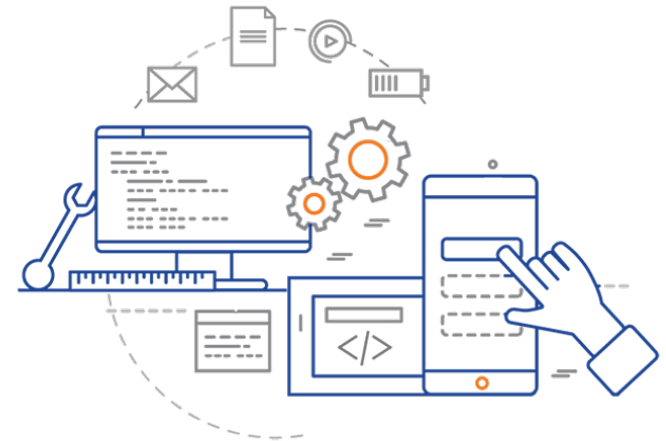


CS 398: Application Development

Week 03 Video: Analysis & Design 2

Architectural Patterns

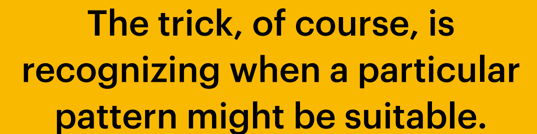
What is an architectural pattern?

An architectural pattern (or *architectural style*) is the overall structure that we create to represent our software. In describes how our components are organized and structured.

Similar to design patterns, an architectural style is a general solution that has been found to work well at solving specific types of problems.

An architectural pattern describes both the topology (organization of components) and the associated architectural characteristics.

- Architectural style :: Architecture
- Design pattern :: Design

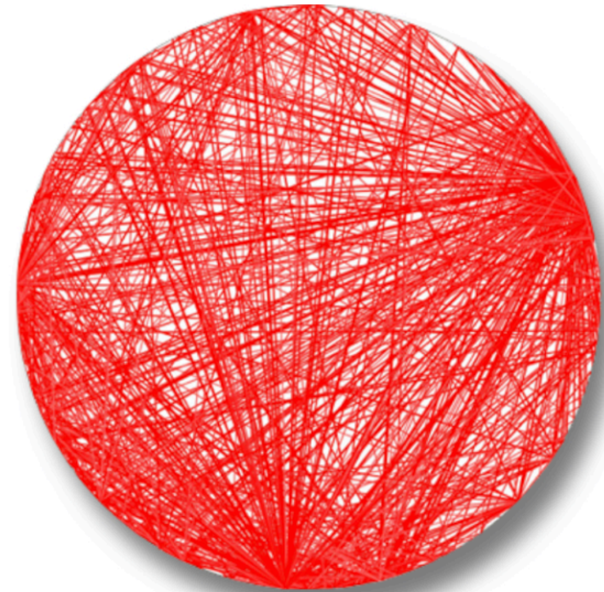


Fundamental Patterns
Big Ball of Mud

A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle.

These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.

-- Foote & Yoder 1997.

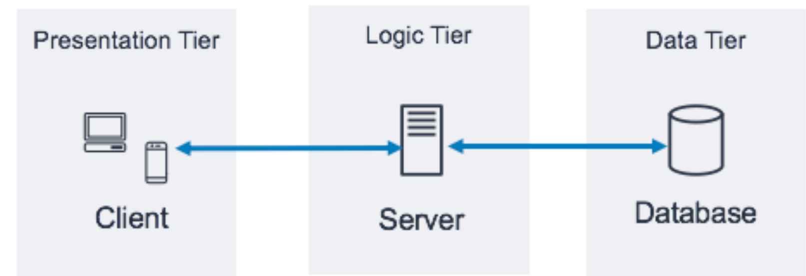


Fundamental Patterns
Client-Server

Client-server architectures were the first major break away from a monolithic architecture, and split processing into front-end and back-end pieces. This is also called a **two-tier architecture**.

