

Architecture 3D Cube

A Short Description

Kim Horn

Version 1.14

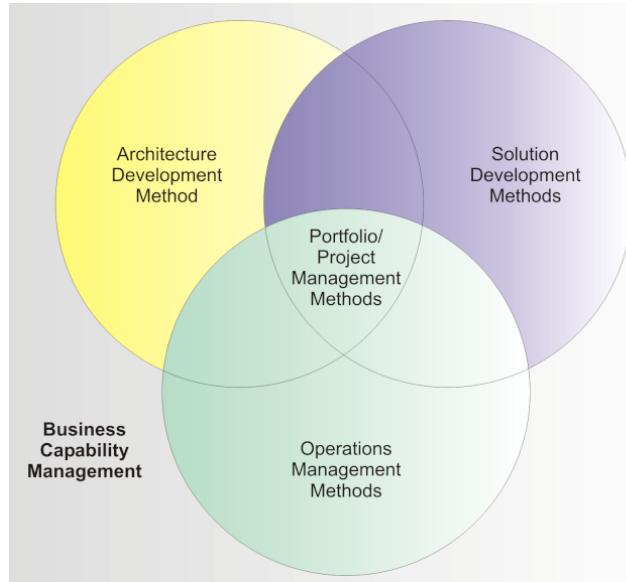
1/1/2017



Agenda



- Stakeholders and Views
- Prior Approaches
- The 3D Cube:
 - Tiers
 - Layers
 - Qualities
 - Stakeholder Views
- System Architecture Document
- Value of the Cube

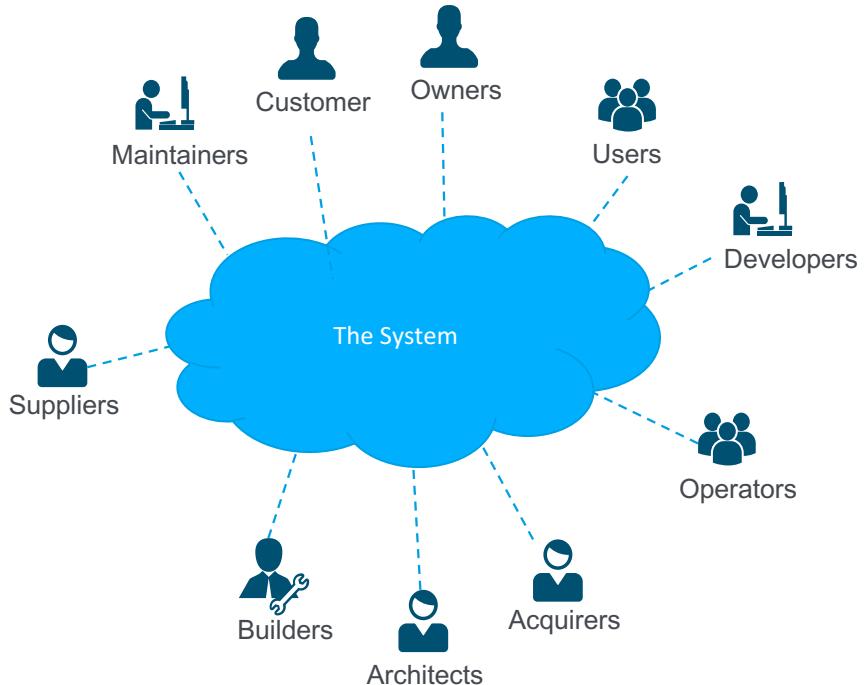
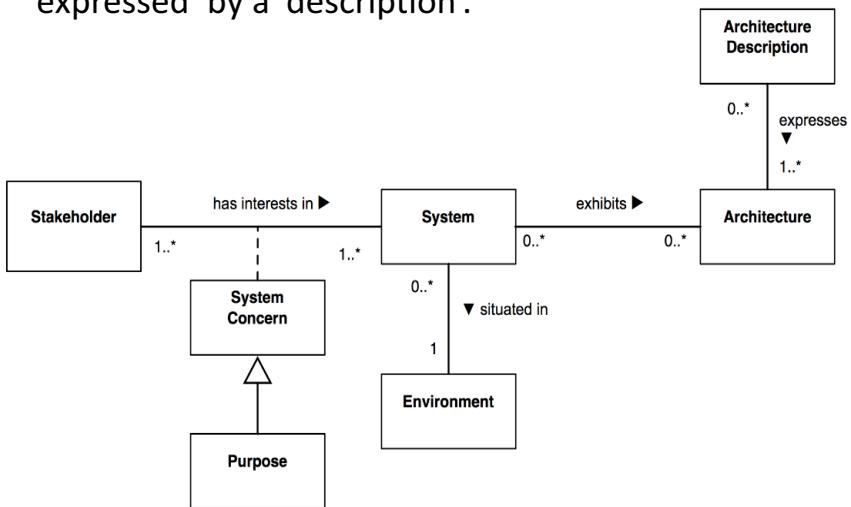


This is a short introduction, all aspects are detailed in further packs.

Stakeholders



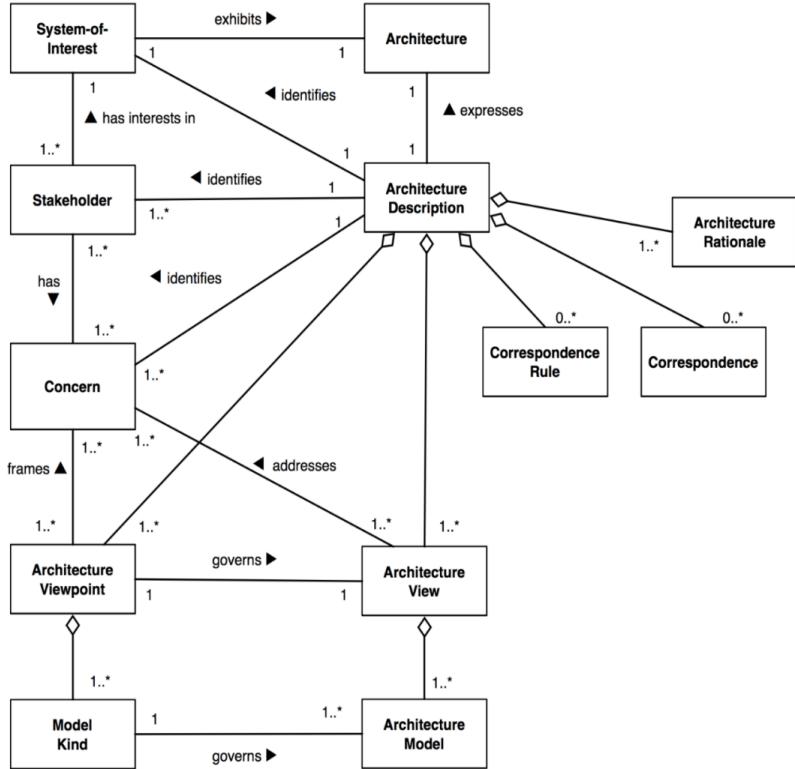
- A **stakeholder** is any person or group who have some vested interest or **concerns** about the **system**.
- Each Stakeholders has a different need or concern and views the system in a different way; they have points of view.
- A deployed system ‘exhibits’ an architecture ‘expressed’ by a ‘description’.



Architecture Description ISO Standard



System	Collection of components (business and technical) organized to accomplish a specific function or set of functions
Architecture	The system's fundamental organization, embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution
Architecture Description	Collection of artifacts that document an architecture – to meet the concerns of stakeholders
Stakeholder	Person (or group of people) who has a key role in, or concern about, the system
Concern	Key interests that are crucially important to the stakeholders in the system and determine the acceptability of the system
View	Representation of a whole system from the perspective of a related set of concerns
Viewpoint	Defines the perspective from which a view is taken: <ul style="list-style-type: none"> • How to construct and use a view • Information that should appear in the view • Modeling techniques for expressing and analysing the information • Rationale for these choice



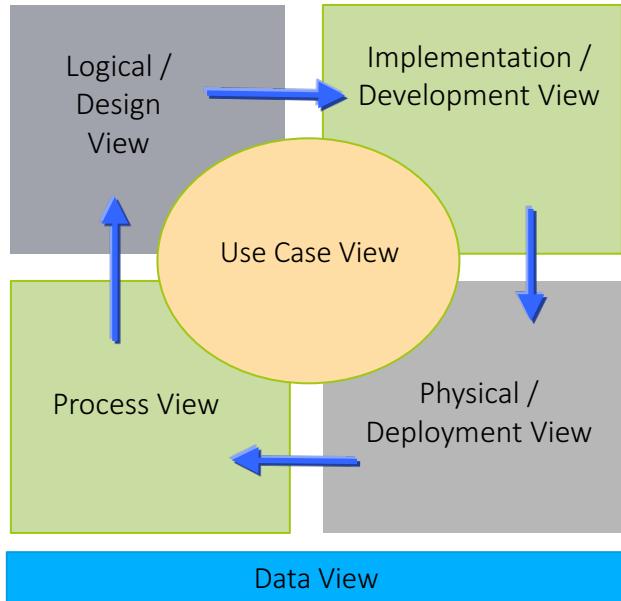
Ref: ISO/IEC/IEEE 42010 Systems and software engineering — Architecture description

Have been many different View / ViewPoint Models



8 of them are:

1. RUP 4+1 Views – Philip Kruchten, 1995 (IEEE Software);
2. Siemens Four – Soni, Nord & Hofmeister, 1995;
3. Business Component Factory – Herzum & Sims 1999;
4. SEI View Model – Clements et al, 2002;
5. Garland & Anthony 2003;
6. ISO RM-ODP - *ISO/IEC 10746*;
7. Microsoft DDD
8. Rozanski & Woods – 7 Viewpoints 2012;



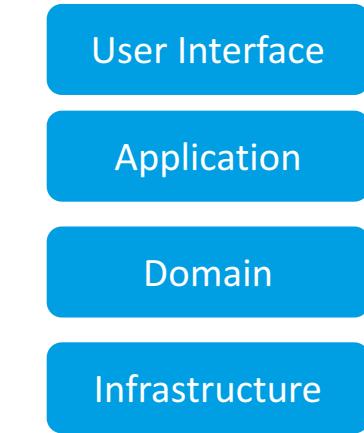
- An 'Ancient' Model from last century.
- Variety of naming schemes and many interpretations.....
- Issues:
 - What are viewpoints ?
 - Where are the services, platforms, infrastructure, etc ?
 - Where are the NFRs, that drive Architecture; not Functional Use Cases !!!!
 - Someone added a Data View ?
 - Out of date for SOA, Cloud...
 - So generic it provides little value
 - Where are the tiers/layers ?
 - Etc.....

A detailed document on issues with 4+1, comparing the Cube, is available on request.

Six Common Layering Models



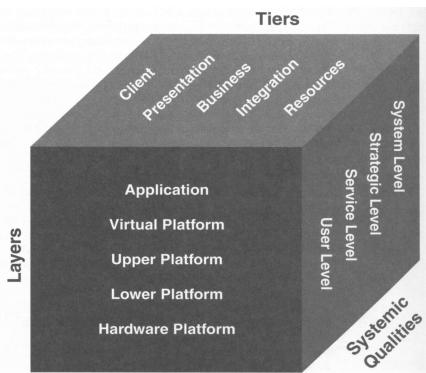
1. DDD – Eric Evans
2. Enterprise Java - Brown
3. Core J2EE – origin of Cube from Sun
4. Microsoft DNA
5. Microsoft DDD – updated DNA, based on Domain Driven Patterns
6. EJB Design - Marinescu
7. Fowler
8. .Net Enterprise Design - Nilson



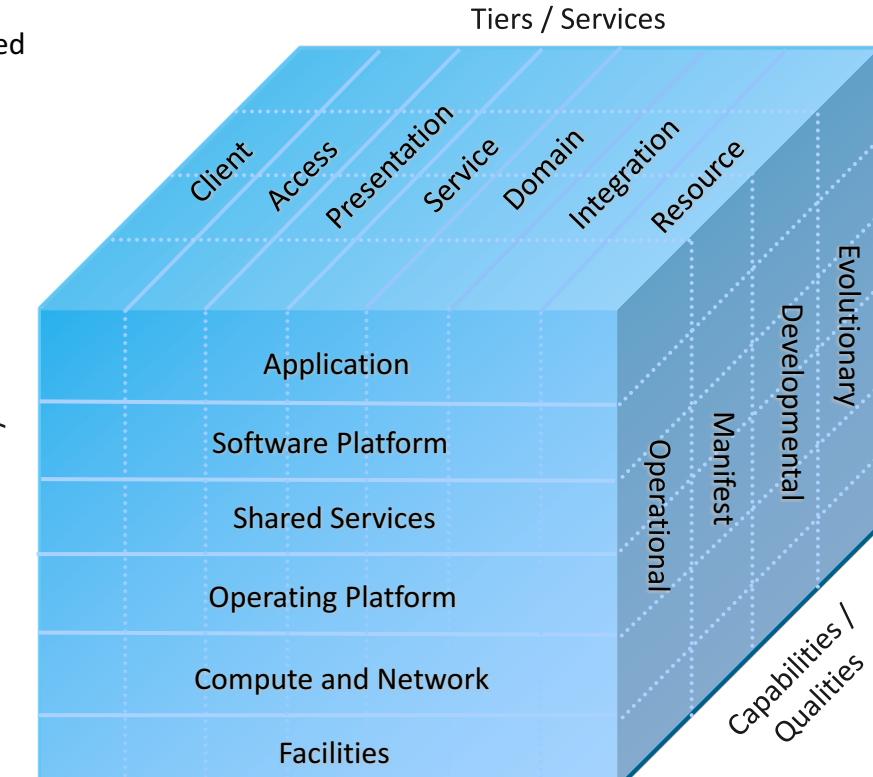


The 3D Cube

- A 3-D Conceptual Partitioning of an N-Tier or Service Based Composite System.
- Provides a well defined context for views and viewpoints.
- Originally published in 2001 by Sun.
- Things have changed, and improved:
 - Cloud, Containers, DevOps;
 - Evolutionary Architecture;
 - Microservices, API, Lambda;
 - Polyglot(Java, .Net, Ruby...)



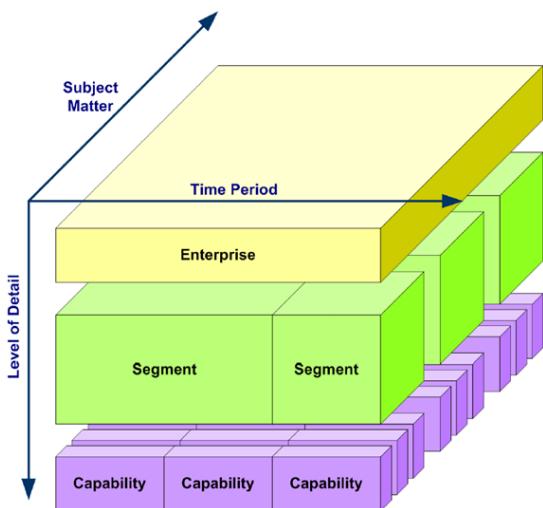
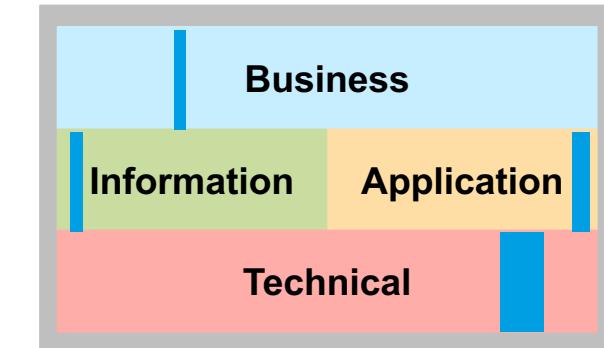
Old (2000)



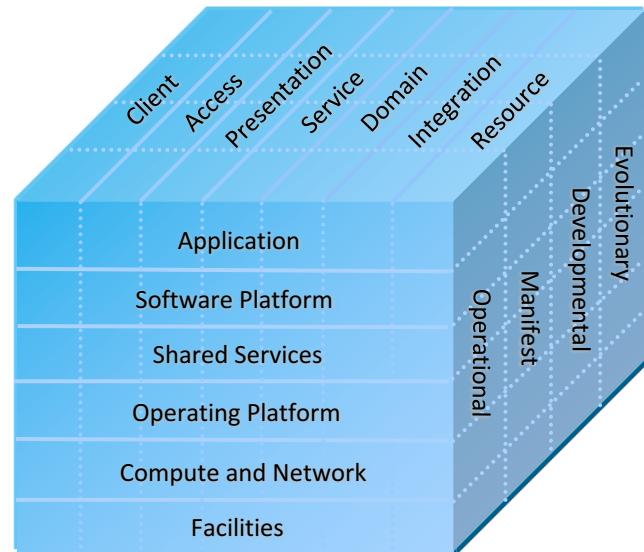
New (2017)



The Cube and an Enterprise Architecture



A cube provides a capability; a thin slice across each of the 4 enterprise architecture domains





The Tiers

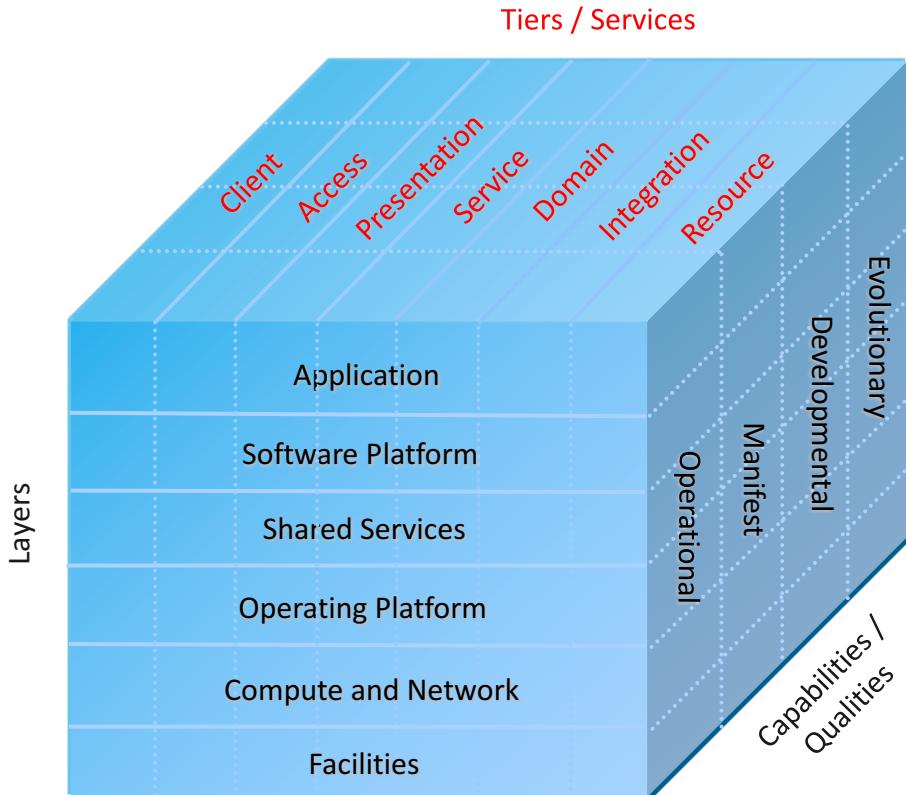


The Tiers

A tier is a *component* boundary that exists to provide isolation. The components provides a distributed or specific role, responsibility or service, supporting a business process.

The tiers:

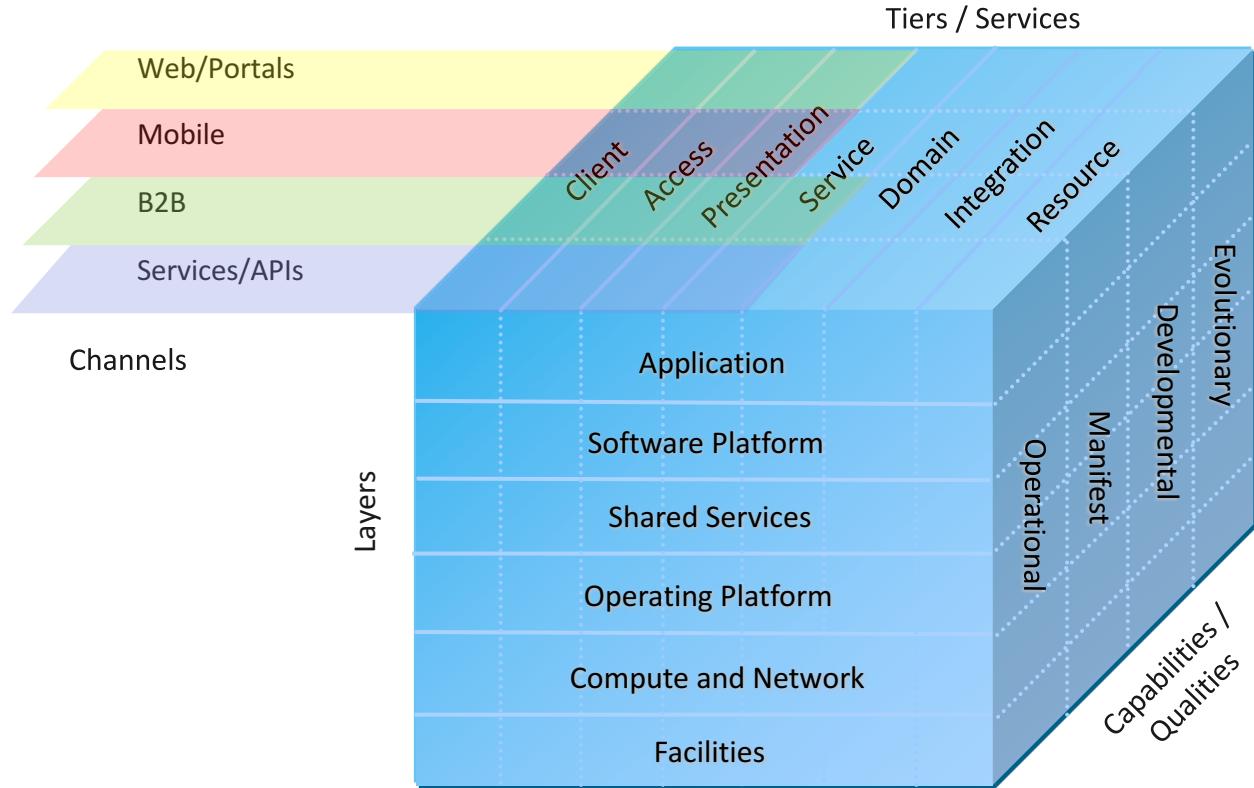
- applies the layering decomposition pattern to system structure;
- may not reside on the same physical box;
- may not be provided by the same technology and infrastructure;
- may be in different locations, distributed and reside on different devices;
- are a primary technique for building scalable systems that support continuous delivery and dev ops;
- can be layered; where *Layers* are logical slices that carry out specific responsibilities in an application, e.g. Services Tier is most often further decomposed.





