Lecture 1: What is software architecture & design?

Software Design:

Using a set of primitive components and subject to constraints to create the specifications of a *software artifact*.

Software Architecture:

Set of structures used to reason about a software system.

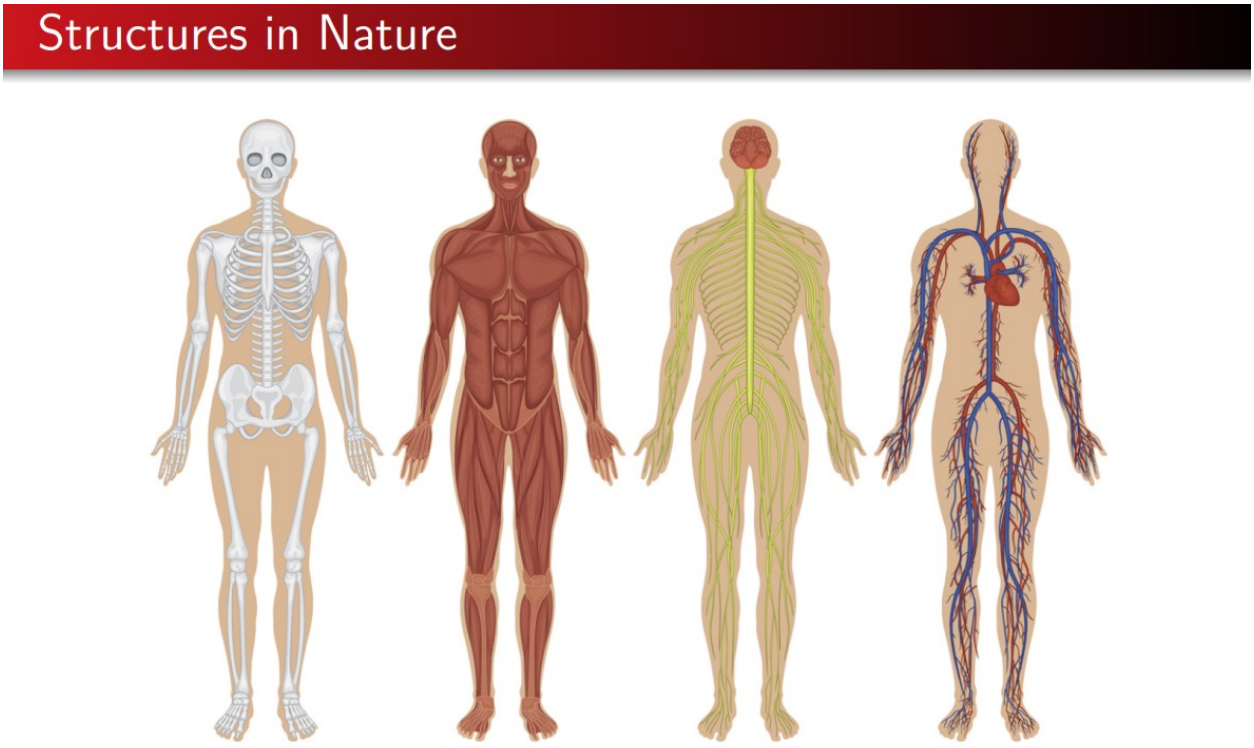
Laws of Software Architecture:

First Law:

everything is a tradeoff.

Second Law:

Why is more important that how.

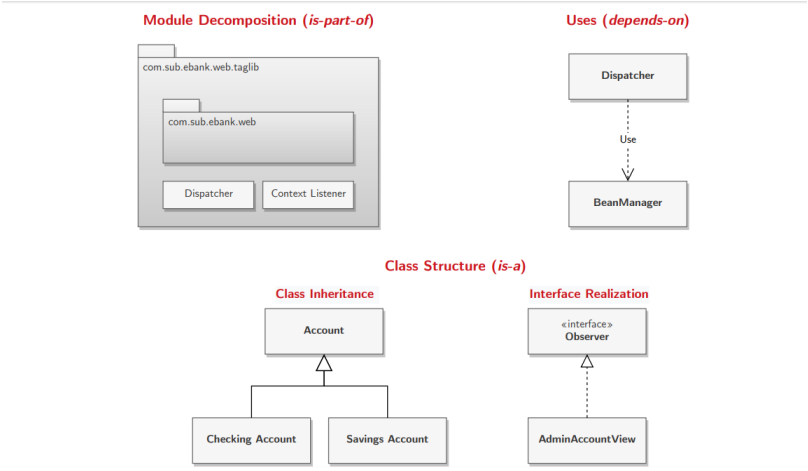


Lecture 2: Architectural Structure & Views

Architectural Structure Types:

a. Module Structures:

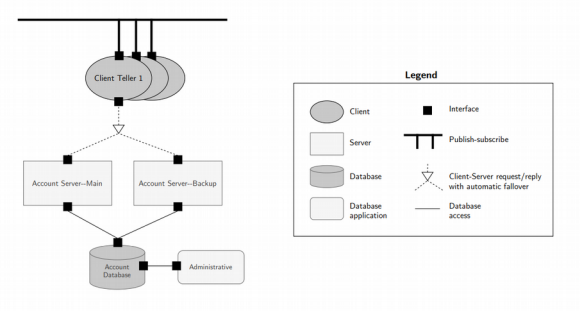
- Show the system as a set of data units that needs to be constructed
  - Example



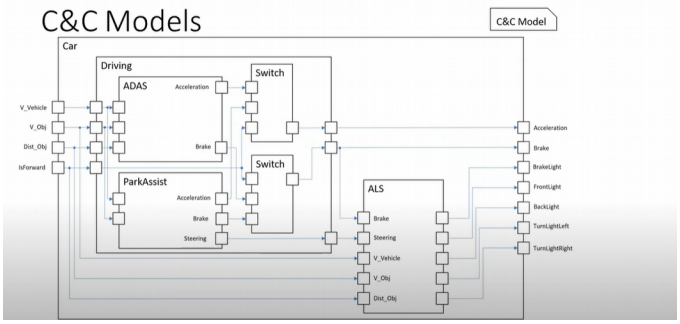
b. C&C Structures:

- Shows the system as a set of elements that have run time behavior(components) and interactions(connectors)

  - Example 1

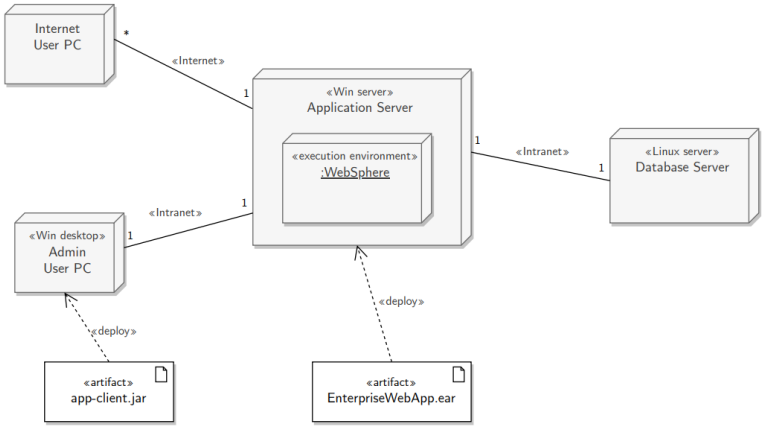


  - Example 2

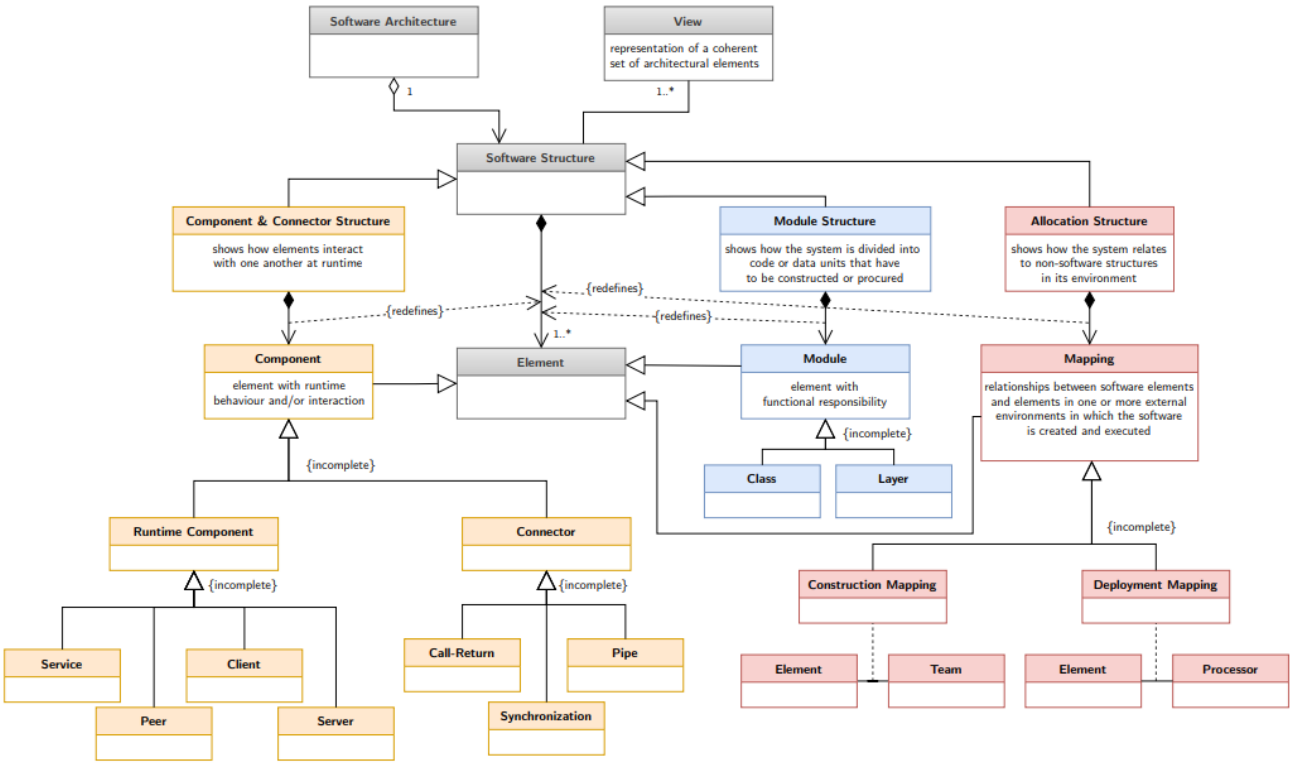


c. Allocation Structures:

- Show how elements from software structures like module structures and the C&C structures relate to the non-software elements like CPU, and networks.
  - Example



Concepts Diagram:



Lecture 3: Architecture Characteristics Defined

Describes a concern critical to the success of the architecture, and therefore the system as a whole, without discounting its importance.

Architecture Characteristics Categories:

a. Operational Characteristics

Availability	Continuity	Performance	Recoverability	Reliability	Robustness	Scalability

b. Structural Characteristics

Configurability	Extensibility	Installability	Leveragability	Maintainability	Portability	Supportability	Localization	Upgradability

c. Cross-Cutting Characteristics

Accessibility	Archivability	Authentication	Authorization	Legal	Privacy	Security	Supportability	Usability

Lecture 4: Identifying Architecture Characteristics

Remember: There is no best design in architecture, only a least worst collection of trade-offs!

