OOP Design 1

Here, we'll look at simple, elemental OOP design. Later, we'll look at the more complex issues.

#1 — Encapsulation

The most basic rule for OOP design is to associate behavior with the object it operates on. No reaching in: -- avoid reaching into an object to get its data to perform an operation. The whole Message / Method system encourages the right style. In OOP vocabulary, this is "encapsulation" -- each object maintains and protects its own state independently. From a client point of view, to operate on the state of an object, send the object a message (a request), and the object operates on itself.

Example 1 - Wrong

This first example is bad code because it reaches into the Binky object data to compute something. Typically you prevent this by making your instance variables private:

```
// client side code
private int computeSum(Binky binky) {
   int i;
   int sum = 0;
   for (i=0; i<binky.length; i++) { // NO -- reaching in
      sum += binky.data[i];         // NO -- reaching in
   }

   return sum;
}
```

Example 2 - Wrong

This next wrong example follows the letter but not the spirit of OOP. Accessors getLength() and getDatum() have been added to the Binky object, so we're not accessing data directly, but it's not really OOP because such a major operation on Binky data should be performed by the Binky on itself...

```
// client side code
private int computeSum(Binky binky) {
   int i;
   int sum = 0;
   for (i=0; i<binky.getLength(); i++) {  // NO -- external entity doing
      sum += binky.getDatum(i);           // too much work on object's data
   }

   return sum;
}
```

Example 3 - Right

If you find yourself doing the Foo operation on the data from an object, just endow the class with the Foo capability, and **it can do the operation on itself**. Notice how easy it is to write the code for computeSum() once it's been moved to be a method in the Binky class. No argument is necessary since it's just the receiver, and the instance variables are available with no further syntax. Move the operation to the data.

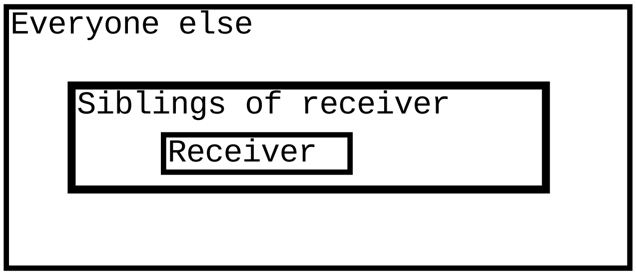
```
// Give Binky the capability
// (this is a method in the Binky class)
public int computeSum() {
   int i;
   int sum = 0;
   for (i=0; i<length; i++) {
      sum += data[i];
   }

   return sum;
}



// Now on the client side we just ask the object to perform the operation
// on itself which is the way it should be!
{
   Binky binky;

   ...
   int sum = binky.computeSum();
}
```

Reality

Not all cases are quite this tidy. Sometimes an operation requires significant access to data from two or more objects. In that case, you end up making it a behavior of one object and pass in the other object as a parameter. (Comparison operations always have this problem.) Almost always though, you can add some helper behaviors to one of the objects so the other can interact with it while still maintaining encapsulation.

3 Levels of Access

In C++ and Java the world actually divides into three: the receiver, the class of the receiver, and everyone else. Ideally the receiver operates on itself. Slightly worse but still allowed : another object in the *same class* performs the operation on the receiver -- "sibling access". Sibling access is not quite as nice as having the object operate on itself, but it is not terrible. Two objects in the same class (siblings) have literally the same implementation code, so at least an object and its sibling will have consistent code. Even if an ivar is declared "private", other objects in the same class are allowed access. Having an object operate on itself is the best. Having an object operate on a sibling is not quite as good, but is allowed.



Advantages

Advantages of having objects operate on their own state...

- Methods are easier to write -- fewer parameters to keep track of, the ivars are right there. After coding OOP for a while, you get fast at writing behaviors that operate against their receiver. Somehow it becomes more intuitive than writing functions, figuring out which parameter do what, etc.

- Client code is also better -- send the request to the object, don't worry about the implementation.

- Class independence -- can change the code in either client or implementor class without breaking the other. The message/request side is nice and separate from the method/implementation side.

Drill Bits vs Holes

There's the old story of the drill bit salesperson who was much more successful once they realized that their clients didn't want to talk about drill bits, the clients wanted to talk about holes. By the same token, if you find yourself making accessors for GetLength() and GetNthElement(), you have to think about what your clients really wanted — probably something more like FindElement() or WriteElements(). Think about what the client wanted to **accomplish**, not the details and mechanism of doing the computation.

Client Orientation

The OOP abstraction exposed by an object should be designed to meet the client needs. The messages should match the needs and vocabulary and conceptual orientation of the client. This is quite different from just exposing the object implementation. In fact, quite often, the implementation is hidden or heavily disguised to meet the client orientation.