Channels Reach Into Tiers through Services

Clients (users) via various channels, access the organisation's services.

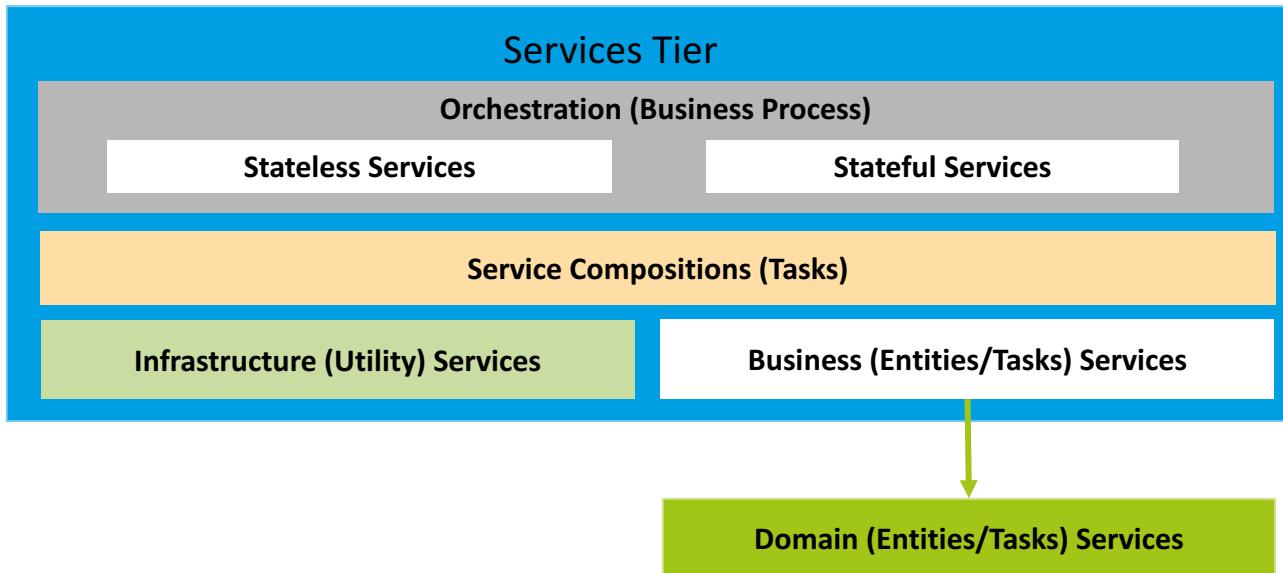




Service Layering

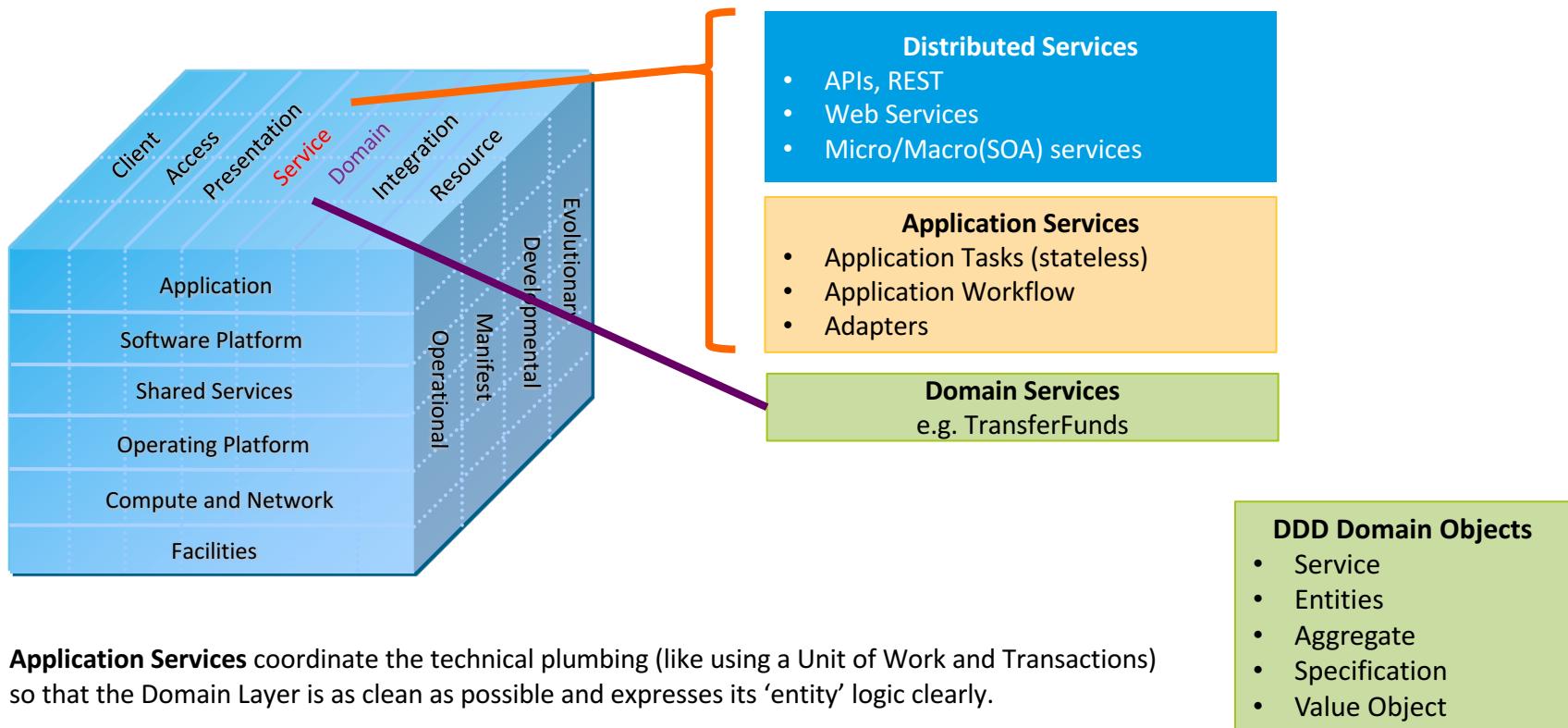
OASIS defines service as:

“A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.”



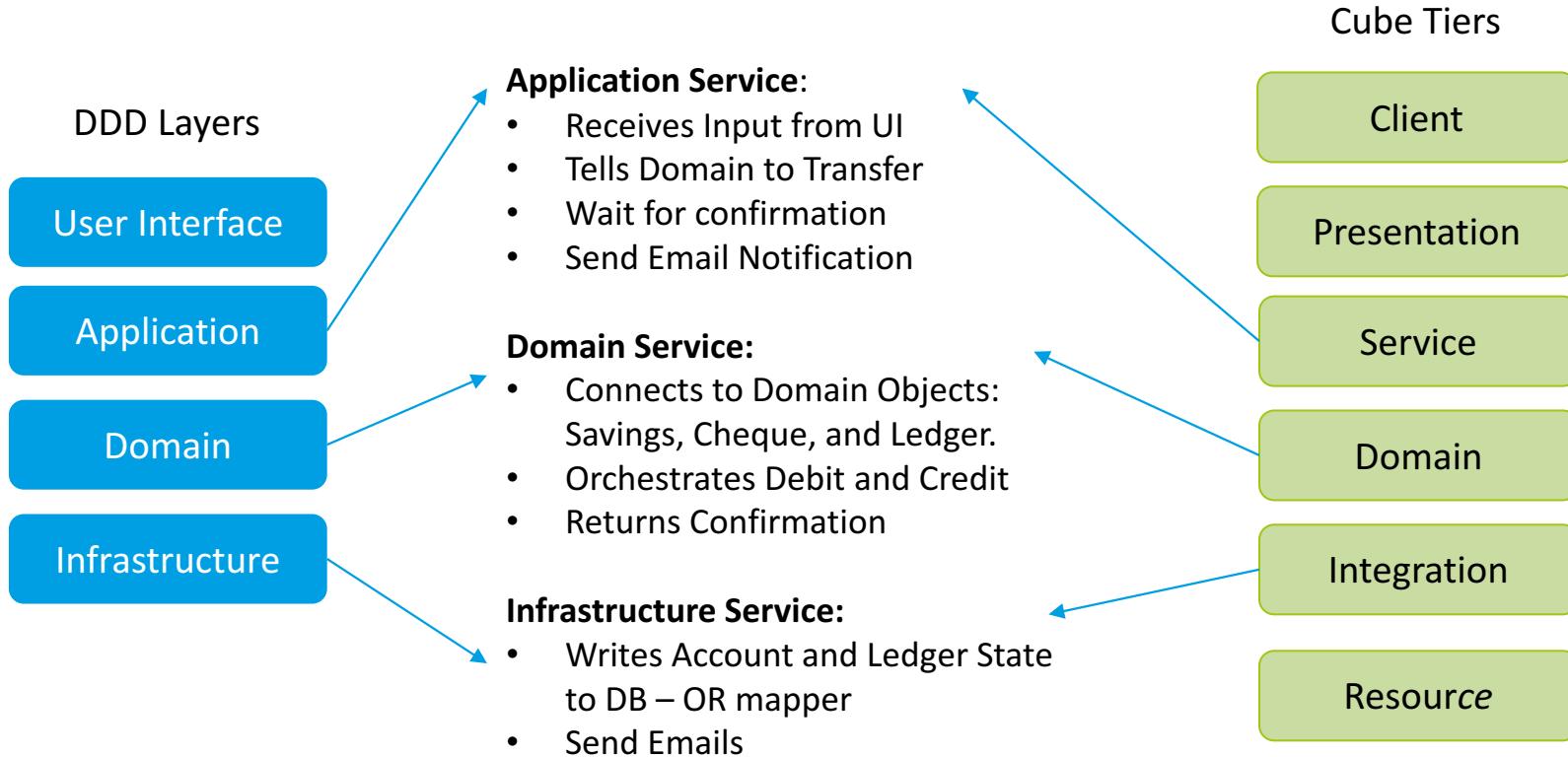


Service Layer Services Vs Domain Layer Services



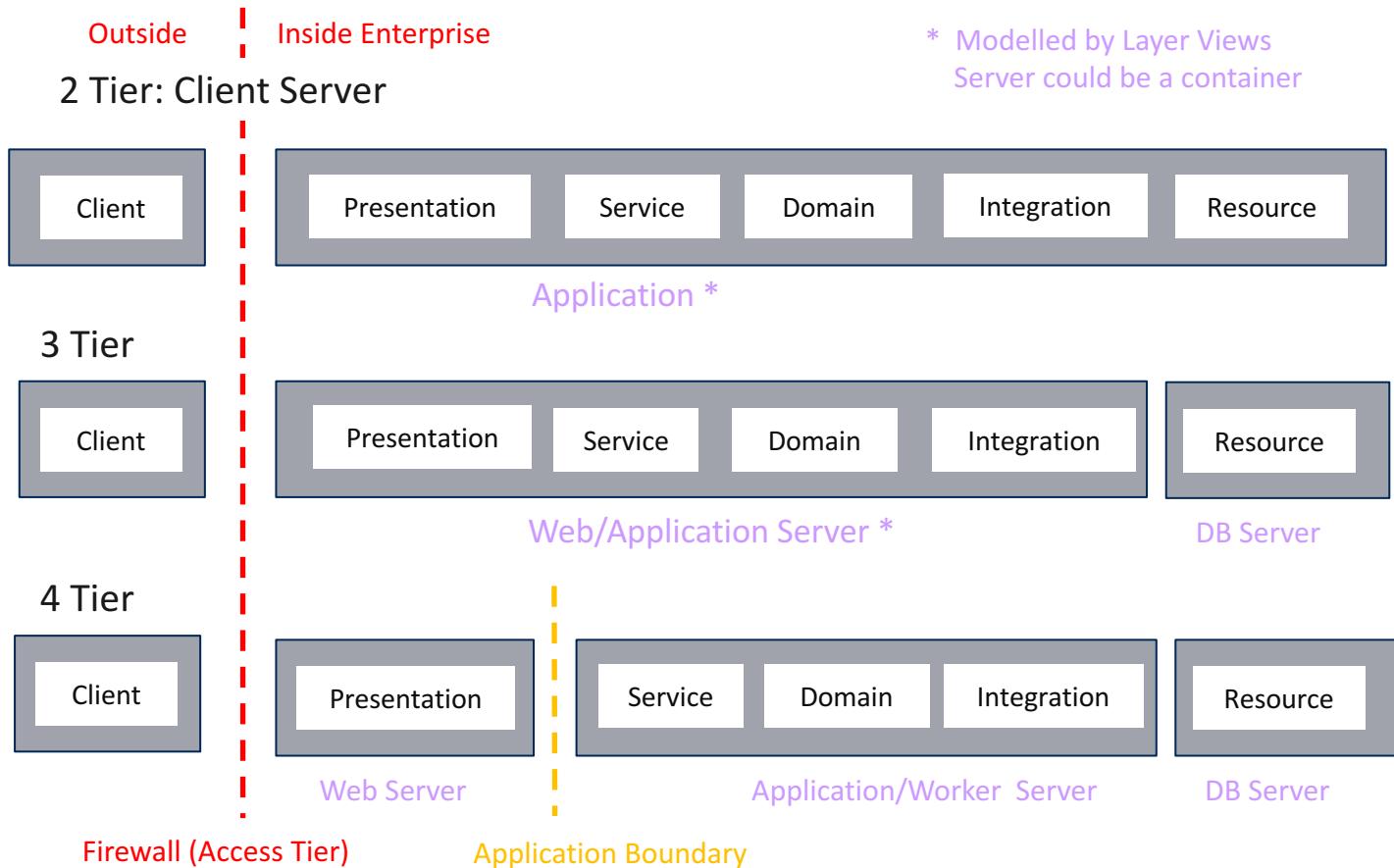
Application Services coordinate the technical plumbing (like using a Unit of Work and Transactions) so that the Domain Layer is as clean as possible and expresses its 'entity' logic clearly.

DDD Example – Transfer Funds From Savings to Cheque Accounts

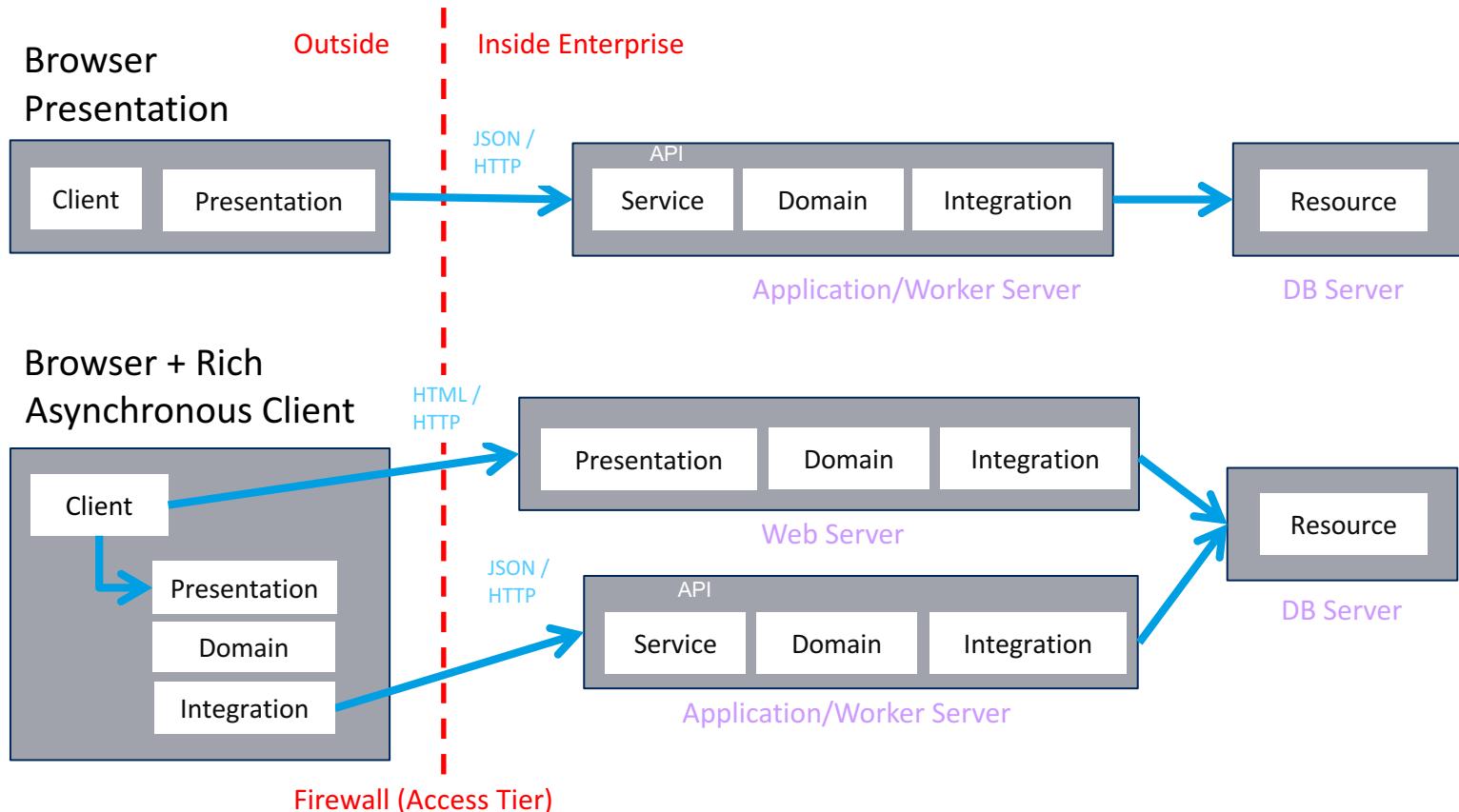




Physical Vs Logical Tiering – Runtime Structure Decision



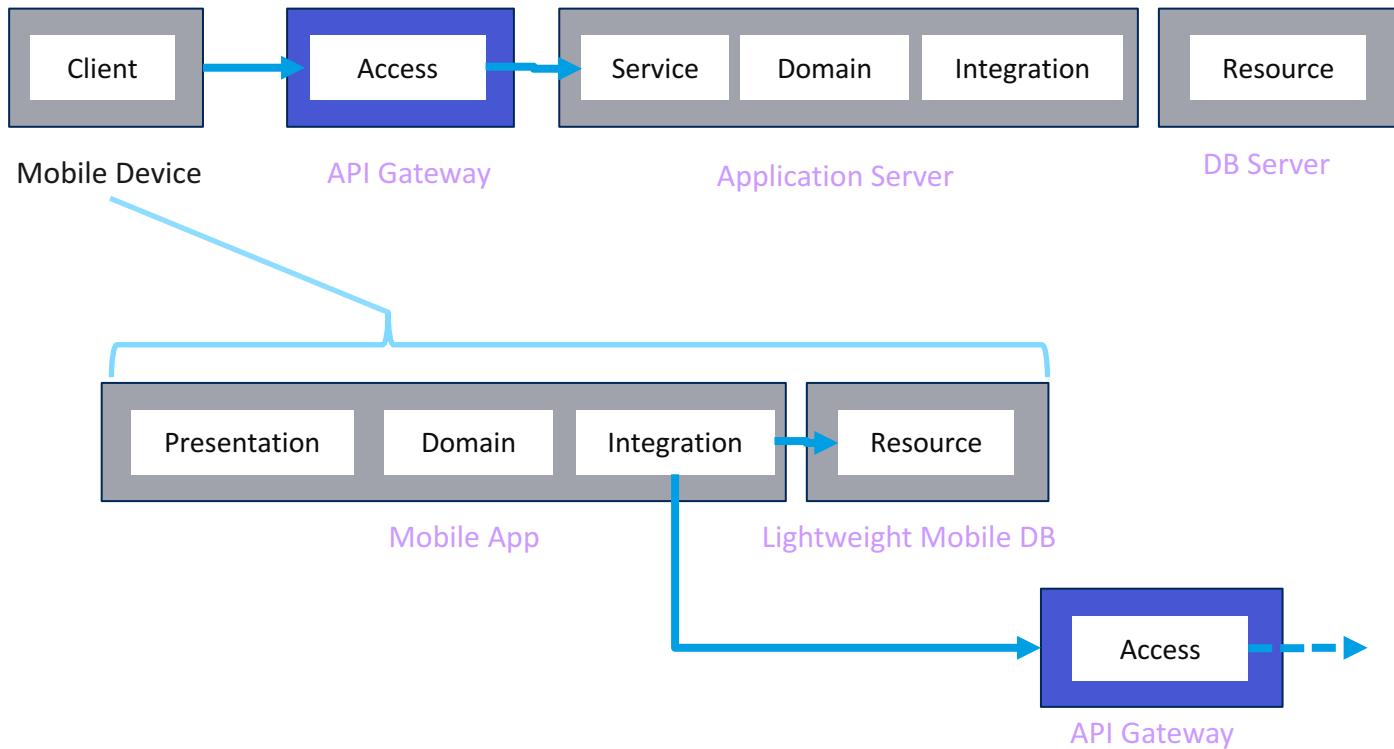
Physical Tiering – Example: Web Rich Client / API Model