There are different ways to divide up the system into front-end and back-end. Examples include splitting between desktop application (front-end) and shared relational database (back-end), or web browser (front-end) and web server (back-end).

Three-tier architectures were also popular in the 1990s and 2000s, which would also include a middle business-logic tier.



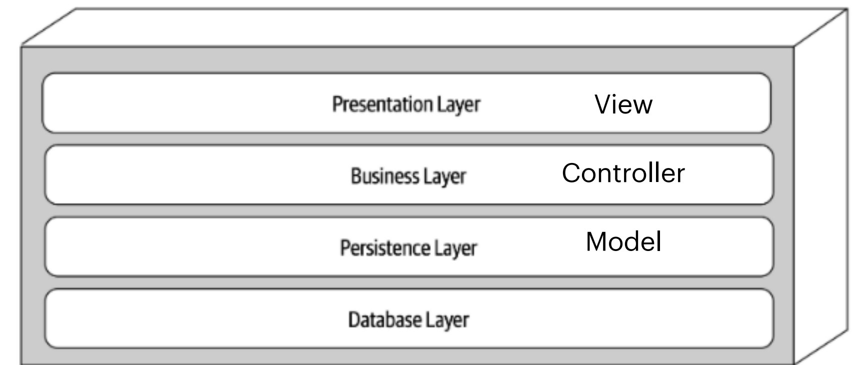
Monolithic Patterns

Layered Architecture

A layered or n-tier architecture is a very common architectural style that organizes software into horizontal layers, where each layer represents some logical functionality.

Standard layers in this style of architecture include:

- **Presentation**: UI layer that the user interacts with.
- **Business Layer**: application logic, or "business rules".
- **Persistence Layer**: describes how to manage and save application data.
- **Database** Layer: the underlying data store that actually stores the data.

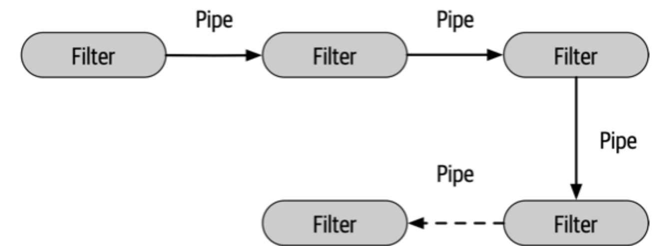


Note: these can be logical tiers (i.e. modules in the same system)

The major characteristic of a layered architecture is that it enforces a clear **separation of concerns** between layers.

Monolithic Patterns
Pipeline Architecture

A pipeline (or pipes and filters) architecture is appropriate when we want to transform data in a sequential manner. It consists of pipes and filters:



Pipes form the communication channel between filters. Each pipe is unidirectional, accepting input on one end, and producing output.

Filters are entities that perform operation on data that they are fed. Each filter performs a single operation, and they are stateless. There are different types of filters:

- **Producer**: The outbound starting point (also called a source).
- **Transformer**: Accepts input, optionally transforms it, and then forwards to a filter (this resembles a *map* operation).
- **Tester**: Accepts input, optionally transforms it based on the results of a test, and then forwards to a filter (this resembles a *reduce* operation).
- **Consumer**: The termination point, where the data can be saved, displayed.

These abstractions may appear familiar, as they are used in **shell programming**. It's broadly applicable anytime you want to process data sequentially according to fixed rules.

