An Introduction to Object-Oriented Design

Lane W. Surface

4 March 2019

Among many of the frameworks, APIs, languages, and other skill sets you will be expected to know as a professional programmer, a sound knowledge of the principles of object-oriented design (henceforth referred to as OOD) will be invaluable in your career. Though it may seem simple on the surface, even professionals struggle with the decisions that come with a well-designed and easy to maintain system which strives to follow the best OOD practices. This document will serve primarily as a brief introduction for developers with no or limited experience in this domain.

1 Classes, Methods, and State: It All Ties Into Encapsulation

A recurring theme which you will come to see in the evolution of programming languages, and thus software development in general, is the tendency for a language to make the solution to a given problem both easier to write and easier to read. Assembly languages were originally the de-facto standard for writing applications, and many were written solely in it, including important and critical software like that which ran on the Apollo Guidance Computers and put men on the moon. From these struggles, a new way of writing software was proposed, and higher-level languages based on the notion of functions were created. From these languages has come one of the newest developments in the programming language world, which is based around the notion of the *object*.

But what is an object anyway? For how often the term is thrown around, there really isn't must of an attempt to explain it or the principles on which it is based. An object is simply a component of an object-oriented system; this means that there are many objects which make up an object-based software system. Each of these objects interacts with the others which make up this application to create a working program.

Let's expound this definition further: Each object itself has both state and behavior. This state, often called fields of the object, resides in class-level variables which are *instance-dependent.* In some languages these are called instance variables. State is a property of the object itself. Say you have a human object. This object might have state such as hair color, race, ethnicity, nationality, etc.

Although the state of an object is important, it is mostly internal (and should be kept hidden; we'll address this further in a moment). *The primary way that objects should expose their functionality to other objects in a system is though instance-dependent functions, which are often referred to as methods of that object.* These methods use the state of their object to determine the action that should be taken, without requiring the objects which call that function to know about the state itself. Therefore, calling a method such as `pray()` on a human object should determine the way that it prays by these factors, without a caller needing to know all of the internal details.

Now I've skipped over an important distinction to make, as I wanted to avoid confusing you up front. How do we create and address the objects in our system? This state is not intrinsic, and we have to define it somewhere in our application. We do so by defining classes. Classes are a template for the construction of an object. In other words, the class defines the *how*, and when we construct an object from a class, we define the *what.* A class will determine the kinds of things which we need to know about. Taking from the earlier example of a human object, the human class will tell the compiler that we need each of our human objects to have an integer for their age, string for their name, etc. Then when we initialize a human object (as there may be many in our system), we provide the values of these fields. It is only after an object has been constructed and initialized that we can call the methods which it defines. The type of the object is the same as the class which defines it, and the name that the variable is given when constructed is the object itself.

You may be wondering why it's considered good practice to not expose the state of an object. The reasons are twofold. First, forcing a client to interact with the state of an object increases the complexity of your code. The purpose of designing an object-oriented system is to hide away information that is not important to the rest of the system. If the only thing we need to know is what behavior an object supports (i.e., the methods which it defines in its public interface), then we are much less likely to be surprised by the code that we write. Another reason is because, as software is maintained over its lifetime, the fields and their respective types are very likely to change; it is not often that we write code once and are done with it. *The majority of the lifetime of a program is in maintaining it.* We will see that it is much easier to make modifications to the state and behavior of an object when we do not expose this state directly.

2 Further Conveniences of Object-Oriented Systems

Although encapsulation leads to reduced system complexity in general, the real power is not in this principle at all. In this section, we will explore ways to create cohesive and easily intelligible software in an object-oriented fashion, and see the way that the design decisions made in the implementation of one of these systems increases the legibility of complex software overall. Proper and thoughtful design is difficult; however, with practice, you will see how object-oriented programming languages can increase developer productivity significantly when in the right hands.

