6

Objects and Data Structures

There is a reason that we keep our variables private. We don't want anyone else to depend on them. We want to keep the freedom to change their type or implementation on a whim or an impulse. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?

Data Abstraction

Consider the difference between Listing 6-1 and Listing 6-2. Both represent the data of a point on the Cartesian plane. And yet one exposes its implementation and the other completely hides it.

Listing 6-1
Concrete Point

```
public class Point {
  public double x;
  public double y;
}
```

Listing 6-2
Abstract Point

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

The beautiful thing about Listing 6-2 is that there is no way you can tell whether the implementation is in rectangular or polar coordinates. It might be neither! And yet the interface still unmistakably represents a data structure.

But it represents more than just a data structure. The methods enforce an access policy. You can read the individual coordinates independently, but you must set the coordinates together as an atomic operation.

Listing 6-1, on the other hand, is very clearly implemented in rectangular coordinates, and it forces us to manipulate those coordinates independently. This exposes implementation. Indeed, it would expose implementation even if the variables were private and we were using single variable getters and setters.

Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions! A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its users to manipulate the *essence* of the data, without having to know its implementation.

Consider Listing 6-3 and Listing 6-4. The first uses concrete terms to communicate the fuel level of a vehicle, whereas the second does so with the abstraction of percentage. In the concrete case you can be pretty sure that these are just accessors of variables. In the abstract case you have no clue at all about the form of the data.

Listing 6-3
Concrete Vehicle

```
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}
```

Listing 6-4
Abstract Vehicle

```
public interface Vehicle {
  double getPercentFuelRemaining();
}
```

In both of the above cases the second option is preferable. We do not want to expose the details of our data. Rather we want to express our data in abstract terms. This is not merely accomplished by using interfaces and/or getters and setters. Serious thought needs to be put into the best way to represent the data that an object contains. The worst option is to blithely add getters and setters.

Data/Object Anti-Symmetry

These two examples show the difference between objects and data structures. Objects hide their data behind abstractions and expose functions that operate on that data. Data structure expose their data and have no meaningful functions. Go back and read that again. Notice the complimentary nature of the two definitions. They are virtual opposites. This difference may seem trivial, but it has far-reaching implications.

Consider, for example, the procedural shape example in Listing 6-5. The `Geometry` class operates on the three shape classes. The shape classes are simple data structures without any behavior. All the behavior is in the `Geometry` class.

Listing 6-5
Procedural Shape

```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
```

Listing 6-5 (continued)
Procedural Shape

```
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Object-oriented programmers might wrinkle their noses at this and complain that it is procedural—and they'd be right. But the sneer may not be warranted. Consider what would happen if a `perimeter()` function were added to `Geometry`. The shape classes would be unaffected! Any other classes that depended upon the shapes would also be unaffected! On the other hand, if I add a new shape, I must change all the functions in `Geometry` to deal with it. Again, read that over. Notice that the two conditions are diametrically opposed.

Now consider the object-oriented solution in Listing 6-6. Here the `area()` method is polymorphic. No `Geometry` class is necessary. So if I add a new shape, none of the existing *functions* are affected, but if I add a new function all of the *shapes* must be changed![1]

Listing 6-6
Polymorphic Shapes

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}

public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
```

1. There are ways around this that are well known to experienced object-oriented designers: VISITOR, or dual-dispatch, for example. But these techniques carry costs of their own and generally return the structure to that of a procedural program.

