**Software Architecture, Design and Patterns**

1. **Introduction to Software Architecture**

Software design and documentation take place during the design phase of the SDLC. Software architecture, simply put, is the organization of the system. Software architecture serves as a blueprint for the software system that the programmers use to develop the interacting components of the software. The architecture comprises the fundamental structures of a software system and explains the behavior of that system. The architecture defines how components should interact with each other, the operating environment, and the principles used to design the software.

The software architecture captures early design decisions that are often costly to change once implemented. A software’s architecture addresses non-functional aspects of the application such as performance, scalability, maintainability, interoperability, security, and manageability. Well-designed software architecture is important for a number of reasons. First, it balances the differing needs of the stakeholders and serves as a basis for communication among team members. Next, the architecture represents the earliest design decisions, and those decisions conflate other coding implementation decisions later in the development process. Also, the well-designed architecture allows for agility due to changing requirements. A well-organized architecture increases the lifespan of the software system even when implementation details change.

Architectural design also guides the choice of technology stacks used for the system. Remember that architecture addresses non-functional capabilities so choosing stacks that address these requirements is paramount in the design phase. Recall that a tech stack is a list of all the technologies including software, programming languages, libraries, and frameworks that will be used to create the system. The architects must be aware of the stack’s advantages and disadvantages to anticipate development needs. Much like blueprints communicate design decisions to the builders of a house, there are also several artifacts produced during the architectural design phase that are used to communicate the software design to the stakeholders. These artifacts include a software design document, or SDD, an architectural diagram, and unified modeling language, or UML, diagrams. The SDD is a collection of technical specifications that indicate how the design should be implemented.

It provides a functional description of the software and design considerations such as assumptions, dependencies, constraints, requirements, objectives, and methodologies. The architectural diagram displays components, their interactions, their constraints, and their confines. It displays the architectural patterns used in the design. Architectural patterns are general, reusable solutions to commonly occurring problems and will be discussed in more detail in an upcoming video. UML diagrams are diagrams that communicate structure and behavior using common programming language agnostic notation. UML diagrams will also be discussed in more detail in another video. Another topic to be discussed in this module as it relates to software architecture includes production deployment considerations.

The architecture drives choices about the environment in which the software is released. The production environment is comprised of the infrastructure that runs and delivers the application to the end-user such as the servers, load balancers, and databases. In this video you learned that: Software architecture functions as a blueprint and represents the underlying organization of the application. A good architectural design is important because it serves as a basis for communication among team members. Software architecture represents the earliest design decisions, is hard to change once development starts, and accommodates changing requirements during development. Architectural design influences technology stack choices and the production environment and Artifacts resulting from the design include the SDD, the architecture diagram, and UML diagram.

1. **Software Design and Modeling**

Software Design is a process during which structural components and behavioral attributes of the software are documented before it can be developed. One of the key activities of the design process is modeling the software to express its design. This involves creating visual or diagrammatic representations of the bigger software solution, and its sub-components, as well as the interactions between them.

This can be done using simple flowcharts or more standardized methods like UML. A software system can be construed in terms of structural elements. Structured design conceptualizes a software problem into well-organized smaller solution elements called modules and sub-modules. Structured design stresses organization in order to achieve a solution. A well-structured design should contain modules that are cohesive and loosely coupled. Cohesion means that all functionally related elements are grouped together. Coupling is the communication between different modules. For a system to be loosely coupled the modules should be weakly associated so changes in one component have minimal effect on another. Loose coupling is an architectural principle often used in service-oriented architectures and microservices based architectural patterns, which will be discussed later in the module. A diagram of a service claim

AI-generated content may be incorrect.

The diagram shows a simplified example billing system. Modules are arranged in a hierarchy and communicate with each other. The rectangles represent the modules and sub-modules. You can see that “billing” is the main module, and the other rectangles are sub-modules to the main billing module. In this example the sub-modules are “insurance verification,” “submit claim,” and “output total.” The arrows represent the flow of the data in the system. Behavioral models describe what a system does, without explaining how it does it.

