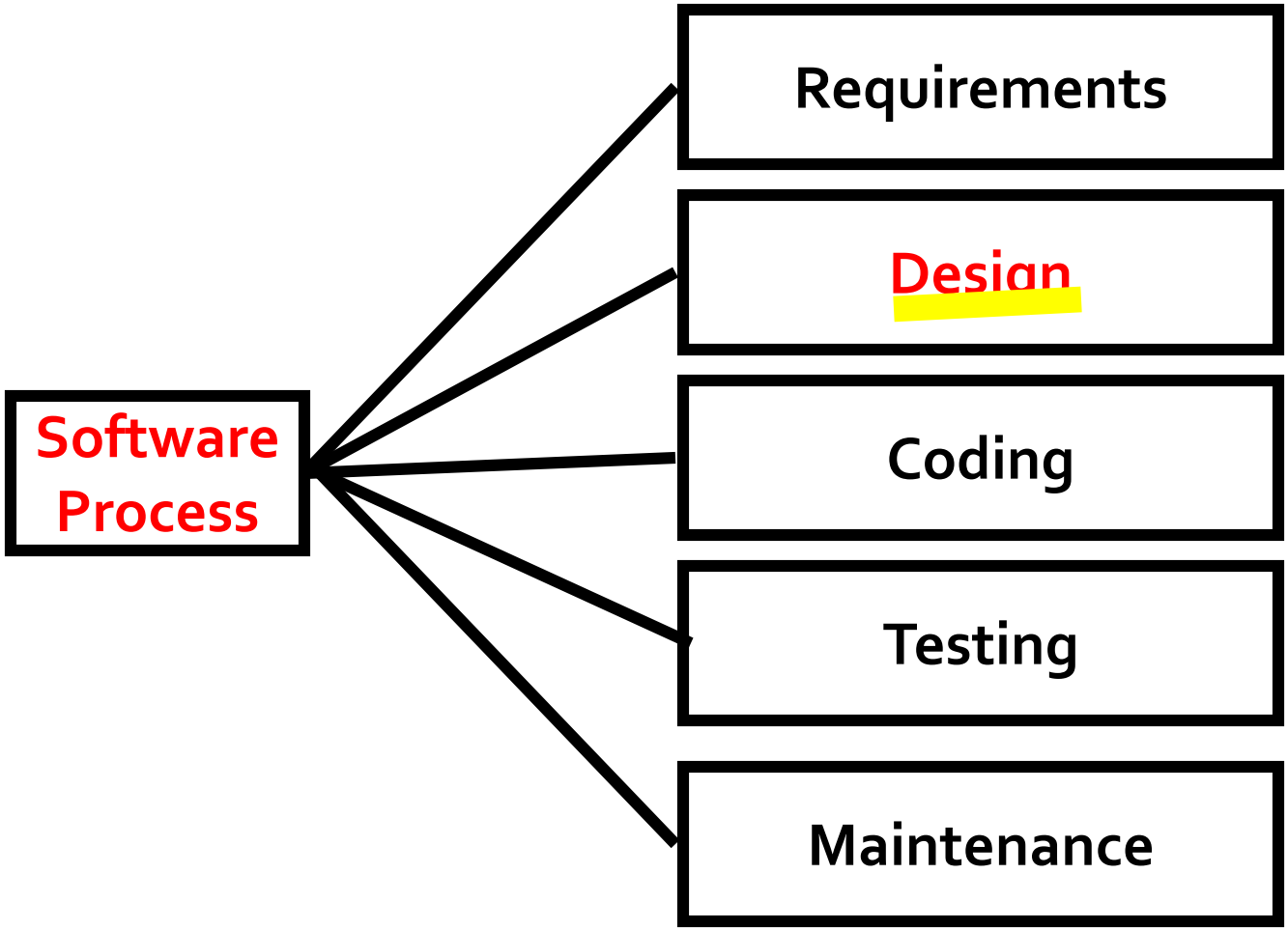
Software Design

Design Principles & Architectural Design

Software Design Concepts

Activities in the Software Process



Software Design

- To produce an implementable solution to a problem.
- An iterative process through which a customer's requirements are translated into a blue print for constructing the software code.
- All design work products must be traceable to the software requirements document (SRS).

√ Software Design

- The process of defining the architecture, components, interfaces and other characteristics of a system and the result of that process.
- The activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction.

Software Design

- Many software projects iterate through the analysis and design phases several times.
- Pure separation of analysis and design may not always be possible or desirable.
- Design is not coding, coding is not design.
- Still a creative process!

Software Design

- The design must implement all of the requirements.
- The design must be readable, understandable guide.
- The design should provide a complete picture of the software code.

Software Design

- All design work products must be reviewed for quality.
- The quality of the design is assessed by formal technical reviews/design walkthroughs.

Software Design

- The design should be traceable to the analysis model.
- The design should "minimize the intellectual distance" between the software code and the problem as it exists in the real world.
- The design should be structured to accommodate change.

√ Software Design Principles

- Abstraction
- Decomposition & Modularization
- Encapsulation & Information Hiding
- Software Architecture
- Software Design Pattern