Gateway Example: Mobile Device via API to Business



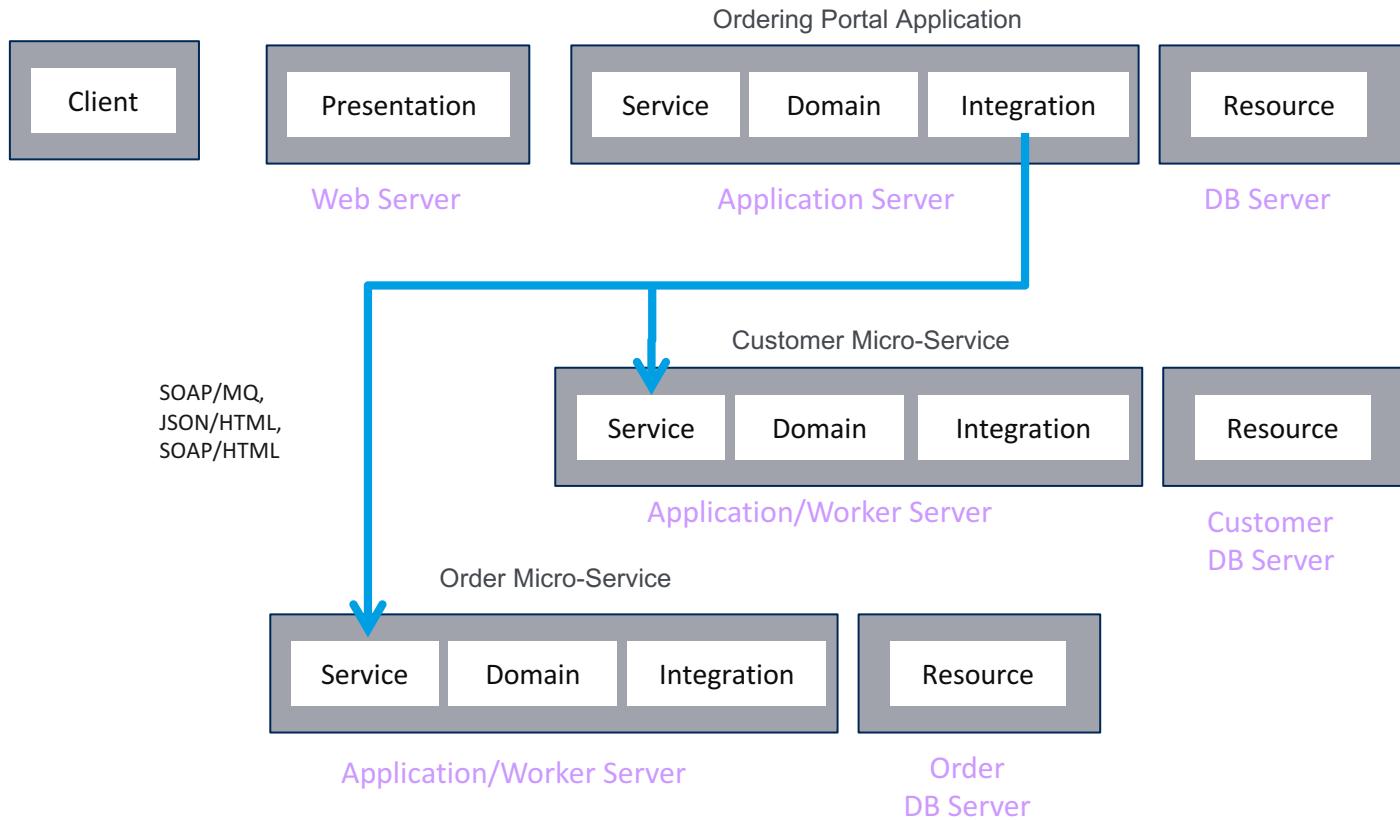
Example of API Gateway Pattern





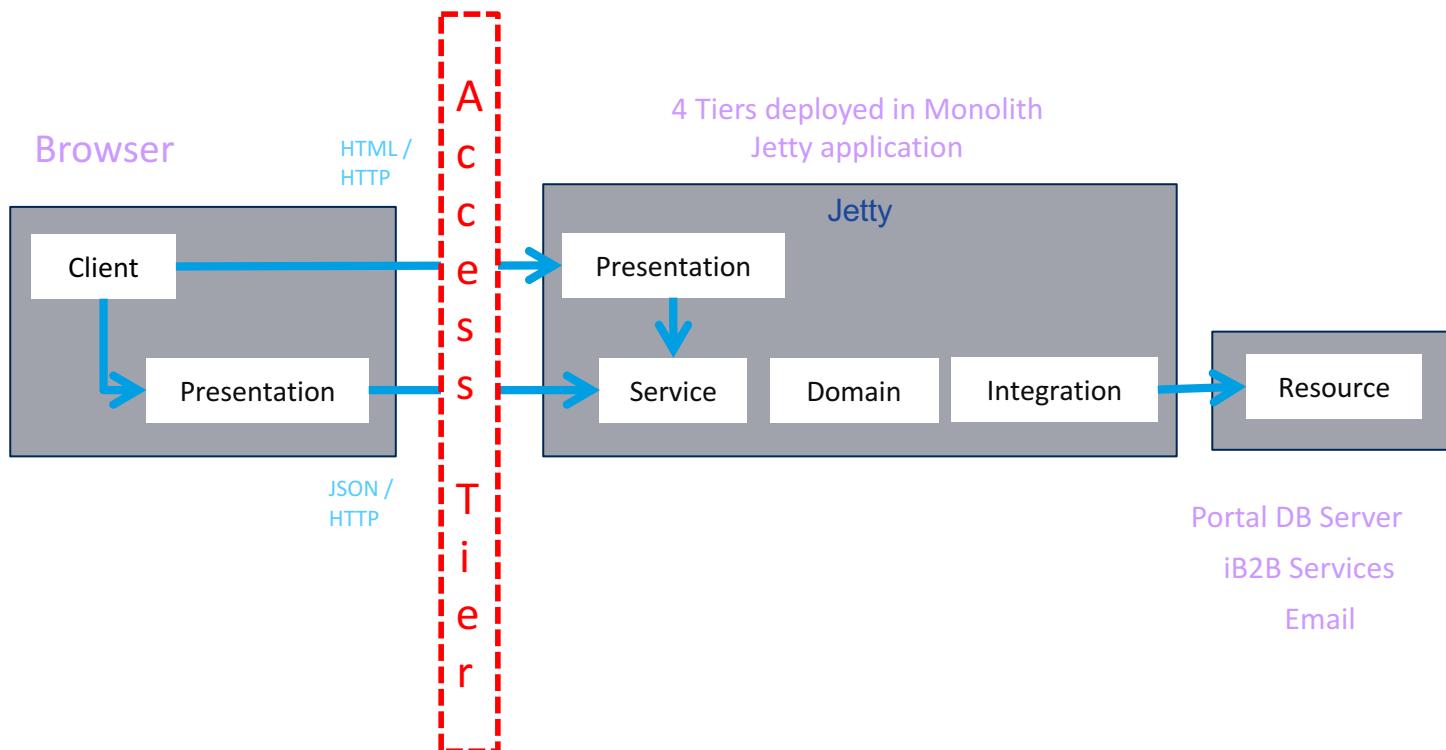
Composite Example: SOA / Microservices Model

Example of Composite Microservice Pattern





Example: Ordering Portal Tiers and Deployment



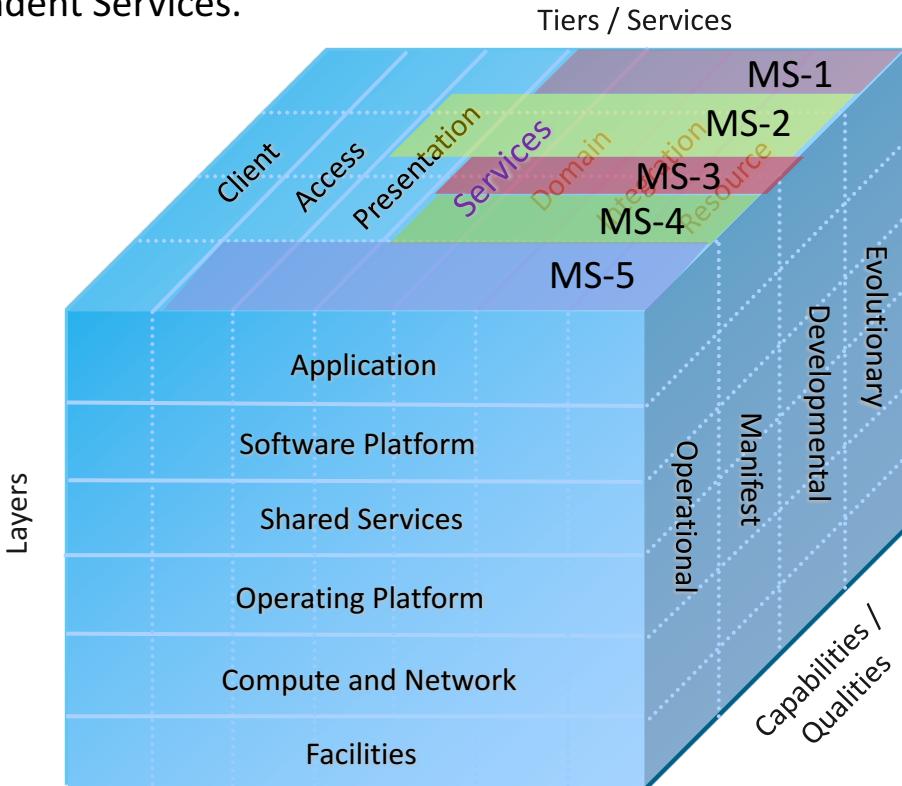
Examples: Global Site Selector, WAF, Load Balancers, Apache Proxy, WebGate, API Gateway

Microservices / Self Contained System - Slice the Tiers

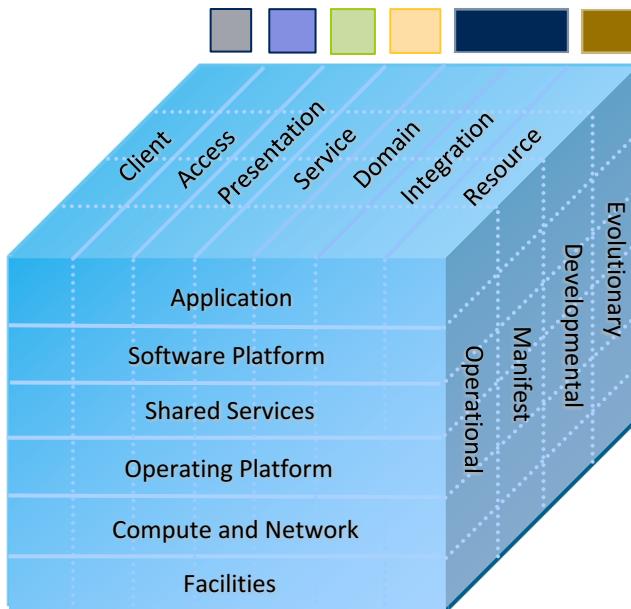
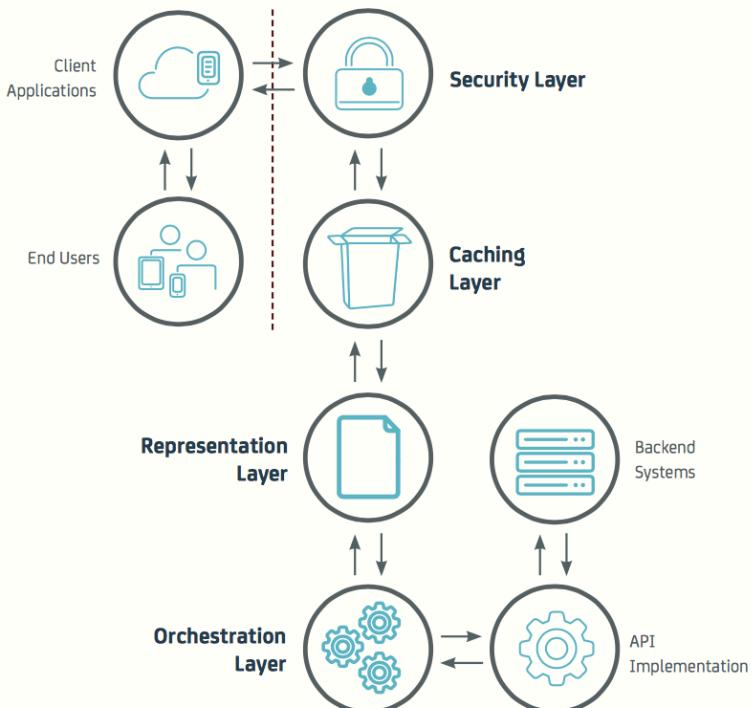


Decompose, slice, the domain into independent Services.

- May separate a Monolith into separate Bounded Contexts each provided by an independent system/service;
- Two approaches:
 1. **'Self Contained System' (SCS)** is a vertical slice of a Monolith that is autonomous, it has its own UI, business logic and data storage.
 2. **'Microservices'** also cut across the Tiers for a specific part of the business domain; however unlike and SCS they share tiers/services;
 - 'API Gateway' provides Access via boundary external/internal or domain/subdomain;
 - Microservices may provide their own Presentation, e.g. Protobuf/ JSON/HTML/XML.....



CAs Layered API Architecture - To Cube / Tiers





Infrastructure Layers

The Layers - Infrastructure

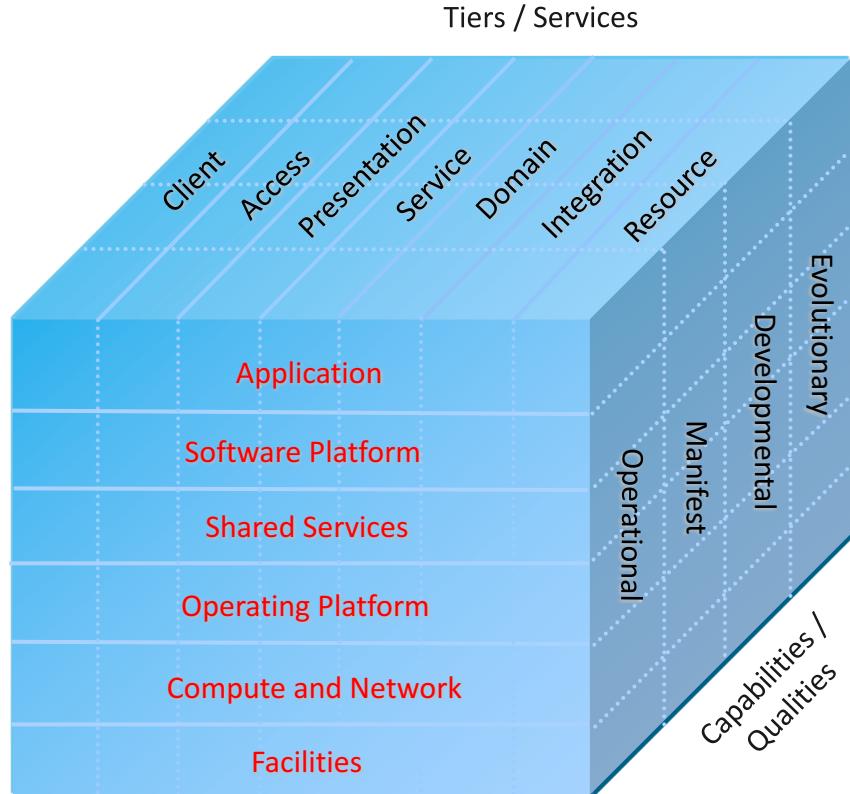


Infrastructure Layers Provide the Hardware and Software Stack at each Tier.

- Decouples the supporting ‘Infrastructure’, cross cutting concerns and mechanisms from the Tiers and Services, so the former are easier to model and understand.
- The Software Platform provides ‘Standards’ we build against, that implicitly require infrastructure.
- The Operating Platform abstracts away the Boxes and Wires (Compute and Network).
- Facilities are deployment locations, environments and physical access points.
- Provide Context for views

Platforms

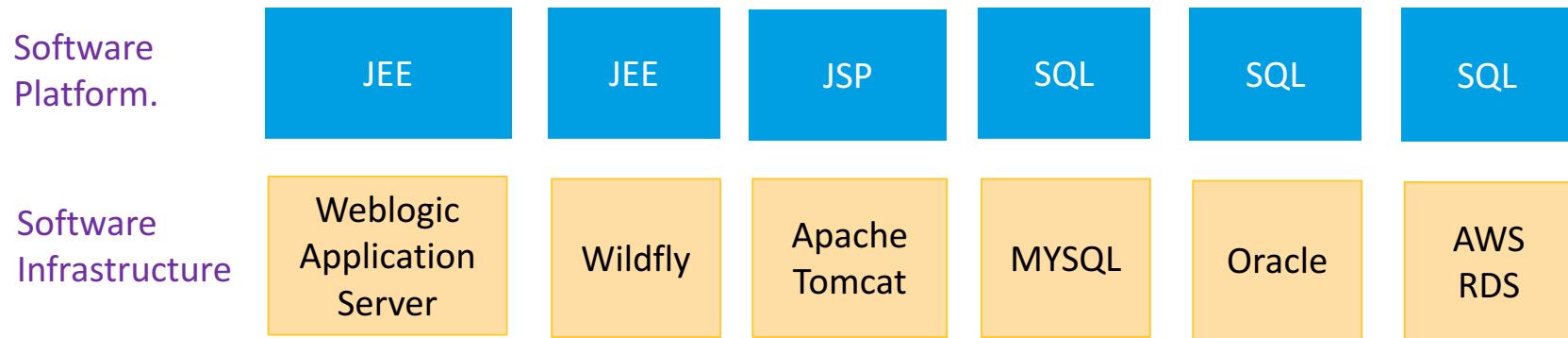
Infrastructure layers





Software Platform has implicit Software Infrastructure Layer

The **Software Platform** offers portability, exposing a software standard, such as J2EE, JEE or SQL, and provides for choice of implementation. **Software Infrastructure** layer implements it, and is vendor/provider specific. Example Application Servers include: Glassfish, JBOSS, Wildfly, Tomcat, TomcatEE, Geronimo, Jetty, Jonas, Resin etc. There are many SQL Server implementations.



Despite being a distinct layer, it is implicit in the cube, being so tightly coupled with the Software Platform. Often the Infrastructure and Platform are all provided and not standardised, for example NOSQL DBMSs, e.g. MONGO. The Infrastructure can also be provided in the layer below as a shared service like AWS RDS. ***When there is distinct infrastructure it should be described.***



Kafka Example

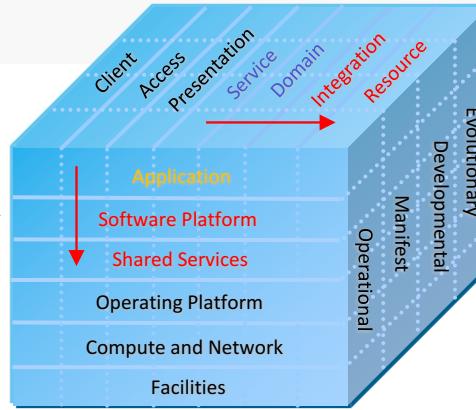
```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-stream</artifactId>
</dependency>

<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>

<dependency>
<groupId>postgresql</groupId>
<artifactId>postgresql</artifactId>
<version>9.1-901.jdbc4</version>
<dependency>
```

Java Streams
& Repository Standards

Kafka,
DBMS



App Service Processes Stream (Java Standard)
Software Platform hides Kafka

```
@SpringBootApplication
public class Application {

    @StreamListener(Sink.INPUT)
    public void statusSink(MessageModel messEvent) {
        logger.debug("Received an event Type: {} Status: {}",
                    messEvent.getOrderId(),
                    messEvent.getStatus());
        orderService.updateOrderStatus(messEvent.getOrderId(),
                                         messEvent.getStatus());
    }
}
```

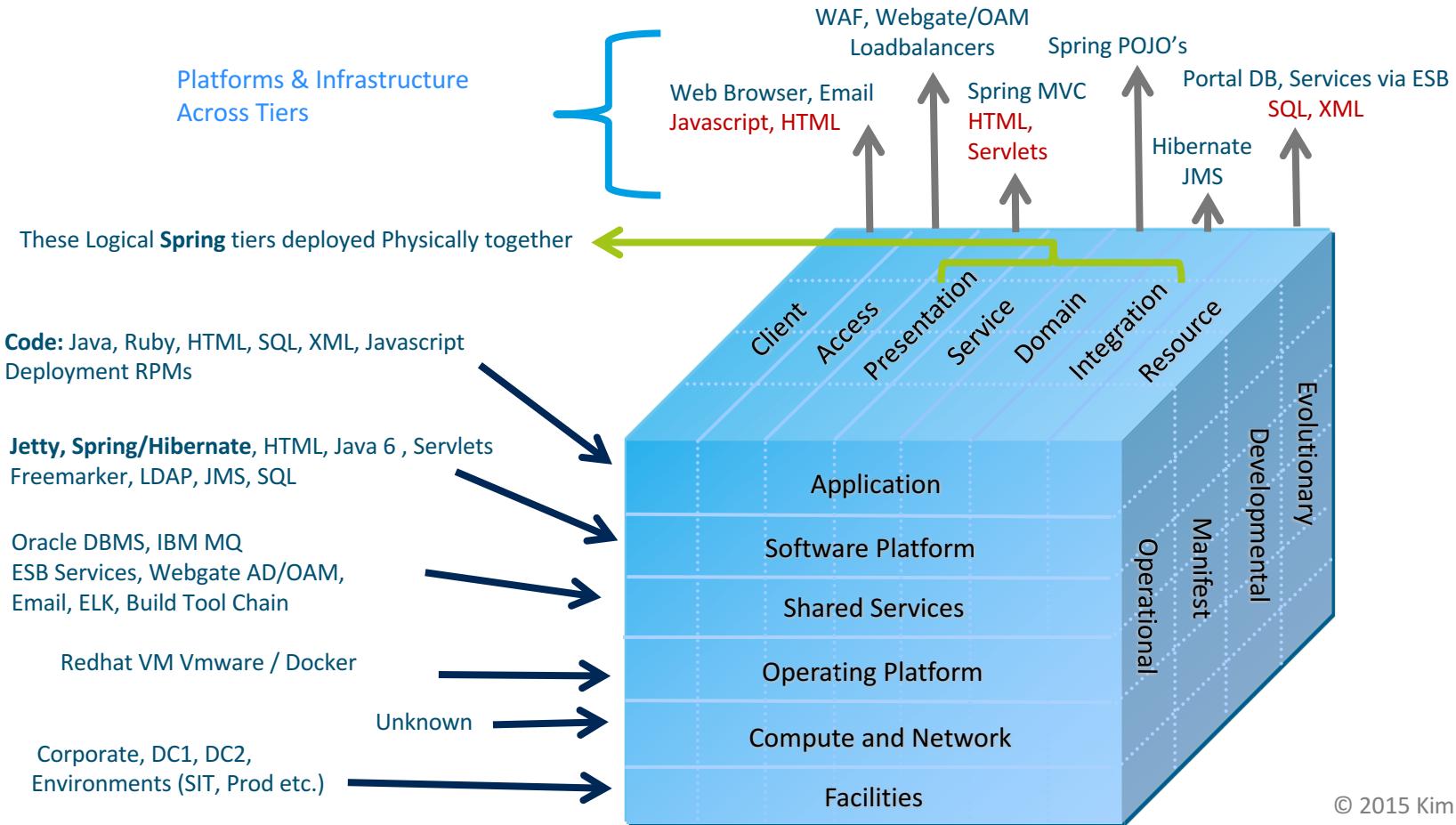
Service, Domain Entity, Repository, Resource

```
@Service
public class OrderService {

    @HystrixCommand
    public void updateOrderStatus(String orderId, String
status) {
        logger.debug("Getting the Order by OrderID");
        Order order = orderRepository.findById(orderId);
        order.setStatus(status);
        orderRepository.save(order);
        logger.debug("updated Order Status");
    }
}
```



Example: Product Ordering Web Site

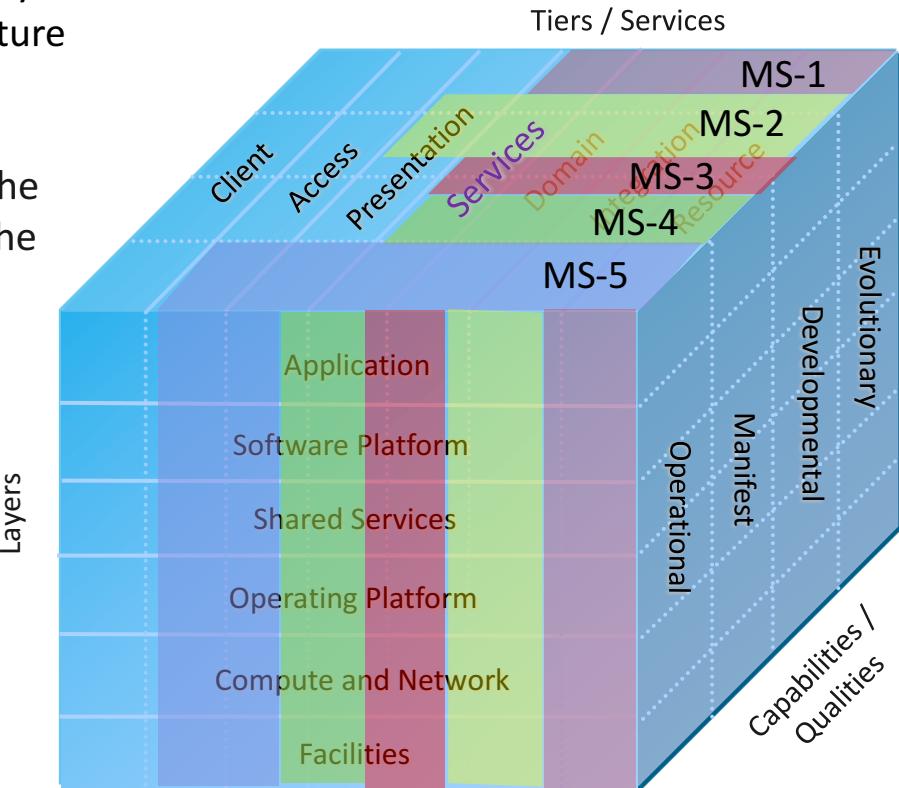




Microservices Slice the Tiers and the Layers

Microservices may be provided by different frameworks, infrastructure and platforms; polyglot.

