# Chapter 4: Software Architecture

## Introduction to software architecture:

Software architecture is the process of converting software characteristics such as flexibility, scalability, feasibility, reusability, and security into a structured solution that meets the technical and the business expectations.

*Software architecture* is the process of designing the global organization of a software system, including dividing software into subsystems, deciding how these will interact, and determining their interfaces.

The architecture is the core of the design, so all software engineers need to understand it.

The architectural model will often constrain the overall efficiency, reusability and maintainability of the system.

Poor decisions made while creating this model will constrain subsequent design.

### Reasons needed to develop software architectural model

1. **To enable everyone to better understand the system**

   A good architectural model allows people to understand how the system as a whole works; it also defines the terms that people use when they communicate with each other about lower-level details.

2. **To allow people to work on individual pieces of the system in isolation**

   The architecture allows the planning and coordination of this distributed work. The architecture should provide sufficient information so that the work of the individual people or teams can later on be integrated to form the final system. It is for that reason that the interfaces and dynamic interactions among the subsystems are an important part of the architecture.

3. **To prepare for extension of the system**

   With a complete architectural model, it becomes easier to plan the evolution of the system. Subsystems that are envisioned to be part of a future release can be included in the architecture, even though they are not to be developed immediately. It is then possible to see how the new elements will be integrated, and where they will be connected to the system.

4. **To facilitate reuse and reusability**

   The architectural model makes each system component visible. This is an important benefit since it encourages reuse. By analyzing the architecture, identify components that have high potential reusability. Making the architecture as generic as possible is a key to ensuring reusability.

## Contents of good architectural model

An important challenge in architectural modeling is to produce a relevant and synthetic picture of a large and complex system.

To ensure the maintainability and reliability of a system, an architectural model must be designed to be stable.

The architecture should be expressed clearly enough that it can be used to communicate effectively with clients.

A system's architecture will often be expressed in terms of several different *views*. These can include:

- The logical breakdown into subsystems
- The interfaces among the subsystems
- The dynamics of the interaction among components at run time
- The data that will be shared among the subsystems
- The components that will exist at run time, and the machines or devices on which they will be located

## Developing an architectural model

**Start by sketching an outline of the architecture**

- Based on the principal requirements and use cases
- Determine the main components that will be needed
- Choose among the various architectural patterns
- *Suggestion*: have several different teams independently develop a first draft of the architecture and merge together the best ideas

**Refine the architecture**

- Identify the main ways in which the components will interact and the interfaces between them
- Decide how each piece of data and functionality will be distributed among the various components
- Determine if you can re-use an existing framework, if you can build a framework

**Consider each use case and adjust the architecture to make it realizable**

**Mature the architecture**

## Architecture Centric Process

Aspects of an architecture include static elements, dynamic elements, how those elements work together, and the overall architectural style that guides the organization of the system.

Architecture also addresses issues such as performance, scalability, reuse, and economic and technological constraints.

The Unified Process specifies that the architecture of the system being built, as the fundamental foundation on which that system will rest,must sit at the heart of project team's efforts to shape the system.

**Need of architecture centric:**

- Understanding the big picture
- Organizing development effort
- Facilitating the possibilities for reuse
- Guiding the use cases
- Evolving the system

### Describing an architecture using UML

All UML diagrams can be useful to describe aspects of the architectural model

Four UML diagrams are particularly suitable for architecture modelling:

- Package diagrams
- Subsystem diagrams
- Component diagrams
- Deployment diagrams

### Architectural Pattern

An **architectural pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design pattern but have a broader scope.

Allows you to design flexible systems using components where the components are as independent of each other as possible.

#### *Multi-Layer architectural pattern*

In layered systems each layer communicates only with the layers below it .Each layer has a well-defined API, defining the services it provides.
A complex system can be built by superimposing layers at increasing levels of abstraction. The Multi-Layer architectural pattern makes it possible to replace a layer by an improved version, or one with a different set of capabilities.
It is important to have a separate layer for the UI.

