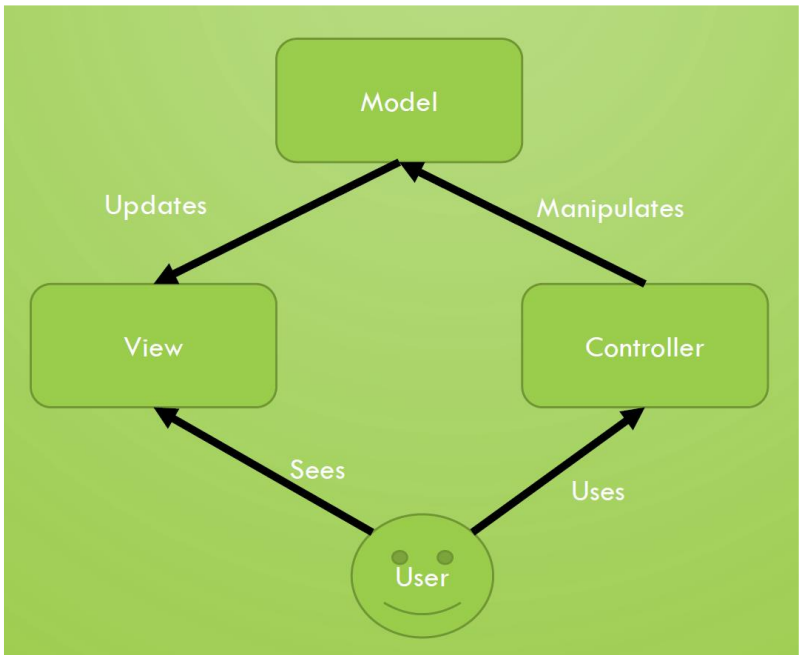
Software Architect's Knowledge checklist (Blue – in external documents)

- MVC. Model responsibilities, View responsibilities, Controller Responsibilities
- MVC (Model-View-Presenter)
- MVVP (Model-View-ViewModel)
- 3-Tier (Presentation Layer, Business Layer, Database Layer)
- Separation of Concerns
- Bad Practices – Resume Driven Development (RDD, based on "cool technology", not the best suitable)
- Dependency injection
- Inversion of control
- REST API methods (GET, POST, PUT, PATCH, DELETE, HEAD). **PUT vs POST, PATCH vs PUT**
- Versioning. **Semantic Versioning.**
- Scaffolding
- VIZERAL Theorem
- CAP Theorem, PARCELC Theorem – when it is **eventual consistency**
- Conway's Law – The need of communication inside organization between teams. Microservices are against Conway's Law.
- ISO/IEC 42010
- Architectural Element
- Architectural definition
- Architectural description
- **Views and viewpoints**
- **Perspectives and concerns of each perspective**
- Test types:
    a. Unit tests (class methods),
    b. integration tests (CodeMVC + ORM DB),
    c. Acceptance Testing (UAT) – Browser Simulation,
    d. Feature Toggling – for consumer group (ENABLE / DISABLE)
    e. Feature Throttling - % of customers to be affected by new feature
    f. A/B Tests
- Scalability – Sharding
- Cisco Enclave Model
- High Availability (99,999+) VS Availability
- Architectural Styles – Multi-Tier (Horizontal), Multi-Layer (Verical, i.e. OSI Model)
- Archetypes
- SOA (Service-Oriented Architecture). Principles of SOA.
- Enterprise Service Bus (ESP)
- Vetro Pattern
- Service Orchestration VS Choreography – When to choose each other, **practical examples**
- Microservices
- Design patterns – **main patterns, which to choose when**, and what are the trade-offs, **gang of four**

(Web Frameworks) Resume Driven Development (RDD)

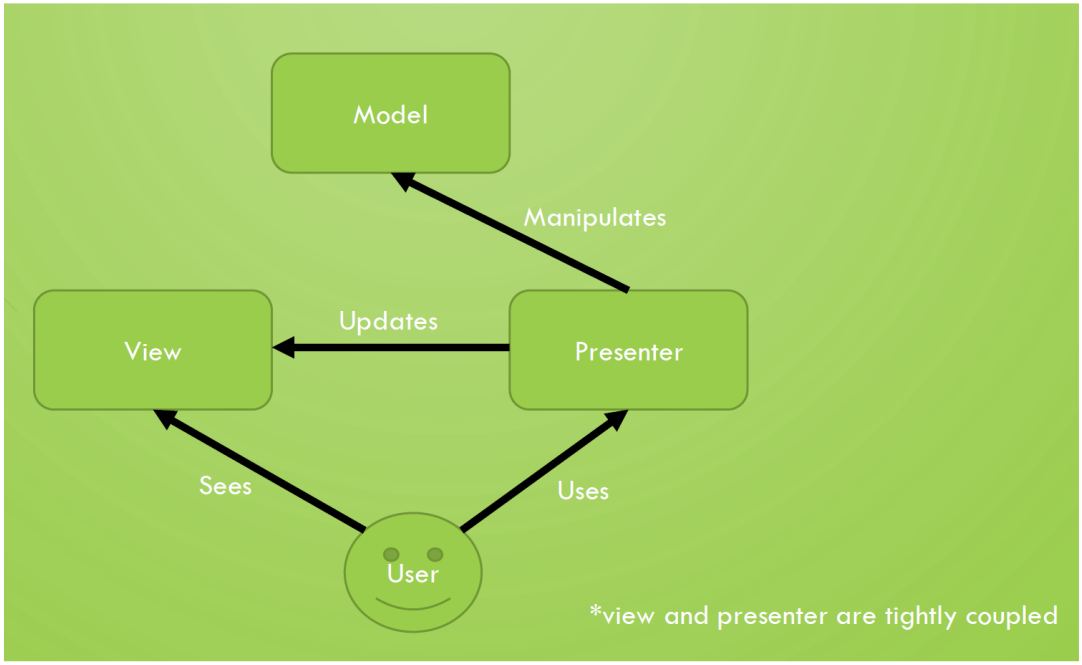
At it's most basic Resume Driven Development is the practice of picking new or popular technologies for your work in order to make your resume look more impressive. This means using unfamiliar technology to solve a problem that could be solved with known technology. You may not even know much about the stack that you are trying to use and what the limitations are before you start doing it.

(Web Frameworks) MODEL-VIEW-CONTROLLER:

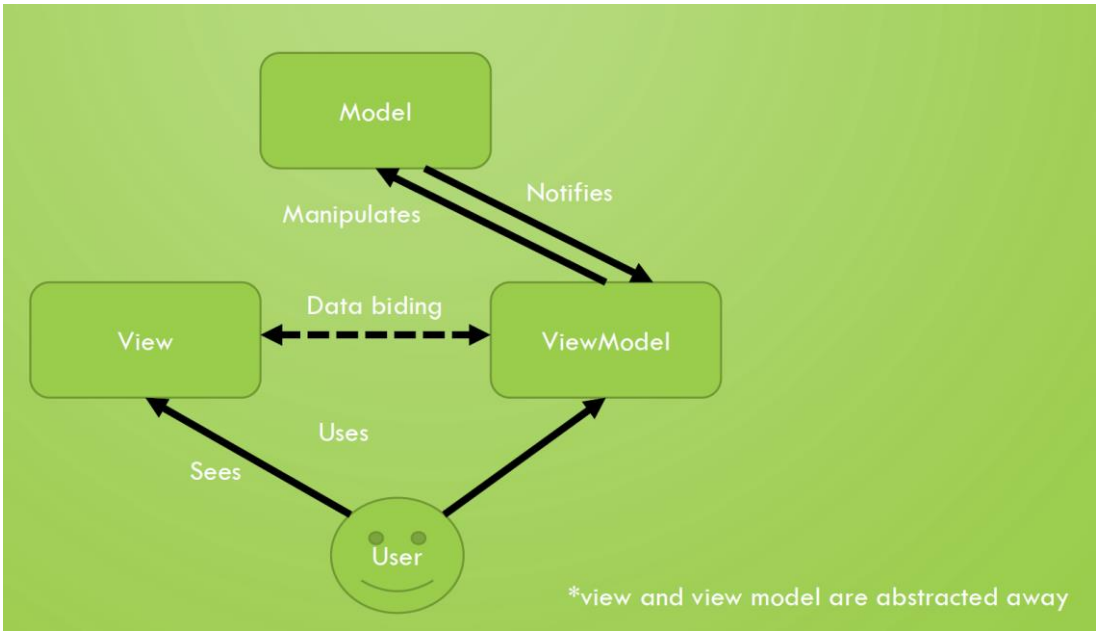


Model Responsibilities:

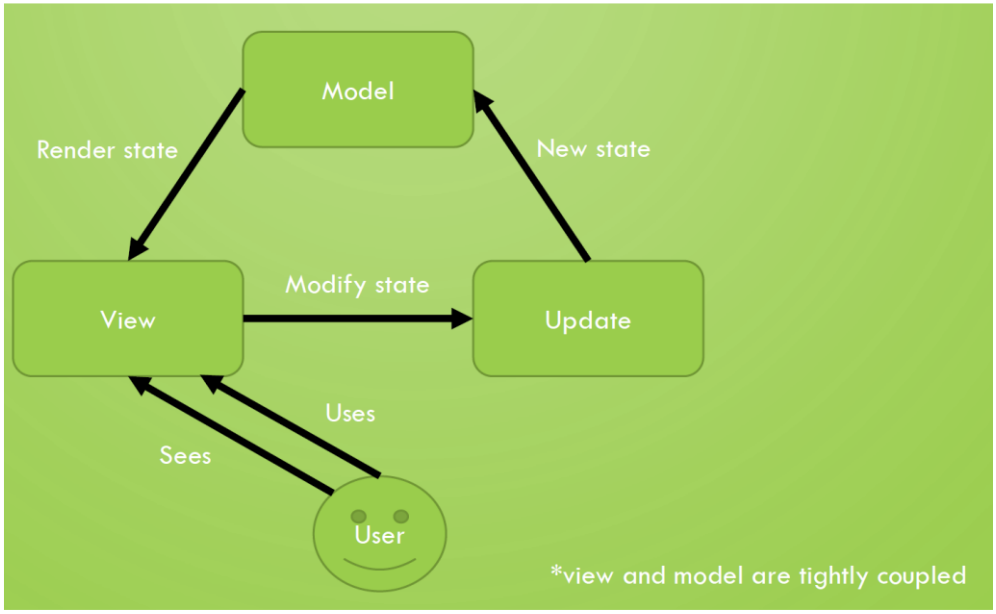
(Web Frameworks) MODEL-VIEW-PRESENTER:

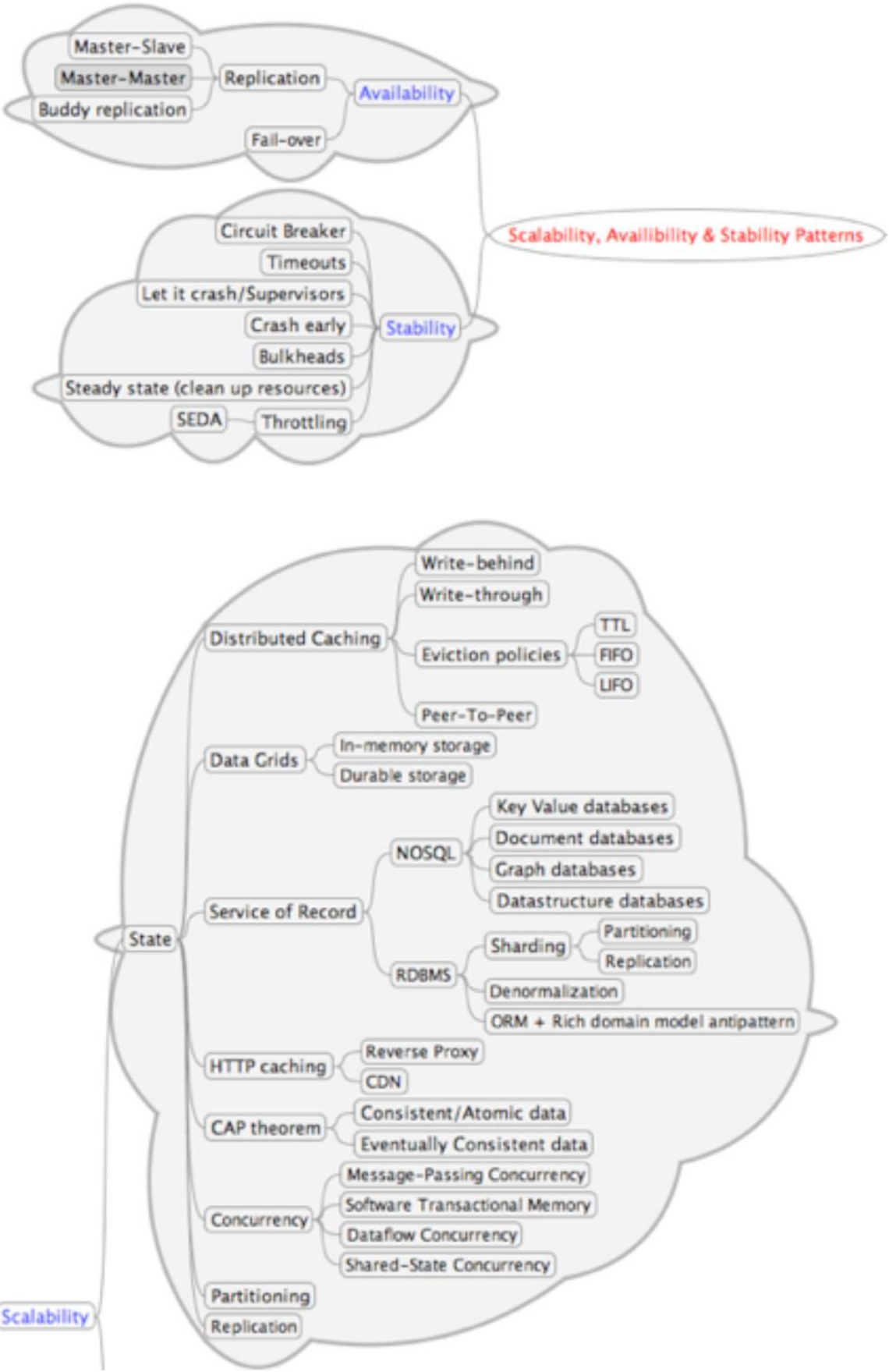


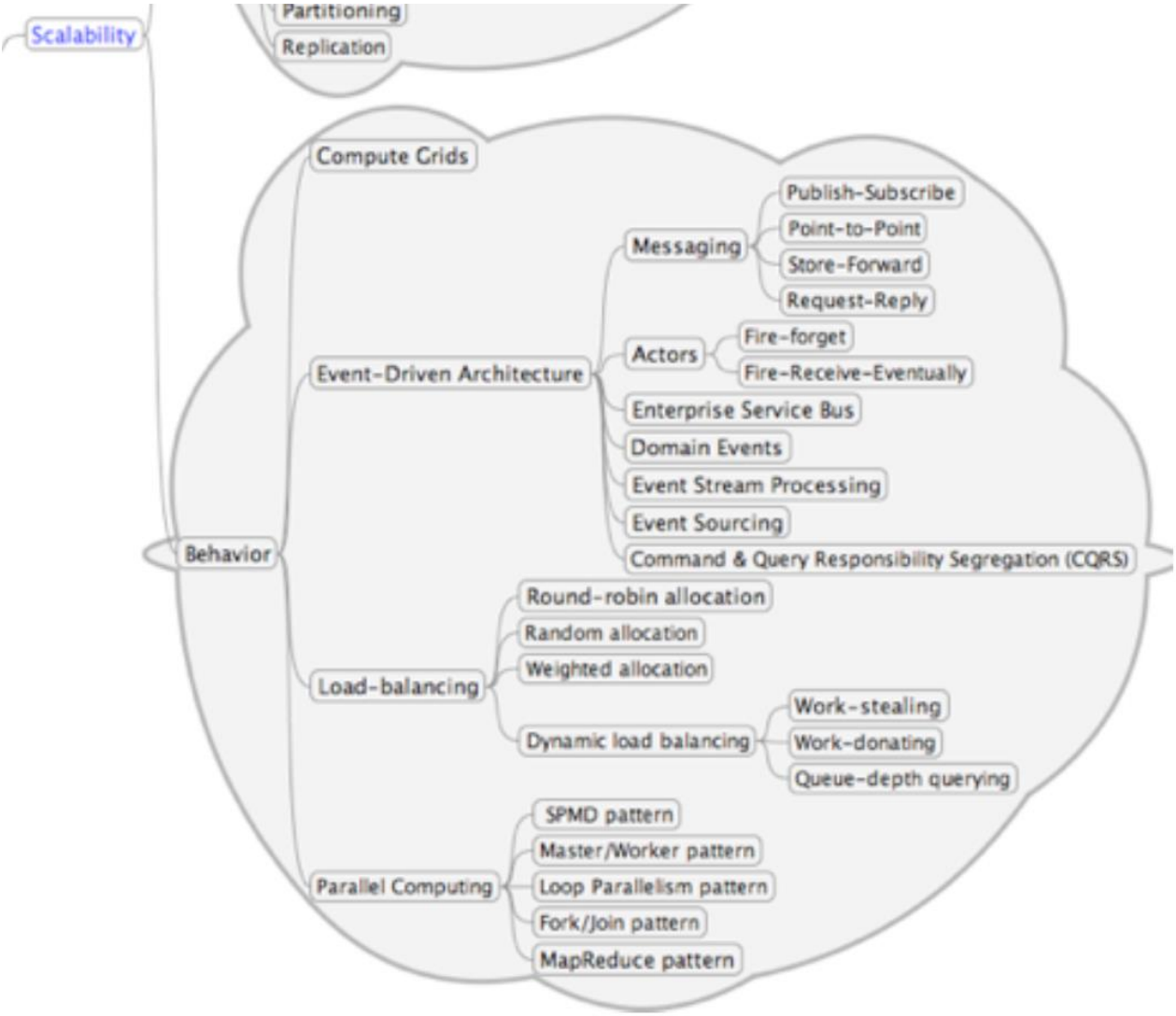
(Web Frameworks) MODEL-VIEW-VIEWMODEL:



(Web Frameworks) MODEL-VIEW-UPDATE:




Scalability, Availibility & Stability Patterns
Availability
Replication
Master-Slave
Master-Master
Buddy replication
Fail-over
Stability
Circuit Breaker
Timeouts
Let it crash/Supervisors
Crash early
Bulkheads
Steady state (clean up resources)
SEDA
Throttling
Scalability
State
Distributed Caching
Write-behind
Write-through
Eviction policies
TTL
FIFO
LIFO
Peer-To-Peer
Data Grids
In-memory storage
Durable storage
Service of Record
NOSQL
Key Value databases
Document databases
Graph databases
Datastructure databases
RDBMS
Sharding
Partitioning
Replication
Denormalization
ORM + Rich domain model antipattern
HTTP caching
Reverse Proxy
CDN
CAP theorem
Consistent/Atomic data
Eventually Consistent data
Concurrency
Message-Passing Concurrency
Software Transactional Memory
Dataflow Concurrency
Shared-State Concurrency
Partitioning
Replication



Scale-Up (left, more power / server) vs Scale-Out (right, more servers)

(12. Intro to architecture) DEVELOPER, SOFTWARE ENGINEER, SOFTWARE ARCHITECT

- Developer + Curiosity + Analytical Mindset = Software Engineer
- Software Engineer + Contextual Knowledge + Strategy Driven Mindset = Software Architect

(12. Intro to architecture) THE FUNDAMENTAL PROPERTIES OF A SYSTEM MANIFEST THEMSELVES IN TWO DIFFERENT WAYS

A. *Externally visible behavior* (what the system does) and functional interactions between the system and its environment (users and other systems).
    a. Defined by functional requirements
B. *Quality properties* (how the system does it).
    a. Nonfunctional properties (performance, security, availability, etc.)

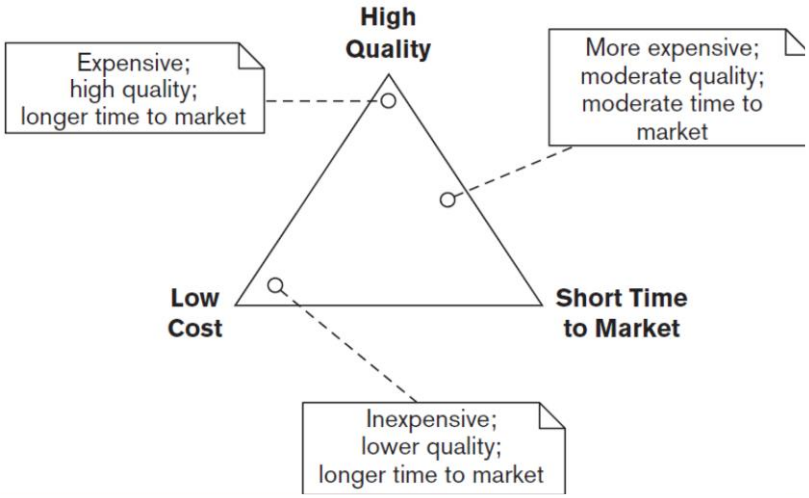
(12. Intro to architecture) ARCHITECTURE VERSUS DESIGN

Architecture	Systems
	Communication between systems
	Services or subsystems
	Communication between services or subsystems
	Modules
Design	Modules
	Packages
	Components
	Classes