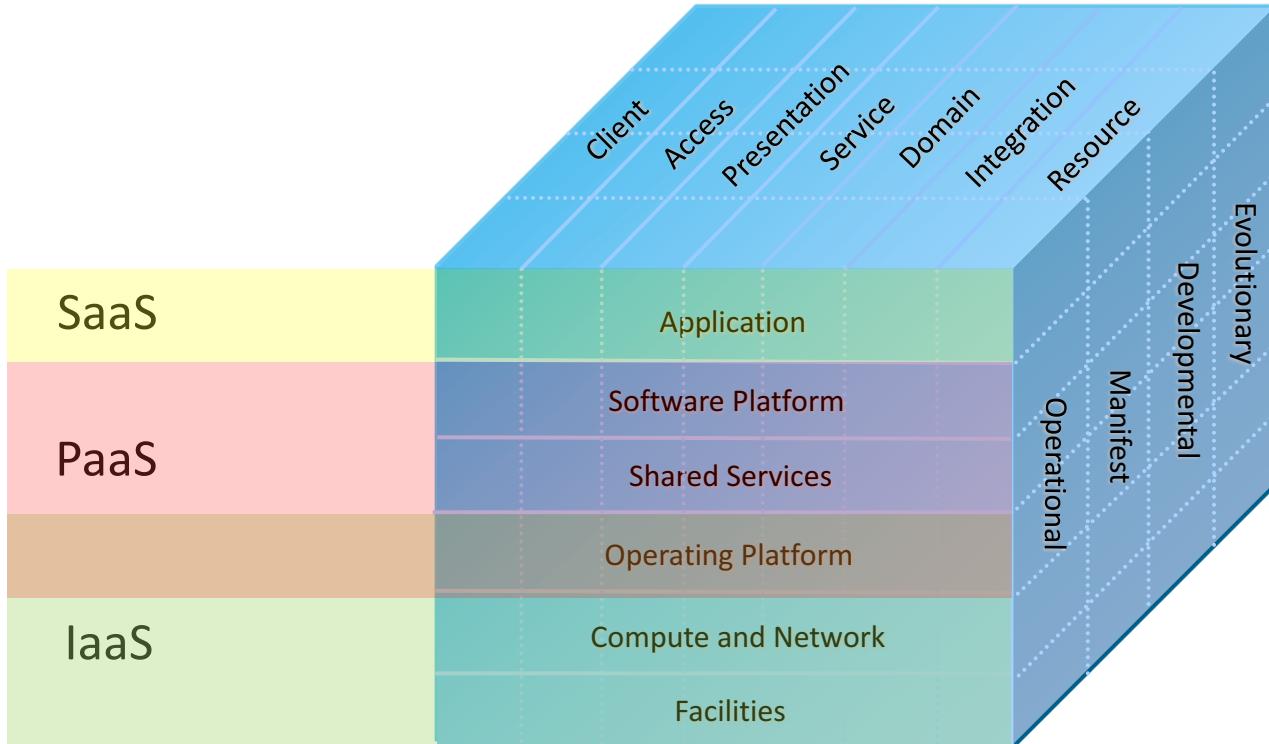
- Each Microservice may use different stacks; cut across the layers for a specific part of the business domain.





Layers 'As A Service'

The layers help model the use of cloud PaaS or IaaS components and the production of SaaS solutions. The point where PaaS and IaaS separate is not often clear.



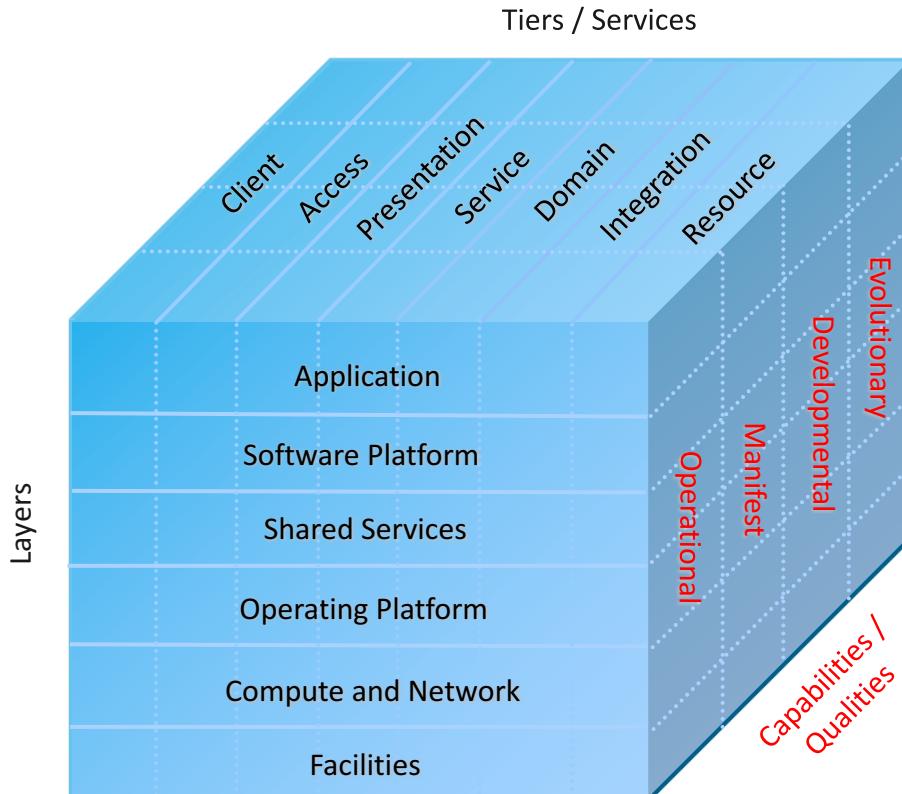


The Qualities

Architecture is primarily
Driven by System Qualities and
Capabilities.

Achieving the qualities determines
the success of the architecture.

- These are critical for Continuous and Evolutionary Approaches to Architecture;
- Operational Capabilities are key early on for Continuous Delivery/Integration;
- Record the tactics, decisions and justifications in meeting the goals with this solution.





Types of Quality

Type Of Quality	Who	When	How
Manifest	End user and business stakeholders	After initial release, production.	Test, customer satisfaction, business objectives, analytics.
Operational *	IT Operations, IT Service Management, IT Security, Business Security, Maintenance and Support.	Development operations, release-time, after release	Test, customer satisfaction, operational metrics.
Developmental *	Developers, Dev-Ops, PMs, Architects.	During and throughout development and delivery.	Productivity metrics, Velocity, delivery frequency.
Evolutionary	Developers, business stakeholders, Architects	After initial release	Strategy and Business Performance

* These are the Architecturally Significant drivers for Continuous Delivery (CD).
CD requires specific qualities to be addressed early.

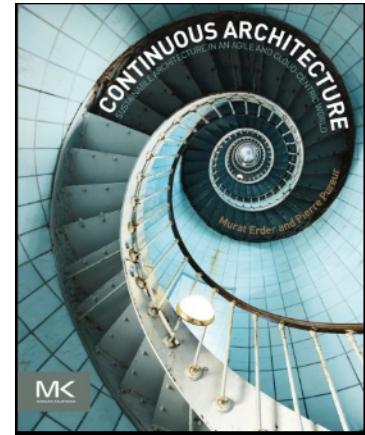


The Qualities

Continuous Architecture ‘Book’ Principles



1. Architect Products – not Projects;
2. Focus on **Quality Attributes** – not on Functional Requirements;
3. Delay design decisions until they are absolutely necessary;
4. Architect for Change – Leverage “The Power of Small”;
5. Architect for **Build, Test and Deploy**;
6. **Model the organization** of your teams after the design of the system you are working on;



Points 2, 5 and 6 are about Qualities.

by Pierre
Pureur, Murat Erder

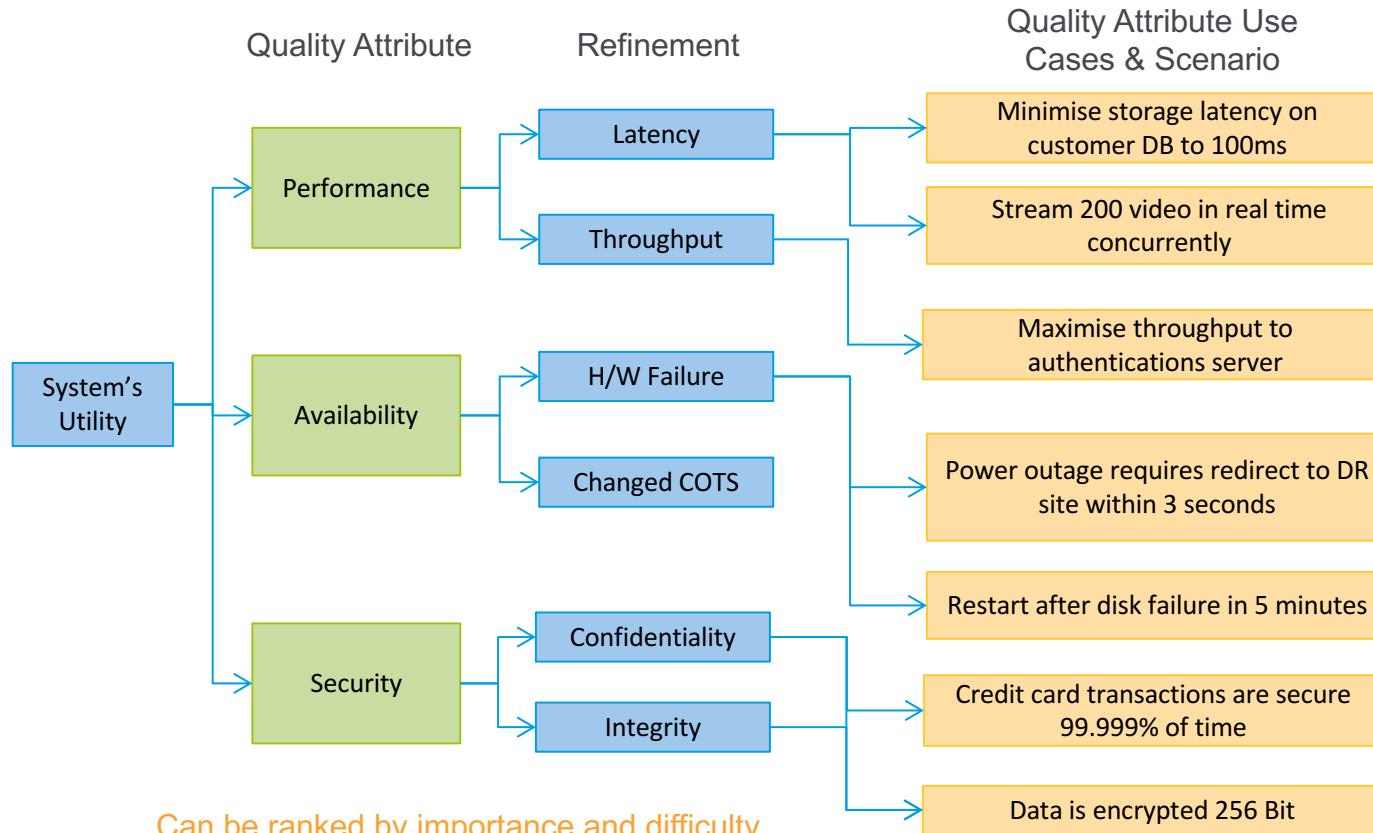


Qualities List by Type

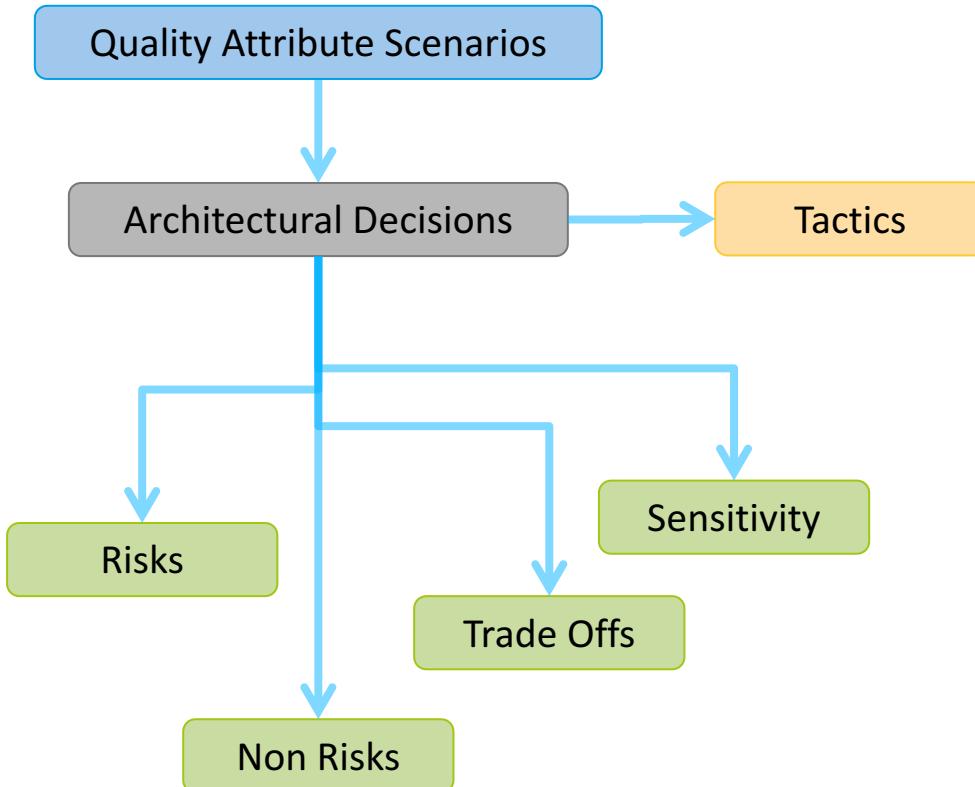
Key	Red = Continuous Architecture is driven by Continuous Delivery(CD)/Integration Focused Quality Drivers Purple = Immediately Impacted by CD Qualities.	
Manifest	Performance	
	Availability	
	Usability	
	Reliability	
	Accessibility	
	Mobility	
Developmental	Buildability	
	Testability	Red
	Understandability	
	Code Quality Measurability	Red
	Conceptual Integrity	
	Budgetability	
	Planability	
	Traceability	Red
	Development Distributability	Red
	Modularity / Packagability	Red
Operational	Throughput	
	Security	
	Manageability	
	Maintainability	
	Serviceability	
	Deployability	Red
	Reproducibility	Red
	Scalability / Elasticity	
	Variability	
	Flexibility	
Evolutionary	Extensibility	
	Reusability	
	Portability	
	Integratability	



Quality Tree – Capturing NFRs - Example,



Scenarios and Architectural Decisions



Scenarios result in multiple architectural decisions.

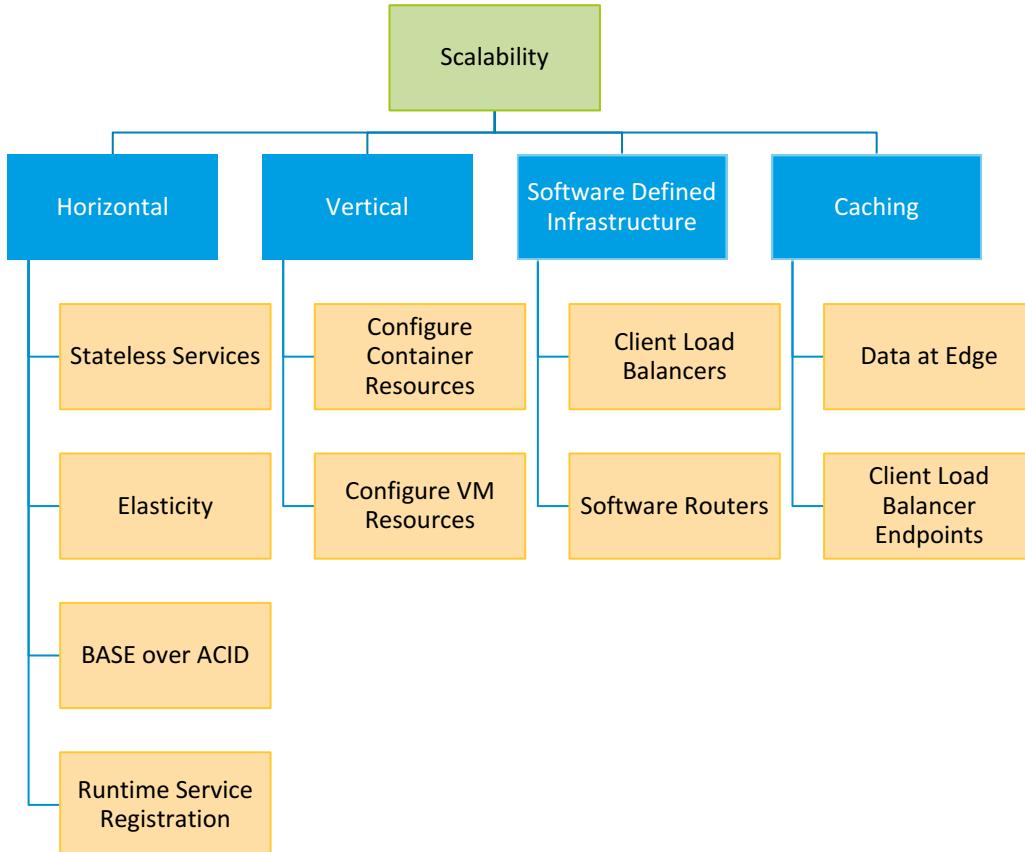
Architectural decisions involve tactics, one or more becomes the chosen solution.

- Need to Justify the choice.

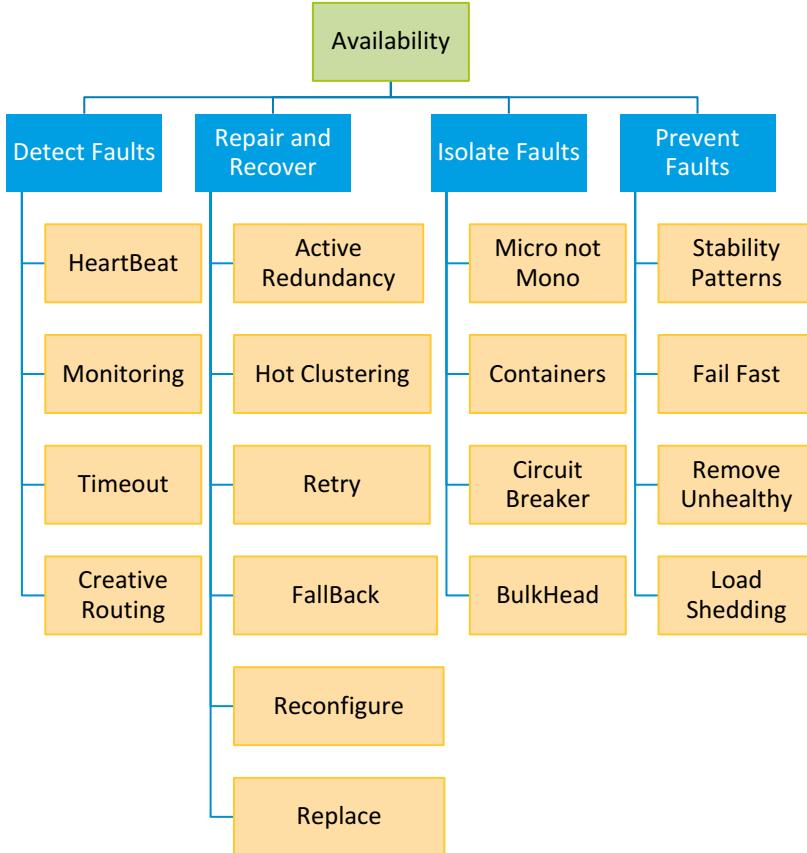
Each decision has:

- Risks
- Non risks
- Sensitivities
- Trade offs

Microservice Scalability Tactics



Microservice Availability Tactics





Intersections



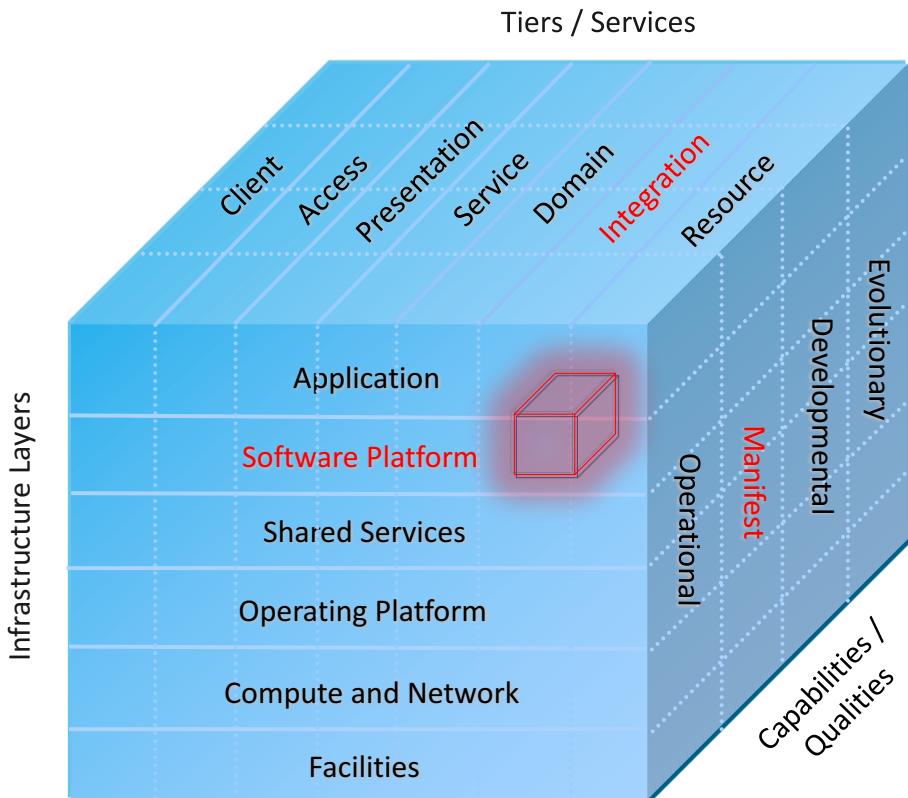
Value: Intersections of Dimensions - Example

The intersections of the 3 dimensions provoke specific evaluation of architecture;

What **Infrastructure** delivers specific **Qualities** for the **Tiers and Services** ?