Abstraction

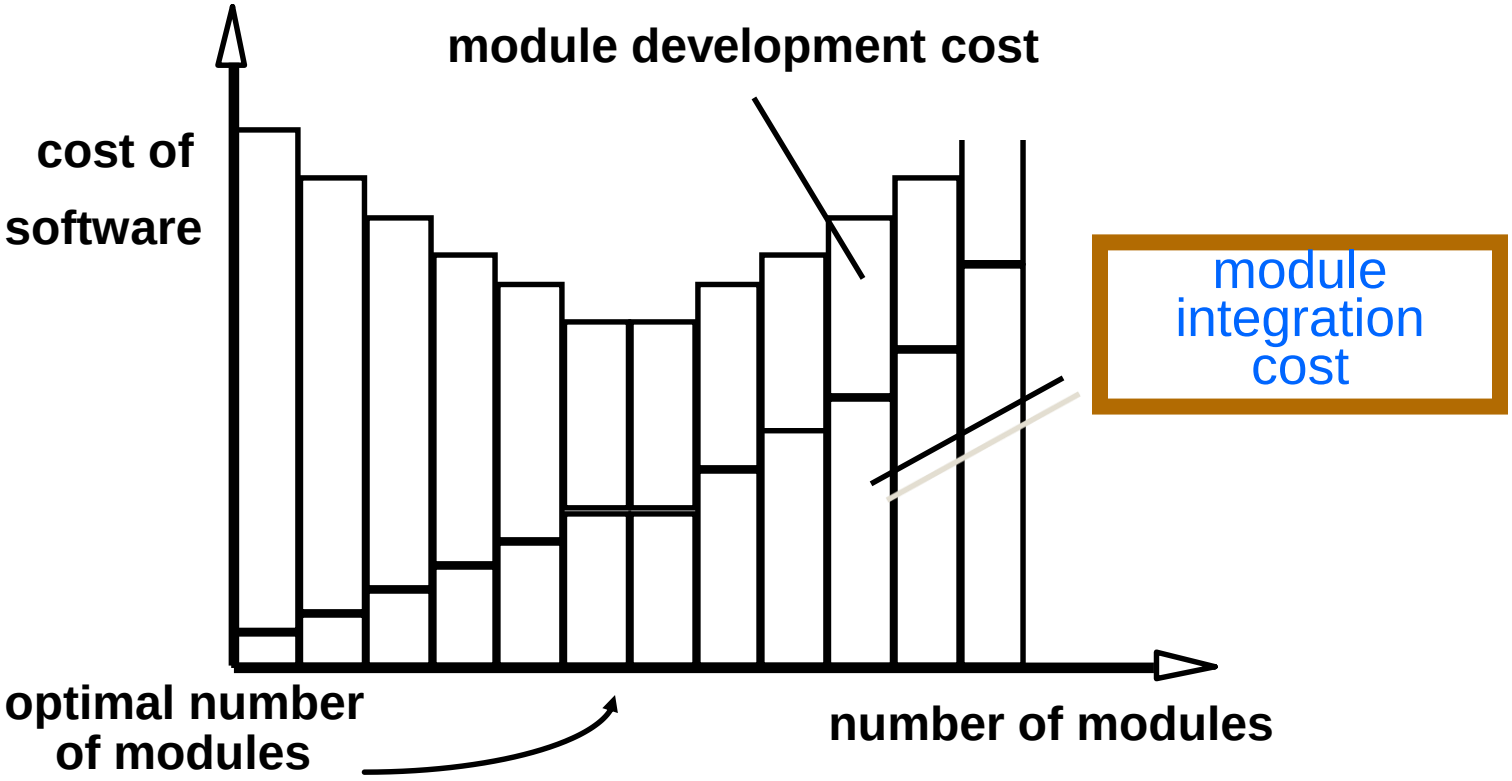
- A view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information
  - Data abstraction
  - Procedural abstraction
  - Control abstraction

Decomposition & Modularization

- Divided into software design units or modules components
  - Compartmentalization of data and function.
  - Computer systems are not monolithic: they are usually composed of multiple, interacting modules.

Modularization: Trade-offs

What is the "right" number of modules for a specific software design?



Modularization: Functional Independence

- Software design units are modules/components with single-minded function and aversion to excessive interactions with other modules.
- Each software design unit – module/component - addresses a specific subfunction of requirements and has a simple interface!

√Coupling & Cohesion: Functional Independence

- Measured using two qualitative criteria:
  - Cohesion
    - A measure of the relative functional strength of a software unit.
    - A measure of the independence among modules in a program.
  - Coupling
    - A measure of the relative interdependence among software units.
    - A measure of the strength of association of the elements within a module.

√ Cohesion

- An attribute of a software unit of high- or detail-level design that identifies the degree to which the elements within that software unit belong or are related together!

Cohesion

- A measure of how well a component "fits together".
- The degree to which a component/module performs one and only one function.
- A component encapsulates only attributes and operations that are closely related!

Types of Cohesion

- 7 Categories of cohesion:
  - Coincidental cohesion
  - Logical cohesion
  - Temporal cohesion
  - Procedural cohesion
  - Communicational cohesion
  - Sequential cohesion
  - Functional cohesion

Types of Cohesion

- Coincidental cohesion
  - Multiple unrelated tasks/elements in a design unit.
  - Harder to understand and not reusable

Types of Cohesion

- Logical cohesion
  - Similar related tasks/elements, but still relatively independent of each other.
- Temporal cohesion
  - A series of tasks/elements that are related by time.
  - Components which are activated at the same time are grouped.

Types of Cohesion

- Procedural cohesion
  - Tasks/elements that are procedurally related.
  - Related in terms of some control sequence.
- Communicational cohesion
  - Related by a sequence of activities targeted on the same data.
  - All ops accessing the same data within one component