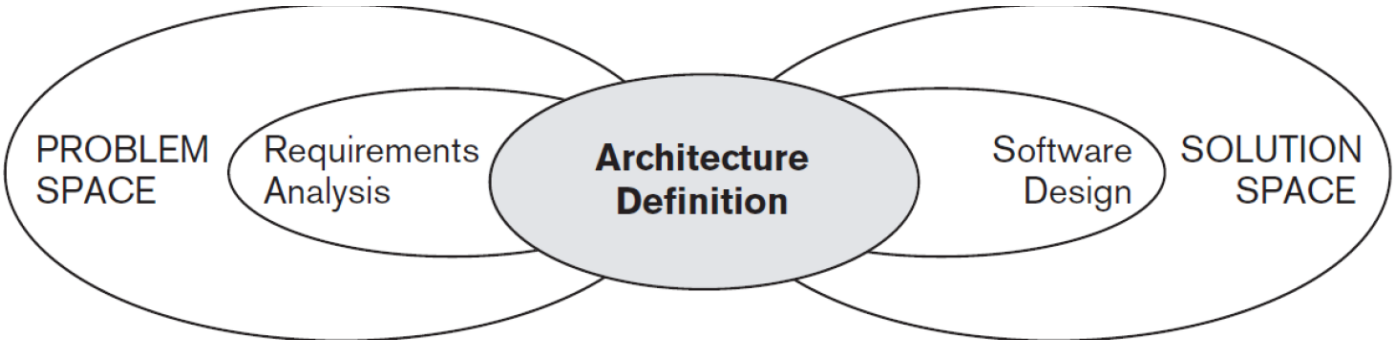
(12. Intro to architecture) ARCHITECTURE MUST CONFORM TO

1. Stakeholder needs
2. Software engineering principles and best practices

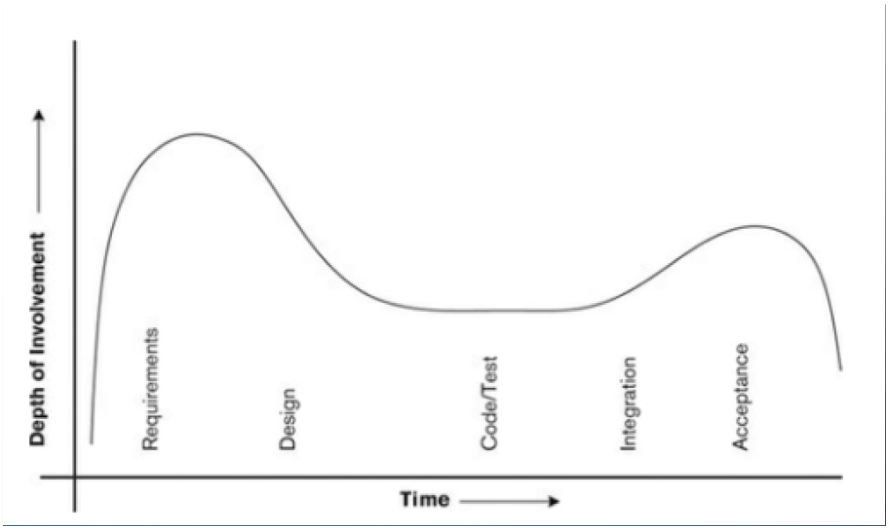
(12. Intro to architecture) QUALITY VS COST VS TIME



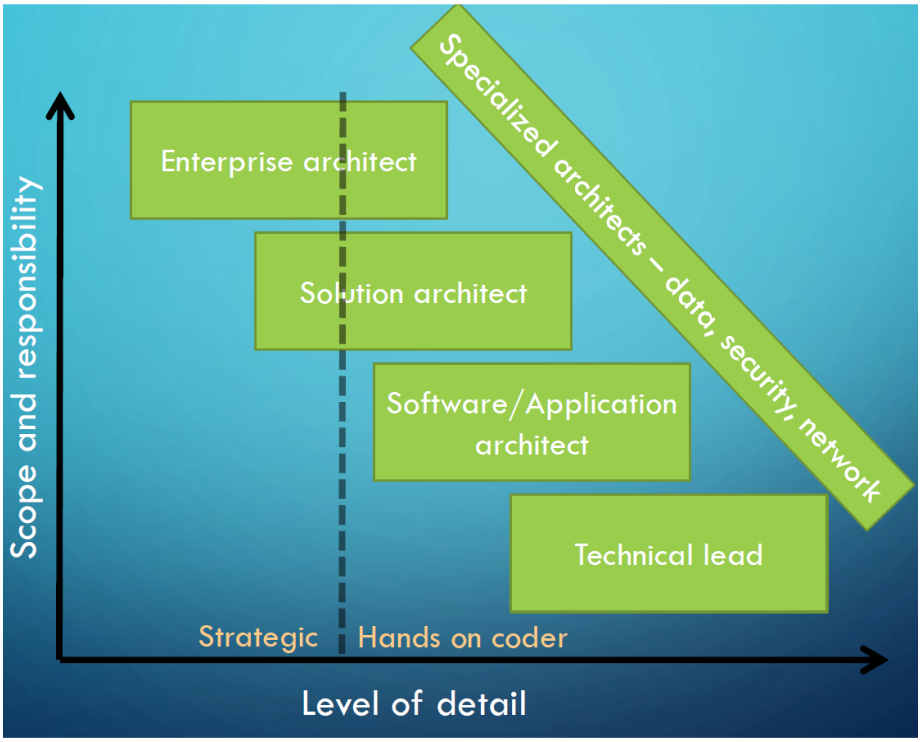
(15. Architect‘s role) ARCHITECT‘S FIELD OF OPERATION:



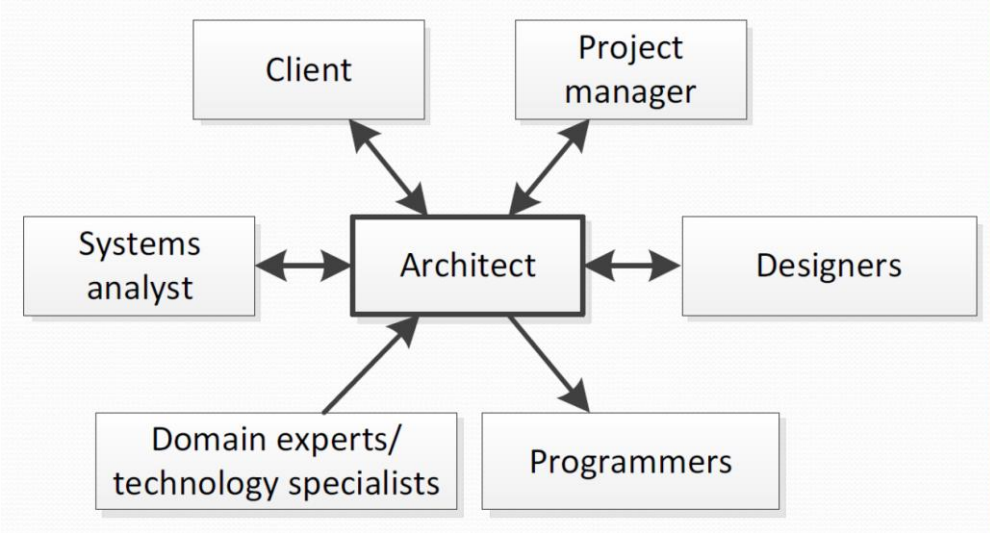
(15. Architect‘s role) ARCHITECT‘S INVOLVEMENT IN SOFTWARE DEVELOPMENT CYCLE:



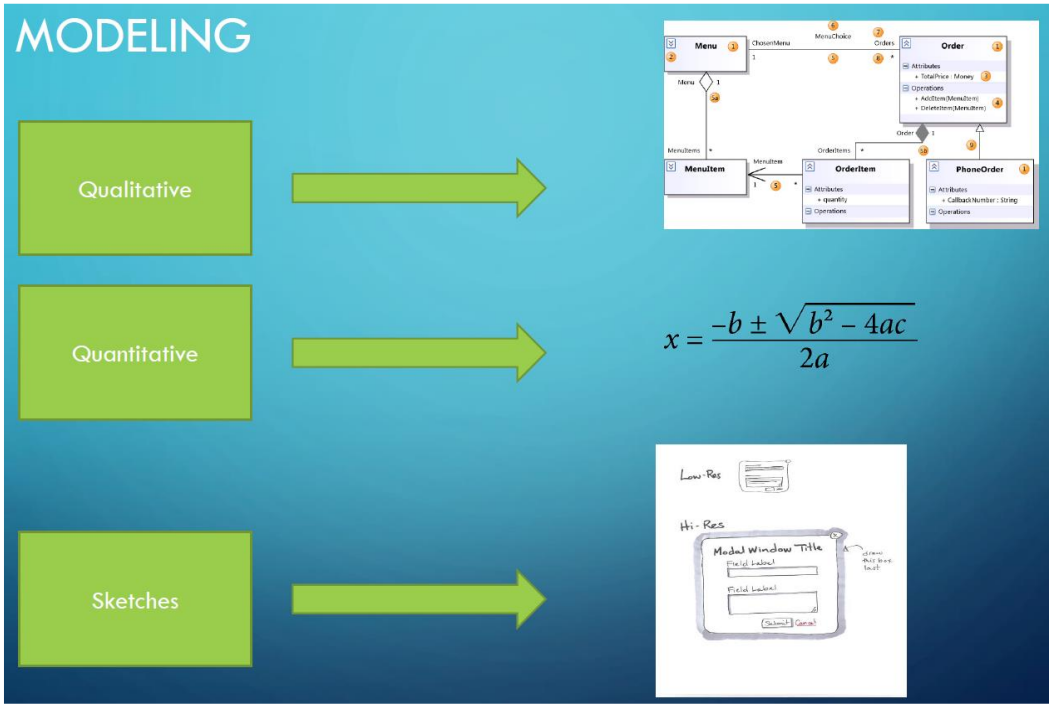
(15. Architect‘s role) TYPES OF ACHITECTS:



(15. Architect‘s role) ARCHITECT AND TEAM



(15. Architect‘s role) MODELING DONE BY ARCHITECT



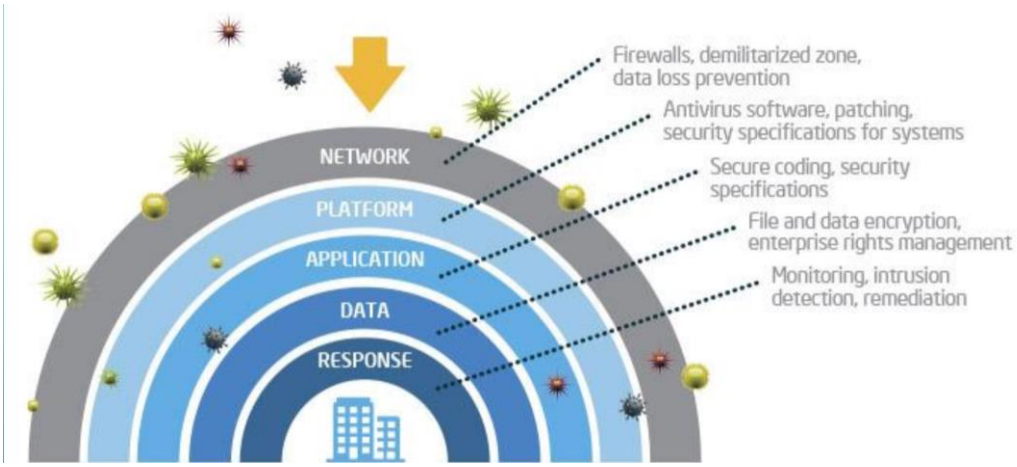
(25. Security Perspective) SECURITY DEFINITION

