JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/jw-05-2001/jw-0518-encapsulation.html

Encapsulation is not information hiding

The principles of information hiding go beyond the Java language facility for encapsulation

By Wm. Paul Rogers, JavaWorld.com, 05/18/01

Words are slippery. Like Humpty Dumpty proclaimed in Lewis Carroll's *Through the Looking Glass*, "When I use a word, it means just what I choose it to mean -- neither more nor less." Certainly the common usage of the words *encapsulation* and *information hiding* seems to follow that logic. Authors rarely distinguish between the two and often directly claim they are the same.

Does that make it so? Not for me. Were it simply a matter of words, I wouldn't write another word on the matter. But there are two distinct concepts behind these terms, concepts engendered separately and best understood separately.

Encapsulation refers to the bundling of data with the methods that operate on that data. Often that definition is misconstrued to mean that the data is somehow hidden. In Java, you can have encapsulated data that is not hidden at all.

However, hiding data is not the full extent of information hiding. David Parnas first introduced the concept of information hiding around 1972. He argued that the primary criteria for system modularization should concern the hiding of critical design decisions. He stressed hiding "difficult design decisions or design decisions which are likely to change." Hiding information in that manner isolates clients from requiring intimate knowledge of the design to use a module, and from the effects of changing those decisions.

In this article, I explore the distinction between encapsulation and information hiding through the development of example code. The discussion shows how Java facilitates encapsulation and investigates the negative ramifications of encapsulation without data hiding. The examples also show how to improve class design through the principle of information hiding.

Position class

With a growing awareness of the wireless Internet's vast potential, many pundits expect location-based services to provide opportunity for the first wireless killer app. For this article's sample code, I've chosen a class representing the geographical location of a point on the earth's surface. As a domain entity, the class, named `Position`, represents Global Position System (GPS) information. A first cut at the class looks as simple as:

```
public class Position
{
```

```
  public double latitude;
  public double longitude;
}
```

The class contains two data items: GPS `latitude` and `longitude`. At present, `Position` is nothing more than a small bag of data. Nonetheless, `Position` is a class, and `Position` objects may be instantiated using the class. To utilize those objects, class `PositionUtility` contains methods for calculating the distance and heading -- that is, direction -- between specified `Position` objects:

```
public class PositionUtility
{
  public static double distance( Position position1, Position position2 )
  {
    // Calculate and return the distance between the specified positions.
  }
  public static double heading( Position position1, Position position2 )
  {
    // Calculate and return the heading from position1 to position2.
  }
}
```

I omit the actual implementation code for the distance and heading calculations.

The following code represents a typical use of `Position` and `PositionUtility`:

```
// Create a Position representing my house
Position myHouse = new Position();
myHouse.latitude =    36.538611;
myHouse.longitude = -121.797500;
// Create a Position representing a local coffee shop
Position coffeeShop = new Position();
coffeeShop.latitude =    36.539722;
coffeeShop.longitude = -121.907222;
// Use a PositionUtility to calculate distance and heading from my house
// to the local coffee shop.
double distance = PositionUtility.distance( myHouse, coffeeShop );
double heading  = PositionUtility.heading(  myHouse, coffeeShop );
// Print results
System.out.println
  ( "From my house at (" +
    myHouse.latitude + ", " + myHouse.longitude +
    ") to the coffee shop at (" +
    coffeeShop.latitude + ", " + coffeeShop.longitude +
    ") is a distance of " + distance +
    " at a heading of " + heading + " degrees."
  );
```

The code generates the output below, which indicates that the coffee shop is due west (270.8 degrees) of my house at a distance of 6.09. Later discussion addresses the lack of distance units.

```
==========================================================================
From my house at (36.538611, -121.7975) to the coffee shop at
(36.539722, -121.907222) is distance of 6.0873776351893385 at a
heading of 270.7547022304523 degrees.
==========================================================================
```

`Position`, `PositionUtility`, and their code usage are a bit disquieting and certainly not very object-oriented. But how can that be? Java is an object-oriented language, and the code uses objects!

Though the code may use Java objects, it does so in a manner reminiscent of a by-gone era: utility functions operating on data structures. Welcome to 1972! As President Nixon huddled over secret tape recordings, computer professionals coding in the procedural language Fortran excitedly used the new International Mathematics and Statistics Library (IMSL) in just this manner. Code repositories such as IMSL were replete with functions for numerical calculations. Users passed data to these functions in long parameter lists, which at times included not only the input but also the output data structures. (IMSL has continued to evolve over the years, and a version is now available to Java developers.)

