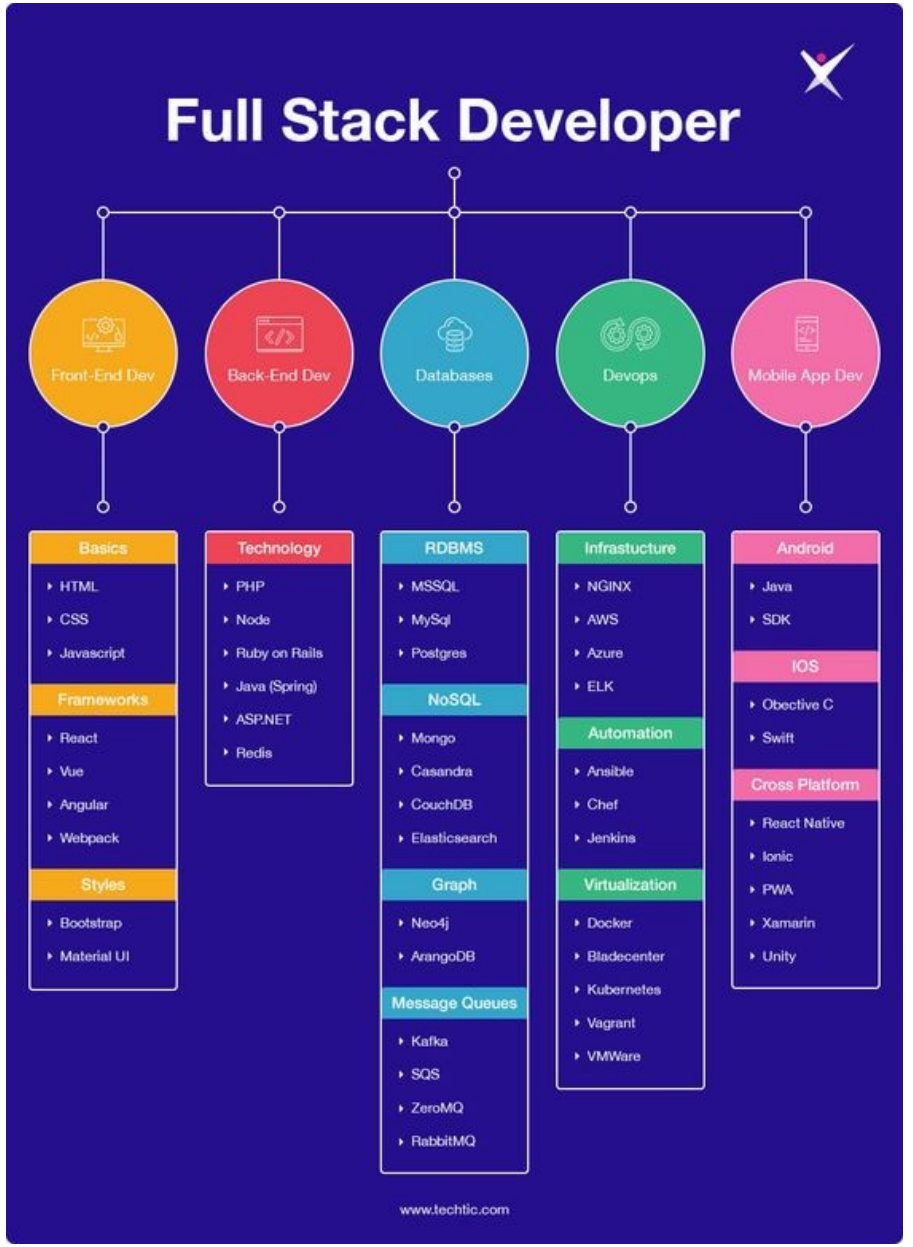

Full Stack Development
Containers, Microservices and UI
1. Modern Full Stack Development

Full Stack

- A stack is a set of technologies used across the full spectrum of development and operations
- Covers both tools used to build the app and tools used to manage the development process.
- Many different kinds of stacks exist
  - LAMP (Linux, Apache, MySQL, PHP)
- The stack used in this course is Java microservice containerization and front end JavaScript frameworks
  - Along with the DevOps operational stack

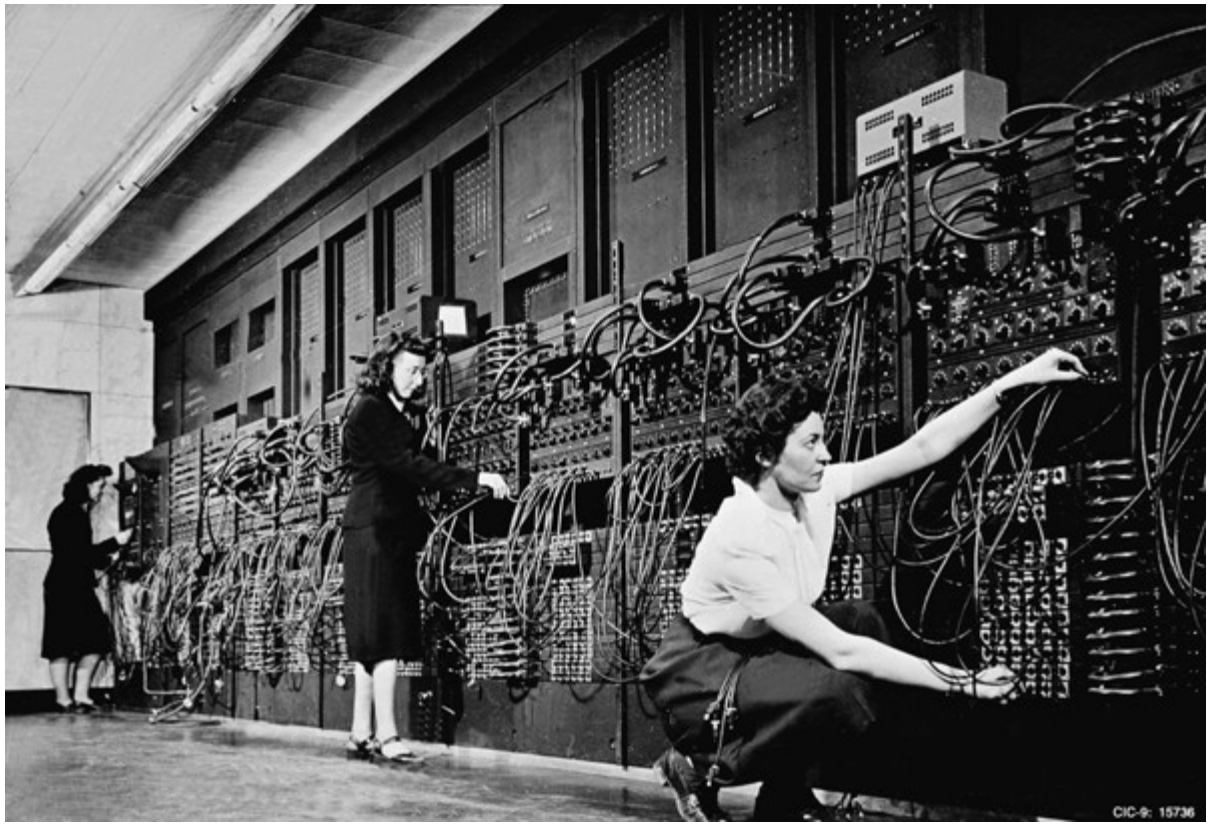


The Evolution of Development

- Software development has gone through a number of eras
- Defined by:
  - The business drivers – specifically meeting the requirements of users
  - The technology available – both enables and limits what can be done
  - The tools and techniques in use – the “way we do things”
- Main Eras
  - 1940s to 1950s Hardware Rules
  - 1950s to 1970s Code Cowboys
  - 1970s to 1990s Mainframe and Procedural Software Engineering
  - 1990s to 2010s Human Cyber Systems and Networks
  - 2010s to Present Cyber Physical Systems

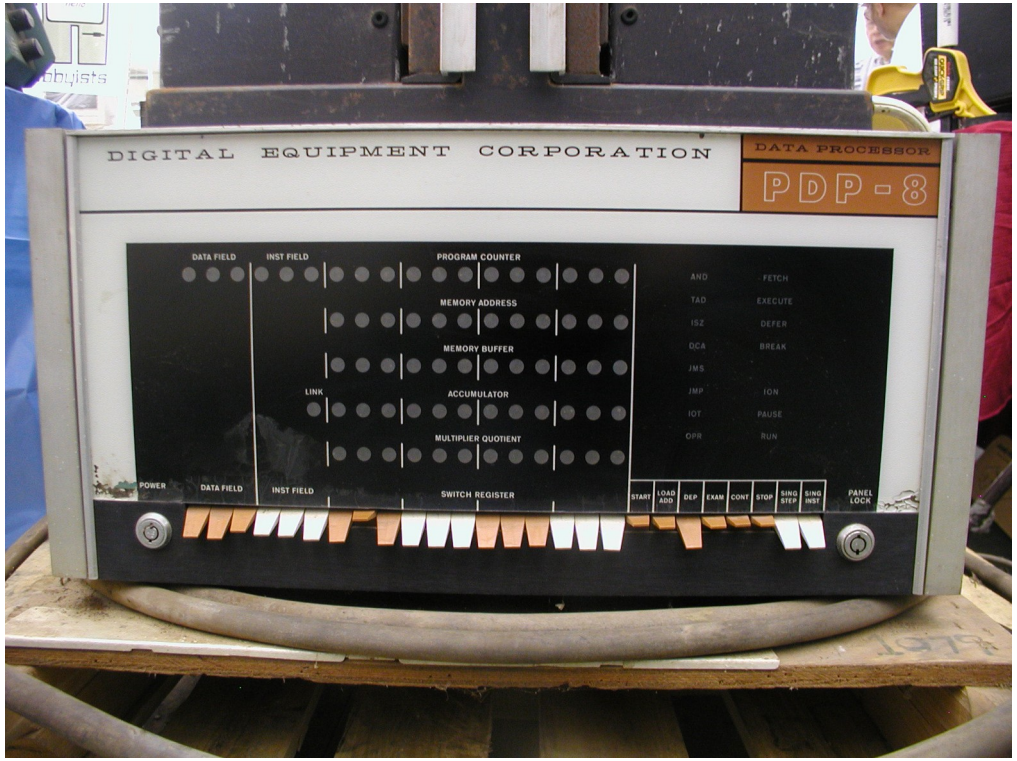
Hardware Rules 1940s - 1950s

- There was no software
- Programming was hardware based



Code Cowboys 1950s - 1960s

- Primitive operating systems
- Hardware was very expensive
- Program quality measured by how efficiently it used physical resource
- Programming done as close to machine code as possible
- Primary purpose was to automate repetitive and error prone calculations and tasks



Mainframe 1960s - 1990s

- OS created a layer of abstraction between hardware and applications
- Demand for automation of business workflows and procedures
- Required adapting engineering construction principles to software development
- Programming languages were procedural (ALGOL, FORTRAN, C)
- Mainframes represented “islands of computing”
- Hardware still very expensive.

Users of the370 Model 165 will have a choice of five main core storage sizes, ranging from 512,000 to over 3-million bytes. Seven main memory sizes are available for the Model 155, ranging from 256,000 to over 2-million bytes. [IBM Marketing Material]

