## Multitier App Architecture (Architectural Design)

Multitier apps divide functionality into separate tiers (that is logical grouping of functionality). Although tiers can be located on the same computer, the tiers of web-based apps commonly reside on separate computers for security and scalability.

* The **bottom tier** (also called the **information tier**) maintains the app’s data. This tier typically store data in a relational database management system. This tier can contain multiple databases, which together comprise the data needed for an app.
* The **middle tier** implements *business logic, controller logic, and presentation logic* to control interactions between the app’s clients and its data. The middle tier acts as an intermediary between data in the information tier and app’s clients. The middle tier *controller logic* processes client requests and retrieved data from the database. The middle tier *presentation logic* then processes data from the information tier and presents the content to the client. For example, web apps present data to clients as web pages. *Business logic* in the middle tier enforces *business rules* and ensures that data is reliable before the app updates the database or presents the data to users. Business rules dictate how clients can and cannot access app’s data, and how apps process data.
* The **top tier,** or **client tier**, is the app’s user interface, which gathers input and displays output. Users interact directly with the app through the user interface (typically viewed in a web browser), keyboard and mouse. In response to the user actions (for example, clicking a hyperlink), the client tier interacts with the middle tier to make requests and to retrieve data from the information tier. The client tier then displays to the user the data retrieved from the middle tier. The client tier never directly interacts with the information tier.

## Fundamental Detailed Design Principles

* **Coupling** is a qualitative measure of how closely the classes in a design class diagram are linked. A simple way to think about coupling is as the number of navigation arrows on the design class diagram. *Low coupling is usually better for a system than high coupling*. In other words, fewer navigation visibility arrows indicate that a system is easier to understand and maintain. So, why the high coupling is bad? The main reason is that a change in one class ripples through the entire system. Therefore, experienced analysts make every effort to simplify coupling and reduce ripple effects in the design of a new system.
* **Cohesion** refers to the consistency of the functions within a single class. Cohesion is a qualitative measure of the focus or unity of purpose within a single class. Classes with low cohesion have several negative effects. First, they are hard to maintain. Second, it is hard to reuse such classes. Finally, classes with low cohesion are usually difficult to understand, their functions are intertwined and their logic is complex. *High cohesion is the most desirable*.
* **Protection from variation** – is the idea that the parts of a system that are unlikely to change should be segregated (or protected) from those that will. As you design systems, *you should try to isolate the parts that will change from those that are more stable*. Protection from variation is a principle that drives the multitier design pattern.
* **Indirection** – is a design principle in which an intermediate class (or component) is placed between two classes (or system components) to decouple but still link them. Inserting an intermediate object allows any variations in one system to be isolated in that intermediate object. Indirection is also very useful for many corporate security systems (for example, firewalls and proxy servers).
* **Open-Close Principle**. Software entities (classes, modules, functions, etc.) should be open for extension, but close for modification.
* **Substitution Principle**. Subtypes must be substitutable for their base types (polymorphism).
* **Dependency Inversion Principle**. Abstractions should not depend upon details. Details should depend upon the abstractions.
* **Release-Reuse Equivalency Principle**. The granule of reuse is the granule of release.
* **Common Reuse Principle**. The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.
* **Acyclic Dependencies Principle**. Allow no cycles in the package dependency diagram.