In the current design, `Position` is a simple data structure and `PositionUtility` is an IMSL-style repository of library functions that operates on `Position` data. As the example above shows, modern object-oriented languages don't necessarily preclude the use of antiquated, procedural techniques.

Bundling data and methods

The code can be easily improved. For starters, why place data and the functions that operate on that data in separate modules? Java classes allow bundling data and methods together:

```
public class Position
{
  public double distance( Position position )
  {
    // Calculate and return the distance from this object to the specified
    // position.
  }
  public double heading( Position position )
  {
    // Calculate and return the heading from this object to the specified
    // position.
  }
  public double latitude;
  public double longitude;
}
```

Putting the position data items and the implementation code for calculating distance and heading in the same class obviates the need for a separate `PositionUtility` class. Now `Position` begins to resemble a true object-oriented class. The following code uses this new version that bundles the data and methods together:

```
Position myHouse = new Position();
myHouse.latitude =    36.538611;
myHouse.longitude = -121.797500;
Position coffeeShop = new Position();
coffeeShop.latitude =    36.539722;
coffeeShop.longitude = -121.907222;
double distance = myHouse.distance( coffeeShop );
double heading = myHouse.heading( coffeeShop );
System.out.println
  ( "From my house at (" +
    myHouse.latitude + ", " + myHouse.longitude +
    ") to the coffee shop at (" +
    coffeeShop.latitude + ", " + coffeeShop.longitude +
    ") is a distance of " + distance +
    " at a heading of " + heading + " degrees."
  );
```

The output is identical as before, and more importantly, the above code seems more natural. The previous version passed two `Position` objects to a function in a separate utility class to calculate distance and heading. In that code, calculating the heading with the method call `util.heading( myHouse, coffeeShop )` didn't clearly indicate the calculation's direction. A developer must remember that the utility function calculates the heading from the first parameter to the second.

In comparison, the above code uses the statement `myHouse.heading(coffeeShop)` to calculate the same heading. The call's semantics clearly indicate that the direction proceeds from my house to the coffee shop. Converting the two-argument function `heading(Position, Position)` to a one-argument function `position.heading(Position)` is known as *currying* the function. Currying effectively specializes the function on its first argument, resulting in clearer semantics.

Placing the methods utilizing `Position` class data in the `Position` class itself makes currying the functions `distance` and `heading` possible. Changing the call structure of the functions in this way is a significant advantage over procedural languages. Class `Position` now represents an abstract data type that encapsulates data and the algorithms that operate on that data. As a user-defined type, `Position` objects are also first class citizens that enjoy all the benefits of the Java language type system.

The language facility that bundles data with the operations that perform on that data is encapsulation. Note that encapsulation guarantees neither data protection nor information hiding. Nor does encapsulation ensure a cohesive class design. To achieve those quality design attributes requires techniques beyond the encapsulation provided by the language. As currently implemented, class `Position` doesn't contain superfluous or nonrelated data and methods, but `Position` does expose both `latitude` and `longitude` in raw form. That allows any client of class `Position` to directly change either internal data item without any intervention by `Position`. Clearly, encapsulation is not enough.

Defensive programming

To further investigate the ramifications of exposing internal data items, suppose I decide to add a bit of defensive programming to `Position` by restricting the latitude and longitude to ranges specified by GPS. Latitude falls in the range [-90, 90] and longitude in the range (-180, 180]. The exposure of the data items `latitude` and `longitude` in `Position`'s current implementation renders this defensive programming impossible.

Making attributes latitude and longitude `private` data members of class `Position` and adding simple accessor and mutator methods, also commonly called getters and setters, provides a simple remedy to exposing raw data items. In the example code below, the setter methods appropriately screen the internal values of `latitude` and `longitude`. Rather than throw an exception, I specify performing modulo arithmetic on input values to keep the internal values within specified ranges. For

example, attempting to set the latitude to 181.0 results in an internal setting of -179.0 for `latitude`.

