12

Objects and Data Structures

Niklaus Wirth once wrote a lovely book titled *Algorithms + Data Structures = Programs*.[1] Though it wasn't his intent, the title very neatly captures the fundamental concept behind object-oriented programming: that there is a deep relation between functions and the data they manipulate.

1. [A+DS=P].

Object orientation was invented[2]/discovered by Ole-Johann Dahl and Kristen Nygaard in the mid- to late '60s through the gradual evolution of their Simula language, culminating in Simula 67. Bjarne Stroustrup and Alan Kay both used Simula 67 in the early '70s and then extended the ideas in C++ and Smalltalk, respectively.

2. The story of this invention is fascinating. I wrote about it in [We Programmers].

From there, C++ evolved into Java and C# and then Go, while Smalltalk evolved into Objective-C, Ruby, and Python. Both of those lines of evolution borrowed from each other, causing a lot of cross-pollination. Today, virtually all of our languages are object oriented in one sense or another.

What Is an Object?

An object is a bundle of data manipulated by dedicated functions, some of which are public. From the outside looking in, only those public functions are visible.[3] The structure of the data within the object is hidden.

3. Some languages enforce that visibility with keywords like `public` and `protected`. Other languages depend upon programmers to adopt conventions to determine which functions are public and private.

An object-oriented program is composed of a group of such objects, each with its own responsibilities and each collaborating with the others by calling each other’s public functions.

Alan Kay once imagined this using a biology analogy—a set of cells collaborating by passing molecular messages to each other. The cells do not know of each other’s internals, and may not even know the particular cells they are communicating with. They simply emit messages (molecules) that other cells detect and respond to. Notice that cells never ask other cells questions. They send messages but do not expect answers.

The message-passing concept is a very powerful principle for organizing behavior. If you only tell an object what to do and never ask that object anything, then the data in that object is not relevant to you.

Avoiding all queries is impractical in software; but as we shall see, there are ways to keep such queries to a minimum. This lack of dependence upon data means that object-oriented designs focus primarily on behavior. Objects are constructed and related so as to support the flow of messages that elicit the required behaviors of the system. The internal structure of the data within the individual objects is hidden from, and therefore nearly irrelevant to, the majority of that behavior.

But OO is not the only game in town.

In 1970, Edgar Codd published *A Relational Model of Data for Large Shared Databanks*.[4] From there he, Raymond Boyce, and Chris Date derived/discovered the concept that currently drives all relational databases. The overarching concept is that the structure of the data is critical and visible, and all programs that use it are subservient to it.

---

4. *Communications of the ACM* 13, no. 6 (June 1970), 377–387.

A large enterprise database is a complex, sprawling data structure, whereas the programs that manipulate it are little more than minions that surround it. The schema of the database dominates those programs.

This is the opposite of OO. To recap: An object is a set of public functions with data hidden deep inside. A database is a published set of data structures with no intrinsic functions. Again, they are very nearly opposites. And, as we shall see, that opposing nature goes very deep indeed.

Data Abstraction

There is a reason that we tend to 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?

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

```
---Point 1---
public class Point {
  public double x;
  public double y;
}

---Point 2---
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 `Point 2` is that there is no way you can tell whether the implementation is in rectangular or polar coordinates. It might be neither! The interface represents an *abstract* data structure. We can tell that there must be a data structure in there somewhere, but we do not know what form that data structure takes.

But `Point 2` represents more than just a data structure. It also represents behavioral policies. Despite the fact that the getter methods allow you to access individual coordinate values, the two setter methods enforce the policy that the two coordinate values are set together as an atomic operation.

`Point 1`, on the other hand, is very clearly implemented in rectangular coordinates, and it forces us to manipulate the coordinate values 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 how that data is implemented.

Consider `Vehicle 1` and `Vehicle 2` below. 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.

```
---Vehicle 1---
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}

---Vehicle 2---
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 Antisymmetry

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 structures expose their data and have no meaningful functions. Go back and read that again. Notice the complementary 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 Shapes` example below. 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.

```
---Procedural Shapes---
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 NoSuchShape
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    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 data structures would be unaffected! Any other modules that depended upon those 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. When you add a new function, the data structures remain unchanged; but when you add a new data structure, the functions must all be changed.

Now consider the `OO Shapes` example below. 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![5]

---

[5]. 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.

```
--OO 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;
  }
}
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 complementary 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 data structures (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 these cases, 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[6] that recommends that 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.