Layers immediately below the UI layer provide the application functions determined by the use-cases.
Bottom layers provide general services.

The most commonly found 4 layers of a general information system are as follows.

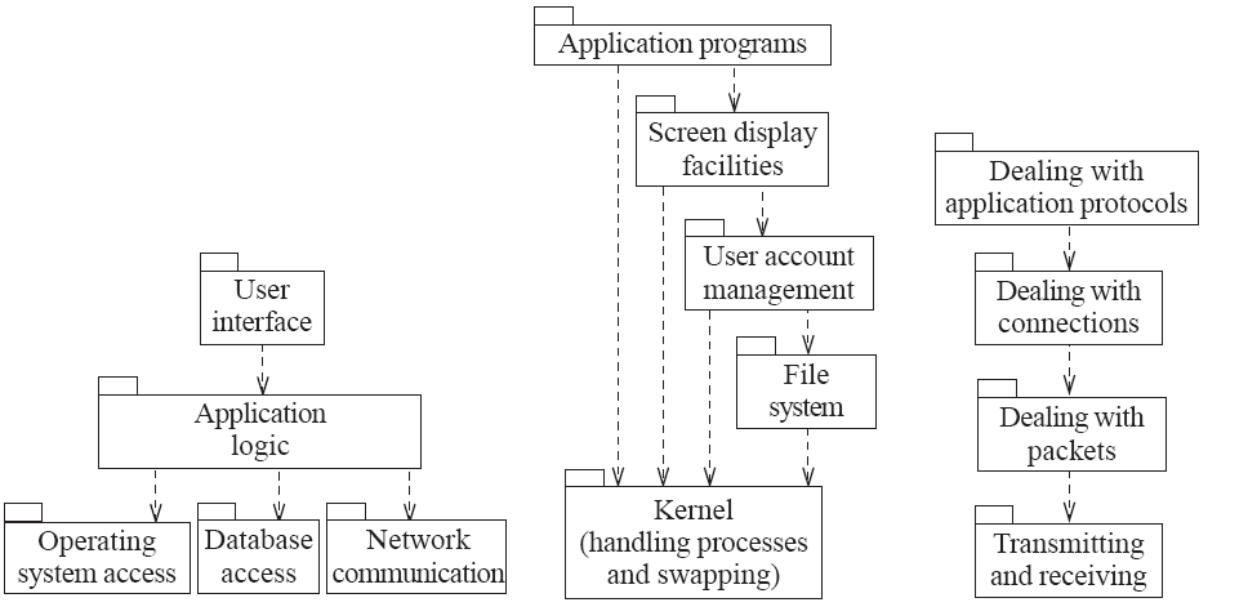
- **Presentation layer** (also known as **UI layer**)
- **Application layer** (also known as **service layer**)
- **Business logic layer** (also known as **domain layer**)
- **Data access layer** (also known as **persistence layer**)

Usage:

- General desktop applications.
- E commerce web applications.

**The Multi-Layer architectural pattern and design principles:**

***1. Divide and conquer*:** The layers can be independently designed.
**2. *Increase cohesion*:** Well-designed layers have layer cohesion.
**3. *Reduce coupling***: Well-designed lower layers do not know about the higher layers and the only connection between layers is through the API.
**4. *Increase abstraction*:** you do not need to know the details of how the lower layers are implemented.
**5. *Increase reusability*:** The lower layers can often be designed generically.
**6. *Increase reuse*:** You can often reuse layers built by others that provide the services you need.
***7. Increase flexibility:*** you can add new facilities built on lower-level services, or replace higher-level layers.
***8. Anticipate obsolescence:*** By isolating components in separate layers, the system becomes more resistant to obsolescence.
***9. Design for portability:*** All the dependent facilities can be isolated in one of the lower layers.
***10. Design for testability:*** Layers can be tested independently.
***11. Design defensively:*** The APIs of layers are natural places to build in rigorous assertion-checking.




