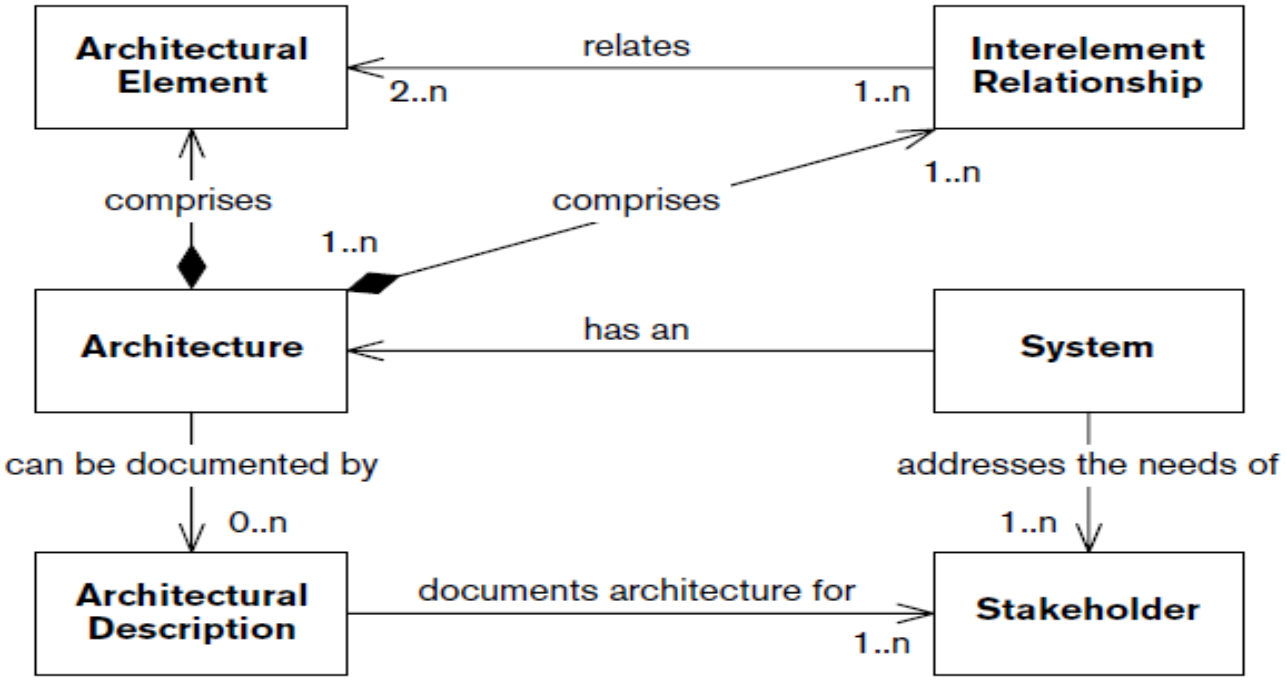
Software Architecture

Architecture Fundamentals

ARCHITECTURE FUNDAMENTALS



CORE CONCEPT RELATIONSHIPS

What is software architecture?

- An architecture is “the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.”
  - - ANSI/IEEE Std 1471-2000

The **architecture of a software-intensive system is the structure** or structures of the system, which comprise software elements, the **externally visible properties** of those elements, and the **relationships** among them.

System Structures

- There are two types of system structure of interest to the software architect
- ***static (design-time organization)***
  - The static structures of a software system define its internal design-time elements and their arrangement.
- ***dynamic (runtime organization).***
  - The dynamic structures of a software system define its runtime elements and their interactions.

As a noun…

Structure

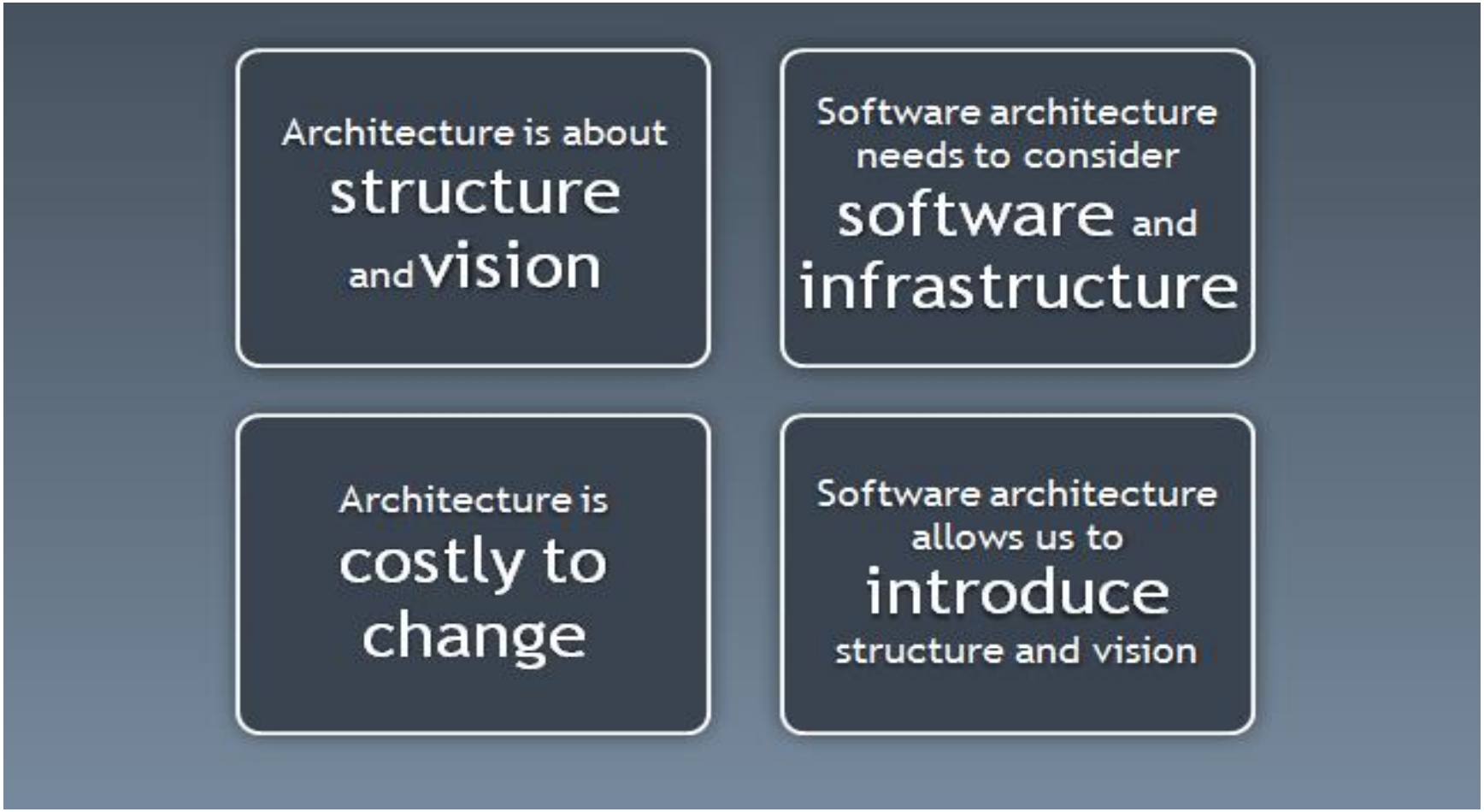
The definition of something in terms of its components and interactions

As a verb...

Vision

The process of architecting;
making decisions based upon business goals,
requirements and constraints,
plus being able to communicate this to a team

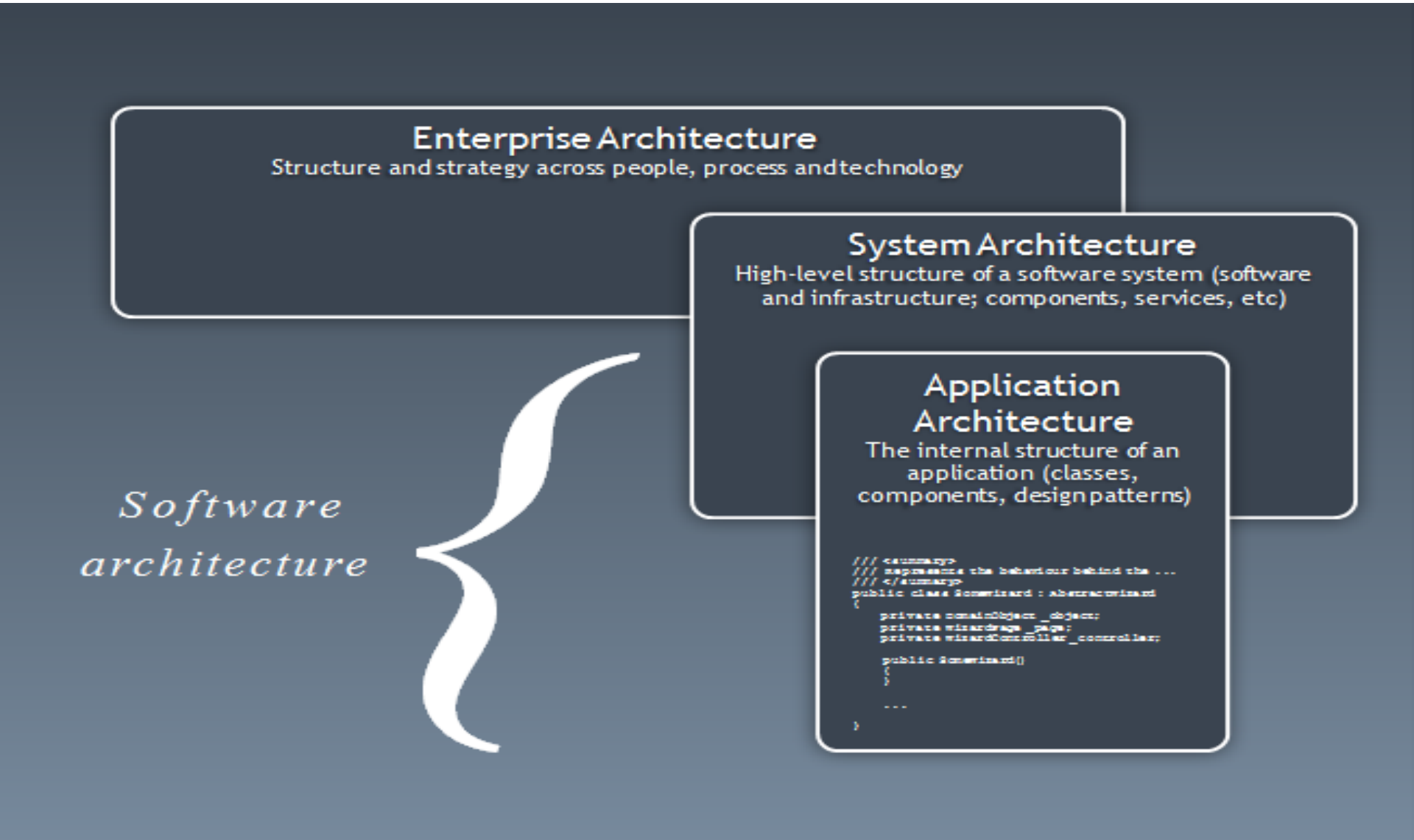
Software Architecture



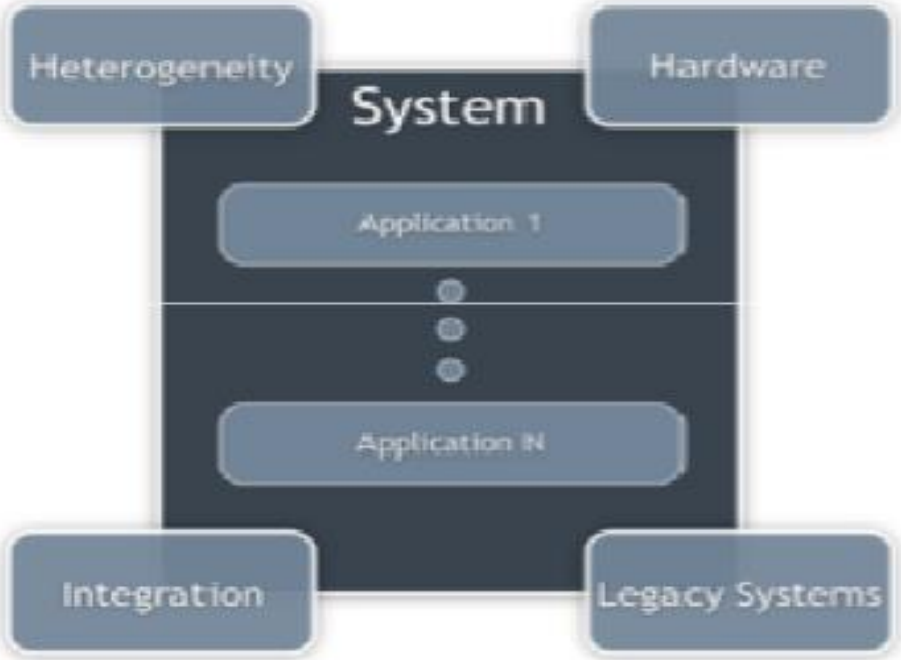
Types of architecture

- Application
- Business
- Data
- Enterprise
- Hardware
- Information
- Infrastructure
- Network
- Platform
- Performance
- Security
- Software
- Solution
- System
- Technical
- Technology
- Web
- ...

Different Definitions of Architecture

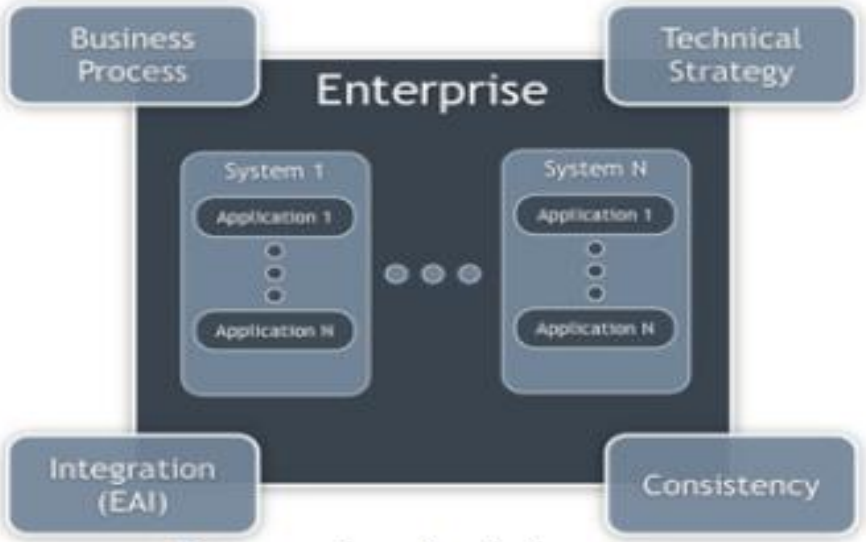


Contd….



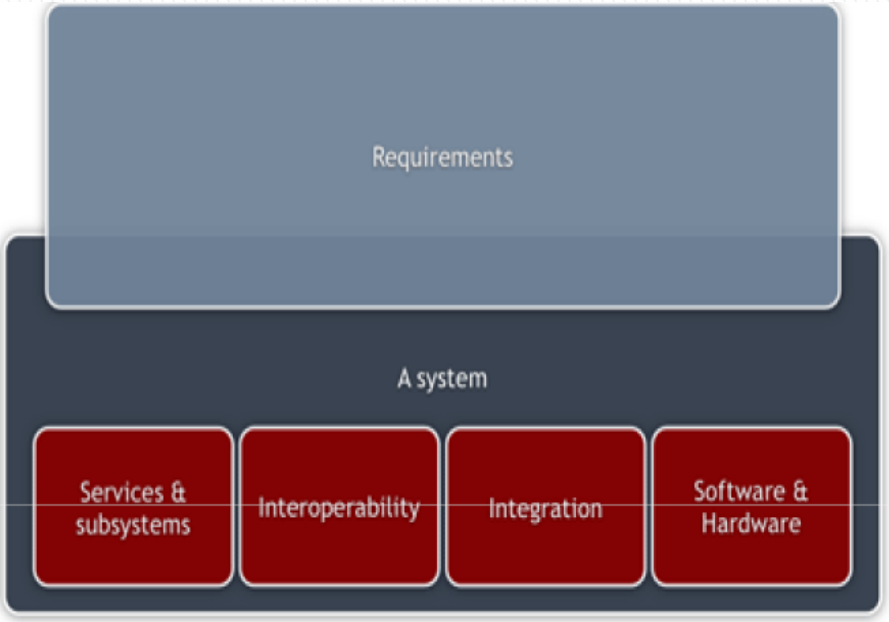
System Architecture

Application Architecture



Enterprise Architecture

System Architecture

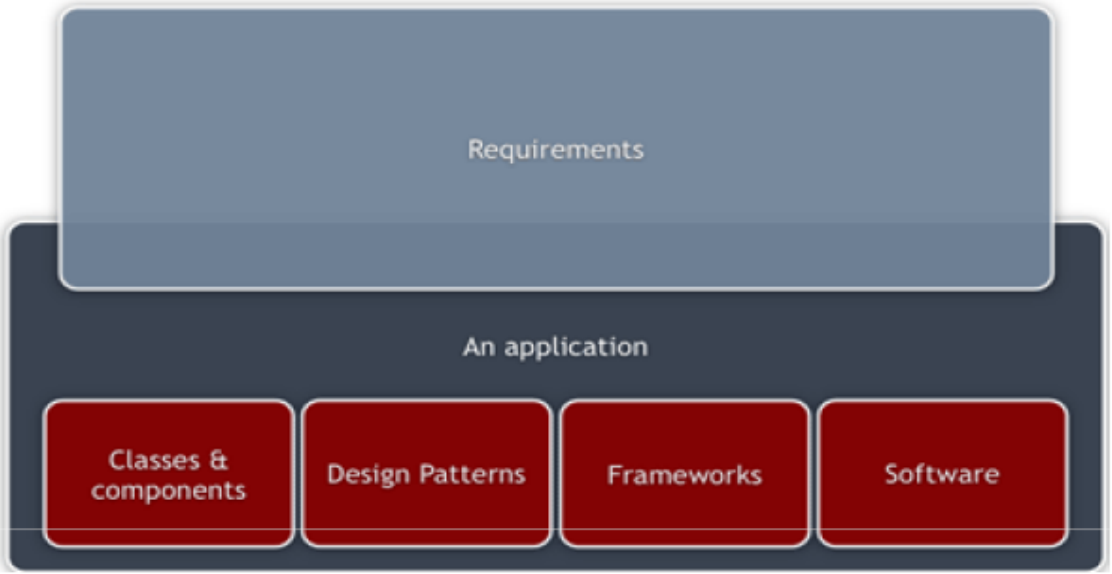


- One step up in scale from Application Architecture
- Does not take into account influences of the technology stack
- Diverse system with many application System and tiers
- Each tier may have its own technology stack
- Each application may have its own application architecture

Decompose the overall system into bigger components (applications) that have the appropriate level of coupling and cohesion.

The term "architecture" is often generically used to refer to the system architecture

Application Architecture



- "Application architecture" is really a subset of the system architecture.
  - Influenced completely by business function and technology stack
- Defined at a lower level than the system architecture.
- We're probably most familiar with application architecture as developers.

- Decompose the application into its constituent classes and components
- Make sure design principles & patterns are used in the right way
- Building using frameworks, etc.
- In essence, application architecture is inherently about software design and usually covers only a single technology stack (e.g. .Net, etc).

Enterprise Architecture

What it is not…

- An architecture for "enterprise" systems or systems that involve components that are touted as enterprise-level.

- Generally refers to the sort of work that happens at the CTO and CIO level.
- It looks at how to organize and utilize people, process and technology to make an organisation work effectively and efficiently.
- Doesn't necessarily look at technology in any detail.

The enterprise architect role is therefore extremely wide-reaching, being enterprise-wide, and requires careful inspection of all the business' functions and their strategic requirements.

How design relates to architecture

- All architecture is design but not all design is architecture.

"Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change"

- Grady Booch

- Architectural design decisions are separated from other design decisions by how expensive a mistake would be.

What you are thinking as a developer…

- Most of our focus is placed on the code. Here, we're thinking about
  - object oriented principles,
  - classes,
  - interfaces,
  - inversion of control,
  - refactoring,
  - automated unit testing and
  - the countless other practices that help us build better software.