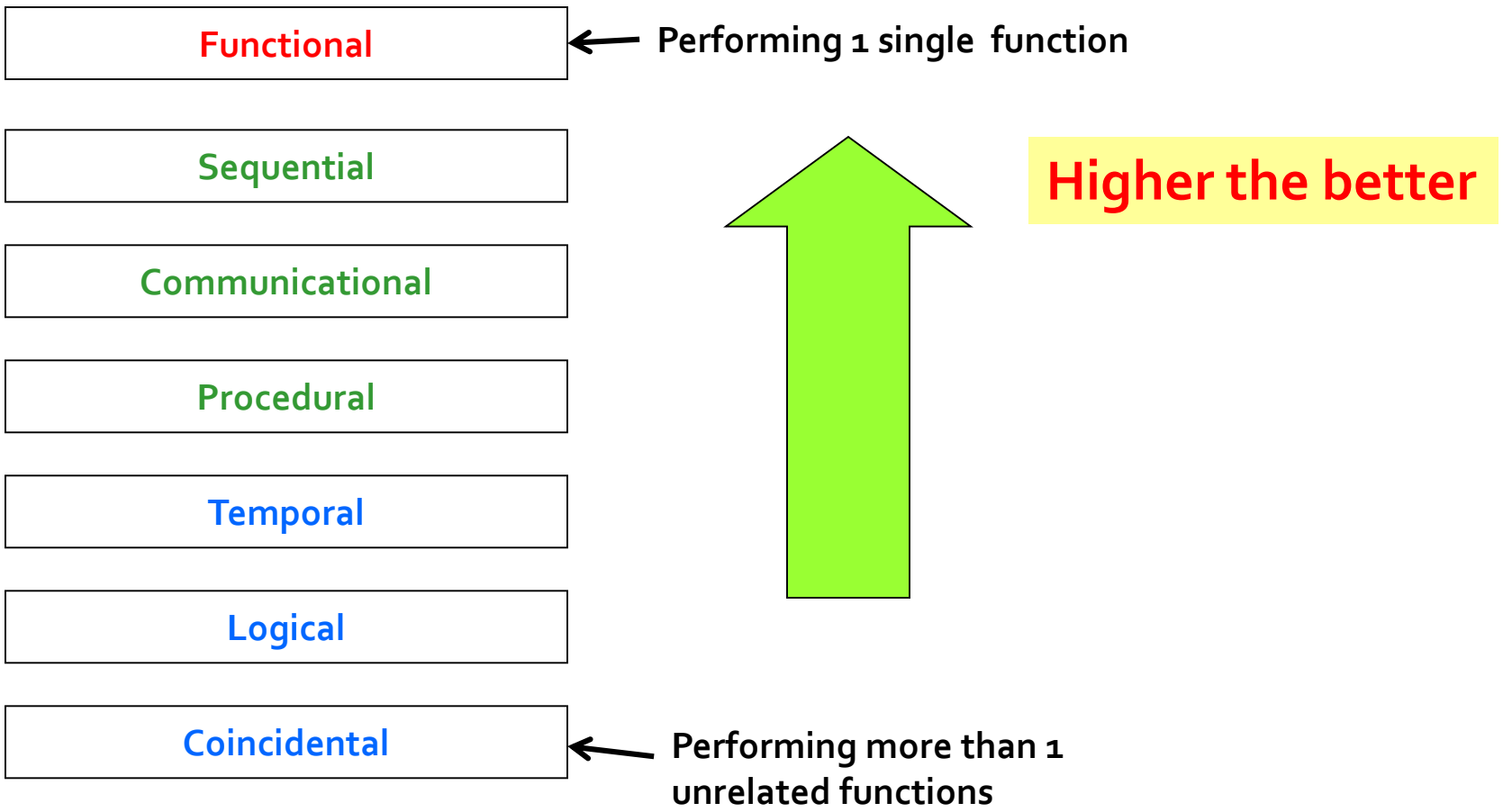
Types of Cohesion

- Sequential cohesion
  - Output of one is input of next.
  - The output for one part of a component is the input to another part.

Types of Cohesion

- Functional cohesion
  - Performs one main activity.
  - Achieves one goal.
  - Only one computation by operations
  - Usually reusable in other contexts and maintenance easier.

Types of Cohesion



Cohesion in OO Systems

- Method cohesion
- Class cohesion
- Inheritance cohesion

Cohesion in OO Systems

- Method cohesion
  - Same as cohesion in functional modules.
  - Why the different code elements of a method are together within this method?
- Class cohesion
  - Why different attributes and methods are together in this class?
- Inheritance cohesion
  - Why classes are together in a class hierarchy?

Cohesion

- High (strong) cohesion!
  - Cohesion is a desirable design component attribute as when a change has to be made, it is localized in a single cohesive component.
- Design Principle: Increase cohesion where possible!

√ Coupling

- An attribute that addresses the degree of interaction and interdependence between two software units!

Coupling

- A measure of the strength of the inter-connections between system components.
- The degree to which a module is "connected" to other modules in the system.

Types of Coupling

- 6 Categories of coupling:
  - Content coupling
  - Common coupling
  - Control coupling
  - Stamp coupling
  - Data coupling
  - No Coupling

Types of Coupling

- Content coupling
  - Two units access each other's internal data.
  - One component modifies data of another component.
  - Tightest coupling

Types of Coupling

- Common coupling
  - Two units refer to the same global variable.
- Control coupling
  - One software unit passes a control information/flag and explicitly influences the logic of another software unit.