(a) Typical layers in an application program

(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system

### *The Client/Server and other distributed architectural patterns*

This pattern consists of two parties; a ***server*** and multiple ***clients***. The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.

- There is at least one component that has the role of *server*, waiting for and then handling connections.
- There is at least one component that has the role of *client*, initiating connections in order to obtain some service.
- A further extension is the Peer-to-Peer pattern.
    - A system composed of various software components that are distributed over several hosts.

Usage

- Online applications such as email, document sharing and banking.

**The Distributed architecture and design principles**

**1. *Divide and conquer*:** Dividing the system into client and server processes is a strong way to divide the system.

— Each can be separately developed.

**2. *Increase cohesion*:** The server can provide a cohesive service to clients.

**3. *Reduce coupling*:** There is usually only one communication channel exchanging simple messages.

**4. *Increase abstraction*:** Separate distributed components are often good abstractions.

**6. *Increase reuse*:** It is often possible to find suitable frameworks on which to build good distributed systems

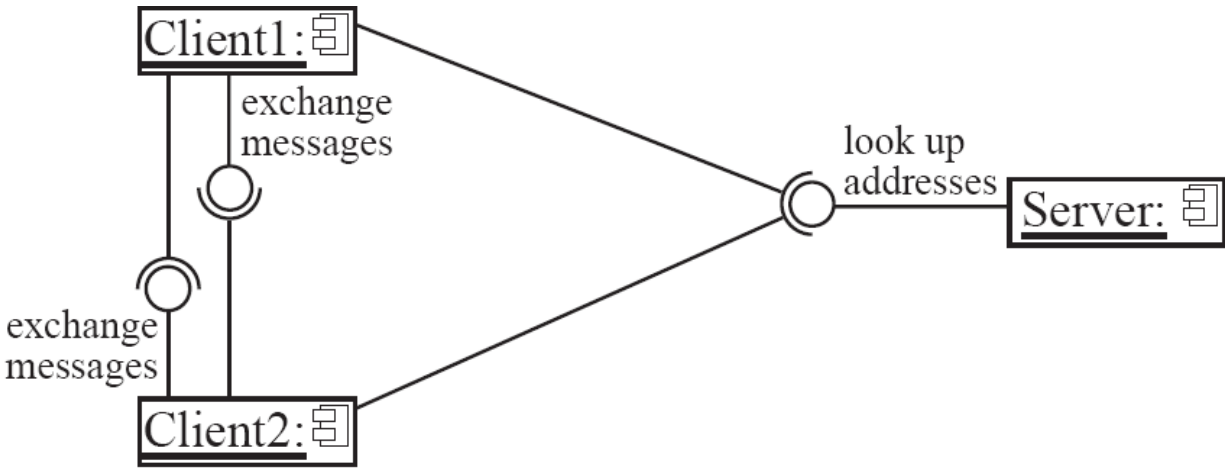
— However, client-server systems are often very application specific.

**7. *Design for flexibility*:** Distributed systems can often be easily reconfigured by adding extra servers or clients.

**9. *Design for portability*:** You can write clients for new platforms without having to port the server.

**10. *Design for testability*:** You can test clients and servers independently.

**11. *Design defensively*:** You can put rigorous checks in the message handling code.



### *The Model View Controller Architectural Patterns*

This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,

1. **model** — contains the core functionality and data, contains the underlying classes whose instances are to be viewed and manipulated.
2. **view** — displays the information to the user (more than one view may be defined), contains objects used to render the appearance of the data from the model in the user interface.
3. **controller** — handles the input from the user, contains the objects that control and handle the user's interaction with the view and the model.

This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. It decouples components and allows efficient code reuse.

Usage

- Architecture for World Wide Web applications in major programming languages.
- Web frameworks such as **Django** and **Rails**.

