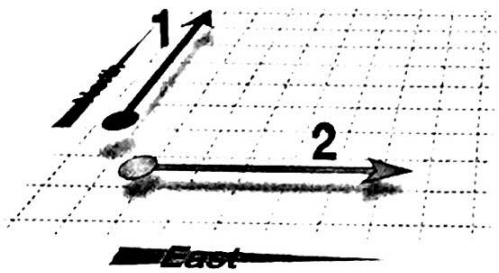
Orthogonality

Orthogonality is a critical concept if you want to produce systems that are easy to design, build, test, and extend. However, the concept of orthogonality is rarely taught directly. Often it is an implicit feature of various other methods and techniques you learn. This is a mistake. Once you learn to apply the principle of orthogonality directly, you'll notice an immediate improvement in the quality of systems you produce.

What Is Orthogonality?

"Orthogonality" is a term borrowed from geometry. Two lines are orthogonal if they meet at right angles, such as the axes on a graph. In vector terms, the two lines are *independent*. As the number 1 on the diagram moves north, it doesn't change how far east or west it is. The number 2 moves east, but not north or south.



In computing, the term has come to signify a kind of independence or decoupling. Two or more things are orthogonal if changes in one do not affect any of the others. In a well-designed system, the database code will be orthogonal to the user interface: you can change the interface without affecting the database, and swap databases without changing the interface.

Before we look at the benefits of orthogonal systems, let's first look at a system that isn't orthogonal.

A Nonorthogonal System

You're on a helicopter tour of the Grand Canyon when the pilot, who made the obvious mistake of eating fish for lunch, suddenly groans and faints. Fortunately, he left you hovering 100 feet above the ground.

As luck would have it, you had read a Wikipedia page about helicopters the previous night. You know that helicopters have four basic controls. The *cyclic* is the stick you hold in your right hand. Move it, and the helicopter moves in the corresponding direction. Your left hand holds the *collective pitch lever*. Pull up on this and you increase the pitch on all the blades, generating lift. At the end of the pitch lever is the *throttle*. Finally you have two *foot pedals*, which vary the amount of tail rotor thrust and so help turn the helicopter.

"Easy!," you think. "Gently lower the collective pitch lever and you'll descend gracefully to the ground, a hero." However, when you try it, you discover that life isn't that simple. The helicopter's nose drops, and you start to spiral down to the left. Suddenly you discover that you're flying a system where every control input has secondary effects. Lower the left-hand lever and you need to add compensating backward movement to the right-hand stick and push the right pedal. But then each of these changes affects all of the other controls again. Suddenly you're juggling an unbelievably complex system, where every change impacts all the other inputs. Your workload is phenomenal: your hands and feet are constantly moving, trying to balance all the interacting forces.

Helicopter controls are decidedly not orthogonal.

Benefits of Orthogonality

As the helicopter example illustrates, nonorthogonal systems are inherently more complex to change and control. When components of any system are highly interdependent, there is no such thing as a local fix.

Eliminate Effects Between Unrelated Things

We want to design components that are self-contained: independent, and with a single, well-defined purpose (what Yourdon and Constantine call *cohesion* in *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design [YC79]*). When components are isolated from one another, you know that you can change one without having to worry about the rest. As long as you don't change that component's external interfaces, you can be confident that you won't cause problems that ripple through the entire system.

You get two major benefits if you write orthogonal systems: increased productivity and reduced risk.

Gain Productivity

- Changes are localized, so development time and testing time are reduced. It is easier to write relatively small, self-contained components than a single large block of code. Simple components can be designed, coded, tested, and then forgotten—there is no need to keep changing existing code as you add new code.

- An orthogonal approach also promotes reuse. If components have specific, well-defined responsibilities, they can be combined with new components in ways that were not envisioned by their original implementors. The more loosely coupled your systems, the easier they are to reconfigure and reengineer.
- There is a fairly subtle gain in productivity when you combine orthogonal components. Assume that one component does M distinct things and another does N things. If they are orthogonal and you combine them, the result does $M \times N$ things. However, if the two components are not orthogonal, there will be overlap, and the result will do less. You get more functionality per unit effort by combining orthogonal components.