---

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

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

- *C*
- An object created by *f*
- 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, be shy, talk to friends, and don't talk to strangers.

The following code[7] 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()`.

---

7. Found somewhere in the Apache framework.

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

Trainwrecks

This kind of code is often called a *trainwreck* 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. 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 in the examples above confuses the issue. If the code had been written as follows, then we probably wouldn't be worried about Demeter violations.

```
final String outputDir = ctxt.options.scratchDir.absc
```

This issue would be a lot less confusing if we adopted the convention that data structures had public variables and no functions, whereas objects had private variables and public functions that abstracted the implementation.

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.[8]

8. This is sometimes called *feature envy* in [Refactoring].

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('.', '/') + ".clas
FileOutputStream fout = new FileOutputStream(outFile)
BufferedOutputStream bos = new BufferedOutputStream(f
```

The admixture of different levels of detail 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.

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.

So, we can often fix trainwrecks and Law of Demeter violations by telling objects to do what we want, as opposed to traversing the network of objects to gather all the resources. This advice is called *Tell the Other Guy*.

Long ago Dave Thomas and Andy Hunt wrote a lovely article titled “OO in One Sentence: Keep it DRY, Shy, and Tell the Other Guy.”[9] DRY, of course, is Don’t Repeat Yourself, and being shy refers to the Law of Demeter. Telling the other guy is a reference to Alan Kay’s message-passing idea or command/query separation (CQS). Good advice all around.

9. *IEEE Software* 21, no. 3 (May/June 2004), 101–103.

Data Transfer Objects

The quintessential form of a data structure is a context, class, or struct 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, 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.

Early in the Java era, the “bean” form shown below was popular. 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. The `Address` DTO might as well have public fields.

```
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 z
    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;
  }
  public String getState() {
    return state;
  }
  public String getZip() {
    return zip;
  }
}
```

The Object/Relational “Impedance Mismatch”

In the late ’80s, as OO grew in popularity, the difference between the OO concept of public behavior and private data and the relational concept of public data and no behavior became clear. Programmers sought a merger of the two ideas; but they were disappointed with the result. For a while, object-oriented databases were tried; but these never really solved the issue. In the end, we fell back on object-relational mappers (ORMs).

The hope was that ORMs would allow programmers who were using object-oriented languages to conveniently access the data within a relational database. The ORM would query the database, unpack the data from the resulting table rows, and repackage the data into the data structures used by the objects in their programs.

This works well enough as a way to convert tables into a more convenient form, but it does not resolve the dilemma, because objects are more about functions than data. In the end, there is no way to map a relational database to an object, because databases contain data structures, and data structures and objects are orthogonal. All we can really manage to do is extract data from a database and place it into a form that an object can internalize. ORMs don’t create objects, they just load data structures.

Using Objects and Data Structures

So why do we have these two opposite concepts? Why do we sometimes want the data structure to be hidden and the behavior exposed and other times want the behavior to be hidden while the data structure is exposed?

It all comes down to which is more likely to grow and change.

If there are likely to be new behaviors, and if existing behaviors are likely to change, then we want to hide the volatile behaviors while exposing the static data structure. If, on the other hand, it is the data structures that are likely to grow and change, then we want to expose the static behavior and hide the volatile data structures.

We hide the volatile so that the changes don’t affect the static. We build a barrier that protects the static from the volatile.

Example 1: The structure of a large enterprise database is relatively static. The data itself may change as new records are added and old records are updated or deleted, but the structure changes very infrequently. However, the behaviors of the programs that use that database change frequently. Business rules are volatile, but business structure is stable in comparison.

Example 2: The behavior of an IDE is relatively static. It is an editor of text files. But the structure of the data changes every time a new language is added to its capabilities. Users expect the IDE to behave consistently, whether programming in Ruby, Python, Java, or Clojure; and yet those languages have remarkably different structures. The overall data structure is volatile compared to the behavior.

Determining whether data or behavior is the more static is a decision that requires some thought and experience. It is also a decision that may have to be made differently in different parts of a system.

A complex system can have components of both kinds. Some components may have behavior that is static relative to the data structure, and others may have data structures that are static relative to the behavior.

Oddly enough, all this boils down to the decision to use `switch` statements.

`Switch` Statements

`Switch` statements and their siblings, `if` / `else` chains, are considered by some to be bad form and by others to be utterly essential. The reason behind this disagreement is the same as the decision over static behavior or data structures. Consider this simple program.