The overall behavior of a system can be communicated through behavior models. There are a number of different UML diagrams that can be used to communicate the behavior of a system. We will discuss two such diagrams, a state transition diagram, and an interaction diagram. When developing a complex software system with interconnected modules, it can be difficult to remember the relationships, behaviors, and hierarchies among different elements. UML, which stands for Unified Modeling Language, is a way to visually represent the architecture, design, and implementation of complex software systems. UML is a standardized modeling language that can be used throughout development processes. UML diagrams can be divided into two classes: either structural or behavioral.

UML is programming language agnostic, so software developers can readily interpret and apply it to their work no matter which language they are developing in. There are several advantages of using UML to communicate architecture, behavior, and structure with development teams. The biggest advantage of UML diagrams is that they allow you to plan out features before any coding takes place which saves time and money. Secondly, the diagrams can be used to bring new team members or developers switching teams to get up to speed quickly. Also, the diagrams can be used to facilitate communication between technical and non-technical audiences more easily. And finally, having a visual representation of the system allows developers to navigate the source code because they can see the relationships among modules. So there are many types of UML diagrams but generally, UML diagrams can be classified as either behavioral or structural.

The behavior of a system can be explained and represented with the help of a UML diagram called a state transition diagram. The state transition diagram is a collection of states and events that describes the different states that a system has and the events which cause a change of state in the system.

A diagram of a procedure

AI-generated content may be incorrect.

The diagram shown is an example of a state transition diagram that models a patient going to see a doctor at a clinic. The different states include “waiting,” “testing,” and “with the doctor.” The arrows represent possible transitions from one state to another and names the event that triggers the transition. An interaction diagram is used to model the dynamic nature of a software system. They help visualize objects and their relationships. A sequence diagram, which is the type of interaction diagram shown here, displays the communication between objects with respect to time. This example shows a patient making an appointment in an online portal. This is another example of a behavioral UML diagram.

Structured design breaks down a software problem into well-organized smaller solution elements. Behavioral models describe the behavior of the system without explaining how the system implements the behavior. Developing UML diagrams saves time and money by helping developers quickly get up to speed on a project, plan features in advance of coding, and help developers navigate source code easily. A state transition diagram is a behavioral model containing a collection of states and events that describe the different states of a system and the events which cause a change of state. And finally, an interaction diagram describes how interacting objects communicate.

1. **Object-oriented Analysis and Design**

Object-oriented analysis and design, or OOAD for short, is an approach for analyzing and designing a software system when the system will use object-oriented programming languages to develop it. So, before we discuss OOAD, let’s learn a little more about object-oriented programming in languages like Java, C++, or Python. At the heart of OOAD are objects. Objects contain data, and they also have behaviors that prescribe the actions the object can take.

I could create an object for example, that represents a patient. Let’s say the patient’s name is Naya Patel and Naya needs to cancel an appointment she made. Before creating Naya, however, we must first create a generic version of a patient object. The generic version of an object is called a “class”. Let’s discuss classes next before we discuss more about Naya. Specific objects, also called instances, are created from “classes” which are blueprints or templates for an object. So, from our previous example, considering Naya Patel, Naya would be an instance of the patient class.

The class contains the object’s generic attributes – the properties and methods – but it is only when the object is created, which is called “instantiation,” inside the code that these generic attributes are set to particular values. So, the patient class might have a variable called LastName, which is a property but does not specify what that last name is. LastName is just a placeholder until the object is created and assigned a name. And once the object has been instantiated its methods can be called to make the object perform some action such as making or canceling an appointment. OOAD is used for a system that can be broken down into objects that interact with each other. In this way, multiple developers can work on different aspects of the application at the same time. As noted earlier, visual UML diagrams can be created that show both the static structure and dynamic behavior of a system.

A diagram of a medical personnel

AI-generated content may be incorrect.

We’ve seen a couple of examples of behavioral UML diagrams already, so now let’s look at a structural UML diagram called a “class diagram” next now that we know what classes are. This is an example of a “class diagram.” Class diagrams are commonly used to communicate a software system’s structure in OOAD. The class diagram shows how the classes in an object-oriented design relate to one another. Each box represents a class and shows its attributes. Recall that an object’s attributes are both its properties or its data, and its available actions, called methods. A class diagram also shows the relationships between classes.