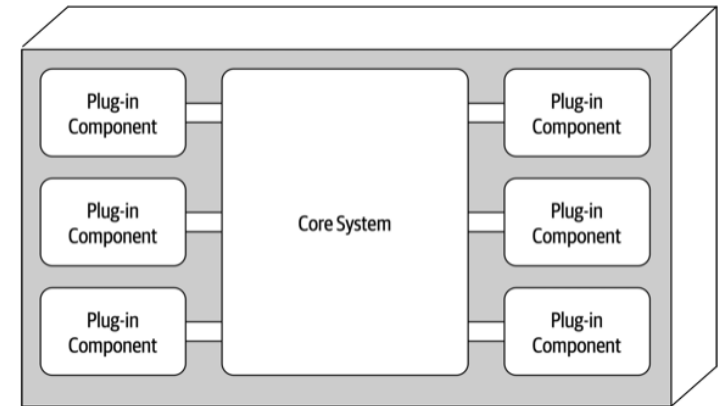
Monolithic Patterns

Microkernel Architecture

A microkernel architecture (also called plugin architecture) is a popular pattern that provides the ability to easily extend application logic to external, pluggable components.

This architecture works by focusing the primary functionality into the core system, and providing extensibility through the plugin system.

This allows the developer, for instance, to invoke functionality in a plugin when the plugin is present, using a defined interface that describes how to invoke it (without need to understand the underlying code).



Examples of this architecture include web browsers (which support extensions), and IDEA (which support plugins for various programming languages).

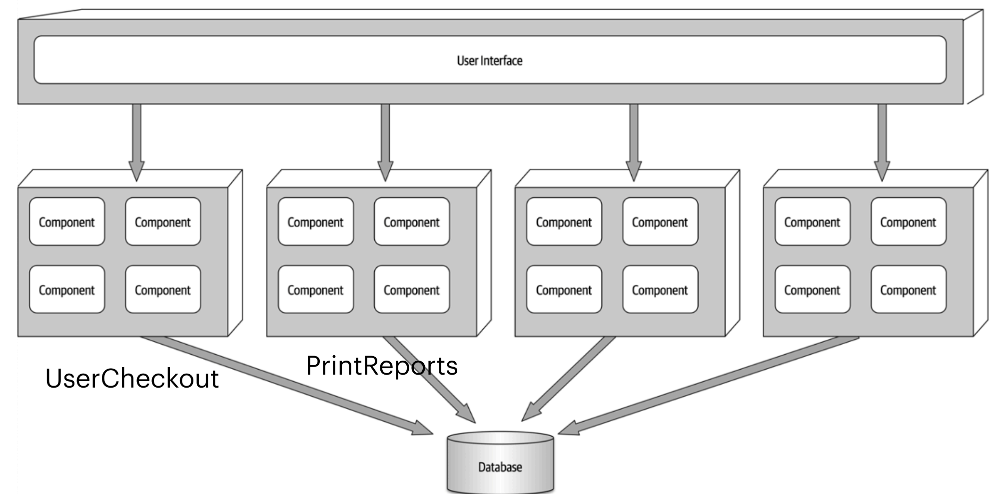
Distributed Patterns

Service Architecture

A services-based architecture splits functionality into small "portions of an application" (also called domain services) that are independent and separately deployed.

Each service is a separate monolithic application that provides services to the application, and they share a single monolithic database.

Each service provides coarse-grained domain functionality.



e.g. a service might handle a customer checkout request to process an order; this could be processed in its entirely by the service, as a single transaction.