```
——Shape.java——
enum ShapeType {CIRCLE, SQUARE}

public class Shape {
  public ShapeType type;
}

——Square.java——
class Square extends Shape {
  public int side;
  public Point topLeft;
}

——Circle.java——
class Circle extends Shape {
  public int radius;
  public Point center;
}

——ShapeManager.java——

public class ShapeManager {
  public static void renderShapes(List<Shape> shapes)
    for (Shape shape : shapes) {
      switch (shape.type) {
        case CIRCLE:
          renderCircle((Circle) shape);
          break;
        case SQUARE:
          renderSquare((Square) shape);
          break;
      }
    }
  }

  private static void renderSquare(Square square) {
    //…
  }

  private static void renderCircle(Circle circle) {
    //…
  }
}
```

If you are a Java programmer, this might look a little odd to you. If you are a C programmer, this might look quite normal to you.

Now consider the consequence of adding a `Triangle`.

The first file that needs to change is *Shape.java*, because `TRIANGLE` has to be added to the enum. Next, of course, the `Triangle` data structure (class) must be written. And then, finally, the `Triangle` case needs to be added to the `switch` statement in the `ShapeManager` class.

This design forces several modules to change, and others to be added, in response to one simple change to the requirements. This violates the Single Responsibility Principle (SRP) and the Open–Closed Principle (OCP). The number of changes also means that this is a rigid design. The more changes that are required, the more rigid the design becomes.

But it's worse than that, because that `renderShapes` method in `ShapeManager` is not the only method of that class. To save space I didn't write them all, but there's `dragShapes`, `eraseShapes`, `rotateShapes`, `scaleShapes`, and a menagerie of other functions that manipulate shapes. And every one of them has that very same `switch` statement in it. And every one of them is going to need the `Triangle` case added to it.

Except that some of those methods don't use `switch` statements. Some of them use `if` / `else` chains. What's more, clever programmers may use logical optimizations to eliminate or modify some of the `switch` statements and `if` / `else` chains. For example, the `switch` statement in `rotateShapes` may not contain a `CIRCLE` case, because a clever programmer realized that circles have rotational symmetry.

So, we are going to have to find all those `switch` / `if` / `else` statements, decode all the logical optimizations, and figure out some way to get them all to properly deal with the new `TRIANGLE` case. And, in large systems, we'll almost certainly miss one … or more.

A design that forces many modifications to be made within modules when only one simple requirement is changed or added is both rigid and fragile—it's hard to change and it breaks easily. The more changes that are required, the more fragile the design is.

But it's worse than that. Because, you see, our business has been trying to sell this shape management system for several months now, and without a lot of success. We were all starting to worry about our jobs. Life at a startup rests on a tenuous thread. Fortunately, our CEO has just made a critical announcement. He *knows* what the problem is, and he knows how to solve it.

"Our problem," he says, "is that we've been forcing our customers to deal with circles and squares. Customers don't want that. They want a shape management system that allows them to choose whether they use circles or squares. They don't want to pay for circles if they only use squares. And who can blame them?

"So," he continues, "We are going to give the `ShapeManager` facility away for free. Customers can download it from our website. But … He pauses here for effect, because this is the insight that will save the company. "… We'll make them pay to download circles. And we'll make them pay to download squares. And if they want other shapes, we'll make them pay for those as well. Instead of selling one big system that manages shapes, we are going to sell the shapes themselves—and we're going to make a blithering fortune!"

And amid all the cheers, balloons, and popping of champaign corks, you realize that the CEO's dream is dead on arrival. It is dead because `Circle` and `Square` are woven through the entire fabric of this system. All those `switch` statements, all those `if` / `else` statements, all those clever logical optimizations: They all deal specifically with `Circle` and `Square`. There's just no way the architecture of this system is going to allow the CEO to give the `ShapeManager` away for free without including the `Circle` and `Square` with it. They are inextricably bound together.

A design that prevents its parts from being separated and deployed independently from its other parts violates the Dependency Inversion Principle (DIP) and is immobile. The fewer parts that are independently deployable, the more immobile it is.

This design is rigid, fragile, and immobile. It's trash. It's a horror scene. It's going to take the company down, and all of us with it!

The OO Solution

We hire a famous consultant who flies in, makes a lot of noise, craps all over everything, and then flies home.[10] His message is: "Your design is procedural. It should have been object oriented. Here, let me show you." And what he shows is this:

---

10. These are sometimes known as *seagull consultants*.

