**Design Principles:**

**Code to interfaces:** As you recall, an interface is a kind of a contract between two objects.When a class implements an interface, it’s saying in effect: “My objects can speak your language.” Another huge benefit of interface is **polymorphism.** Many classes can implement the same interface. The calling object doesn’t care who it’s talking to as long as the contract is upheld. For example, the web container can use any component that implements the Servlet interface.

**Separation of Concerns & Cohesion:** We all know that when we specialize the capabilities of our software components, they get easier to create, maintain, and reuse. Natural fallout of separating concerns is that **cohesion** tends to increase. Cohesion means the degree to which a class is designed for one, *cohesive,* task or purpose.

**Hide Complexity:** Hiding complexity often goes hand in with separating concerns. For instance if your system needs to communicate with a lookup service, it’s best to hide the complexity of that operation in a single component, and allow all the other components that need access to the lookup service to use that specialized component. This approach simplifies all of the system components that are involved.

**Loose Coupling:** By their very nature, OO systems involve objects talking to each other. By coding to interfaces, you can reduce the number of things that one class needs to know about another class to communicate with it. The less two classes know about each other, the more loosely coupled they are to each other. A very common approach when class A wants to use methods in class B is to create an interface between the two. Once class B implements this interface, class A can use an updated class B or even an entirely different class as long as it upholds the contract of the interface.

**Remote Proxy:** Today, when a web site grows, the answer is to lash together more servers, as opposed to upgrading a single, huge, monolithic server. The outcome is that Java objects on different machines, in their own separate heaps, have to communicate with each other.

Leveraging the power of interface, a remote proxy is an object local to the “client” object that Java that *pretends* to be a remote object. (The proxy is remote in that it is remote from the object it is emulating.) The client object communicates with the proxy, and the proxy handles all the networking complexities of communicating with the actual “service” object. As far as the client object is concerned, it’s talking to a local object.

**Increase Declarative Control:** Declarative control over applications is a powerful feature of J2EE Containers. Most commonly, this declarative control is implemented using the application’s deployment descriptor (or DD). Modifying the DD gives us the power to change system behaviors without changing code. The DD is an XML file that can be maintained and updated by non-programmers. The more that we write our web applications to leverage the power of the DD, the more abstract and generic our code becomes.