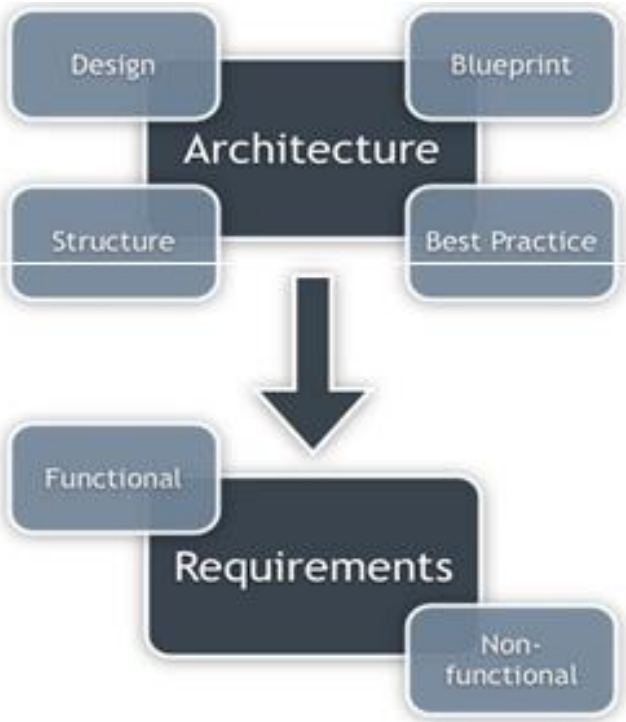
    - If you have a team of people that are only thinking about this, then who is thinking about the other stuff?

The other stuff …

- Cross-cutting concerns such as logging and exception handling.
- Security, such as authentication, authorization and confidentiality of sensitive information.
- Performance, scalability and availability.
- Audit and other regulatory requirements.
- Interoperability with other systems in the environment.
- Operational and support requirements.
- Consistency of structure and approach across the code base.

Architecture helps you meet your requirements

- Architecture is one of the important means by which your project's requirements are met.

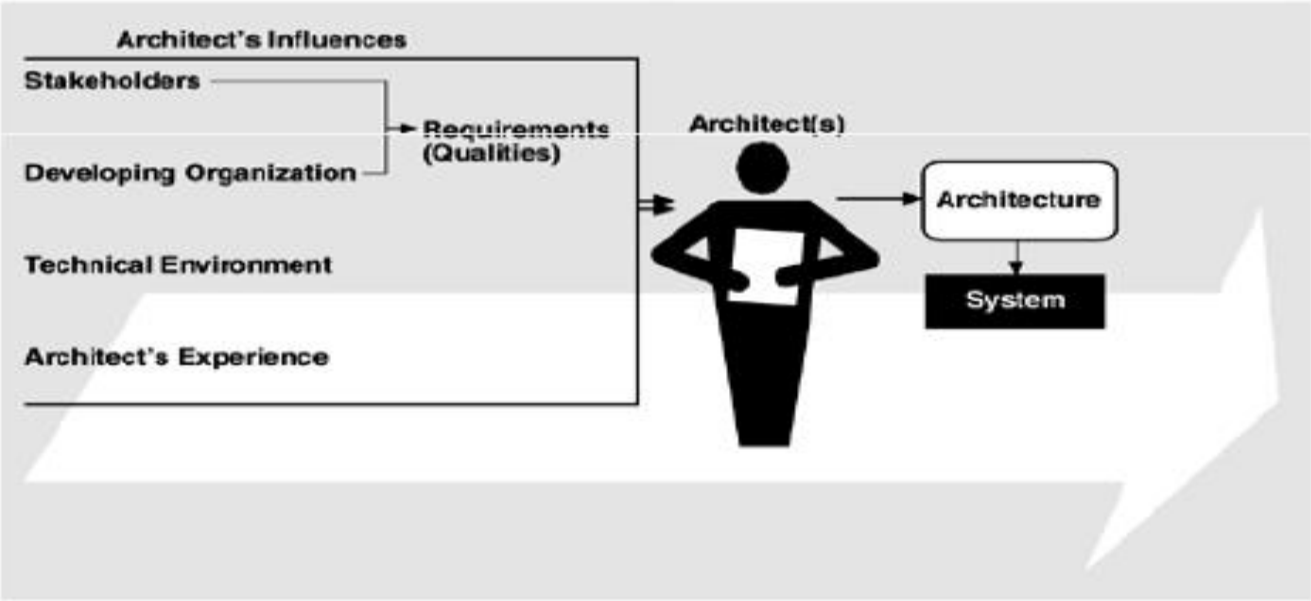


Where do architectures come from?

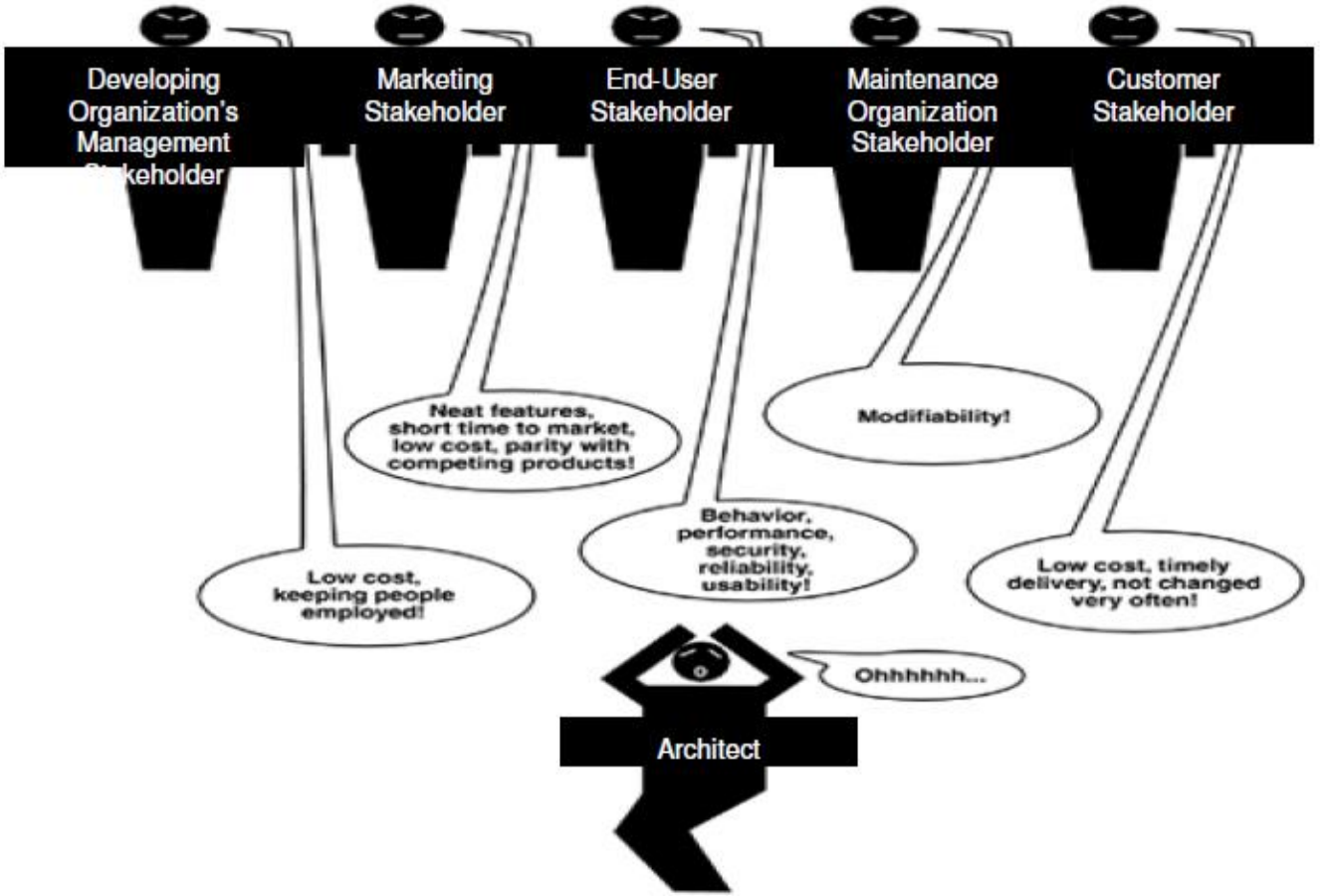
- An architecture is the result of a set of business and technical decisions
- Architecture varies based on real-time situations
  - An application for which the deadlines are tight
  - An application for which the deadlines can be easily satisfied
  - An architect designing a non-real-time system
  - An architect designing a system today vs. designed five years ago.

Factors that influence architecture

- SYSTEM STAKEHOLDERS
- THE DEVELOPING ORGANIZATION
- THE BACKGROUND AND EXPERIENCE OF THE ARCHITECTS
- THE TECHNICAL ENVIRONMENT



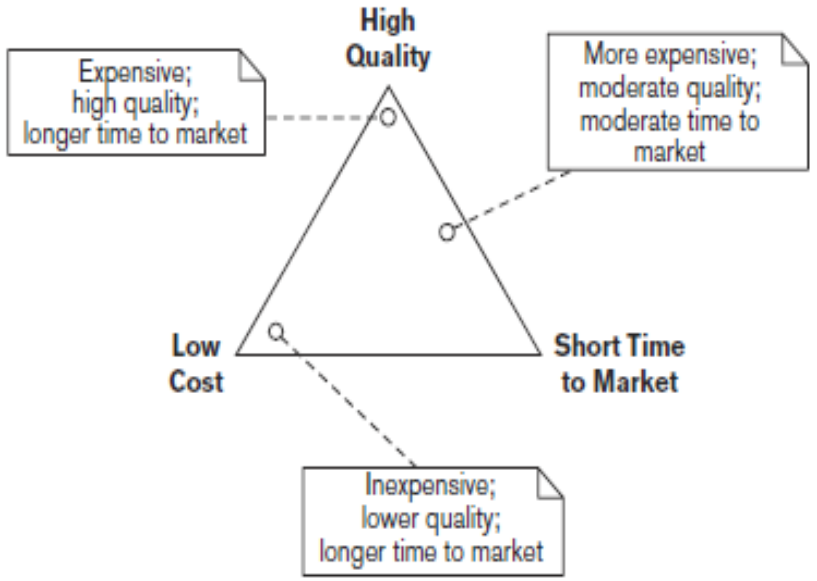
- Architectures are influenced by system stakeholders



StakeHolders

- Software Systems have to be ***built*** and ***tested***, *they have to be* ***operated***, *they may have to be* ***repaired***, *they are* usually ***enhanced*** , ***deployed ,Versionised*** *and of course they have to be* ***paid*** *for. Each of these activities* involves a number—possibly a significant number—of people in addition to the users..
- ***A stakeholder*** *in a software architecture is a person, group, or entity with an* ***interest in or concerns*** *about* ***the realization*** *of the architecture****.***
- ***A concern*** *about an architecture is a requirement, an objective, an intention, or an aspiration a stakeholder has for that architecture.*

Quality Triangle (Example)



THE QUALITY TRIANGLE

Architectures are influenced by system stakeholders

- performance,
- reliability,
- availability,
- platform compatibility,
- memory utilization,
- network usage,
- security,
- modifiability,
- usability, and Interoperability with other systems and behavior

- Does your requirements document capture all the quality requirements?

The Importance of Stakeholders

- Stakeholders (explicitly or implicitly) drive the whole shape and direction of the architecture
- Stakeholders ultimately make or direct the fundamental decisions about scope, functionality, operational characteristics, and structure of the eventual product or system—under **the guidance of the architect**

PRINCIPLE Architectures are created solely to meet stakeholder needs.

PRINCIPLE A good architecture is one that successfully meets the objectives, goals, and needs of its stakeholders.

Developing Organization

- Structure or nature of the development organization.
  - Organization has an abundance of idle programmers skilled in client-server communications
- Staff skills are one additional influence.
- There are three classes of influence that come from the developing organization:
  - Immediate business
  - Long-term business
  - Organizational structure.

The background and experience of an architect

- Previous experience
  - Architects have had good results or disastrous experiences using a particular architectural approach.
- Architectural choices may also come from an architect's
  - education and training,
  - exposure to successful architectural patterns, or
  - exposure to systems that have worked particularly poorly or particularly well.
- The architects may also wish to experiment with an architectural pattern or technique learned recently.

Technical Environment

- The environment that is current when an architecture is designed will influence that architecture.
- Standard industry practices or
- Software engineering techniques prevalent in the architect's professional community.

Software architecture vs. Software Development

- The line between software development and software architecture is a tricky one.
  - Simply an extension of the design process
  - It is all abstractions and do not get bogged down by those pesky implementation details.

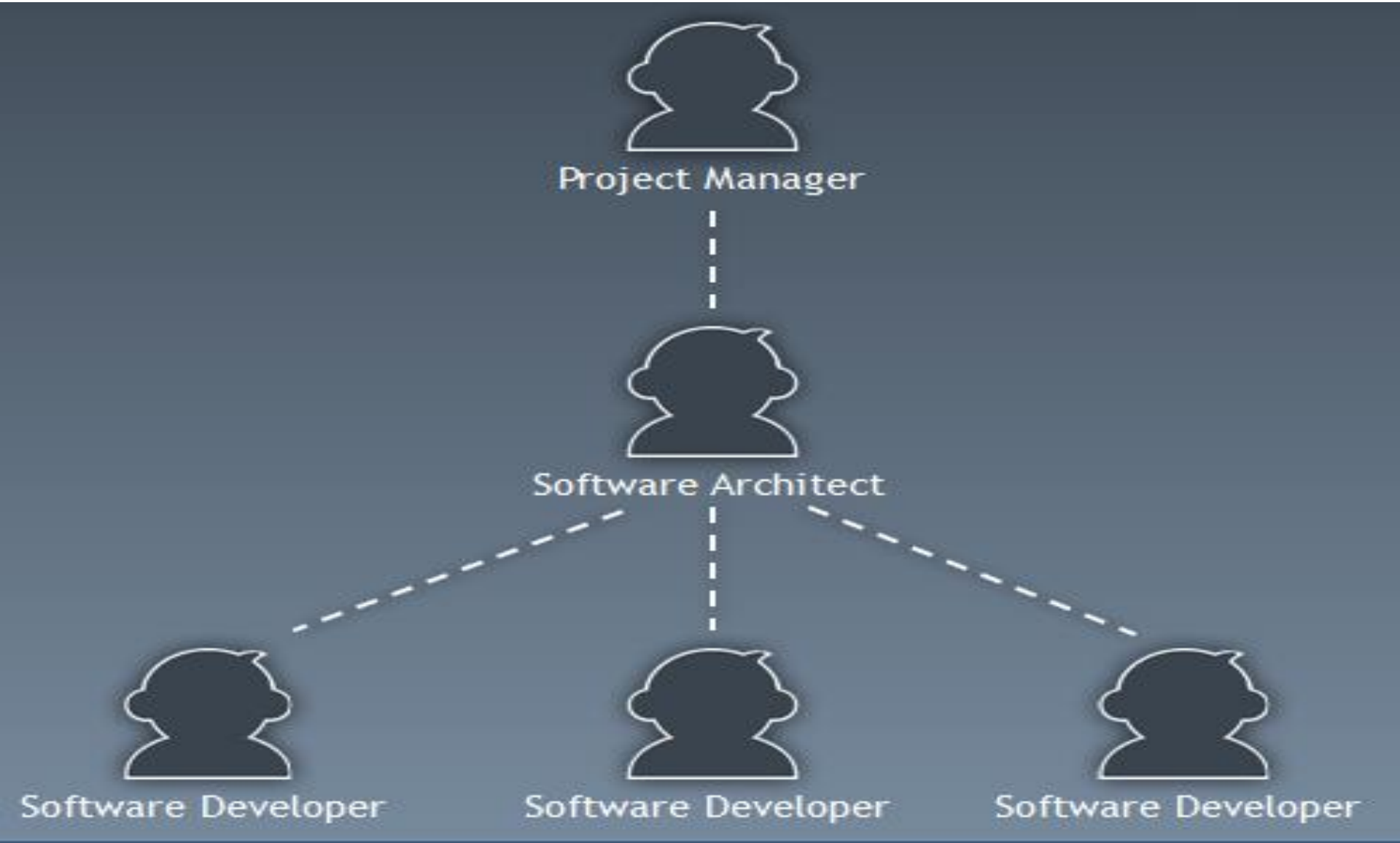
An Architect Perspective

- Software Architecture's perspective is different from Software development
  - An increase in scale
  - An increase in the level of abstraction
  - An increase in the significance of making the right design decisions.
  - All about having a holistic view and seeing the bigger picture
    - Understand how the software system works as a whole

Who is an Architect

- It’s a role not a rank
- It's an evolutionary process
- You'll gradually gain the experience and confidence that you need to undertake the role.
- Does not simply happens overnight or with a Promotion
- What are the roles and responsibilities of an Architect ?

Software architect as team leader



Contd…

- Software Architecture is a post technical Carrier
- It's a technical leadership role
  - somebody needs to steer the ship continuously
- Software Architect Role and Coding ?
  - Does Software architect Need to Code ?

Architect and Coding

- A Rockstar Software Engineer can become good Architect
  - Who is the **RockStar** Software Engineer ?
- Software Architect must be a master builders
  - **Coding** is a great way to retain these skills
- Don't code all of the time

Software Architect and Soft Skills

- What sort of softskills does a software architect need

Software Architect

Leadership

Communication

Influencing

Negotiation

Collaboration

Coaching and Mentoring

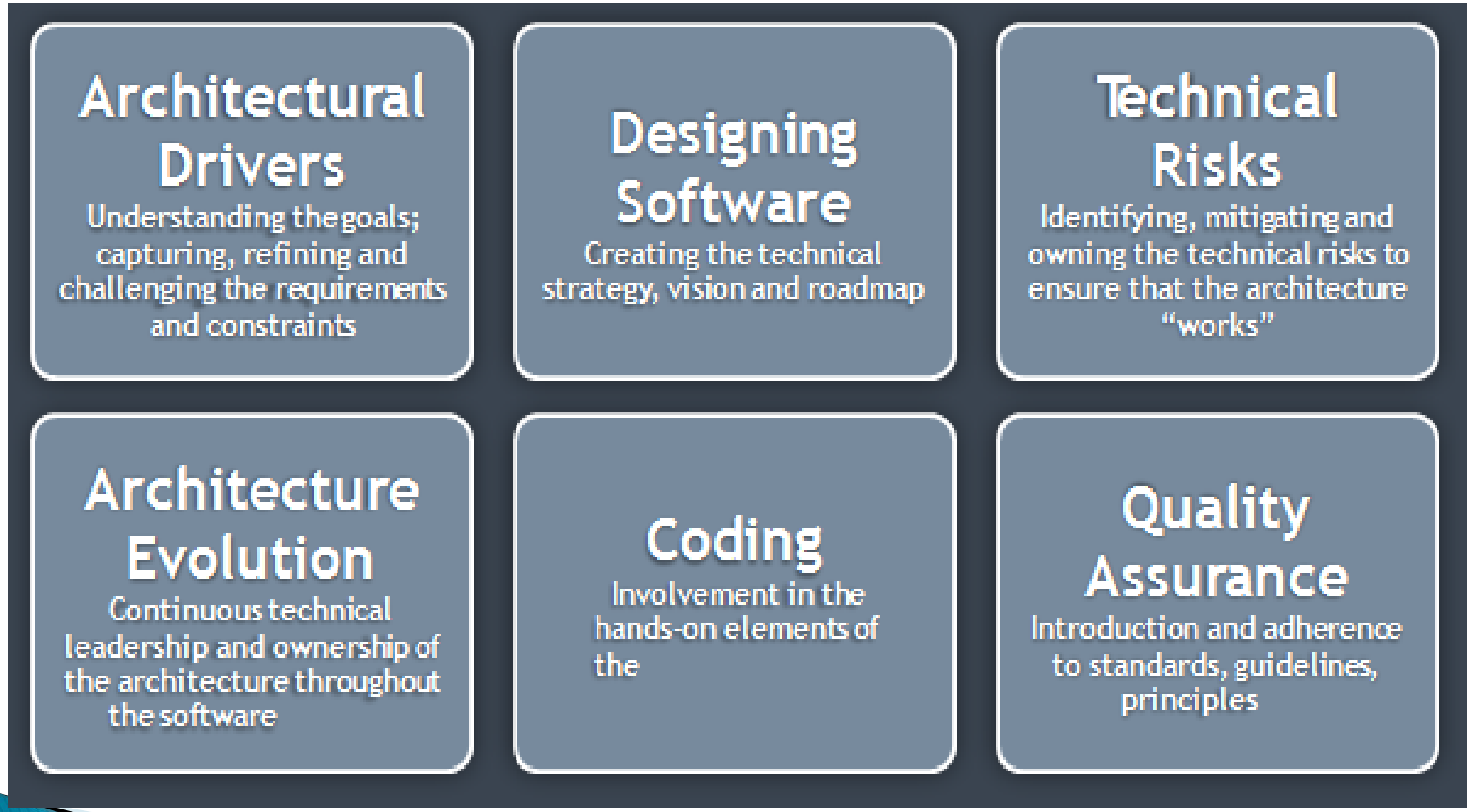
Motivation

Facilitation

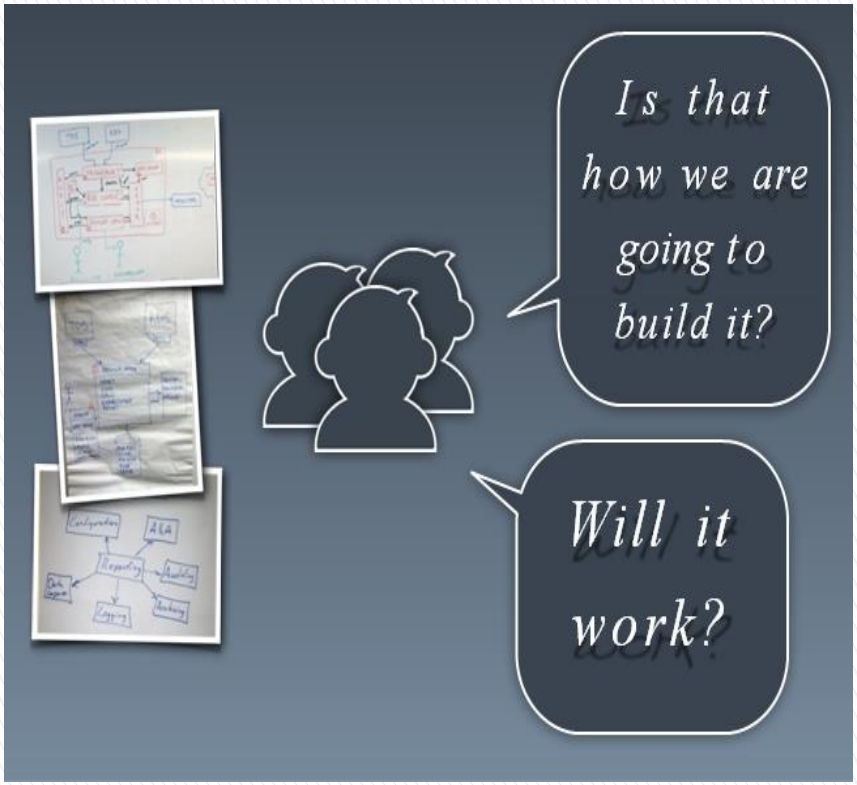
Political

How do you learn to deal with people?