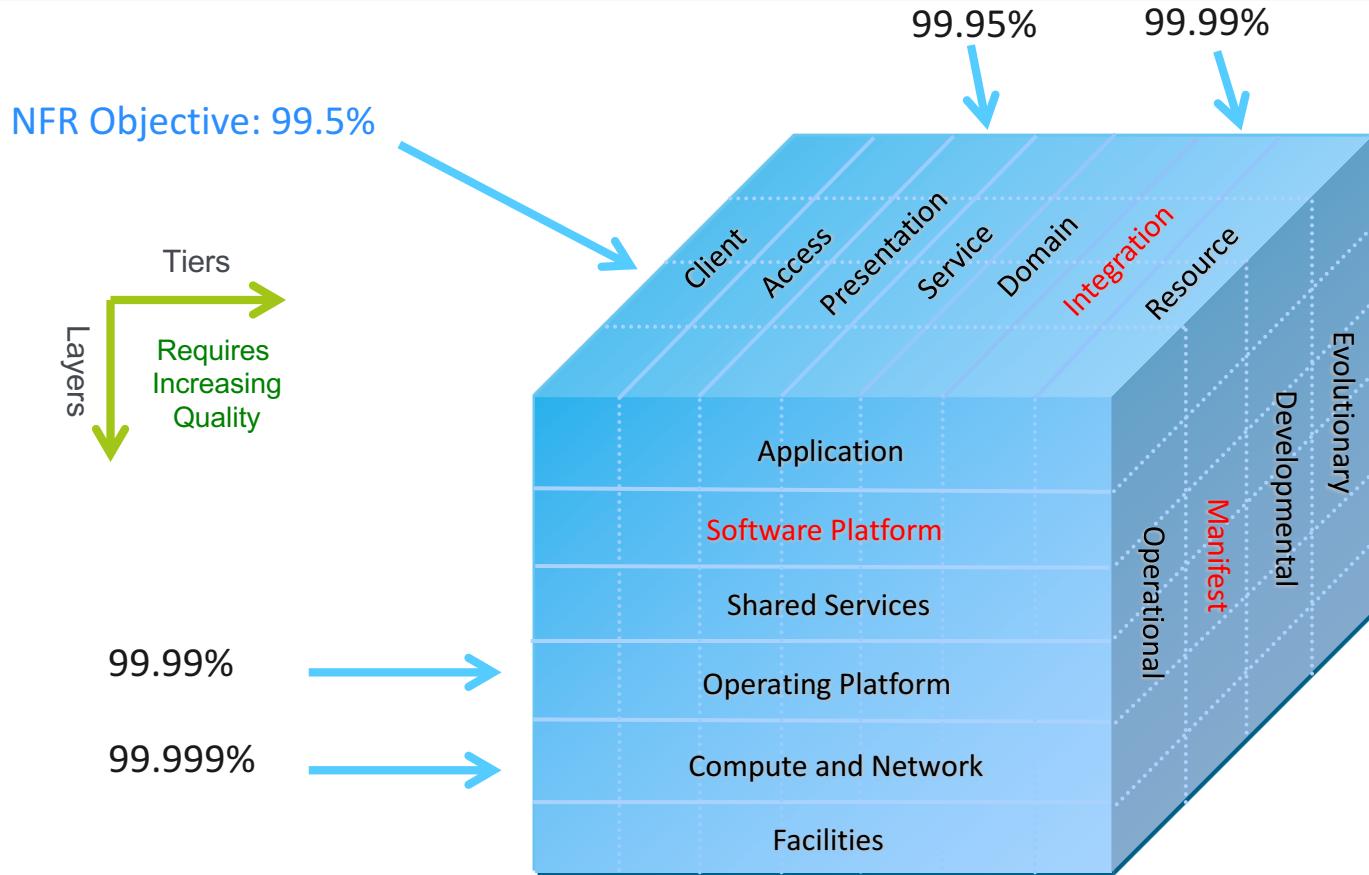
Ask, for example:

- How Sensitive is **Availability** to issues with the **Software Platform** at the **Integration Tier** ?
- How will issues here **manifest** to the user ?





Availability





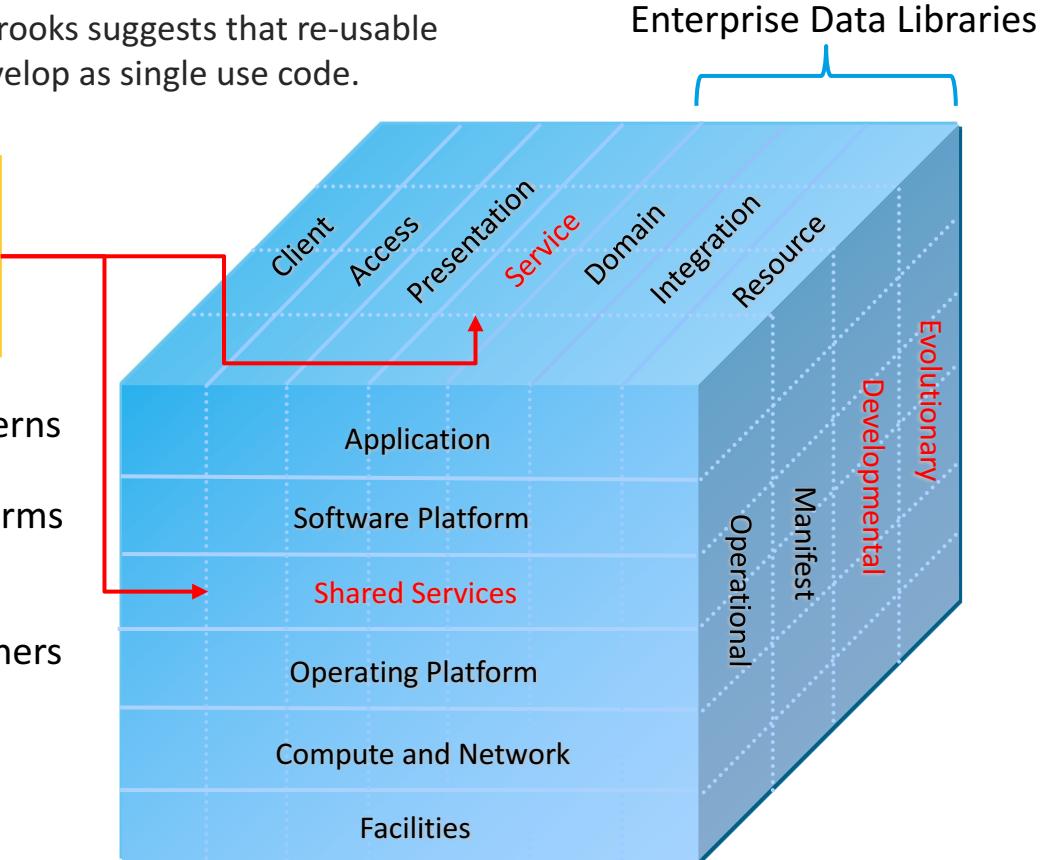
Reusability

In Mythical Man Month (1974) Fred Brooks suggests that re-usable code costs three times as much to develop as single use code.

API / SOA Services built Tactically or Strategically for re-use.
• \$\$\$'s to get to *Shared Services*.
• New qualities: scalability, etc...

Harder - Costly

Code, Patterns
Frameworks, Platforms
VMs, Containers
Easier - Cheaper





Re-use by layers

3 Types of Reuse:

1. *Black-box* - the artifact is used unmodified → only one copy to maintain (*measured*)
2. *White-box* - the artifact is modified for use on projects → multiple copies to maintain (*not measured*)
3. *As-example* - ideas are drawn from the artifact (*not measured*)

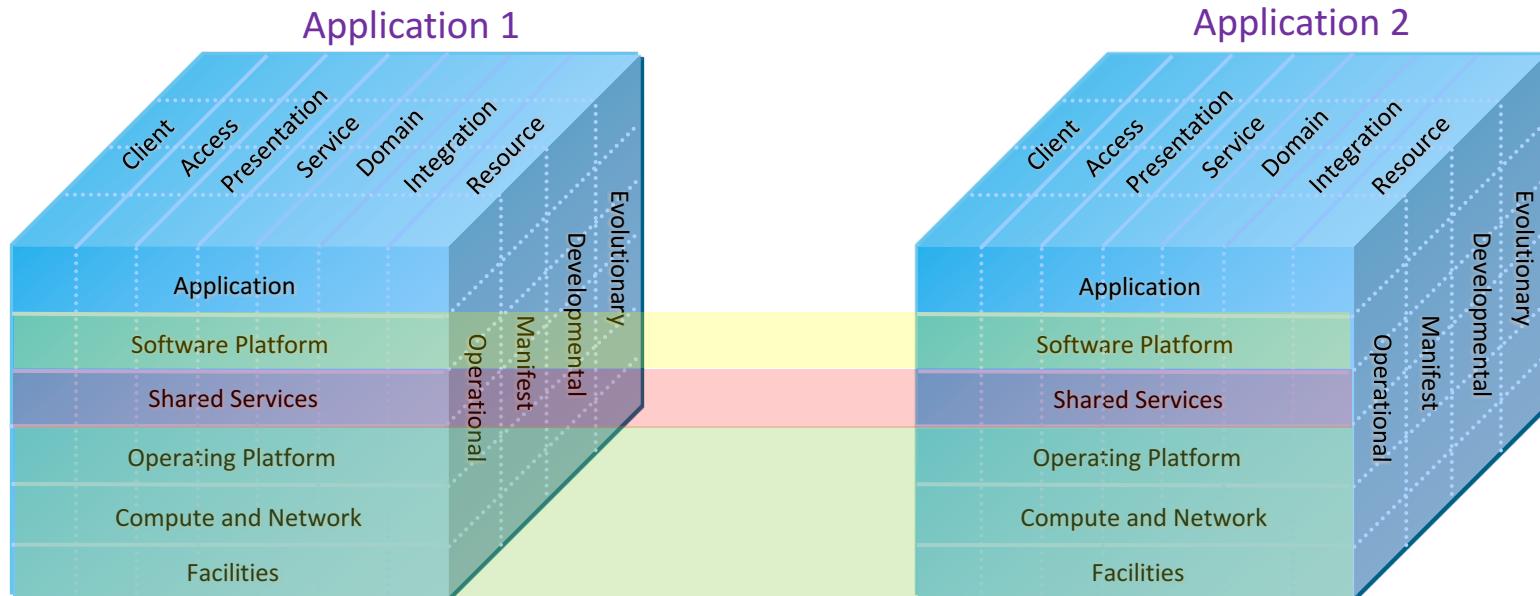
Layers	Across All	Each Layer
Application	<ul style="list-style-type: none">• Solutions• Designs• Requirements	Code (cut& Paste), Libraries, White Box, Black Box, Common Business Services, Business Service, Utilities Modules, Libraries
Software Platform	Models	Languages, Black Box (Frameworks, Platforms)
Shared Services	Styles Patterns	Services (SOA) Custom Mechanisms Incorporated Mechanisms – many are part of products
Operating Platform	Architectural Design Platform	Reuse whole layer VMs, Containers, Deployment/Build Scripts Incorporated Mechanisms
Compute & Network	Specifications Views	Reuse whole layer



The Cubes across the Enterprise

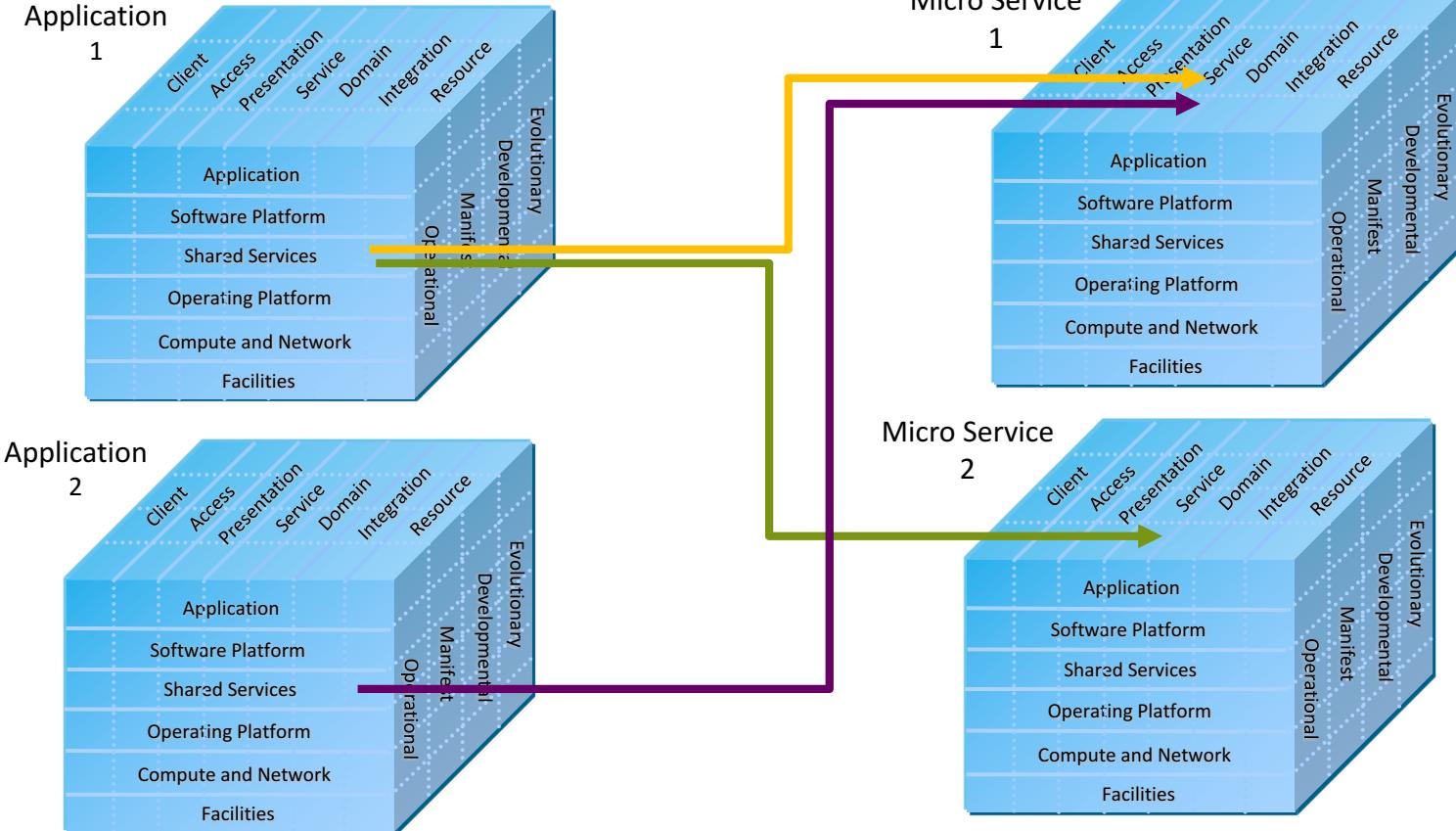
Applications and Services can share infrastructure, including: Enterprise Services, Hardware, Network and Facilities.

- Shared and cross cutting concerns can be moved into their own PAD (physically shared) or Reference Architecture Document (conceptually shared)
- The Layers where a shared 'Platform' starts and ends varies.





Composite Applications and Micro Services





The Views

Static Vs Runtime Views - The code is all you need ?

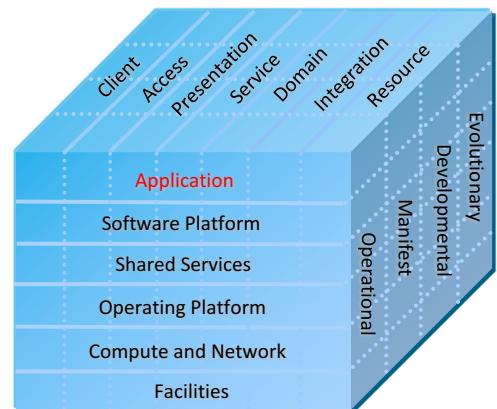


"An object-oriented program's runtime structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's runtime structure consists of rapidly changing networks of communicating objects.

In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa."

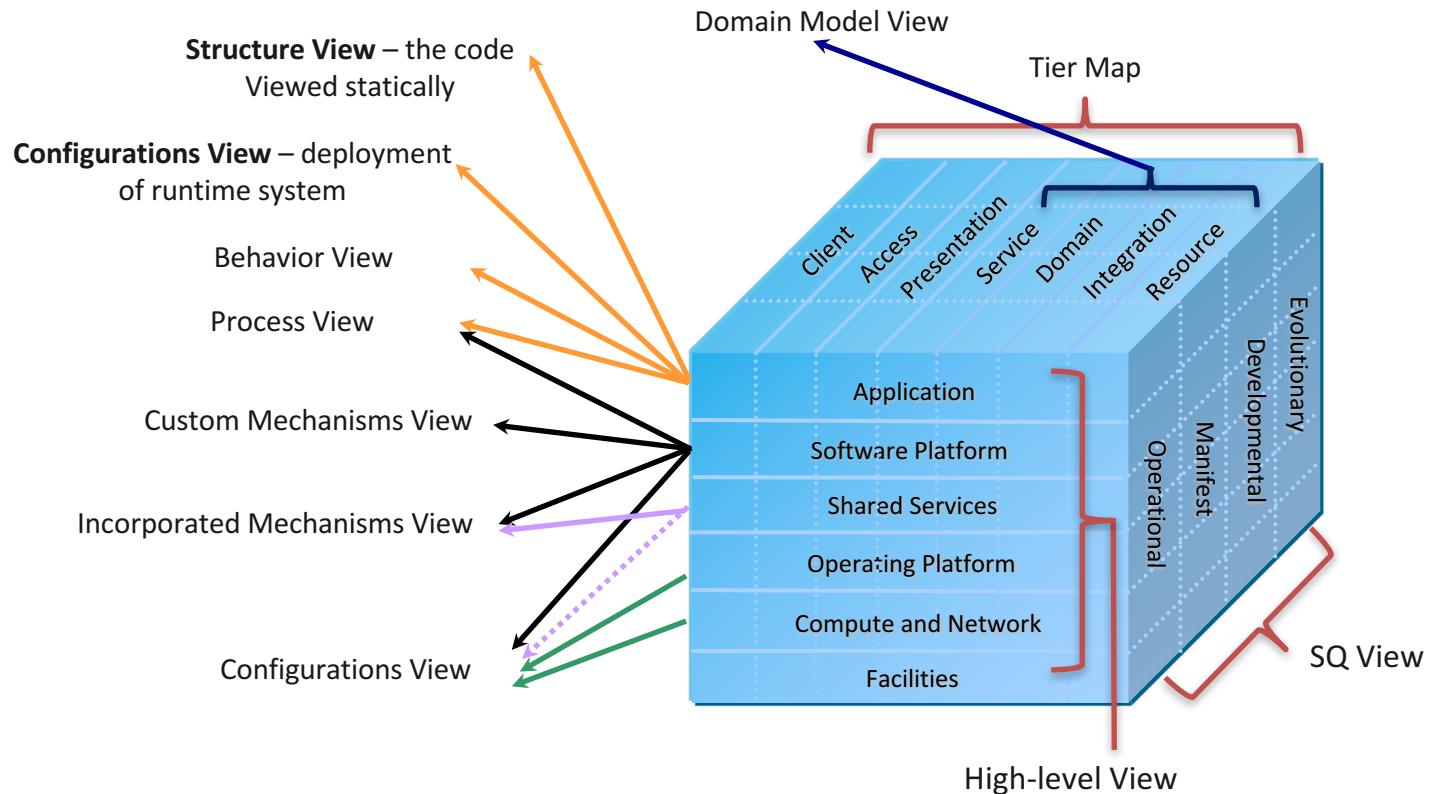
Gamma, Helms, Johnson, and Vlissides

- It is often suggested that the code has the last say over describing a system.
- However, the code is just one view; a very static view of the application.
- It says nothing about the dynamic system, how it is deployed, horizontally scaled, where, when and to what the code is deployed, and how the qualities are delivered;
- Static Vs Runtime are just 2 views - there are many more, required to support the many stakeholders viewpoints.