Distributed Patterns

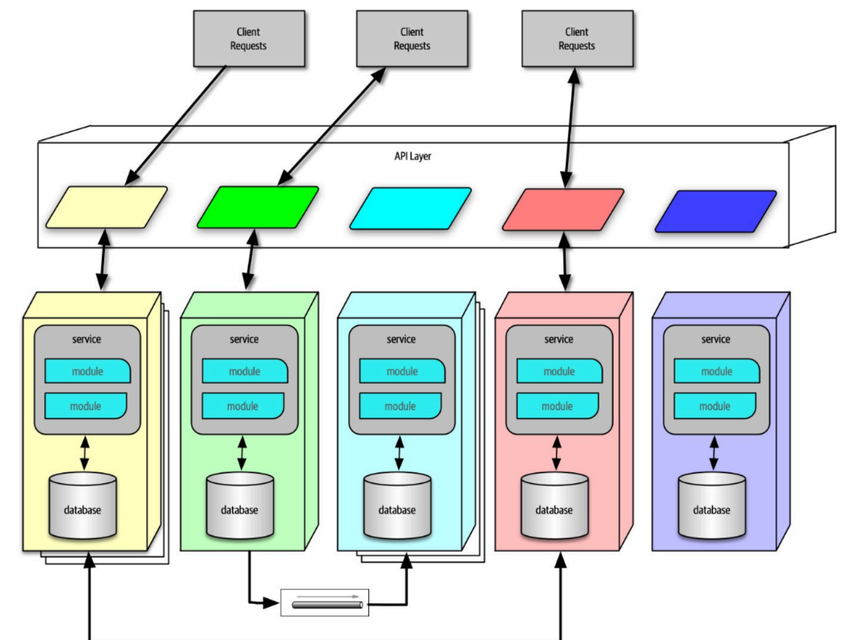
Microservice Architecture

A microservices architecture arranges an application as a collection of loosely coupled services, using a lightweight protocol.

Some of the defining characteristics of microservices:

- Services are usually processes that communicate over a network.
- Services are organized around business capabilities i.e. they provide specialized, domain-specific services to applications.
- Service are not tied to any one programming language, platform or set of technologies.
- Services are small, decentralized, and independently deployable

Each micro-service is expected to operate independently, and contain all of the logic that it requires to perform its task.



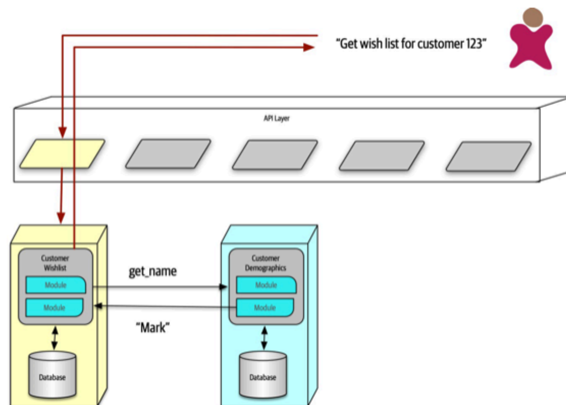
Distributed Patterns

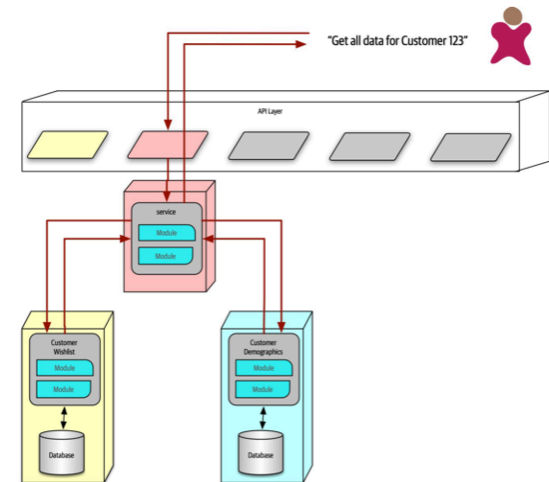
Microservice Architecture

Although the services themselves are independent, they need to be able to communicate.

This suggests that communication between microservices is a key requirement. The architect utilizing this architecture would typically define a standard communication protocol e.g. message queues, or REST.