2.1 Class-Based Inheritance

Of all the features available to you in an object-oriented toolset, inheritance is one of the best-known and most frequently abused aspects of any object-oriented programming language. Although powerful in its ability to reduce program size and shared object features, inheritance comes with a price and should be used carefully.

Inheritance is the ability for related classes to share the parts which are similar between them. If two classes have parts which are identical, they can each inherit from a common ancestor, which is usually referred to as the *superclass* of each of them. Each of these objects, in turn, can serve as the superclass of another class of objects, and the inheritance hierarchy continues naturally, as you would expect for it to. For example, the human class which we defined earlier could decide to inherit from a class named mammal. If we decided to add another class named cat, this class could also decide to inherit functionality from the common mammal class. These so-called *subclasses* can add their own functionality, yet retain the characteristics which they have in common. As the complexity of our system grows and more classes are added, the power of this ability becomes more prevalent as the amount of code is decreased significantly. That is to say, as the number of subclasses grows, the simpler it is to understand the system.

As mentioned before, there are some drawbacks to this ability as the hierarchy grows more complex (as there are to most things in object-oriented programming).If the division between these classes is not chosen carefully, the class hierarchy can quickly grow out of control. When this happens, other programmers who add to this system later are likely to reimplement something that already exists in one of these superclasses. This drawback mitigates the decision to employ inheritance in the first place, and can actually make the code base much more complex than was originally called for.

2.2 Interfaces and Implementations

I have mentioned numerous times within this document the "interface" of an object. What does this mean exactly? An interface is the way that other objects can interact with another. *Essentially, it is the behavior, or methods, which an object makes publicly available for use by any other.* In the event that encapsulation is used properly, the idea is that knowing an object's interface, or the behavior it exposes, is enough to properly use the object itself.

This has some advantages in terms of creating robust systems. (When I state that a system is robust, what I really mean is it is robust to change. This means that we can add functionality, most likely in the form of new classes, later down the road without crippling the architecture we have so carefully designed.) Recalling the information concerning inheritance which was detailed in the previous section, we know that classes may have inheritance-based relationships with one another. This is significant in terms of fortifying our system against change and easing the duty of expanding it in the future.

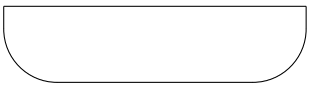
We can provide for objects which share some common behavior a common superclass which defines the methods for this behavior. This implies, of course, that any objects which are related to that superclass both support the behavior which it defines and in the way that it defines it. *We can interact with any object that inherits from this superclass as if it were the superclass itself, and we don't even need to know the class of the object we are dealing with.* If we know the interface for a class of objects, we know how to work with them. This principle of knowing how to interact with an object without knowing its concrete class is known as *transparency*—the idea that our system only sees the parts which are important and has the liberty to ignore the rest.

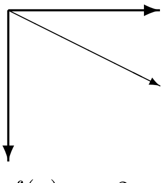
Using interfaces in a system to define the behavior which classes support is a good practice in object-oriented programming. This design practice is called *programming to an interface* and creates classes which are decoupled from each other. The state of one class being decoupled from another means that one can vary without affecting other objects which use it.

2.3 Polymorphism; Or How To Predict The Future

When our class inheritance hierarchy is based around interfaces which define the behavior which objects support, we have several other advantages which we need to keep in mind. One of these advantages, as we have already noted, is the decoupling of behavior from the implementation by classes which inherit, or implement, that interface. When we define the state of our objects, or the parameters to our methods then, we can use this interface as the object type instead of the concrete type itself. By doing so, we future-proof ourselves against changes in the system.

When we design our interfaces well, and classes which support a behavior which could be subject to change implement an interface, it is a painless process to expand this system. Adding new functionality is as simple as writing a new class which inherits from that same interface and exchanging the instances of that class whereever it is deemed convenient. The actual implementation in this new class may do different things than the original, but it supports that behavior through the same mechanisms as the old instances.





$f(x) = -2x$