Lecture 5: Measuring & Governing Architecture Characteristics

Modularity is the degree to which the system components are separated and then reconstructed again as a Lego building.

Metrics to measure the modularity architectural characteristic:

a. Coupling:

how much do the code artifacts depend on each other.

Has two variations:

Afferent Coupling	Efferent Coupling
Measures the number of incoming connections to each code artifact.	Measures the number of outcoming connections to each code artifact.

Measuring metrics for coupling:

Abstractness:

The ratio of abstract artifacts to concrete artifacts; represents a measure of abstractness versus implementation.

$$A = \frac{\sum m^a}{\sum m^c}$$

Instability:

The ratio of efferent coupling to the sum of both efferent and afferent coupling. Determines the volatility of a code base.

$$I = \frac{C^e}{C^e + C^a}$$

A code base that exhibits high degrees of instability breaks more easily when changed because of high coupling. If a class calls to many other classes to delegate work, the calling class shows high susceptibility to breakage if one or more of the called methods change.

Main Sequence:

A metric that imagines an ideal relationship between abstractness and instability (known as the main sequence)

$$D = |A + I - 1|$$

where a and m are number of attributes and methods respectively, and μ(Aj) is the number of methods accessing attribute Aj .

b. Cohesion:

The extent to which the parts of a module should be contained within the same module; A measure of how related the parts are to one another.

Lack of Cohesion in methods(LCOM):

The sum of sets of methods not shared via sharing fields.

$$LCOM = \frac{1}{a} \sum_{j=1}^{a} \frac{m - \mu(Aj)}{m}$$

c. Connascence

If a change in one would require the other to be modified in order to maintain the overall correctness of the system.

Has 2 variations:

Static Connascence		Dynamic Connascence
Source Code Coupling		Run Time Coupling

Connascence Properties:

Strength

Determined by the ease with which a developer can refactor that type of coupling.(A7e metric)

Locality

Measures how proximal the modules are to each other in the code base.()

Degree

Relates to the size of its impact—does it impact a few classes or many?

Cyclomatic Complexity:

A code-level metric designed to provide an objective measure for the complexity of code, at the function/method, class, or application level.

$$CC = E - N + 2P$$

Computed by applying graph theory to code, specifically decision points, which cause different execution paths.

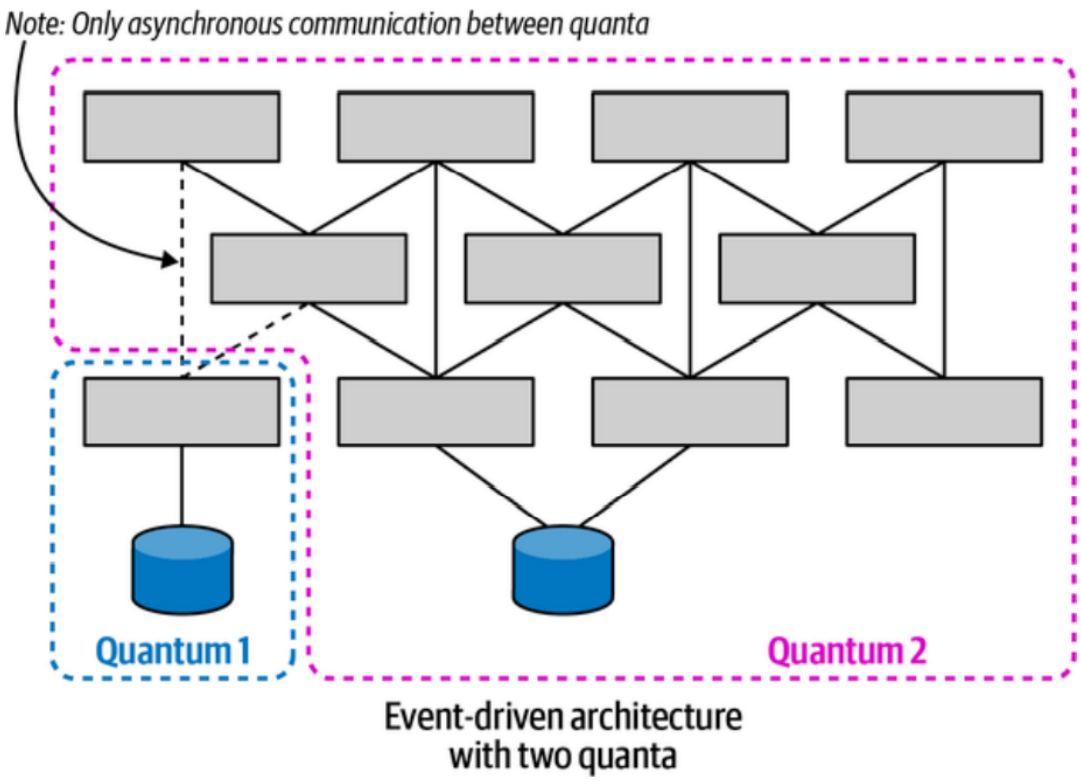
Architecture Fitness Function

Any mechanism that provides an objective integrity assessment of some architecture characteristic or combination of architecture characteristics

Lecture 6: Scoop of Architecture Characteristics

Architecture Quantum:

An independently deployable artifact with high functional cohesion and synchronous Connascence



Event-driven architecture with two quanta

An architect can analyze the differing architecture characteristics per architecture quantum, leading to hybrid architecture design earlier in the process.

Lecture 7: Component Based Thinking: Architecture Partitioning

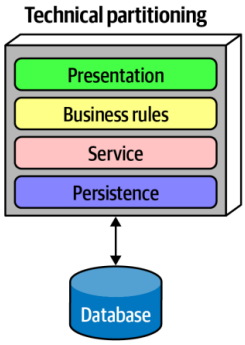
One of the primary decisions an architect must make concerns the top-level partitioning of components in the architecture.

Architects typically think in terms of **components** – the physical manifestation of a **module**.