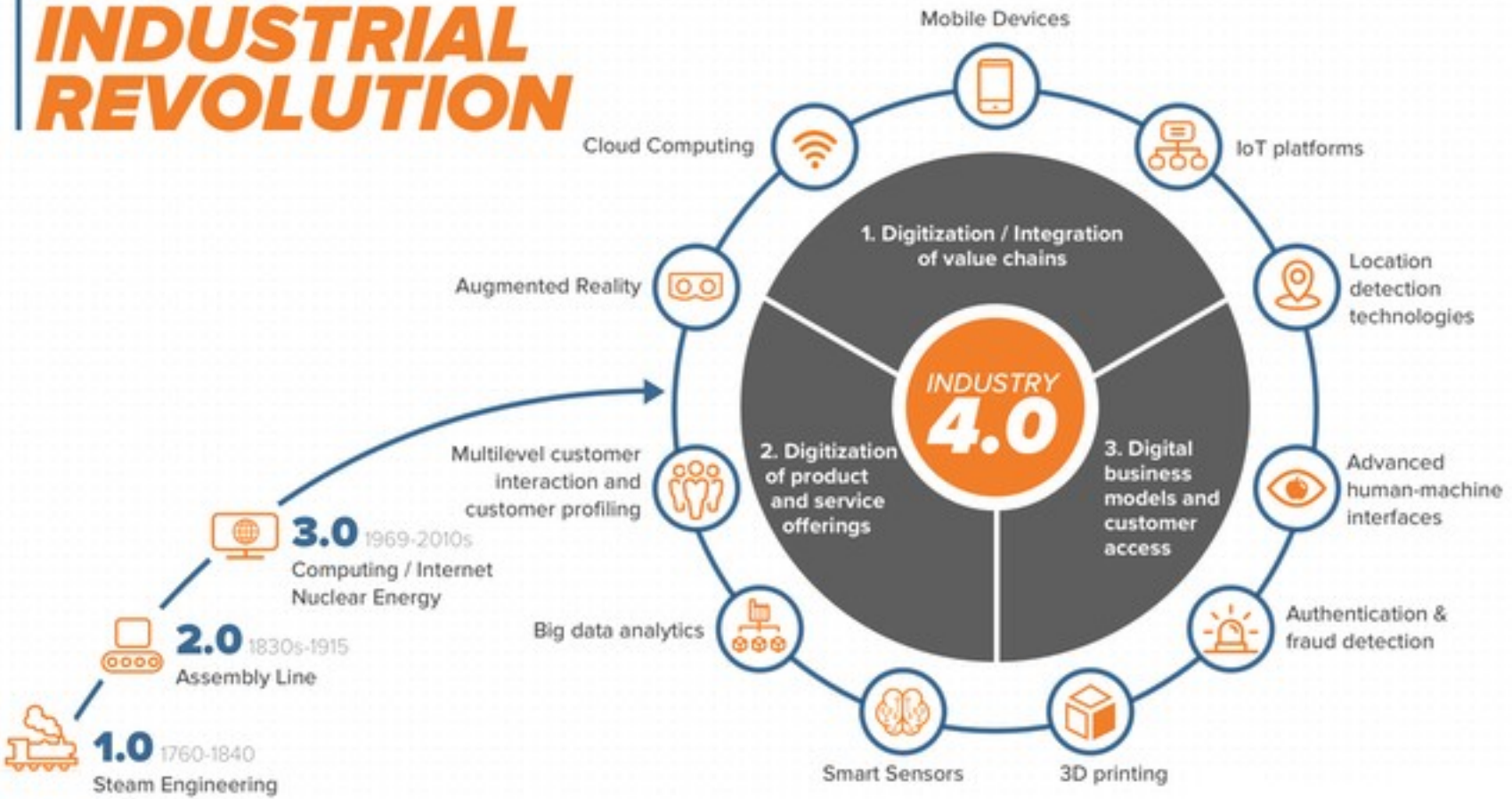
Cyber Human 1990s - 2010s

- Drivers
  - Rise of personal computing
  - Rise of the Internet
  - Businesses going “on-line”
- Programming
  - Increased used of OO languages like Java
  - Separated out front-end from back-end
  - Needed to access main-frame data on the back end
  - Emphasis was on network based infrastructure

Drivers of the Current Era

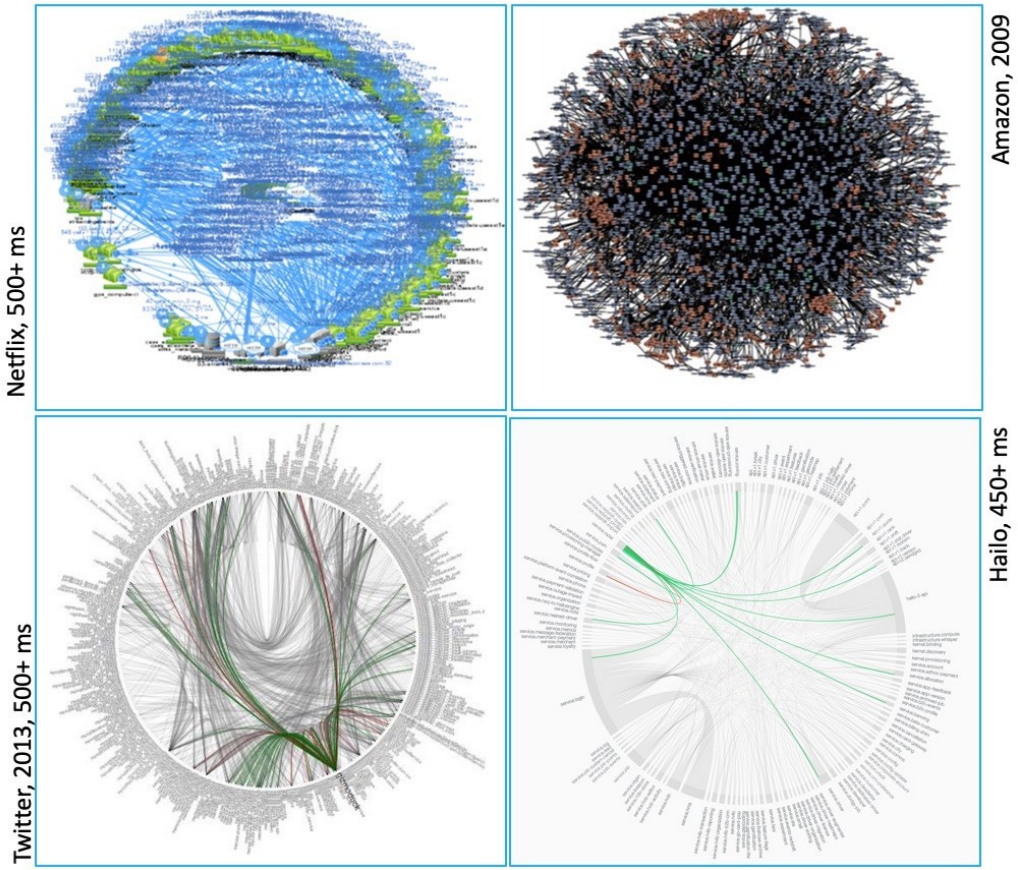
- Exponential increases in the amount of data being generated and needing to be processed “Big Data”
- Supporting new models of computing – machine learning and analytics
- Rapidly changing hardware capabilities
- Virtualization, cloud computing and infrastructure as code
- Reduced time to market for new or upgraded applications
- Deployment of applications at massive scales
- No down-time or disruption of services

FOURTH INDUSTRIAL REVOLUTION



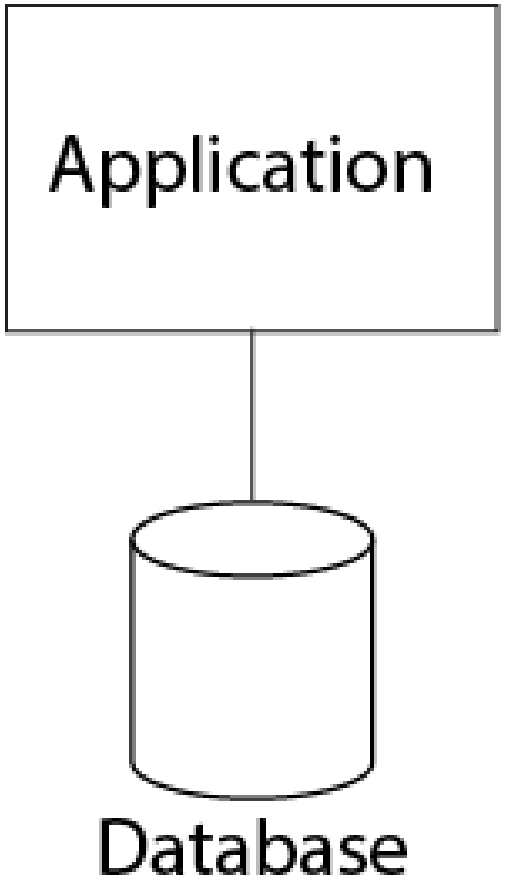
Drivers of the Current Era

- Requires new ways of thinking about app development
  - Release time needs to be days, not months
  - Robustness is essential
- Massive scales
  - Hundreds of millions of transaction per second
  - Zeta-bytes of data (a zeta-byte = billion terabytes)
  - Data is generated in various forms from user interactions and devices
- Application are clusters of smaller scalable components working together
  - Called microservices
  - Often result in a “death star” architecture



Single Process Monolith

- All code is packed into a single monolith
  - Often a good choice for small applications and organizations
- Does not scale well in terms of
  - Functionality
  - Performance
  - Complexity
- Advantages when the app is small
  - Easier to manage and deploy
  - Doesn’t require elaborate infrastructure
  - Often adequate for many apps

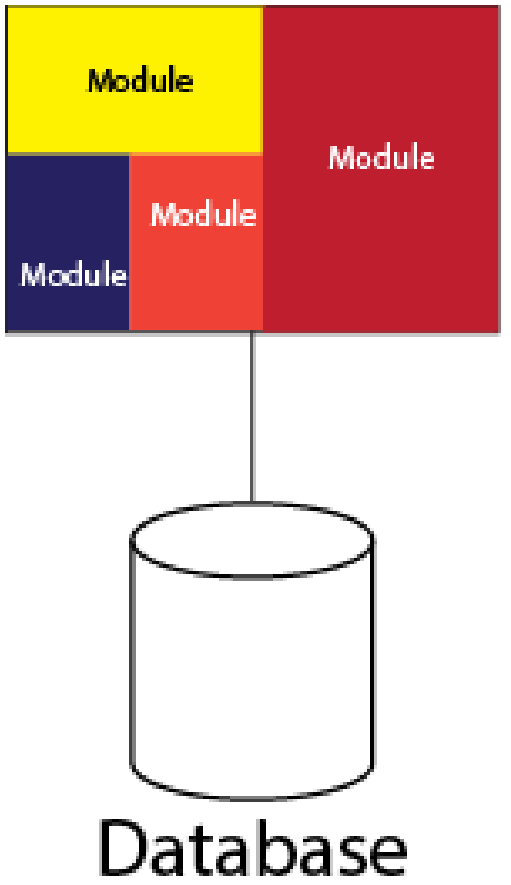


Business Analogy

- 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 systems structure

Modular Monolith

- Process is divided into multiple modules
- All modules must be deployed together
- Does not scale well in terms of
  - Functionality
  - Performance
  - Complexity
- The working definition of a monolith is that all of the code has to be deployed as a single unit
  - An application may not be a overall monolith
  - But it may have monolithic components

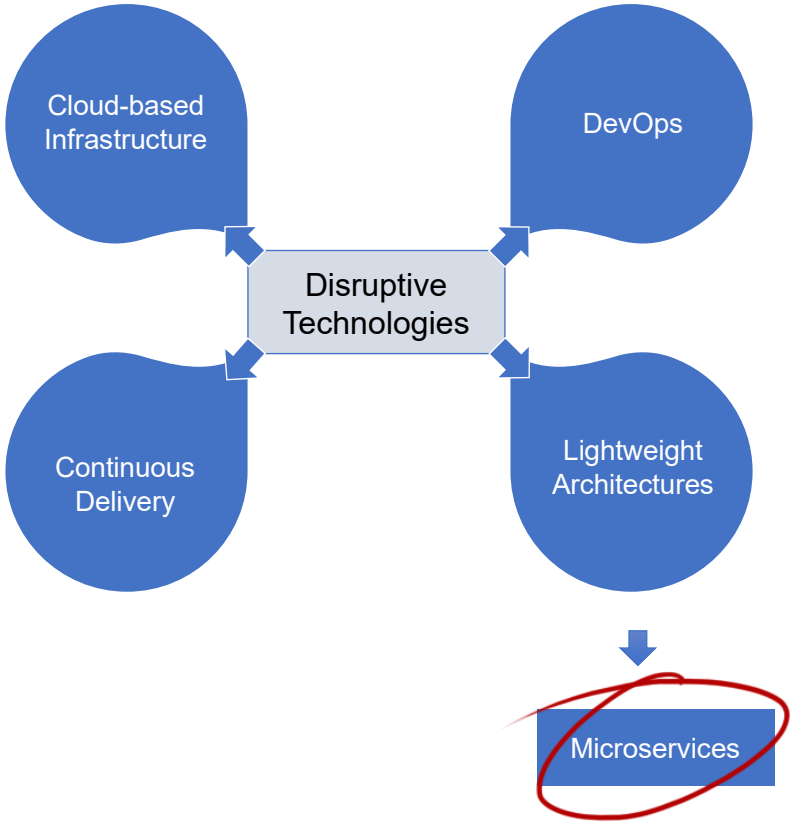


Business Analogy

- As 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 reorganize along functional lines

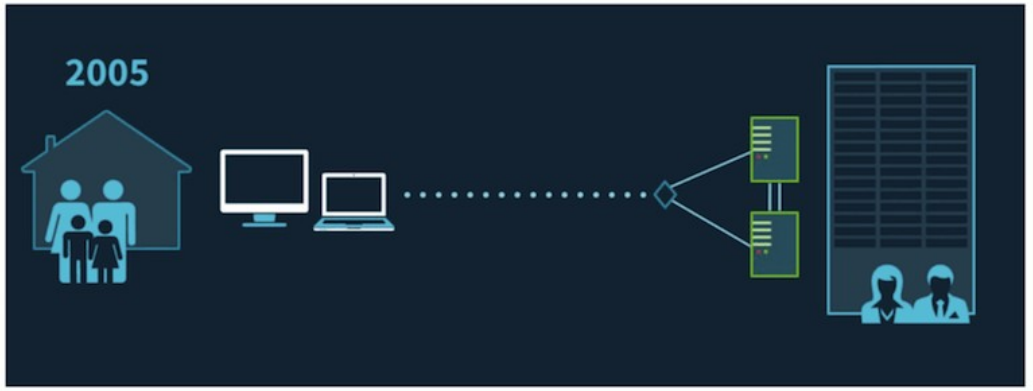
Disruptive Innovation

- The departmental version of the business can scale only so far
- The introduction of the Cyber dimension, both human and physical, enables massive scaling
  - Not limited by physical resources
  - Unlimited virtual resources
  - Creates a single world-wide enterprise
- In order to meet these new demands, new technologies have to be employed
  - Also may require a re-engineering of the business structure