The following code adds getter and setter methods for accessing the private data members `latitude` and `longitude`:

```
public class Position
{
  public Position( double latitude, double longitude )
  {
    setLatitude( latitude );
    setLongitude( longitude );
  }
  public void setLatitude( double latitude )
  {
    // Ensure -90 <= latitude <= 90 using modulo arithmetic.
    // Code not shown.
    // Then set instance variable.
    this.latitude = latitude;
  }
  public void setLongitude( double longitude )
  {
    // Ensure -180 < longitude <= 180 using modulo arithmetic.
    // Code not shown.
    // Then set instance variable.
    this.longitude = longitude;
  }
  public double getLatitude()
  {
    return latitude;
  }
  public double getLongitude()
  {
    return longitude;
  }
  public double distance( Position position )
  {
    // Calculate and return the distance from this object to the specified
    // position.
    // Code not shown.
  }
  public double heading( Position position )
  {
    // Calculate and return the heading from this object to the specified
    // position.
  }
  private double latitude;
  private double longitude;
}
```

Using the above version of `Position` requires only minor changes. As a first change, since the above code specifies a constructor that takes two `double` arguments, the default constructor is no longer available. The following example uses the new constructor, as well as the new getter methods. The output remains the same as in the first example.

```
Position myHouse = new Position( 36.538611, -121.797500 );
Position coffeeShop = new Position( 36.539722, -121.907222 );

double distance = myHouse.distance( coffeeShop );
double heading = myHouse.heading( coffeeShop );

System.out.println
  ( "From my house at (" +
    myHouse.getLatitude() + ", " + myHouse.getLongitude() +
    ") to the coffee shop at (" +
    coffeeShop.getLatitude() + ", " + coffeeShop.getLongitude() +
    ") is a distance of " + distance +
    " at a heading of " + heading + " degrees."
  );
```

Choosing to restrict the acceptable values of `latitude` and `longitude` through setter methods is strictly a design decision. Encapsulation does not play a role. That is, encapsulation, as manifested in the Java language, does not guarantee protection of internal data. As a developer, you are free to expose the internals of your class. Nevertheless, you should restrict access and modification of internal data items through the use of getter and setter methods.

Isolating potential change

Protecting internal data is only one of many concerns driving design decisions on top of language encapsulation. Isolation to change is another. Modifying the internal structure of a class should not, if at all possible, affect client classes.

For example, I previously noted that the distance calculation in class `Position` did not indicate units. To be useful, the reported distance of 6.09 from my house to the coffee shop clearly needs a unit of measure. I may know the direction to take, but I don't know whether to walk 6.09 meters, drive 6.09 miles, or fly 6.09 thousand kilometers.

Analysis of the modifications necessary to add units to class `Position` reveals another possible design flaw: the geometry to use has not been specified. You can calculate the distance and travel between any two points on the Earth's surface in several ways. Depending on the need for accuracy, distances within 50 kilometers could use the local Cartesian coordinates of plane geometry. These simple calculations may suffice for locations within the same area, but calculating the distance and traveling from San Francisco to Paris requires more difficult geometric calculations along a great circle path. And conducting earthquake studies by measuring distance and direction between precise locations along the San Andreas Fault in California may require the use of hyper accurate elliptical geometry, even for locations within 10 kilometers of each other.

Rather than directly include all of these domain choices in `Position`, I add reference variables of type `Units` and `Geometry`. The objects referenced by these variables handle the details concerning units and geometry. I don't show the actual code for the definition of interfaces `Units` and `Geometry` nor any concrete classes implementing these interfaces. Suffice it to say there are four implementations for units:

- `Kilometers`
- `NauticalMiles`
- `StatueMiles`
- `Radians`

and three implementations for geometry:

- `PlaneGeometry`
- `SphericalGeometry`
- `ElipticalGeometry`

Appropriate `Position` getters and setters allow the dynamic change of either the units or the geometry used for distance and heading calculations.