Architecture Partitioning Types:

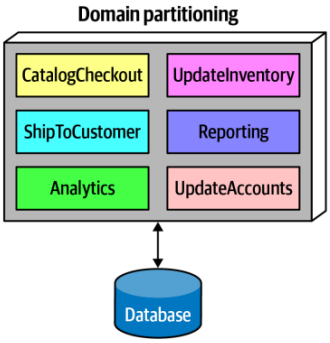
a. Technical partitioning:

Partitioning the software based on technical capabilities.



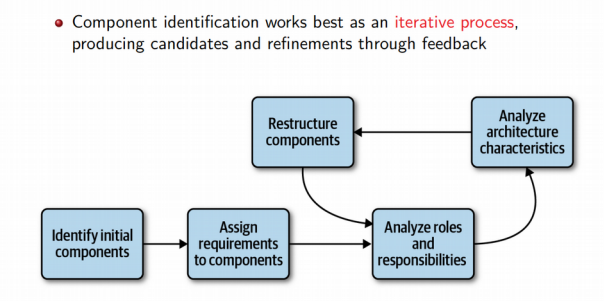
b. Domain Partitioning:

Partitioning the software based on domains and workflows independent and decoupled from one another.



Domain stands **for areas of expertise within the system**. A domain in this context refers to a cohesive subset of functionality or business logic that can be encapsulated within a component.

Lecture 8: Component Based Thinking: Component Identification and Design



Component Identification Steps:

1. Identifying Initial Components:

An architect must somehow determine what top-level components to begin with, based on what type of top-level partitioning they choose.

2. Assigning Requirements to Components:

Aligns requirements (or user stories) to those components to see how well they fit.

This may entail:

- Creating new components.
- Consolidating existing components.
- Breaking components apart because they have too much responsibility.

3. Analyze Roles & Responsibilities:

When assigning stories to components, the architect also looks at the roles and responsibilities elucidated during the requirements to make sure that the granularity matches.

Important Note

Finding the proper granularity for components is one of an architect's most difficult tasks!

- Too fine-grained a component design leads to too much communication between components to achieve results.
- Too coarse-grained components encourage high internal coupling, which leads to difficulties in deployability and testability, as well as modularity-related negative side effects.

4. Analyze Architecture Characteristics

When assigning requirements to components, the architect should also look at the architecture characteristics discovered earlier in order to think about how they might impact component division and granularity.

5. Restructure Components

No one can anticipate all the unknown issues that usually occur during software projects.

Component Design Approaches:

a. Entity Trap:

A common anti-pattern arising when an architect incorrectly identifies the database relationships as workflows in the application.

- Components created with the entity trap also tend to be too coarse-grained.
- Offer no guidance whatsoever to the development team in terms of the packaging and overall structuring of the source code.

b. Actors/Actions Approach:

Architects identify actors who perform activities with the application and the actions those actors may perform.

- Works well for all types of systems, monolithic or distributed.

c. Event Storming:

Architects assume the project will use messages and/or events to communicate between the various components.

- Provides a component discovery technique inspired by domain-driven design (DDD)
- build components around those event and message handlers.
- Works well for distributed architectures that use events and messages.

d. Workflow Approach:

- Much like event storming, but without the explicit constraints of building a message-based system.

Lecture 9: Architecture Style and Patterns

Fundamental Styles and patterns:

a. Big Ball of Mud:

The absence of any Architectural style

Tradeoffs:

- **Deployment:** making changes to one part of the system can have unintended consequences in other parts of the system.
- **Testability:** highly complex, making it difficult to create tests that cover all interactions and edge cases between components or modules.
- **Scalability:** As the system grows in size and complexity, it may become more difficult to add new features or scale the system.
- **Performance:** The lack of structure and governance in such a system can lead to inefficiencies and bottlenecks

b. Unitary Architecture:

Considers hardware and software as one entity.

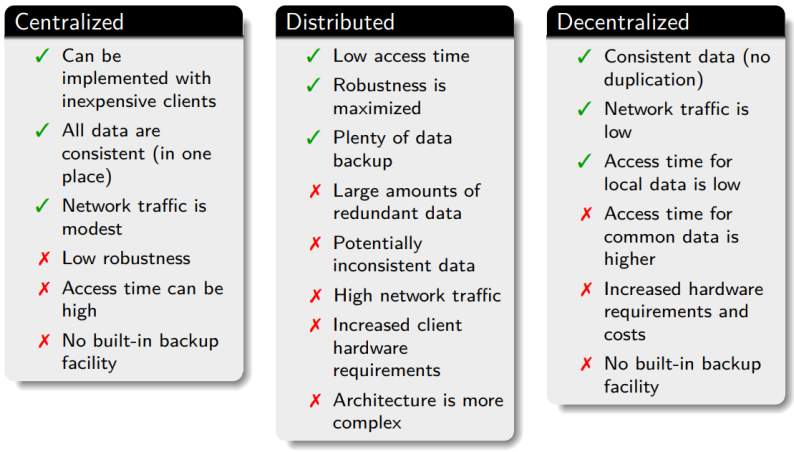
Early implementation. Many software architecture styles later were established to primarily deal with how to efficiently separates parts of the system from each other.