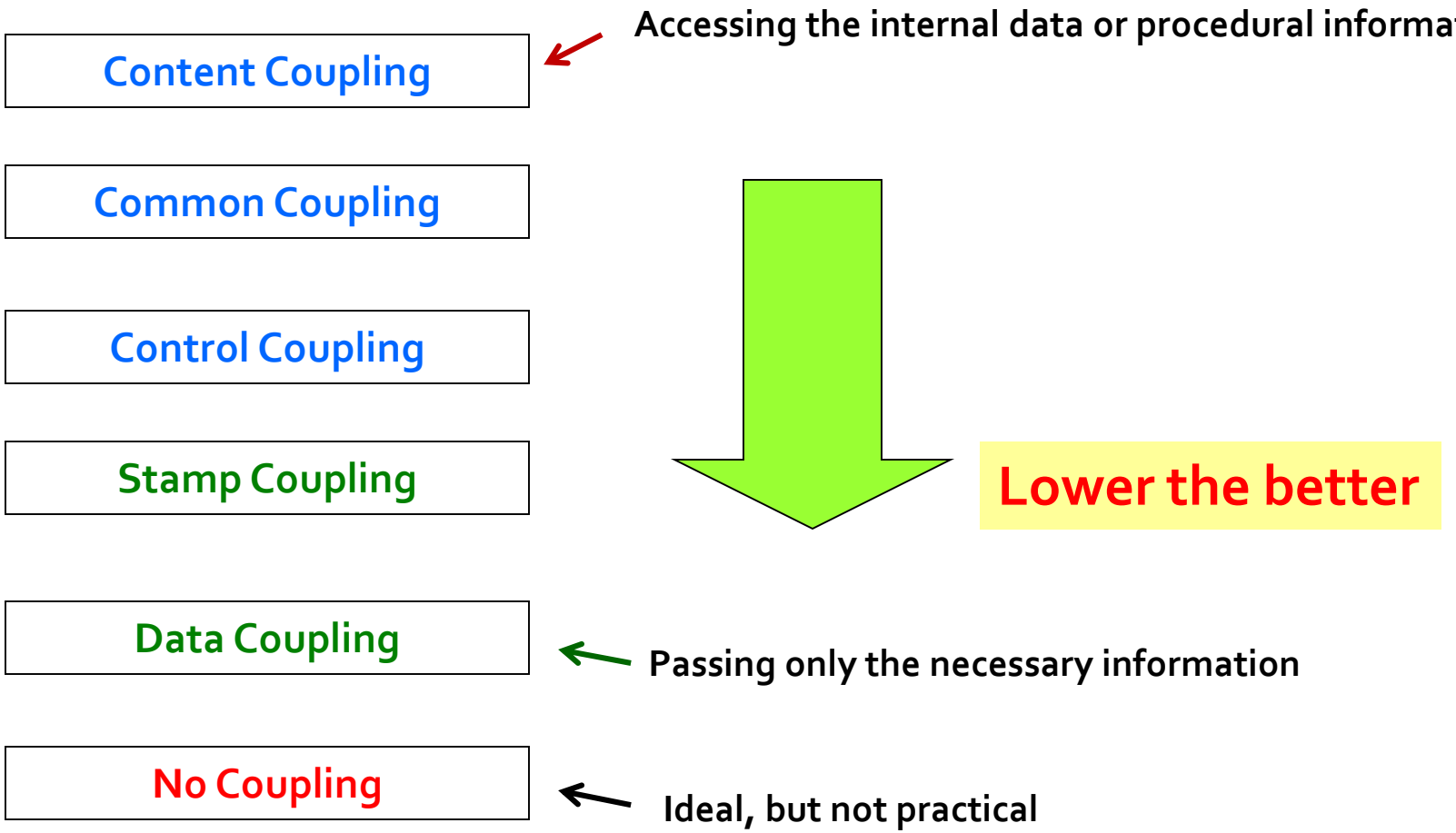
Types of Coupling

- Stamp coupling
  - A software unit passes a group of data to another software unit.
  - Passes more than necessary info
  - A lesser version of data coupling
- Data coupling
  - Only the needed data are passed between software units.

Types of Coupling

- No Coupling
  - Ideal, but not practical!

Types of Coupling



Coupling in OO Systems

- Interaction coupling
- Component coupling
- Inheritance coupling

Coupling in OO Systems

- Interaction coupling
  - Method of a class invoke methods of other classes.
  - Similar to coupling between functional modules.
- Component coupling
  - Interaction between two classes where a class has variables of the other class.
- Inheritance coupling
  - Inheritance relationship between classes.

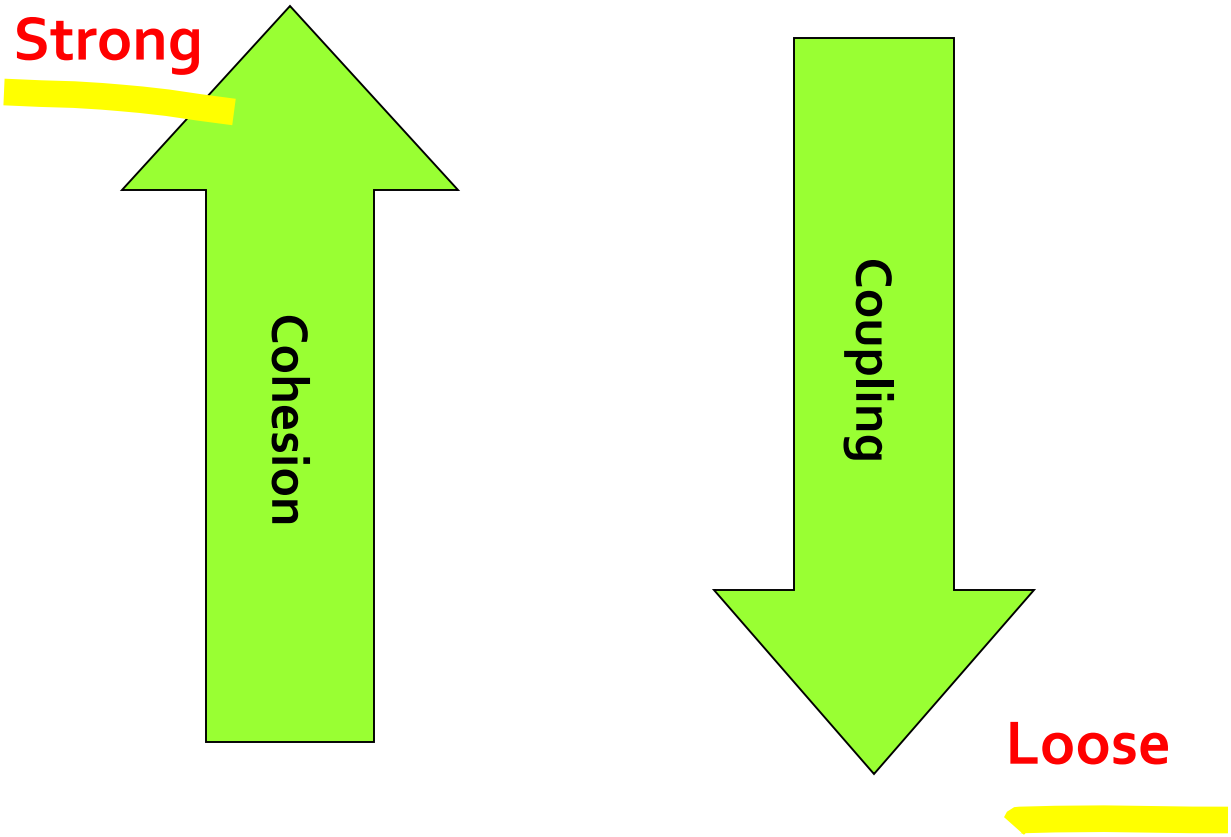
Coupling

- Low (Loose) coupling!
  - Component changes are unlikely to affect other components.
- Design Principle: Reduce coupling where possible!

√ Effective Modular Design

- Functional independence is the goal for each module.
- This is more likely when modules have single purposes (high cohesion) and rely on their own resources for data and control information (low coupling).

High Cohesion & Low Coupling



Effective Modular Design

- Every module should communicate with as few others as possible.
- If any two modules communicate, they should exchange as little information as possible.

Encapsulation & Information hiding

- Encapsulation/information hiding means grouping and packaging the elements and internal details of an abstraction and making those details inaccessible.
- Separation of Interfaces and Implementations

Why Information Hiding?

- Reduces the likelihood of "side effects"
- Limits the global impact of local design decisions
- Emphasizes communication through controlled interfaces
- Discourages the use of global data
- Leads to encapsulation—an attribute of high quality design
- Results in higher quality software

√ Software Architecture

- The fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.
- The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.

Why Software Architecture?

- To allow the software engineer to view and evaluate the system as a whole before moving to component design.
  - Analyze the effectiveness of the design in meeting its stated requirements,
  - Consider architectural alternatives at a stage when making design changes is still relatively easy, and
  - Reduce the risks associated with the construction of the software.