A subclass is said to “inherit” its parent class attributes meaning it has the same properties and methods as the parent class but also may add additional properties and methods. In this diagram, the nurse, doctor, and technician classes are subclasses of medical personnel, and the specialist class is a subclass of the doctor class. This means that doctors can do anything medical personnel can do and specialists can do anything a doctor can do. In this video you learned that: Object-oriented analysis and design is the process of planning a software system based on the behaviors of interacting objects. Objects contain data, and they also have behaviors that prescribe the actions the object can take. Classes are blueprints for objects. And a class diagram is a structural UML diagram that shows the relationship between objects.

1. **Approaches to Application Architecture**

A component is an individual unit of encapsulated functionality that serves as a part of an application in conjunction with other components. There are six characteristics of components. Components should be reusable, replaceable, independent, extensible, encapsulated, and non-context specific. Let’s briefly discuss each of these principles.

**Reusable** implies that components should be designed such that they can be reused in different applications. Replaceable means that a component should be easily replaced with another component.

**Independent** means the component should be designed so it doesn’t have dependencies on other components.

**Extensibility** entails the ability to add behavior to a component without changing other components.

**Encapsulation** consists of bundling a component’s data and methods to hide its internal state, so it doesn’t expose its specific implementation.

Creating a component that is non-context-specific involves designing it so it operates in different environments. Data that sets its internal state should be passed to the component rather than included within or accessed by the component.

Let’s take a look at some examples of components. An API can be packaged as a component, if it can be reused across multiple systems and applications. For instance, a component could be an open-source API that connects a system to a particular database. A component can also be the interface for a database, called a data access object, that switches the user to a different database without the application knowing about the switch. And a controller is a type of component that determines which other components need to be called for a particular event. It controls the flow of data between two other components. Component-based architecture, then, focuses on the decomposition of the design into these logical components.

Component-based architecture provides a higher level of abstraction than object-oriented designs. A component-based architecture should define, compose, and implement loosely coupled independent components so they work together to create an application.

A service is like a component, also a unit of functionality, but it is designed to be deployed independently and reused by multiple systems. A service focuses on a solution to a business need. A key difference between a component and a service is that a service will only have one unique, always running instance with whom multiple clients communicate.

A diagram of a diagram

AI-generated content may be incorrect.

This diagram displays the relationship between objects, components, and services in a layered architecture. Services are made of components and components consist of objects.

A service is a type of component. It is meant to be deployed independently of the overall system. Examples of services include: checking a customer’s credit, calculating a monthly loan payment, or processing a mortgage application.

In a service-oriented architecture, or SOA, services are loosely coupled and interface with each other via a communication protocol over a network. SOA supports building distributed systems that deliver services to other applications through the communication protocol. So, let’s discuss distributed systems next. A distributed system is a system with multiple services located on different machines that coordinate interactions by passing messages to each other via a communication protocol such as hypertext transfer protocol, also known as HTTP. Even though the services on a distributed system operate on multiple machines, a distributed system appears to the end-user as a single coherent system.

A distributed system shares resources such as hardware, software, and data. They are fault-tolerant, meaning if a node or a service fails the system continues to run also implying that the system may change during execution without service interruption. Multiple activities run concurrently on a distributed system reducing latency and increasing throughput. Another property of distributed systems is that they are scalable as the number of users increases. The computers running the distributed system do not need to use the same kind of hardware or operating systems. A distributed system may be made up of different kinds of computers and programming languages. Now a node is any device on a network that can recognize, process, and transmit data to other nodes on the network.

A distributed system consists of multiple interconnected nodes where the nodes are running one or more services in an SOA. Distributed systems generally use one or more of the following basic types of architecture: client-server, three-tier, peer-to-peer or microservices.

Components are reusable, independent, replaceable, extensible, encapsulated, and non-context specific. Component-based architecture is the decomposition of the system into logical independent components. Services are made of components and components consist of objects. Services are deployed independently and can be reused by multiple systems. In an SOA services are loosely coupled and interface with each other via a communication protocol over a network. And lastly, distributed systems run on multiple services on different machines and they appear to the end-user as a single coherent system .

1. **Architectural patterns in software!**

An architectural pattern is a repeatable solution to a problem in software architecture. Patterns highlight common internal elements and structures of a software system. Different architecture patterns may share related characteristics. In this video, we will discuss 2-tier, 3-tier, peer-to-peer, event-driven, and microservices.