Technical Leadership



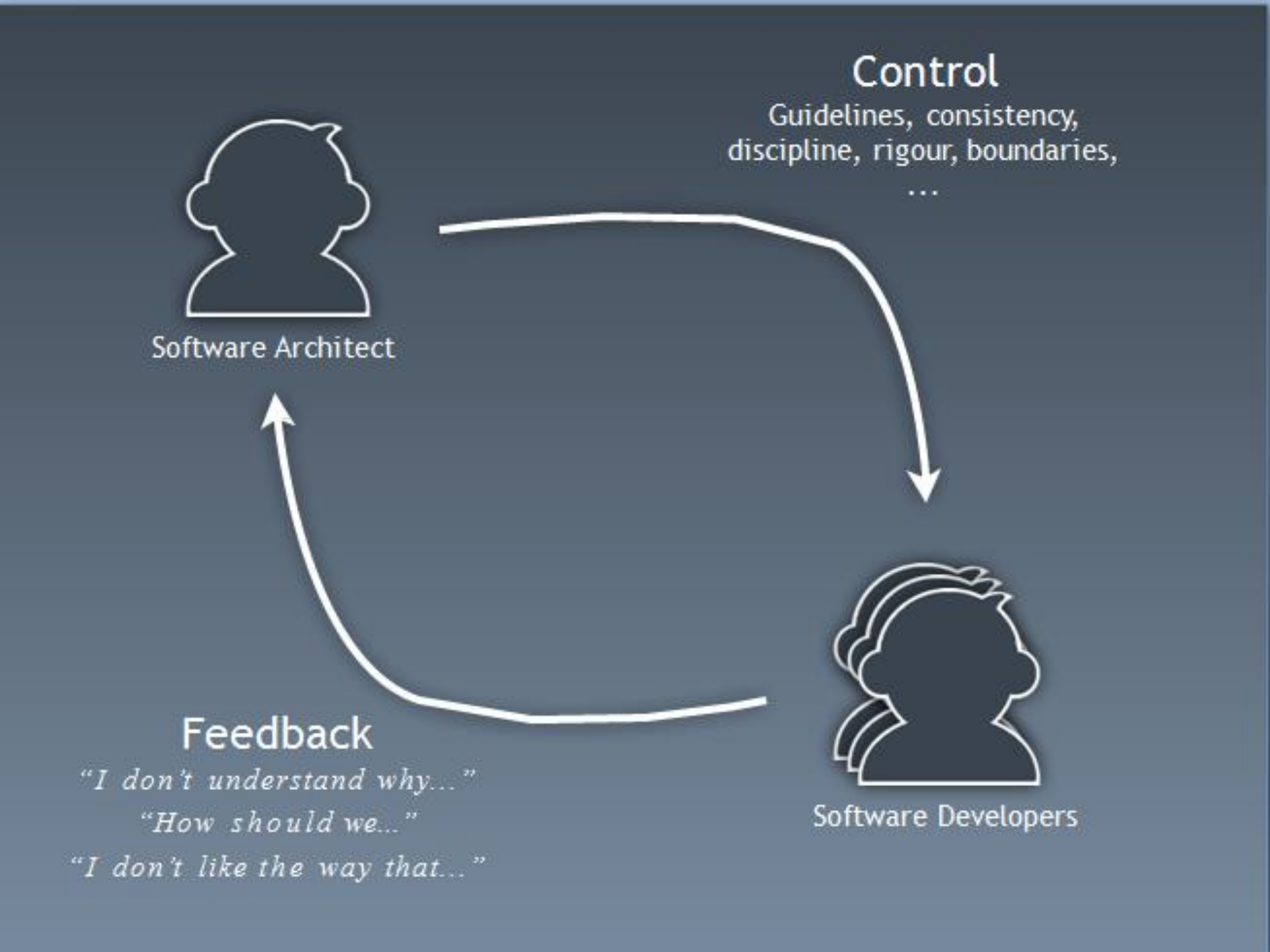
Softare Architect as a Controller

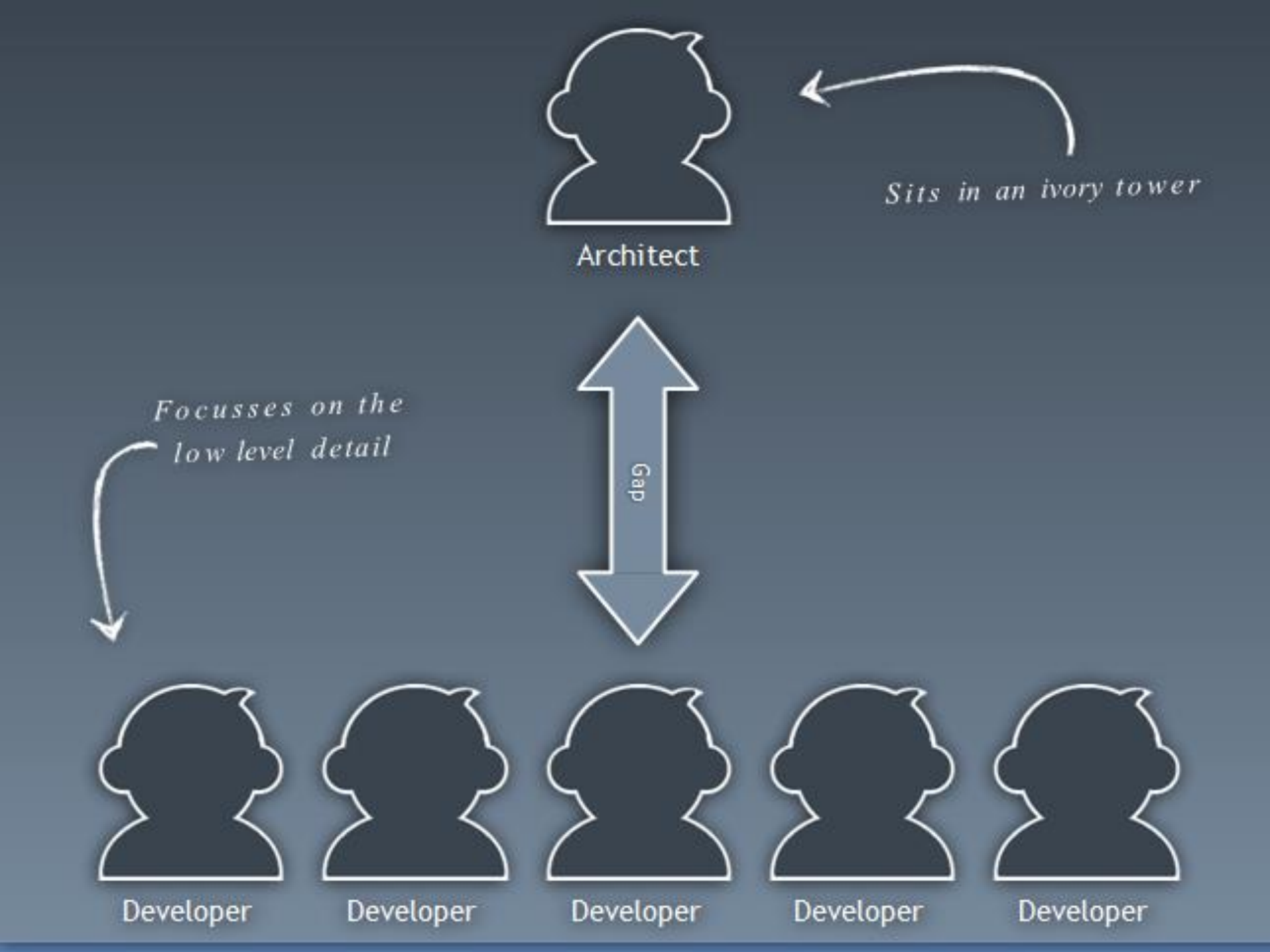


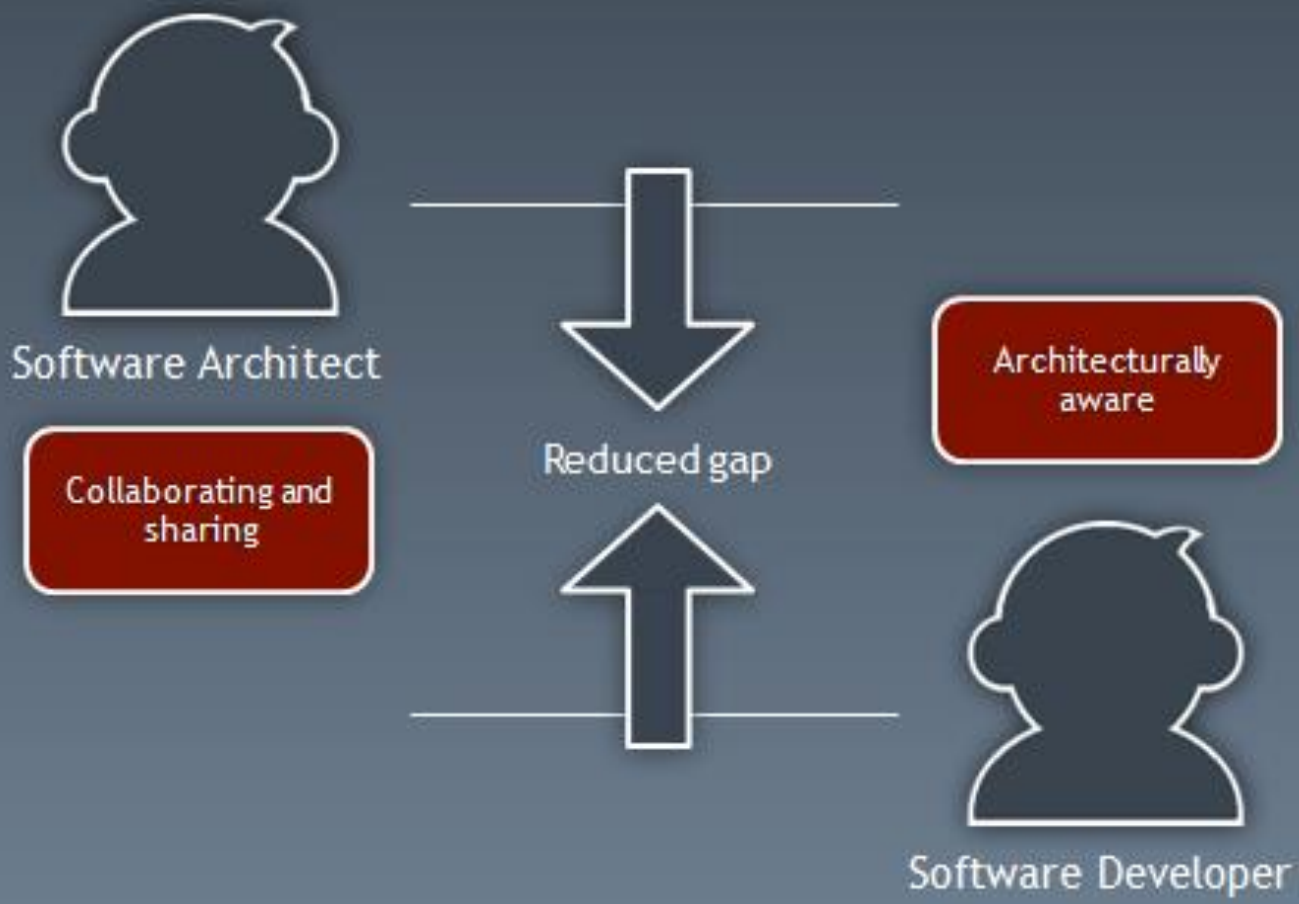
- Does your team members understand what they are building and how they are building ?

Agree on Something's

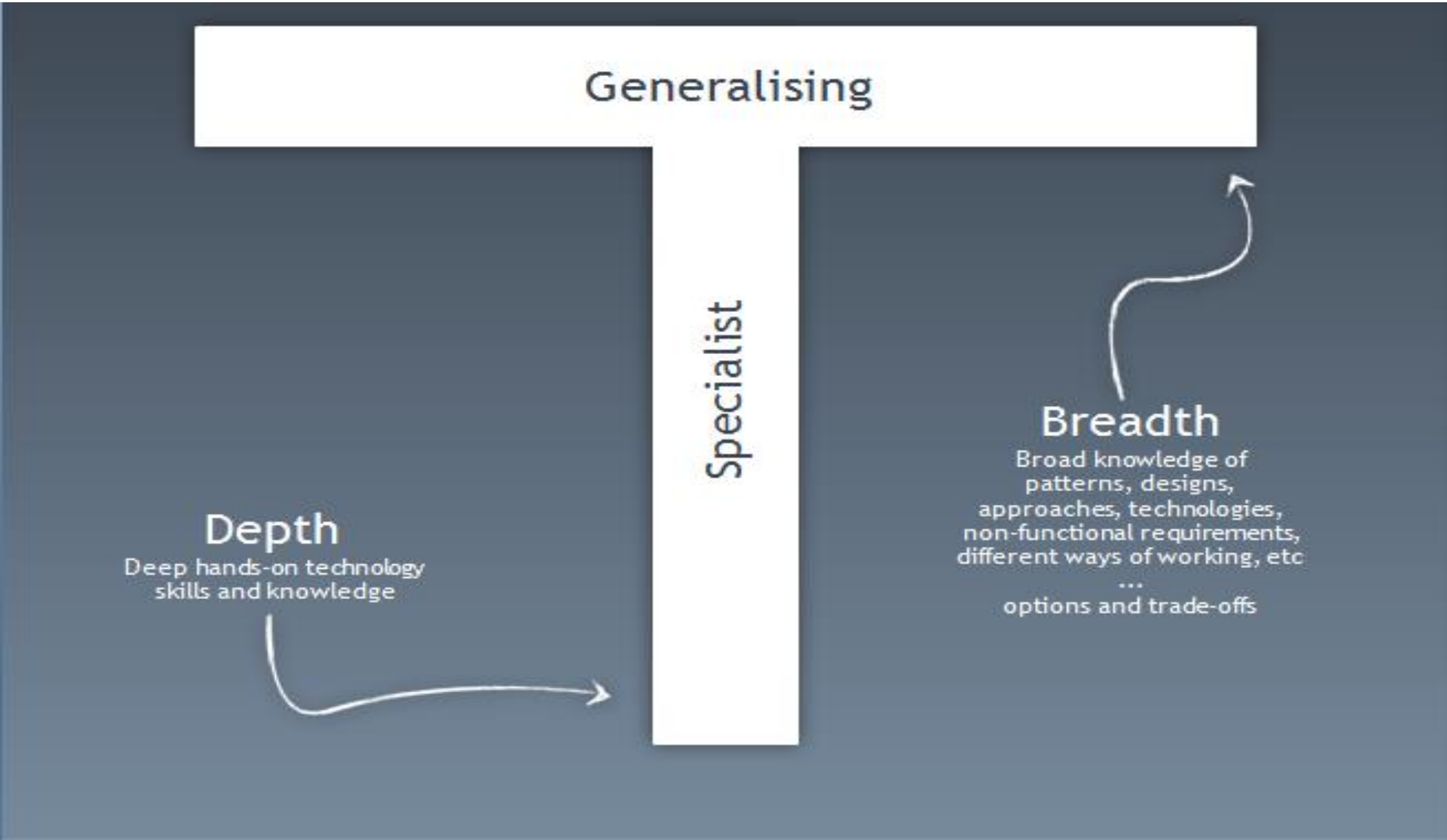
Put Boundaries and Guidelines in place


Control
Guidelines, consistency, discipline, rigour, boundaries, ...
Software Architect
Software Developers
Feedback
"I don't understand why..."
"How should we..."
"I don't like the way that..."