√ Software Architectural Styles

- Repeatable patterns that are commonly encountered in the design of families of similar systems.
- In essence, the design should have the ability to reuse architectural building blocks.
- Macro-architectural Patterns

√ Software Design Patterns

- A pattern is a common solution to a common problem in a given context.
- Proven design solutions for a variety of problems.
- A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context.
- Micro-architectural Patterns

Design Patterns

- Many design patterns have been systematically documented for all software developers to use.
- A good pattern should
  - Be as general as possible
  - Contain a solution that has been proven to effectively solve the problem in the indicated context.
- Studying patterns is an effective way to learn from the experience of others!

Architectural Styles vs Design Patterns

- Architectural styles can be viewed as patterns describing the high-level organization of software (their macro-architecture)!
- Design patterns can be used to describe details at a lower, more local level (their micro-architecture)!

√ Software Design Notations

- Textual descriptions
- Models – Diagrammatical descriptions
  - Structural modeling (static view)
  - Behavioral modeling (dynamic view)

Software Design Notations

- Structural modeling description (static view)
  - Class diagrams
  - Object diagrams
  - Component diagrams
  - Deployment diagrams
  - Structure charts

Software Design Notations

- Behavioral modeling description (dynamic view)
  - Activity diagrams
  - Interaction diagrams – Sequence diagrams and Collaboration diagrams
  - Data Flow diagram
  - State Transition/ Statechart diagrams
  - Pseudocode and Program design languages

√The Software Design Process

- Architectural design
- Component-level design
- User-interface design
- Data design

Software Design Description (SDD)

Software Design Description (SDD)

TABLE OF CONTENTS

1. Architectural Design

Software Architectural Design

- Establishing the overall structure of a software system!
- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication.
- The output is a description of the software architecture

Architectural Design

- An early stage of the system design process.
- Represents the link between specification and design processes.
- It involves identifying major system components and their communications.

Advantages of Explicit Architecture

- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
  - The architecture may be reusable across a range of systems.

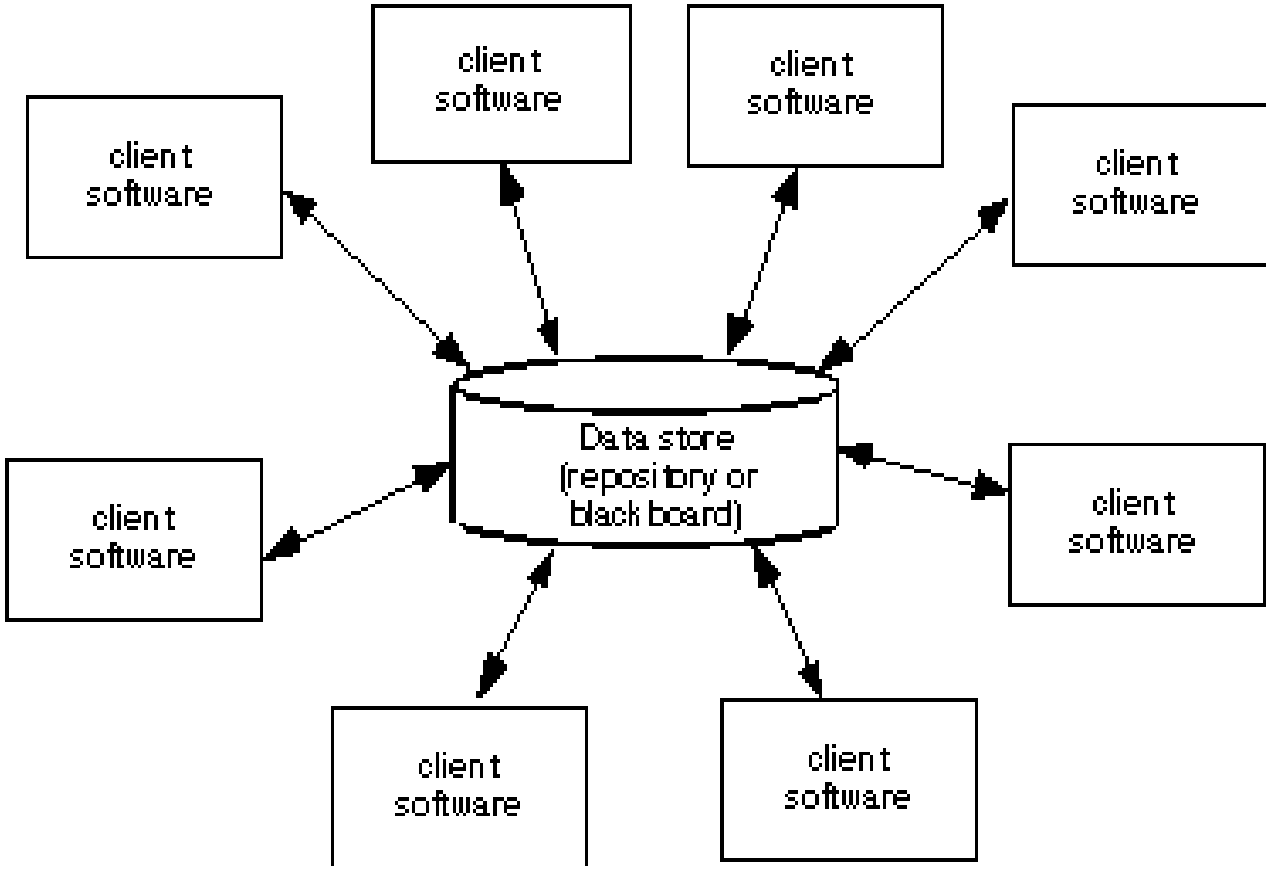
Architectural Styles

- The architectural model of a system may conform to a generic architectural model or style.
- An awareness of these styles can simplify the problem of defining system architectures.
- Architectural Patterns

√ Architectural Styles

- Data-centered architecture
- Data flow architecture
- Call and return architecture
- Layered architecture
- Client-server Architecture
- Three-tier Architecture
- Model-View-Controller Architecture
- Event-driven Architecture
- …

