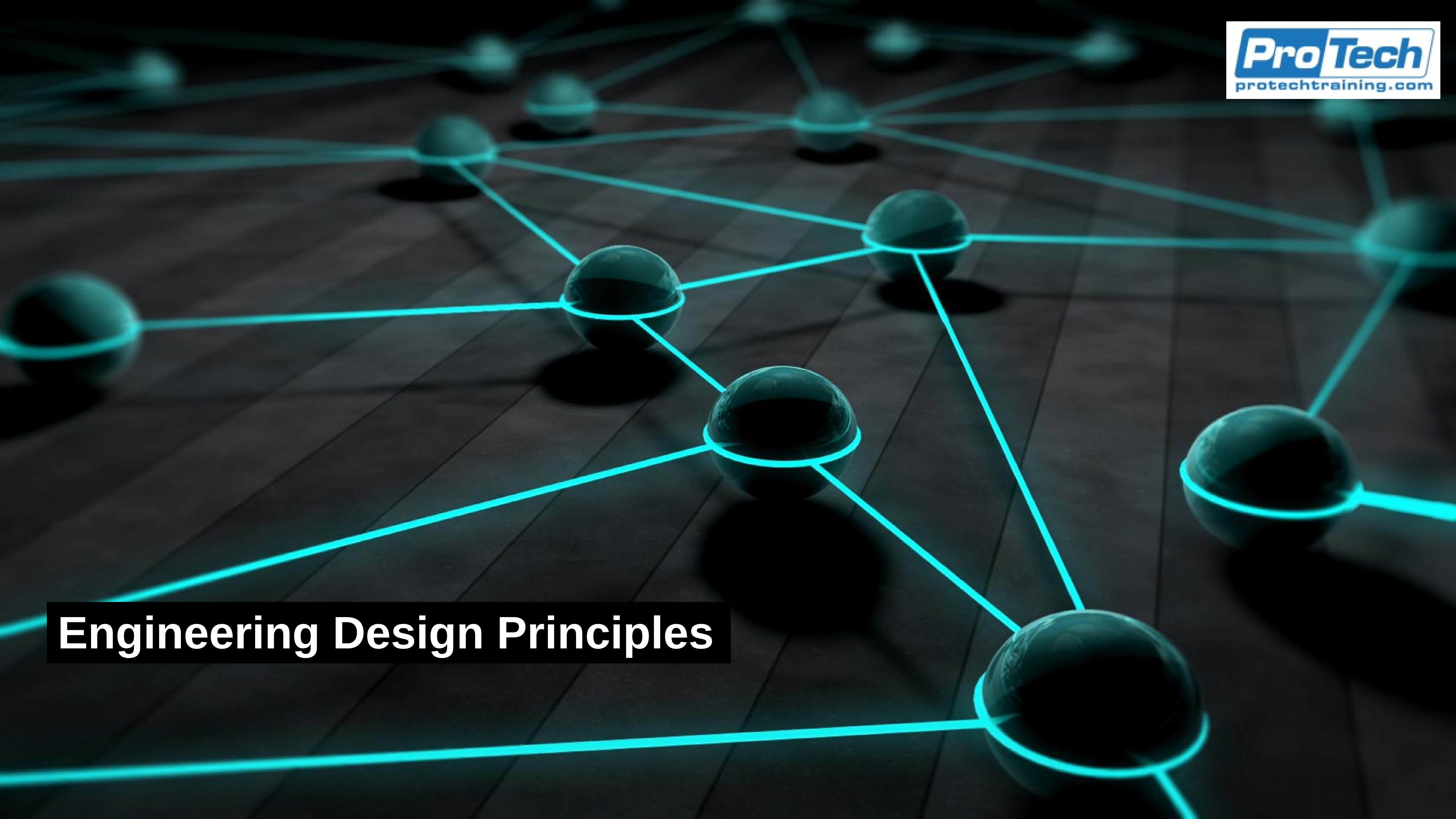


ProTech
protechtraining.com

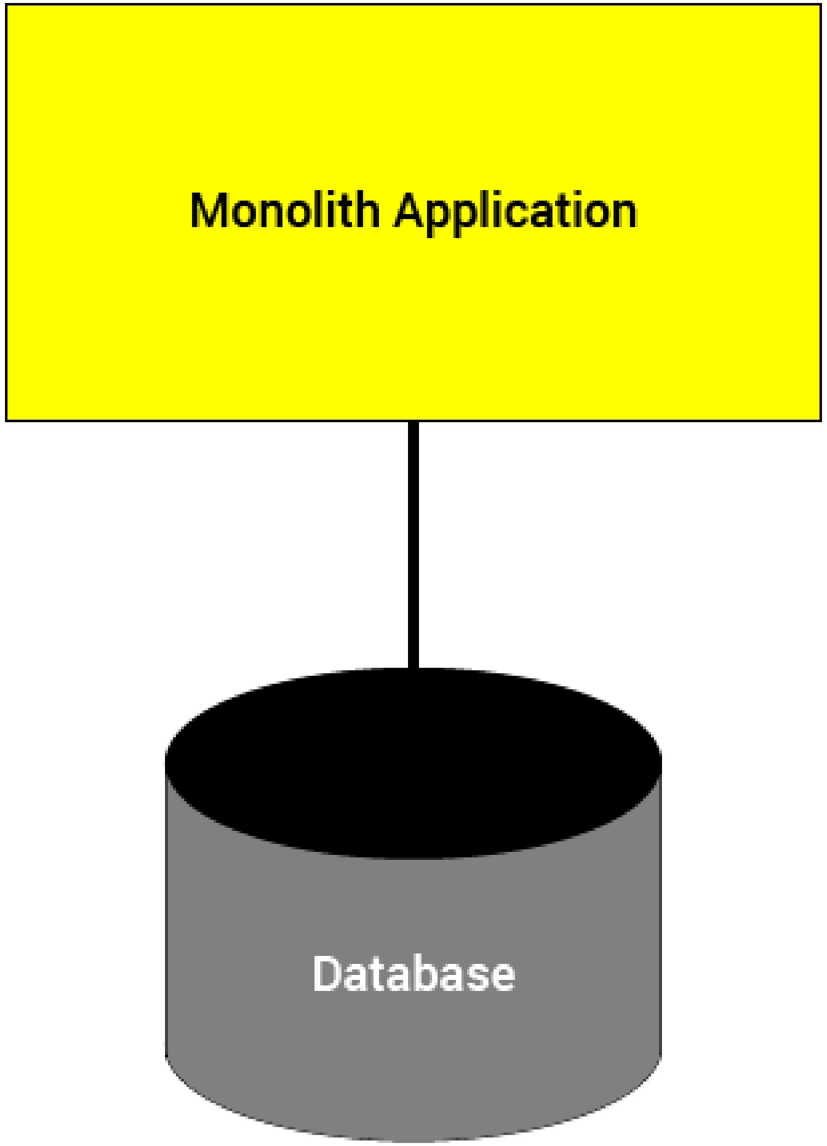
Engineering Design Principles

Design Principles

- Design principles are high-level guidelines or best practices that guide how we structure, organize, and create systems
    - Those systems can be software architectures, mechanical devices, user interfaces, or business processes.
    - Design principles are universal heuristics that help designers create solutions that are maintainable, scalable, and understandable.
    - Think of them as the “laws of good design:” not rigid rules, but patterns of thinking.
    - Generally following these principles leads to more efficient and effective engineering processes
- Many of the principles are derived from watching how natural systems evolve to solve problems of scale and complexity
    - Natural systems include those in nature and those that occur in human society, like the organization of a business