Reduce Risk

An orthogonal approach reduces the risks inherent in any development.

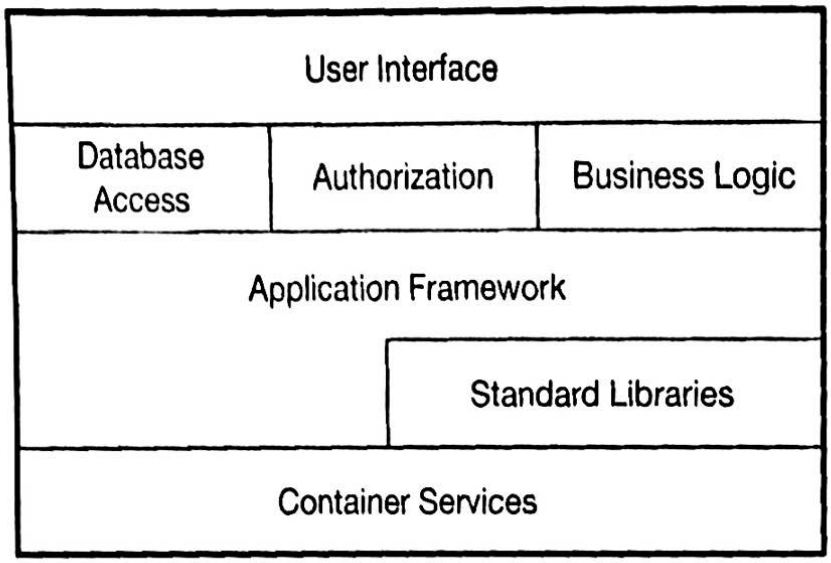
- Diseased sections of code are isolated. If a module is sick, it is less likely to spread the symptoms around the rest of the system. It is also easier to slice it out and transplant in something new and healthy.
- The resulting system is less fragile. Make small changes and fixes to a particular area, and any problems you generate will be restricted to that area.
- An orthogonal system will probably be better tested, because it will be easier to design and run tests on its components.
- You will not be as tightly tied to a particular vendor, product, or platform, because the interfaces to these third-party components will be isolated to smaller parts of the overall development.

Let's look at some of the ways you can apply the principle of orthogonality to your work.

Design

Most developers are familiar with the need to design orthogonal systems, although they may use words such as *modular*, *component-based*, and *layered* to describe the process. Systems should be composed of a set of cooperating modules, each of which implements functionality independent of the others. Sometimes these components are organized into layers, each providing a level of abstraction. This layered approach is a powerful way to design orthogonal systems. Because each layer uses only the abstractions provided by the layers below it, you have great flexibility in changing underlying

implementations without affecting code. Layering also reduces the risk of runaway dependencies between modules. You'll often see layering expressed in diagrams:



There is an easy test for orthogonal design. Once you have your components mapped out, ask yourself: *If I dramatically change the requirements behind a particular function, how many modules are affected?* In an orthogonal system, the answer should be "one."[4] Moving a button on a GUI panel should not require a change in the database schema. Adding context-sensitive help should not change the billing subsystem.

Let's consider a complex system for monitoring and controlling a heating plant. The original requirement called for a graphical user interface, but the requirements were changed to add a mobile interface that lets engineers monitor key values. In an orthogonally designed system, you would need to change only those modules associated with the user interface to handle this: the underlying logic of controlling the plant would remain unchanged. In fact, if you structure your system carefully, you should be able to support both interfaces with the same underlying code base.

Also ask yourself how decoupled your design is from changes in the real world. Are you using a telephone number as a customer identifier? What happens when the phone company reassigns area codes? Postal codes, Social Security Numbers or government IDs, email addresses, and domains are all external identifiers that you have no control over, and could change at any time for any reason. *Don't rely on the properties of things you can't control.*

4. In reality, this is naive. Unless you are remarkably lucky, most real-world requirements changes will affect multiple functions in the system. However, if you analyze the change in terms of functions, each functional change should still ideally affect just one module.