Data-centered Architecture



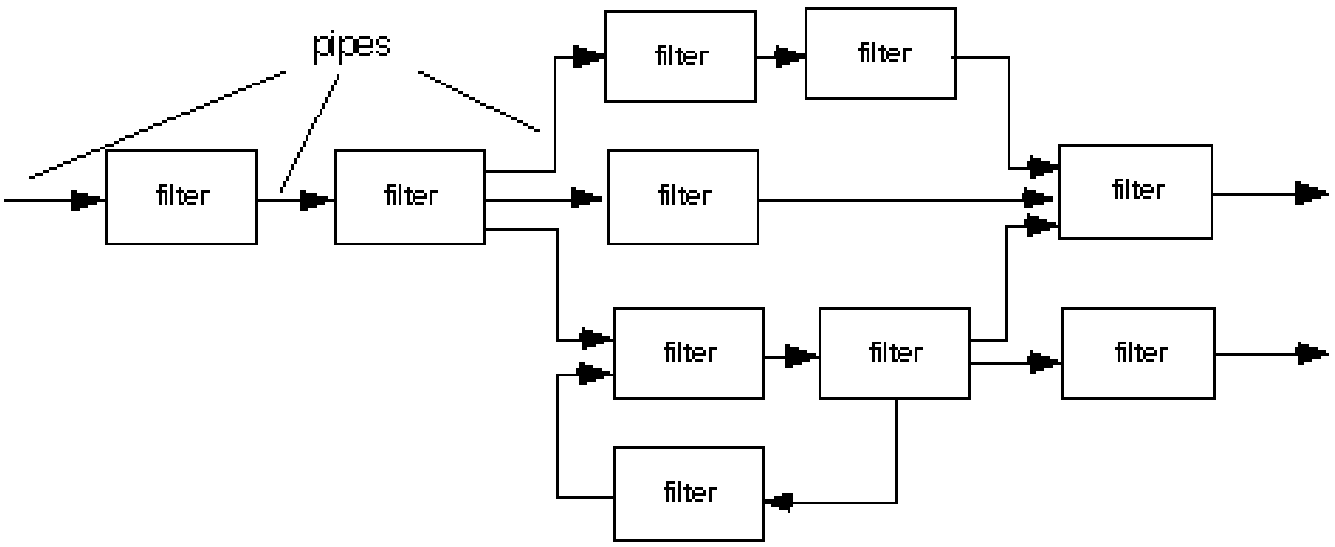
The Repository Model

- Database-centric style
- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

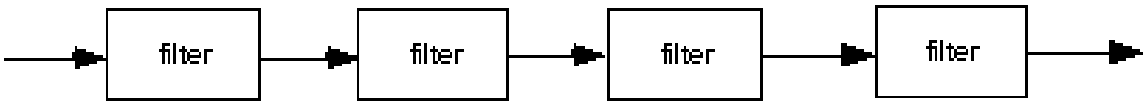
Repository Model Characteristics

- Efficient way to share large amounts of data.
- Sub-systems must agree on a repository data model. Inevitably a compromise.
- Difficult to distribute efficiently.

Data Flow Architecture



(a) pipes and filters

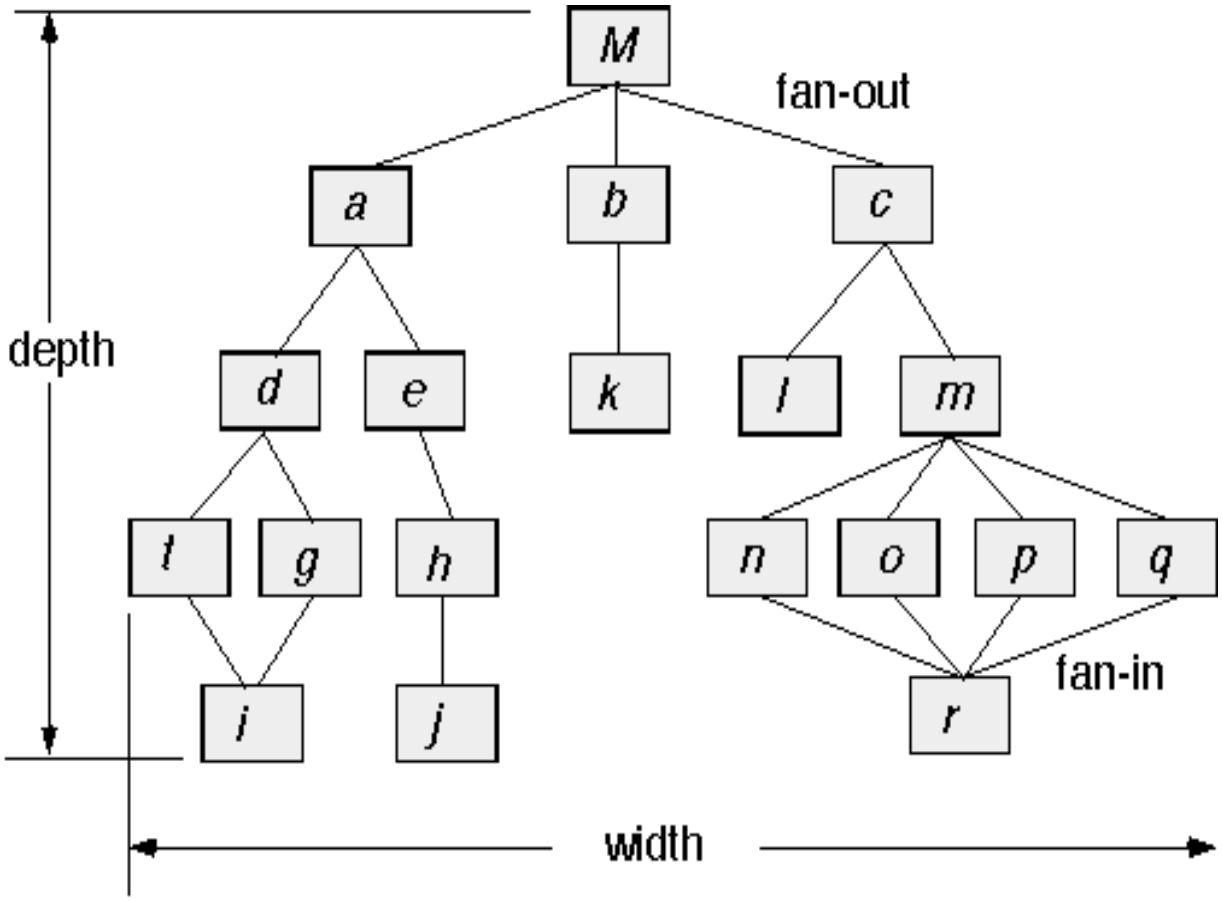


(b) batch sequential

The Data-flow Model

- Filters-and-Pipes style
- Pipes-and-Filters style
- Functional transformations process their inputs to produce outputs.
- When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

