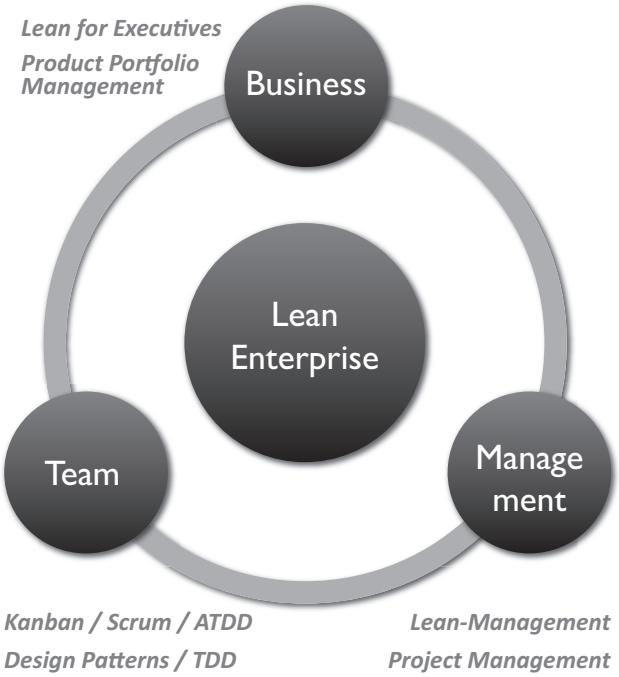
Assessments, Consulting, Training, and Coaching

Business-Driven Software Development (BDSD) is Net Objectives' proprietary integration of Lean-Thinking with Agile methods across the business, management and development teams to maximize the value delivered from a software development organization. BDSD has built a reputation and track record of delivering higher quality products faster and with lower cost than other methods

BDSD goes beyond the first generation of Agile methods such as Scrum and XP by viewing the entire value stream of development. Lean-Thinking enables product portfolio management, release planning and critical metrics to create a top-down vision while still promoting a bottom-up implementation.

BDSD integrates business, management and teams. Popular Agile methods, such as Scrum, tend to isolate teams from the business side and seem to have forgotten management's role altogether. These are critical aspects of all successful organizations. In BDSD:

- **Business** provides the vision and direction; properly selecting, sizing and prioritizing those products and enhancements that will maximize your investment
- **Teams** self-organize and do the work; consistently delivering value quickly while reducing the risk of developing what is not needed
- **Management** bridges the two; providing the right environment for successful development by creating an organizational structure that removes impediments to the production of value. This increases productivity, lowers cost and improves quality



Become a Lean-Agile Enterprise

All levels of your organization will experience impacts and require change management. We help prepare executive, mid-management and the front-line with the competencies required to successfully change the culture to a Lean-Agile enterprise.

Prioritization is only half the problem. Learn how to both prioritize and size your initiatives to enable your teams to implement them quickly.

Learn to come from business need not just system capability. There is a disconnect between the business side and development side in many organizations. Learn how BDSD can bridge this gap by providing the practices for managing the flow of work.

Why Net Objectives

While many organizations are having success with Agile methods, many more are not. Much of this is due to organizations either starting in the wrong place (e.g., the team when that is not the main problem) or using the wrong method (e.g., Scrum, just because it is popular). Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban, Scrumban) and integrates business, management and teams. This lets us help you select the right method for you.

Assessments

See where you are, where you want to go, and how to get there.

Business and Management Training

Lean Software Development

Product Portfolio Management

Enterprise Release Planning

Productive Lean-Agile Team Training

Team training in Kanban, Scrum

Technical Training in ATDD, TDD, Design Patterns

Roles Training

Lean-Agile Project Manager

Product Owner

For more information about Business Driven Software Development Services
Contact Mike Shalloway at Net Objectives: mike.shalloway@netobjectives.com / 404-593-8375
NetObjectives.com / 1-888-LEAN-244 (888-532-6244)