```
——Shape.java——
interface Shape {
  void render();
}

——Square.java——
class Square implements Shape {
  public int side;
  public Point topLeft;

  public void render() {
    //…
  }
}

——Circle.java——
class Circle implements Shape {
  public int radius;
  public Point center;

  public void render() {
    //…
  }
}
```

```
——ShapeManager.java——
public class ShapeManager {
  public static void renderShapes(List<Shape> shapes)
    for (Shape shape : shapes) {
      shape.render();
    }
  }
}
```

Not a `switch` statement in sight. Now consider what happens when you add the `TRIANGLE` case.

- `Shape.java` does *not* need to change.
- `Square.java` does *not* need to change.
- `Circle.java` does *not* need to change.
- `ShapeManager.java` does *not* need to change.

Nothing needs to be changed; except, perhaps, for the `main` program. But none of these modules need changing. And that means the design is not rigid.

There are no `switch` / `if` / `else` statements to find and change either. All the methods, like `erase`, `drag`, `scale`, and `rotate`, are contained within the `Shape` classes. This design is not fragile.

Best of all, we can deploy `Circle` and `Square` separately from `ShapeManager`. We could put `ShapeManager`, `Circle`, and `Square` into their own `jar` files and distribute them individually!

This design conforms to the OCP and the DIP. It is mobile, flexible, and robust. Hallelujah! We are saved!

Not So Fast, Johnson

What does this design protect us from? It protects us from new shapes. Whenever a customer wants a new shape, all we need to do is write a new derivative of the `Shape` class and put it into its own `jar` file that can be independently distributed and deployed.

But what if the customers don't ask for a new shape? What if they ask instead for a new method on `Shape` that renders drop shadows?

Now our `Shape` abstraction doesn't help us at all. We're going to have to take that lovely OO design and touch every single one of those classes. `Shape` , `Circle` , and `Square` will all need a `renderDropShadow` method; and the `ShapeManager` will have to call it. That `Shape` abstraction gave us no benefit at all.

But now look back at that first solution, the one with the `switch` statement. All we have to do to that one is add the `renderDropShadow` functions to the `ShapeManager` . We don't have to touch the `Circle` or `Square` at all!

The OO/Procedural Trade-off

If our design interprets changes to the requirements as new types, then OO is a good choice because it is easy to add new types, like `Triangle` , without changing anything else. However, if our design interprets changes as new functions, then the procedural `switch` statement is best because we don't have to change any of the types.

Or, to say this differently, we use OO when we want to add new data types to existing functions, and we use procedural `switch` statements when we want to add new functions to existing data types.

Again we see this interesting opposition. Hiding data and exposing functions allows the data to change without significant impact. Hiding functions and exposing data allows the functions to be changed without significant impact.

So which is it? Which would we rather change, functions or data? And the answer to that is that it depends on how your application and your design interpret changes to that application.

In large enterprise models, we often write new applications against the existing data schema. Whereas when we add new capabilities to a given application, we often encapsulate that behavior into new data types.

Personally, I prefer the latter. Where possible, I create designs that interpret changes as new data types. And that means I prefer an OO design. For the most part, I find that works pretty well.

Thus, my general rule for `switch` statements and their ugly duckling sibling, `if` / `else` chains, is that they should be located at the periphery of the system (e.g., the `main` module); there should be, at most, one for each switchable type; and the cases within them must create the polymorphic objects that the rest of the system uses.

However, that preference and that rule do not mean that I won't use a `switch` statement in those parts of the system where behavior is overwhelmingly more volatile than data. I won't force an OO round peg into a procedural square hole, or vice versa.

But What About Performance?

It is fair to say that today's compilers can take advantage of the local nature of `switch` statements and specific processor architectures to create executable code that runs somewhat faster than a polymorphic dispatch. Thus, OO designs can incur a slight performance penalty. If you are working on a project where a few nanoseconds here or there are critical, then it may be that you will have to abandon the OCP and the DIP and use `switch` statements instead of polymorphism.

Most programmers have no need to care about stray nanoseconds. But if you do, be careful to concentrate them where they do you the most good. Don't abandon OO design throughout the entire system just because a few time-critical inner loops must use switches.

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.