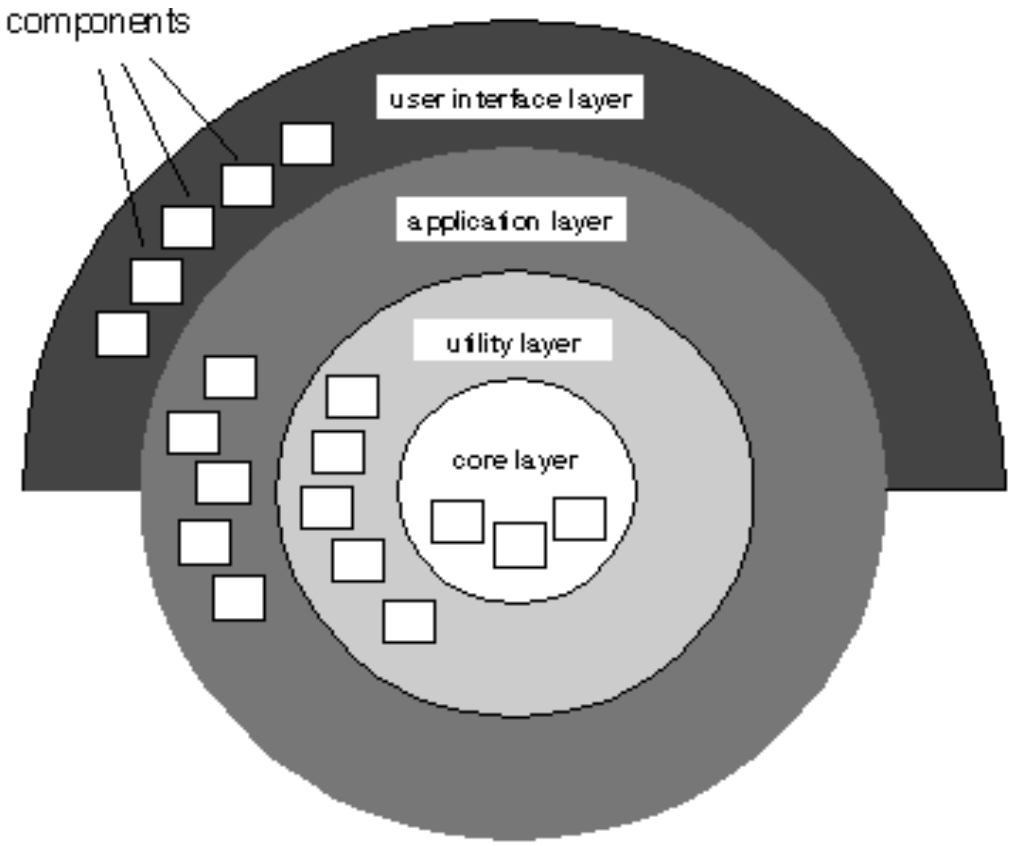
Call and Return Architecture



The Call-Return Model

- Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards.
- Applicable to sequential systems.

Layered Architecture



The Abstract Machine Model

- Used to model the interfacing of sub-systems.
- Organizes the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers.
- When a layer interface changes, only the adjacent layer is affected.

Distributed Systems

- Software that executes on more than one processor - on a loosely integrated group of cooperating processors linked by a network.
- Information processing is distributed over several computers (rather than confined to a single machine).

Distributed Systems

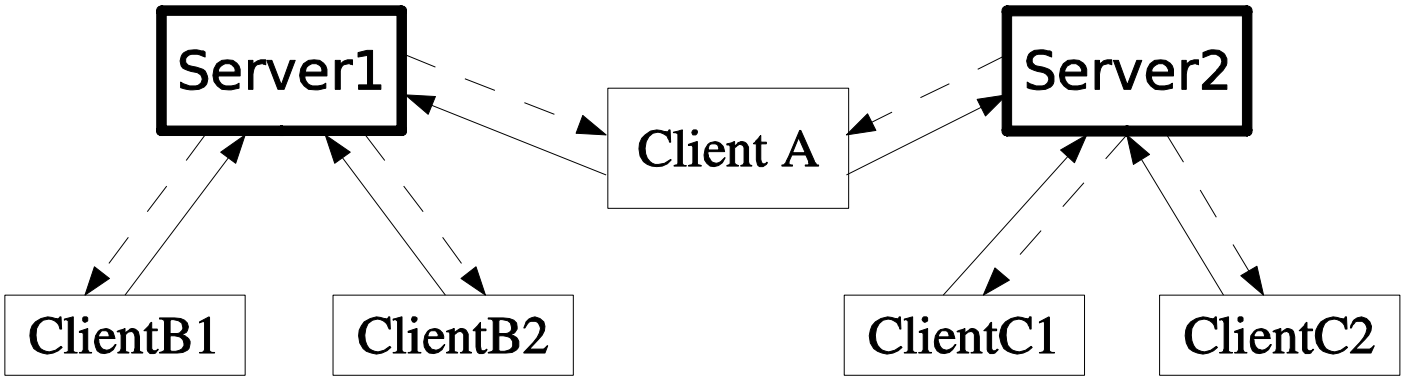
- Virtually all large computer-based systems are now distributed systems.
- Distributed software engineering is now very important!