The Cube – The Context for Organising Stakeholder Views

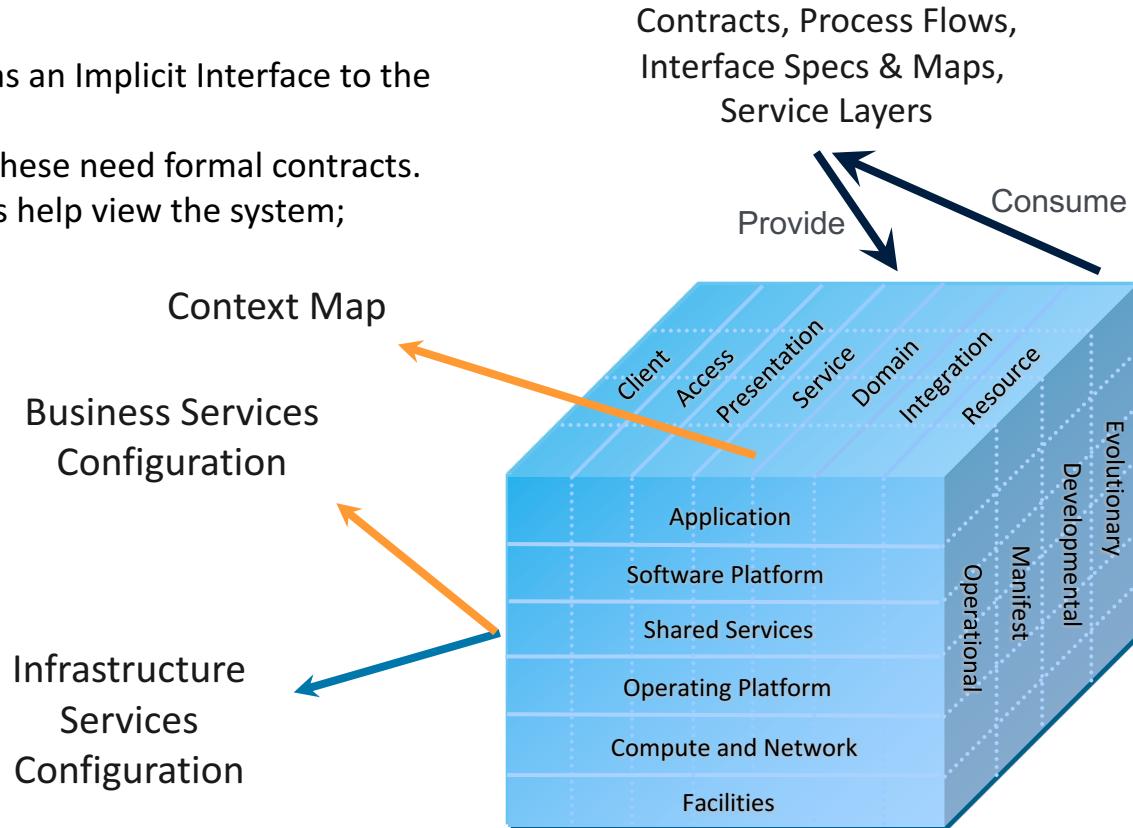




Interface Contracts

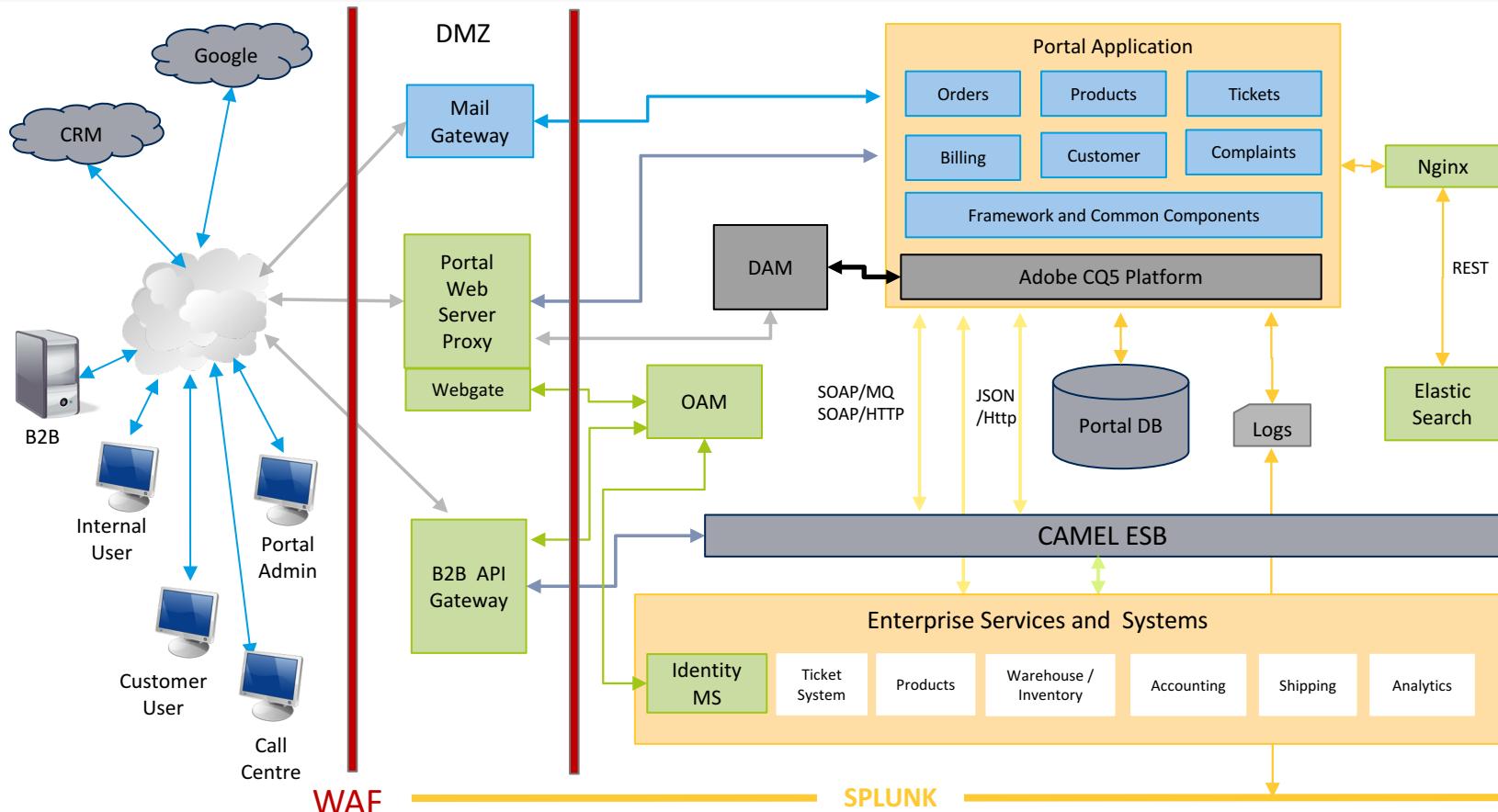
Each Tier/ Layer has an Implicit Interface to the next; layering.

- Only some of these need formal contracts.
- Other artefacts help view the system;



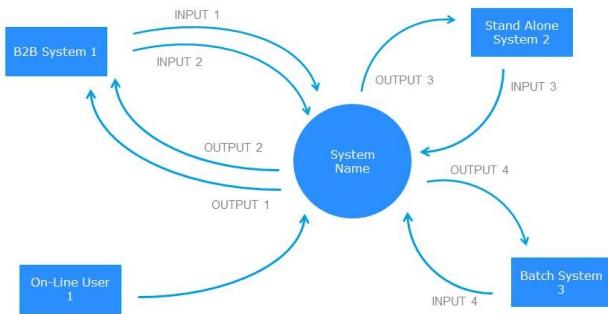


High Level View – Ordering Portal and API Example





High Level Views – Context and Scope

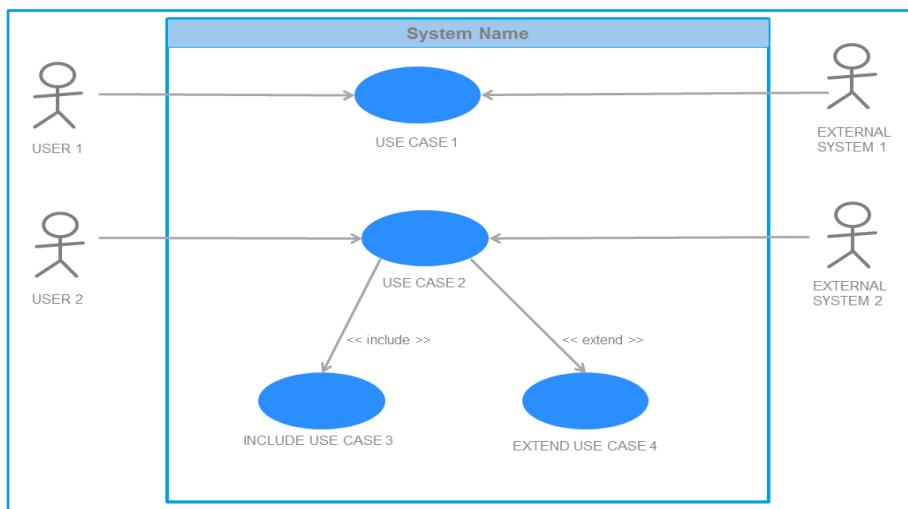


The Solution Context artefact helps define the scope of the system at an early stage in the project.

- Used amongst the key stakeholders to set expectations, gain agreement on the scope and assist in delineating the project team and client responsibilities.
- Can be developed Iteratively

A Use Case Diagram describes the scope from point of view of the various users of the system and how they achieve their goals.

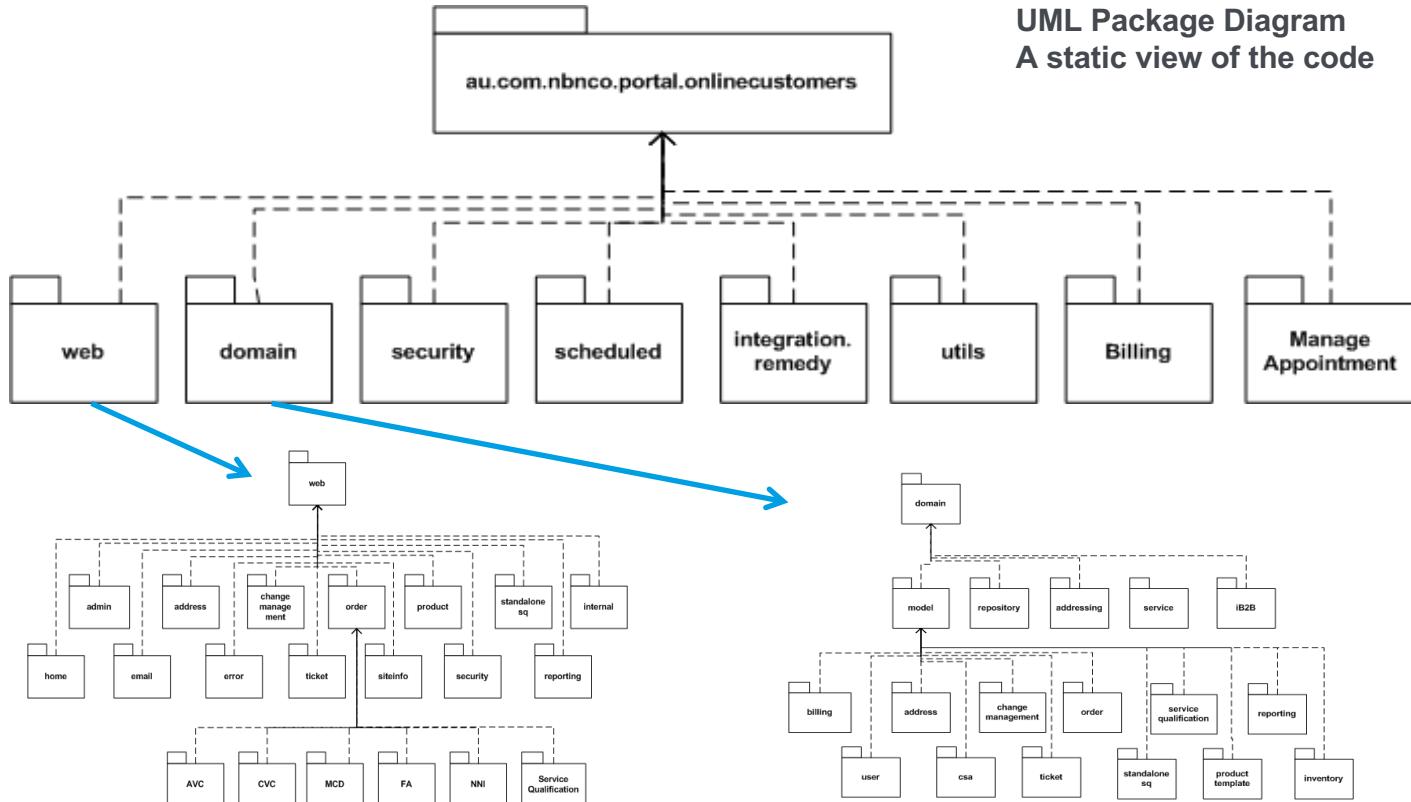
- Shows how actors use the system or how they are impacted.
- Provides some visibility into the system explaining the systems capabilities.



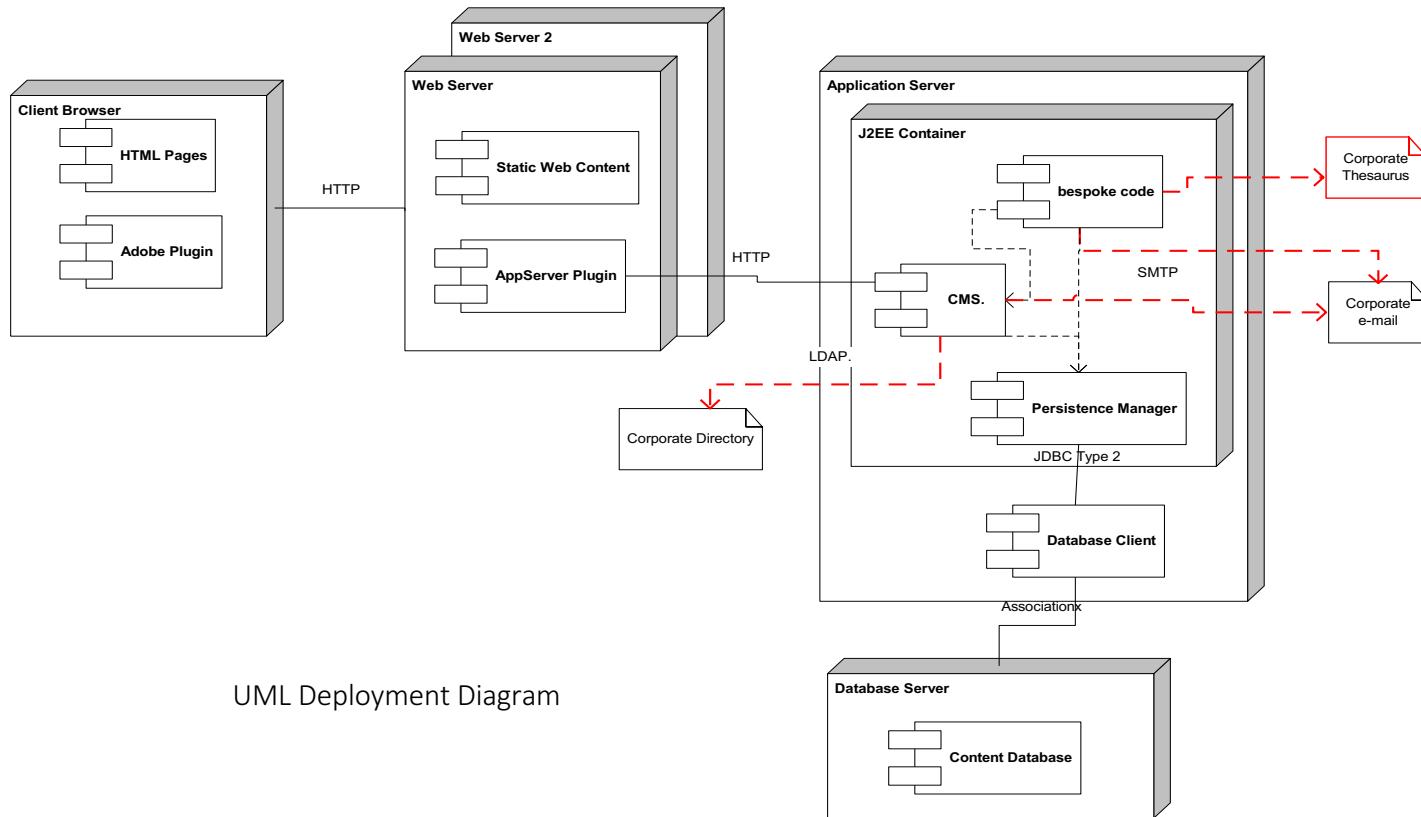
Application Structure View - Example



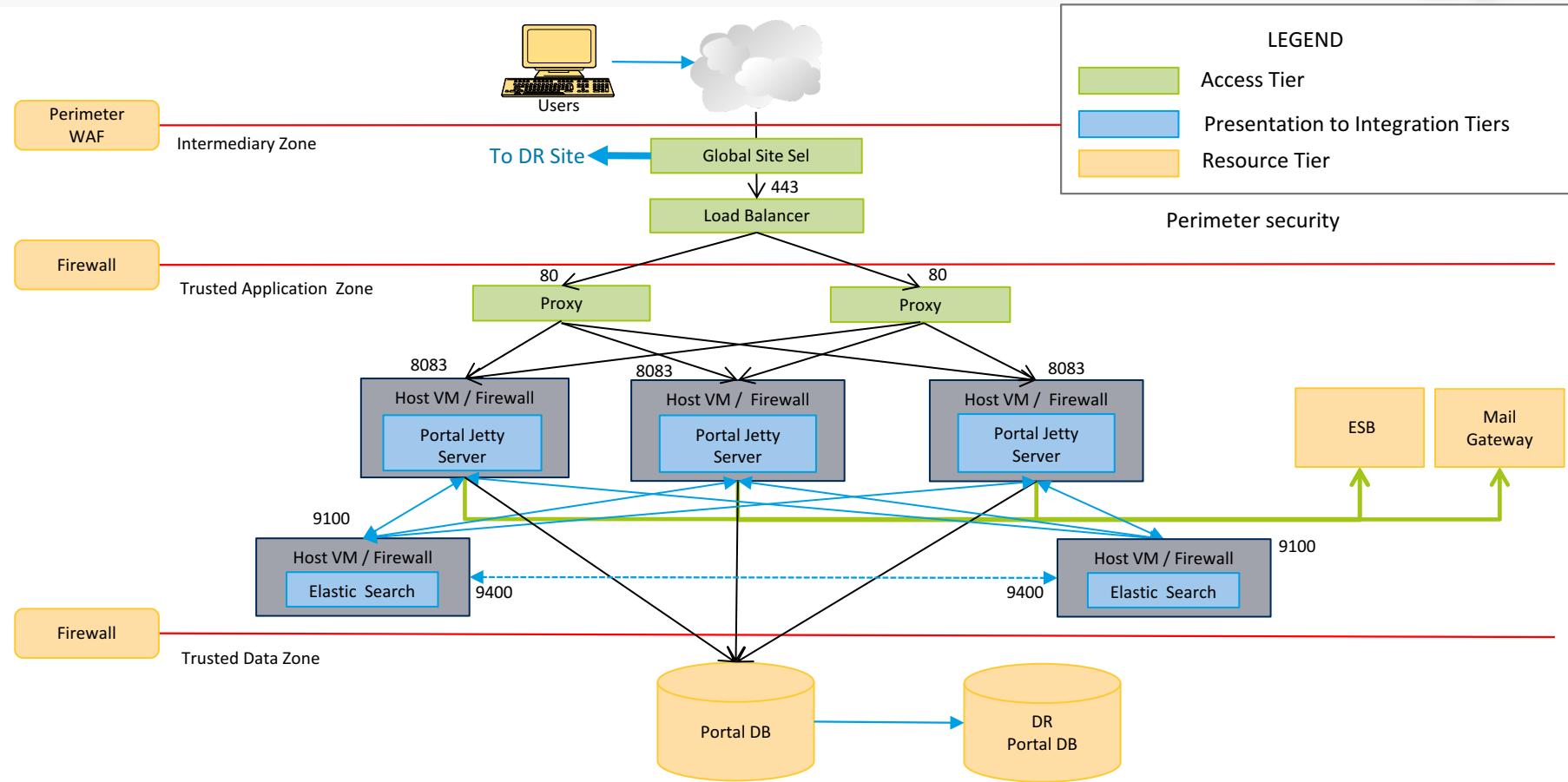
UML Package Diagram
A static view of the code



Application Configuration View – Runtime Deployment - Example



Network Configuration + Tiers across Security zones



Tier Map View – Entity/Data Management at each Tier - Example



Data gets transformed and managed by different technology as it goes up the Tiers

Arch Decision	Presentation	Service	Domain/Integration	Resource
Data Structure	Value objects and JSP helper beans	XML/JSON	JEE Entity beans with POJO by-value dependents	SQL primitive types and Oracle BLOBs for images
Collection Handling	Lists of value objects	List of XML/JSON Array	List of Entity Bean keys	Relational tables
Query API	JEE and JQuery calls	XPATH (XML Only)	JEE calls	Relational tables and views for reads, stored procedures for updates
Navigational API	No local navigation	XML as above	Embedded by-value dependent objects, otherwise through canned services	SQL
Integrity Management	Local format validation for structured fields, drop-down lists from back-end data sources	XSD (XML Only)	Full integrity checking with some redundancy with resource and client tiers; exception errors with explanation strings	Foreign key and some triggers in database
Transaction Control	None	XML - WS Atomic Transactions JSON None	For each method invocations using EJB mechanisms	Transaction boundaries managed externally, SQL isolation level of repeatable reads

Qualities View – Describe the Solution Tactics - Example



How are the ‘relevant’ qualities achieved at each layer.