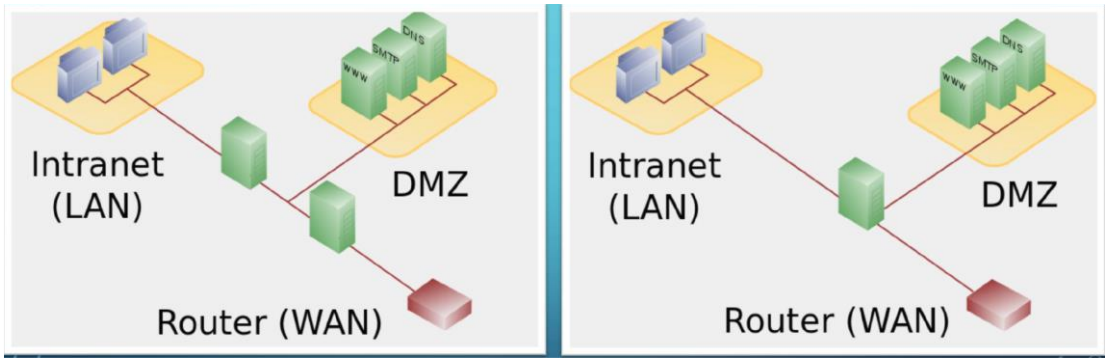
- **SECURITY** IS AN ABILITY OF THE SYSTEM TO RELIABLY CONTROL, MONITOR AND AUDIT WHO CAN PERFORM WHAT ACTIONS ON WHICH RESOURCES AND THE ABILITY TO DETECT AND RECOVER FROM SECURITY BREACHES.
- SECURITY IS ALSO A RISK MANAGEMENT PROCESS AGAINST COSTS OF GUARDING FROM THREATS

(25. Security Perspective) TRUST MODEL

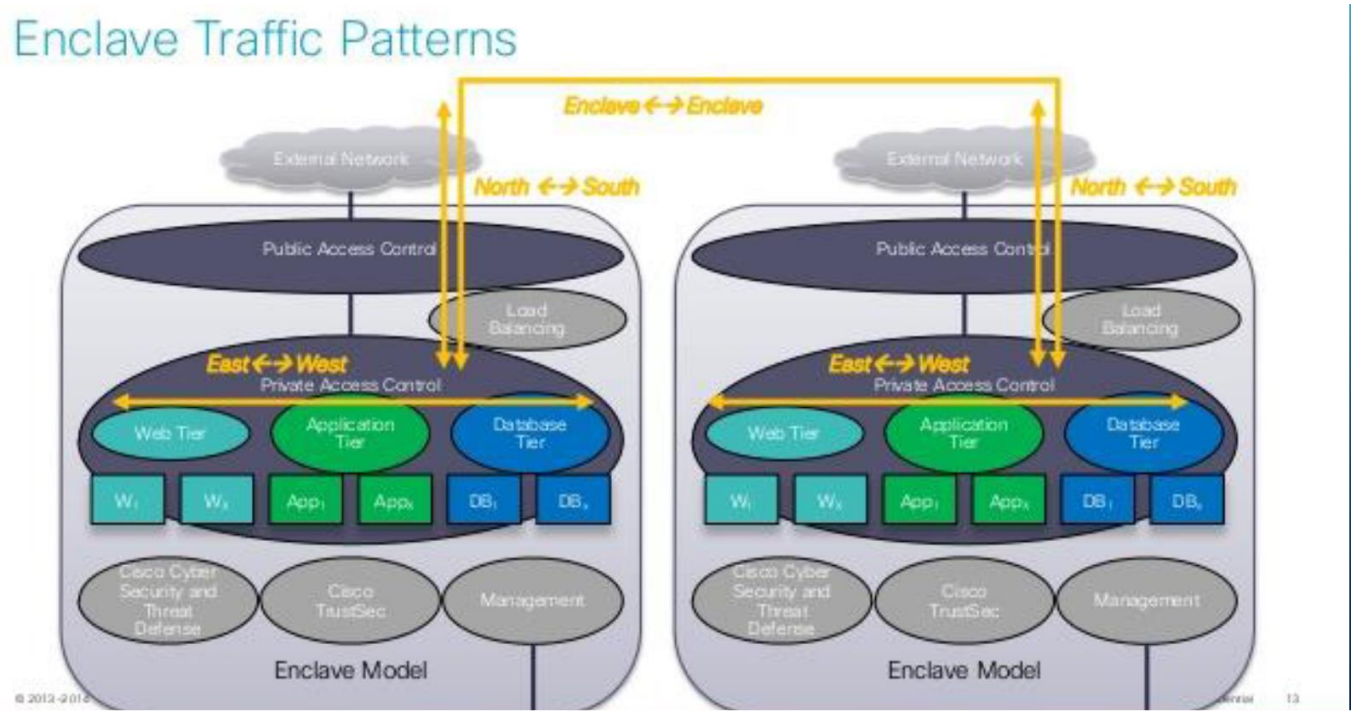
	User Account Records	Product Catalog Records	Pricing Records	User Account Operations	Product Catalog Operations	Price Change Operations
Data administrator	Full with audit	Full with audit	Full with audit	All with approval and audit	All with audit	All with approval from a product price administrator
Catalog clerk	None	None	None	All	Read-only operations	None
Catalog manager	None	None	None	Read-only operations with audit	All	All with audit
Product price administrator	None	None	None	None	Read-only operations	All with audit
Customer care clerk	None	None	None	All with audit	Read-only operations	None
Registered customer	None	None	None	All on own record	Read-only operations	None
Unknown Website user	None	None	None	None	Read-only operations	None

(25. Security Perspective) DEMILITARIZED ZONE (DMZ) IN NETWORKS

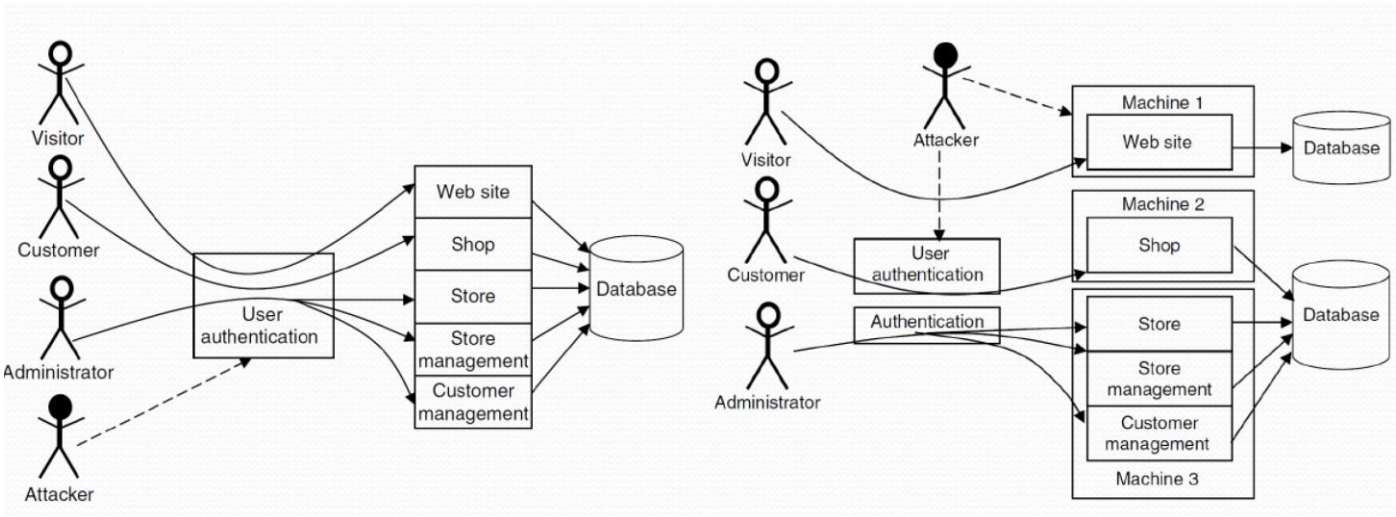




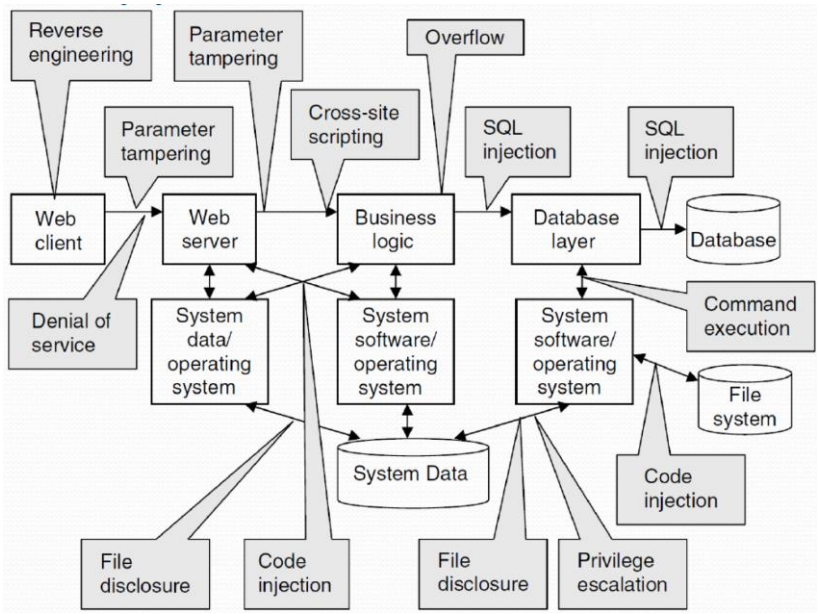
(25. Security Perspective) CISCO ENCLAVE MODEL ARCHITECTURE