Architect
Sits in an ivory tower
Focusses on the low level detail
Gap
Developer
Developer
Developer
Developer
Developer


Software Architect
Collaborating and sharing
Reduced gap
Architecturally aware
Software Developer

Its not a Rank, It's a Role



Conclusion

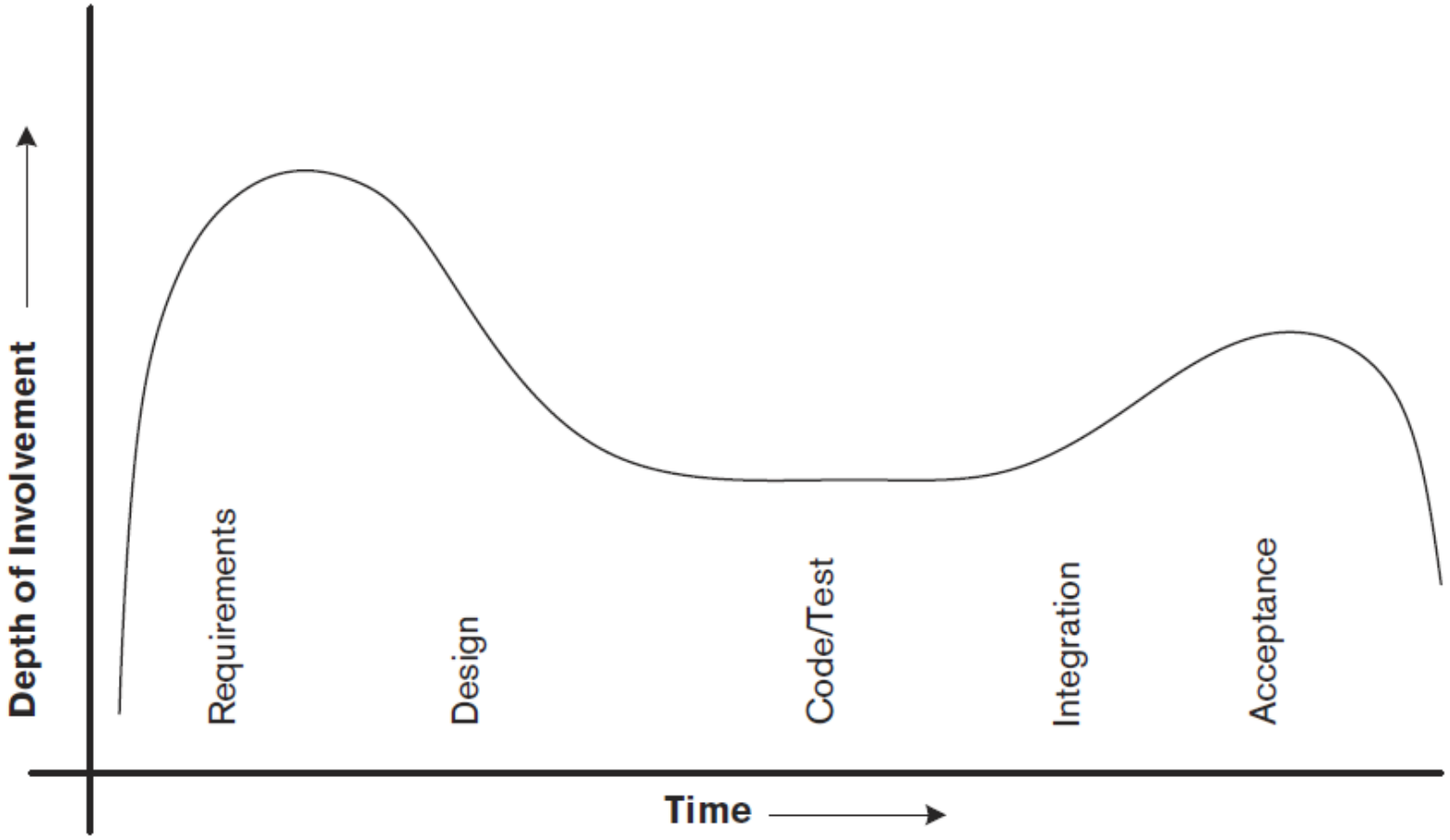
A software architect looks after the **big picture**

It's a **technical leadership** role that includes collaboration and coaching

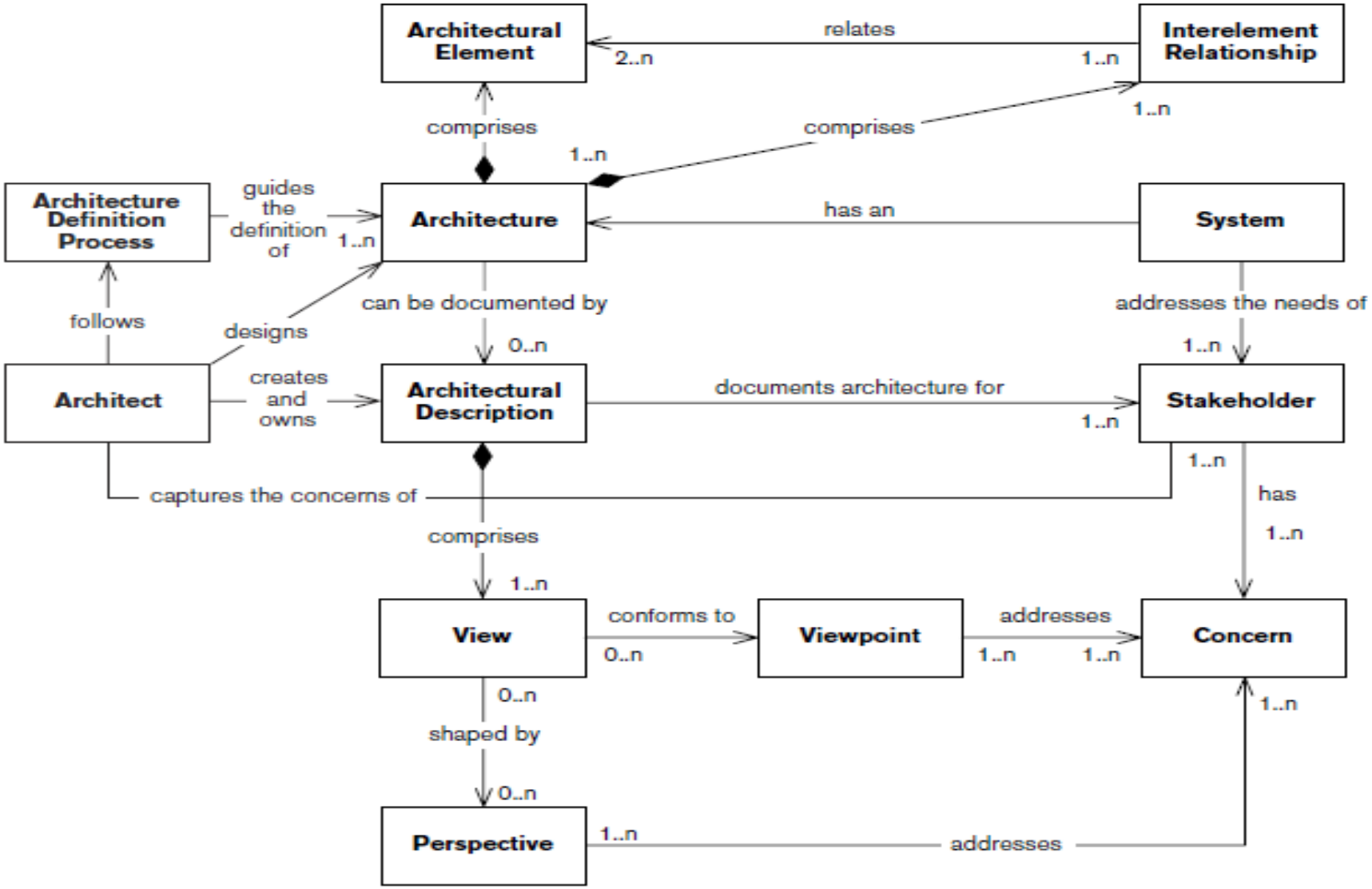
Software architects need technical depth and breadth plus good **soft skills**

Ideally architects should `code`, but they should certainly be a **master builder**

The Architects Involvement



ARCHITECTURE DEFINITION AND THE ARCHITECT IN CONTEXT



The role

- Responsibility 1
- Responsibility 2
- Responsibility 3
- ...
- Task 1
- Task 2
- Task 3
- ...

How would you

define

the software architect role in your team?

Architectural Description

- What are the main functional elements of your architecture?
- How will these elements interact with one another and with the outside world?
- What information will be managed, stored, and presented?
- What physical hardware and software elements will be required to support these functional and information elements?
- What operational features and capabilities will be provided?
- What development, test, support, and training environments will be provided?

Views

PRINCIPLE It is not possible to capture the functional features and quality properties of a complex system in a single comprehensible model that is understandable by and of value to all stakeholders.

AD is partitioned into a number of separate but interrelated **views,** each of which describes a separate aspect of the architecture

STRATEGY A complex system is much more effectively described by using a set of interrelated views, which collectively illustrate its functional features and quality properties and demonstrate that it meets its goals.

DEFINITION A **view** is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders.

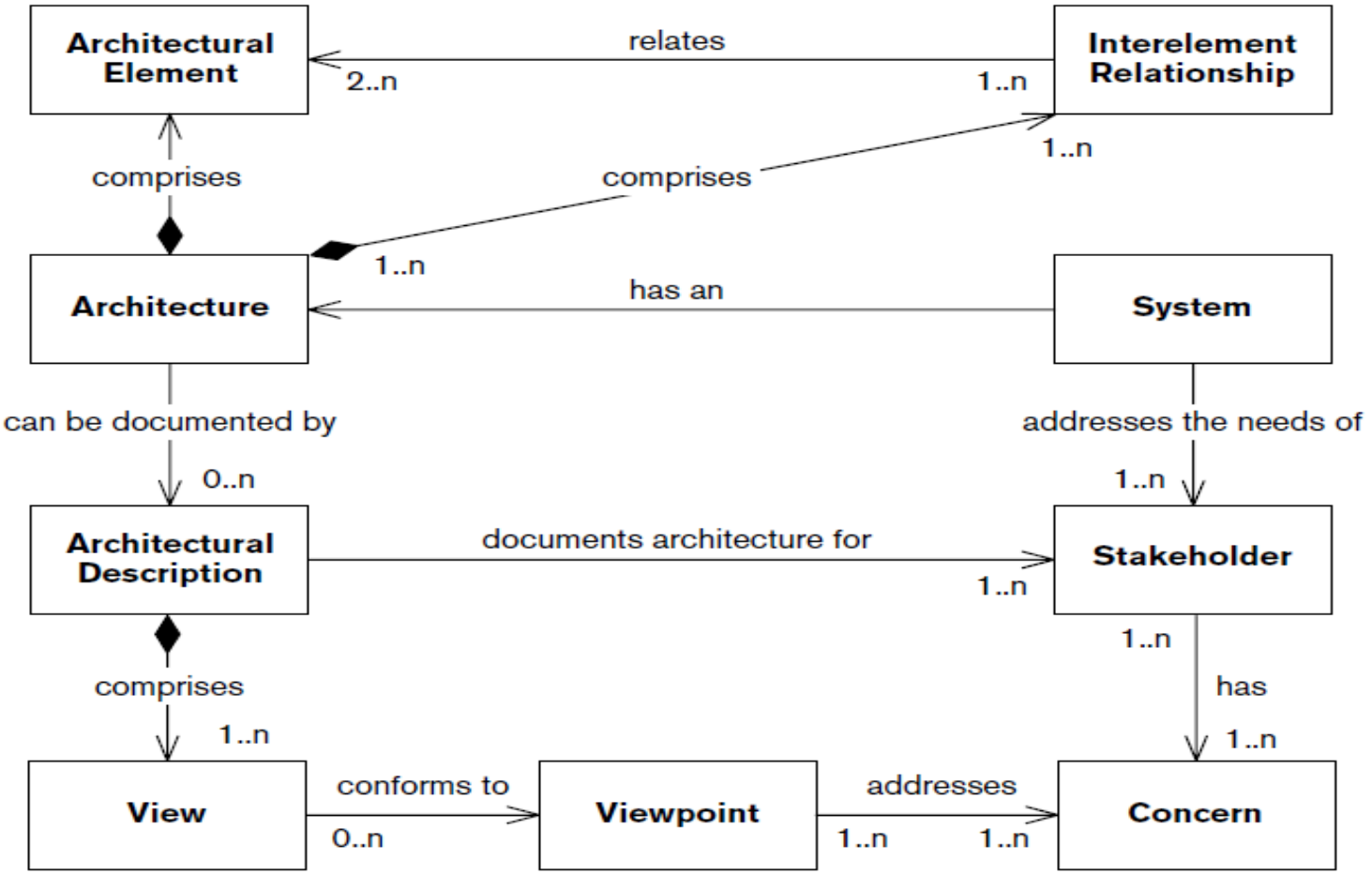
ViewPoints

- A **viewpoint is a collection of patterns, templates, and conventions** for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views.
- Define the content of each architectural view

View and ViewPoint

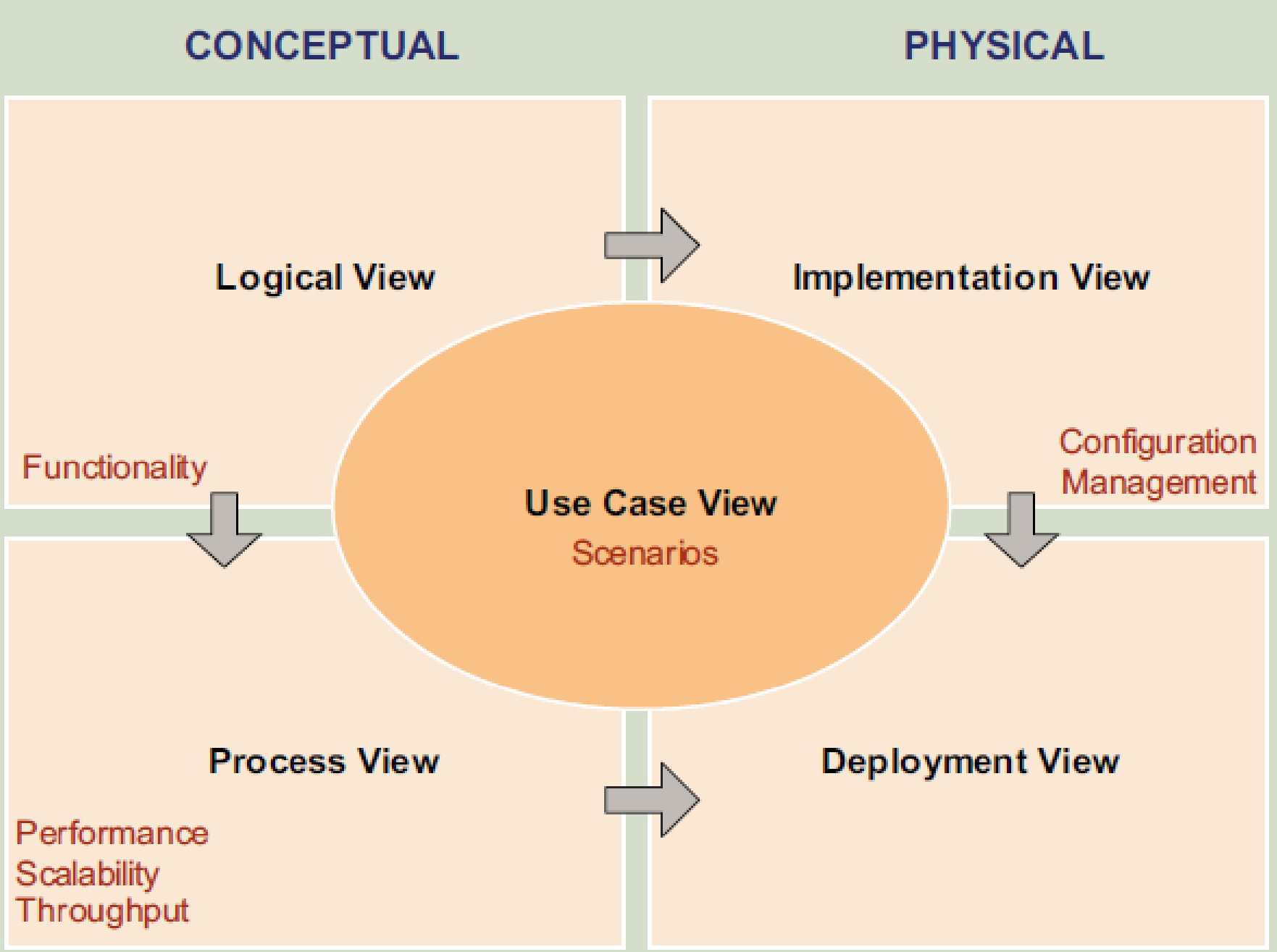
- A viewpoint is a way of looking System
- A view is what you see
- A viewpoint defines a conventions(such as notations , languages , models) for constructing a certain kind of view
- A view is the result of applying a viewpoint to a particular system of interest
- View:ViewPoint :: Program : Program Language

View and ViewPoint in Context



4+1 View

- The 4+1 View Approach is an ‘architecture style’ to organize an application’s architecture representations into views to meet individual stakeholder’s needs
- The fundamental organization of a software system can be represented by:
  - Structural elements and their interfaces that comprise or form a system
  - Behavior represented by collaboration among the structural elements
  - Composition of Structural and Behavioral elements into larger subsystems


CONCEPTUAL
PHYSICAL
Logical View
Implementation View
Functionality
Use Case View
Scenarios
Configuration
Management
Process View
Deployment View
Performance
Scalability
Throughput

Logical View (Object Oriented Decomposition)

- This view focuses on realizing an application's functionality in terms of structural key elements, key abstractions and mechanisms, separation of concerns and distribution of responsibilities.

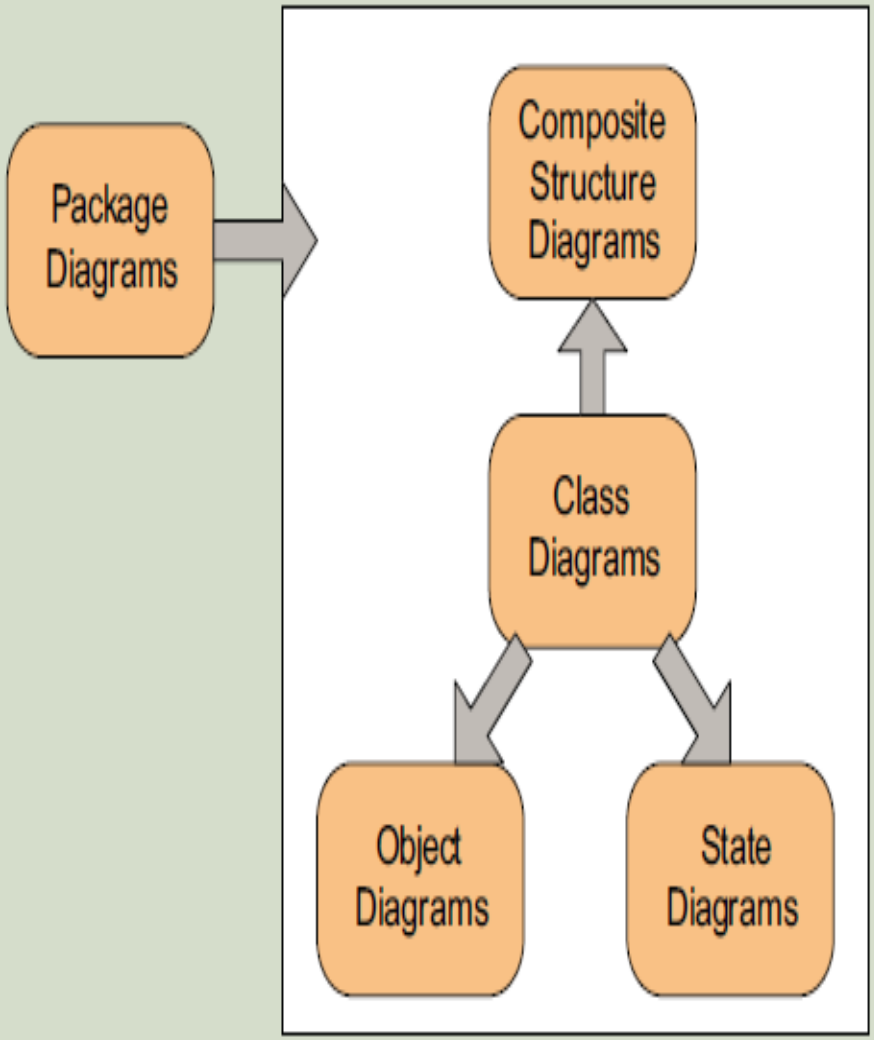
1. Vertical and horizontal divisions

   – The application can be vertically divided into significant functional areas (i.e., order capture subsystems, order processing subsystems).

   – Or, it can be horizontally divided into a layered architecture distributing responsibilities among these layers (i.e., presentation layers, services layers, business logic layers, and data access layers).

2. Representation of structural elements as classes or objects and their relationships.

Logical View and UML



1. Start with class diagrams to model the system
2. Use package diagrams to logically group diagrams

Optional use

3. Object diagrams when relationships between classes need to be explained through instances
4. State Charts when internal states of a specific class are to be explained
5. Composite Structures when parts of a class and relationships between parts are to be modeled

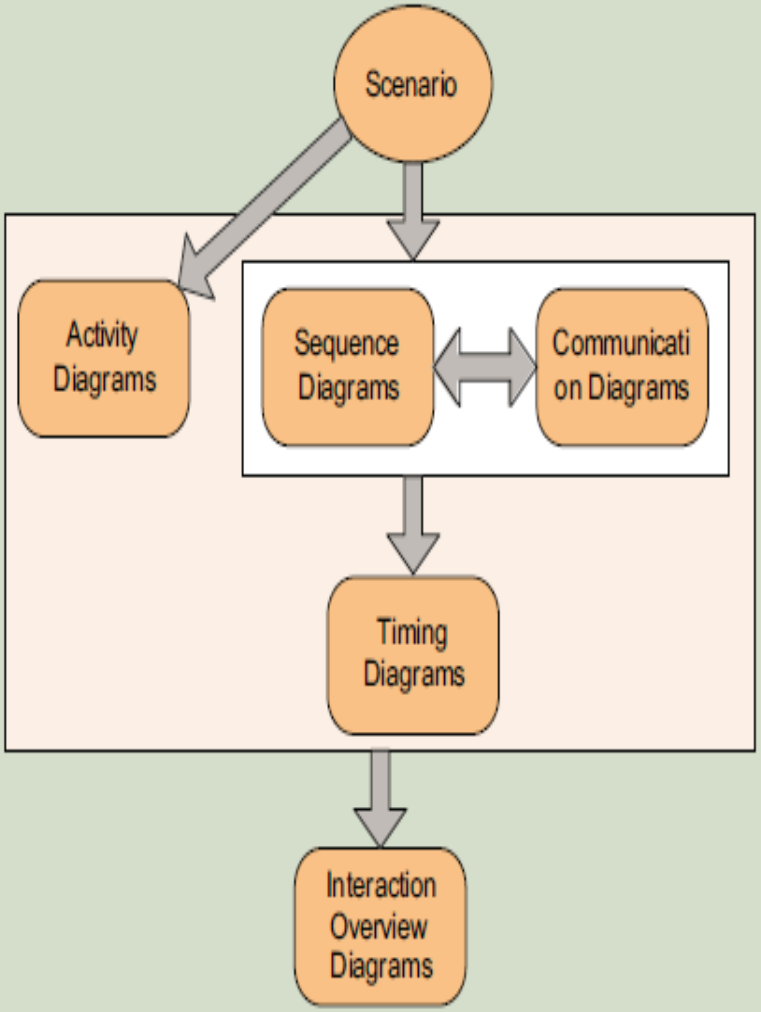
Implementation or Development View (Subsystem Decomposition)

- It should show dependencies and relationships between modules, how modules are organized, reuse, and portability.
- This is a view of a system's architecture that encompasses the components used to assemble and release a physical system
- This view focuses on configuration management and actual software module organization in the development environment
- **Component Diagrams** are used to represent the Implementation View
- The development view is primarily for **developers** who will be building the modules and the subsystems.