**The MVC architecture and design principles**

**1. *Divide and conquer*:** The three components can be somewhat independently designed.

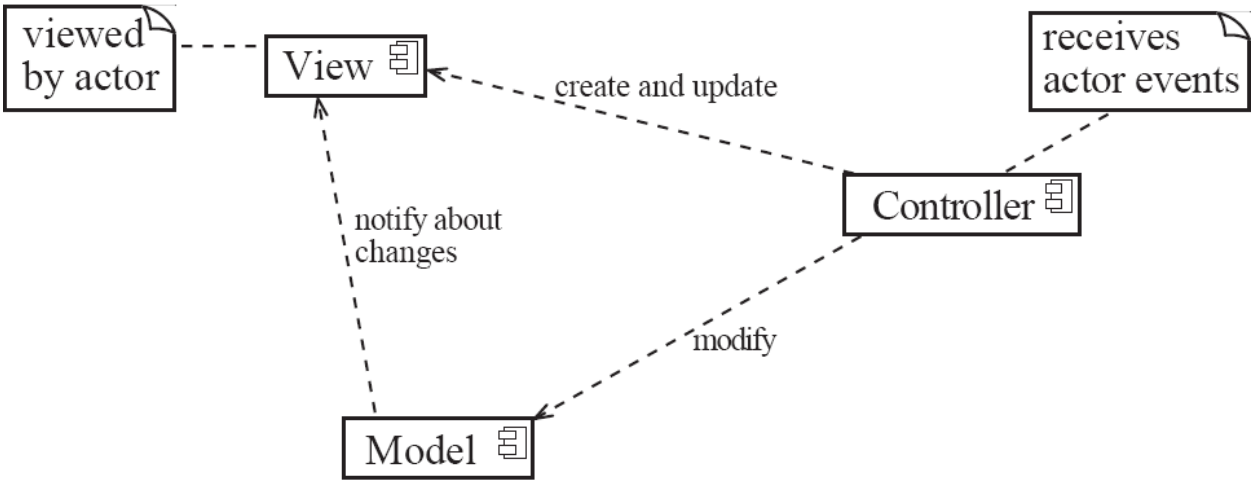
**2. *Increase cohesion*:** The components have stronger layer cohesion than if the view and controller were together in a single UI layer.

**3. *Reduce coupling*:** The communication channels between the three components are minimal.

**6. *Increase reuse*:** The view and controller normally make extensive use of reusable components for various kinds of UI controls.

**7. *Design for flexibility*:** It is usually quite easy to change the UI by changing the view, the controller, or both.

**10. *Design for testability*:** You can test the application separately from the UI.




**Example of MVC in web architecture**

- The *View* component generates the HTML code to be displayed by the browser.
- The *Controller* is the component that interprets 'HTTP post' transmissions coming back from the browser.
- The *Model* is the underlying system that manages the information.

### *The Service Oriented Architectural Patterns*

This architecture organizes an application as a collection of services that communicate with each other through well-defined interfaces.

In the context of the Internet, the services are called *Web services*.

A web service is an application, accessible through the Internet, which can be integrated with other services to form a complete system

The different components generally communicate with each other using open standards such as XML.

Web services can perform a wide range of tasks. Some may handle simple requests for information while others can perform more complex business processing. Enterprises can use Web services to automate and improve their operations.

For example, an electronic commerce application can make use of web services to:

- access the product databases of several suppliers;
- process credit cards using a Web service offered by a bank;
- arrange for delivery using a Web service offered by a shipping company.

The toughest challenge facing the developer of a Web service is *security*.

Other important considerations are reliability, availability and scalability of the Web service.