Now for the most important and potentially damaging discovery: While creating the various geometry classes, it is determined that to delegate work to these objects, class `Position` should not maintain data items `latitude` and `longitude`, but should instead represent the internal location using spherical coordinate angles `theta` and `phi`. I won't digress into the reason. Regardless of why such a change is necessary, the change itself can prove painful or impossible if `Position`'s clients directly access internal data. Before examining the ramifications of these changes, let's look at the updated version of `Position` that features the new reference variables:

```
public class Position
{
  public Position( double latitude, double longitude )
  {
    setLatitude( latitude );
    setLongitude( longitude );
    // Default to plane geometry and kilometers
    geometry = new PlaneGeometry();
    units = new Kilometers();
  }
  public void setLatitude( double latitude )
  {
    setPhi( Math.toRadians( latitude ) );
  }
  public void setLongitude( double longitude )
  {
    setTheta( Math.toRadians( longitude ) );
  }
  public void setPhi( double phi )
  {
    // Ensure -pi/2 <= phi <= pi/2 using modulo arithmetic.
    // Code not shown.
    this.phi = phi;
  }
  public void setTheta( double theta )
  {
    // Ensure -pi < theta <= pi using modulo arithmetic.
    // Code not shown.
    this.theta = theta;
  }
  // Setters for geometry and units not shown
  public double getLatitude()
  {
    return( Math.toDegrees( phi ) );
  }
  public double getLongitude()
```

```
  {
    return( Math.toDegrees( theta ) );
  }
  // Getters for geometry and units not shown
  public double distance( Position position )
  {
    // Calculate and return the distance from this object to the specified
    // position using the current geometry and units.
  }
  public double heading( Position position )
  {
    // Calculate and return the heading from this object to the specified
    // position using the current geometry and units.
  }
  private double phi;
  private double theta;
  private Geometry geometry;
  private Units units;
}
```

Notice that although `Position` no longer maintains internal data items `latitude` and `longitude`, the corresponding getter and setter methods remain. That stability isolates client objects from the internal change. By having getters and setters access the latitude and longitude attributes, clients needn't even know about the change. The only modifications necessary in the following code usage of the new `Position` class concern the addition of the units and geometry attributes:

```
Position myHouse = new Position( 36.538611, -121.797500 );
Position coffeeShop = new Position( 36.539722, -121.907222 );
double distance = myHouse.distance( coffeeShop );
double heading = myHouse.heading( coffeeShop );
System.out.println
  ( "Using " + myHouse.getGeometry() + " geometry, " +
    "from my house at (" +
    myHouse.getLatitude() + ", " + myHouse.getLongitude() +
    ") to the coffee shop at (" +
    coffeeShop.getLatitude() + ", " + coffeeShop.getLongitude() +
    ") is a distance of " + distance + " " + myHouse.getUnits() +
    " at a heading of " + heading + " degrees."
  );
myHouse.setGeometry( Geometry.SPHERICAL );
myHouse.setUnits( Units.STATUTE_MILES );
distance = myHouse.distance( coffeeShop );
heading  = myHouse.heading(  coffeeShop );
System.out.println
  ( "Using " + myHouse.getGeometry() + " geometry, " +
    "from my house at (" +
    myHouse.getLatitude() + ", " + myHouse.getLongitude() +
    ") to the coffee shop at (" +
    coffeeShop.getLatitude() + ", " + coffeeShop.getLongitude() +
    ") is a distance of " + distance + " " + myHouse.getUnits() +
    " at a heading of " + heading + " degrees."
  );
```

The above code generates the following output:

```
==========================================================================
Using Plane geometry, from my house at (36.538611, -121.7975)
to the coffee shop at (36.539722, -121.907222) is a distance of
9.79675938972254 Kilometers at a heading of 270.58013375337254
degrees.
Using Spherical geometry, from my house at (36.538611, -121.7975)
to the coffee shop at (36.539722, -121.907222) is a distance of
6.0873776351893385 Statute Miles at a heading of 270.7547022304523
degrees.
==========================================================================
```

The output includes both the geometry used for calculations and the distance units. The relatively small distance yields a negligible, though not imperceptible, difference between using plane and spherical geometries. The calculated headings differ slightly, and comparing the distances using the conversion factor of 1.61 kilometers per mile shows that the distances also vary slightly.

Cars and crows

Now that I know the coffee shop is 6.09 miles west of my house, I decide to drive there. In desperate need of a coffee fix, I jump in my car and prepare to head west. Unfortunately, that takes me right through the back of my garage, across the back lawn, and into a 20-meter deep ravine. So much for location-based services! I'd need more than coffee to get out of that mess.

Though I may think the idea great, the municipality in which I live hasn't yet approved a road directly linking my house with the coffee shop. So if I want java, I'll have to alter my planned route. `Position`'s distance and direction information works fine for crows, but not quite so well for cars.