Layers	Component	Security	Throughput	Scalability	Reliability Availability	Manageability
Application	Order Web Page (Java, JSP)	UID & Password	Minimal session state, Lazy Finish	Separable Order System interface	Replicated Session, Loosely coupled functions	Common LDAP User records
Software Platform	Apache Tomcat	SSL, LDAP ACLs, Webgate Proxy	Resource pooling	App Server Clustering	Primary/ Secondary LDAP	Remote SNMP admin
Operating Platform	VMWare VM	Restricted user base, VM isolation	VM Memory On demand	VM Memory On demand, VM duplication	VM Restart (Test, pre-prod environments)	Remote SNMP admin, VMware vSphere vMotion
Compute & Network	Cisco UCS	Restricted user base, physical isolation, firewalls	Load balanced, dedicated server, multi-processor	Expandable width, expandable processor	Clustering, redundant network connections	Remote SNMP admin

Viewpoint Models / Artefacts



Layer	View	Typical Model/Artefact
High Level		A functional description of the application that all stakeholders can appreciate. A custom graphic, system context or UML use case diagram.
Application	Structure	Mainly UML package, Composite Structure diagrams. UML class, object, State machine when required.
	Configuration	UML component or deployment diagrams. Component diagrams can be overlaid on deployment diagrams. Include Interfaces and Ports.
	Behaviour	Patterns, Process diagrams, BPM models, Tier Map. UML Sequence, Activity, Communication and Interaction Diagrams.
	Tier Map	Tier Map (Across Layers) - Information Model (IA), Domain Object Model, Class/Object or ER Models.
	Data @ Layer	
	Process	UML Object Diagram with Process Annotation, Sequence, Communication and Timing Diagrams.
Software Platform	Incorporated Mechanisms	UML Component/Interface Diagrams, Reference to APIs, Specs/Contracts.
	Custom Mechanisms	UML diagrams as for the Application Layer. UML Interface Diagram Reference to APIs, Specs/Contracts
Shared Services	Configuration	UML Component Diagram, reference to Services, APIs, Specs/Contracts
Operating Platform	Configuration	UML deployment diagram showing nodes and dependencies
Compute & Network	Configuration	Compute and Storage : UML deployment diagram showing nodes and dependencies. Network: Diagram (Routers, Load Balancers, Firewalls etc..)
Systemic Qualities		Systemic Qualities Decisions / Tactics by Layer



Views by Stakeholders (subset)

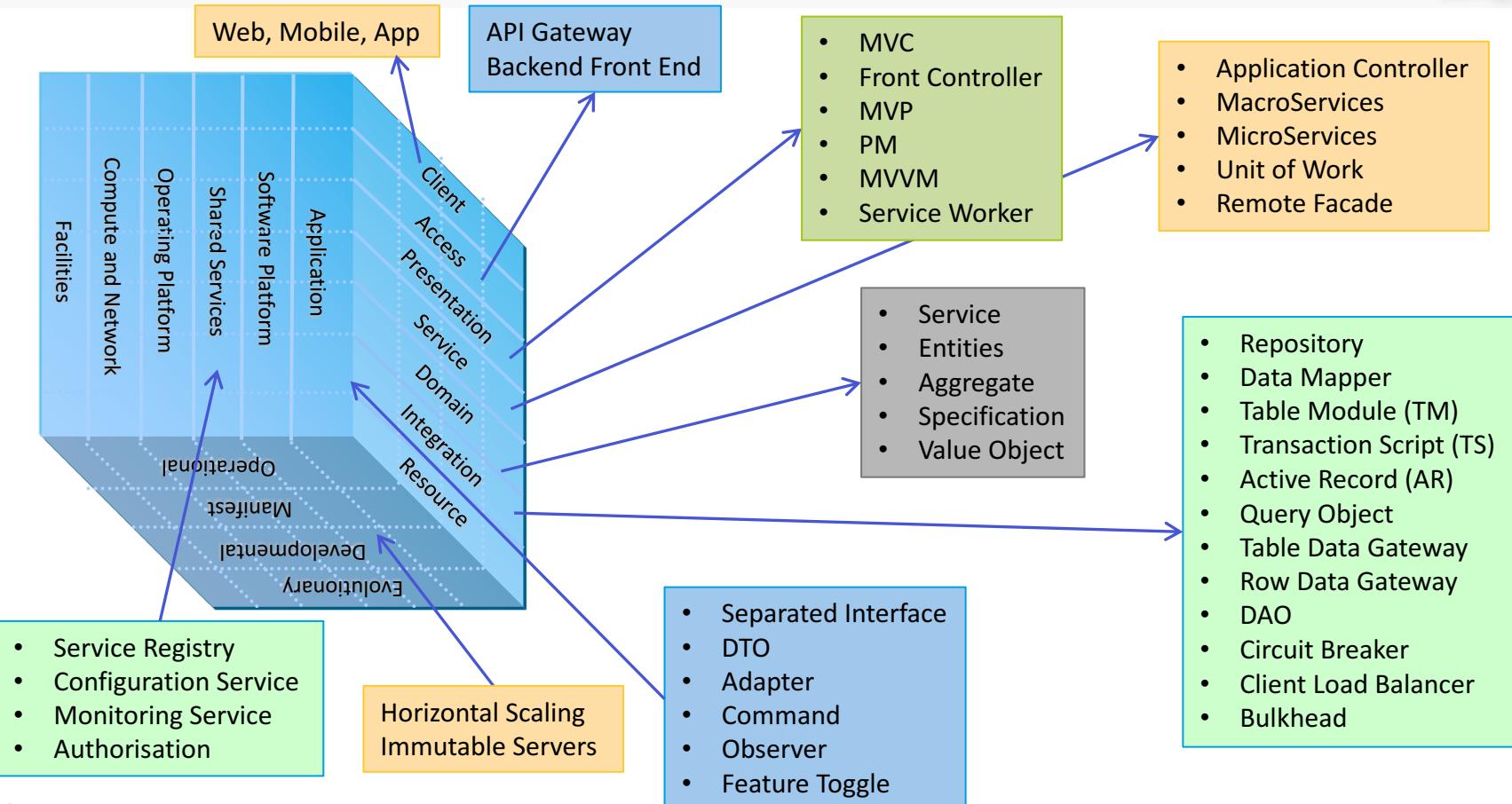
Layer	View	Architect/ Reviewer	Cost Reviewer	Designer/ Builder	Project Mngr	Maintainer	Operator/ Installer
Application	Structure	P	S	P	P	P	
	Config	S	P	S	S	S	P
	Behaviour	S		P		P	
	Process	P		S		P	
Software Platform	Incorp Mechs	P	P	S	S	P	
	Custom Mechs	P	S	P	P	P	
Shared Services	Config	S	P	S	S	S	P
Operating	Config	S	P	S	S	S	P
C&N	Config	S	P	S	S	S	P
Systemic Qualities		P		S		P	

P = Primary Interest

S = Secondary Interest



Patterns View – Organised by the cube - Example





1. Introduction

- 1.1. Background
- 1.2. Purpose
- 1.3. Scope

2. Goals and Requirements

- 2.1. Quality of Service Goals
- 2.2. Technical Requirements
- 2.3. Constraints and Dependencies
 - 2.3.1. Development Process and Team
 - 2.3.2. Environment and Technology
 - 2.3.3. Delivery and Deployment
- 2.4. Assumptions

3. Risks and Architecturally Significant Requirements

- 3.1. Risks Addressed
- 3.2. Architecturally Significant Requirements

4. High Level Solution Overview

5. Software Application Layer

- 5.1. Structure View
 - 5.1.1. Client Tier
 - 5.1.2. Access Tier
 - 5.1.3. Presentation Tier
 - 5.1.4. Services Tier
 - 5.1.5. Domain Tier
 - 5.1.6. Integration Tier
 - 5.1.7. Resource Tier
- 5.2. Configurations View
- 5.3. Behaviour View
- 5.4. Process View
- 5.5. Security and Identity
 - 5.5.1. Identity Management
 - 5.5.2. Access Management
 - 5.5.3. Encryption / Tokens / Certificates
- 5.6. Data Across Tiers
- 5.7. Evolutionary Considerations

Evolutionary Considerations – where are we heading strategically, next release, architectural runway.

*SAD = System Architecture Document



6. Software Platform Layer

- 6.1. Client Tier
- 6.2. Presentation Tier
- 6.3. Integration tier
- 6.4. Resources Tier

6.5 Software Infrastructure

- 7.1. Incorporated Mechanisms
- 7.2. Custom Mechanisms
- 7.3. Evolutionary Considerations

8. Enterprise Services and Infrastructure

- 8.1. Services Configuration View
- 8.2. Infrastructure Configuration View
- 8.3. Evolutionary Considerations

9. Operating Platform Layer

- 9.1. Configurations View
- 9.2. Evolutionary Considerations

10. Compute and Network Layer

- 10.1. Configurations View
- 10.2. Evolutionary Considerations

11. Facilities

11. Qualities of Service Review

- 11.1. Availability
- 11.2. Scalability
- 11.3. Performance
- 11.5. Agility, Extensibility and Flexibility
- 11.6. Security, etc



For each Quality / Concern

Item	Description
Requirement	
Architectural Solution	
Validation Strategy	
Evolutionary Considerations	

The Value of the Cube – a tool for analysing your architecture



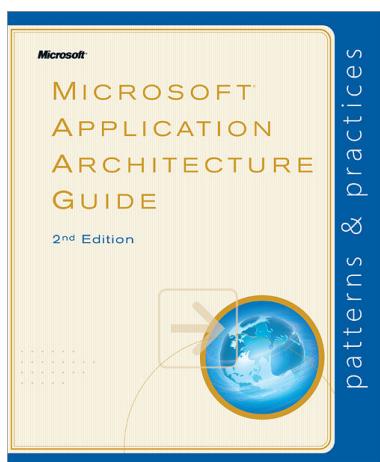
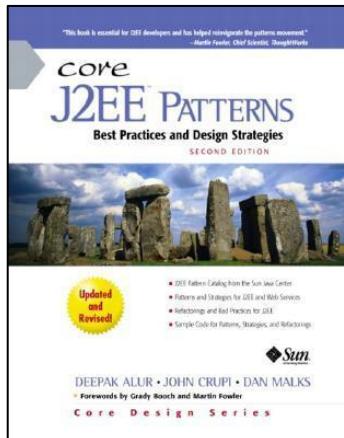
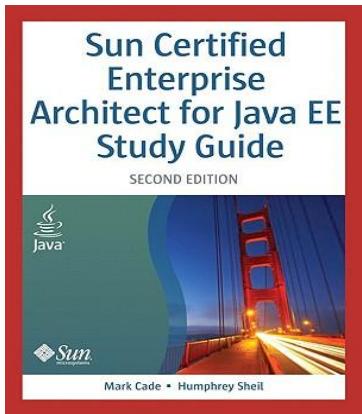
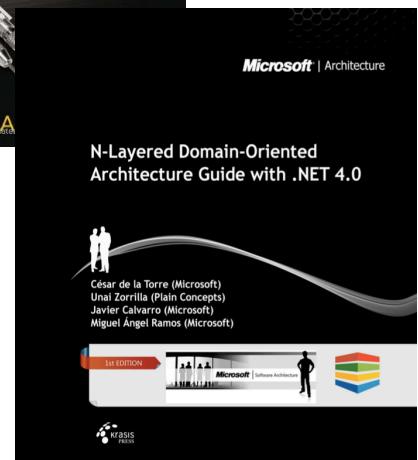
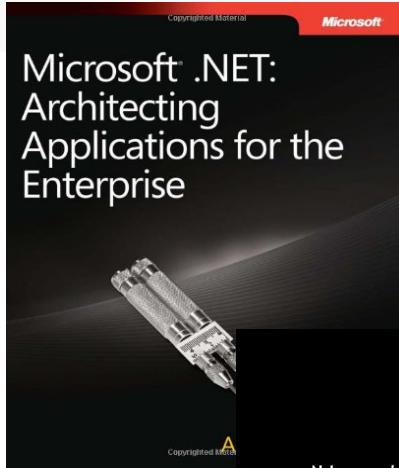
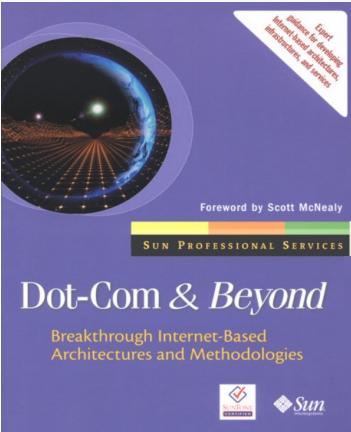
- Make Architecture Orthogonality Explicit by Clarifying:
 - **Tiers:** Conceptual, Logical and Physical:
 - Services;
 - Decomposition Decisions;
 - Deployment Decisions.
 - **Layers:** of Software/Hardware, Platforms and Infrastructure;
 - **Qualities Of Service (QOS)**
 - Cross Cutting concerns that drive continuous architecture.
- Provide Views and Viewpoints:
 - Provide a clear set, relevant to stakeholders;
 - Organised in context, within layers, across tiers and QOS.
- A Framework that is both:
 - Descriptive – use to analyse, review, evaluate and explain existing systems;
 - Prescriptive – model new systems.
- Basis for an architectural process and decision making.
- Covers:
 - Modern Styles: N-Tier, SOA, Micro-Services and Cloud(PaaS, IaaS, SaaS)
- Benefits: Agility, Continuous Delivery, Reuse

References:



Cube was basis of:

- Sun's 'Dot-Com & Beyond' - Architecture Methodology
Published: June 11, 2001;
- Sun Certified Enterprise Architect for Java EE Study Guide, 2010;
- J2EE Core Patterns;



Extended to cover
.Net platforms

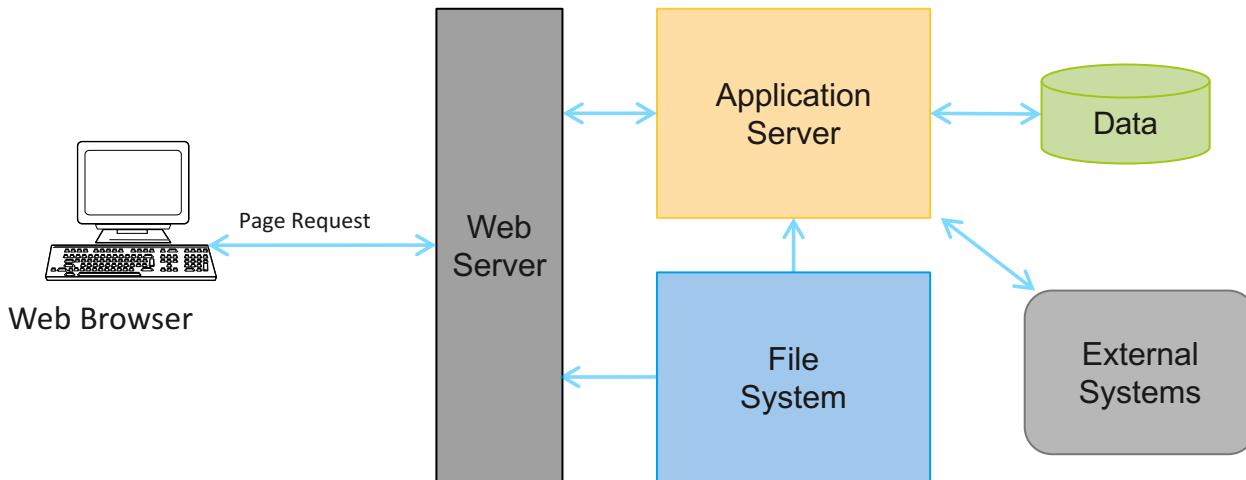


The End

Web Architecture - Back in 2001



“The Architecture of Web Applications”, Grady Booch, 1 Jun 2001



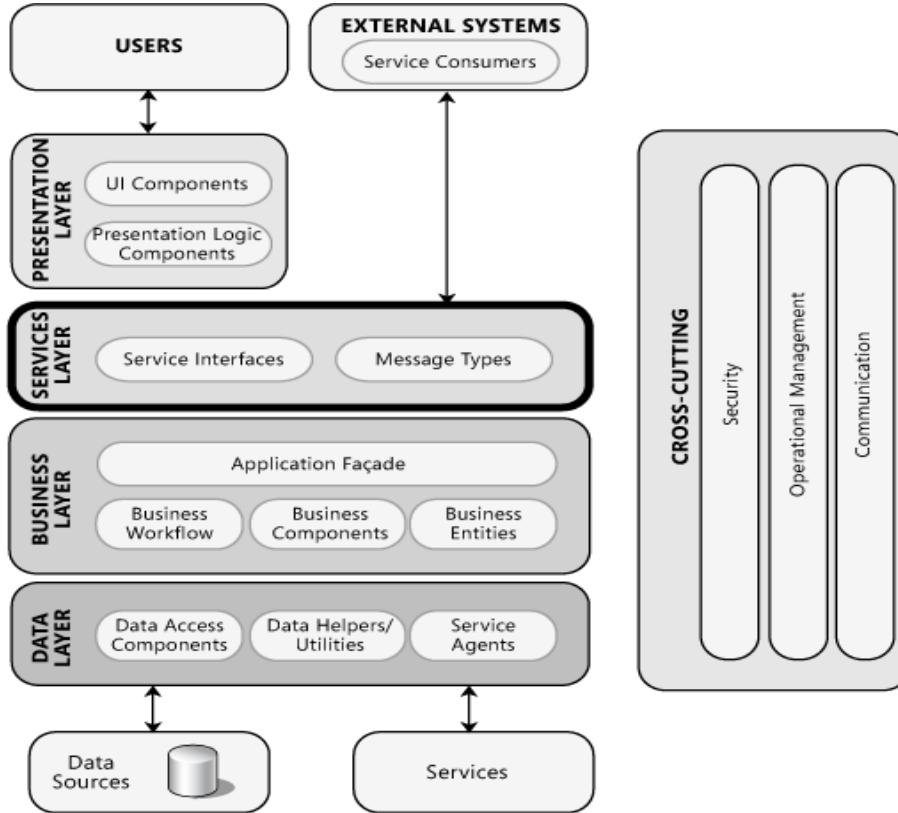
Issue: This is a single very specific, simplistic and high level view.

MicroSoft Application Architecture (Early)

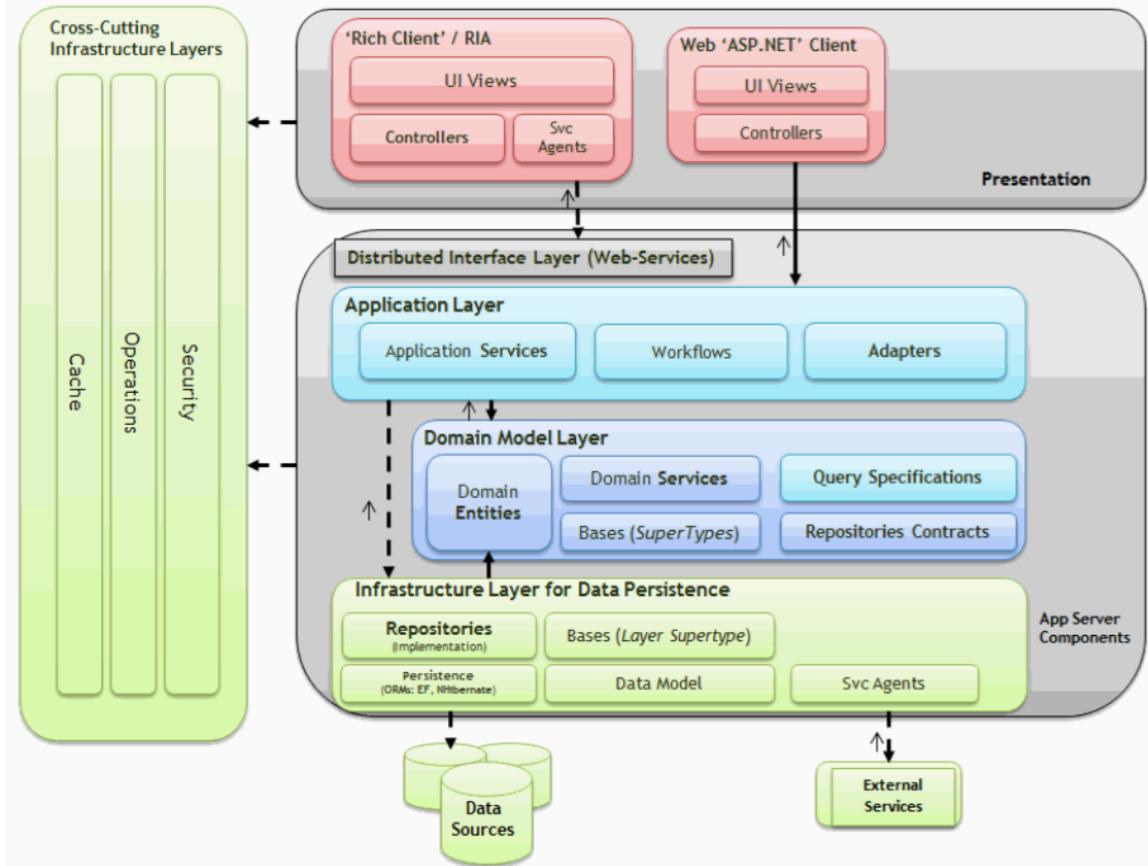


Add:

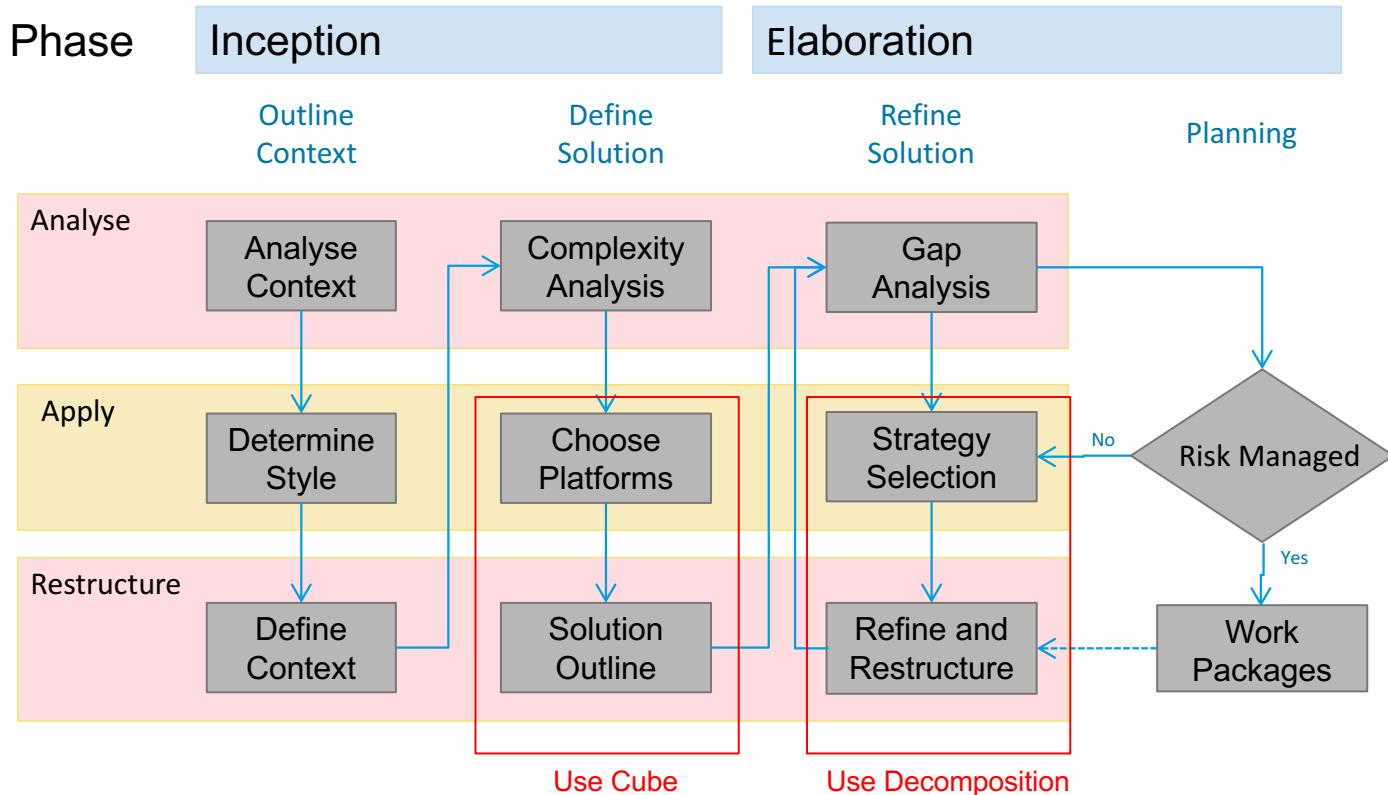
- Layers
 - Service Layer
 - B2B by-passing presentation
 - Components
 - Patterns
 - Cross-cutting concerns