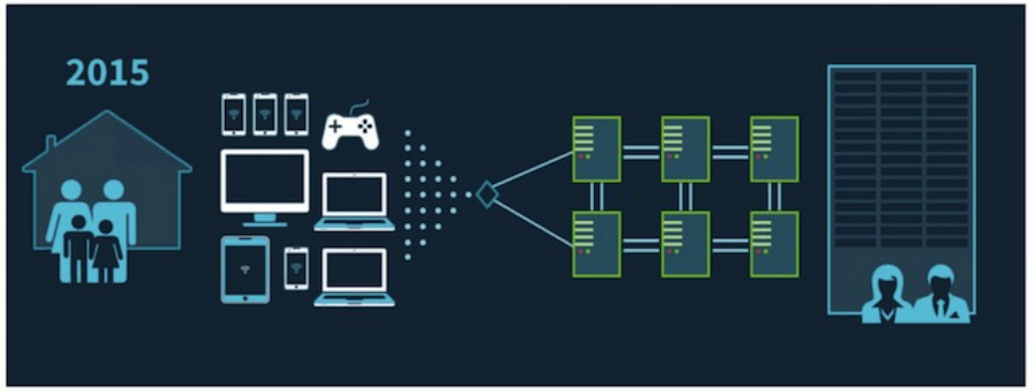


Real World Scalability - WalMart

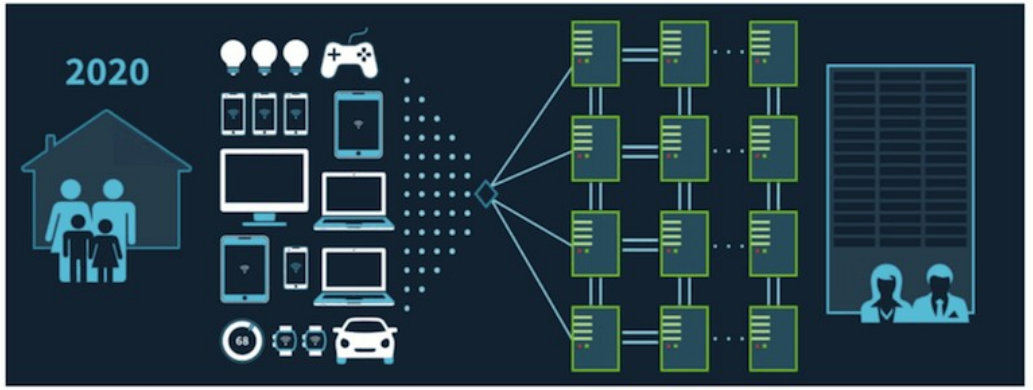
2005 ARCHITECTURE



2015 ARCHITECTURE



2020 ARCHITECTURE



THE WORLD BY 2020

» 4 billion connected people

» 25+ million apps

» 25+ billion embedded systems

» 40 zettabytes (40 trillion gigabytes)

» 5,200 GB of data for every person on Earth

Business Drivers of Microservices

Business Drivers

- Business Agility
  - Speed of change is faster with a more modular architecture
  - React quicker to feature and enhancement requirements
  - Easy integration with new technology, processes (Mobile, Cloud, Continuous DevOps)
- Composability
  - Allows for reuse of capability and functionality
  - Allows for integration with other internal and external services
  - Reduces technical debt and replication
- Scalability
  - Services can scale up to handle peak loads, or down to save costs

Business Drivers

- Robustness and Migration
  - A single microservice failures does not bring down an application
  - The business functionality is always available
  - Updated functionality can be rolled out in a controlled and safe manner
  - Smaller components reduces risk exposure
- Enable Polyglot Development
  - Different technologies an be used for different services
  - No lock-in to a single technology across the whole business

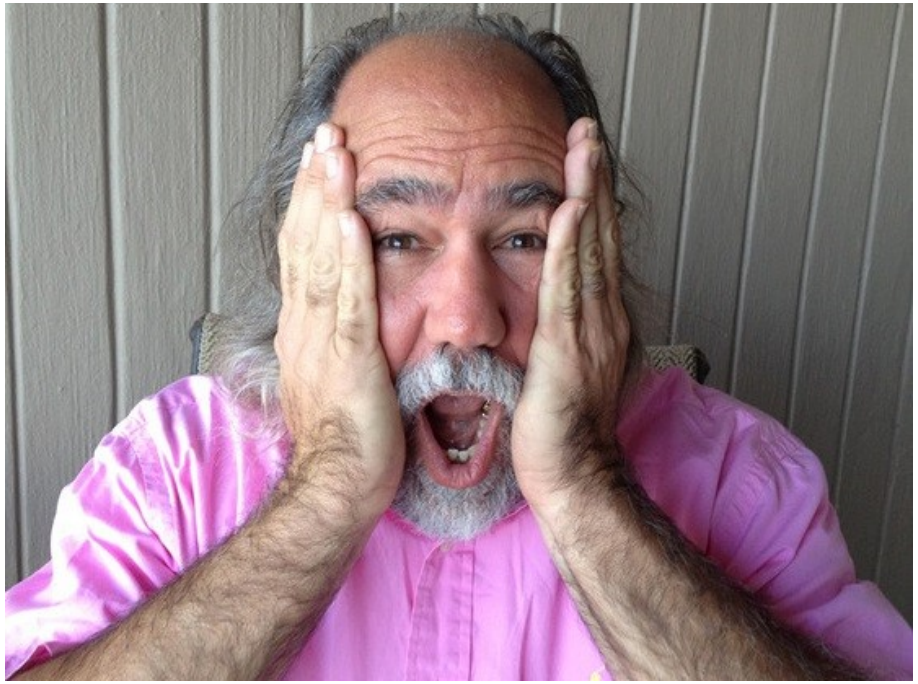
Mission Critical Industrial Strength Software

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 sys tems. By 'essential' we mean that we may master this complex ity, 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.

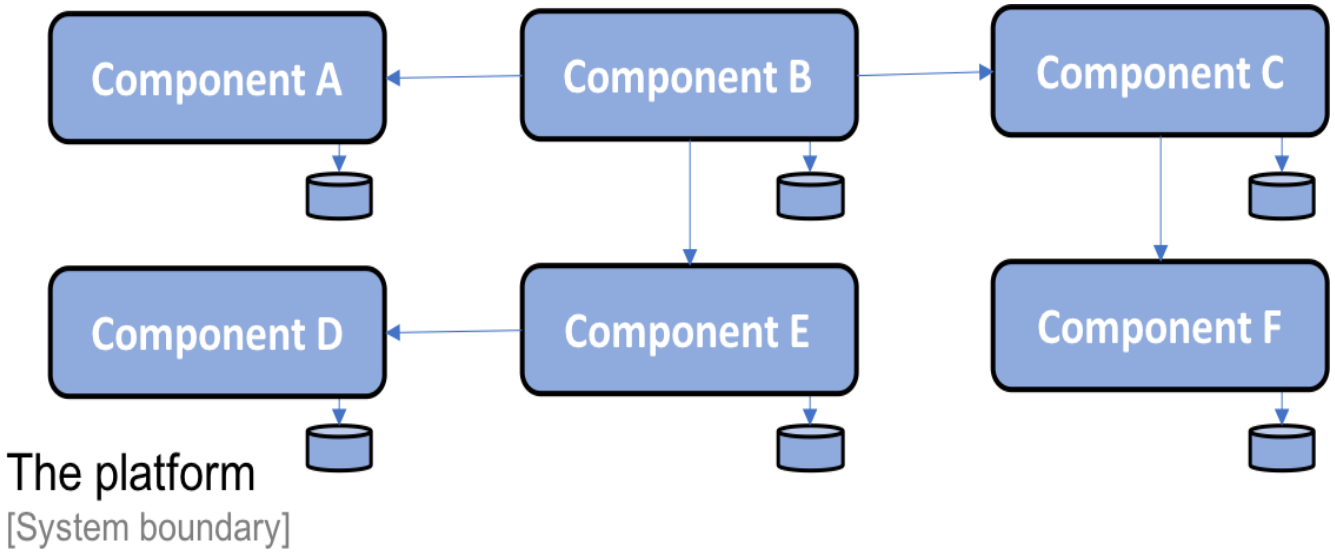
Roger Sessions

Microservices Core Concepts

- Microservices are small, independent, composable services
- Each service can be accessed by way of a well-known API format
  - Like REST, GraphQL, gRPC or in response to some event notifications
- Breaks large business processes into basic cohesive components
  - These basic process actions are implemented as microservices
  - The business process is executed by making calls to these microservices
- Each microservice provides one cohesive service
  - Microservices do not exist in isolation
  - They are part of a larger organization framework

Microservices Core Concepts

- Microservices coordinate with other microservices to accomplish the tasks normally handled by a monolithic application
- Microservices communicate with each other synchronously or asynchronously
- Microservices make applications easier to develop and maintain
- BUT A LOT HARDER TO MANAGE



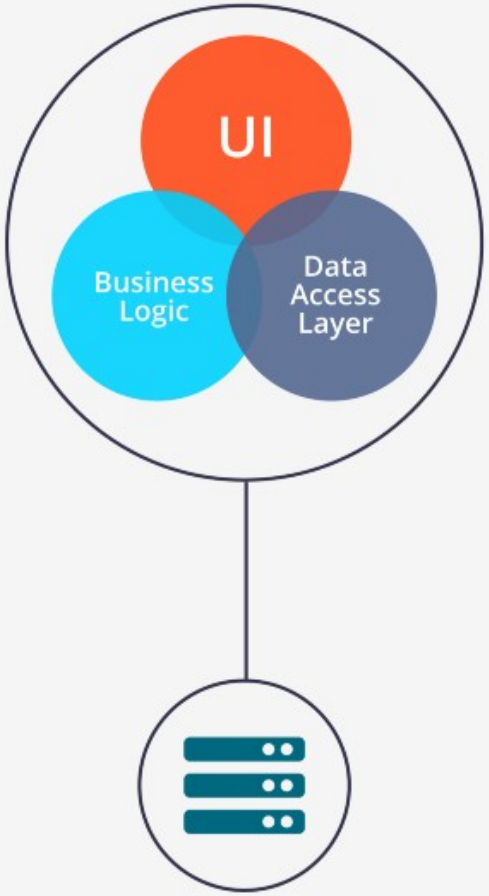
Characteristics of Microservice Deployments

- Light-weight, independent, and loosely-coupled
- Each has its own codebase
- Responsible for a single unit of business functionality
- Uses the best technology stack for its use cases
- Has its own DevOps plan for test, release, deploy, scale, integrate, and maintain independent of other services
- Deployed in a self-contained environment
- Communicates with other services by using well-defined APIs and simple protocols like REST over HTTP
- Responsible for persisting its own data and keeping external state