Monolith Application

- Characterized by a single code base
  - May be modular at the programming language level
- Integrated with a single database
  - All code uses a common schema
- “Monolith” means
  - If a change to the code base or to the data schema is made
  - Then the entire application needs to be redeployed

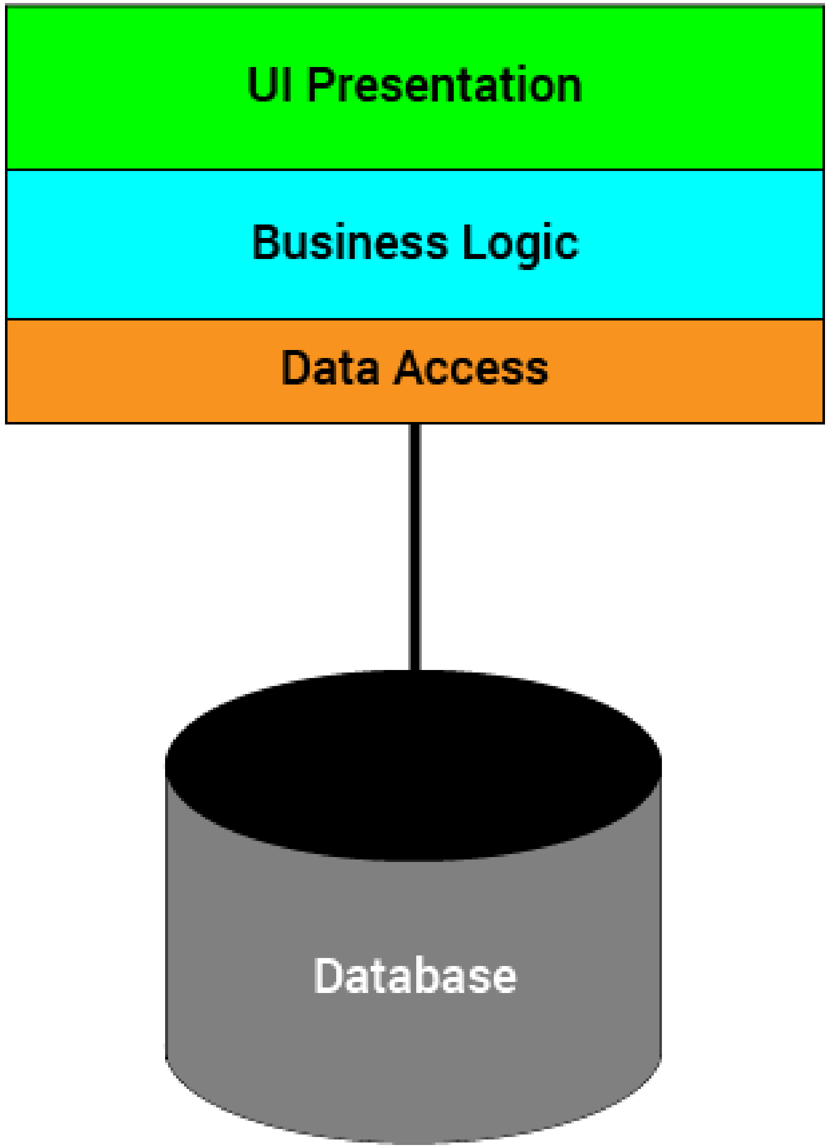


Business Analogy

- Start-up businesses are monoliths
  - There are a few people who do everything
  - The enterprise is small enough that this model works
  - It’s actually counterproductive to have a highly structured departmental organization with just a few employees
- Early versions of applications are similar
  - Simple enough that all of the code is manageable
  - Single code base for the whole app
  - Flat or minimal architectural structure

Modular Monolith Application

- Code is modularized
- Organized along job descriptions
  - Front end dev has their modules
  - Programmers have their modules
  - Data engineers have their modules
- Shows up historically as a n-tier architecture

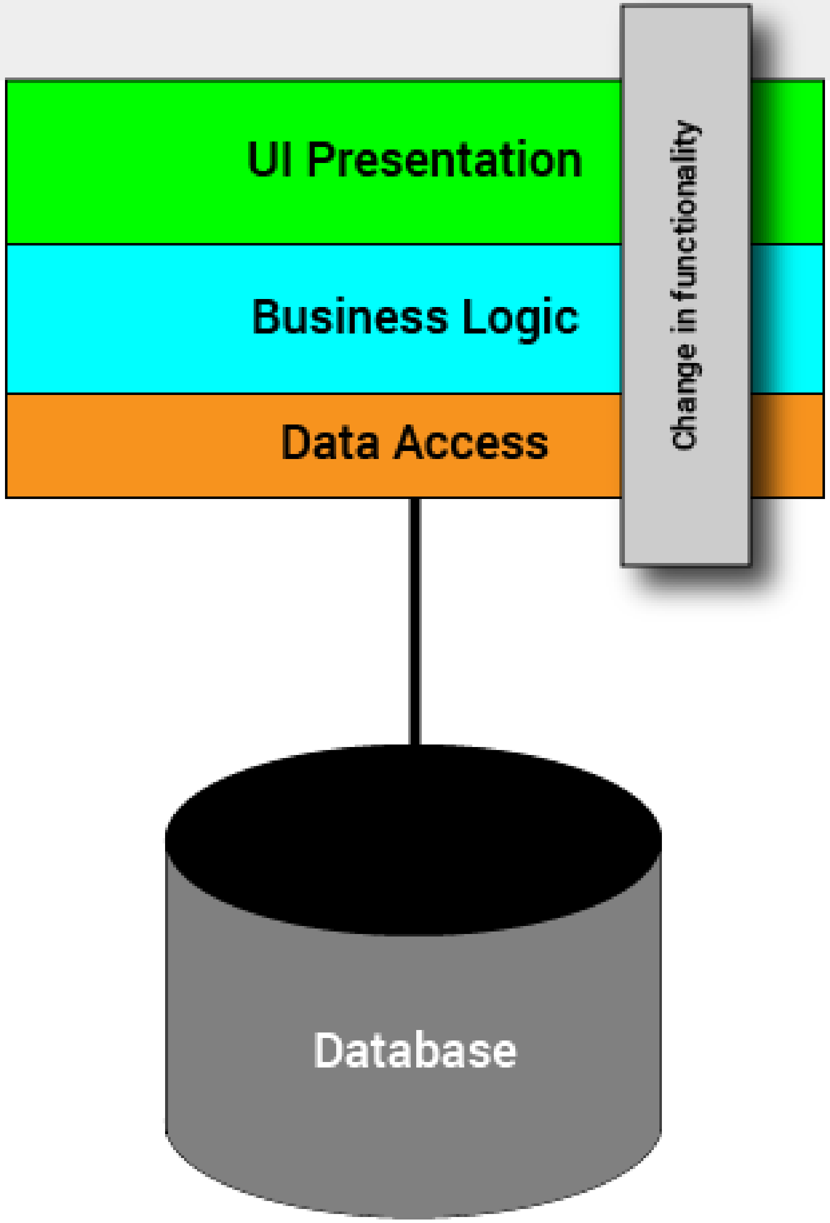


Business Analogy

- As a start-up grows, it adds more people
  - The original organization starts to become ineffective
  - The entrepreneurial “all hands-on deck” model doesn’t scale well
  - Processes become chaotic
  - Difficult to manage
  - Productivity stalls
- At some point the business must modularize
  - Usually by creating specialized departments
  - Accounting, sales, HR, etc.