Tradeoffs:

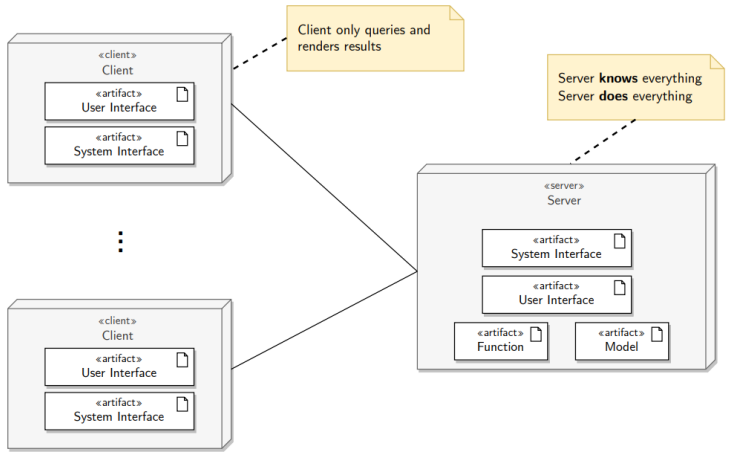
- **Fragility:** A change in any one component may require significant rework of the entire system, leading to a brittle and inflexible architecture.
- **Scalability:** Without modularity or abstraction, it may be difficult or impossible to scale the system horizontally or vertically.
- **Extensibility:** It may be difficult to add new features or functionality to the system without significant changes to the existing architecture.

Ps: Few unitary architectures exist outside embedded systems.

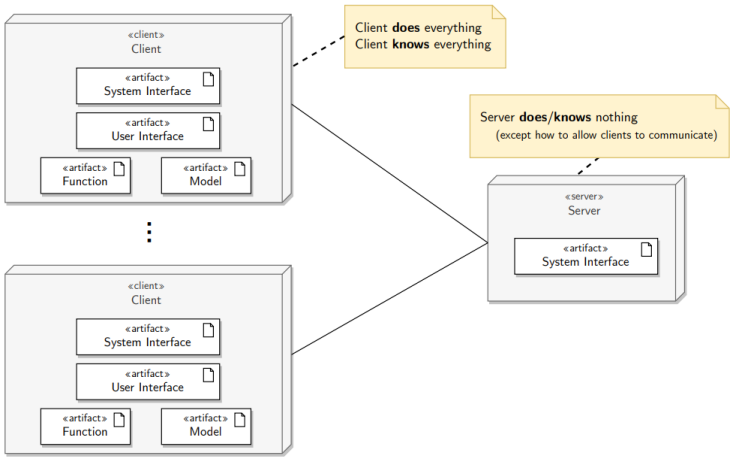
c. Client Server Architecture:



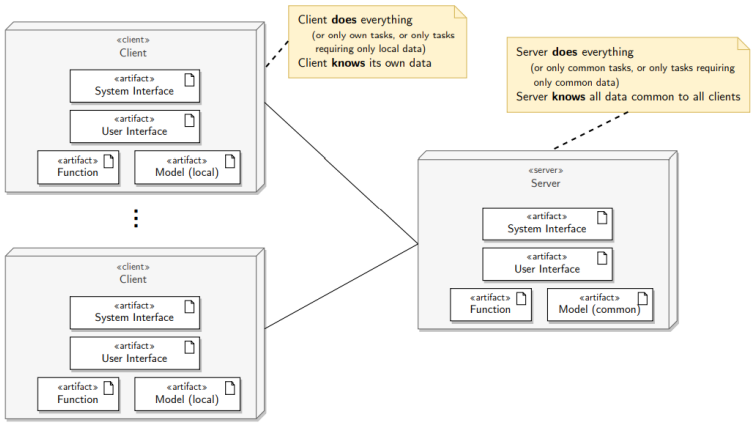
- ***Centralized Client-Server:***



- ***Distributed Client-Server:***

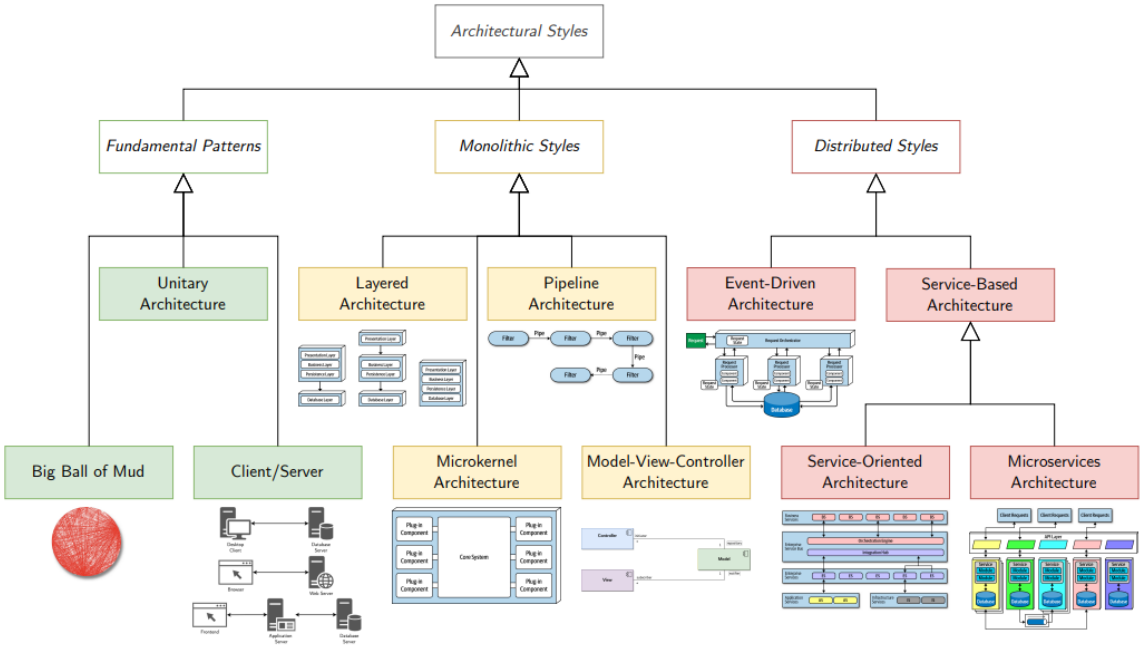


- ***Decentralized Client-Server:***



Lecture 10: Classification of Architecture Styles

Catalog and Classification of Architecture Styles:



Two Classifications of Architecture Styles:

a. Monolithic Architecture Style:

Typically features a single deployable unit and connected to a single database.

b. Distributed Architecture Style:

Typically consists of deployment units (services) running in their own ecosystem, communicating via networking protocols.