However, there are many other patterns that will not be covered in this video. These patterns include model-view-controller, message-broker, blackboard, pipe-filter, and controller-responder.

A close-up of a computer screen

AI-generated content may be incorrect.

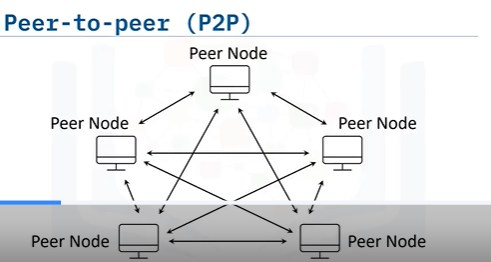
**The 2-tier architecture,** also called client-server, is a computing model in which the server hosts, delivers, and manages most of the resources and services delivered to the client. The interface resides on the client machine and makes requests to a server for data or services. This type of architecture usually has more than one client computer connected to a server component over a network connection.

A screen shot of a computer

AI-generated content may be incorrect.

**A 3-tier architecture**, or an n-tier architecture where there are more than three layers, is the most common software architecture. The 3-tier architecture is composed of several horizontal tiers that function together as a single unit of software. A tier only communicates with other tiers located directly above and below it. Related components are placed within the same tier. Changes in one tier do not affect the other tier. The 3-tier architecture organizes applications into three logical and physical computing tiers: the presentation tier, or user interface; the middle tier which is usually the application tier, is where business logic is processed ;the data tier, where the data is stored and managed

**The peer-to-peer architecture**, or P2P for short, consists of a decentralized network of nodes that are both clients and servers. The workload is partitioned among these nodes. Peers make a portion of their resources directly available to other network participants, without the need for central coordination by servers. Resources are things like processing power, disk storage, or network bandwidth.



Peers both supply and consume resources, in contrast to the traditional client-server architecture in which the consumption happens strictly by the client and the servers, supply the resources. Peer-to-peer architecture is useful for file sharing, instant messaging, collaboration, and high-performance computing.

A diagram of a router

AI-generated content may be incorrect.

An event is anything that results in a change of state. An event can be thought of as an action that is triggered by the end-user, such as a mouse click, or another part of the program. **Event-driven** architecture focuses on producers and consumers of events. Producers listen for and react to triggers while consumers process an event. The producer publishes the event to an event router.

The router determines which consumer to push the event to. The triggering event generates a message, called an event notification, to the consumer which is listening for the event. The components in event-driven architectures are loosely coupled making the pattern appropriate for use with modern, distributed systems.

A diagram of a software application

AI-generated content may be incorrect.

Microservices are an approach to building an application that breaks its functionality into modular components called services. An application programming interface, also called an API, is the part of an application that communicates with other applications. An API defines how two applications share and modify each other’s data. APIs can be used to create a microservices-based architecture. The API Gateway routes the API from the client to a service. Orchestration handles communication between services. Let’s discuss an example for each of these patterns.

A text messaging app is an example of a 2-tier pattern. The client initiates a request to send a text message through a server and the server responds by sending that message to another different client. Another example of the 2-tier pattern, is Database clients connecting with database servers.

Many web apps use the 3-tier pattern. They use a web server to provide the user interface, an application server to process user inputs, and a database server that handles data management.

Ride-sharing apps such as Lyft and Uber are examples of event-driven patterns. The customer sends a notification that they need a ride from a particular location to another location, and that event is routed to a consumer.

Cryptocurrencies such as Bitcoin and Ethereum use a peer-to-peer pattern. Each computer in the blockchain acts as both server and client.

Finally, social media sites are composed of microservices. A user has an account. That account can request different services such as adding friends, targeted ad recommendations, and displaying content.

Architectural patterns are not necessarily mutually exclusive. In other words, two or more of these patterns can be combined. For instance, a three-tiered architecture can also be microservice-based, or a peer-to-peer architecture can also be event-driven. However, not all architectural patterns can be used in conjunction with others. A peer-to-peer cannot also be two-tier because a single machine in a peer-to-peer architecture represents both a client and a server whereas a two-tier architecture separates the client from the server.