Modular Monolith Application

- The application is still a monolith
- A change in functionality
  - Affects each layer because related changes have to be made in each layer
- Interacts with the data model
  - The data model may constrain what changes can be made
  - Changing the data model might break other parts of the app
- The modules and the database often show high coupling
  - Due to how the modules are defined



Scaling in Systems

- Scaling is an increase in size or quantity along some dimension
- Can take place in the development or operations space
- Development scaling
  - Increase in complexity, functionality or volume of code
  - These dimensions are often related
  - Business analogy is a company increasing the range of services and products they offer or expanding into different markets (like a Canadian company expanding into Europe)
- Operational scaling
  - Increase in the amount of activity of a system
  - Throughput, load, transaction time, simultaneous users, etc
- Traditional monoliths tend to be scalable only to a limited degree
  - There is a certain level of scaling after which they become unmaintainable

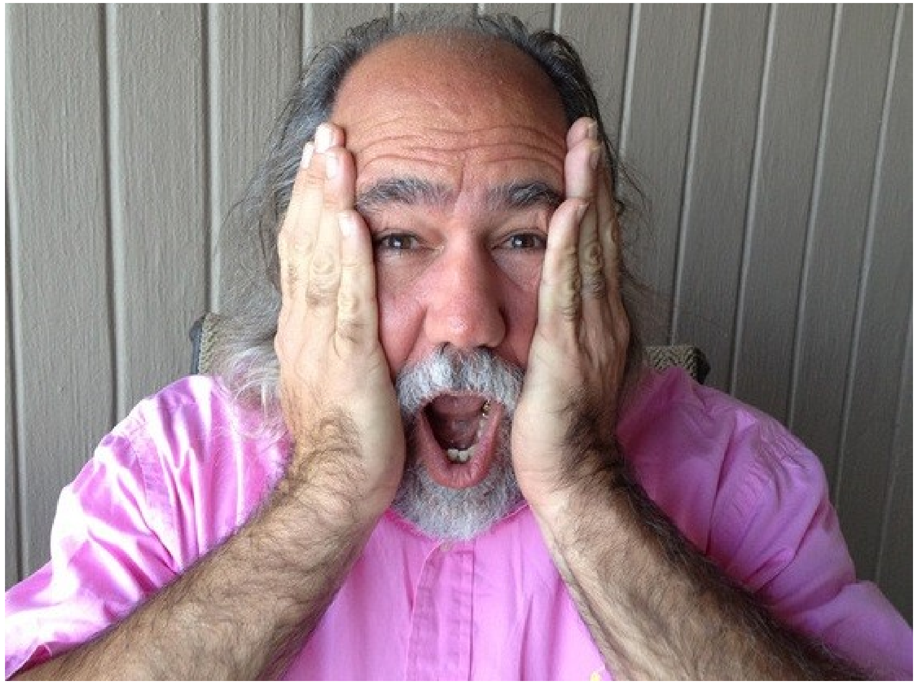
Complexity

Mission critical software tends to have a long lifespan, and over time, many users come to depend on their proper functioning. In fact, the organization becomes so dependent on the software that it can no longer function in its absence. At this point, we can say the software has become industrial-strength.

The distinguishing characteristic of industrial strength software is that it is intensely difficult, if not impossible, for the individual developer to comprehend all of the subtleties of its design.

Stated in blunt terms, the complexity of such systems exceeds the human intellectual capacity. Alas, this complexity we speak of seems to be an essential property of all large software systems. By 'essential' we mean that we may master this complexity, but we can never make it go away.

Grady Booch



IT Failures and Complexity

The United States is losing almost as much money per year to IT failure as it did to the [2008] financial meltdown. However the financial meltdown was presumably a onetime affair. The cost of IT failure is paid year after year, with no end in sight. These numbers are bad enough, but the news gets worse. According to the 2009 US Budget [02], the failure rate is increasing at the rate of around 15% per year.

Is there a primary cause of these IT failures? If so, what is it?.... The almost certain culprit is complexity.... Complexity seems to track nicely to system failure.

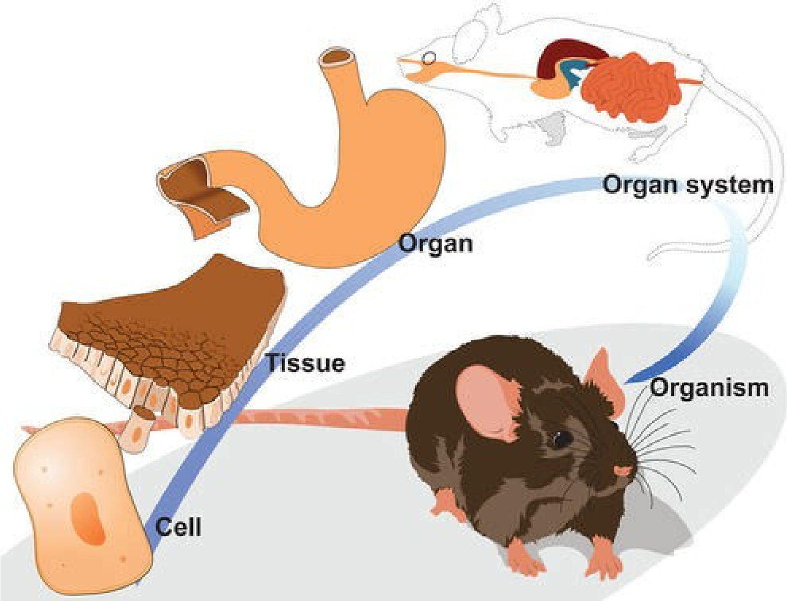
Once we understand how complex some of our systems are, we understand why they have such high failure rates.

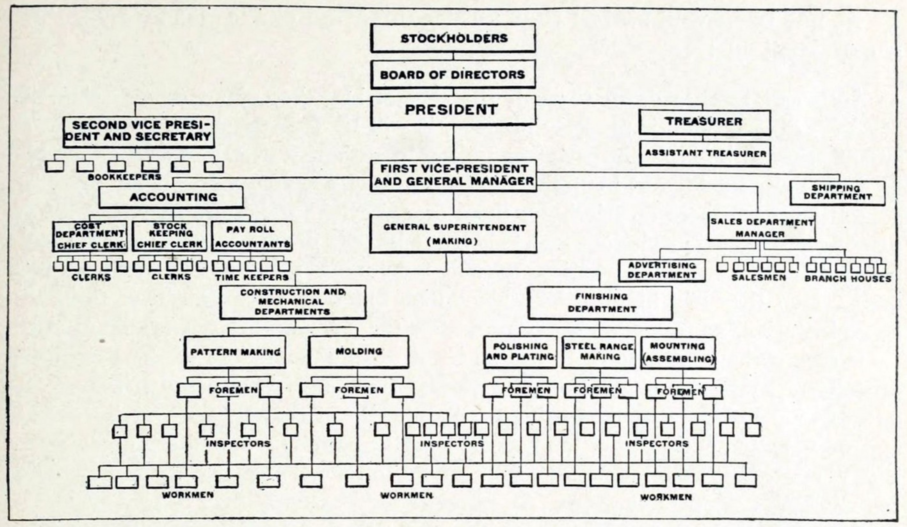
We are not good at designing highly complex systems. That is the bad news. But we are very good at architecting simple systems. So all we need is a process for making the systems simple in the first place.