Monolithic Architectures	Distributed Architectures
✓ Lower overall cost and complexity	✗ Higher overall cost and complexity
✓ Smaller attack surface	✗ Larger attack surface
✓ Lower levels of communication and collaboration required	✗ Higher levels of communication and collaboration required
✗ Lower performance, scalability, and availability	✓ Higher performance, scalability, and availability
✗ Restricted to a single architecture quantum	✓ Can handle multiple architecture quanta

Fallacies of Distributed Computing:

Fallacy #1: The Network is Reliable

- **Tradeoff: Reliability**

Fallacy #2: Latency is zero

When using any distributed architecture, architects must know the average and “long tail” latency. Helps determine whether a distributed architecture is feasible.

- **Tradeoff: Feasibility**

Fallacy #3: Bandwidth is Infiniti

- **Tradeoff: Usability, Performance**

Fallacy #4: The Network is Secure

- **Tradeoff: Security, Privacy**

Fallacy #5: The Topology Never Change

- **Tradeoff: Supportability, Performance**

Fallacy #6: There is Only One Administrator

- **Tradeoff: Supportability, Usability**

Fallacy #7: Data Transportation Cost is Zero

- **Tradeoff: Feasibility**

Fallacy #8: The Network is Homogeneous

Networks infrastructure are often built with hardware from multiple vendors that sometimes “do not play well” together

- **Tradeoff: Reliability**

Lecture 11 & 12 : Monolithic Architecture Styles

Monolithic Architecture Styles:

a. Layered Architecture Style:

Each layer is independent of the other layers, thereby having little or no knowledge of the inner workings of other layers in the architecture.

Architecture Characteristic	Rating
Partitioning Type	Technical
Number of Quanta	1
Deployability	★
Elasticity	★
Evolutionary	★
Fault Tolerance	★
Modularity	★
Overall Cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★

Types of Layers:

- **Closed Layer:**

  Requests between layers can't bypass closed layers in between without going through it.

- **Open Layer:**

  Requests between Layers can bypass open layers.

b. Pipeline Architecture Style:

Processing data streams.

The data source, the filters and the data sink are connected sequentially by pipes.

The sequence of filters combined by pipes is called a processing pipeline.

Architecture Characteristic	Rating
Partitioning Type	Technical
Number of Quanta	1
Deployability	★★
Elasticity	★
Evolutionary	★★★
Fault Tolerance	★
Modularity	★★★
Overall Cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★★

c. Model-View-Controller Architecture Style:

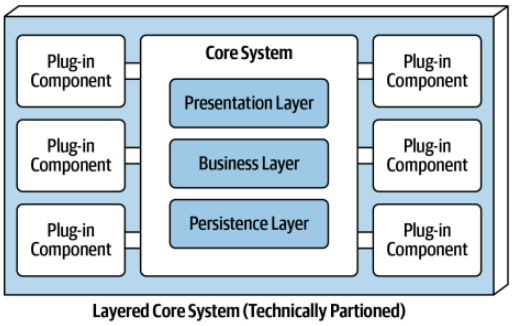
Architecture Characteristic	Rating
Partitioning Type	Technical
Number of Quanta	1
Deployability	★★★
Elasticity	★
Evolutionary	★★★
Fault Tolerance	★★★
Modularity	★★★★
Overall Cost	★★★★★
Performance	★★★
Reliability	★★★
Scalability	★
Simplicity	★★★★
Testability	★★★★

d. Microkernel Architecture Style:

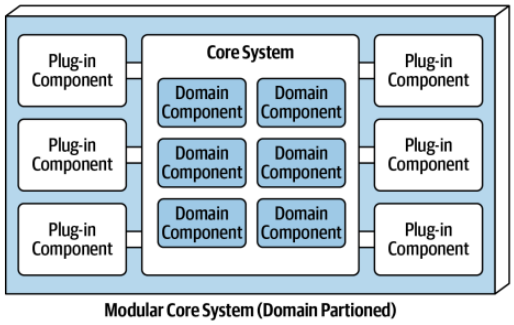
Microkernel is responsible for maintaining common functionality, system-wide resources such as files or processes, and providing interfaces that enable other components to access its functionality.

Architecture Characteristic	Rating
Partitioning Type	Domain and Technical
Number of Quanta	1
Deployability	★★★
Elasticity	★
Evolutionary	★★★
Fault Tolerance	★
Modularity	★★★
Overall Cost	★★★★★
Performance	★★★
Reliability	★★★
Scalability	★
Simplicity	★★★★
Testability	★★★

Microkernel Architecture Topology Variants:



Layered Core System (Technically Partioned)



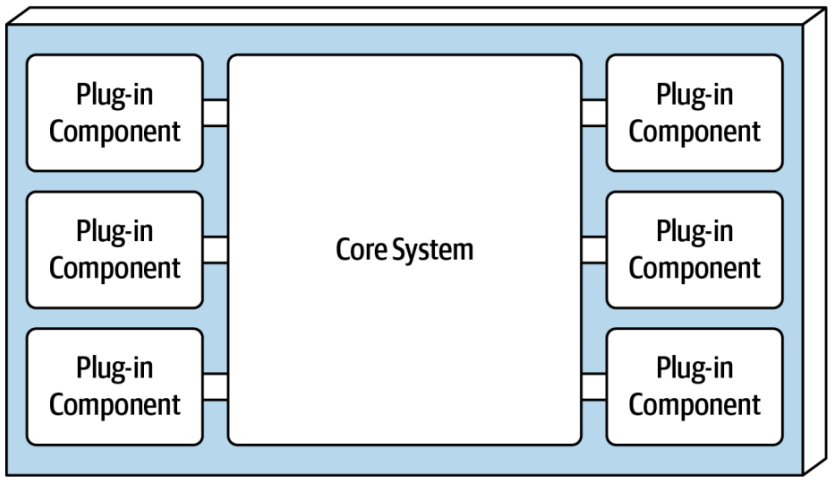
Modular Core System (Domain Partioned)

Plug-In Registry:

An implementation enabling the core system to know about which plug-in modules are available and how to get to them.