Distributed Systems Architectures

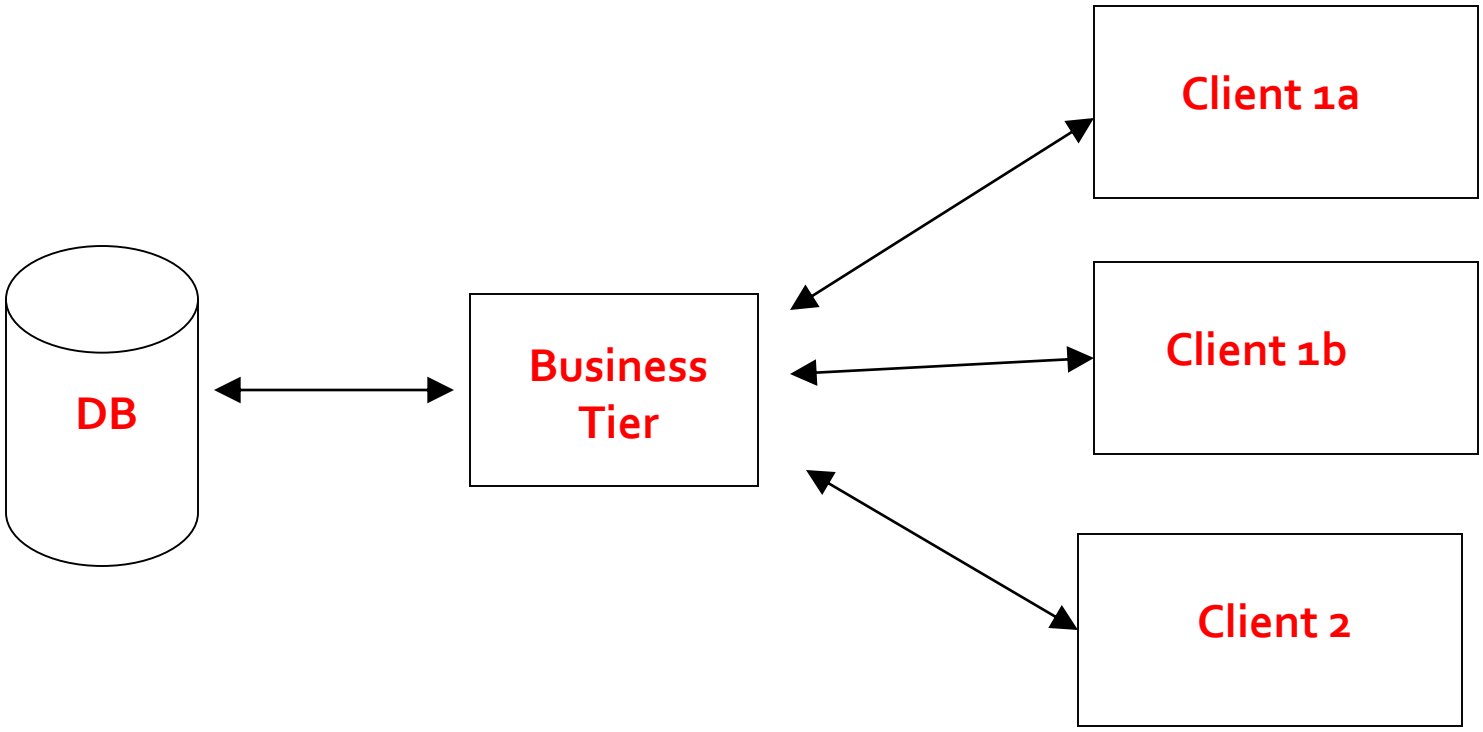
- Client-Server Architecture
  - Distributed services called on by clients.
  - Servers provide services.
- Distributed Object Architecture
  - No distinction between clients and servers.
  - Any object on the system may provide and use services from other objects.

Client-Server Architecture

- Client may connect to more than one server (servers are usually independent)

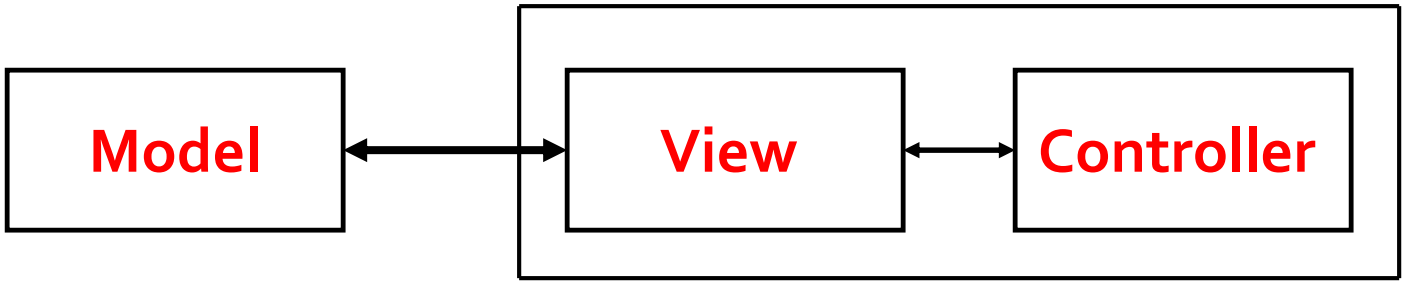


Three-Tier Client-Server Architecture

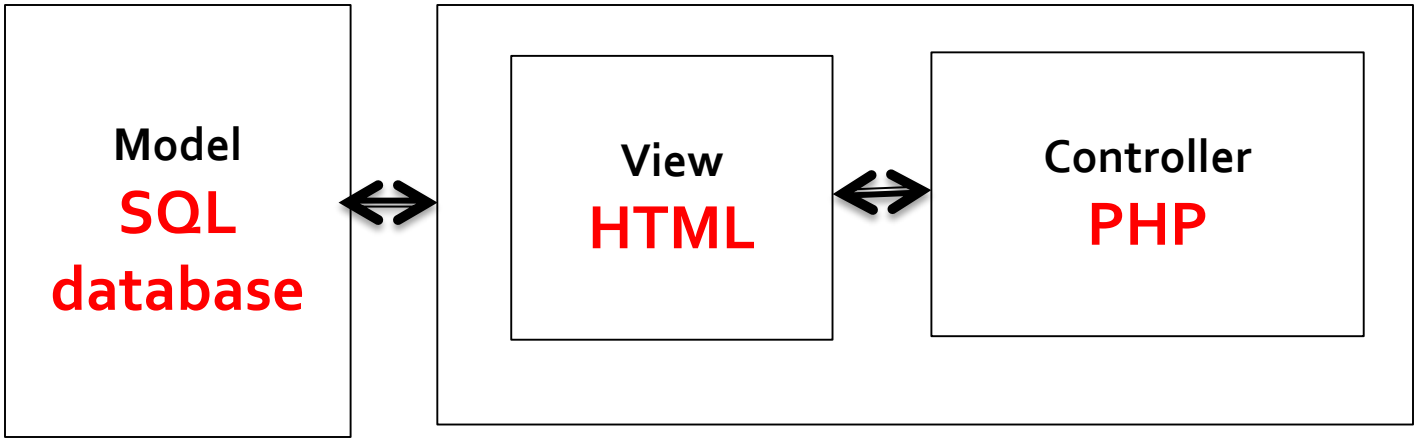


Model-View-Controller (MVC) Architecture

- Separates model (data) from view.
- Controller often integrated with view nowadays.
- Most of the internet web applications fall under this style!



Model View Control (MVC) Architecture



Event-Driven (Real-Time) Architecture

- System components react to externally generated events and communicate with other components with events.
- It is based on an event dispatcher which manages events and the functionalities which depends on those events.