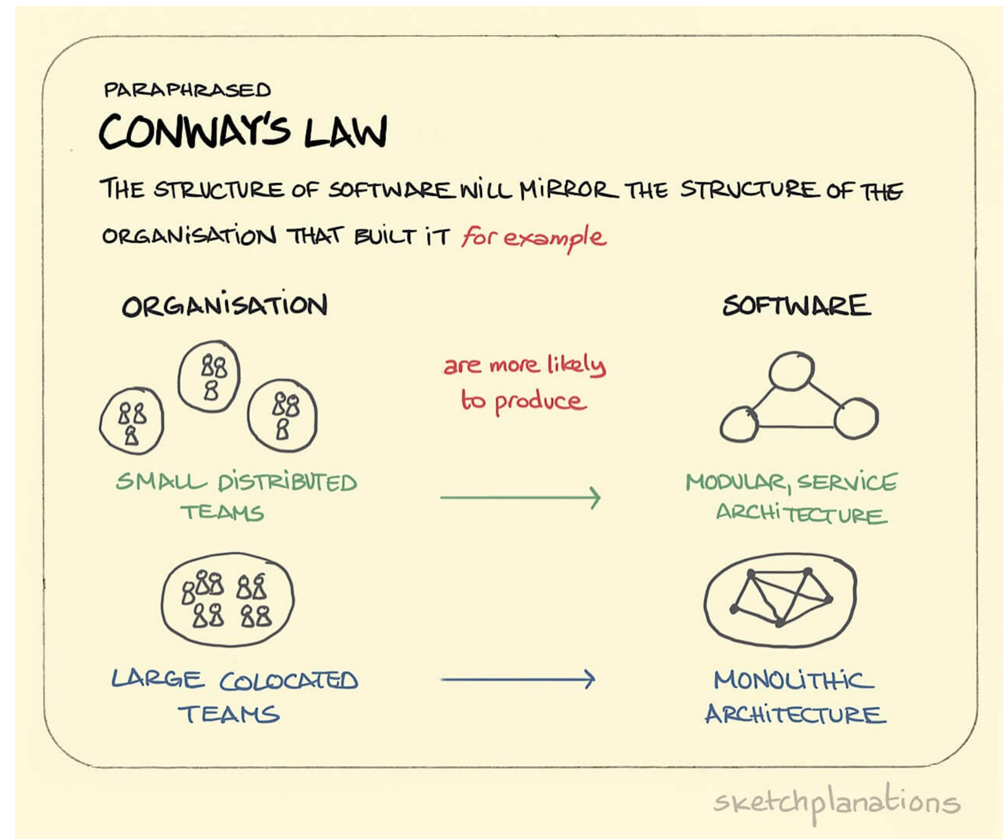
Coordinating a multi-step process like this involves either cooperation between services, or a third coordinating service.





What Architectural Style to Pick?

- Look at the characteristics of the specific architectural style.
- What are the tradeoffs? There are positive and negative characteristics of each.
- Conway's Law
  - Organizations will proceed a design that mirrors their communication structure.
  - e.g. layered architecture, with a common technical database team.
  - e.g. domain focused teams building reusable business services.



Documenting Architecture

Drawing Diagrams

Architecture and design are all about making important, critical decisions early in the process. It's extremely valuable to have a standard way of documenting systems, components, and interactions to aid in visualizing and communicating our designs.