**The Service Oriented Architecture and design principles**

**1. *Divide and conquer*:** The application is made of independently designed services.

**2. *Increase cohesion*:** The Web services are structured as layers and generally have good functional cohesion.

**3. *Reduce coupling*:** Web-based applications are loosely coupled built by binding together distributed components.

**5. *Increase reusability*:** A Web service is a highly reusable component.

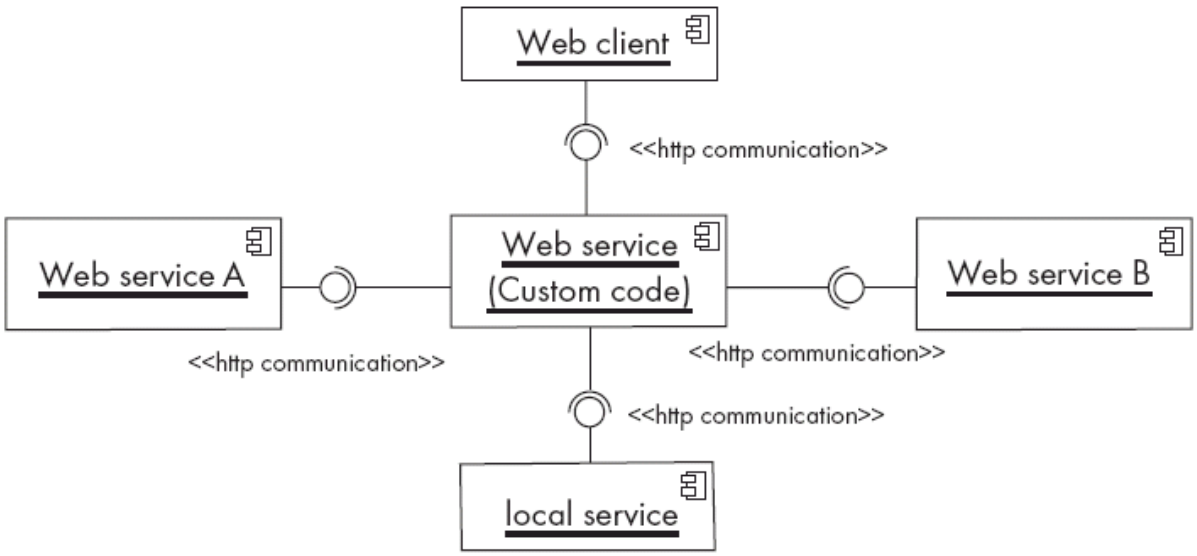
**6. *Increase reuse*:** Web-based applications are built by reusing existing Web services.

**8. *Anticipate obsolescence*:** Obsolete services can be replaced by new implementation without impacting the applications that use them.

**9. *Design for portability*:** A service can be implemented on any platform that supports the required standards.

**10. *Design for testability*:** Each service can be tested independently.

**11. *Design defensively*:** Web services enforce defensive design since different applications can access the service.




---

### *The Message Oriented Architectural Patterns*

Also known as Message-oriented Middleware (MOM).

This architecture is based on the idea that since humans can communicate and collaborate to accomplish some task by exchanging emails or instant messages, then software applications should also be able to operate in a similar manner i.e. the different sub-systems communicate and collaborate to accomplish some task only by exchanging messages.

The core of this architecture is an application-to-application messaging system

Senders and receivers need only to know what are the message formats

In addition, the communicating applications do not have to be available at the same time (i.e. messages can be made persistent)

The self-contained messages are sent by one component (the *publisher*) through virtual channels (*topics*) to which other interested software components can subscribe (*subscribers*)

The application can choose to ignore a received message or react to it by, for instance, sending a reply containing requested information. The exchange of these messages is governed by two important principles. First, message delivery is completely asynchronous; that means the exact moment at which a given message is delivered to a given subscribing application is unknown. Secondly, reliability mechanisms are in place such that the messaging system can offer the guarantee that a given message is delivered once and only once.

Text messaging using cellular phones can be seen as a simple message oriented application, but more complex systems can also adopt this architecture.

