The collection of small services that are combined to form the actual application is the concept of microservices pattern. Instead of building a bigger application, small programs are built for every service (function) of an application independently. And those small programs are bundled together to be a full-fledged application.

So adding new features and modifying existing microservices without affecting other microservices are no longer a challenge when an application is built in a microservices pattern.

Modules in the application of microservices patterns are loosely coupled. So they are easily understandable, modifiable and scalable.

Example Netflix is one of the most popular examples of software built-in microservices architecture. This pattern is most suitable for websites and web apps having small components.

❖ Application Architectures

An application architecture is a structural map of how a software application is assembled and how applications interact with each other to meet business or user requirements. An application architecture helps ensure applications are scalable and reliable, and assists enterprises in identifying gaps in functionality.

In general, an application architecture defines how applications interact with entities such as middleware, databases and other applications. Application architectures usually follow software design principles that are generally accepted among adherents, but might lack formal industry standards.

The application architecture can be thought of like architectural blueprints when constructing a building. The blueprints set out how the building should be laid out and where things such as electric and plumbing service should go. The builders can then use

Prof. Sheetal Jadhav

these plans to construct the building and refer to these plans later if the building needs to be refurbished or extended. Building a modern software application or service stack without a well-considered application architecture would be as difficult as trying to construct a building without blueprints.

An application architect will often have the principal responsibility for the application's design and ensuring it meets the stated business goals.

Benefits of an application architecture

Overall, an application architecture helps IT and business planners work together so that the right technology is available to meet the business objectives. More specifically, an application architecture offers the following benefits:

Reduces costs by identifying redundancies, such as the use of two independent databases that can be replaced by one.

Improves efficiency by identifying gaps, such as essential services that users can't access through mobile apps.

Creates an enterprise platform for application accessibility and third-party integration.

Allows for interoperable, modular systems that are easier to use and maintain.

Helps architects "see the big picture" and align software strategies with the organization's overall business objectives.

Common organization application architecture patterns

These patterns define how an organization arranges the applications it uses and how they interact with each other. They are used to define the data flow in an N-tier or microservices architecture.

Prof. Sheetal Jadhav

Event-driven architecture uses events passed between components. An event in event-driven architecture can be any change in the data. An example of an event might be a new order or a box being delivered. These components can generate events or consume events. The two main subtypes of event-driven architecture are the publisher-subscriber model and event stream model.

Service-oriented architecture uses smaller services to perform business processes. These services in service-oriented architecture are independent and reusable. The services often communicate using an enterprise service bus.

Prof.Sheetal Jadhav

❖ Modeling Component level Design: component

As a software engineer, you understand the importance of solid architecture in building a successful product. The foundation of any software architecture is component-level design — you break a system into its component parts and define how they interact. Done well, component-level design facilitates reuse, reduces complexity, and enables parallel development.

However, many engineers struggle with determining the right level of granularity for components and defining clear interfaces. This article provides an overview of key principles and best practices for component-level design that you can apply to your next project. We explore strategies for identifying components, defining interfaces, managing dependencies, and achieving the right level of abstraction.

Finally, With a thoughtful approach to component design, you can create a flexible, scalable architecture that stands the test of time. Strong components form the building blocks of strong systems, so investing in this foundational layer will pay dividends as your software grows in scope and complexity.

What Is Component Level Design?

Component level design refers to the process of breaking down a system into its constituent parts to better understand how they interact and connect. In software engineering, it is the phase that focuses on defining and developing the software components that will be used to build the overall system architecture.

Component level design has two main goals:

- Identify the components that are needed to build the system. This includes determining the boundaries of each component and how they relate to one another.
- Define the interfaces between components. This makes the components loosely coupled and independent, allowing them to be developed and tested separately before being integrated into the full system.

When designing components, software engineers consider:

- Functionality: What is the purpose of the component? What features will it provide?
- Data: What data does the component need to operate? What data does it produce? How will it interface with data from other components?
- Dependencies: On what other components does this component rely? How will those dependencies be handled?
- Reusability: Can this component be reused in other systems? If so, how can its design be made more generic?
- Scalability: How will this component handle increases in load or volume? Can its performance be optimized?
- Component-level design is a fundamental step in building a robust, modular software architecture. By determining how to break a complex system into discrete, interconnected parts, software engineers can create solutions that are flexible, extensible, and resilient to changes in requirements. The end result is a system that is greater than the sum of its parts.

The Benefits of Component Level Design

Component level design is an essential part of software architecture that provides many benefits. Include,

Prof. Sheetal Jadhav

- Increased reusability: Well-designed components can be reused in other systems, saving time and resources. Components are self-contained and independent, so they can function in a variety of applications.
- Improved maintainability: When software needs updating or fixing, clearly defined components make the system easier to understand, modify and debug. Changes to one component do not impact others, reducing issues.
- Enhanced flexibility: The modular nature of components allows them to be arranged and combined in different ways. New components can also be added to expand functionality without disrupting existing ones. This makes the overall system adaptable to changing needs.
- Reduced complexity: Breaking a large system into discrete components simplifies development and management. Components have specific, limited functions that are easier to build and comprehend than an overly broad, monolithic system.
- Increased reliability: Properly designed components are thoroughly tested to ensure they function correctly. They also have defined interfaces that limit unintended interactions with other components. This results in a robust, stable system.