Process View (Process Decomposition)

- This view considers non-functional aspects
- Designing the whole system and then integrating the subsystems or the system into a system of systems
- This view shows tasks and processes that the system has, interfaces to the outside world and/or between components within the system

Process View and UML



1. Use either Sequence or Communication Diagrams for modeling simple interactions in use case realizations

Optional use

2. Add Activity diagrams to realize scenarios where business logic is a sequence of actions and involves branching and parallel processing
3. Add timing diagrams when modeling for performance
4. For complex scenarios, that can be composed of other scenarios, use Interaction overview diagrams

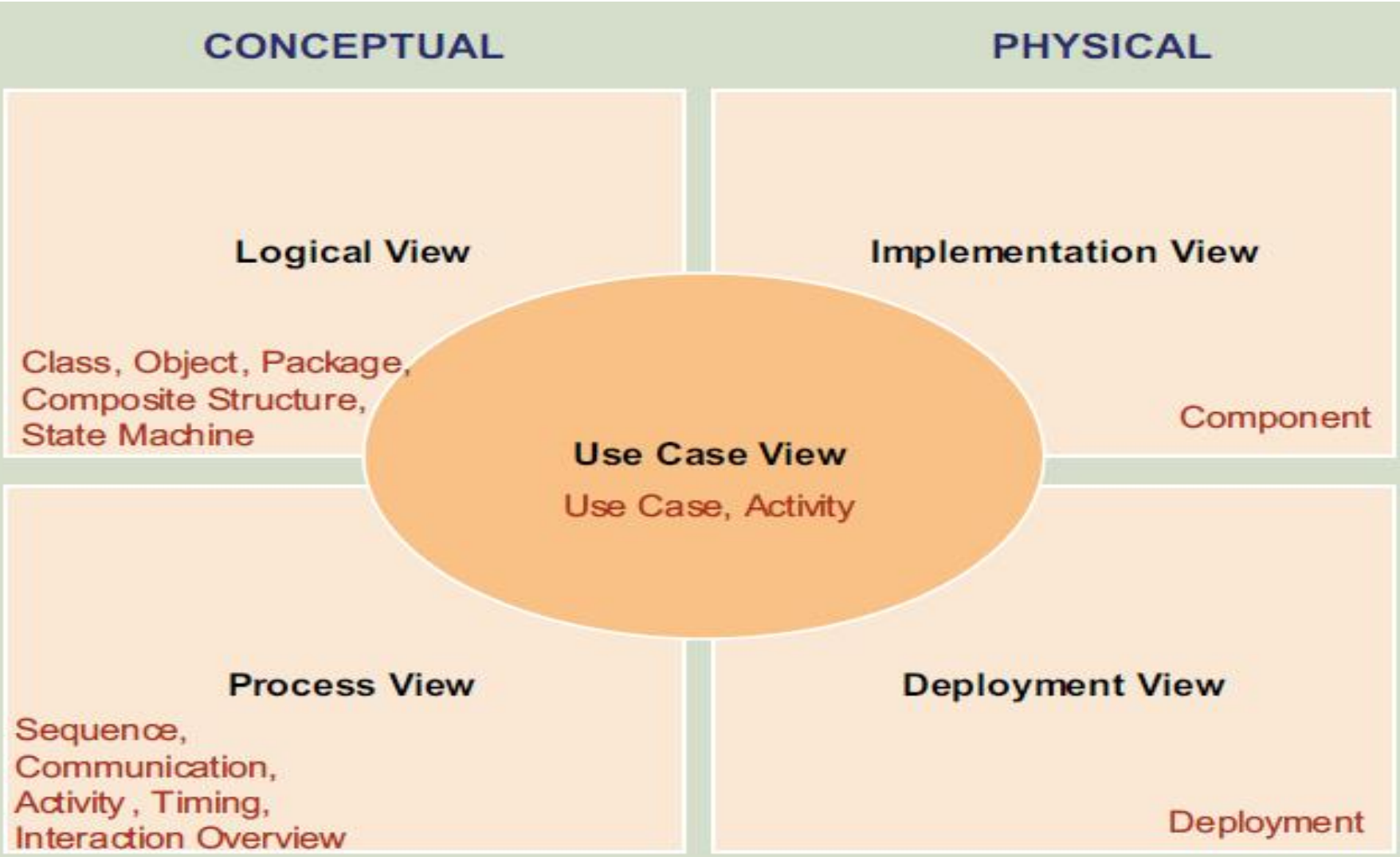
Deployment or Physical View (Mapping Software to Hardware)

- This view encompasses the nodes that form the system's hardware topology on which the system executes
- It focuses on distribution, communication and provisioning
- This view provides all possible hardware configurations, and maps the components from the Implementation View to these configurations
- The physical view is primarily for **system designers** and **administrators** who need to understand the physical locations of the software, physical connections between nodes, deployment and installation, and scalability
- **Deployment Diagrams** show the physical disposition of the artifacts in the real-world setting

Use Case View or Scenarios (putting all together)

- The scenarios help to capture the requirements so that all the stakeholders understand how the system is intended to be used.
- The Use Case View encompasses the use cases that describe the behavior of the system as seen by its end users and other stakeholders
- This view represents the scenarios that tie the four views together
- Use UseCase Diagram to represent this view
- Use Activity Diagram to represent scenarios and business processes

The UML diagrams allocated to the views on the 4+1 View Model.



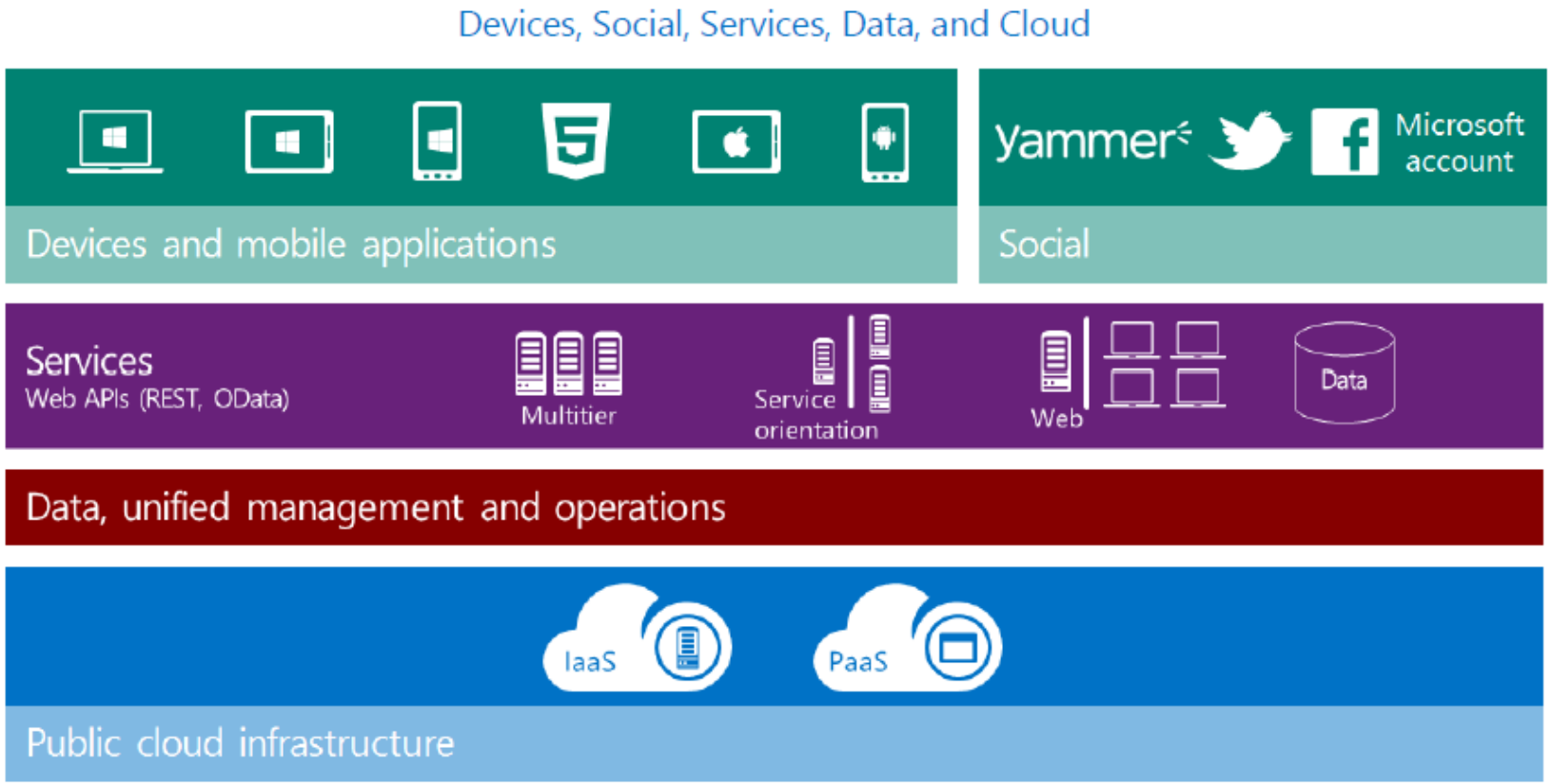
Relationships between Views

- The Logical View and the Process View are at a conceptual level and are used from analysis to design.
- The Implementation View and the Deployment View are at the physical level and represent the actual application components built and deployed.
- The Logical View and the Implementation View depict how functionality is modeled and implemented.
- The Process View and Deployment View realize the non-functional aspects using behavioral and physical modeling.
- Use Case View leads to structural elements being analyzed in the Logical View and implemented in the Development View. The scenarios in the Use Case View are realized in the Process View and deployed in the Physical View.

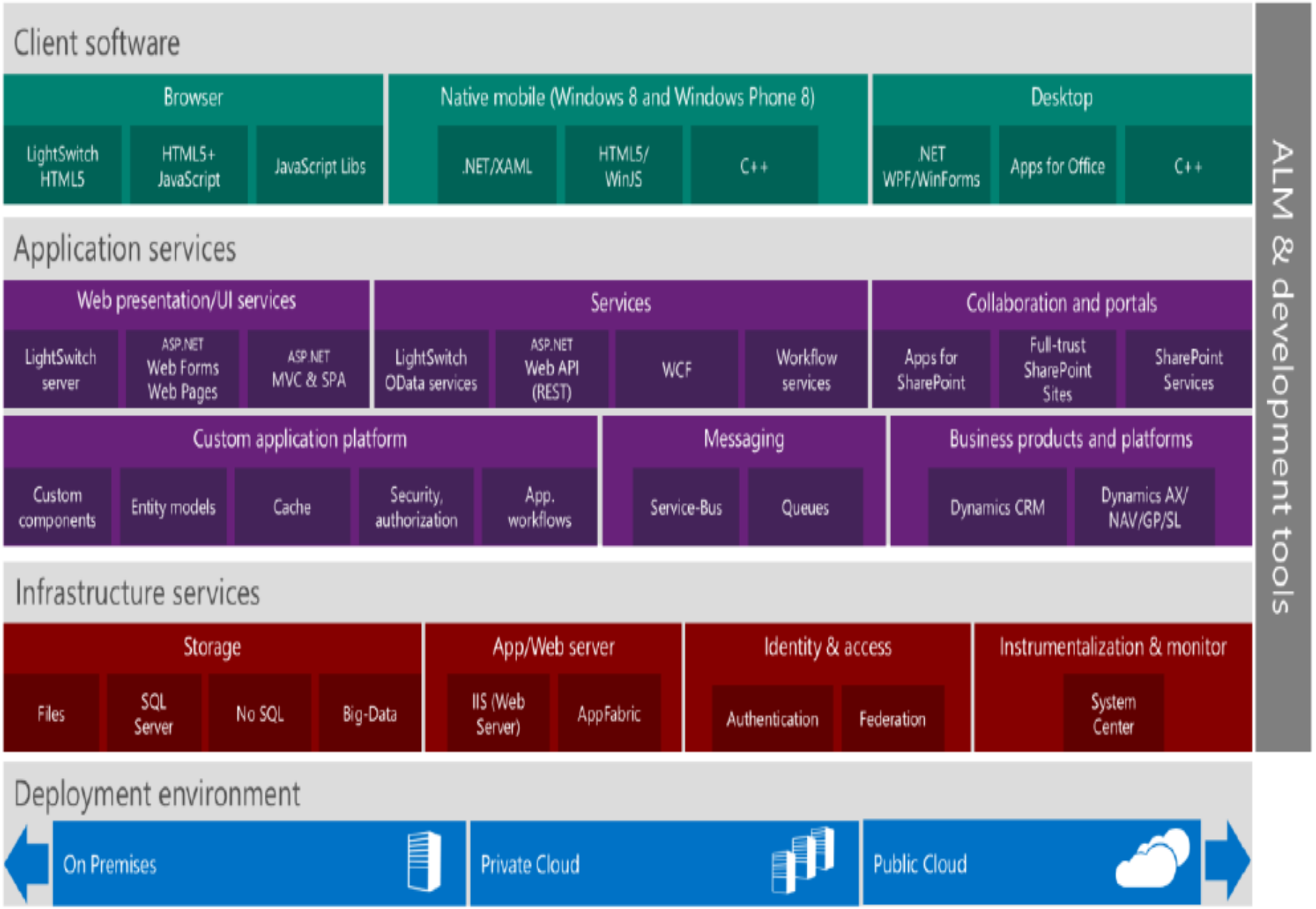
Relating Structures and Quality Attributes with Viewpoints

	Logical	Process	Physical	Development
Structure				
Module				✓
Conceptual (logical)	✓			
Process		✓		
Physical			✓	
Uses				✓
Calls	✓	✓		
Data flow		✓		
Control flow	✓	✓		
Class	✓			
Quality Attributes				
Performance		✓	✓	
Security			✓	
Availability			✓	
Functionality	✓			
Usability	✓	✓		
Modifiability	✓			✓
Portability	✓	✓	✓	✓
Reusability	✓	✓		✓
Integrability				✓
Testability	✓	✓		✓

APPLICATION LAYERS



Microsoft Development Platform Technologies



Scenario: End-to-end Small/Medium Web Business Application

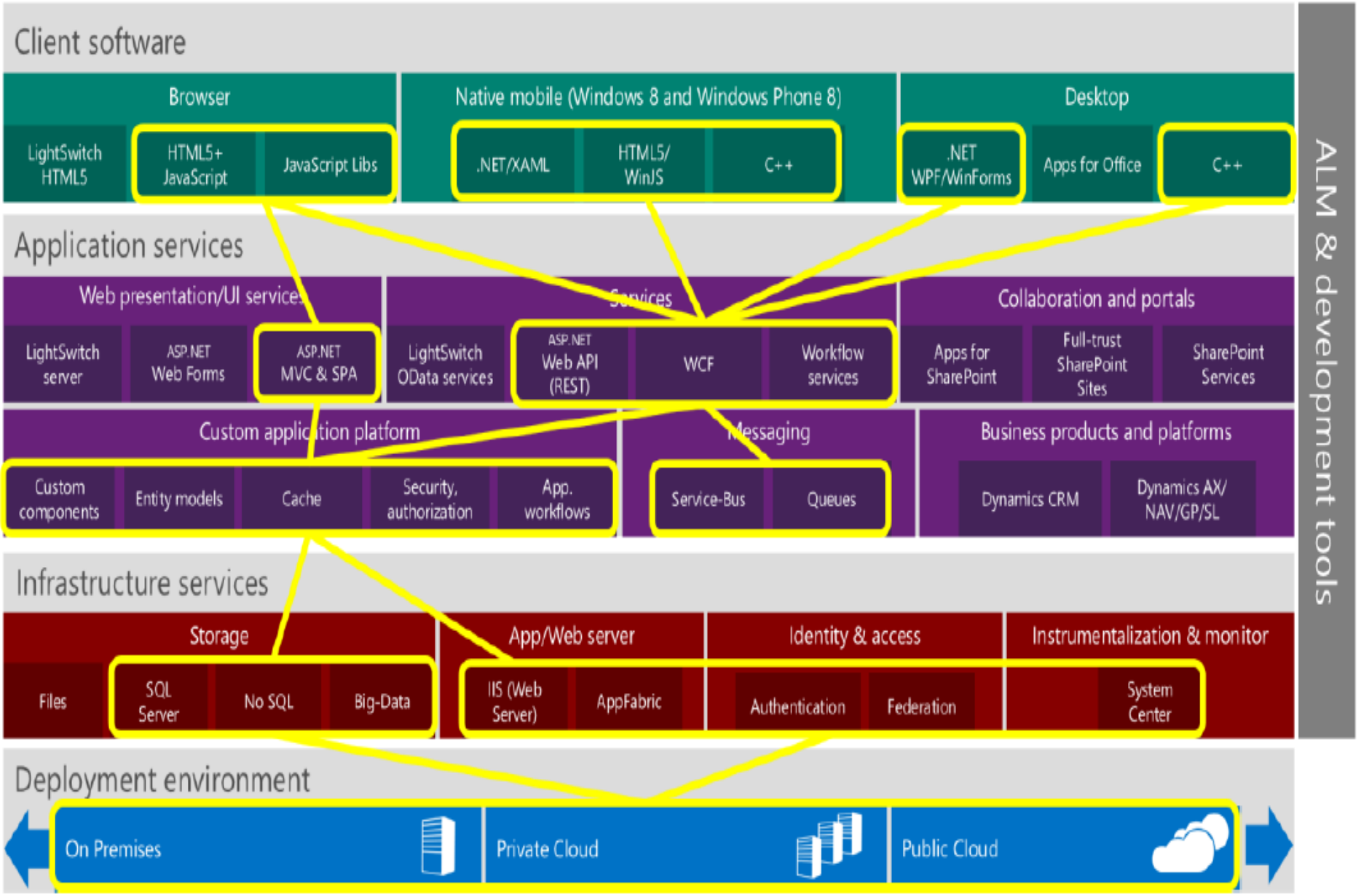
Scenario: Modern Web Applications for Any Mobile Device