I decide to build a class that represents the actual driving route between my house and the coffee shop. The path consists of a series of segments that connect points along the way. So class `Route` maintains the `Position` objects necessary to define a route. A first-cut (and certainly not complete) design looks like:

```
public class Route
{
  public Route( int segments )
  {
    positions = new Position[ segments + 1 ];
  }
  public void setPosition( int index, Position position )
  {
    positions[ index ] = position;
  }
  public Position getPosition( int index )
  {
    return position[ index ];
  }
  public Position[] getPositions()
  {
```

```
    return positions;
  }
  public double distance( int segmentNumber )
  {
    // Calculate the distance of the specified segment number
  }
  public double distance()
  {
    // Iterate over the positions and accumulate the distances between each.
  }
  public double heading( int segmentNumber )
  {
    // Calculate the heading for the specified segment number
  }
  private Position[] positions;
}
```

The following outlines a usage of class `Route`:

- Create a `Route` object with 10 segments
- Create 11 `Position` objects
- Place each `Position` in the appropriate spot along the route
- Use `Route` to calculate the distance and direction of the first segment
- Use `Route` to calculate the total distance

Exposing internal structure

Many problems abound in the simple first-cut design of class `Route`. From an encapsulation and information-hiding perspective, the accessor method `getPositions()`, which returns the array of `Position` objects used within class `Route`, proves potentially troublesome. Though class `Route` encapsulates the array, it does not protect the design decision that resulted in using an array. That is, by returning the internal array of `Position`'s objects, `Route` has exposed the decision to use an array. This is particularly egregious when the design uses a method name like `getPositionsArray()`. If at a later time the design is changed to an `ArrayList`, `Route`'s clients are exposed to the change. You could create and return a primitive array from the `ArrayList`, but the issue still remains. The choice of internal data structure used to manage the route should not affect external clients. That is a distinct difference between encapsulation and information hiding.

A second-cut design looks like this:

```
public class Route
{
  public Route()
  {
    positions = new ArrayList();
  }
  public void append( Position position )
  {
    positions.append( position );
  }
  public Position getPosition( int index )
```

```
  {
    return positions.get( index );
  }
  public double distance( int segmentNumber )
  {
    // Calculate the distance of the specified segment number
  }
  public double distance()
  {
    // Calculate the accumulated distance between each segment.
  }
  public double heading( int segmentNumber )
  {
    // Calculate the heading for the specified segment number
  }
  private List positions;
}
```

Removing `getPositions()` corrects that method's unwarranted exposure of internal details. Changing the data structure `positions` to a `List` and using `append(Position)` rather than `setPosition(int,Position)` further isolates the design decision regarding the internal collection being used.

Exposing internal implementation

More information-hiding problems lurk in this design. The method `getPosition(int)` exposes the actual internal data items maintained by `Route`. Any client can obtain a reference to any of the `Position` objects along the route and freely *change* the state of that `Position` object.

To appreciate the ramifications of this exposure, consider a possible class `Route` implementation of method `distance()`, which calculates the accumulated distance across all route segments. Suppose that the implementation iterates over the internal `Position` objects and uses each object to calculate the distance to the next `Position`. The calculation clearly requires using a single type of distance unit. To enforce this requirement, the method `append(Position)` in the code below uniformly sets the units of all input `Position` objects.

And now for the full vulnerability of the `getPosition(int)` method exposure: Though all the `Position` objects added to class `Route` might have their units properly set when added to the route, any client object could obtain a `Position` object and set the units to whatever the client chooses, *without* informing `Route`. The potential damage when calculating the result in method `distance()` proves particularly unnerving. `distance()` could unknowingly and easily add kilometers to statute miles.

The following version of class `Route` modifies the `append(Position)` method and corrects the `getPosition(int)` method by returning an equivalent `Position` object rather than the internal `Position` object:

```
public class Route
{
  public Route()
  {
    positions = new ArrayList();
  }
  public void append( Position position )
```

```
  {
    Position aPosition = new Position( position );
    aPosition.setUnits( myUnits );
    aPosition.setGeometry( myGeometry );
    positions.append( aPosition );
  }
  public Position getPosition( int index )
  {
    Position position = new Position( positions.get( index ) );
    return position;
  }
  public double distance( int segmentNumber )
  {
    // Calculate the distance of the specified segment number
  }
  public double distance()
  {
    // Calculate the accumulated distance between each segment.
  }
  public double heading( int segmentNumber )
  {
    // Calculate the heading for the specified segment number
  }
  private ArrayList positions;
  private Units myUnits;
  private Geometry myGeometry;
}
```

Class `Route` is not yet complete, but I have strengthened the design by properly hiding the design decisions made thus far.