In summary, component level design is key to crafting a high-quality software architecture. When done well, it yields reusable, maintainable and flexible systems that can handle complexity, evolve over time and provide reliable performance. For any software project, it is a foundation worth building upon.

Types of Components in Software Engineering

There are several types of components used in software engineering and component-level design. Understanding the distinctions between them will help in selecting the appropriate components for your system.

- Procedural Components

Procedural components, also known as modules, contain a sequence of steps to perform a specific function. They have a single entry and exit point, encapsulating the steps into a reusable block. Procedural components are best for simple, linear tasks without complex logic.

Object-oriented components, or objects, combine data and procedures into a single unit. They have a defined interface to interact with the object while hiding the implementation details. Objects are well-suited for modeling real-world entities or abstract concepts. They enable encapsulation, inheritance, and polymorphism.

Services, or application programming interfaces (APIs), provide a defined interface to access a set of related functions or resources. They abstract away the implementation details of the service, allowing it to be used by various clients. Services are commonly used for accessing data or business logic over a network. They enable loose coupling between systems.

- Data Stores

Data stores, such as databases, files, and data structures, contain organized collections of data. They provide a mechanism to persistently store and access data for use throughout a system. Selecting an appropriate data store involves considering factors like query performance, scalability, and data integrity.

In summary, procedural components, objects, services, and data stores are fundamental building blocks used in component-level design. By understanding their unique characteristics and relationships, you can craft a robust architecture with components that are cohesive, loosely coupled, and highly reusable.

Prof. Sheetal Jadhav

Guidelines for Component Design

When designing components in software architecture, following some guidelines will help ensure robust, reusable, and maintainable components.

Components should have a loose coupling, meaning little interdependence. Loosely coupled components can be changed or replaced independently without affecting other components. Some ways to achieve loose coupling include:

Defining clear interfaces that encapsulate implementation details.

Using asynchronous messaging instead of direct function calls.

Avoiding global variables and sharing data only through interfaces.

Components should have a high cohesion, focusing on a single, well-defined purpose. High cohesion makes components easier to understand, maintain, test, and reuse. Some tips for high cohesion include:

Assigning components logical and closely related responsibilities.

Not combining unrelated responsibilities in the same component.

Naming components clearly and consistently based on their purpose.

Components should provide abstraction, hiding their internal implementation details behind interfaces. This allows components to be used without understanding their inner workings. Some examples of abstraction include:

Defining interfaces that specify the services a component provides without specifying how they work.

Using access modifiers like "private" to hide implementation details.

Prof. Sheetal Jadhav

Choosing descriptive names that convey a component's purpose without revealing implementation.

Components should be designed with reusability in mind. Reusable components reduce duplication and improve maintainability. To design reusable components:

Make them self-contained, independent, and focused on a single purpose.

Define a clear interface and keep implementation details private.

Avoid assumptions about the context the component will be used in.

Make the component flexible and customizable through configuration rather than modification.

Following these guidelines will help you design robust, maintainable software components that can be reused and combined in many ways. Well-designed components form the foundation for a flexible, adaptable software architecture.

Common Mistakes in Component Level Design

Lack of Cohesion

Cohesion refers to the degree to which the elements of a component belong together. A cohesive component has a singular, focused purpose and contains closely related elements. Lack of cohesion is one of the most common mistakes in component level design. It often results in components that are difficult to understand, reuse, and maintain.

Excessive Coupling

Coupling refers to the degree of interdependence between components. Excessively coupled components are too dependent on each other, making them hard to reuse and maintain independently. Loose coupling, on the other hand, reduces interdependencies

Prof. Sheetal Jadhav

and allows components to operate independently. Excessive coupling is a frequent mistake that contradicts the goals of component-based design.

Violating Encapsulation

Encapsulation means hiding the internal details and implementation of a component from the outside. It enables the internal workings of a component to change without impacting other components. Failing to properly encapsulate components exposes their inner workings, creating tight coupling and dependencies on implementation details that may change. This mistake compromises the flexibility and maintainability of the overall system.

Lack of Abstraction

Abstraction involves suppressing irrelevant details about a component in order to reduce complexity and facilitate understanding. Lack of abstraction results in low-level components that are overly complex, difficult to understand, and reuse. Effective abstraction focuses on the essential purpose and functionality of a component, hiding unnecessary details.

Not Considering Reusability

If components are not designed with reuse in mind, the system will end up with many redundant components that cannot be reused in other contexts. This error leads to increased development and maintenance efforts. On the other hand, reusable components can be used in multiple systems, reducing their development cost.

In summary, avoiding the common mistakes in component-level design-lack of synchronization, over-coupling, content overload, lack of abstraction, and failure to consider reuse-will result in a flexible, maintainable software architecture with high-quality components.

Prof. Sheetal Jadhav