Scenario: Small/Medium 2-Tier Desktop Application

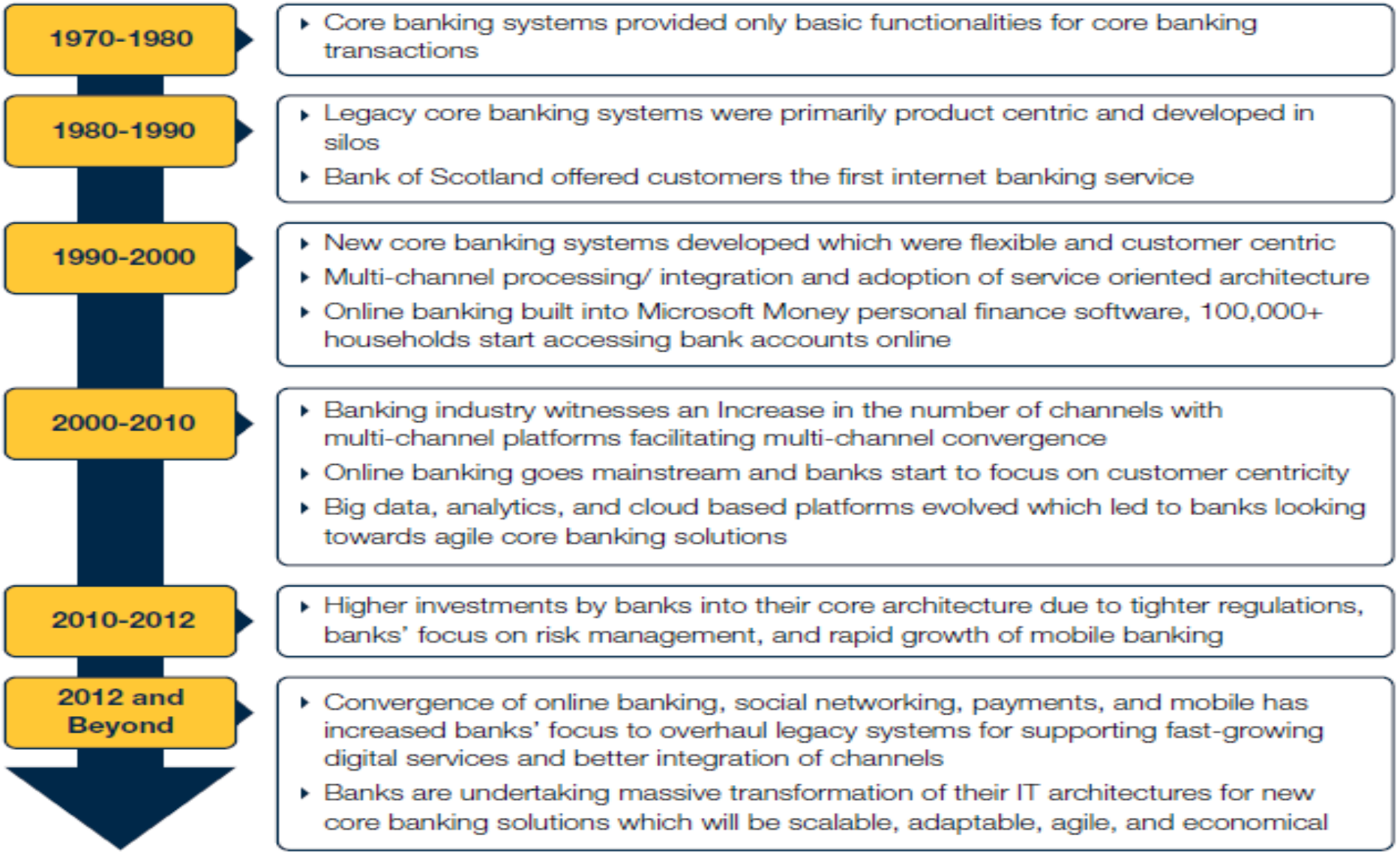
Scenario: Small/Medium 3-Tier Desktop Applications

Scenario: Large, Core-Business Applications

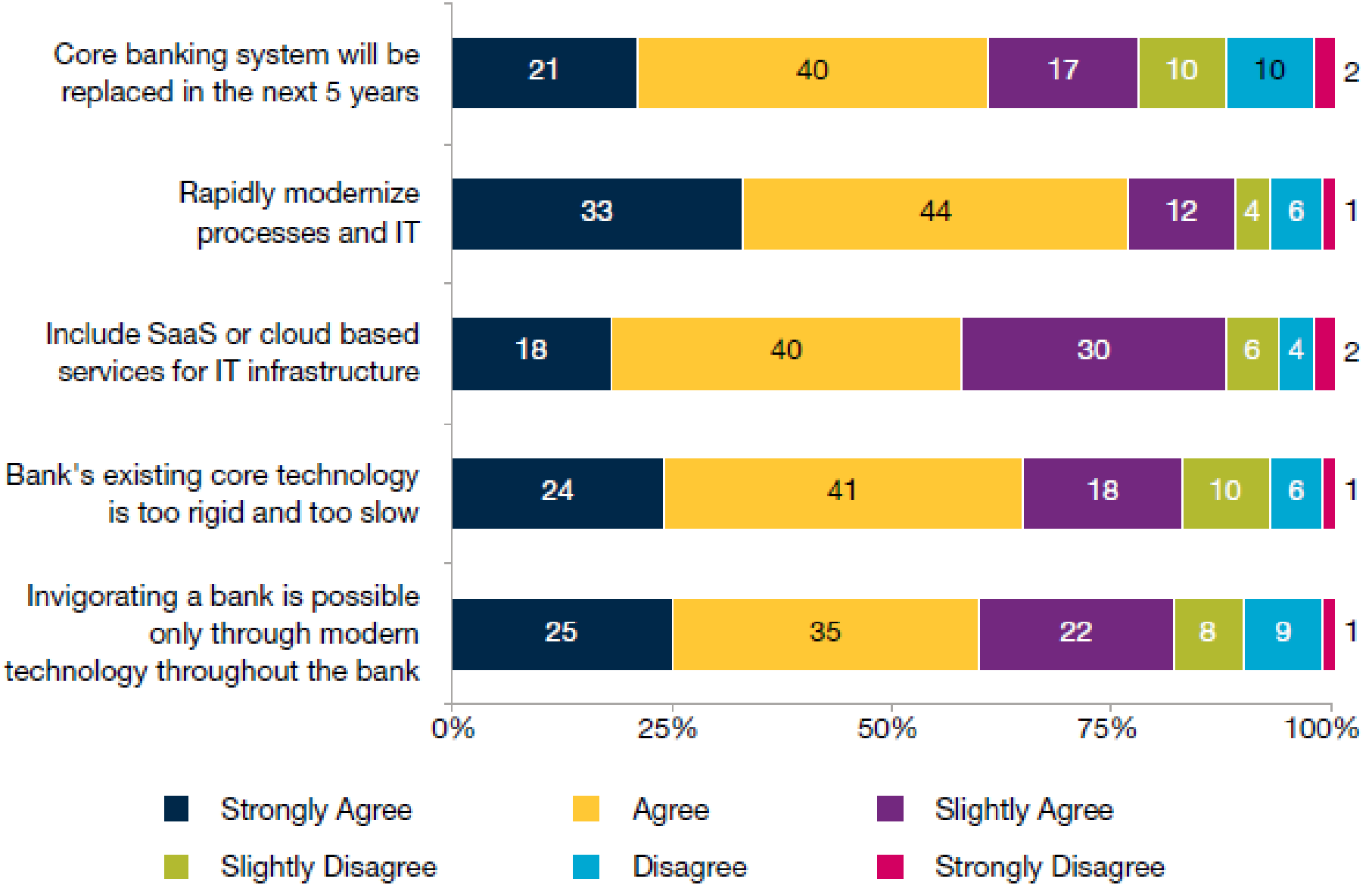
(Many subsystems)



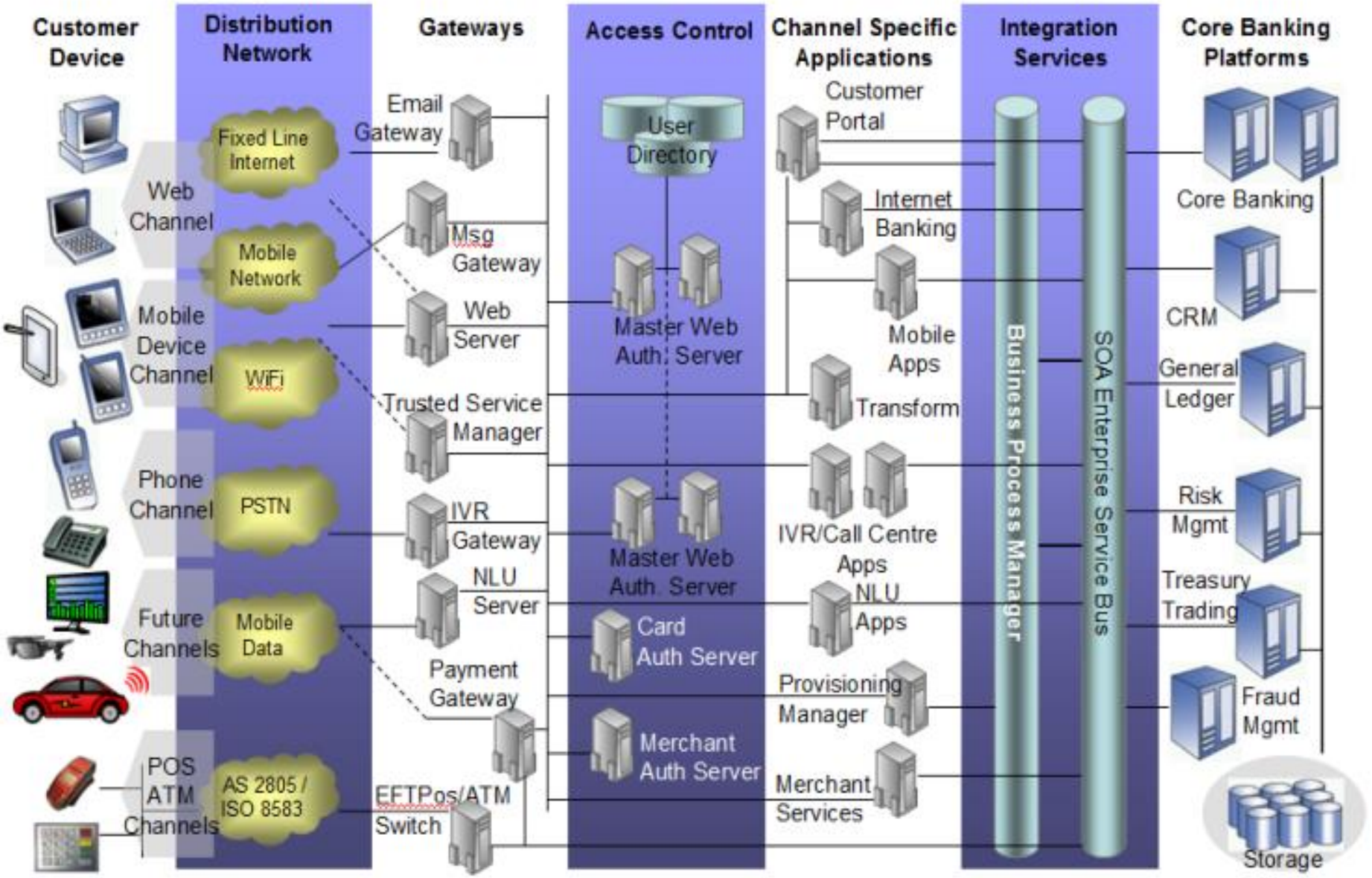
Evolution in Banking



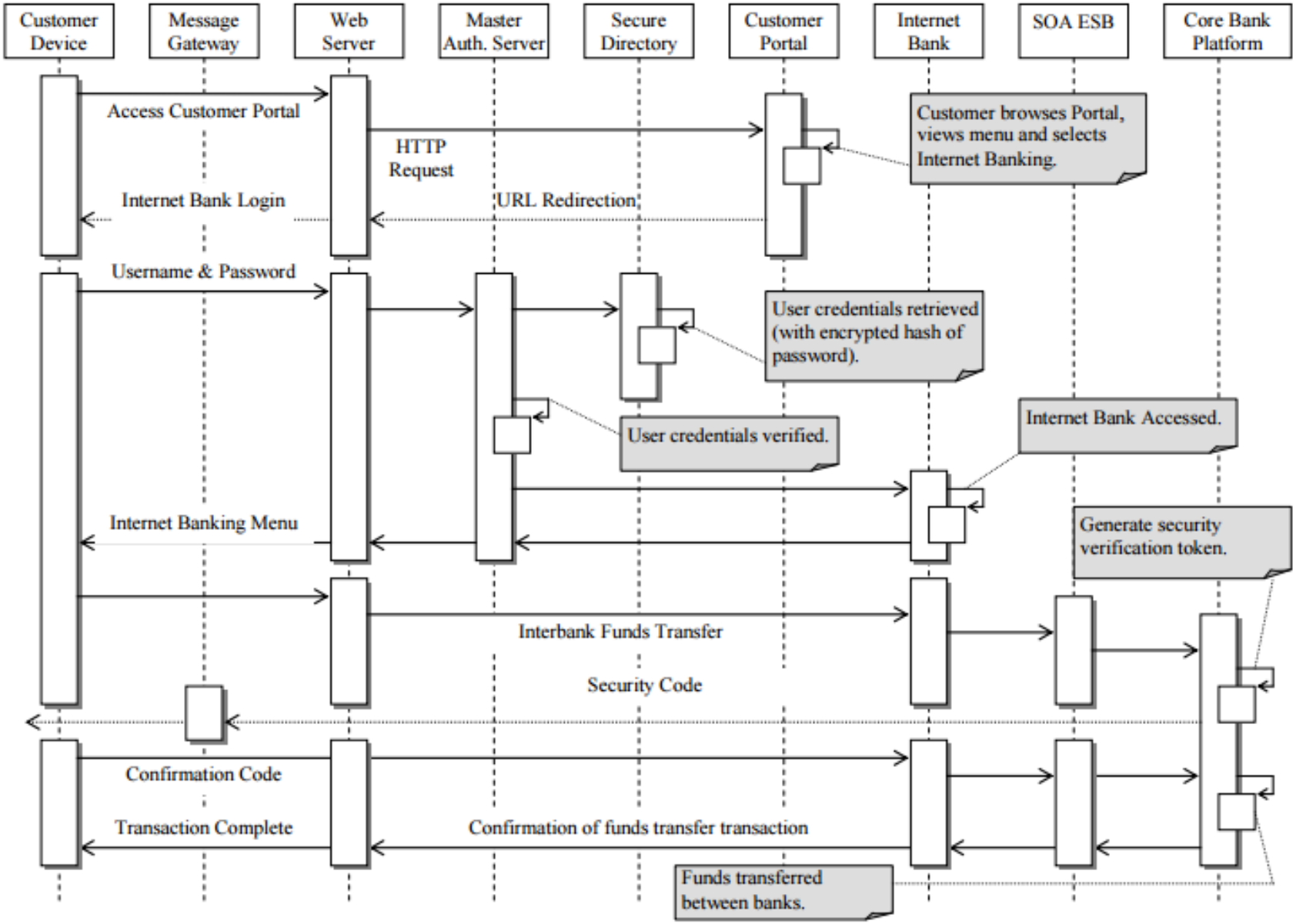
Banking Core Modernization Survey, 2015



Blueprint for Multi-channel Banking Architecture



Customer transfer funds to another bank using Web Channel



Customer applies for credit card limit increase using Mobile Device Channel

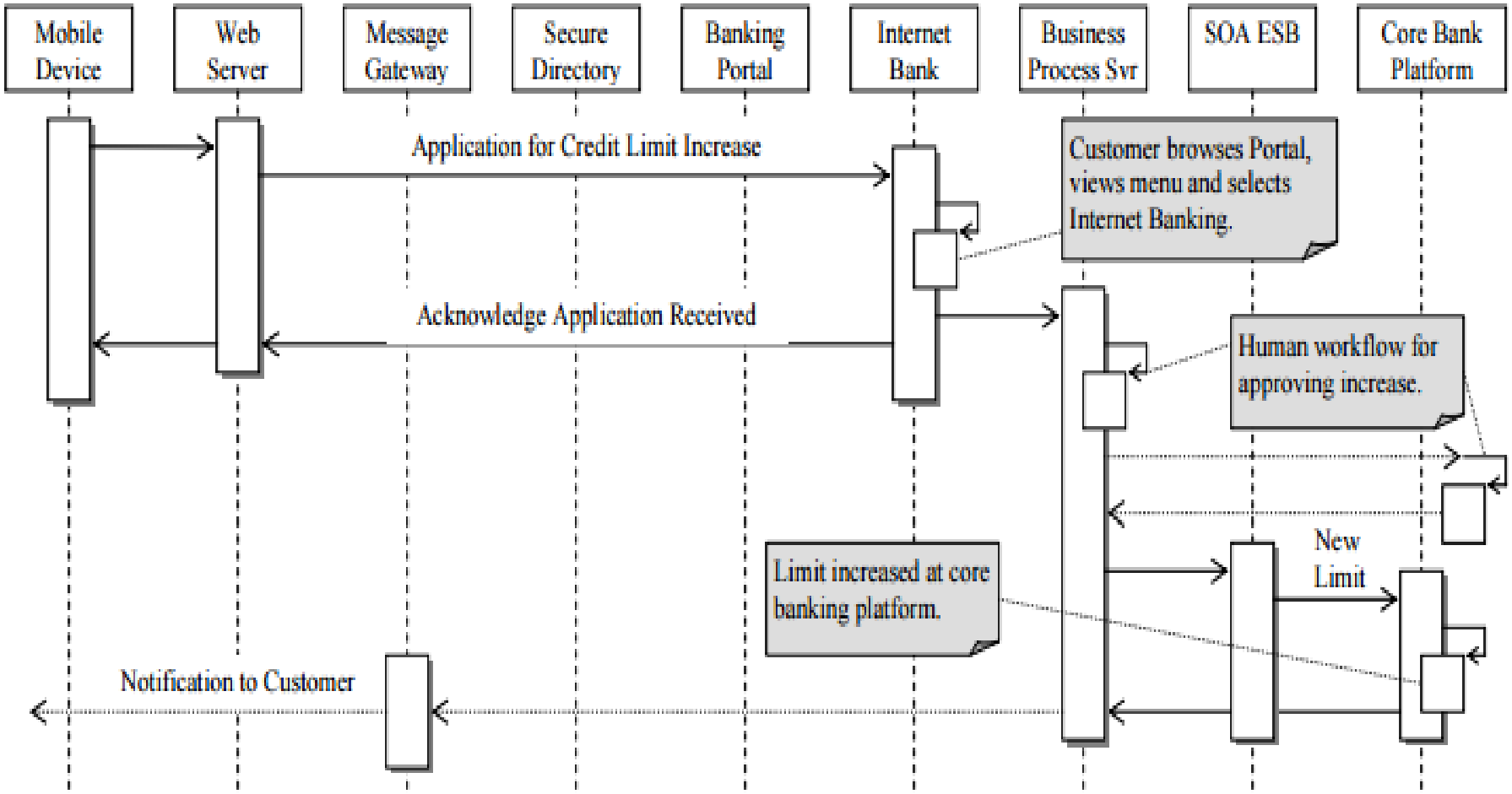
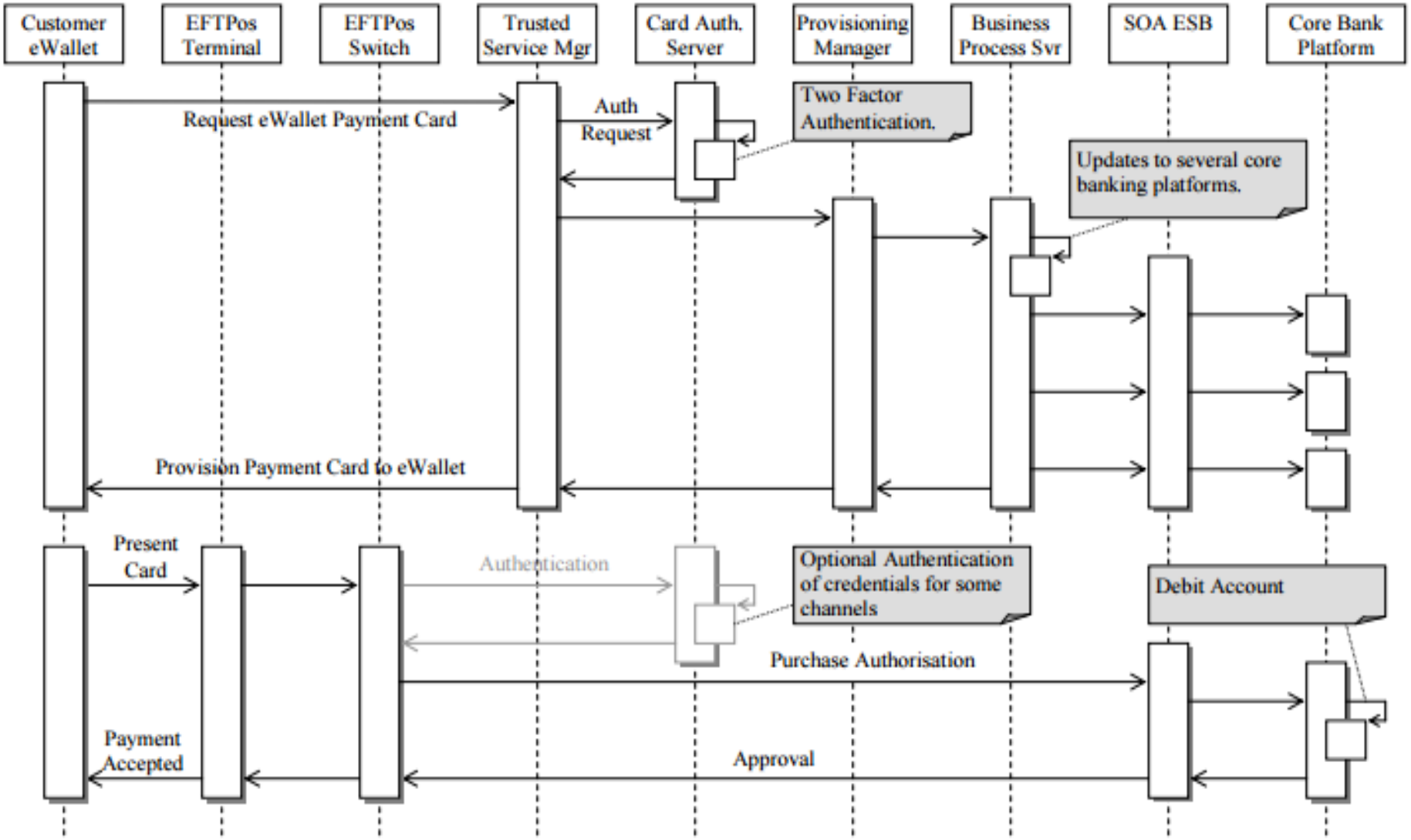
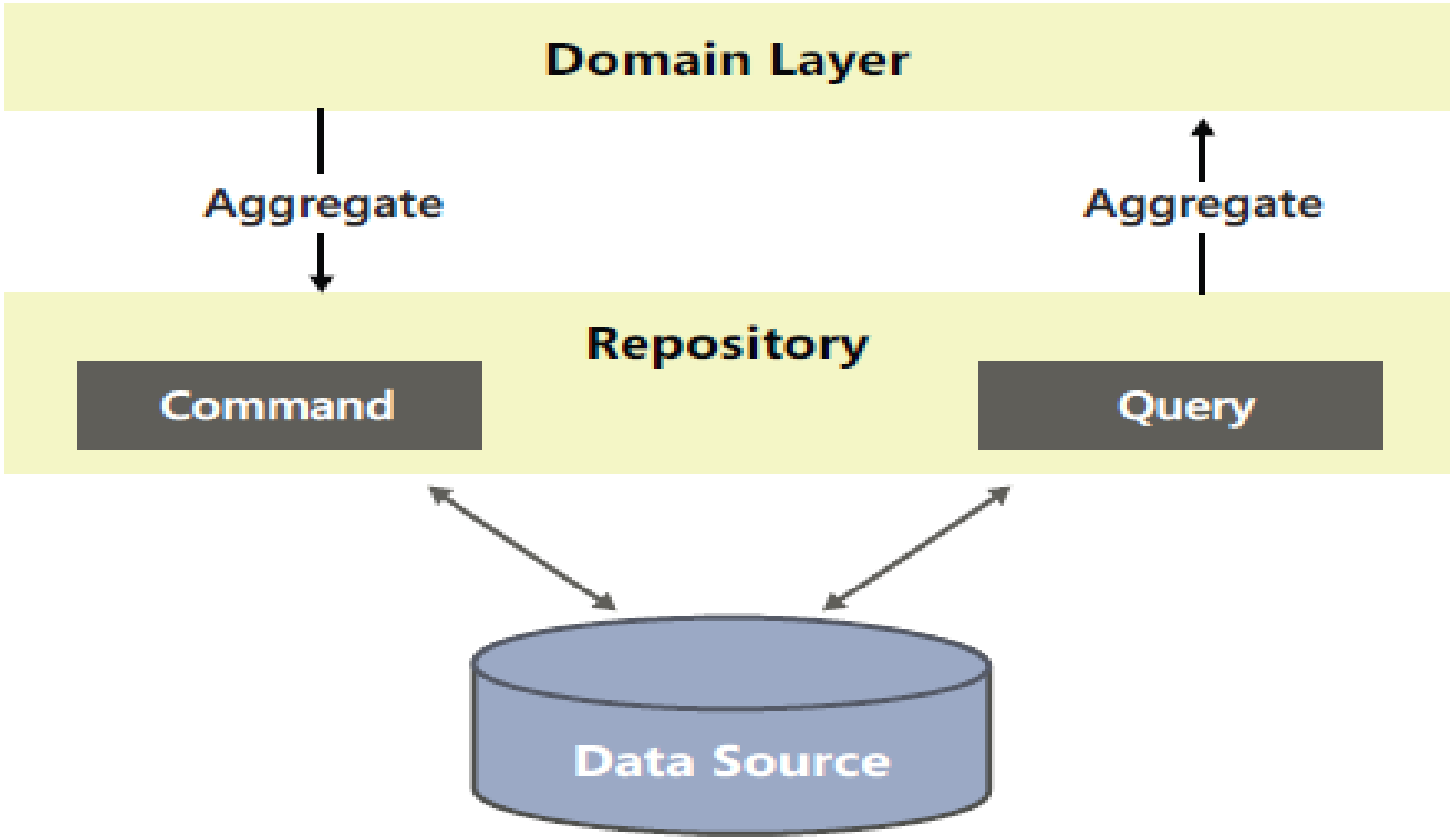