Conclusion

Encapsulation is a language construct that facilitates the bundling of data with the methods operating on that data. Information hiding is a design principle that strives to shield client classes from the internal workings of a class. Encapsulation facilitates, but does not guarantee, information hiding. Smearing the two into one concept prevents a clear understanding of either.

The Java language manifestation of encapsulation doesn't even ensure basic object-oriented objects. The argument is not necessarily that it should, just that it doesn't. Java developers can blithely create bags of data in one class and place utility functions operating on that data in a separate class. So as a first rule:

> **Encapsulation rule 1: Place data and the operations that perform on that data in the same class**

This standard practice creates classes that adhere to the principles of abstract data types.

But you want more of your objects; you want them to represent cohesive, workable entities. A second rule concerns the manner of choosing the data and operations to encapsulate:

> **Encapsulation rule 2: Use responsibility-driven design to determine the grouping of data and operations into classes**

This second encapsulation rule actually pertains to information hiding as well: don't just bundle data and operations together; let design principles guide you. Ask yourself what actions an object of the class will be responsible for. Don't let the class encapsulate more or less than a comprehensive set of reasonable responsibilities.

As you have seen from the many examples above, the Java language encapsulation facility isn't enough to ensure a solid class design. The principle of information hiding stipulates that you shield an object's clients from the internal design decisions made for that class of objects. So as a first rule for information hiding:

Information hiding rule 1: Don't expose data items

Make all data items `private` and use getters and setters. Don't fool yourself into believing no harm will result from directly accessing an object's internal data items. Even if only you code against those internals, future vulnerability still exists. You can't predict when you might need to change the internal data's nature, and brittle coupling with client objects sounds unnerving when shattered.

Go one step further when hiding design decisions concerning internal data. When possible, don't even reveal whether an attribute is stored or derived. Client objects don't need to know the difference. An attribute is a quality or characteristic inherent in a class of objects. From the client's perspective, object attributes correspond to responsibilities, not internal data structure. That brings us to the next rule:

Information hiding rule 2: Don't expose the difference between stored data and derived data

For example, a client object only needs to know that an object has an attribute of *speed*. Use an accessor method named `speed()` or `getSpeed()` rather than `calculateSpeed()`. Whether the value is stored or derived is a design decision best kept hidden.

As a corollary to the first two information hiding rules, I place all internal data at the bottom of the class text, after the methods that operate on the data. When I examine a class to understand its code, looking first at its internal data leads me to ask the wrong initial questions. I strive to understand the responsibilities of a class before concerning myself with any data structure details. Placing data after methods in the class text reminds me, every time I look at the code, to think first about a class's behavior, not its structure.

The next rule concerns the choice of internal structure:

Information hiding rule 3: Don't expose a class's internal structure

Clients should remain isolated from the design decisions driving the selection of internal class structure. For example, a client should not know whether a primitive array or an `ArrayList` is used to maintain an internal collection of objects. Internal structure is particularly apparent through the use of method names like `getDataArray()` or `getTreeMap()`.

The final rule concerns choices of implementation details:

Information hiding rule 4: Don't expose implementation details of a class

Don't allow clients to know or invisibly affect a class's implementation details. For example, a client should not be able to alter an internal calculation's result by changing the state of objects used in that supposedly hidden calculation.

Though not an exhaustive list, those rules help separate the concept of encapsulation provided by the language from the information hiding provided by design decisions. Each rule is fairly easy to follow, and each will assist in creating classes that are not only more resilient to change, but also more easily used by client objects.

About the author

Wm. Paul Rogers is a senior engineering manager and application architect at Lutris Technologies, where he builds computer solutions utilizing Enhydra, the leading open source Java/XML application server. He began using Java in the fall of 1995 in support of oceanographic studies conducted at the Monterey Bay Aquarium Research Institute, where he led the charge in utilizing new technologies to expand the possibilities of ocean science research. Paul has been using object-oriented methods and technologies for more than nine years.