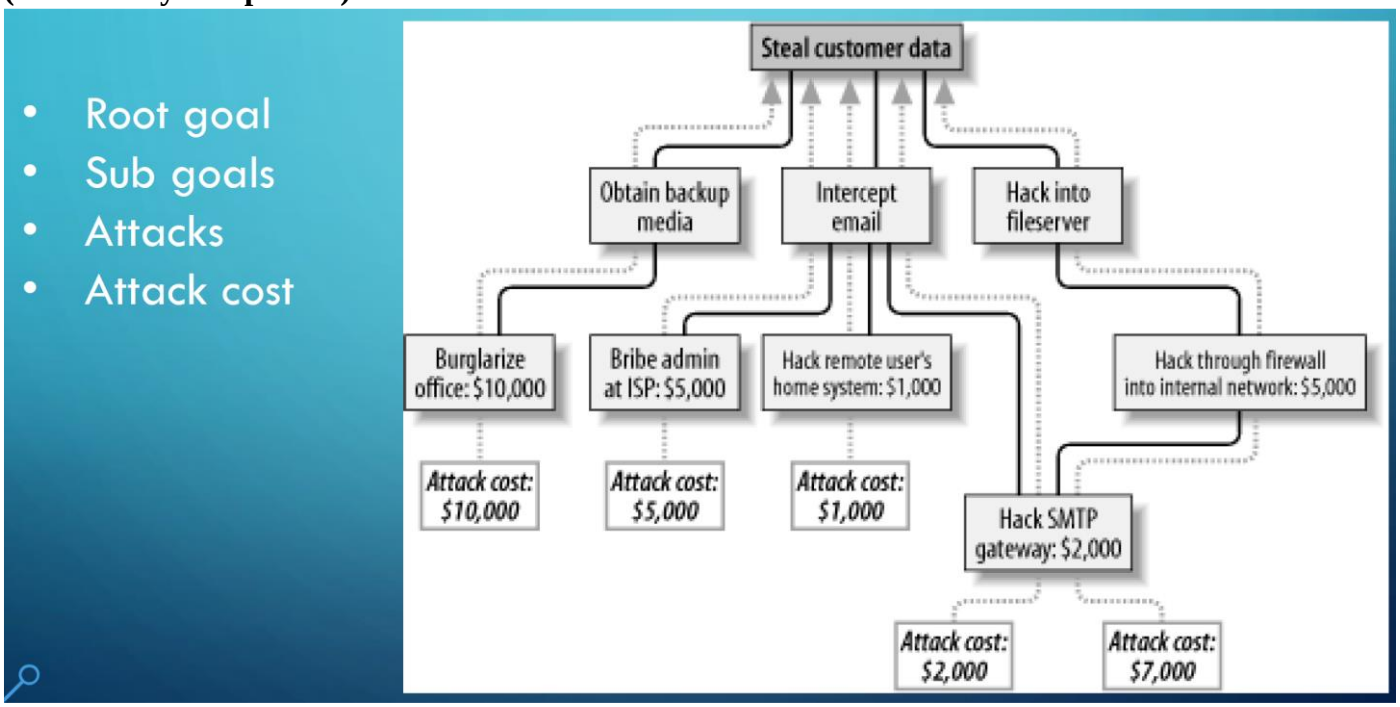
(25. Security Perspective) ATTACK SURFACE AND IT'S REDUCTION



(25. Security Perspective) VULNERABILITY POINTS



(25. Security Perspective) ATTACK TREES



(25. Security Perspective) RISK ASSESMENT TABLES

Risk	Estimated cost	Estimated likelihood	Notional cost
Direct access to database	8,000,000	0.2%	16,000
Web-site flaw, free orders placed and fulfilled	800,000	4.0%	32,000
Social-engineering attack, access to customer accounts	4,000,000	1.5%	60,000

(25. Security Perspective) GOOD SECURITY PRINCIPLES

- Grant least amount of privileges possible
- Secure the weakest link
- Defend in depth
- Separate and compartmentalize
- Keep security design simple
- Don’t rely on obscurity
- Use secure defaults
- Fail securely
- Assume external entities are untrusted
- Audit sensitive events

(28. Evolution Perspective) 4 TYPES OF MAINTENANCE

• ***Corrective maintenance*:** Reactive modification of a software product performed after delivery to correct discovered problems
• ***Adaptive maintenance*:** Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment
• ***Perfective maintenance*:** Modification of a software product after delivery to improve performance or maintainability
• ***Preventive maintenance*:** Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults

(28. Evolution Perspective) TYPES OF EVOLUTION

•Functional evolution – can relate to 4 types of maintenance
•Platform evolution – software and hardware
•Integration evolution – evolution due to 3rd party software system changes
•Growth – increased system use / load / complexity

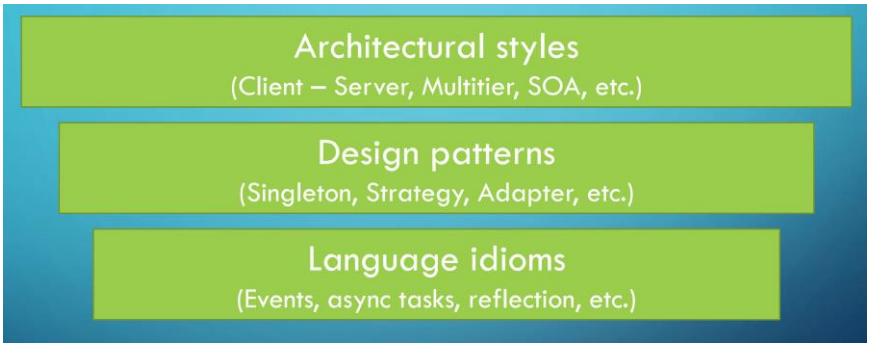
(28. Evolution Perspective) DESIGN TACTICS

- Separation of concerns
- Encapsulation
- Single point of definition – DRY (Do not Repeat Yourself)
- Functional cohesion
- Low coupling
- Abstract common services
- Abstraction and layering
- Generalization patterns
- Inversion of control / dependency injection
- Interface segregation

(28. Evolution Perspective) VARIATION POINTS

What varies	Design Pattern
Algorithms	Strategy, Visitor
Actions	Command
Implementations	Bridge
Response to change	Observer
Interactions between objects	Mediator
Object being created	Factory Method, Abstract Factory, Prototype
Structure being created	Builder
Traversal Algorithm	Iterator
Object interfaces	Adapter
Object behavior	Decorator, State

(29. Architectural styles) ARCHITECTURAL STYLES, DESIGN PATTERNS, LANG IDIOMS

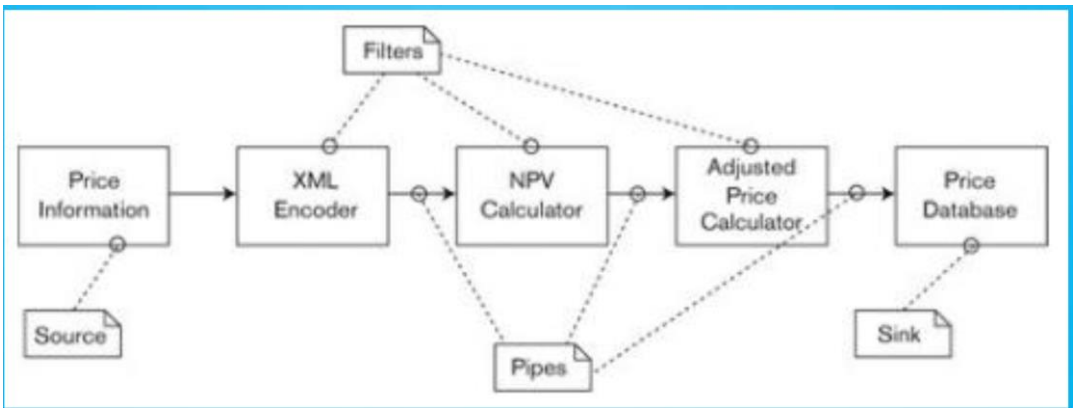


ARCHITECTURAL STYLE DEFINE SYSTEM-LEVEL STRUCTURES VIA SET OF ORGANIZATIONAL PRINCIPLES FOR THE SYSTEM AS A WHOLE.

A DESIGN PATTERN DOCUMENTS A COMMONLY RECURRING AND PROVEN STRUCTURE OF INTERCONNECTED DESIGN ELEMENTS THAT SOLVES A GENERAL DESIGN PROBLEM WITHIN A PARTICULAR CONTEXT.

A LANGUAGE IDIOM IS A PATTERN SPECIFIC TO A PROGRAMMING LANGUAGE. AN IDIOM DESCRIBES HOW TO IMPLEMENT PARTICULAR ASPECTS OF ELEMENTS OR THE RELATIONSHIPS BETWEEN THEM BY USING THE FEATURES OF A GIVEN LANGUAGE

(29. Architectural styles) PIPES AND FILTERS



- The **filter** transforms or *filters* the data it receives via the pipes with which it is connected. A filter can have any number of input pipes and any number of output pipes.
- The **pipe** is the connector that passes data from one filter to the next. It is a directional stream of data, which is usually implemented by a data buffer to store all data, until the next filter has time to process it.
- The **pump** or producer is the data source. It can be a static text file, or a keyboard input device, continuously creating new data.
- The **sink** or consumer is the data target. It can be another file, a database, or a computer screen.

Advantages of pipes & filters:

- Reuse and interchangeability of components
- Supports concurrent execution

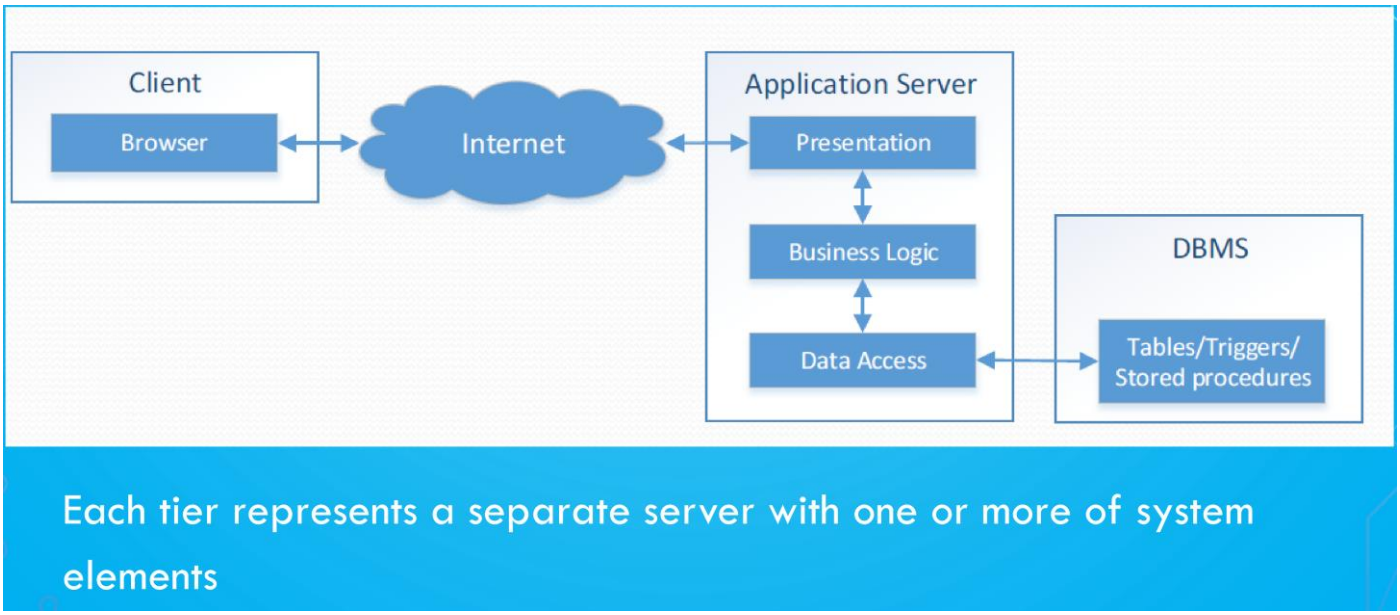
Disadvantages of pipes & filters:

- Performance - slowest filter affects whole chain

(29. Architectural styles) COMMON ARCHITECTURAL STYLES IN WEB SYSTEMS

• Multi-tier / n-tier / MVC
• Layers
• Publish/Subscribe, Message bus, Broker
• Event driven architecture a.k.a. Reactive
• CQS, CQRS, Event Sourcing
• SOA (Service Oriented Architecture)
• DDD (Domain Driven Design)
• RESTful APIs