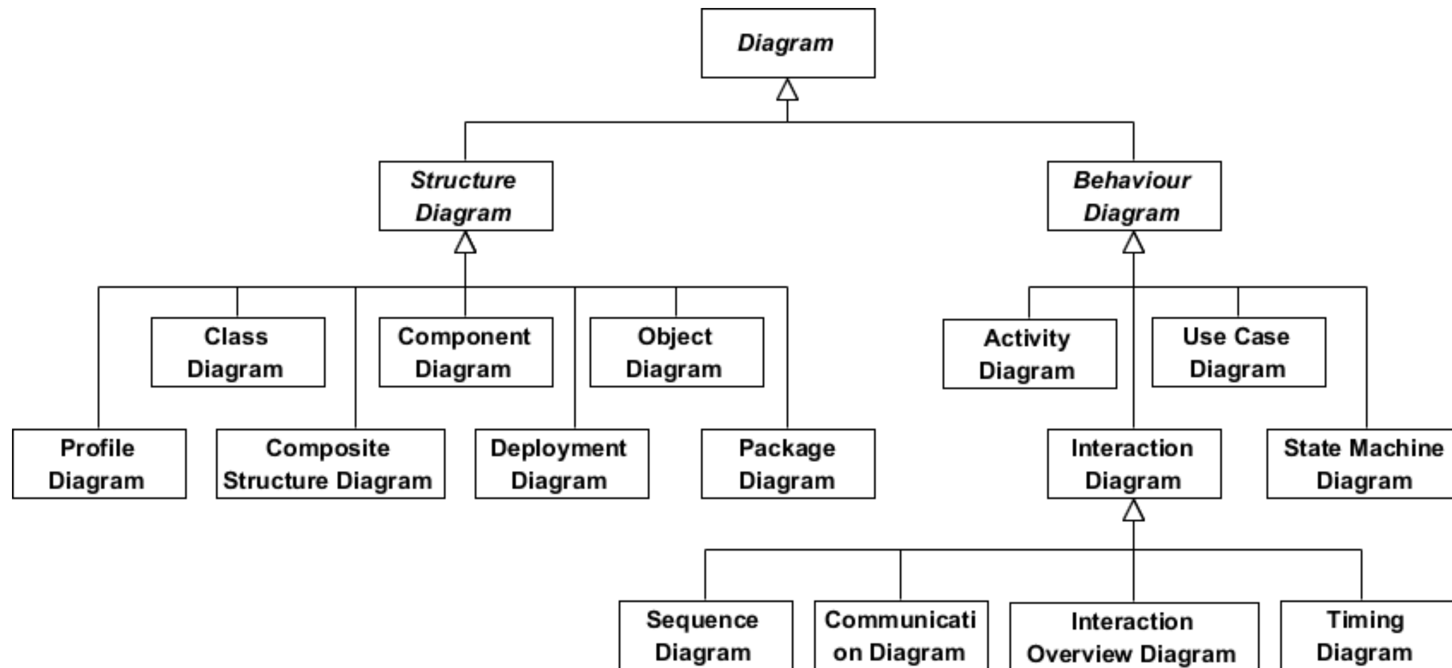
The Unified Modelling Language (aka UML) consists of an integrated set of diagrams, useful to specify, visualize, construct and communicate a design. It’s a ratified standard managed through the Object Management Group.

UML is a useful tool.

- Create whatever is useful for your architecture.
- You should NOT create diagrams for every components, interaction or state in your system. That's overkill for most projects. Instead, focus on building a high-level component diagram that shows the basic component interactions, which you can use to plan your system.

UML contains both structure and behaviour diagrams.

Structure diagrams show the structure of the system and its parts at different level of abstraction, and shows how they are related to one another. **Behaviour diagrams** show the changes in the system over time.