Figure 3. Mobile Channel Interaction

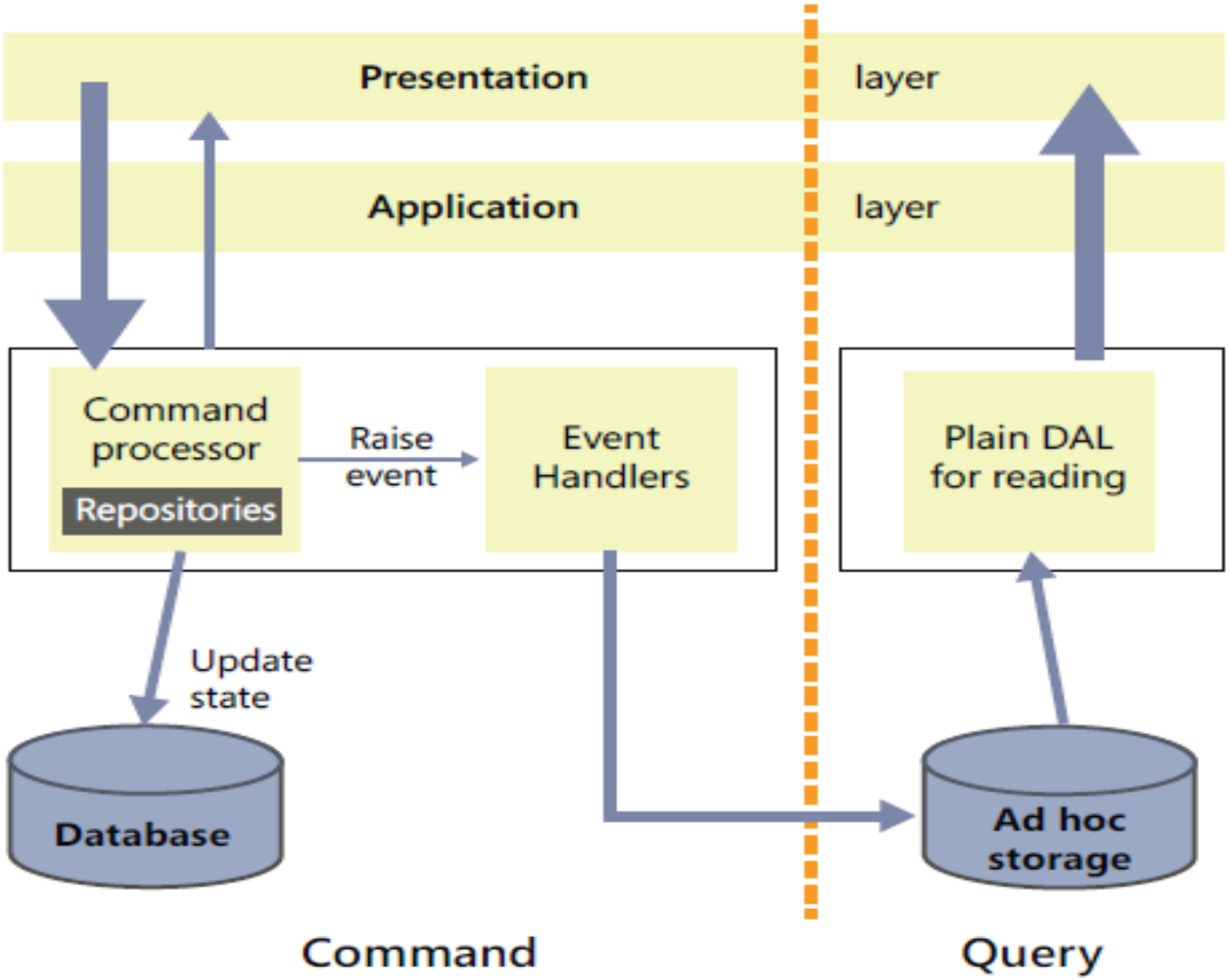
Customer makes purchase using eWallet using Point of Sale Channel



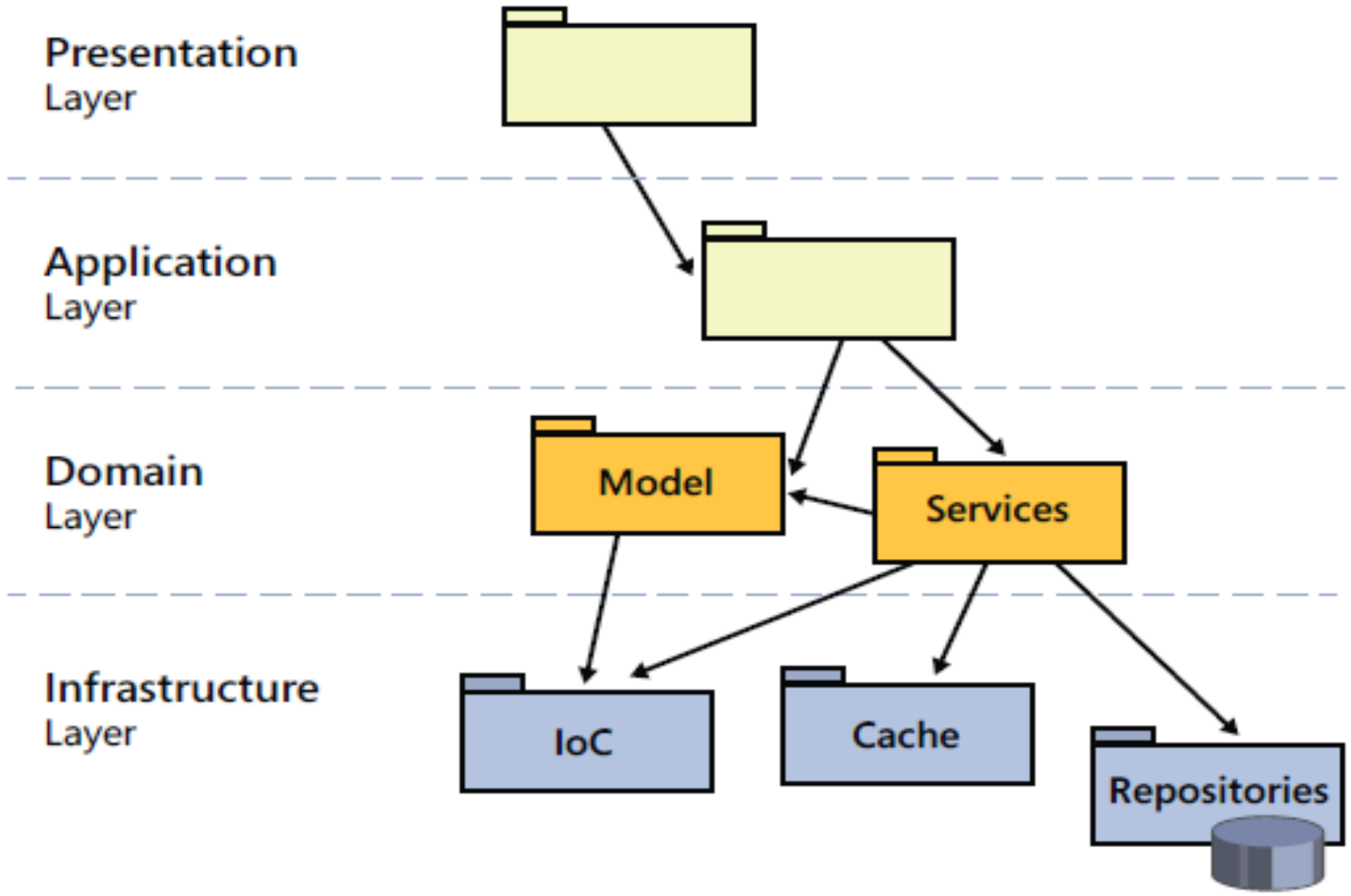
eWallet and Eftpos Channel Interaction



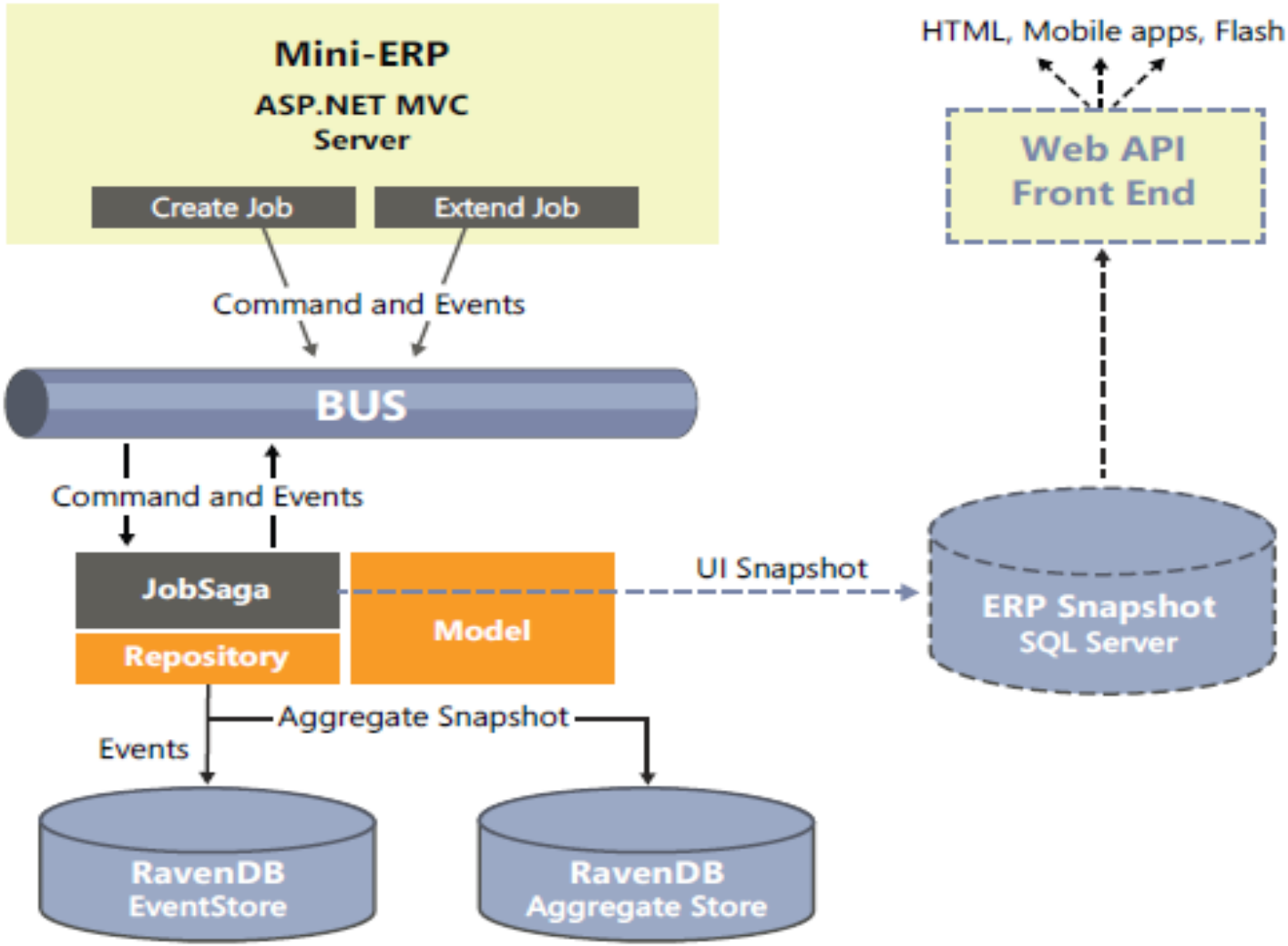
Fitting a repository in a layered architecture.



The big picture of the CQRS implementation of a collaborative system.

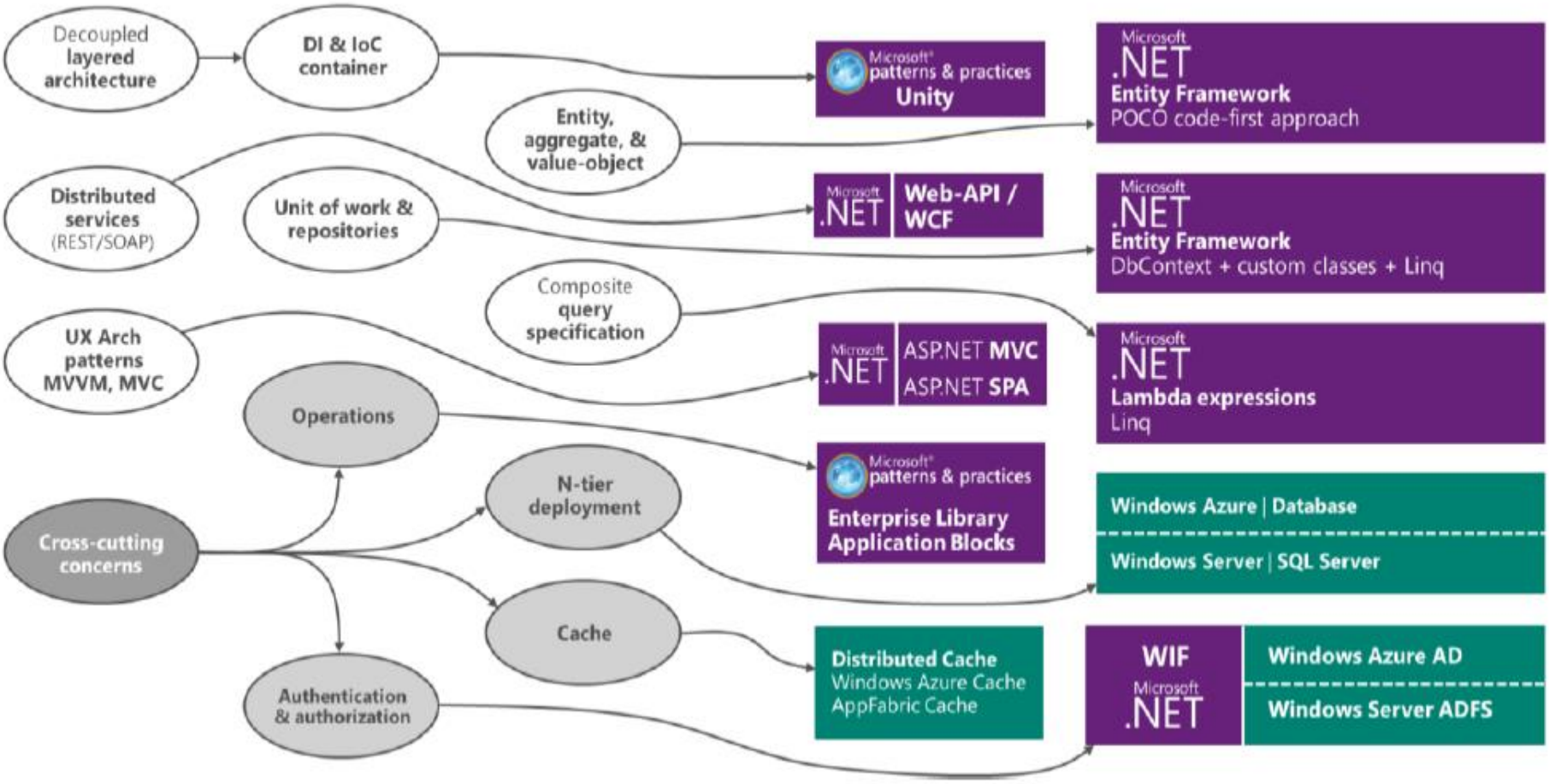


A more modern version of a layered architecture.



