OOP Encapsulation

OOP Design #1 -- Encapsulation

The most basic idea in OOP is that each object encapsulates some data and code. The object takes requests from other client objects, but does not expose its the details of its data or code to them. The object alone is responsible for its own state, exposing public messages for clients, and declaring private the ivars and methods that make up its implementation. The client depends on the (hopefully) simple public interface, and does not know about or depend on the details of the implementation.

For example, a HashTable object will take get() and set() requests from other objects, but does not expose its internal hash table data structures or the code strategies that it uses. The theme here is one of separation — trying to keep the complexity inside one object from interfering with other objects. Or put another way, trying to keep the various objects as independent from each other as possible (aka "loosely coupled" classes).

Each object provides some service or "interface" for the other objects. Ideally, the service is exposed in a way that is simple for the other objects to understand. The complexity of the implementation still exists, but it is isolated inside the implementing class. This works because for most problems there are all sorts of details of the implementation that the clients don't really care about -- how the hash buckets are arranged for example. The interface can capture just the issues relevant to the client, and so be much simpler than the full implementation.

Looking at a whole program, we have many objects, each exposing a simple interface to the other objects and keeping the details of its own implementation hidden. With all the objects following this strategy, we build up a divide-and-conquer solution to the whole program. Instead of a 1000 line program, we have a bunch of 200 line objects with minimal dependencies on each other. In this way, we escape the n^2 trap of writing and debugging large programs.

Public Interface/API Design

In its simplest form, encapsulation is expressed as "don't expose internal data structures," but there's more to it than that. For good OOP design, an object must think of an interface (or "API") to expose that most succinctly meets the needs of its clients. The best design is oriented towards the client's needs and vocabulary. What do the clients want to accomplish? What problem do they want solved, and what is the minimum set of details necessary to express the problem and its solution? The client of the BinaryTree class doesn't want getLeft() and insertRight() messages, they want add() and find().

Suppose an object has ivars x, y, and z. It's not an interesting OOP design if the exposed interface is getX(), setX(), getY(), setY(), and so on. A real OOP design is not just a 1-1 translation of the implementation. A good OOP design invents an interface that meets the needs of the client in terms they understand.

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 getNumElements() and getElement(), you have to think of the underlying problem the client wants solved— probably something more like findElement() or writeElements(). Think about what the client wants to **accomplish**, not the details and mechanism of doing the computation.

3 Examples

Often, the first rule that people learn for encapsulation is that instance variables should be declared private. Client objects should not "reach in" to access the data inside an object. Here are three examples that start with that simple sense of encapsulation and enlarge it to express the larger goals of OOP.

For this example, suppose we are writing client code to a Binky object that contains some integers, and we want to know the sum of those integers.

Example 1 - Reach In

This first example is bad code, since the client reaches into the Binky object to access the data. This violates the simplest notion of encapsulation. Typically an OOP design prevents this by keeping instance variables private.

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

Example 2 - Letter But Not the Spirit

This example is also bad. The code follows the letter of the law of encapsulation, but not its spirit. Accessors getLength() and getItem() have been added to the Binky interface, so the client does not technically access the data directly. However, this does not look like good OOP design. The client is pulling all the data out of the object to do an operation with it. **Ideally, an operation using an object's data should be performed by the object itself**.

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

Example 3 - Right

If you find yourself wanting to do some foo() operation that uses a lot of data from some object, then consider adding foo() as a method of the object and **it can do the operation on itself**. In this case, we move the sum() code to be a method of the Binky class. Notice how easy it is to write the code for sum() once it's been moved to the right class. No parameter is necessary since the Binky is the receiver and the required ivars are right at hand. **Push the code to the data**.

```
// Give Binky the capability
// (this is a method in the Binky class)
public int sum() {
    int sum = 0;
    for (int 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!
...
int sum = binky.sum();
```

Reality

In reality not all cases have a tidy solution where everything fits the encapsulation ideal. However, in general we want to move the operation to the object that contains the data. Sometimes an operation requires significant access to data that is split across two or more objects. In that case, you end up making it a method of one object and passing in the other objects as parameters. Many times though, you can add some helper methods to one or more of the objects so they can interact to get the job done while maintaining the spirit encapsulation.