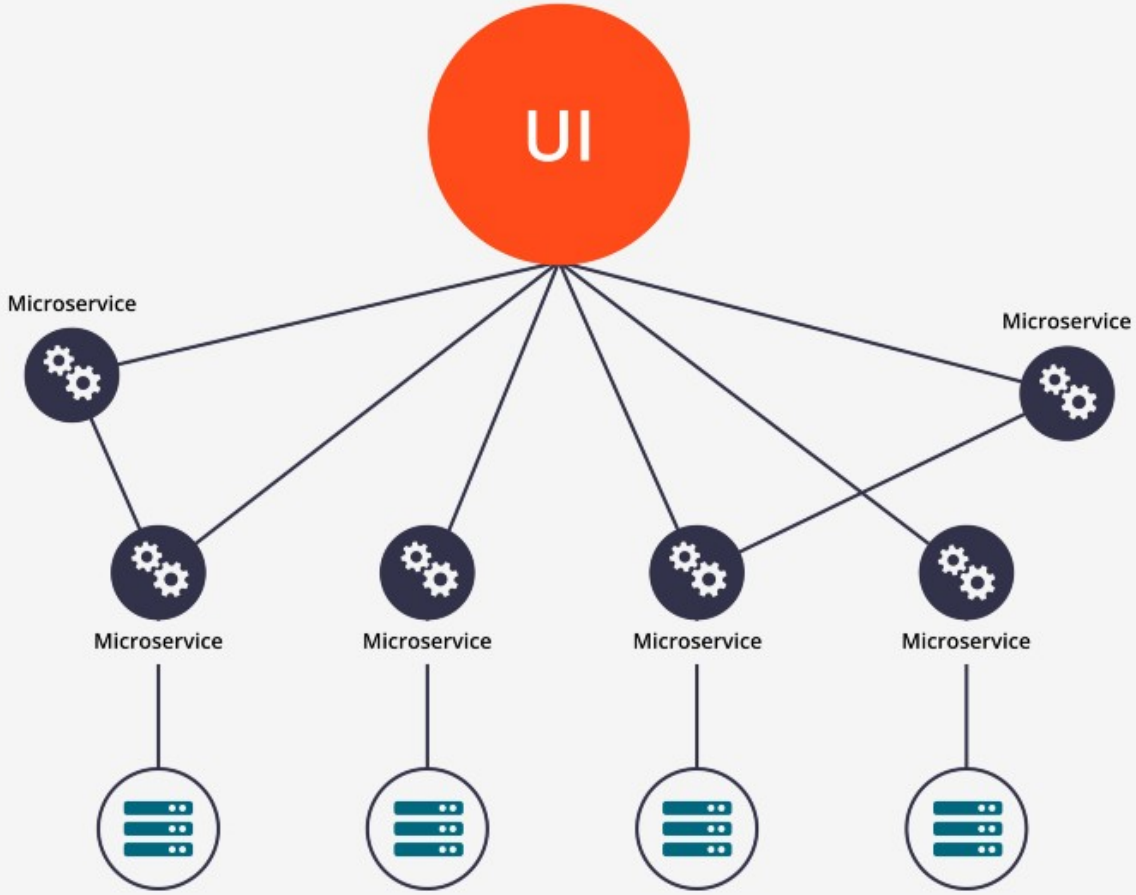
Challenges Building Microservices

- Even the simple applications become very complex
- Requires teams that are autonomous and cross functional
- Requires support for the principles of DevOps throughout the organization, e.g., Infrastructure as code
- Infrastructure management is paramount
- Continuous quality monitoring is essential
- Cannot get any benefit without development pipeline automation (CICD)

Monolith vs Microservice



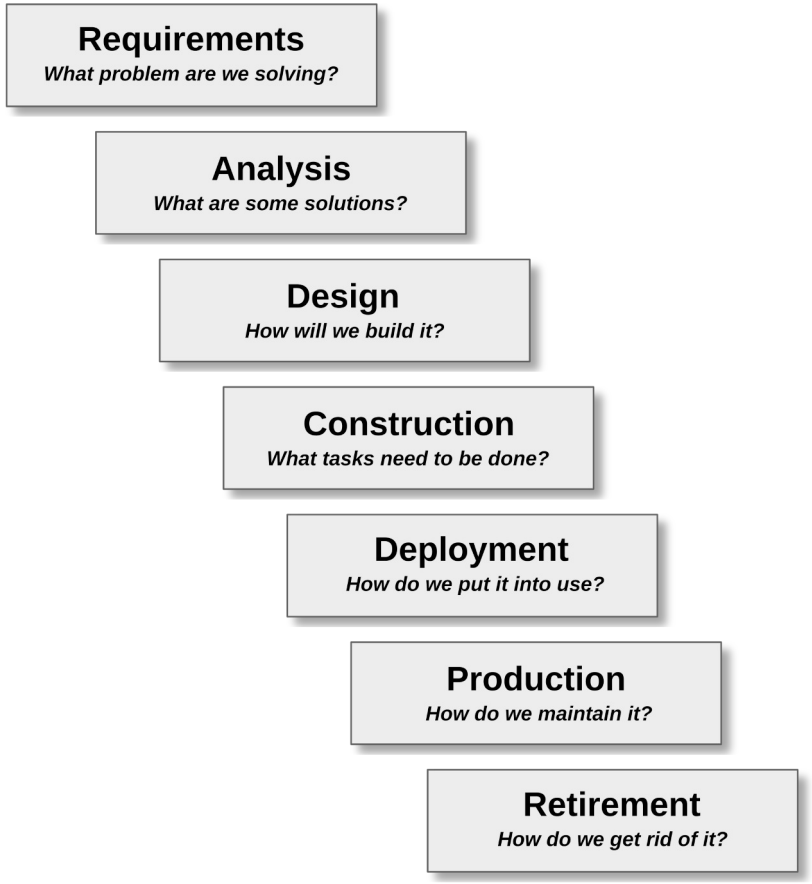
Monolithic Architecture



Microservice Architecture

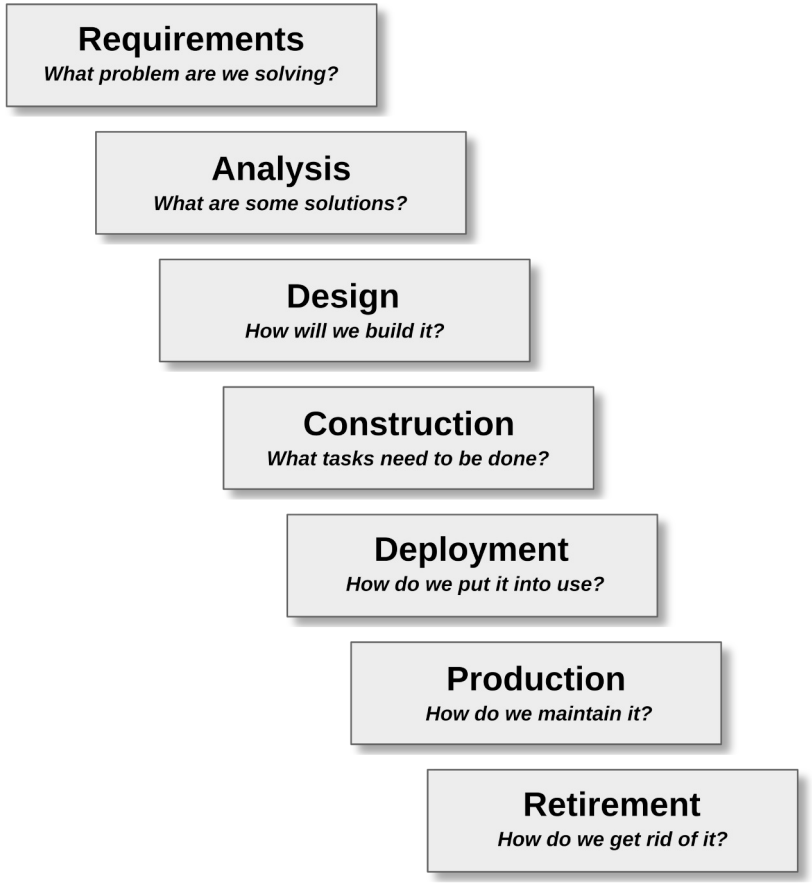
Requirements

- Identify the problem to be solved
  - What is needed by stakeholders
  - What factors constrain solutions
- Acceptance criteria
  - How will we know the problem is solved?
- Requirements types
  - Functional, non-functional (performance) and business



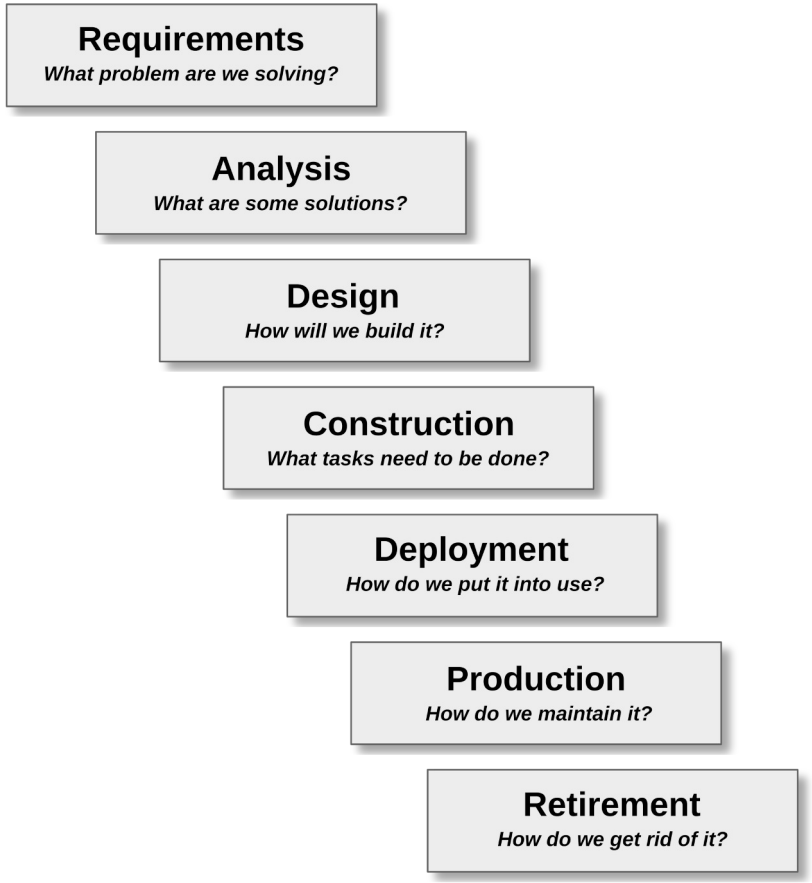
Analysis

- Evaluate possible solutions
  - What would a solution look like?
  - How would it perform?
  - How would it be organized?
  - How would it be tested?
- Evaluate possible architectural choices
  - Monolithic application
  - Microservice ←This is the one we will explore
  - Some other architecture



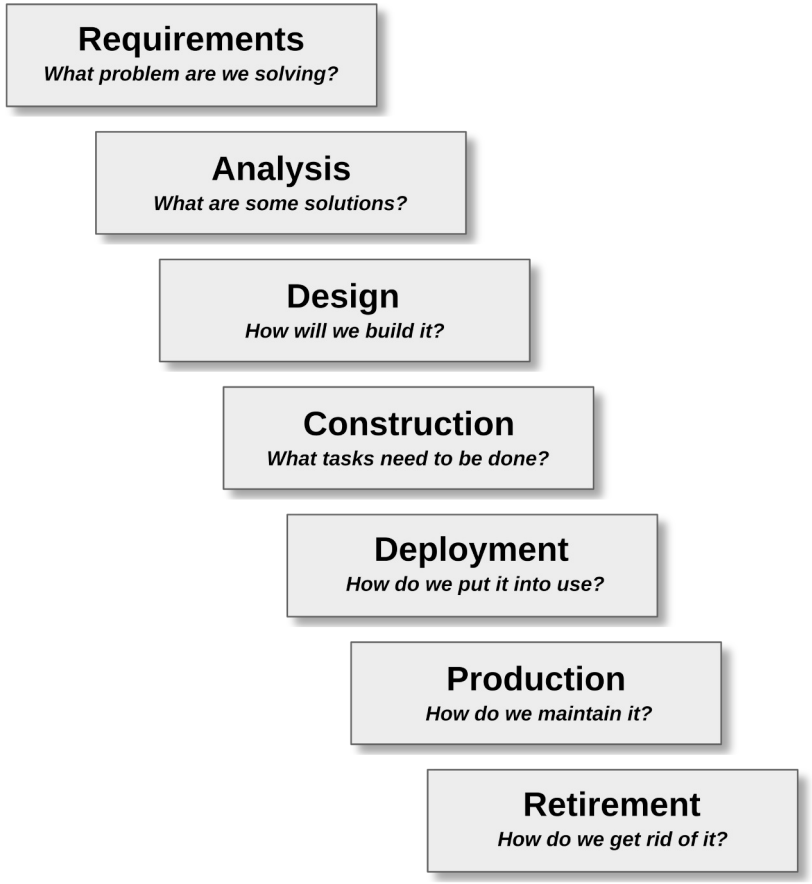
Design

- Only one solution can be built
- What is possible with the available resources?
- Define the concrete architecture
  - Design the components
  - Design how they interact
  - Choose technologies for construction of the components
- Define the plan for construction phase



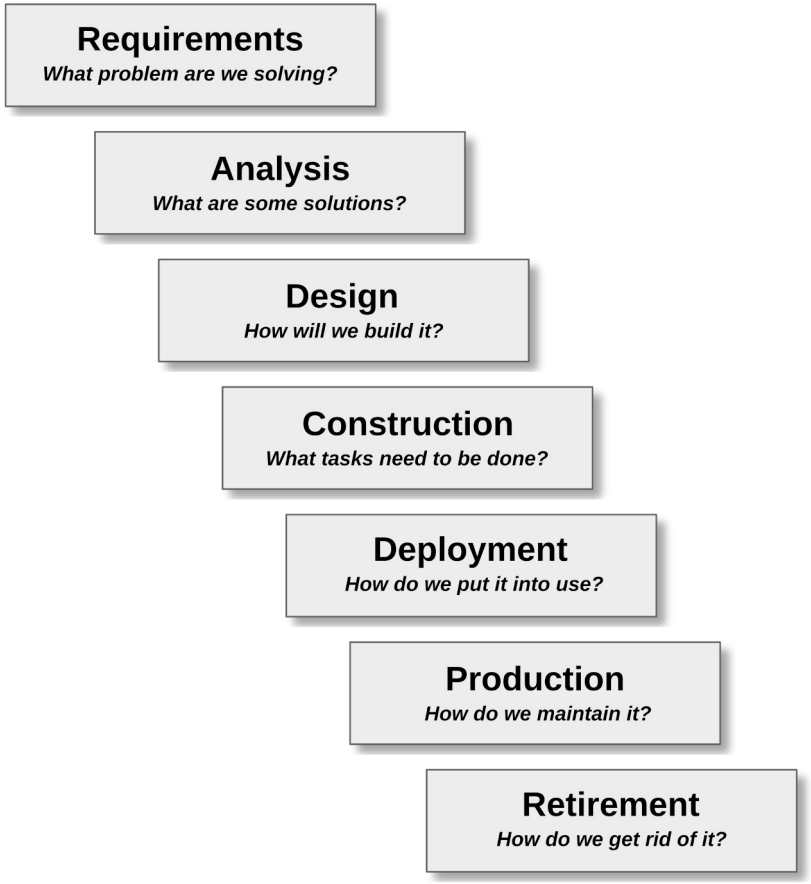
Construction

- Apply a development process
  - Roles and responsibilities
  - Testing, coding, etc.
- Define the methodology
  - Agile/Scrum, etc.
  - Code, build, test