Listing 6-6 (continued)
Polymorphic Shapes

```
public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

Again, we see the complimentary nature of these two definitions; they are virtual opposites! This exposes the fundamental dichotomy between objects and data structures:

Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.

The complement is also true:

Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.

So, the things that are hard for OO are easy for procedures, and the things that are hard for procedures are easy for OO!

In any complex system there are going to be times when we want to add new data types rather than new functions. For these cases objects and OO are most appropriate. On the other hand, there will also be times when we'll want to add new functions as opposed to data types. In that case procedural code and data structures will be more appropriate.

Mature programmers know that the idea that everything is an object *is a myth*. Sometimes you really *do* want simple data structures with procedures operating on them.

The Law of Demeter

There is a well-known heuristic called the *Law of Demeter*[2] that says a module should not know about the innards of the *objects* it manipulates. As we saw in the last section, objects hide their data and expose operations. This means that an object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.

More precisely, the Law of Demeter says that a method *f* of a class *C* should only call the methods of these:

- *C*
- An object created by *f*

2. http://en.wikipedia.org/wiki/Law_of_Demeter

- An object passed as an argument to *f*
- An object held in an instance variable of *C*

The method should *not* invoke methods on objects that are returned by any of the allowed functions. In other words, talk to friends, not to strangers.

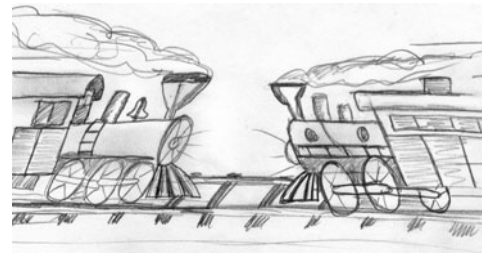
The following code[3] appears to violate the Law of Demeter (among other things) because it calls the `getScratchDir()` function on the return value of `getOptions()` and then calls `getAbsolutePath()` on the return value of `getScratchDir()`.

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Train Wrecks

This kind of code is often called a *train wreck* because it look like a bunch of coupled train cars. Chains of calls like this are generally considered to be sloppy style and should be avoided [G36]. It is usually best to split them up as follows:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```



Are these two snippets of code violations of the Law of Demeter? Certainly the containing module knows that the `ctxt` object contains options, which contain a scratch directory, which has an absolute path. That's a lot of knowledge for one function to know. The calling function knows how to navigate through a lot of different objects.

Whether this is a violation of Demeter depends on whether or not `ctxt`, `Options`, and `ScratchDir` are objects or data structures. If they are objects, then their internal structure should be hidden rather than exposed, and so knowledge of their innards is a clear violation of the Law of Demeter. On the other hand, if `ctxt`, `Options`, and `ScratchDir` are just data structures with no behavior, then they naturally expose their internal structure, and so Demeter does not apply.

The use of accessor functions confuses the issue. If the code had been written as follows, then we probably wouldn't be asking about Demeter violations.

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

This issue would be a lot less confusing if data structures simply had public variables and no functions, whereas objects had private variables and public functions. However,

3. Found somewhere in the apache framework.

there are frameworks and standards (e.g., "beans") that demand that even simple data structures have accessors and mutators.

Hybrids

This confusion sometimes leads to unfortunate hybrid structures that are half object and half data structure. They have functions that do significant things, and they also have either public variables or public accessors and mutators that, for all intents and purposes, make the private variables public, tempting other external functions to use those variables the way a procedural program would use a data structure.[4]

Such hybrids make it hard to add new functions but also make it hard to add new data structures. They are the worst of both worlds. Avoid creating them. They are indicative of a muddled design whose authors are unsure of—or worse, ignorant of—whether they need protection from functions or types.

Hiding Structure

What if `ctxt`, `options`, and `scratchDir` are objects with real behavior? Then, because objects are supposed to hide their internal structure, we should not be able to navigate through them. How then would we get the absolute path of the scratch directory?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

or

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

The first option could lead to an explosion of methods in the ctxt object. The second presumes that `getScratchDirectoryOption()` returns a data structure, not an object. Neither option feels good.

If `ctxt` is an object, we should be telling it to *do something;* we should not be asking it about its internals. So why did we want the absolute path of the scratch directory? What were we going to do with it? Consider this code from (many lines farther down in) the same module:

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

The admixture of different levels of detail [G34][G6] is a bit troubling. Dots, slashes, file extensions, and `File` objects should not be so carelessly mixed together, and mixed with the enclosing code. Ignoring that, however, we see that the intent of getting the absolute path of the scratch directory was to create a scratch file of a given name.

---

4. This is sometimes called Feature Envy from [Refactoring].

So, what if we told the `ctxt` object to do this?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

That seems like a reasonable thing for an object to do! This allows `ctxt` to hide its internals and prevents the current function from having to violate the Law of Demeter by navigating through objects it shouldn't know about.

Data Transfer Objects

The quintessential form of a data structure is a class with public variables and no functions. This is sometimes called a data transfer object, or DTO. DTOs are very useful structures, especially when communicating with databases or parsing messages from sockets, and so on. They often become the first in a series of translation stages that convert raw data in a database into objects in the application code.

Somewhat more common is the "bean" form shown in Listing 6-7. Beans have private variables manipulated by getters and setters. The quasi-encapsulation of beans seems to make some OO purists feel better but usually provides no other benefit.

Listing 6-7
`address.java`

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;

  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }
```

Listing 6-7 (continued)
address.java

```
  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

Active Record

Active Records are special forms of DTOs. They are data structures with public (or bean-accessed) variables; but they typically have navigational methods like `save` and `find`. Typically these Active Records are direct translations from database tables, or other data sources.

Unfortunately we often find that developers try to treat these data structures as though they were objects by putting business rule methods in them. This is awkward because it creates a hybrid between a data structure and an object.

The solution, of course, is to treat the Active Record as a data structure and to create separate objects that contain the business rules and that hide their internal data (which are probably just instances of the Active Record).

Conclusion

Objects expose behavior and hide data. This makes it easy to add new kinds of objects without changing existing behaviors. It also makes it hard to add new behaviors to existing objects. Data structures expose data and have no significant behavior. This makes it easy to add new behaviors to existing data structures but makes it hard to add new data structures to existing functions.

In any given system we will sometimes want the flexibility to add new data types, and so we prefer objects for that part of the system. Other times we will want the flexibility to add new behaviors, and so in that part of the system we prefer data types and procedures. Good software developers understand these issues without prejudice and choose the approach that is best for the job at hand.

Bibliography

[Refactoring]: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

This page intentionally left blank