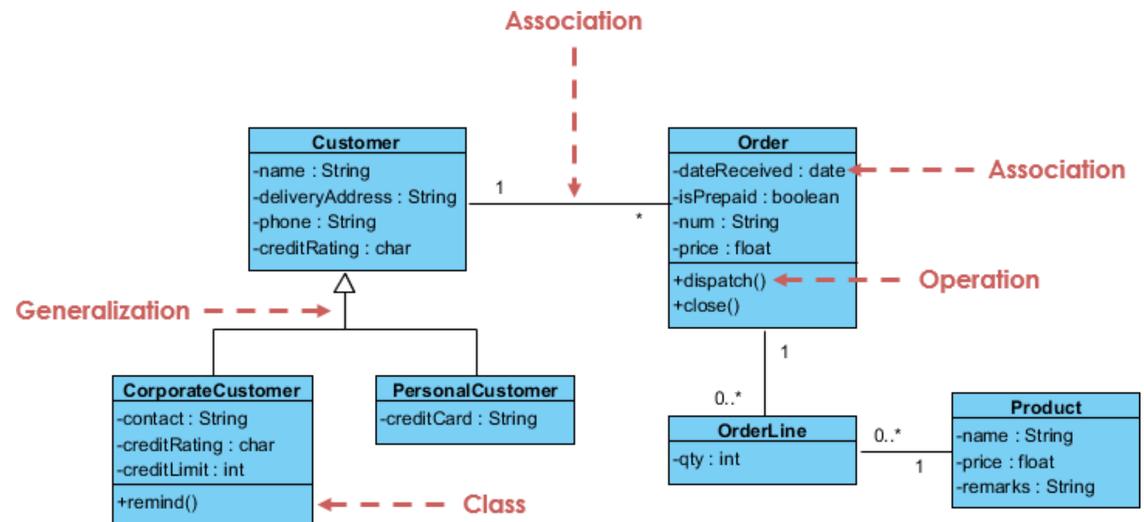
UML Structure Diagrams

Class Diagram

The class diagram is a central modelling technique that runs through nearly all object-oriented methods.

This diagram describes the types of objects in the system and various kinds of static relationships which exist between them.

(Too low level for this stage, but mentioning it here for completeness!)



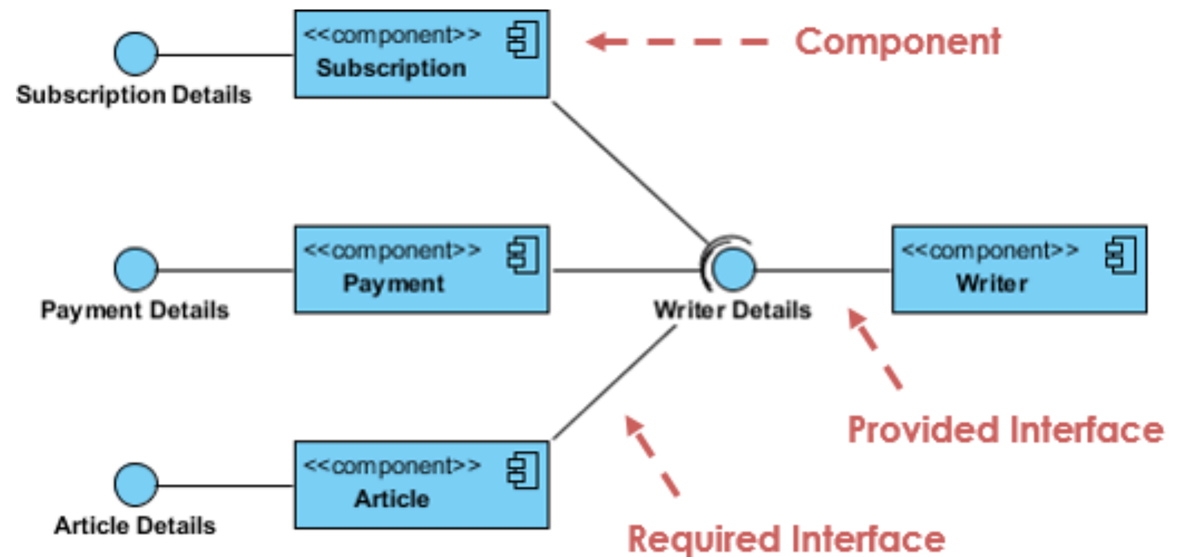
UML Structure Diagrams

Component Diagram

A component diagram depicts how components are wired together to form larger components or software systems.

It illustrates the architectures of the software components and the dependencies between them.

This is the lowest level an architecture should attempt to document.