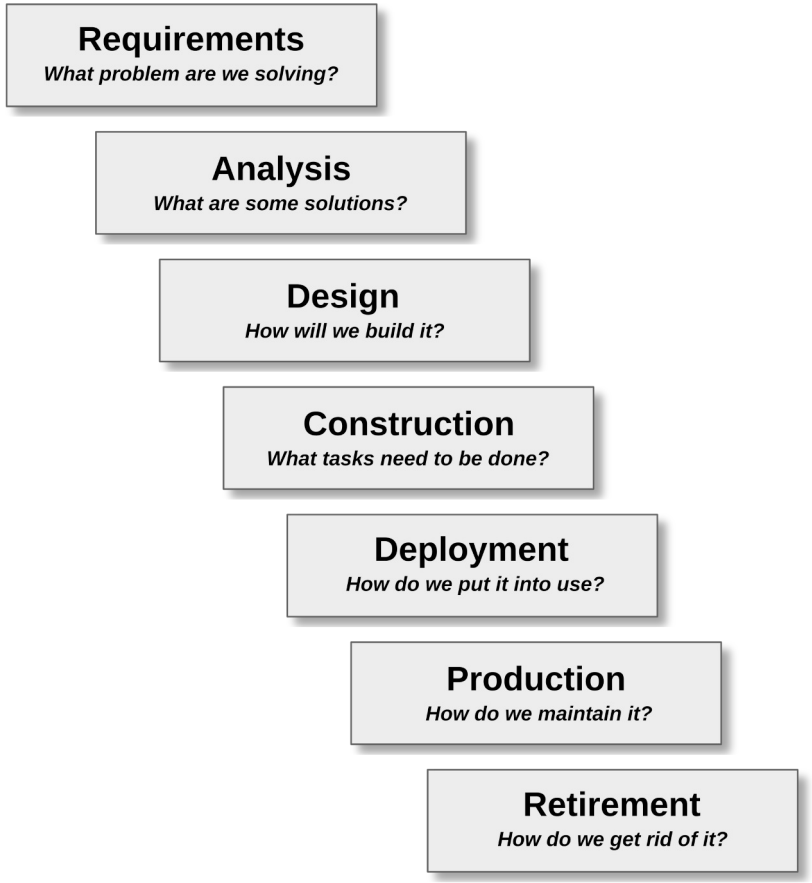
Deployment

- Get the application into production
  - Roll out logistics
  - Continuous Deployment?
  - Beta test and acceptance test
- Transition the users to the deployed app



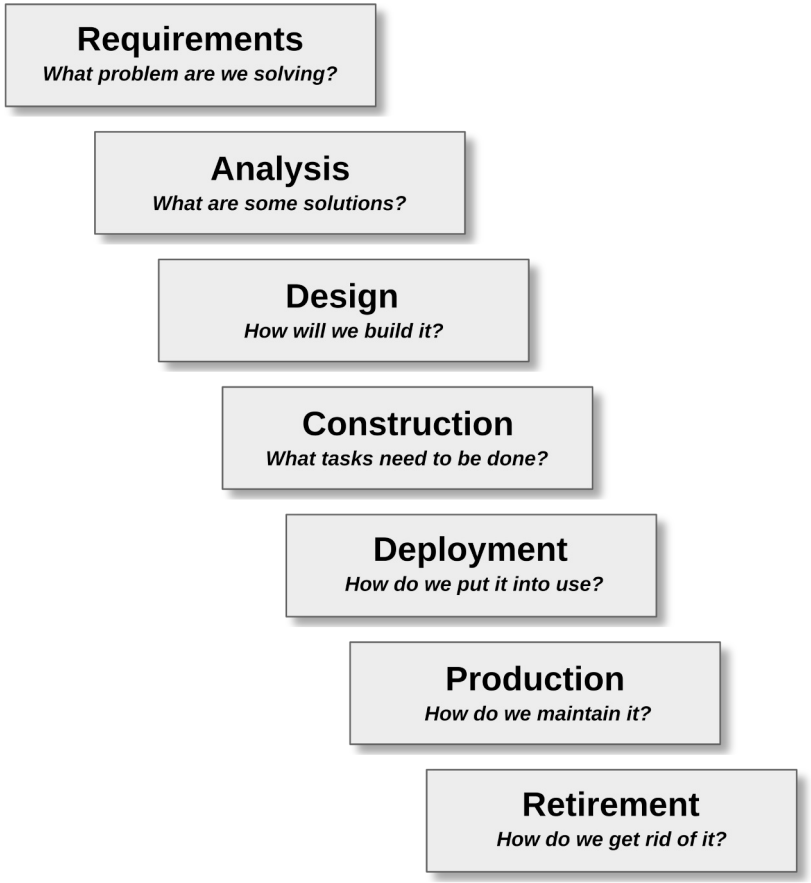
Production

- Monitor performance
- Manage changing requirements
- Collect usage data for future development
- Evaluate service level agreements
- Look for “gotchas” in the real world
- Adapt deployment to changing loads and traffic



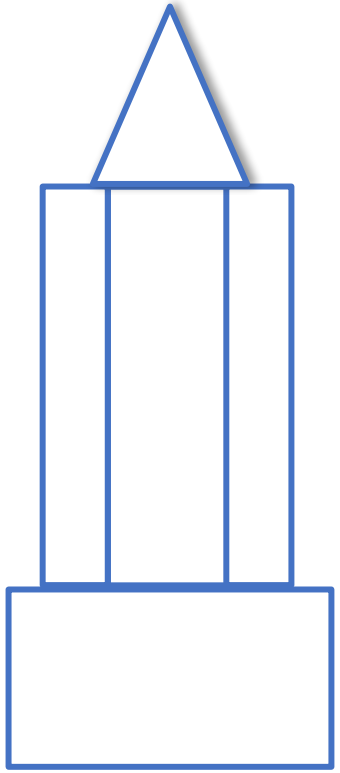
Retirement

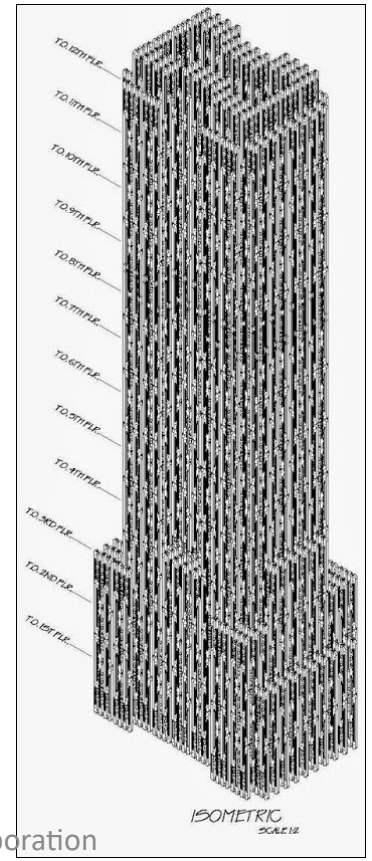
- Plan for taking the application out of service
  - Without disrupting the business
  - Without services to clients becoming unavailable
- Plan for transitioning users to the application's replacement

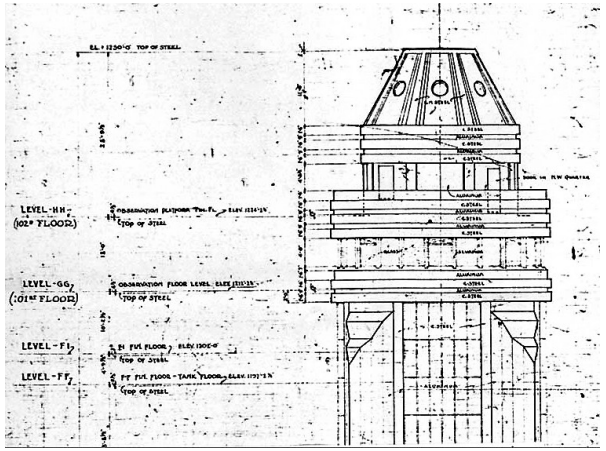


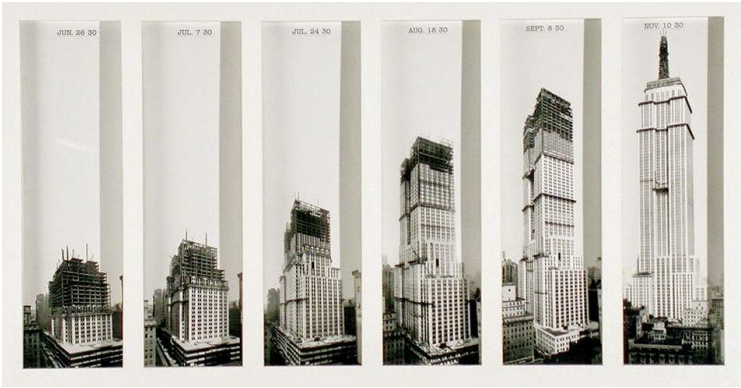
A Philosophy for Architectural Design

Conceptual → Logical → Structural → Concrete









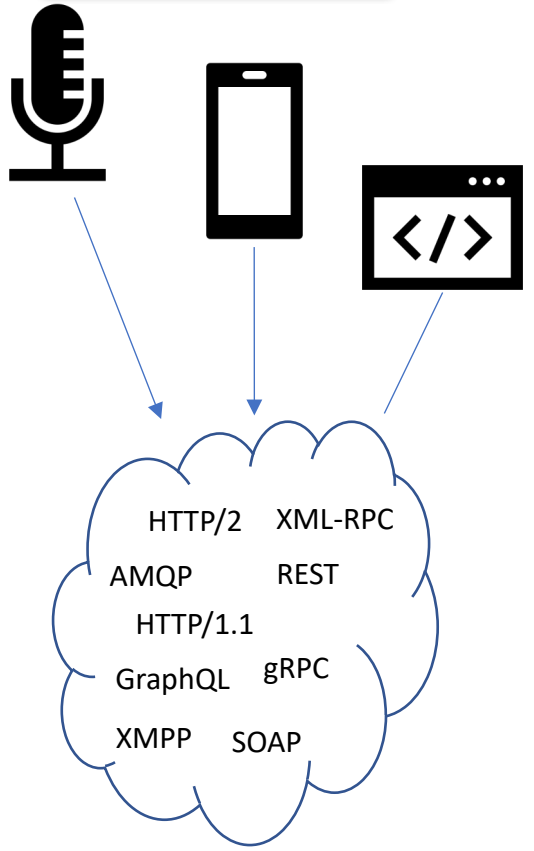
LearnQuest

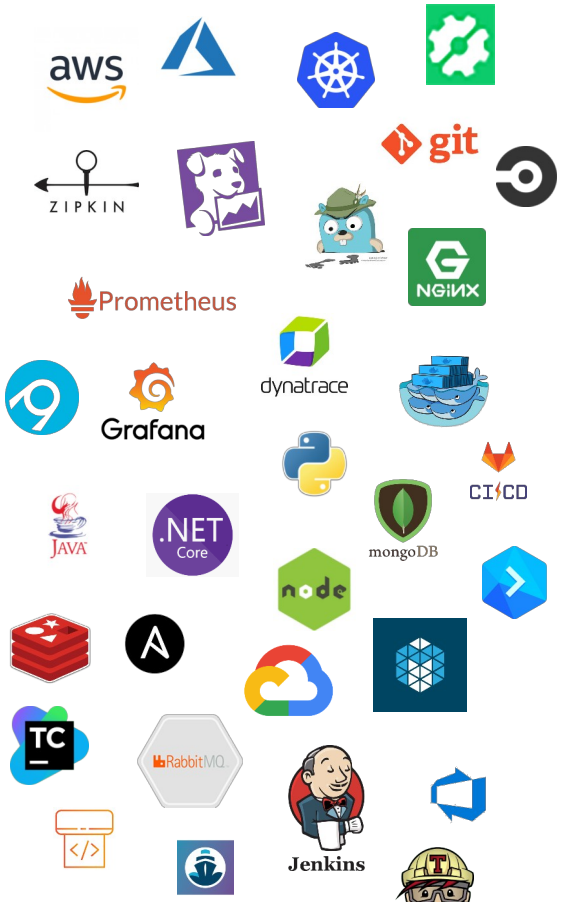
A Philosophy for Architectural Design

Conceptual → Logical → Structural → Concrete

Online shopping allows our customers to make, manage and monitor their purchases and orders online