Contract:

The expected behavior, input data, and output data returned from the plug-in component.



Check your understanding:

L01 -> Exercise (Analyzing Trade-offs):

Consider an item auction system, as illustrated in below, where someone places a bid for an item up for auction. The Bid Producer service generates a bid from the bidder and then sends that bid amount to the Bid Capture, Bid Tracking, and Bid Analytics services. This could be done by:

- **using a topic in a publish and-subscribe messaging fashion?**
- **using queues in a point-to-point messaging fashion?**

Which one should the architect use?

Using a topic in a publish and subscribe messaging. Since it is going to be sending the bid amount to multiple subscribers

If reliable delivery to a single service is more important, then a point-to-point messaging pattern may be a better choice.

L02 -> Exercise (Architectural Structures):

Consider the software architecture of the Chromium Resource Loading process shown below.

Which structures are shown?

Given its purpose, which structures should have been shown?

What analysis does the architecture support?

What questions do you have that the representation does not answer?

--

L03 -> Exercise (Implicit vs. Explicit Characteristics):

Identify and describe some examples of implicit and explicit architecture characteristics in the context of an example system (such as an ATM, mobile application, or some other system you are familiar with).

Easy

Domain concern	Architecture characteristics
Mergers and acquisitions	Interoperability, scalability, adaptability, extensibility
Time to market	Agility, testability, deployability
User satisfaction	Performance, availability, fault tolerance, testability, deployability, agility, security
Competitive advantage	Agility, testability, deployability, scalability, availability, fault tolerance
Time and budget	Simplicity, feasibility

L04 -> Exercise (Room with A View):

Description: A large hotel reservation company wants to build the next generation hotel reservation and management system specifically tailored to high-end resorts and spas where guests can view and reserve specific rooms.

Users: Guests (hundreds), hotel staff (less than 20)

Requirements:

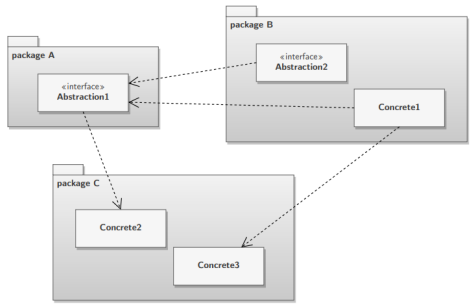
- System will be web-based and hosted by the reservation company; Registration can be made via web, mobile, phone call, or walk-in.
- Guests have the ability to either book a type of room (standard, deluxe, or suite) or choose a specific room to stay in by viewing pictures of each room and its location in the hotel.
- Must be able to maintain room status (booked, available, ready to clean, etc.) as well as when the room will be needed next.
- Must have housekeeping management functionality so that cleaning and maintenance staff can be directed to various rooms based on priority and reservation need using proprietary devices supplied by the reservation company attached to the cleaning carts.
- Standard reservation functionality (e.g., payments, registration info, etc.) will be done by leveraging the existing reservations system.

Additional context: Peak season is quickly approaching, so the system must be ready quickly or we have to wait until next year!''; Company is also investing heavily in cutting edge technology like smart room locks that open via a cell phone; Only interested in the high-end market; Salespeople have tremendous clout in the organization; people often scramble to make their promises true.

- **Identify the explicit architecture characteristics from the domain concerns, specifically the domain-level predictions about expected metrics.**
    - Availability -> needed asap.
    - Elasticity -> Must handle bursts of requests since it will be available in peak season.
    - Privacy -> smart locks purpose
    - Authentication->Locks mechanism (I think this should be encapsulated in privacy too)
    - Portability -> Users will be using smart phones to use the smart lock.
    - Usability -> Needs to have a UI that is easy to use and desirable.
- **Identify the explicit architecture characteristics from the requirements. Be sure to consider each part of the requirements to see if it contributes to an architecture characteristic.**
    - Configurability -> manage room types.
    - Usability -> guests will use it.
    - Reliability -> it has to reflect whatsup for real.
    - Supportability -> housekeeping is on this too.
    - Installability -> will be using other add on services.
    - Leveragability -> leveraging the existing reservations system.
- **Identify the implicit architecture characteristics based on your knowledge of software solutions of managing reservation systems.**
    - Security -> Always a req
    - Scalability -> not necessarily needed here but is usually a concern.
    - Performance -> obvious, any application actually
    - Elasticity -> burst of request
    - Maintainability -> we need to keep it together you know.
    - Availability: Reservation systems need to be available 24/7, as users may need to make reservations at any time.

L05 -> Exercise (Distance from the Main Sequence):

Consider the following design. Calculate the distance from the main sequence for each package.



	Package A	Package B	Package C
Abstractness(A)	1/1 = 1	½ = 0.5	0/3 = 0
Instability(I)	1/3 = 0.333	3/3 = 1	0/2 = 0
Distance From Main Sequence(D) D = |A + I − 1|	|1 + 0.333 − 1| = 0.3333 Perf	|0.5 + 1 − 1| = 0.5 Zone of uselessness	|0 + 0 − 1| = 1 Zone of pain

L05 -> Exercise (Cyclomatic Complexity):

Calculate the cyclomatic complexity of the following C-like code:

```
i = 0;
n = 4;
while (i < n−1) do
        j = i + 1;
        while (j < n) do
                if (A[i ] < A[j]) swap(A[i ], A[j ]) ;
        end do;
        j = j+1;
end do;
```

M = E - N + 2

In the code above, we have:

E = 4;N = 3;P = 1

Thus, we have:

M = 4 - 3 + 2(1) = 3

Therefore, the cyclomatic complexity of the code above is 3.