Roger Sessions

Natural Systems Organization

- Naturally occurring systems scale successfully
  - Biological processes, social organization etc.
- They all show similarities in organization regardless of domain
- Natural occurring systems tend to be
  - Recursive in structure
  - Hierarchical or layered
  - Modular at each layer
  - Loosely coupled and highly cohesive





Operational Complexity in Practice

- In production, systems have to scale operationally
- Consider a diner
  - During the lunch and dinner hour rush, the number of servers and cooks have to scale up
  - There has to be clear boundaries on what each role does
  - During non-peak hours, the amount of staff can be scaled down

Operational Complexity in Practice

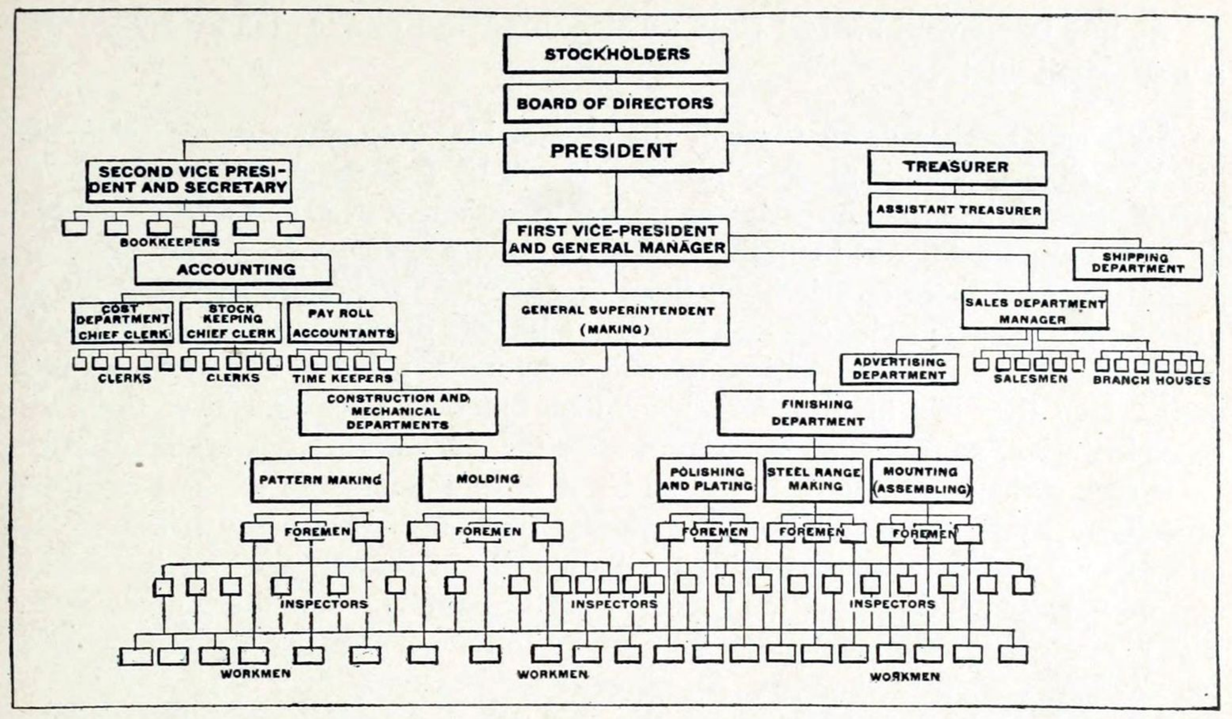
- Operations are specialized
- Tasks are broken down into sub-tasks, and sub-tasks broken down into sub-sub-tasks, and so on
- At some point
  - Specialized systems or agents perform an individual sub-task
  - These are all coordinated
- For example, the specialized positions on a sports team

Complex Systems

Frequently, complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached

Courtois

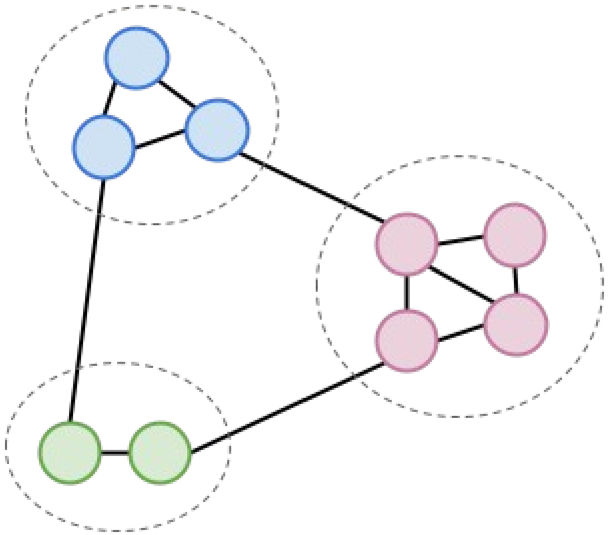
On Time and Space Decomposition of Complex Structures
Communications of the ACM, 1985, 28(6)



Complex Systems

Intra-component linkages are generally stronger than inter-component linkages. This fact has the effect of separating the high frequency dynamics of the components – involving the the internal structure of the components – from the low frequency dynamics – involving the interaction among components

Simeon



Complex Systems

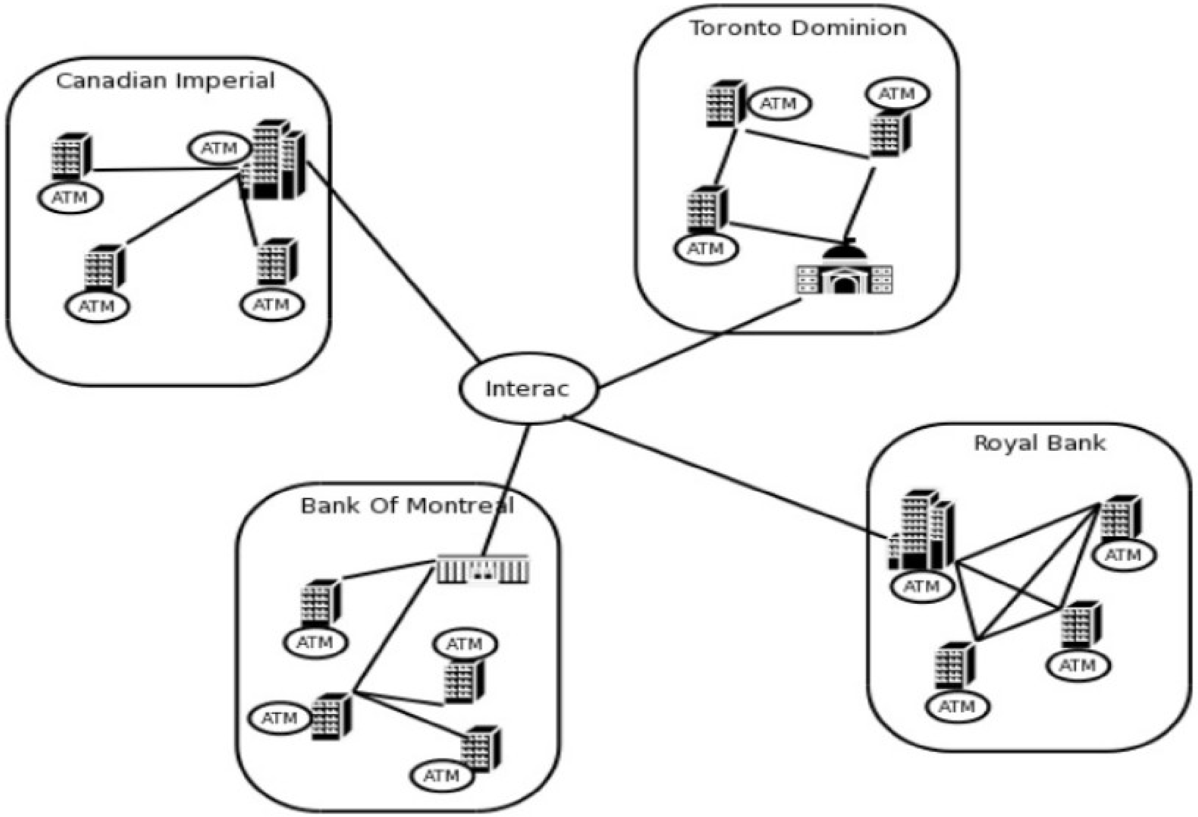
Hierarchical systems are usually composed of only a few different kinds of sub-systems in various combinations and arrangements

Simeon

A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and can never be patched up to make it work. You have to start over, beginning with a simple working system.