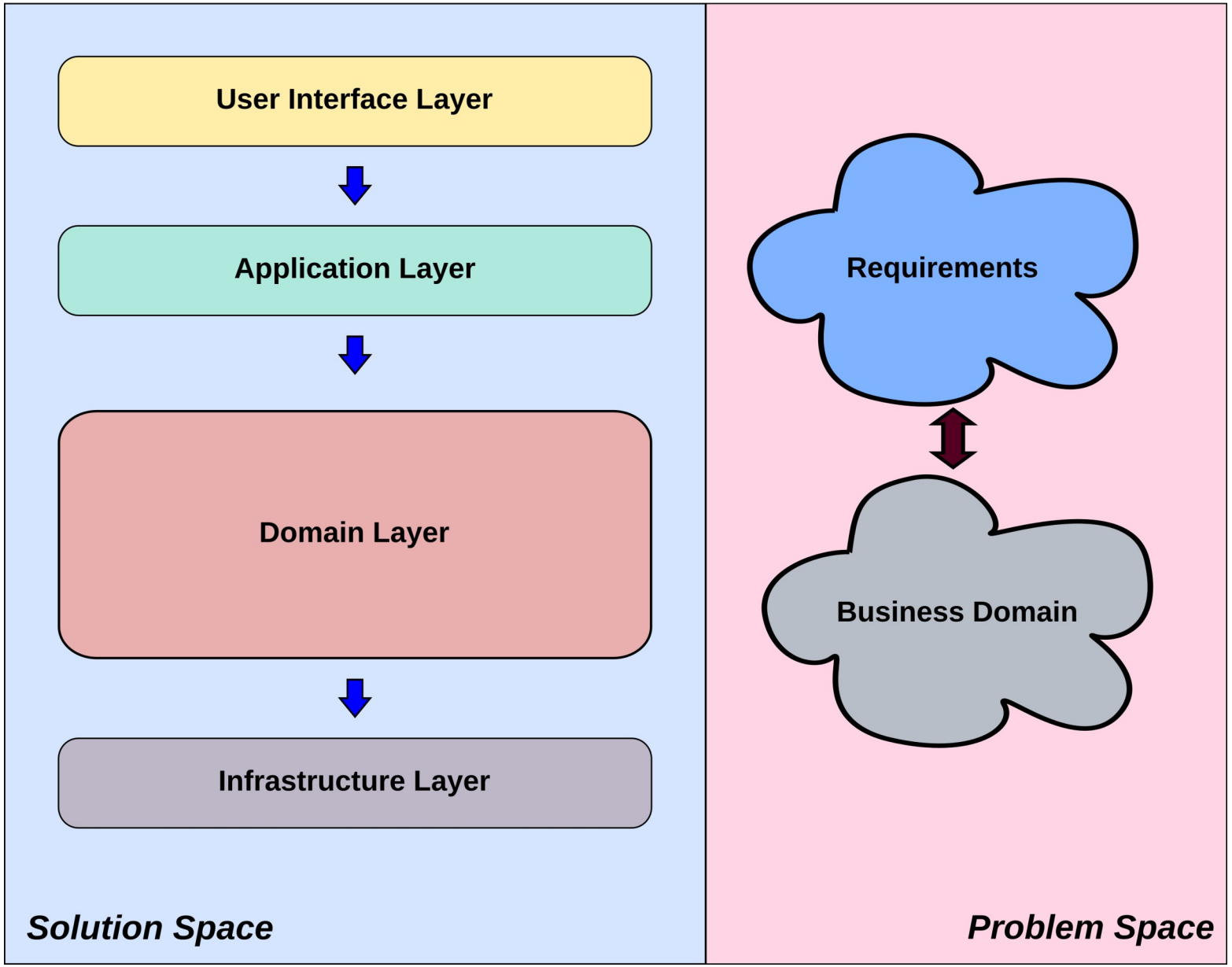
- Accounts
- Payments
- Orders
- Catalogue
- Shipments





LearnQuest

The Layered Architecture



The Problem Space

The Problem Space

Represents the business "as it is now."

The Requirements

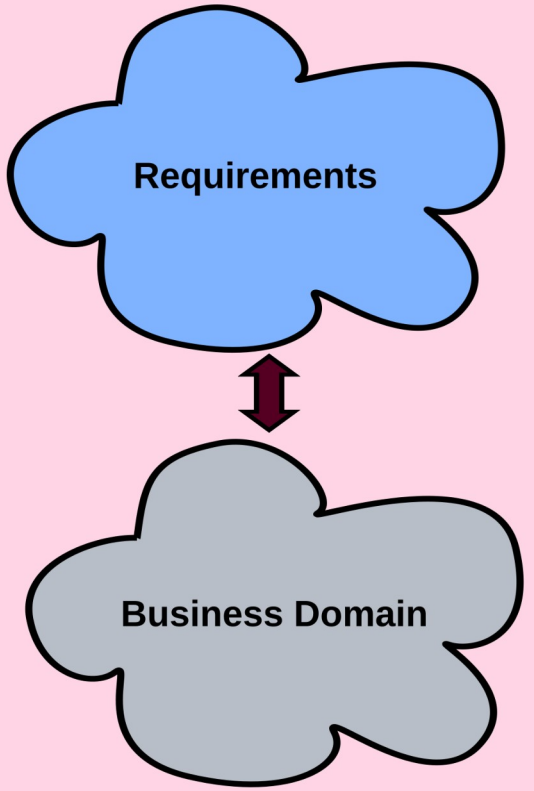
Requirements are statements of the needs of the stakeholders and users. Requirements describe what they need to do (eg. Add an account, make a payment, get a customer history).

These can also be thought of as the tasks that the business needs executed or "what" the business does.

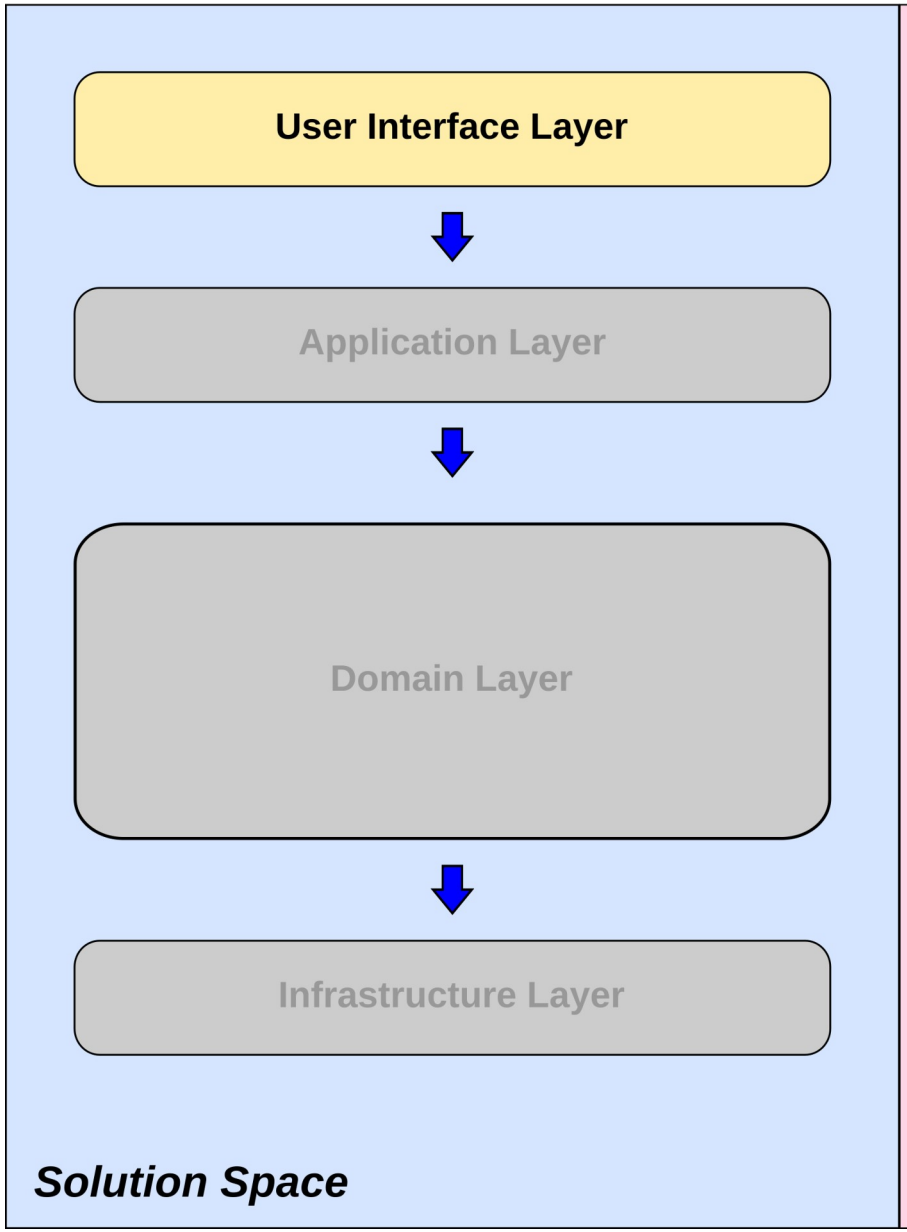
The Business Domain

This contains the algorithms, business rules, data objects, files, conceptual artifacts (eg. accounts, payments, invoices, shipments) used by the business to meet the requirements of the stakeholders.

*The business domain describes "**how**" the business goes about executing the tasks requested in the business requirements.*



User Interface Layer



Responsible for showing information to the user and interpreting the user's commands.

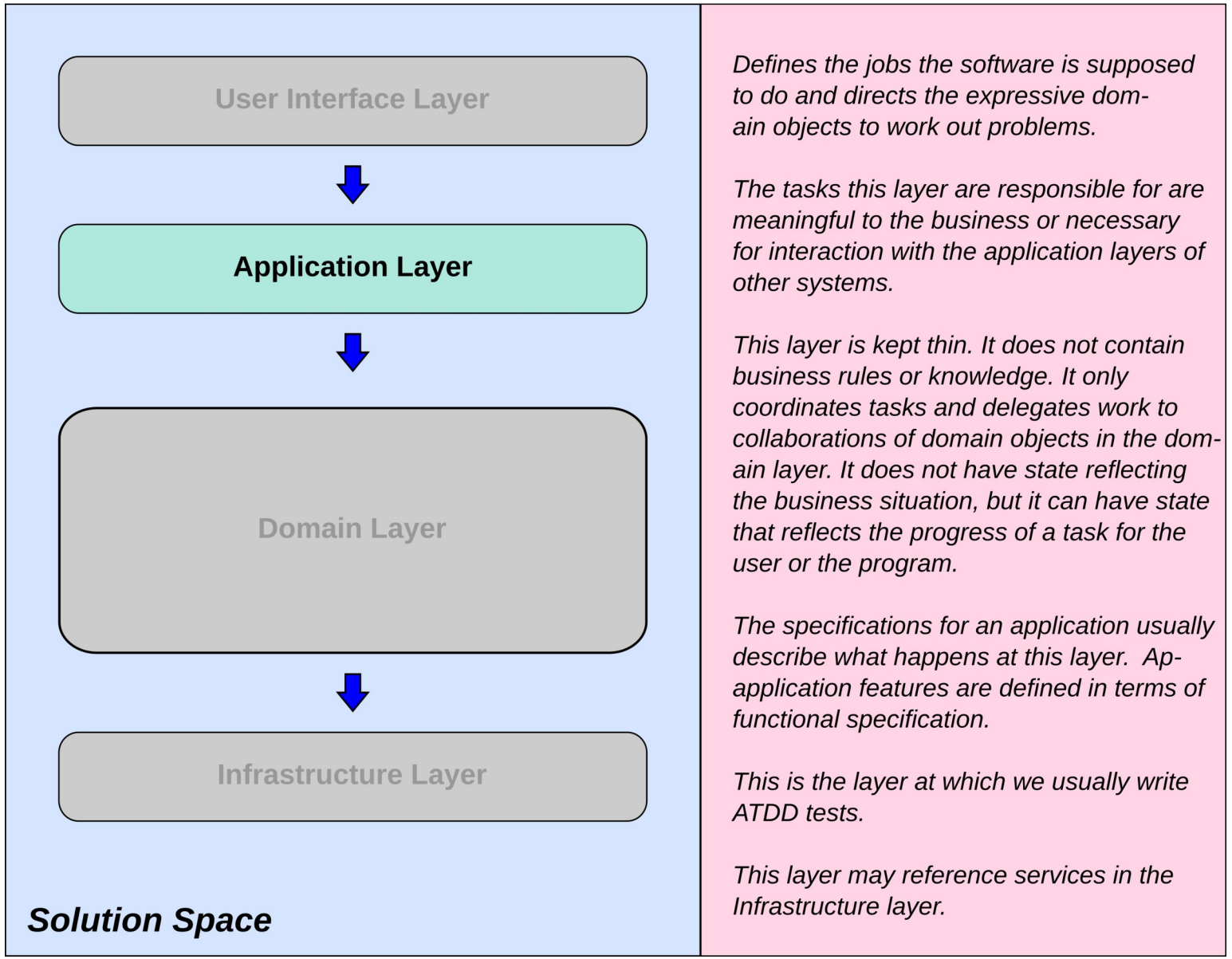
This layer only calls the applications in the Application Layer. It may occasionaly call a service provided by the Intrastructure Layer.