L06 -> Exercise (Understanding Architectural Quanta):

Assume a system consisting of a single user interface with four independently deployed services, each containing its own separate database.

How many quanta would this this system have?

5 Qantas -> 4 independently deployable services + 1 user interface

Each of which can be separated into its own quanta.

Recall that an architectural quantum is a collection of components (an artifact) identified at the system level that has a high functional cohesion, synchronous Connascence, and independently deployable.

L06 -> Exercise (Identifying Architectural Quanta):

Assume a system with an administration portion managing static reference data (such as the product catalog, and warehouse information) and a customer-facing portion managing the placement of orders.

- **How many quanta should this system be and why?**
  2 Quantas -> because that is the best quanta separation, we can have based on what we know and the architectural quanta requirement.
- **If you envision multiple quanta, could the admin quantum and customer-facing quantum share a database?**
  It hurts the cohesion of the system (not highly functional cohesion anymore) but yo, everything is a tradeoff remember.

L07 -> Exercise (Technical vs. Domain Partitioning):

- **What is the difference between technical partitioning and domain partitioning? Provide an example of each.**
  Straight forward
- **What is the advantage of domain partitioning?**
  Easy to migrate data and components to distributed architecture.
- **Under what circumstances would technical partitioning be a better choice over domain partitioning?**
  If the architecture pattern is the layered architecture pattern since it aligns well with it.

L08 -> Exercise (Choosing a Component Decomposition Approach):

- **When might you choose the workflow approach over the Actor/Actions approach when identifying core components?**
  a. The workflow approach is well-suited to systems that involve a series of highly structured, repeatable processes, such as those found in manufacturing, logistics, or finance. In these cases, the focus is on automating and optimizing the steps in the process, rather than on responding to dynamic or unpredictable events.
  b. The workflow approach can be effective in systems where data is a key driver of the process. In these cases, the focus is on moving data through a series of steps or stages in order to transform or analyze it.

For example, a data analytics system might use a workflow approach to perform a series of data transformations and analyses on a large dataset.

L09 -> Exercise (Choosing a Client-Server Distribution):

Suppose you are building a software solution for a company whose employees need to utilize the same, or similar, client applications and files. The company has listed the required business applications needed by each employee, each of which processes company data in many different complex ways. Depending on their role within the company, different employees may need to perform different data processing task. The company expects that some employees will work from home from time-to-time, and the company wants to ensure that the employees don't have to worry about staying connected the whole workday. If the employee loses their internet connection, it is desirable that employees can continue working. While the company wants to minimize network traffic, effective data management and backups are critical to their business operations.

Which client/server distribution pattern would you recommend for this system. Justify your choice.

Decentralized is bad since it is bad in terms of backup. And backup is critical.

Centralized might be the answer since it is the best mediator between the two.

If we really care about backups that much distributed is the safest option but the network traffic is going to be so taxing

L10 -> Exercise (Stamp Coupling):

Stamp coupling (also known as data-structure coupling) occurs when modules share a composite data structure (such as record). When a composite data structure is shared between two modules, some of the fields in the data structure may not even be used.

Suppose that requests for the wish list items in the example on Slide 13 happen about 2,000 times a second.

- **How much bandwidth is consumed per second for these requests?**
  In this scenario, Service A needs to make an interservice call to Service B to get the customer name, but Service B returns 45 attributes totaling 500 kb when only 200 bytes (presumably for the customer name) are needed. This can result in unnecessary network traffic and increased latency.
  To calculate the bandwidth consumed per request for this scenario, we need to consider the amount of data being transferred between Service A and Service B. Assuming that the request and response headers are negligible in size, the total amount of data being transferred per request is:
  500 KB (from Service B to Service A) + 200 bytes (from Service A to the client) = 500,200 bytes

- **If the Service B were to only pass back the data needed by Service A, how much bandwidth is consumed per second for these requests?**
  If Service B were to only pass back the data needed by Service A (i.e., only the 200 bytes for the customer name), then the amount of data being transferred per request would be:
  200 bytes (from Service B to Service A) + 200 bytes (from Service A to the client) = 400 bytes

This would significantly reduce the amount of data being transferred between the two services, resulting in lower network traffic and improved performance.

L11 -> Exercise (Monolithic Architecture Styles):

- **What is the difference between an open layer and a closed layer?**
  Open layer allows for the movement of data easily through it, closed layer does not.
- **Can pipes be bidirectional in a pipeline architecture?**
- Yes, they can be bidirectional but It is not the usual and it does have significant tradeoffs.

L12 -> Exercise (Monolithic Architecture Styles):

- **Why is the microkernel architecture always a single architecture quantum?**
  The microkernel architecture is always a single architecture quantum because it is a highly modular and flexible architecture style that separates the core system services from the application-specific functionality. This allows the system to be easily extended or customized by adding or removing services as needed without affecting the overall system stability. Since the microkernel provides only the essential system services, the rest of the functionality is provided by user-level processes or modules, which can be easily replaced or upgraded without affecting the stability of the core system.
- **What class of systems is the MVC architecture style best suited?**
  The Model-View-Controller (MVC) architecture style is best suited for complex user interface (UI) applications where there is a need to separate the UI presentation logic from the business logic and data model. MVC is a popular architecture style for web applications, desktop applications, and mobile apps. By separating the UI presentation logic from the business logic and data model, MVC allows for easier maintenance, extensibility, and testing of the application. MVC is also suitable for applications that require multiple views of the same data, as the data model can be easily shared across multiple views.