(29. Architectural styles) MULTI-TIER STYLE



Common tiers in multi-tier style:

1. **Client** – usually thin client-browser
2. **Application Server**
    a. Application Server is responsible for:
        i. Presentation (user interface preparation; e.g. HTML generation)
        ii. Business processes (combining a sequence of business transactions into a service)
        iii. Business transactions/Application Services (the fundamental business operations in the system that act on business entities)
        iv. Data Access
3. **Data Storage** – the databases in which the data resides

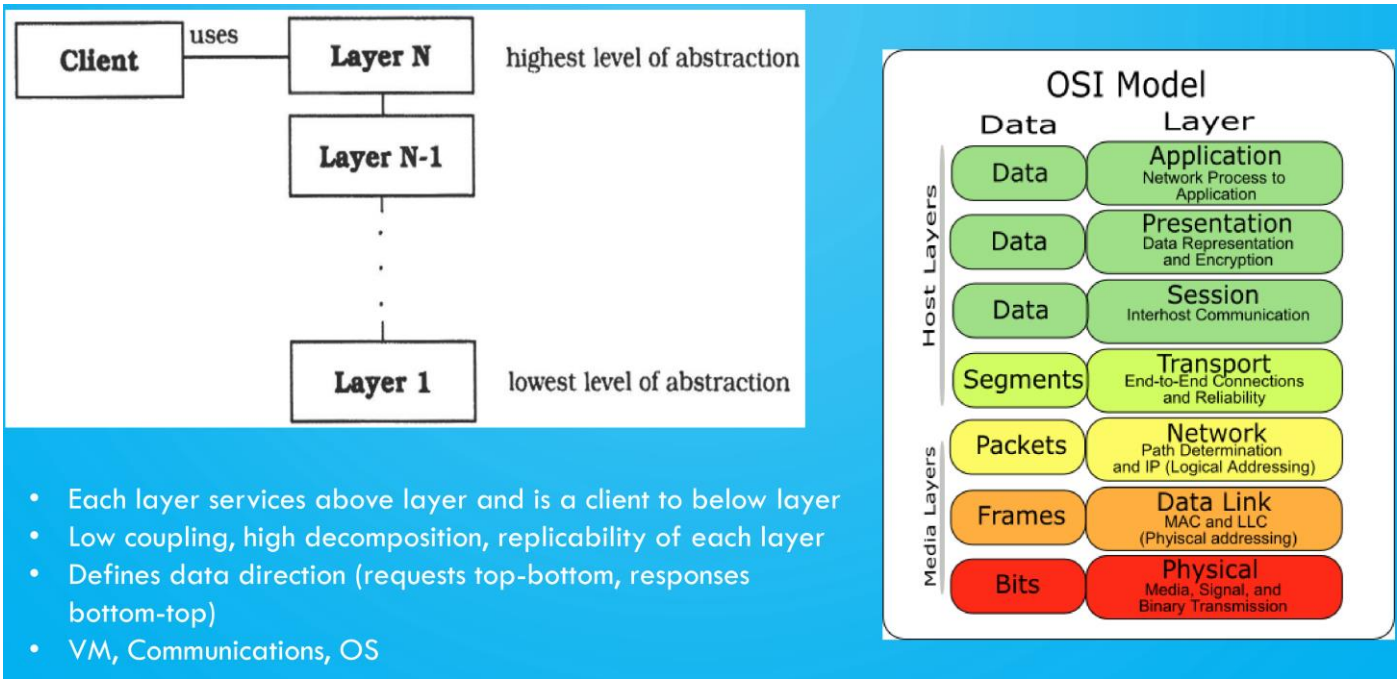
Advantages of multi-tier style:

- Clear separation of concerns
- Distribution across multiple machines

Disadvantages of multi-tier style:

- Network communication overhead
- Maintainability

(29. Architectural styles) LAYERS



OSI Model, as well as TCP/IP model used in networking (internet protocols, hardware's firmware)

1. Layers **not equal** to N-TIER style
2. **Layers** define **logical location**, where **tiers** define **physical location**

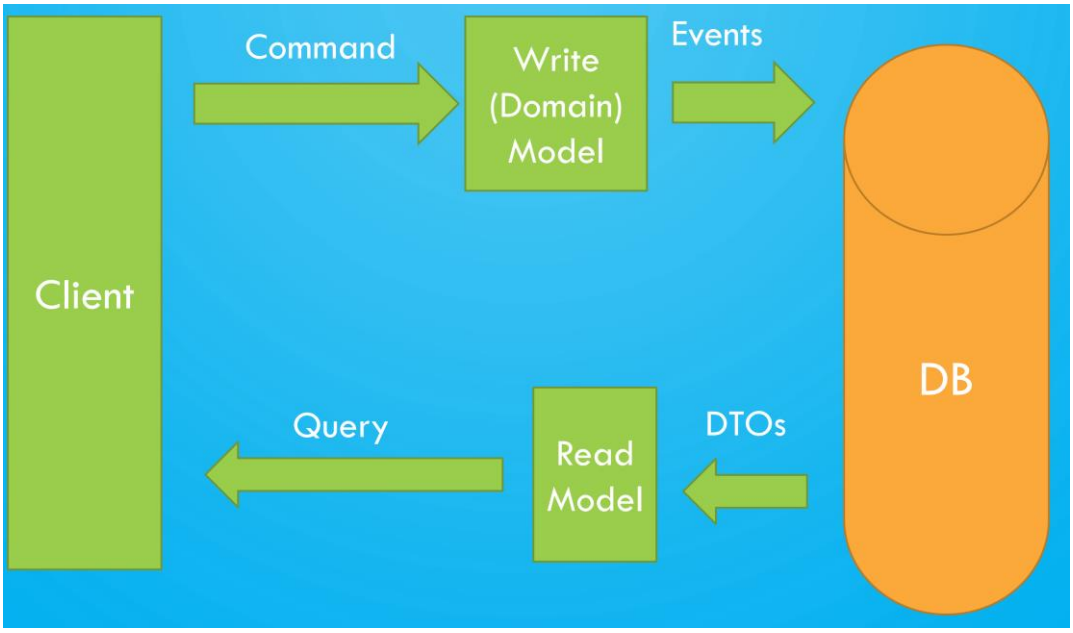
Advantages of layers:

- Standardization
- Reuse of layers
- Localized dependencies

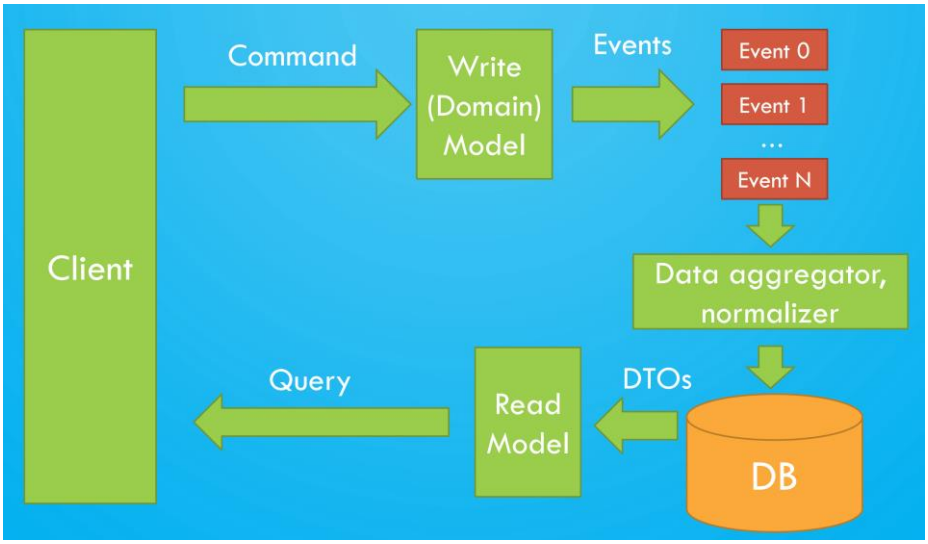
Disadvantages of layers:

- Cascades when changing behavior
- Difficulty to establish correct granularity of layers

(29. Architectural styles) COMMAND-QUERY RESPONSIBILITY SEGREGATION (CQRS)



(29. Architectural styles) CQRS AND EVENT SOURCING



Advantages of Command-Query Responsibility Segregation (CQRS) & event sourcing:

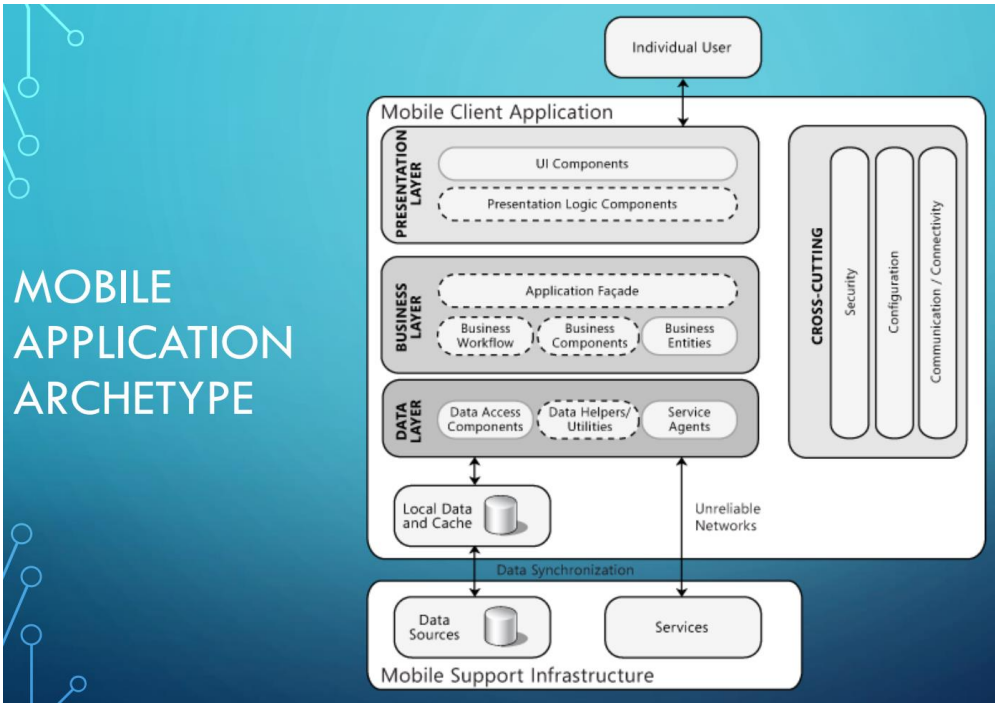
- Ability to replay or restore to a certain point in time
- Tailored write and read for performance
- Clear separation of read/write allows fine tuning the bottlenecking part