Clearly, the effectiveness of system depends on the messaging system that is used to deliver the messages. Two approaches can be taken when designing such a system. The first is to use a centralized architecture where all messages transit through a message server that is responsible for delivering messages to the subscribers. This is the simpler model, but all functionality then relies on the server. The alternative is to use a decentralized architecture where message routing is delegated to the network layer and where some of the server functionality is distributed among all message clients.

**The Message Oriented Architecture and design principles**

**1. *Divide and conquer*:** The application is made of isolated software components.

**3. *Reduce coupling*:** The components are loosely coupled since they share only data format.

**4. *Increase abstraction*:** The prescribed formats of the messages are generally simple to manipulate, all the application details being hidden behind the messaging system.

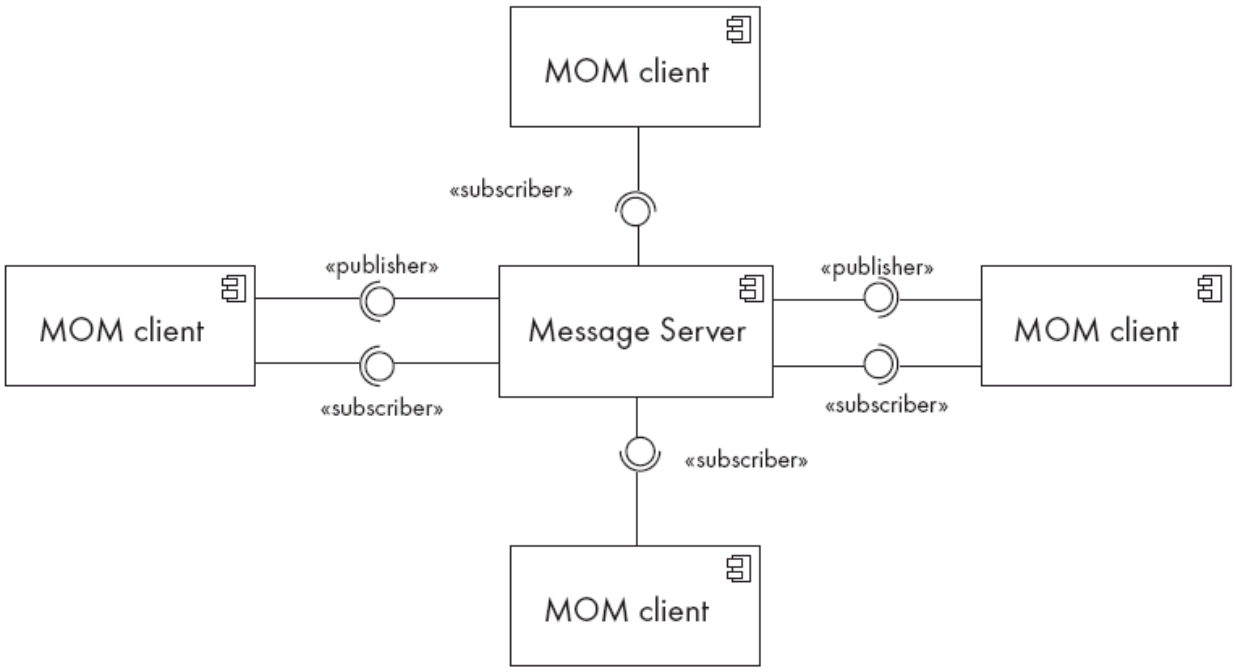
**5. *Increase reusability*:** A component will be reusable is the message formats are flexible enough.

**6. *Increase reuse*:** The components can be reused as long as the new system adheres to the proposed message formats.

**7. *Design for flexibility*:** The functionality of a message-oriented system can be easily updated or enhanced by adding or replacing components in the system.

**10. *Design for testability*:** Each component can be tested independently.

**11. *Design defensively*:** Defensive design consists simply of validating all received messages before processing them.


MOM client
«subscriber»
«publisher»
MOM client
Message Server
«publisher»
MOM client
«subscriber»
«subscriber»
«subscriber»
MOM client