It is up to the system architect to determine which architectural patterns the software system should adhere to.

An architectural pattern is a repeatable solution to an architectural problem. A 2-tier pattern has a client and server. Text messaging apps use a 2-tier pattern. A 3-tier pattern has 3-tiers that interact with each other. Web apps use a 3-tier pattern. An event-driven pattern has actions that are produced and responded to by a consumer. Ride-sharing apps use an event-driven pattern.

The peer-to-peer pattern consists of a decentralized network of nodes that act as clients and servers. Cryptocurrency is an example of the peer-to-peer pattern. Microservices are loosely coupled individual services that behave as a single system and interact with the client. Communication is orchestrated among services. Social media sites are an example and Two or more patterns can be combined in a single system but some are not mutually exclusive

1. **Application Deployment Environment**

An application environment is the combination of the hardware and software resources required to run an application. This includes: the application code and/or binary executables for its various components or modules the software stack it requires for running the application such as modules and libraries it depends on, third party applications and middleware, and the operating system, any networking components and infrastructure, as well as any physical or virtual hardware including computing or processing resources, memory, and storage. There are a variety of environment types depending on the application’s stage in the lifecycle.

The pre-production environments are those platforms that the application resides on in various forms as it gets prepared for production. Common pre-production environments are “development,” “QA,” which stands for quality assurance, and “staging.” The development environment is the platform on which the application is being actively coded, and in many cases it may just be the developer's workstation. The QA environment, sometimes called “testing” is the environment that allows the QA team to test the application’s components. The staging environment is the environment that is as close to replicating the production environment as possible but is not meant for general users.

The production environment, often just called “production” includes the entire solution stack consisting of both hardware and software on which the application runs as additional infrastructure components.

The production environment is intended for all users. Unlike the pre-production environments, this robust environment must take the application “load” into consideration because it is the environment intended for general use, possibly by thousands or millions of people at the same time for enterprise-level applications. Production environments must also take into account non-functional requirements like security, reliability, and scalability. This makes the production environment more complicated than the pre-production ones.

Now, let’s take a look at the production environment infrastructure. There are several options for deploying application environments. In on-premises deployment, the system and its infrastructure reside in-house, within the organization’s physical location, often behind a firewall. Firewalls prevent unauthorized access to or from a private network.

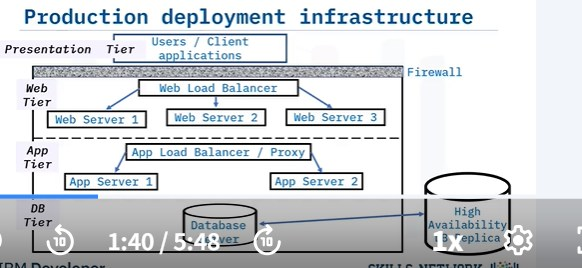
If an organization desires greater security or control of an application and the data in use by that application, it may deploy the application on-premises. For on-premises software deployments, an organization is responsible for the system, hardware, related infrastructure, and maintenance required to run the application. On-premises deployment is usually more expensive when compared to cloud deployment. Let’s discuss cloud deployments next. There are three types of Cloud deployment models—public, private, and hybrid. The public cloud is when you leverage the software’s supporting infrastructure over the open internet on hardware owned by the cloud provider. That hardware and the associated services are shared with other companies. Public cloud providers include Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform, and IBM Cloud. The public cloud is the most common due to its scalability and cost.

With a private cloud, the cloud infrastructure is provisioned for exclusive use by a single organization. The software system can be run on-premises, or the infrastructure could be owned, managed, and operated by a service provider. For example, AWS is also a private cloud service provider. The main advantage of a private cloud is increased security, but it also allows for more flexibility because it can be fully customized. A mix of both public and private clouds, working together seamlessly, is called a hybrid cloud model. A hybrid cloud potentially optimizes the advantages of both public and private cloud models with regard to cost, security, scalability, and flexibility.

Application environments include: development, testing or QA, staging, and production. Production environments must also take into account non-functional requirements like load, security, reliability, and scalability. And application environments can be deployed either on-premises on traditional hardware, or on public, private, or hybrid cloud platforms.