John Gall

Systemantics: How Sytems Really Work an How They Fail 1975



Principles

- A design principle is a guideline abstracted from observing how complex systems are organized
  - When applied, they have proven to produce better applications or products along a number of different dimensions
  - Especially when it comes to managing complexity and scaling.
- These principles are used in all areas of design and engineering
  - This is a sampling of a number of these principles in use
  - There is no definitive list, just informal best practices
  - They are always adapted to apply to a specific domain, like software or architecture, etc

Modular

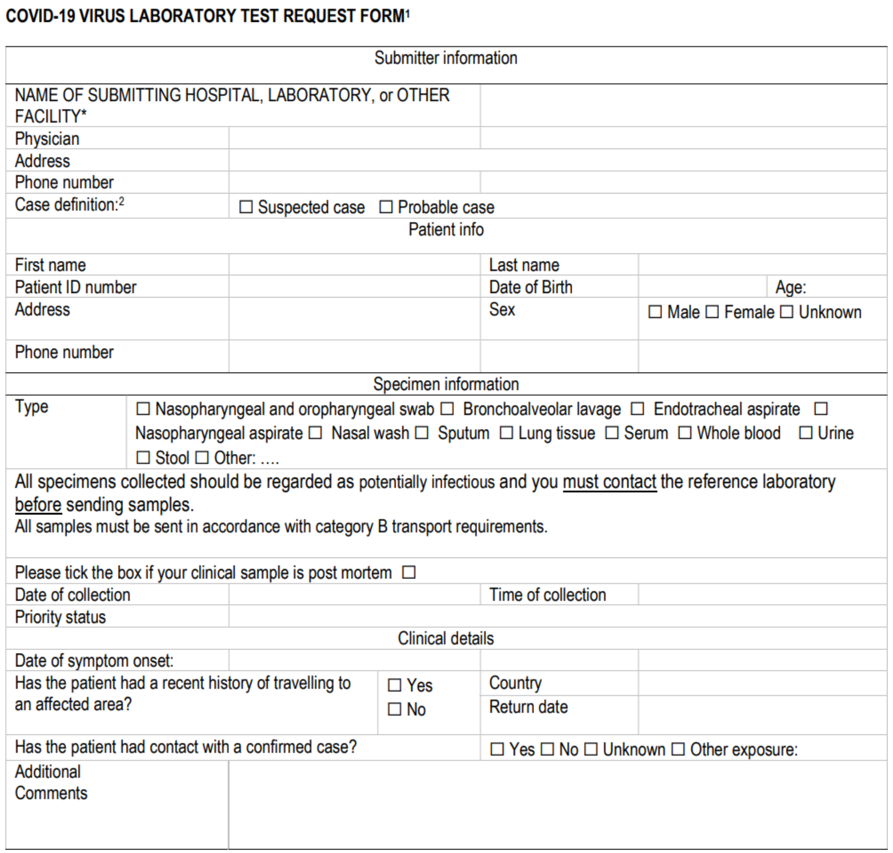
- The system is divided up into components
  - Component may be further divided into components
  - And so on, often resulting in a layered design or a recursive architecture
- Each component is separated into two parts
  - The interface, which is how the component interacts with other component
  - The implementation, the parts of the component that does the actual work
- Encapsulation
  - The component is designed so that the implementation is not visible from outside
  - Other components can only communicate with the component through the interface
  - This allows us to reorganize or redesign the implementation without affecting the inter-component interactions

Real Life Example

- A hospital provides a health care service
- Made up of individual components
  - Laboratory
  - X-Ray
  - Pharmacy
  - Medical Staffing
- Each component:
  - Specializes in a specific domain activity
  - Each component operates autonomously
  - Each component is encapsulated

Real Life Example

- Components request services from each other
  - They do not need to know the internal workings of the other component
- Requests are made through interfaces
  - Often called APIs in software engineering
- These are often paper based in the real world
  - Requisitions and official forms and paperwork used to make requests of a service
  - Response to a request is often a report of some kind

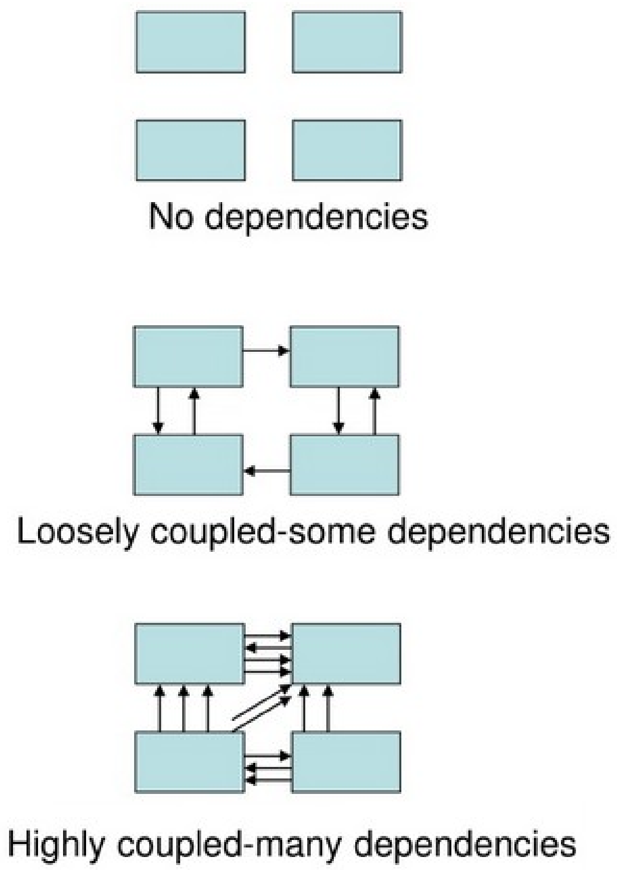


COVID-19 VIRUS LABORATORY TEST REQUEST FORM[1]