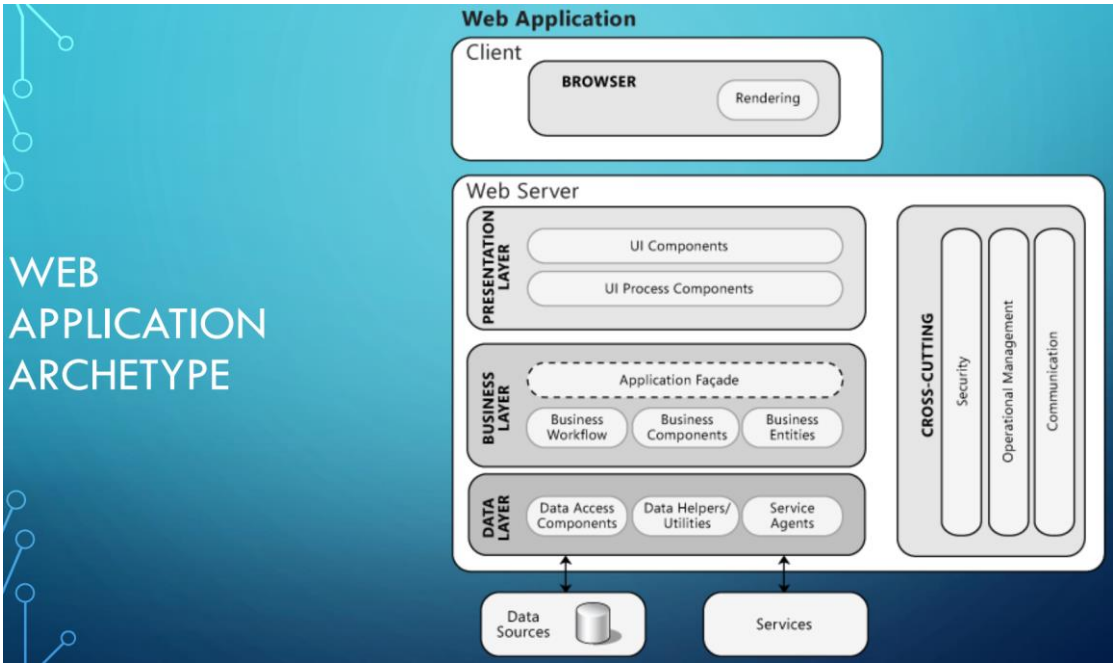
Disadvantages of Command-Query Responsibility Segregation (CQRS) & event sourcing:

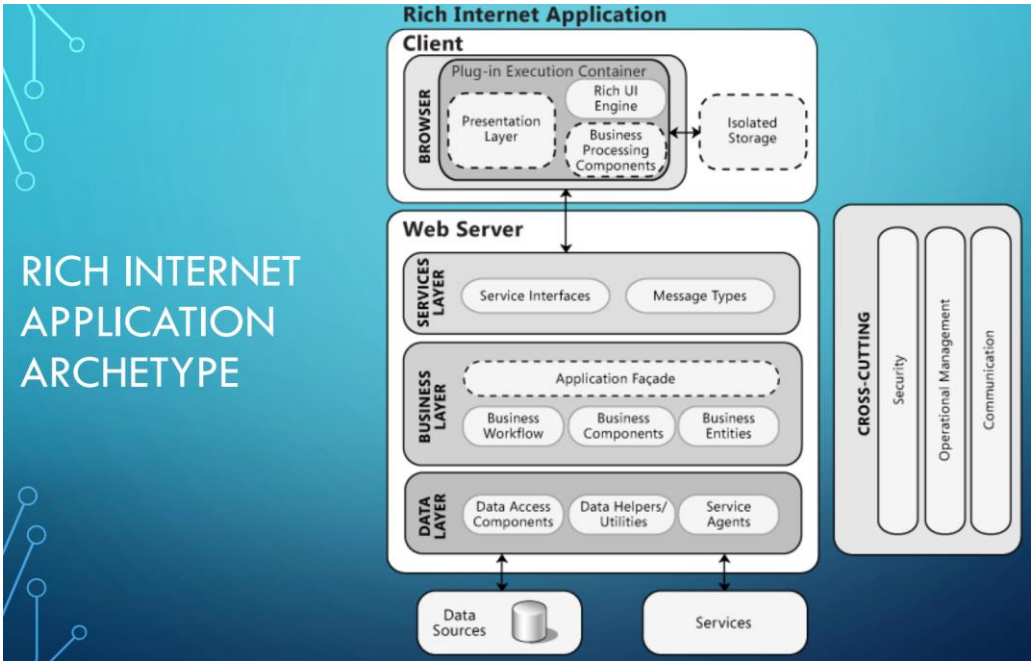
- Complexity of elements required
- High demand on storage
- Eventual consistency

(30. Archetypes) COMMON SOFTWARE ARCHETYPES

- Web applications
- Mobile applications
- Rich client applications (Desktop applications)
- Rich internet applications (Rich media multiplatform client side browser applications)
- Service applications
- Hosted and cloud applications
- Office business applications

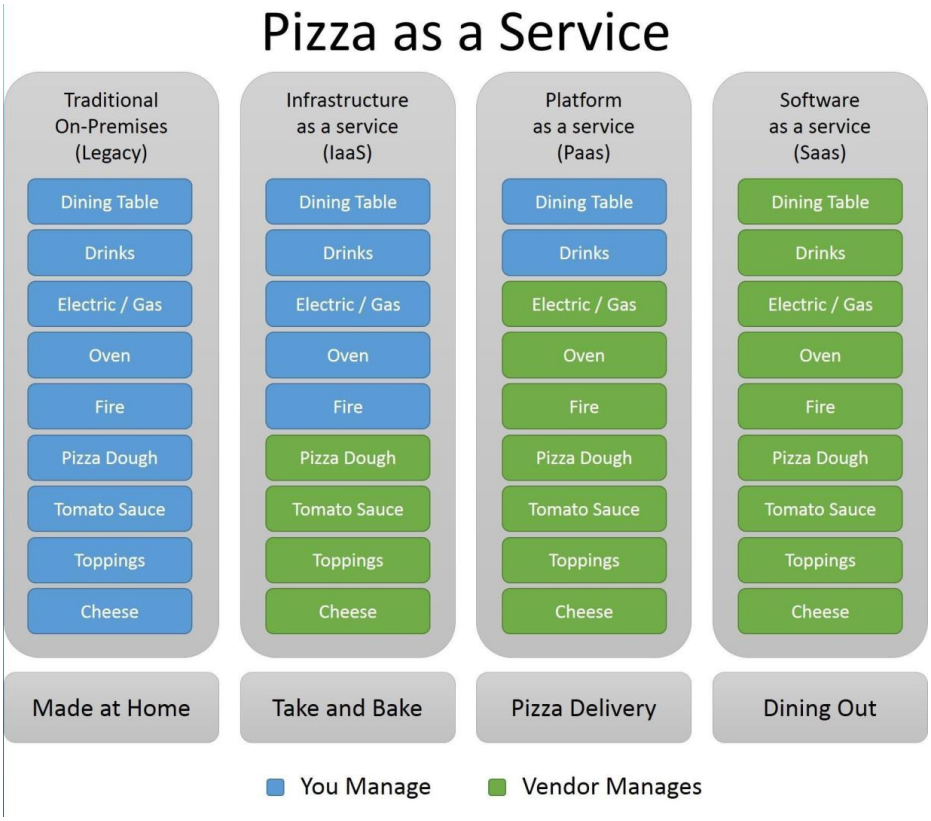




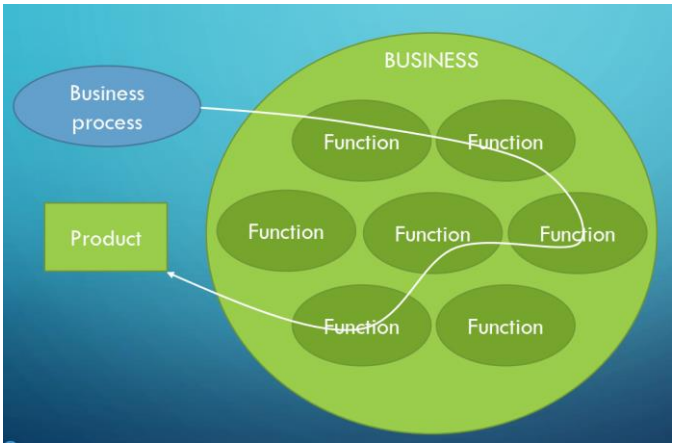


(30. Archetypes) CLOUD SERVICES

- **Infrastructure as a Service (IaaS):**
  Infrastructure as a Service contains the basic building blocks for cloud IT and typically provide access to networking features, computers (virtual or on dedicated hardware), and data storage space. IAAS provides highest flexibility
- **Platform as a Service (PaaS):**
  Platforms as a service remove the need for organizations to manage the underlying infrastructure (usually hardware and operating systems) and allow to focus on the deployment and management of applications. Simplifies maintenance.
- **Software as a Service (SaaS):**
  Software as a Service provides a completed product that is run and managed by the service provider. With a SaaS offering no need to think about how the service is maintained or how the underlying infrastructure is managed, only how you will use that particular piece software.

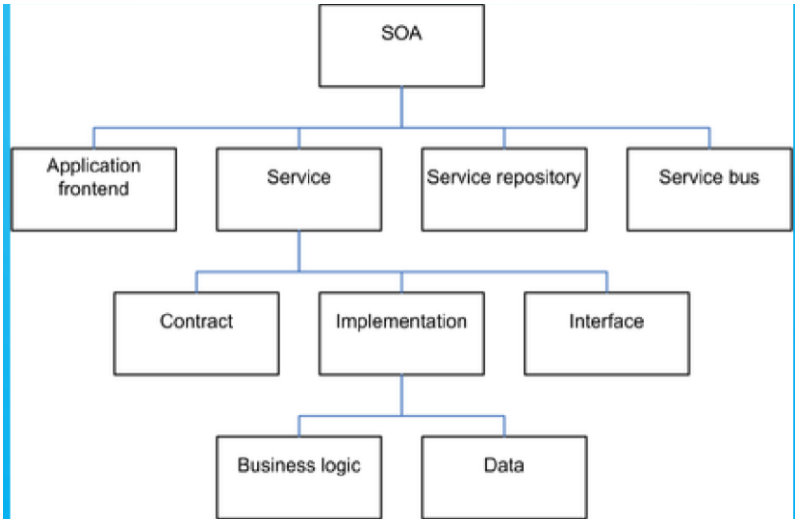


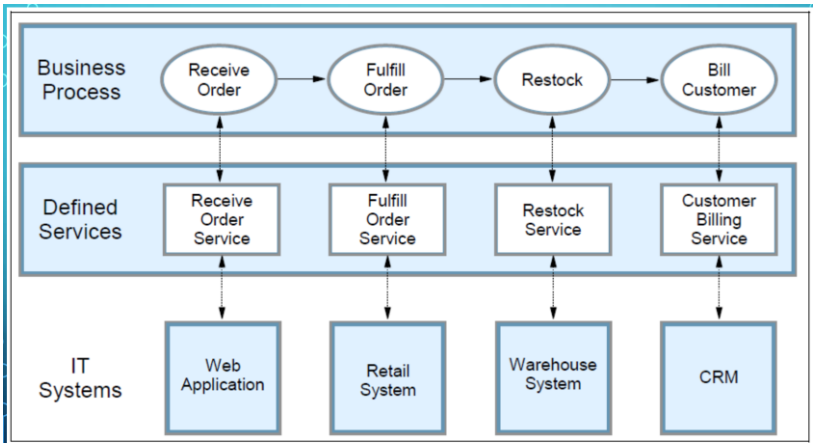
(31. SOA) BUSINESS PROCESS VS BUSINESS FUNCTION