Microsoft DDD N-Layer Architecture Style (Recent)



Cube Architecture Process





Cube Architecture Process

Choose Platforms Step:

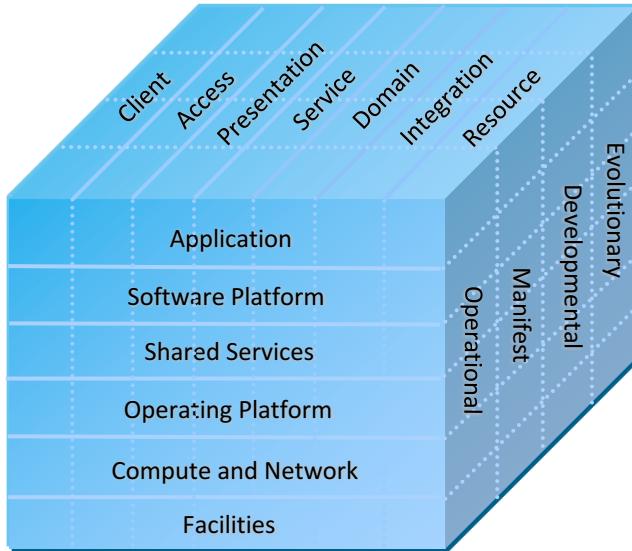
While Risks/Gaps do {

- Select
- Instantiate
- Determine Arch Significant Items
- Analyse Gaps
- Analyse Risks

For each Tier {

- Determine the Software and Hardware Stack
- Document Incrementally using Cube Template

}





12 Decomposition Heuristics

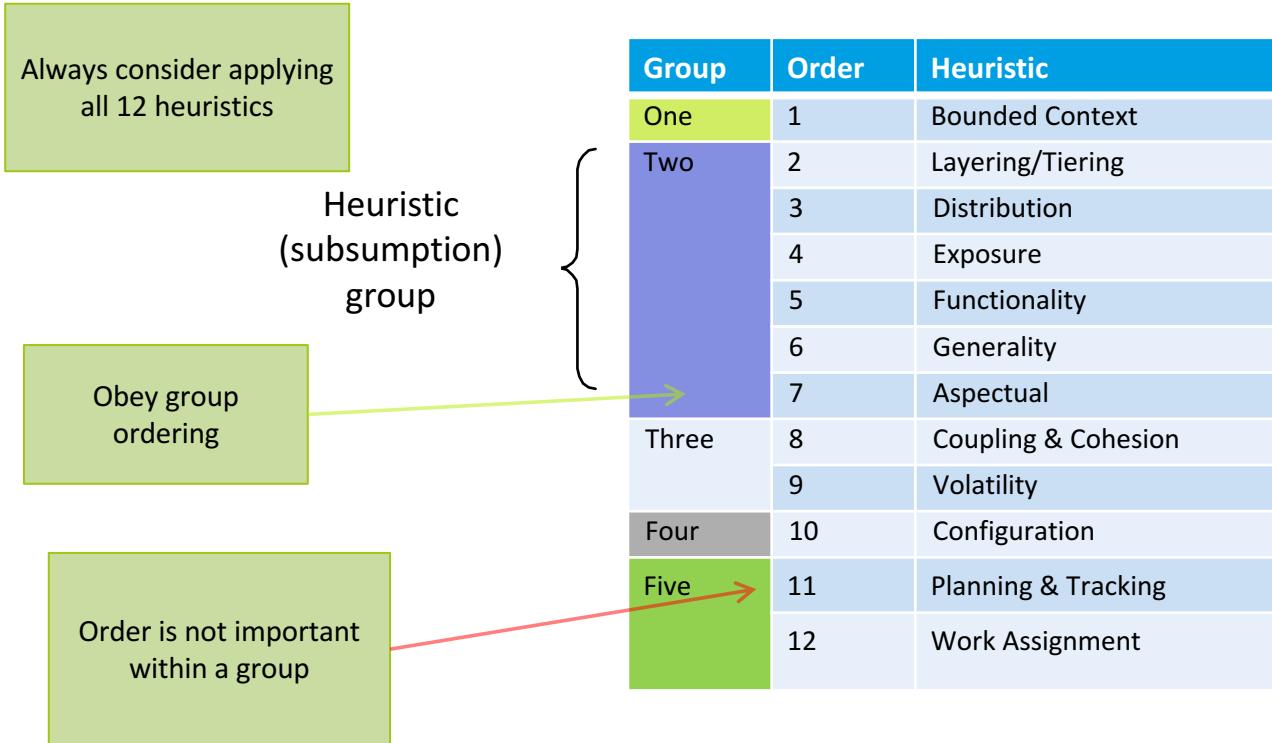
1. Bounded Context
2. Layering/Tiering → 2 Dimensions of Cube
3. Distribution
4. Exposure
5. Functionality
6. Generality
7. Aspectual
8. Coupling & Cohesion
9. Volatility
10. Configuration
11. Planning & Tracking
12. Work Assignment



Decomposition Tactics

Heuristic	Description
Bounded Context	Partition the domain into areas with a common ubiquitous language, use relationship types.
Layering/Tiering	Ordering of concerns; cohesive services within a layer, expose via interface – low coupling.
Distribution	Partitioning by distribution among computational resources, provides scalability;
Exposure	Decide what to expose to other components, via interfaces. Separate implementation.
Functionality	Decompose into Functional units ,e.g. service groupings within the problem space.
Generality	How general or specific to make something to make it reusable across projects or products.
Aspectual	Cross-cutting concerns, implemented as aspects, e.g. logging, security.
Coupling & Cohesion	Partition by Reducing Coupling and Increasing Cohesion.
Volatility	Isolate things that are more, verses less likely to change, or things that simply change on different schedules.
Configuration	Systems must support different configurations (for environments, pricing, usability, performance, security, etc.)
Planning & Tracking	Break a big thing apart so that work can be defined in smaller time units against the smaller parts.
Work Assignment	Partition by physically-distributed teams, matching skill-sets, security areas (clearance), and contractual concerns.

12 Decomposition Heuristics





A microservice architecture is:

"a service-oriented architecture composed of loosely coupled elements that have bounded contexts."

Adrian Cockcroft (Director of Web Engineering Netflix)

"The microservice 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. These services are built around business capabilities and independently deployable by fully automated deployment machinery."

Martin Fowler in March 2014

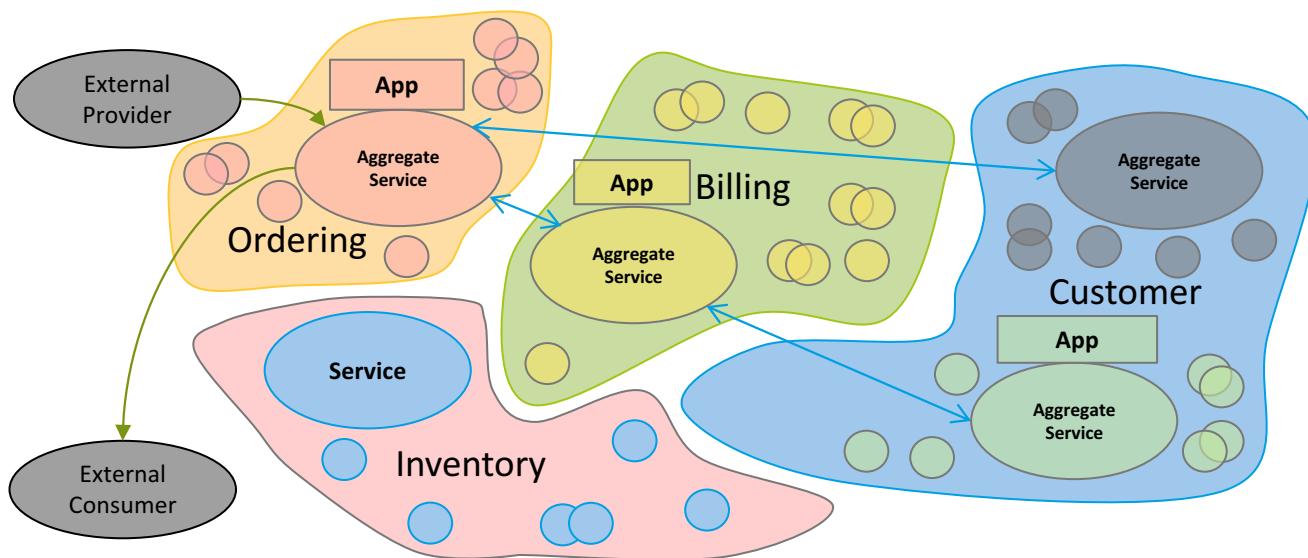
MicroServices - Separated by Concern (Bounded Context)



The business can be broken into 'Small' Fine Grained services that exists within within bounded contexts.

These contexts have a separation of concerns and capability, that are independently and continuously deployed, managed, monitored.

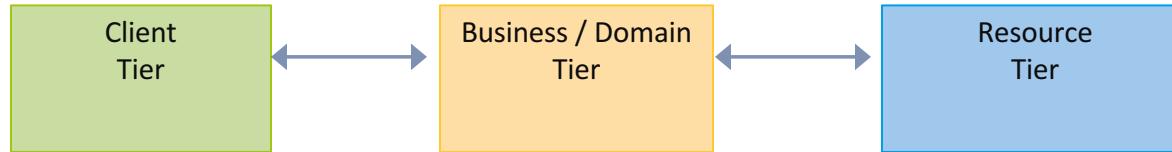
- Applications may be made up of 100's of MicroServices;
- May need to provide encompassing aggregates of these for consumers to consume more easily;
- Need to provide a way for consumers to find and use services.





Distribution (Tiering)

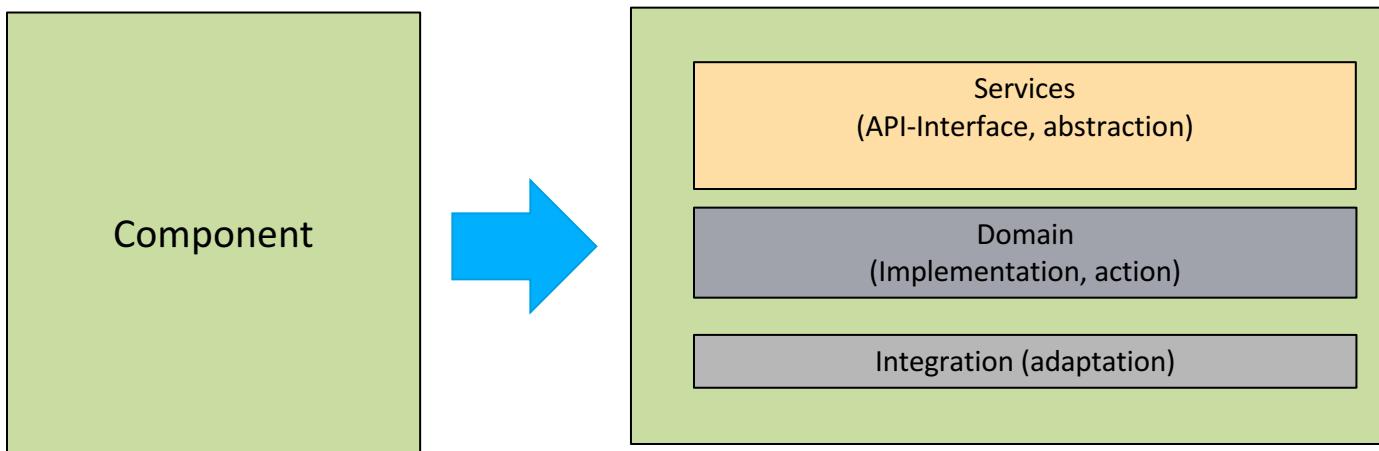
- Decomposition according to some ordering principle, typically partitioning and **distribution** among computational resources (containers/VMs/boxes).
- Primary technique for building scalable systems.
- Separation of concerns to separate layers and then deploy each separately to separate tiers.
- Example: 3 Tier System





Exposure - Break Monolith into Layers

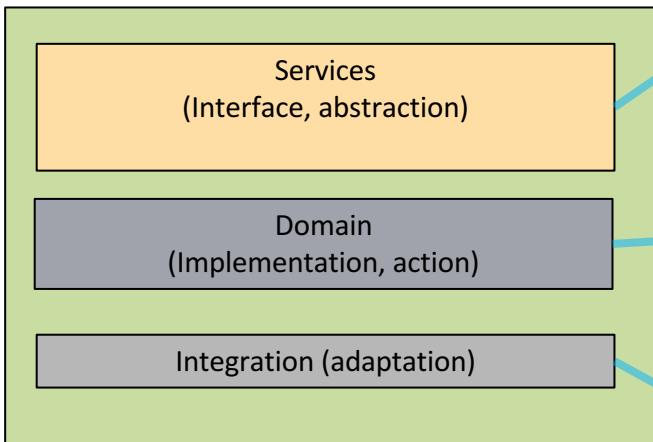
- Decide what to expose to other components, via an API
- Any given component fundamentally has three different aspects (layers)



Micro Service Layers – Exposing a Component



Applying the layered architecture pattern to components, e.g. a Microservice component



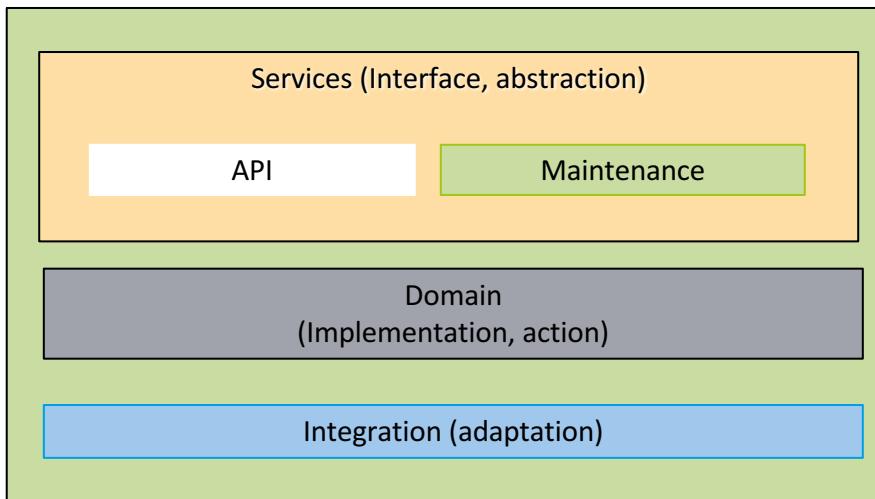
- Protocol Abstraction – Client channel
- Logical Abstraction – expose for particular use cases
- Application Logic Implementation, Workflow/Orchestration.
- Manage Data
- Logical Adaption - of tiers below for what we need (transform and protect via anti corruption layer)
- Protocol Adaption – connect and to talk to tier below



Exposure – Add Operations Interfaces

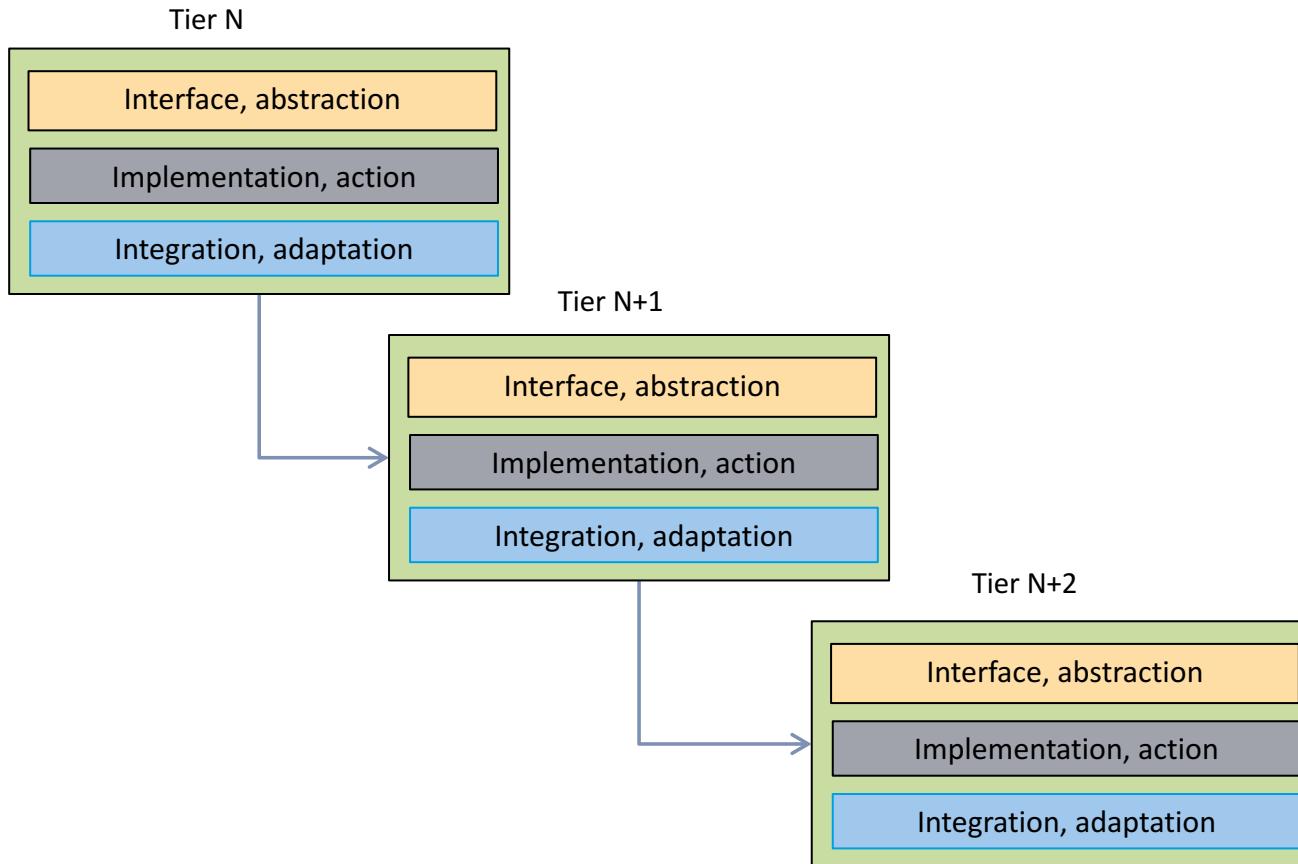
For complex units these may be broken apart into several pieces, and Interfaces for both:

- API (Data and Business Functions)
- Maintenance (Operations), Management of the Component





Exposing Tiers - Layers



Solution Arch Document (Common Mini EA Based Style)



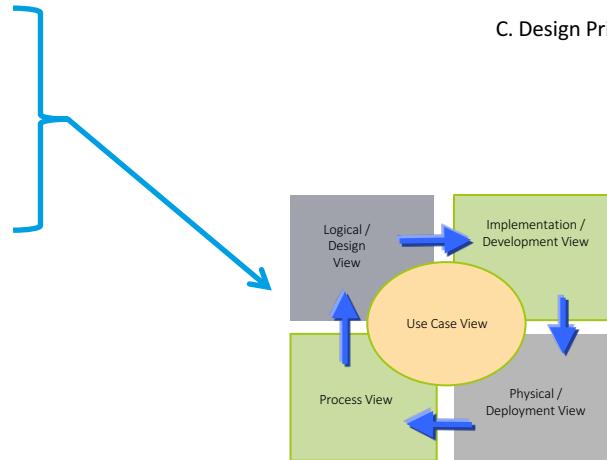
1.	Introduction	7	Application Architecture
2	Capability Definition	7.1	Platform and Capabilities
	2.1 Business Rationale	7.2	Application Context
	2.2 Business Drivers	7.3	Capabilities Definitions
	2.3 Scope	7.5	Non Functional Considerations
	2.3.1 Objectives		
	2.4 Core Functional Requirements		
3	Business Context Overview	8	Data Architecture
4	Solution Overview	9	Integration Architecture
	4.1 Key Non-Functional Requirements	10	Technical Architecture
5	Solution Phase Overview	11	Data Conversions and Migrations
	5.1 Current Solution	13	Risks, Issue, Assumptions and Dependencies
	5.2 Proposed Solution	14	Architectural Decisions
	5.3 Identified Future Capabilities		
6	High Level Process Overview		
	6.2 AS-IS Process Impacts		

A mini EA, a segment capability.

Solution Arch Document (4+1 Based Style)



- 1. Introduction
- 2. Scope
- 3. Software Architecture
- 4. Architectural Goals & Constraints
- 5. Logical Architecture
- 6. Process Architecture
- 7. Development Architecture
- 8. Physical Architecture
- 9. Scenarios
- 10. Size and Performance
- 11. Quality
- Appendices
 - A. Acronyms and Abbreviations
 - B. Definitions
 - C. Design Principles





- **Transaction Script:** A business component that captures procedural user actions and business logic, e.g. order product, book a flight, in a procedure/method.
- **Command Object:** Encapsulate request processing in a separate object with a common execution interface. Used to represent the script, the action.
- **Table Module:** A business component that handles the business logic for a whole DB table, e.g. customer_table, order_table. Although an object is used, it represents a DB table, not a domain object.
- **Active Record:** An object that wraps a record in a DB table or view, represents a row of data. Object holds table/entity attributes, with CRUD methods.
- **Data Mapper.** A mapping layer between objects and the database structure that is used to move data from one structure to another while keeping them independent.
- **Data Transfer Object.** An object that stores the data transported between processes, reducing the number of method calls required.
- **Query Object.** An object that represents a database query.
- **Repository.** An in-memory representation of a data source that works with domain entities.
- **Row Data Gateway.** An object that acts as a gateway to a single record in a data source.
- **Table Data Gateway.** An object that acts as a gateway to a table or view in a data source and centralizes all of the select, insert, update, and delete queries.
- **Record Set:** in memory representation of a DB table;