Submitter information			
NAME OF SUBMITTING HOSPITAL, LABORATORY, or OTHER FACILITY*			
Physician			
Address			
Phone number			
Case definition:[2]	☐ Suspected case ☐ Probable case		
Patient info			
First name		Last name	
Patient ID number		Date of Birth	Age:
Address		Sex	☐ Male ☐ Female ☐ Unknown
Phone number			
Specimen information			
Type	☐ Nasopharyngeal and oropharyngeal swab ☐ Bronchoalveolar lavage ☐ Endotracheal aspirate ☐ Nasopharyngeal aspirate ☐ Nasal wash ☐ Sputum ☐ Lung tissue ☐ Serum ☐ Whole blood ☐ Urine ☐ Stool ☐ Other:		
All specimens collected should be regarded as potentially infectious and you must contact the reference laboratory before sending samples. All samples must be sent in accordance with category B transport requirements.			
Please tick the box if your clinical sample is post mortem ☐			
Date of collection		Time of collection	
Priority status			
Clinical details			
Date of symptom onset:			
Has the patient had a recent history of travelling to an affected area?	☐ Yes ☐ No	Country Return date	
Has the patient had contact with a confirmed case?		☐ Yes ☐ No ☐ Unknown ☐ Other exposure:	
Additional Comments			

Coupling

- Degree of dependence among components
  - High coupling makes modifying different parts of the system difficult
  - Modifying a highly coupled component affects all the other connected components
- High coupling often results in brittle and unusable systems

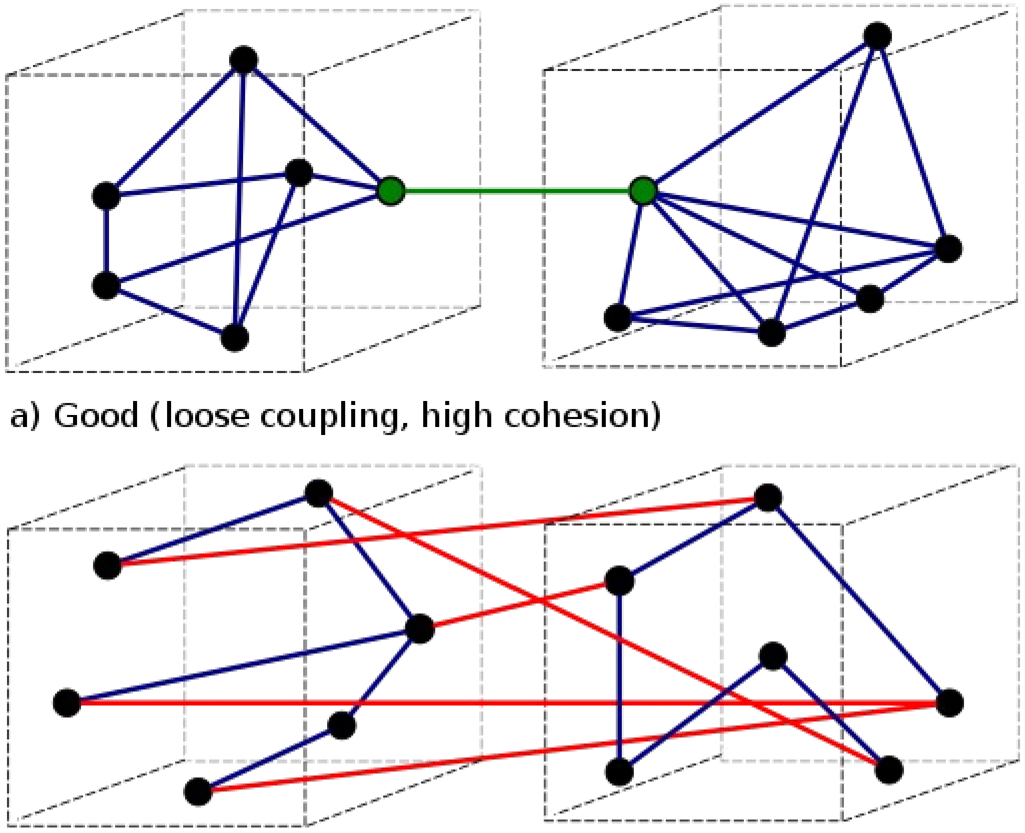

No dependencies
Loosely coupled-some dependencies
Highly coupled-many dependencies

Cohesion

- The degree to which:
  - All elements of a component are directed towards a single task; and,
  - When all elements directed towards a specific task are contained in a single component
  - High cohesion is good
  - Highly cohesive components tend to have low coupling
- Often expressed as the single responsibility principle
  - Highly cohesive components specialize in implementing a single responsibility
  - There is only one component that implements that responsibility
  - Achieving cohesiveness depends on how we modularize an architecture
- Note how the hospital example displays low coupling and high cohesion

Cohesion and Coupling

- A good measure of cohesion is
  - How often an object has to connect to objects in other components directly
  - Most of the interactions are within the same component
- When we have good cohesion
  - It results in looser coupling
- We can replace components that are loosely coupled and highly cohesive easily
  - Critical for modular architecture

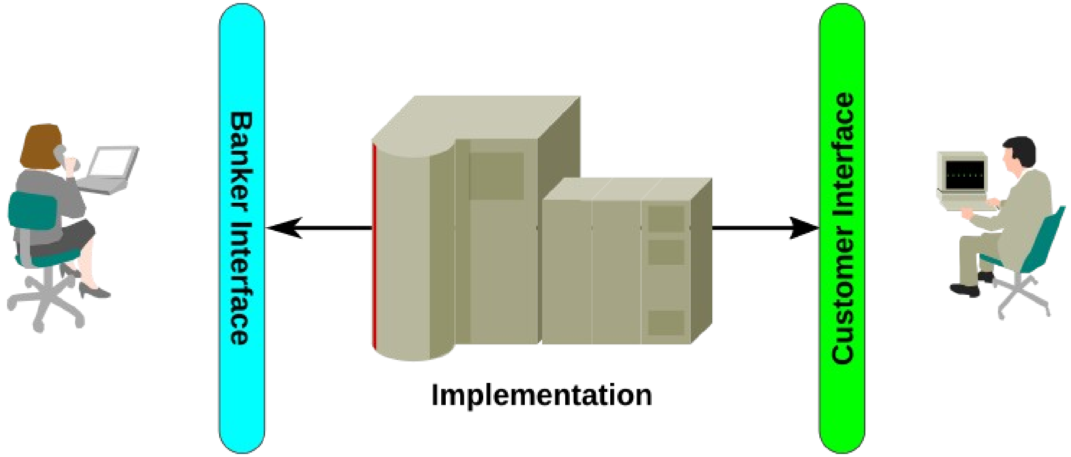


a) Good (loose coupling, high cohesion)

b) Bad (high coupling, low cohesion)

Interface and Implementation

- Consider a bank account system
- The implementation stores data in a relational data base
  - Account queries are executed using SQL
- There is also a customer interface
  - Interacts with the system in terms of the business objects and actions like deposit, withdrawal, etc
  - If we change the implementation the interface remains stable