SOA IS BRIDGING THE GAP BETWEEN BUSINESS AND IT. SOA SUPPORTS EVOLUTION AND REDUCTION IN COMPLEXITY

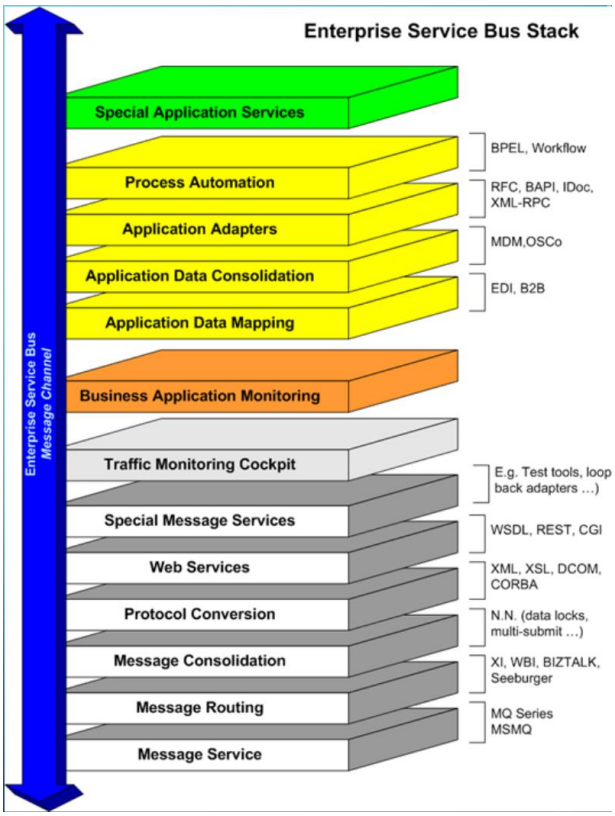
(31. SOA) SERVICE ORIENTED ARCHITECTURAL STYLE



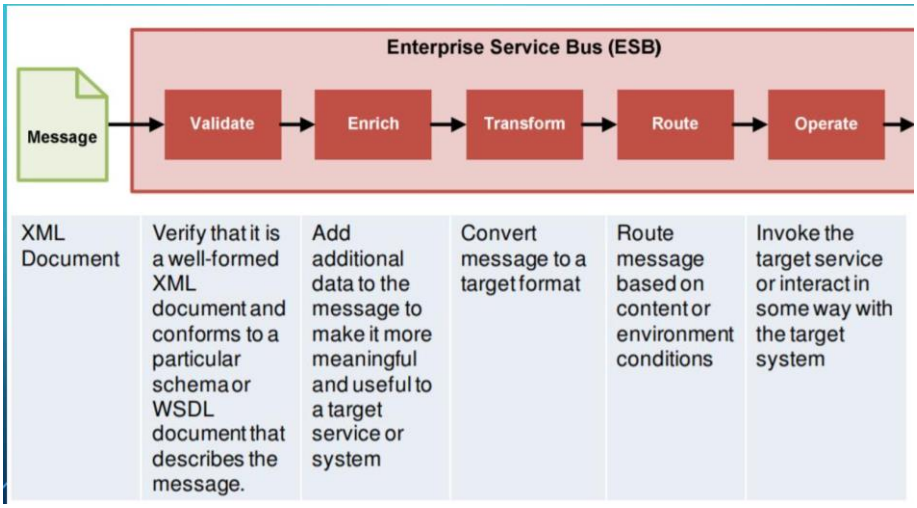


(31. SOA) ESB – ENTERPRISE SERVICE BUS

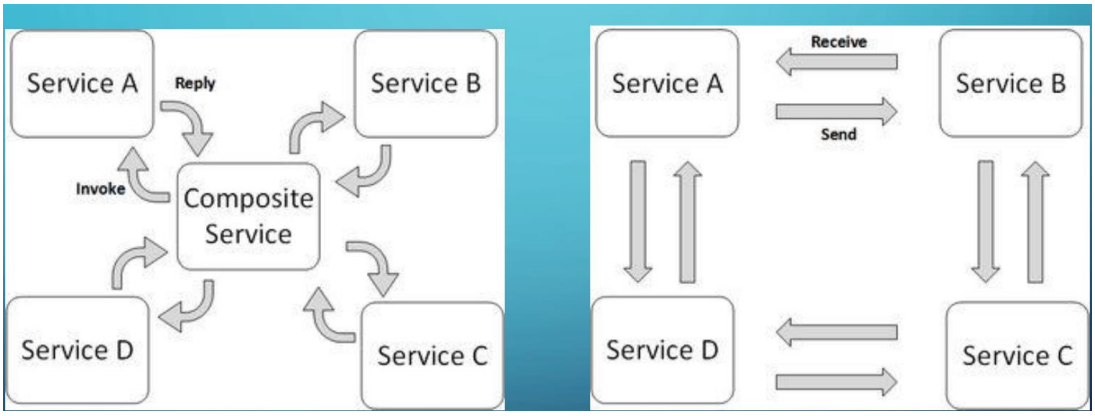
The ESB is implemented in software that operates between the business applications, and enables communication among them. Ideally, the ESB should be able to replace all direct contact with the applications on the bus, so that all communication takes place via the ESB.



(31. SOA) VETRO PATTERN



(31. SOA) SERVICE ORCHESTRATION VS CHOREOGRAPHY



(31. SOA) PRINCIPLES OF SOA

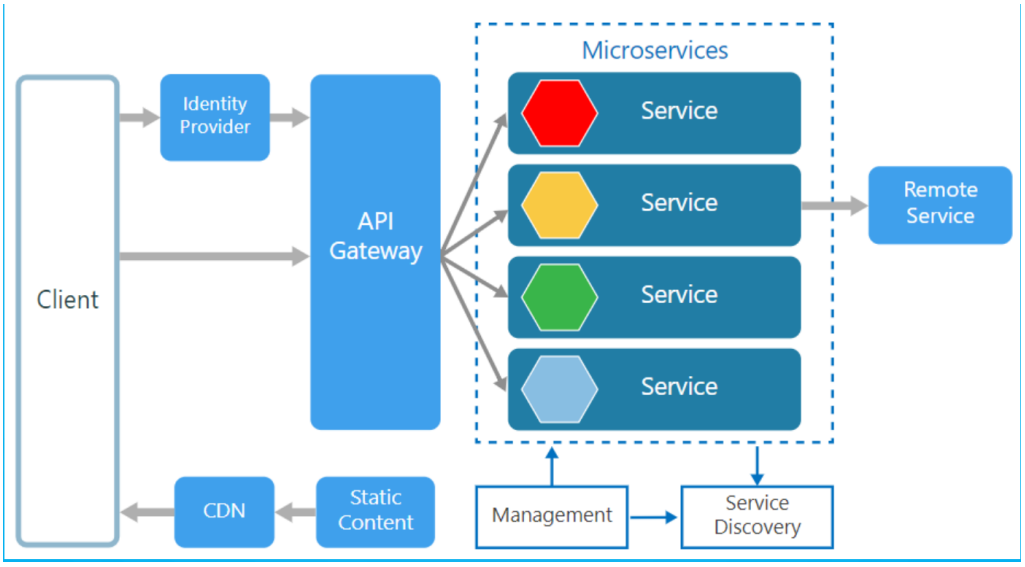
▪ **SOA is not web services**
▪ Standardized contract
▪ Service reference autonomy (only self-aware)
▪ Service location transparency
▪ Service longevity (long lived stable contracts)
▪ Service abstraction (black box)
▪ Service statelessness
▪ Service granularity (functionality must be relevant)
▪ Service composability
▪ Service discovery
▪ Service reusability

(31. SOA) WHY SOA?

• Support agility
• Built for change/flexibility
• Faster time to market
• Leverage of existing assets
• Reduced integration expense
• Complexity management

(32. Microservices) MICROSERVICES

Microservices architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

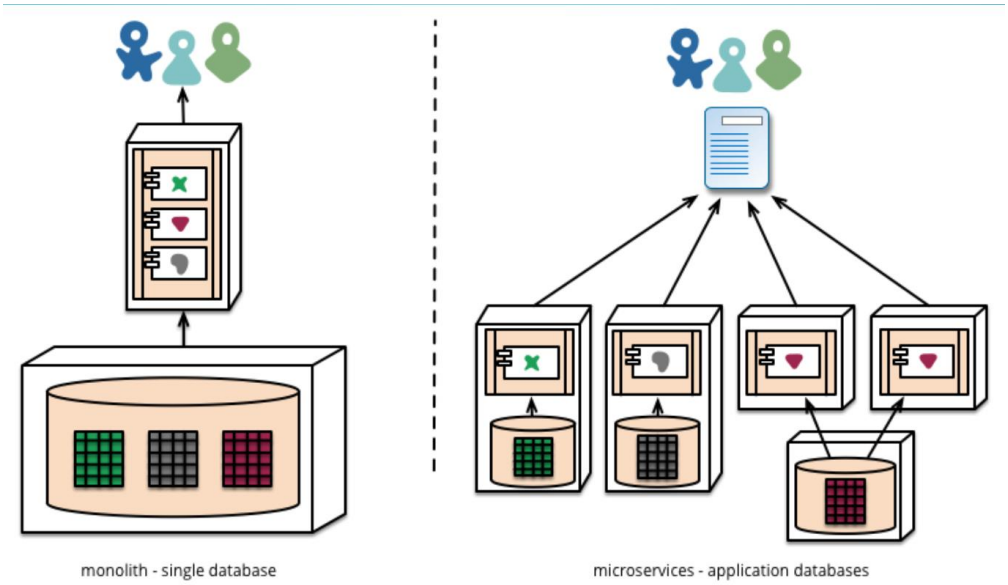


Management component is responsible for placing services on nodes, identifying failures, rebalancing services across nodes, etc.
Service Discovery maintains a list of services and which nodes they are located on. Enables service lookup to find the endpoint for a service.
API Gateway is the entry point for clients. Clients don't call services directly. Instead, they call the API gateway, which forwards the call to the appropriate services on the back end. The API gateway might aggregate the responses from several services and return the aggregated response.

(32. Microservices) PRINCIPLES OF MICROSERVICES:

- Full product ownership
- Focus around business capabilities
- Components by services
- End user interaction
- Full stack teams
- Work against Conway's law (no need for communication of teams inside organization)
  - https://en.wikipedia.org/wiki/Conway%27s_law
- Dumb pipes smart endpoints

(32. Microservices) HIDDEN & OWNED PERSISTENCE (Decentralized data management)



(32. Microservices) 5 ARCHITECTURAL CONSTRAINTS OF MICROSERVICES

- Elastic (scale well)
- Resilient (no impact to other services)
- Composable (defined and uniform API schema)
- Minimal (highly cohesive)
- Complete