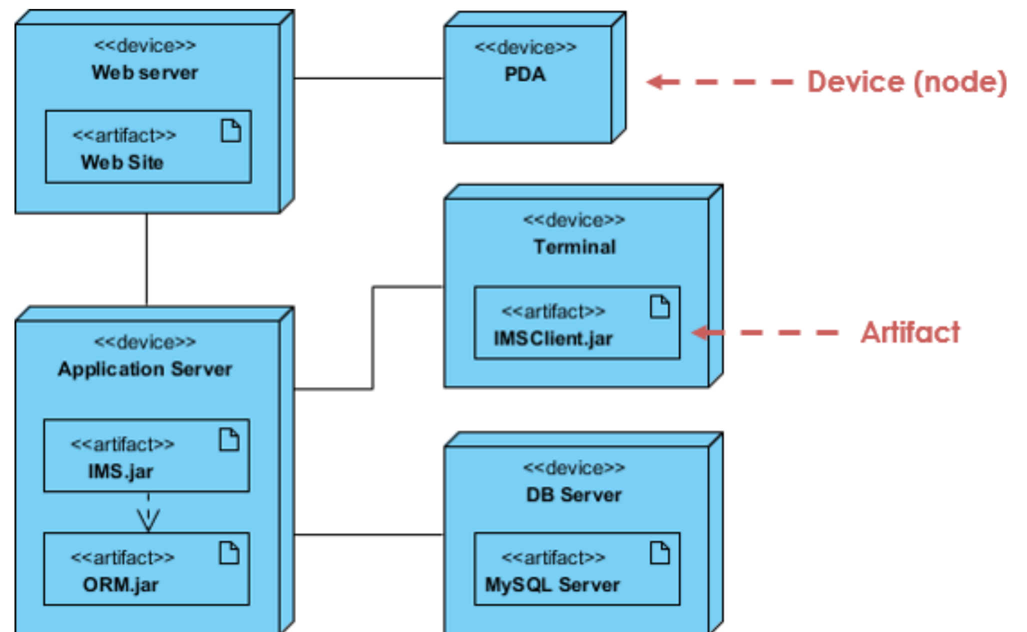


UML Structure Diagrams

Deployment Diagram

The Deployment Diagram helps to model the physical aspect of an Object-Oriented software system.

It is a structure diagram which shows architecture of the system as deployment (distribution) of software artifacts to deployment targets.

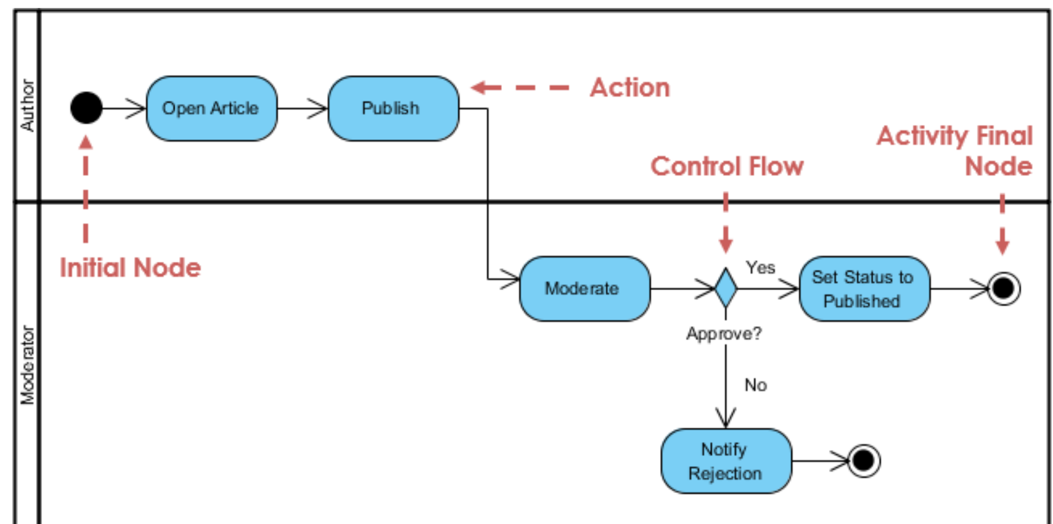


UML Behaviour Diagrams

Activity Diagram

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. It describes the flow of control of the target system.

Activity diagrams are intended to model both computational and organizational processes (i.e. workflows).



UML Behaviour Diagrams

Use Case Diagram

A use-case model describes a system's functional requirements in terms of use cases. It is a model of the system's intended functionality (use cases) and its environment (actors).

Use cases enable you to relate what you need from a system to how the system delivers on those needs.