There may be multiple actual interfaces in this layer (web, RESTful API. etc).

Users may be humans or they may be other systems or computers.

The Application Layer

This is where user stories live



Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems.

The tasks this layer are responsible for are meaningful to the business or necessary for interaction with the application layers of other systems.

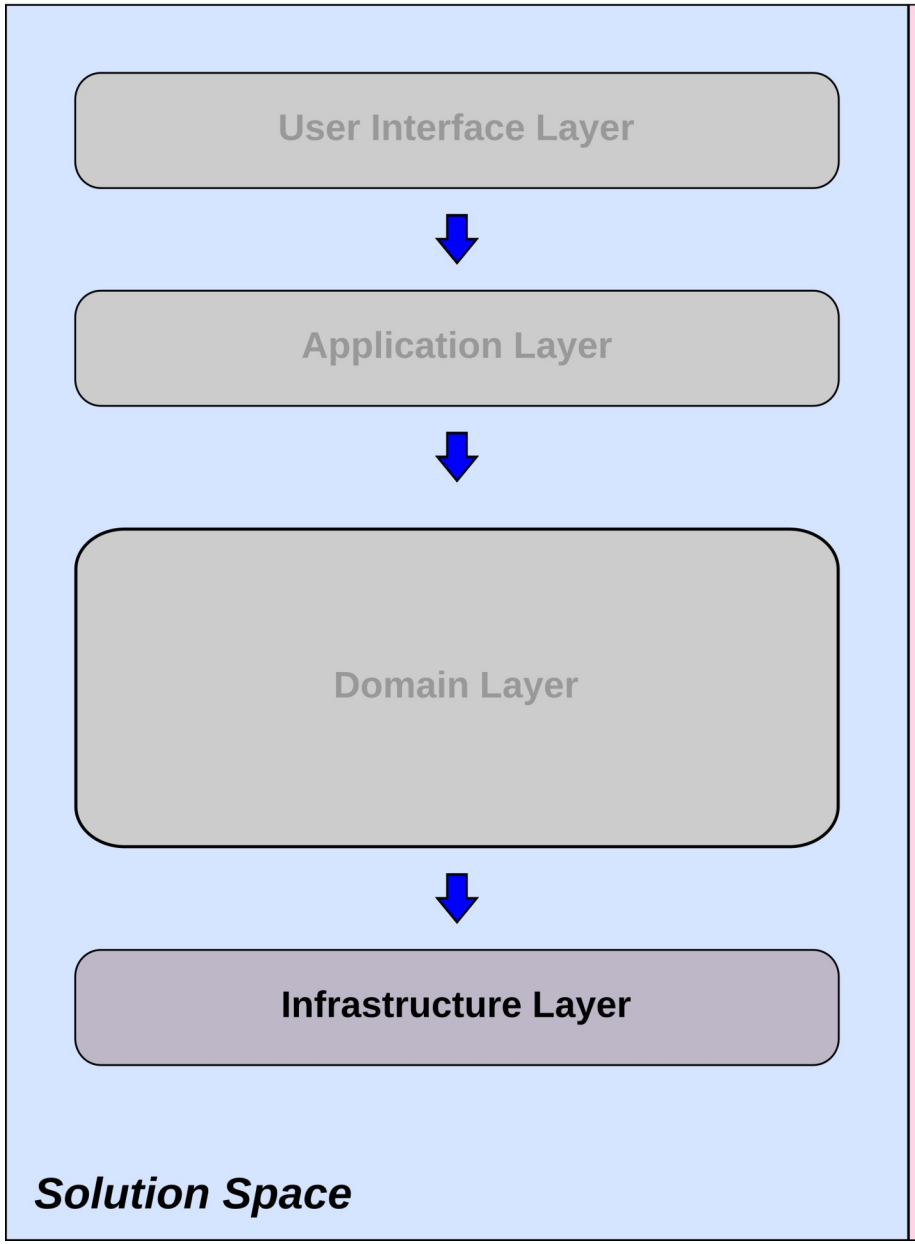
This layer is kept thin. It does not contain business rules or knowledge. It only coordinates tasks and delegates work to collaborations of domain objects in the domain layer. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.

The specifications for an application usually describe what happens at this layer. Application features are defined in terms of functional specification.

This is the layer at which we usually write ATDD tests.

This layer may reference services in the Infrastructure layer.

The Infrastructure Layer



Provides generic technical capabilities that support the higher layers: message sending for the application, persistence for the domain, drawing widgets for the UI, and so on. The infrastructure layer may also support the pattern of interactions between the four layers through an architectural framework.

The application and domain layers call on the SERVICES provided by the infra-structure layer.

For example, sending an email is done by requesting a "message sending service" from the infrastructure layer.

The domain or application may know what message to send and when to send it, but the "how" the message is sent is delegated to the infrastructure layer

Physical data-bases, hardware interfaces

Design Concepts

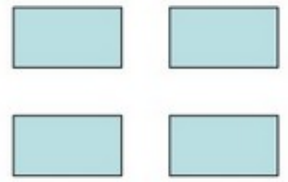
- Modular
- Coupling
- Cohesion
- Suppleness
- Dependency Injection (DI)
- Dependency Inversion Principle (DIP)
- Inversion of Control (IoC)

Modular

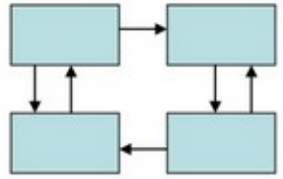
- Closely tied to abstraction
  - Allows the use of independent model implementation
  - Internal implementation is hidden
  - Communication is through stable interfaces
- Modules allow for reuse of components
  - Supports higher level architectural structuring of programs
- Enhances design clarity, simplifies implementation
  - Reduces cost and effort of testing, debugging and maintenance
- Module choice is not arbitrary
  - The choice of components is based on the domain

Coupling

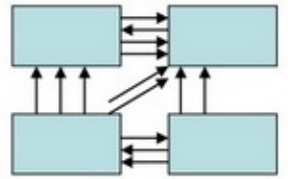
- Degree of dependence among components
  - High coupling makes modifying different parts of the system difficult
  - Modifying a highly coupled component affects all the other connected components
- High coupling often results in brittle and unusable systems



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

Suppleness

- Modular systems interact through interfaces
- Suppleness refers to the design of the interfaces
  - Follow 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
  - E.g. Data seamlessly shared between databases that use different units (pounds versus kilograms for example)

In Real Life

- A hospital provides a healthcare service
- Made up of individual microservices
  - Laboratory
  - X-Ray
  - Pharmacy
  - Medical Staffing
- Each microservice:
  - Specializes in a specific domain activity
  - Only that microservice performs that domain activity (the pharmacy doesn't do x-rays for example)
  - Each microservice operates autonomously

Interfaces in Real Life

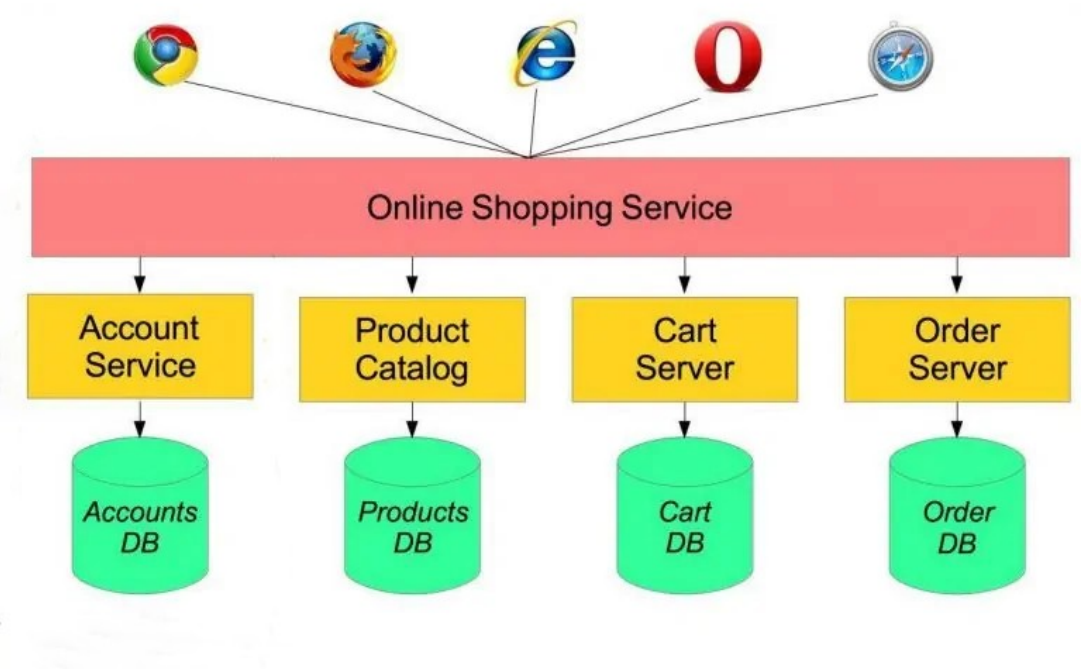
- Microservices request services from each other
  - They do not need to know the internal workings of the other microservice
- Requests are made through interfaces
  - Called APIs in software engineering
  - These are often paper based in the real world
  - Requisitions and official forms and paperwork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

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			

Replicating the Solution

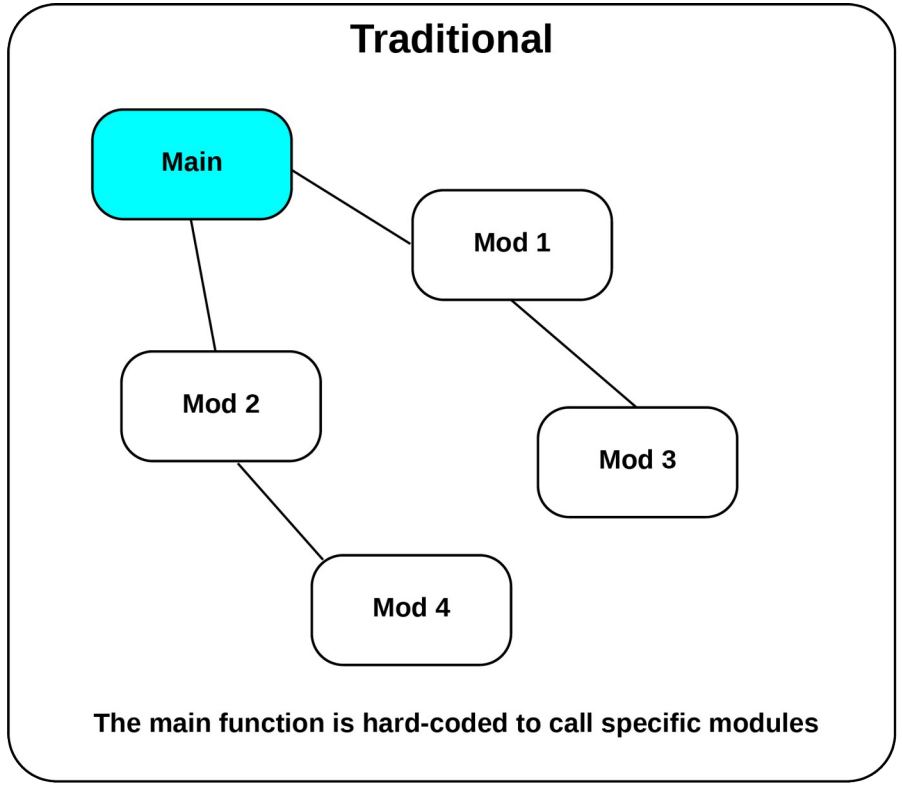
- Software architectural design choice
  - Enterprise applications are deployed as a set of microservices
  - Each microservice represents a clearly defined area of domain functionality
  - Each service manages requests from other services through an API
- Microservices are deployed as individual applications
  - Microservices are isolated from each other
  - They only communicate through APIs
  - They require an infrastructure to deploy them and ensure that they are all running and available
  - As an application executes, it makes requests of various microservices