1. **Production Deployment components**

Let’s consider an n-tier architecture required to deploy an application in a production environment and represent the infrastructure using a diagram.



The top tier is the presentation tier which contains the front-end client applications. All of the other tiers are located behind a firewall. The next tier is the web tier which has a web load balancer that distributes incoming traffic to several web servers. The tier below the web tier is the application server tier. This tier contains an app load balancer or a proxy server that routes traffic to different application servers. The bottom tier is the data tier that contains the database server. A high availability replica of the database is often also used to ensure reliability. Some environments may have additional components or tiers.

Also not all of these components are necessary for every deployment. For example, in some environments there might not be a need for both application servers in addition to web servers. Now, let’s look at these components in further detail. A firewall is a security device that monitors traffic between networks. It permits or blocks requested data based on a set of security rules. It acts as a barrier between networks to block viruses, malware, and hackers from accessing the internal network. The purpose of load balancers is to distribute network traffic efficiently amongst multiple servers, called a server farm, on a network.

Load balancers are used to prevent server traffic overload and are located between clients and the servers. A load balancer determines which servers are capable of fulfilling those requirements in a manner that maximizes availability and responsiveness. Load balancers ensure that no one server is overworked. They manage concurrent requests from clients and return the correct data in a fast and reliable manner.

Web and application servers are either software or machines that provides services, resources, data, or applications to another computer program, called the client. Servers store, process, and manage network data, devices, and systems. A web server delivers content such as web pages, files, images, and videos to a client.

And a web server primarily responds to hypertext transfer protocol requests coming from a web browser such as a user accessing a website. An app server is a server that runs business logic and provides the application to the client rather than the client running the app on their own machine. Its primary job is to enable interaction between the end-user and the server-side application code. The application code represents the business logic that determines how data can be created, stored, and changed. And that logic dictates things like transaction results and what data is written to and retrieved from a database. A proxy server is an intermediate server that sits in between two tiers and handles requests between those tiers. A proxy server can serve multiple purposes such as load balancing, system optimization, caching, acting as a firewall, obscuring the source of the request, encryption, scanning for malware, and more.

And a proxy server can improve the efficiency, privacy, and security of data flowing through a network. Finally, a database is a collection of related data, stored on a computer that can be accessed in various ways. A database is usually controlled by software called a database management system or DBMS. The DBMS controls a database by connecting the database to users or other programs. The database server controls the flow and storage of data. The DBMS connects the database server to an application so data stored in the database can be retrieved or manipulated by the application. Common components needed for a production environment include a firewall, a load balancer, web and application servers, proxy servers, and database servers.

A firewall is a security device that monitors traffic between networks. A load balancer distributes network traffic amongst servers. A web server delivers content such as web pages, files, images, and videos to a client. An application server runs business logic and provides the application to the client rather than the client running the app on their own machine and finally, a database server stores and controls the flow of data through a database management system.

Summary & Highlights

* Software architecture functions as a blueprint and represents the importance of a good architectural design.
* Structured design breaks down a software problem into well-organized smaller solution elements whereas behavioral models describe the behavior of the system without explaining how the system implements the behavior.
* Developing UML diagrams saves time and money by helping developers quickly get up to speed on a project, plan features in advance of coding, and navigate source code easily. Types of UML diagrams include state transition, interaction, and class diagrams.
* Objects contain data, and they also have behaviors that prescribe the actions the object can take, whereas classes are blueprints for objects.
* A service-oriented architecture (SOA) consists of loosely coupled services that interface with each other via a communication protocol over a network. Distributed systems run on multiple services on different machines, but they appear to the end-user as a single coherent system.
* An architectural pattern is a repeatable solution to an architectural problem. Types of architectural patterns include 2-tier, 3-tier, event-driven, peer-to-peer, and microservices. Two or more patterns can be combined in a single system, but some are mutually exclusive.
* Application environments include development, testing or QA, staging, and production. Production environments tend to be more complex than pre-production because they must take into account non-functional requirements like load, security, reliability, and scalability.
* Application environments can be deployed either on-premises on traditional hardware, or on public, private, or hybrid cloud platforms.
* Common components needed for a production environment include a firewall, a load balancer, web and application servers, proxy servers, and database servers.