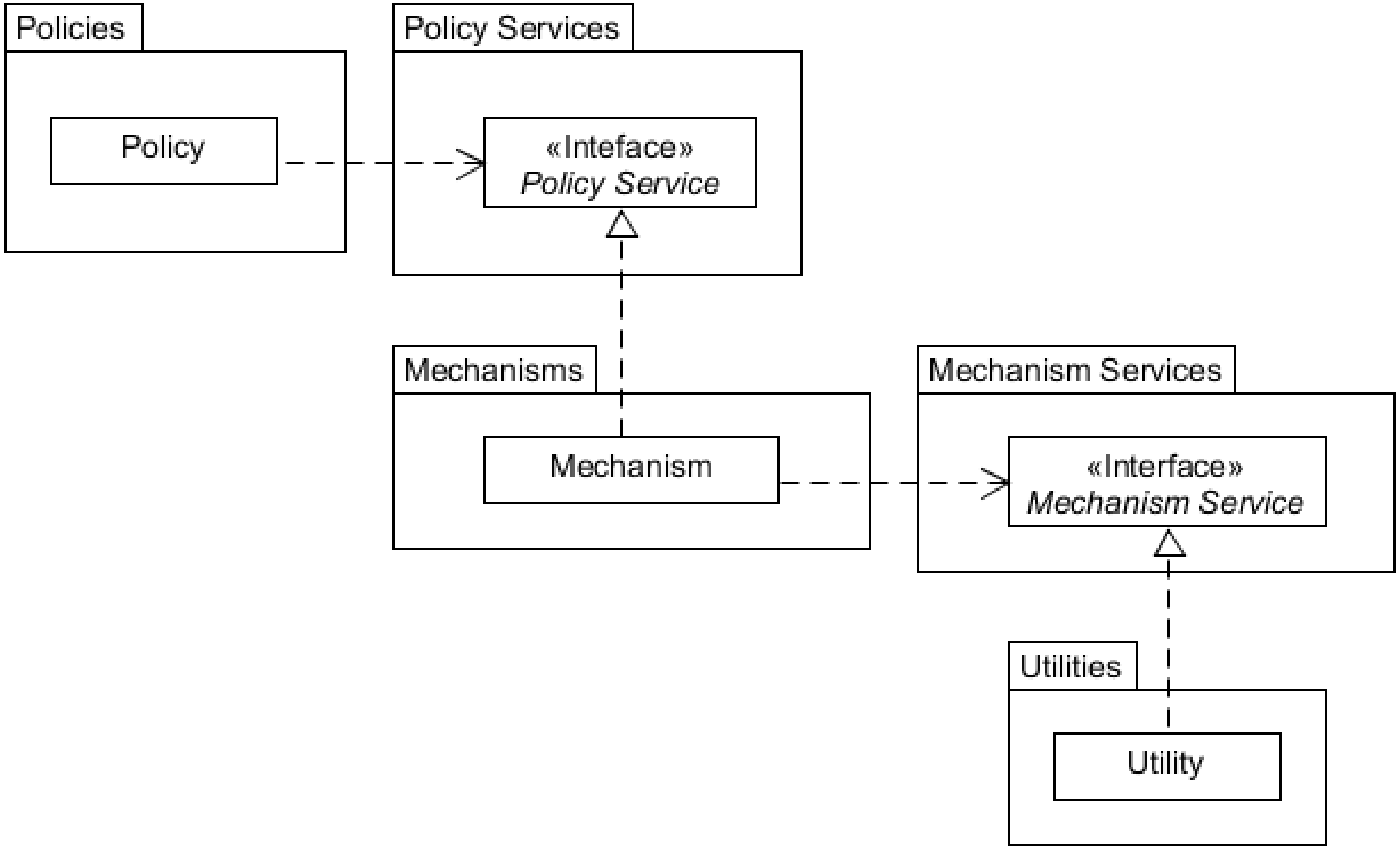
Suppleness

- Modular systems interact through interfaces
- Suppleness refers to the design of the interfaces
  - Follows good interface design rules, e.g., Open-Close principle
- Suppleness allows for future expansion of the architecture
  - Without breaking existing inter-module communications
- For example, anti-corruption layer
  - Links two modules with different internal representations with a “translation” from one representation to the other of some shared data item
  - Use Case: Data seamlessly shared between databases that use different units (pounds versus kilograms for example)

Dependency Inversion Principle

- Abstraction should be used in place of concrete implementations
  - High-level modules should not depend on low-level modules, but both types of modules should depend on abstraction (interfaces)
  - A concrete implementation should depend on a defined abstraction rather than an abstraction that is derived from a concrete implementation
    - In plain English; interface first, then code; not code first then slap on an interface
- The purpose of the DIP is to manage the dependency between high and low-level modules abstractly
  - High and low-level modules are designed independently
  - The bindings between modules are done through abstractions implemented as interfaces (suppleness)
- Interfaces are defined before the code is written

Dependency Inversion Principle

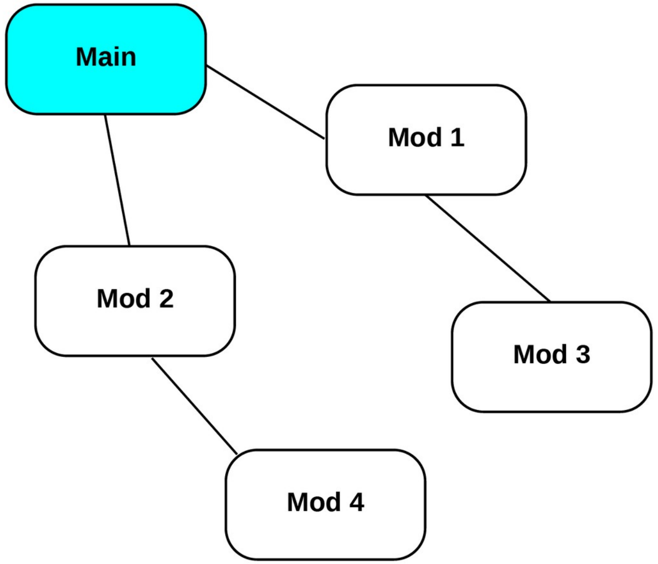


Dependency Injection

- A technique used to implement the dependency inversion principle
- Used when a client module depends on a service module
  - E.g., an invoicing module depends on a catalog module for pricing
- The service module provides an interface
  - Describes how to access the services of the service module
  - This interface is independent of the actual implementation of the service
  - The interface is “injected” or provided to the client module dynamically
  - At build time, the appropriate implementation of the service interface is used
- Requires that
  - The injected interface remain stable even if the implementation changes
  - The injected interface can be easily swapped out in the client code for a different interface

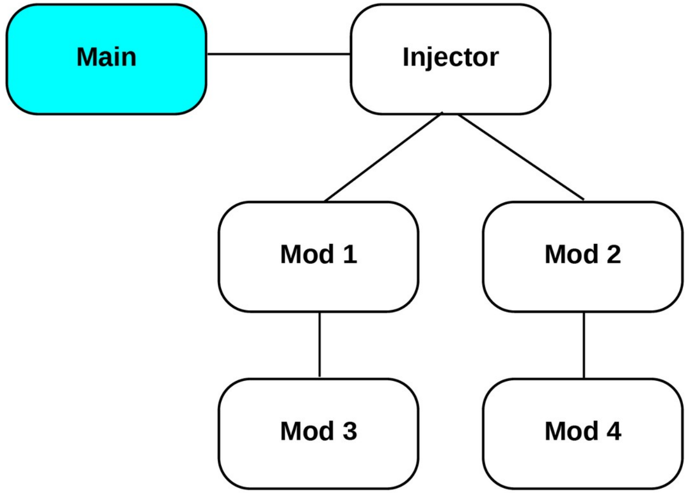
Dependency Injection

Traditional



The main function is hard-coded to call specific modules

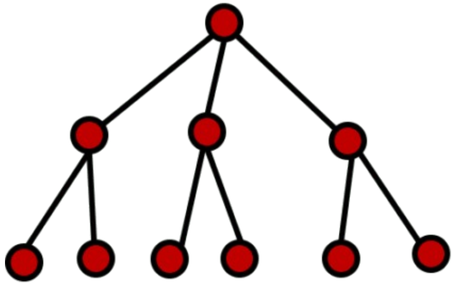
Dependency Injection



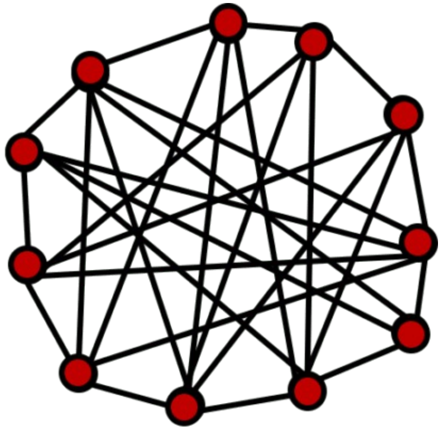
Modules are made accessed through an injector
Changing the injector changes the modules accessed
The hardwiring is replaced by a loose coupling