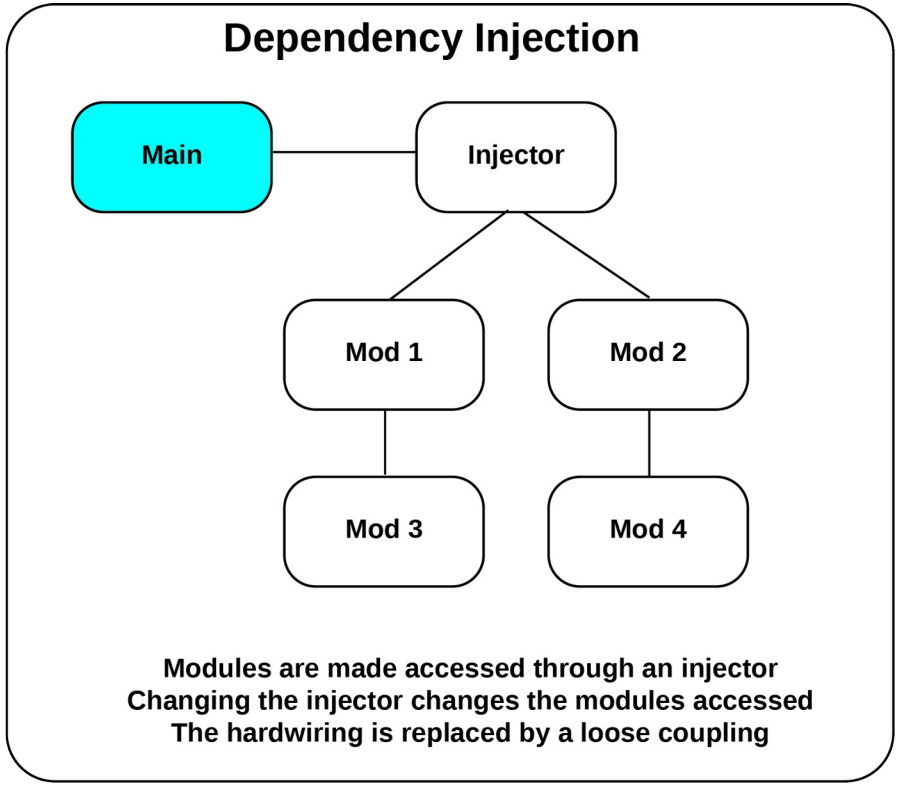


Dependency Injection

- 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





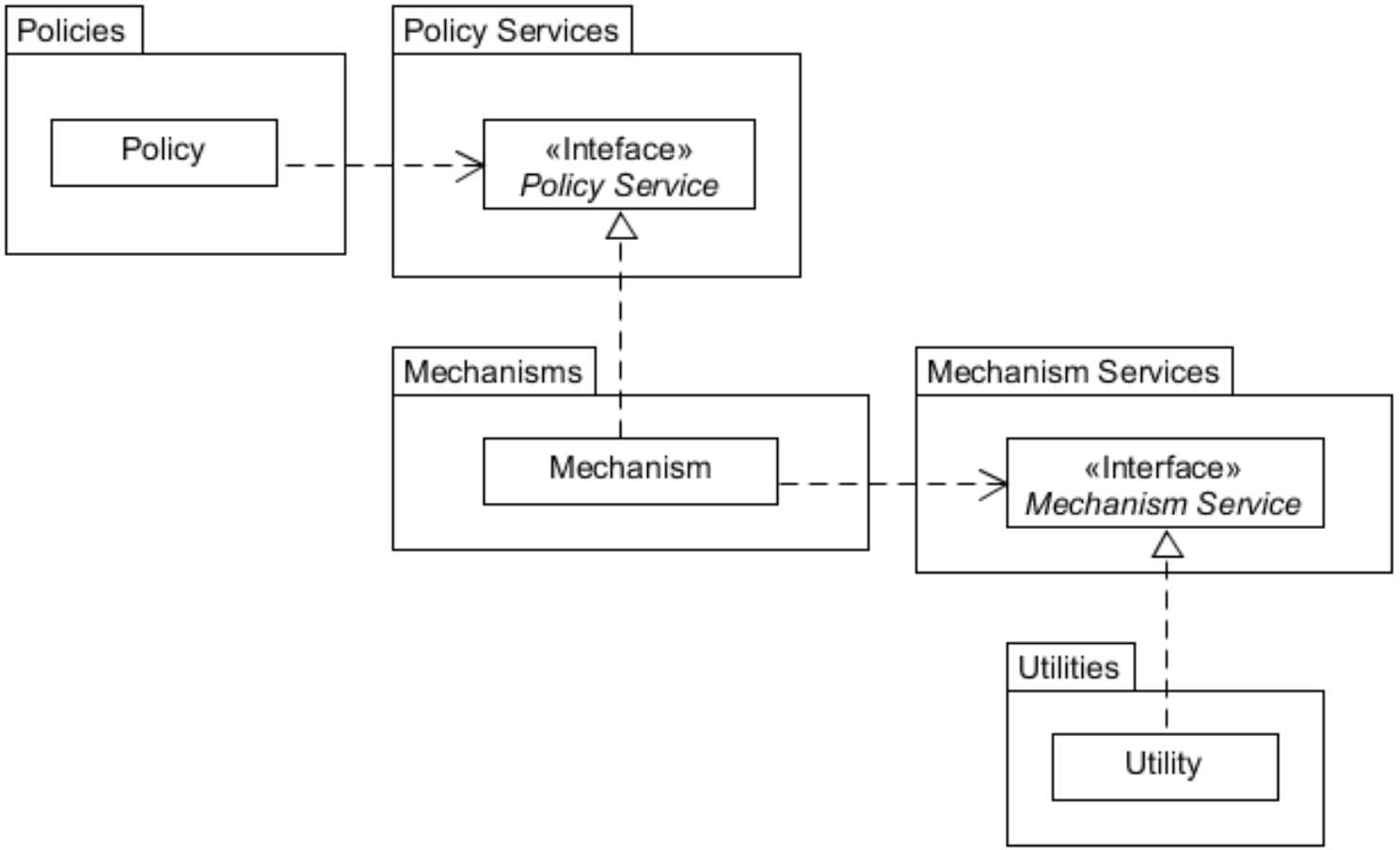
Dependency Inversion Principle

- Abstraction should be used in place of concrete implementations
  - High-level modules should not depend on low-level modules, and both types of modules should depend on abstraction (interface)
  - A concrete implementation should depend on a defined abstraction rather than an abstraction that is derived from a concrete implementation
- 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

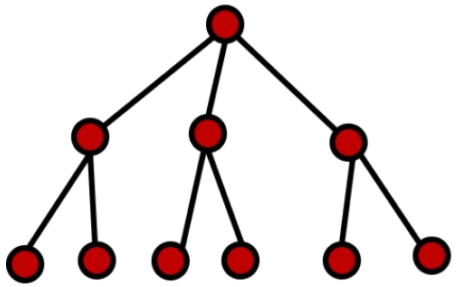
- Abstraction should be used in place of concrete implementations
  - High-level modules should not depend on low-level modules, and both types of modules should depend on abstraction (interface)
  - A concrete implementation should depend on a defined abstraction rather than an abstraction that is derived from a concrete implementation
- 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

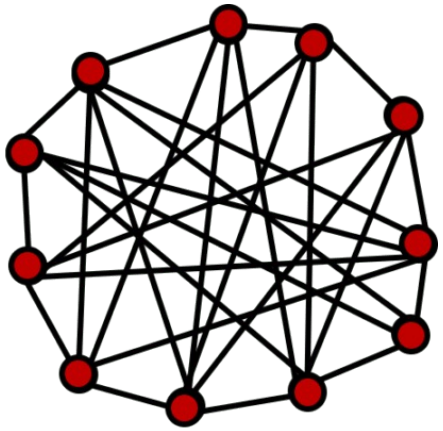


Inversion of Control

- 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



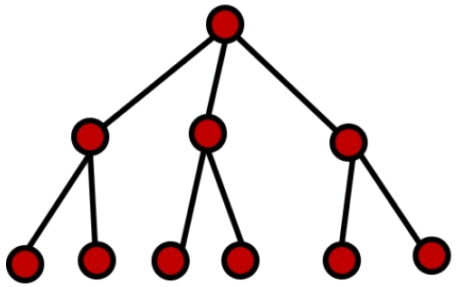
"Top-down"



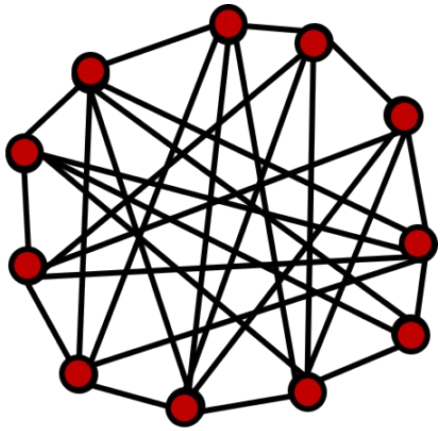
"Bottom-up"

Inversion of Control

- In IoC, control starts with an “event” or client request
  - Different flows of control 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

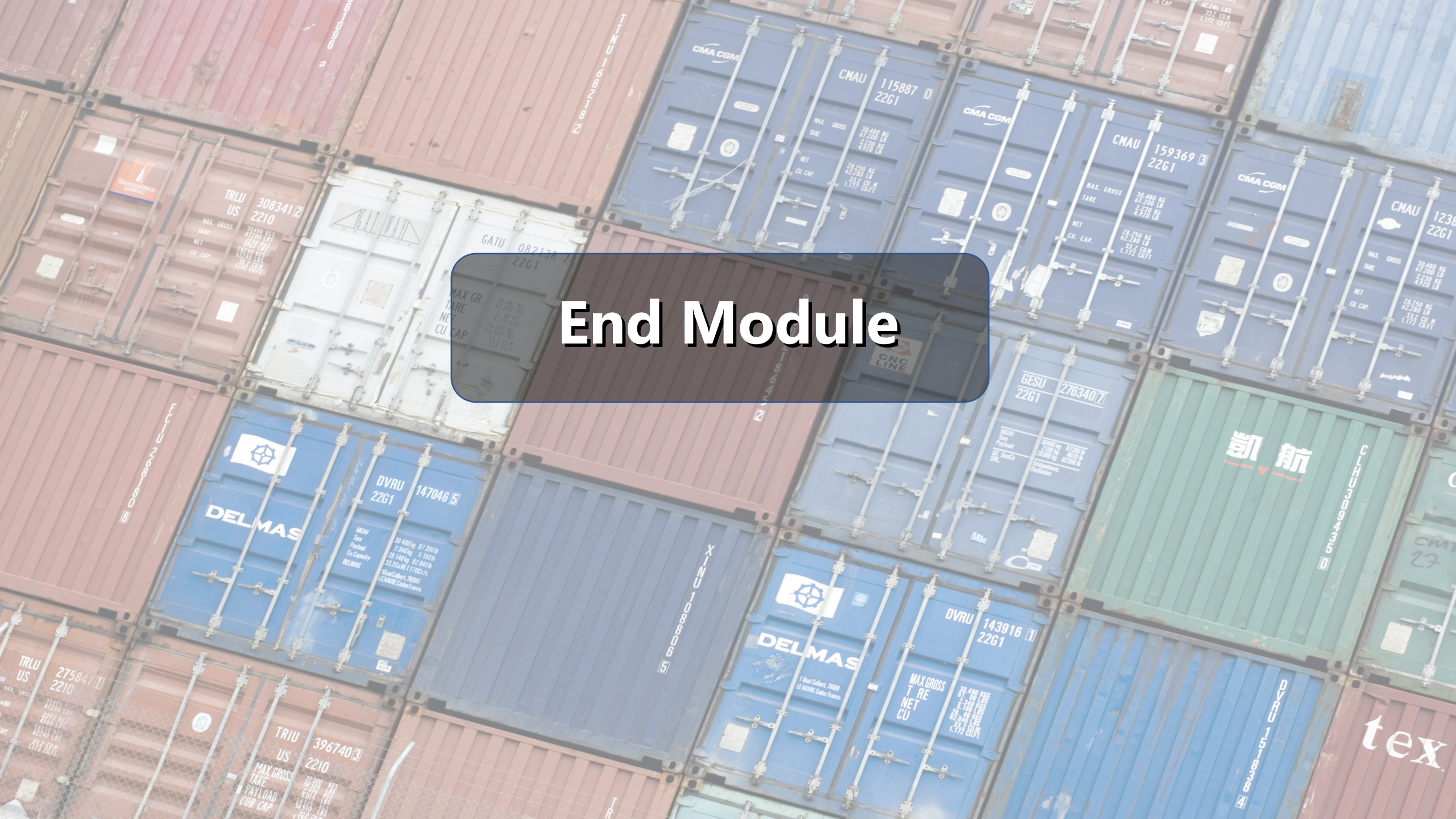


“Top-down”



“Bottom-up”


Questions?


End Module