NAME

classesoop − Object Oriented Primer

DESCRIPTION

If you couldn't really explain the difference between a *class* and an *object* or are confused about *methods* and *attributes* then this primer is for you. We presume no prior object experience covering the main concepts and terms necessary to begin object−oriented programming.

Beyond this primer you can find more in the *Classes Tutorial* (*classestut*), the *Classes Cookbook* (*classescb*), the *Classes FAQ* (*classesfaq*), and the *Classes Reference* (*classes*). Later you may decide you want to read the older *perlobj*, *perltoot*, and friends.

What is an object anyway?

You probably already know the answer to that intuitively. An *object* is just a thing. Things do stuff; they perform *operations* using different *methods*. Things also have characteristics or *attributes*, some you can see and some you can't.

In the real−world objects are grouped together by their similarities into *classes*. Biologists really get this. The rest of us can also if we think about objects we know.

Objects on the Road

One universal class of objects that we most all have experience with is *Vehicle*. A popular subclass of Vehicle is the *Motorcycle*.

A black chromed American V−Twin motorcycle idling at an intersection is a thing (of beauty), an object. So is a sleek Japanese metric motorcycle doing 200 KPH. Both are motorcycles, a *class* of vehicles found on the roadway—another (bigger) *object*.

The roadway might contain all kinds of motorcycles and other vehicles interacting with one another. Each vehicle belongs to a particular group, a *subclass* of vehicles. Each has a color, size, weight, number of wheels, engine type, and other static recognizable *attributes*. Some of the attributes are really attributes of their parent class, color for example. All vehicles have at least some basic color. Current speed, gear, fuel level, choke, and air pressure are *dynamic attributes* some of which all vehicles have and others only certain vehicle subclasses have.

Motorcycles can move, no question about that—that's an *operation* all (working) vehicles can perform. From the outside, we riders don't see much that reveals the how a motorcycle moves. Most of us don't really care about the *method* the engine uses to make it go—so long as it does go. We fill the tank, enjoy the sound of the engine, sometimes smell the exhaust, but we don't know what is going on *internally*—unless we take it apart.

When we do take apart a few engines we see that they can be quite different—that they have different *implementations* and *methods* to do pretty much the same thing, make the bike move. The complexity of any engine can be broken down into the parts that make up the engine—the *objects* that *compose* it—and how all those parts work together—how the *aggregation* of *objects collaborate* to fulfill the 'move' operation.

If a part breaks, we replace it with another that will fit—that has the same *interface*. In fact, we can replace stock parts like the pipes or air intake with more powerful ones so long as the *interface* is the same.

Objects in Programming

Object−oriented programming is nothing more than putting these real−world concepts "from the road" into practice when programming. OO helps make sense of otherwise complex systems by breaking them down into their fundamental parts, describing how those parts work with one another in collaboration, and aggregating them together into bigger parts. This is why OO is so popular and important to the enterprise.

Technically speaking OO is not much more than creating code packages (classes) of variables (attributes) and functions (operations, methods). *Motorcycle* is a class. So is *Engine*, *Throttle*, *Car*, or *Vehicle*. A specific *instance* of a *Motorcycle* is a motorcycle *object*. That motorcycle instance might have different attribute values at different times but it is still a *Motorcycle*. To create a motorcycle we have to *construct* it from some

sort of *Motorcycle* class specification.

Methods and Operations − *What* does this thing do?

What should our *Motorcycle* do? Most agree this question begins the easiest approach to OO design—especially when designing a whole system of classes. The question really is *what responsibilities does this class have?* Writing this up on cards or in UML Entire methodologies, diagramming languages, and tools have been created around just this step. [The Unified Markup Language (UML) is *the* lingua franca of OO.] For now we care only about our simple *Motorcycle* class:

```
startup
idle
shift
speedup
slowdown
turn
shutdown
```

Many of these are actually operations that all vehicles share and fulfill for their users with different methods.

You might have noticed the terms *method* and *operation* used interchangably. Chances are you will hear coders say *method* when they really mean *operation* and architects sometimes say *operation* when they mean *method*.

Technically a *method* is an implementation of an *operation*. This actually matches the more common real−world definitions of those terms as well. *method* has come to mean both in OO programming lingo. You will still find *operations* used in UML diagrams and high−level architect−speak, however. You'll also probably hear coders raised on nothing but OO refer to *procedures*, *functions*, or *subroutines* as ubiquitous *methods*.

Attributes − What does this thing look like from the outside?

Attributes tend to fall out while designing what a class does In fact, if we have designed the *methods* and *interface* really well we might not have a single public attribute in our class. Why? Because well−behaving methods take care of their own work using private variables and attributes to get it done. The public is never involved. In the case of our Motorcycle, however, there are a few attributes that the user must or can set directly:

```
key_position
choke
kill_switch
color
license_plate
```

And a few that the user can observe but that cannot be directly set:

```
speed
odometer
fuel_level
gear
```

Grouping Methods and Attributes

Many of the operations and attributes described for the Motorcycle class actually apply to the Vehicle class even though the Motorcycle implements them differently. In OO there are several design approaches to pulling these out into a Vehicle.

Inheritance

One is to define an *abstract* Vehicle class and make Motorcycle a *subclass* that *inherits* from Vehicle. Each of the operations is declared as *abstract* to show that any subclass must implement its actual method. There is *single inheritance* where classes can only inherit from a single parent and *multiple*

inheritance where multiple parents can be inherited. Biologists are also familiar with the complexity of mixing genetic material in multiple inheritance. The same is true for OO programming. Yet single inheritance tends to slow things down as method calls are propogated through the inheritance tree.

Interfaces

Interfaces and APIs are just groups of methods abstract or otherwise that any class fulfilling the interface agrees to implement. This handles the propagation issue from inheritance, but adding a method to an interface doesn't automatically add it to all the classes that claim to implement the interface.

Mixins

Mixins are relative newcomers to OO and are not commonly known even among long−time object programmers. Mixins are, nevertheless, a powerful balance between inheritance and interfaces. Mixins provide a powerful replacement for inheritance that grants the flexibility of inheritance to add code in an organized way that is automatically propogated to all the users of the mixin but without the bloat and call proporation issues of inheritance.

A *mixin* in the strict sense cannot be considered a class nor an interface. Implementations of mixins vary between languages, but the idea is the same. Mixins have methods and attributes that classes can "mix into" themselves, like mixing different groups of ingredients into vanilla ice cream to create a new flavor. Effectively it is as if the class using the mixin inherited it but method calls are not propogated through an interitance tree because every class that uses the mixin gets its own direct references to the methods, more or less. And, unlike a simple interface specification, the classes actually get something more than a mandate about what they must implement.

Summary

Object−oriented programming provides an effective way to simplify complex problems and systems into their finite components. Processes are then modeled as interactions and collaborations between them. Developers can then focus on the responsibilities and internals of each component.

AUTHOR

Robert S. Muhlestein (rmuhle at cpan.org)

COPYRIGHT AND LICENSE