Inversion of Control

- Traditional top-down flow of control
  - Starts at a code entry point (e.g., the main() method)
  - Flow of control passes from calling modules to called modules
  - Then control returns to the calling module
- Ideal for execution of algorithms
  - Not so much for event driven applications



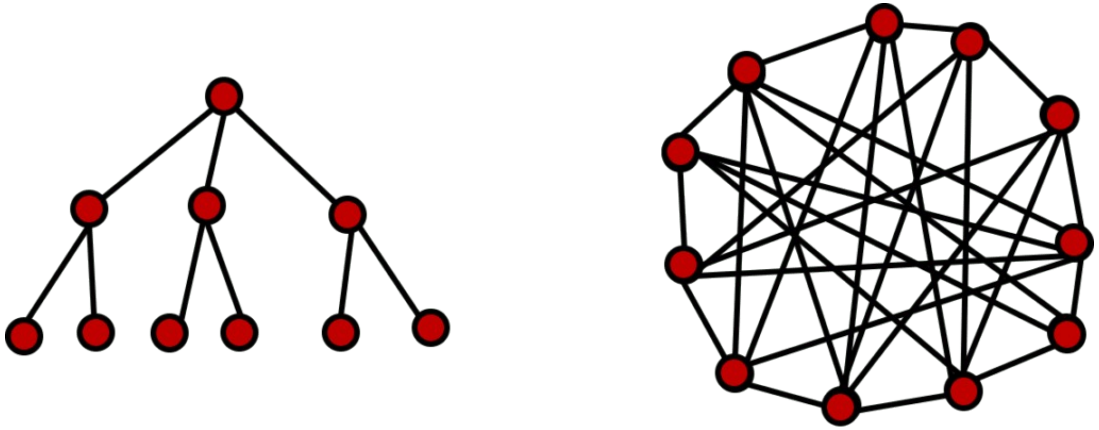
"Top-down"



"Bottom-up"

Inversion of Control

- In IoC, control starts with an "event" or client request
  - Different flows of control will occur depending on the event
  - Assumes a framework of some type (module graph)
- When a module needs a service
  - A service is located
  - The service is bound to the module
  - Often bound together using DiP and DI

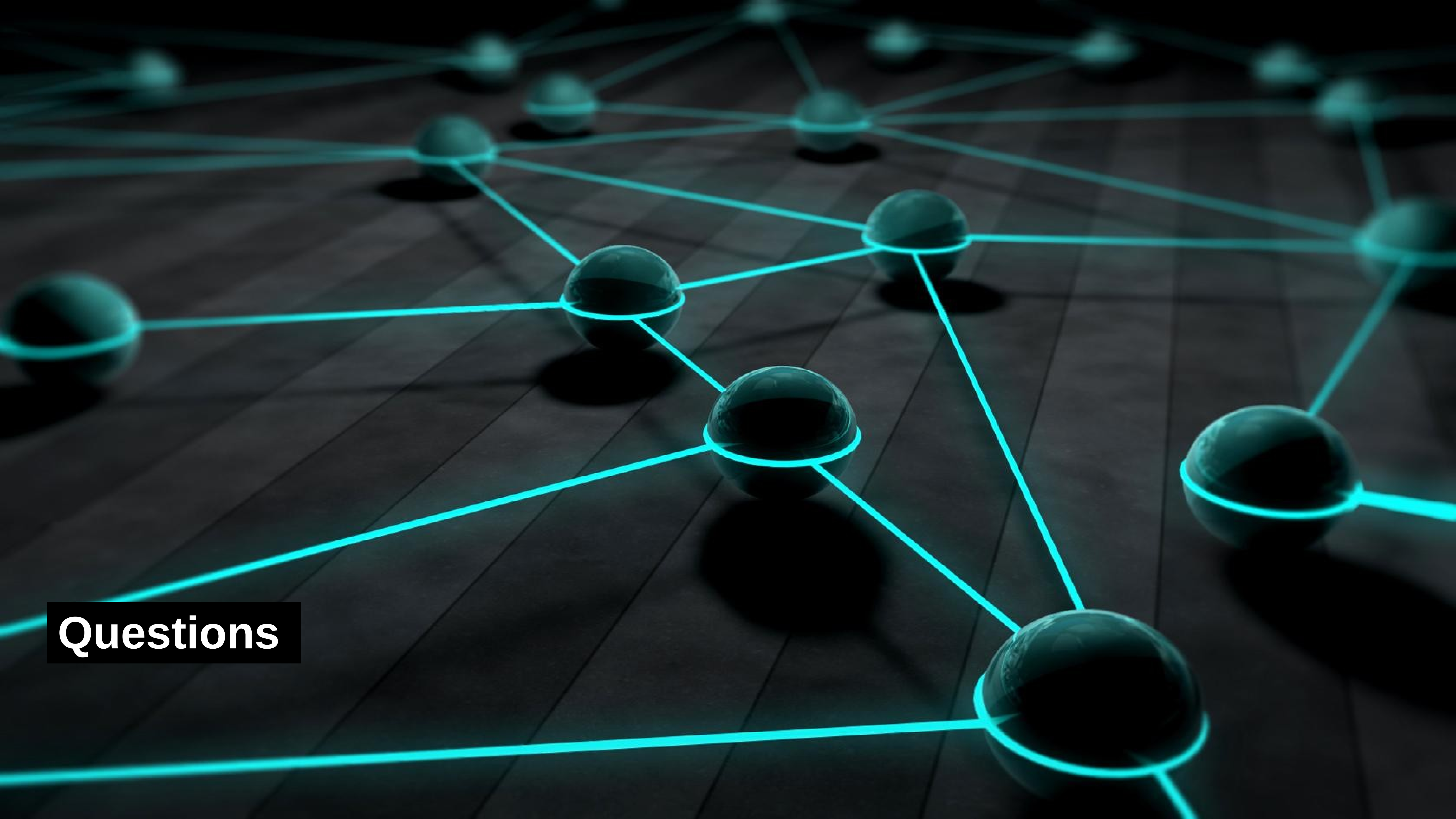


Inversion of Control

- Consider a police unit responding to an emergency call
  - This is a front line or user facing component
- Based on what they find, they may request other services like EMT
  - Additional components are “activated” as needed by the initial component
  - This is Inversion of Control - the initial component decides what other components need to be activated
- In a top down scenario, all units would be dispatched from a central command center
  - Not all units might be needed and some units that might be needed might not be there
- Military command and control used to be top down
  - Now, with modern communications capabilities, they tend to use IoC
  - IoC makes them more responsive and effective

Applying the Principles

- These design principles are applicable at all levels of software design
- In the sessions that follow, you will see them applied at:
  - The application architecture level
  - The design of program components
  - The design of code level constructs like subroutines, algorithms and functions
- These are general best practices
  - They apply across many different types of engineering and construction



Questions