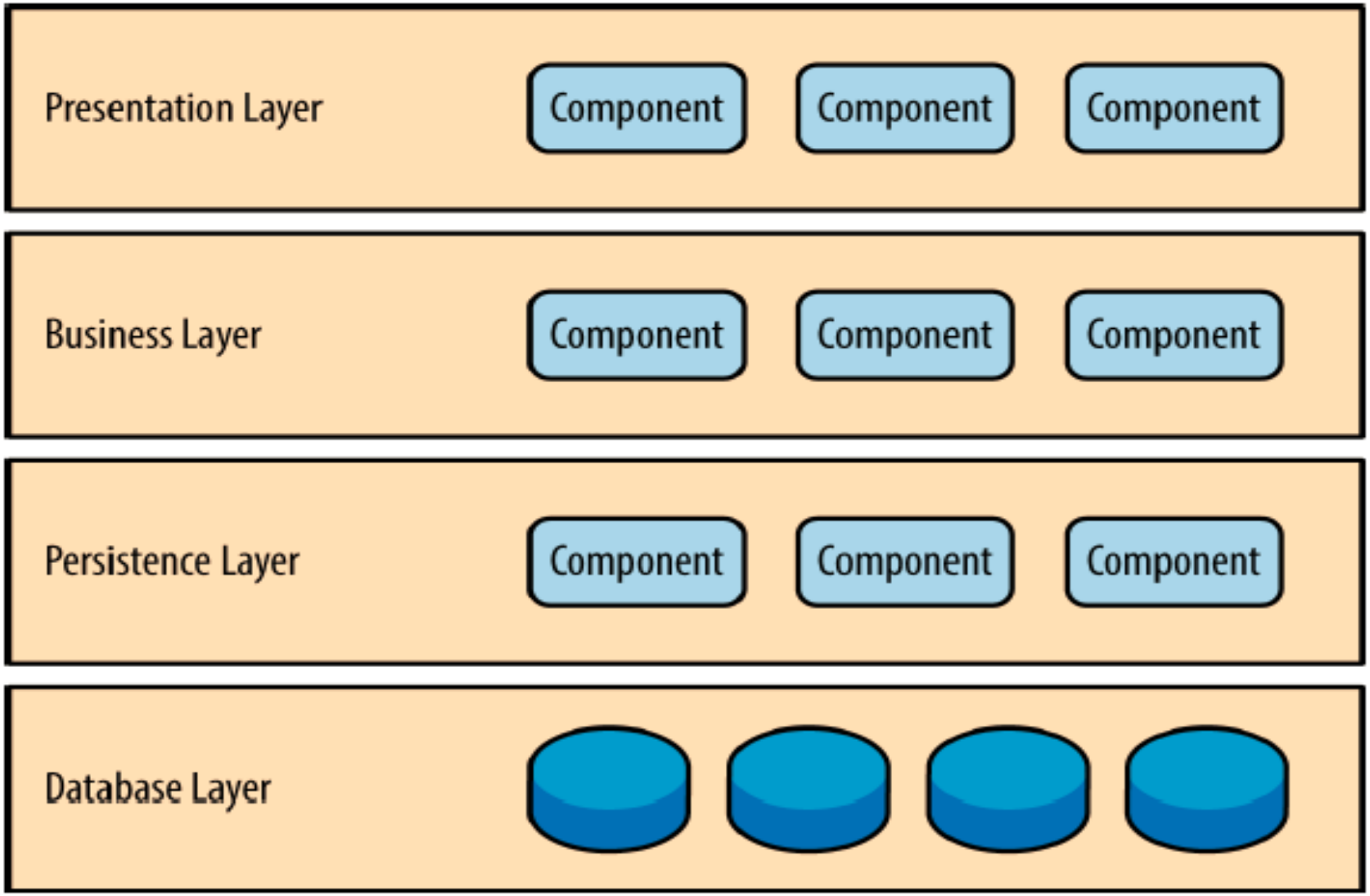
An overview of the mini-ERP system.

Mapping from Patterns to Technologies

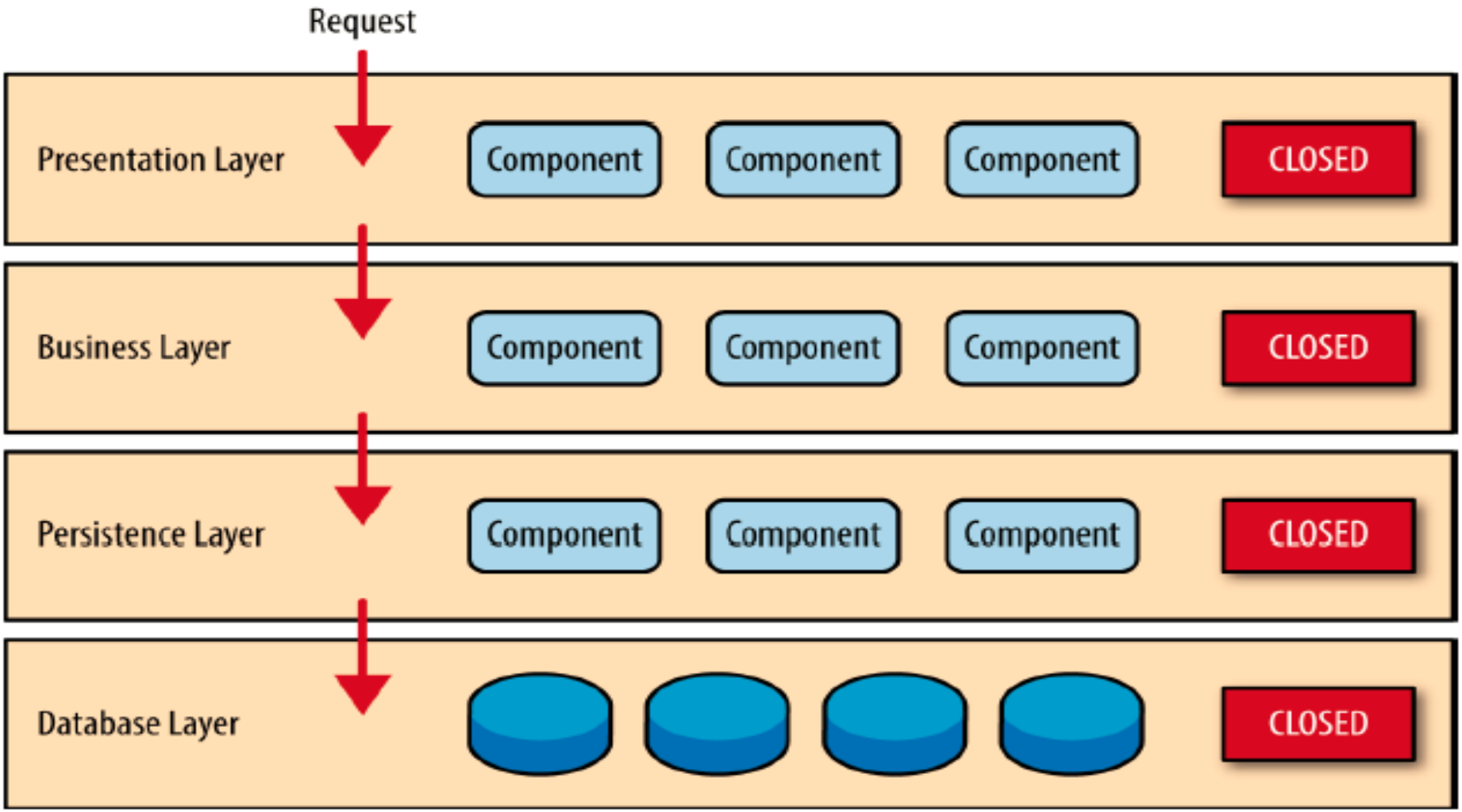


Layered Architecture

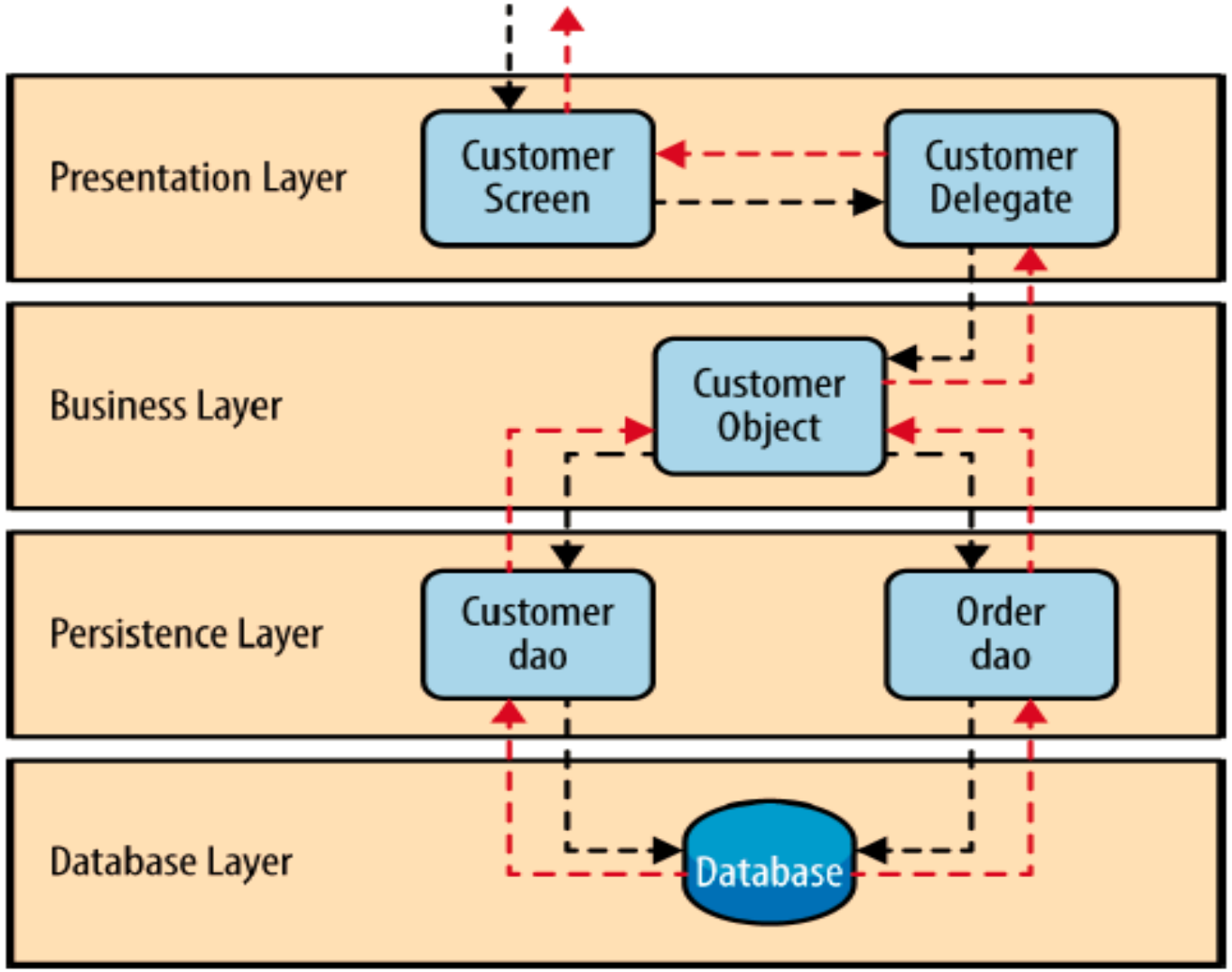
- Classic Architecture Pattern
- Components
- Separation Of Concerns
- Layer of Isolation
- Closed



Layered architecture pattern



Closed layers and request access



Layered architecture example

Considerations

- *Architecture sinkhole anti-pattern*
  - situation where requests flow through multiple layers of the architecture as simple pass-through processing with little or no logic performed within each layer
- Every layered architecture will have at least some scenarios that fall into the architecture sinkhole anti-pattern
  - The 80-20 rule is usually a good practice to follow to determine whether or not you are experiencing the architecture sinkhole anti-pattern
- *IF a majority of your requests are simple pass-through processing (20:80)*
  - *Keep Layers Open*
    - *Lack of Isolation*
    - *Difficult to Control Layer Changes.*

Pattern Analysis

Criteria	*Rating*	Analysis
Agility	Low	is the ability to respond quickly to a constantly changing environment. cumbersome and time-consuming to make changes in this Pattern.
Ease of deployment	Low	deployment can become an issue, particularly for larger applications. deployments that need to be planned, scheduled, and executed during off-hours or on weekends
Testability	High	Because components belong to specific layers in the architecture, other layers can be mocked or stubbed, making this pattern is relatively easy to test
Performance	Low	The pattern does not lend itself to high-performance applications due to the inefficiencies of having to go through multiple layers of the architecture to fulfill a business request
Scalability	Low	Due to tightly coupled and monolithic implementations of this pattern, applications build using this architecture pattern are generally difficult to scale. You can scale a layered architecture by splitting the layers into separate physical deployments or replicating the entire application into multiple nodes, but overall the granularity is too broad, making it expensive to scale.
Ease of development	High	Ease of development gets a relatively high score ,mostly because this pattern is so well known and is not overly complex to implement. Conway's law

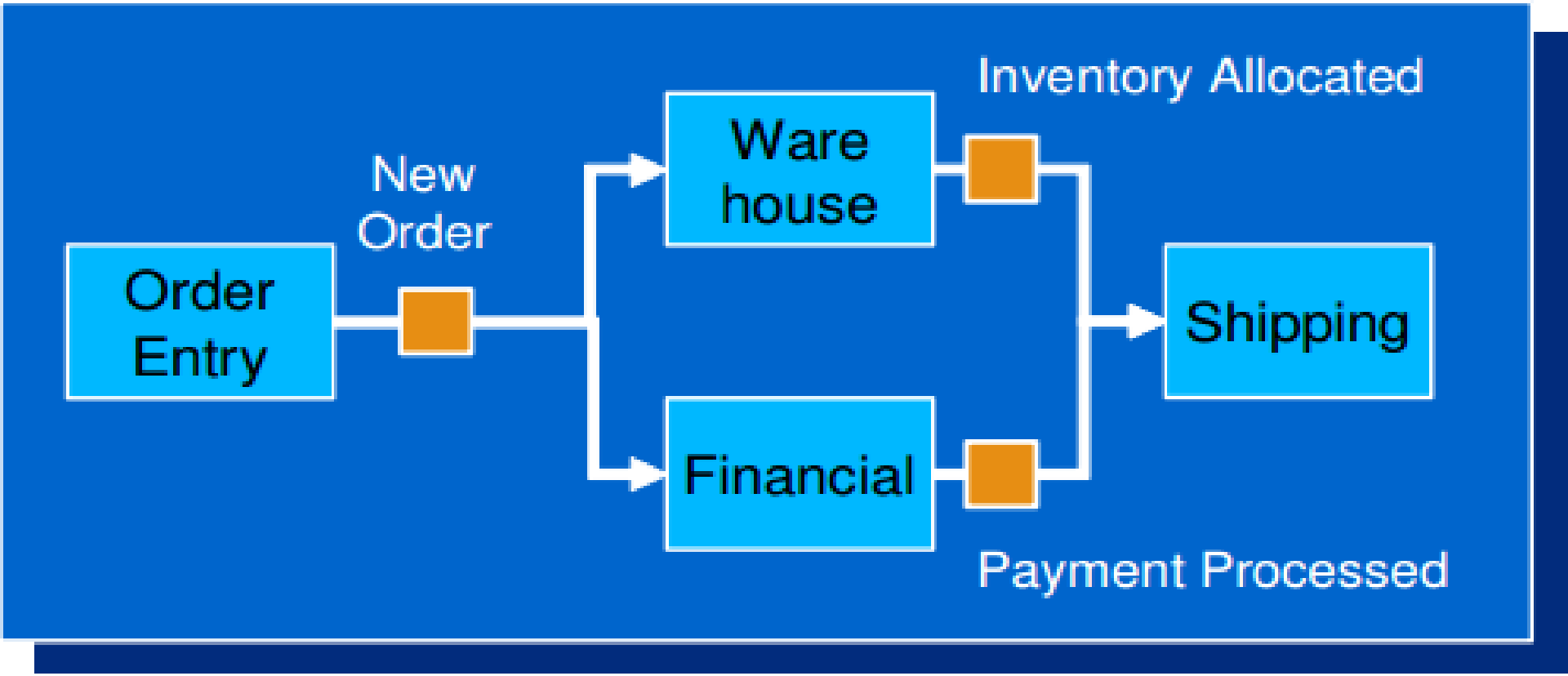
Event-Driven Architecture

- Distributed Asynchronous Architecture Pattern
- A Method of building enterprise systems in which events flow between decoupled components and services.
- A Maintainable , sustainable , and extensible model for building complex, distributed applications.
- Well Suited for Asynchronous , unpredictable environments
- Adapted to produce highly scalable applications
- Building Blocks
  - Highly decoupled, single-purpose event processing components that asynchronously receive and process events.
- Topologies
  - Mediator
  - Broker

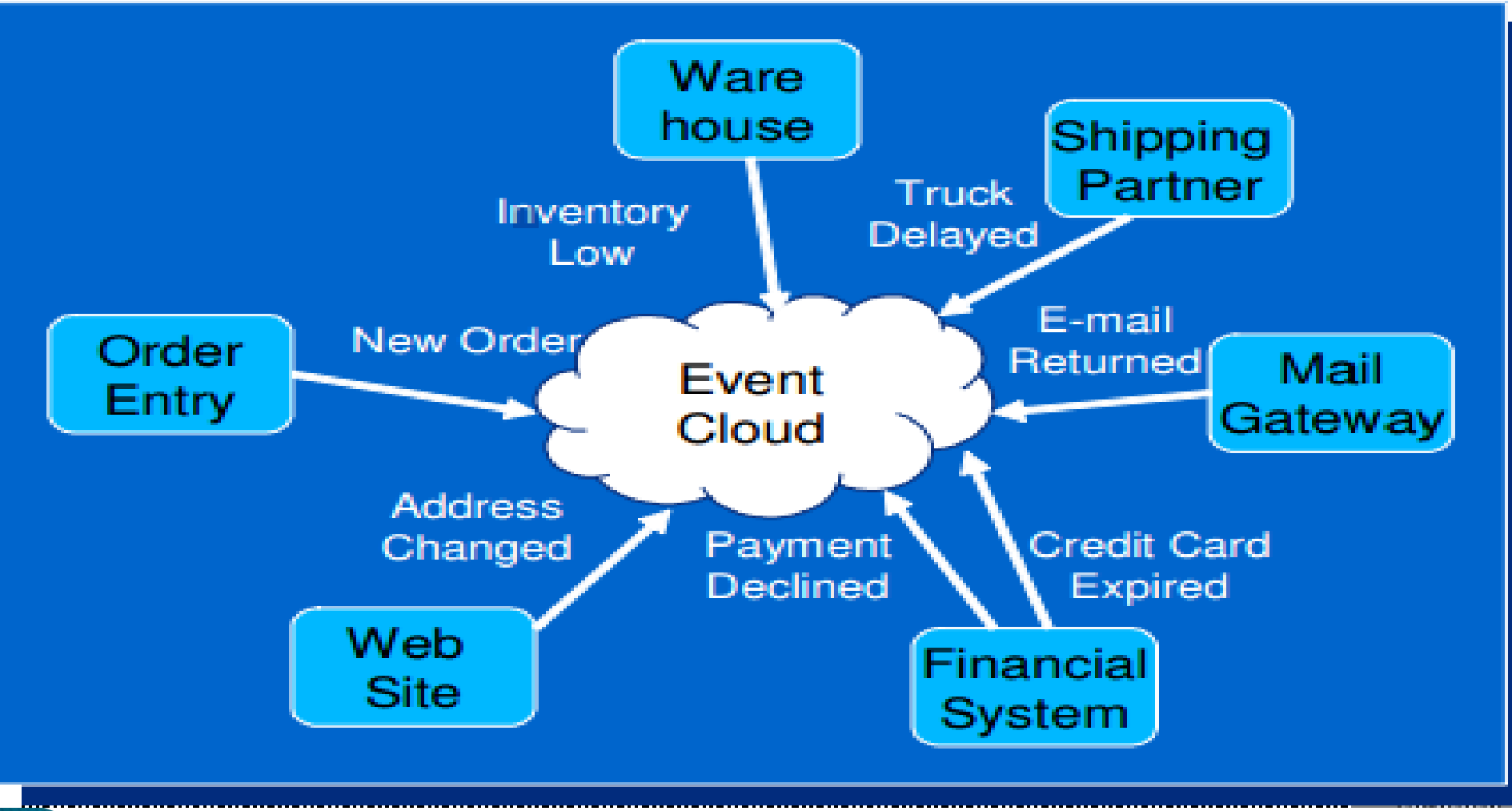
Problems With SOA

- Applications are composed at design time
- Linear flow between Services
- Predictable Behavior
- Request / Response is Common , Overused

Example


Inventory Allocated
Ware house
New Order
Order Entry
Shipping
Financial
Payment Processed

Event Cloud


Ware house
Shipping Partner
Inventory Low
Truck Delayed
Order Entry
New Order
Event Cloud
E-mail Returned
Mail Gateway
Address Changed
Payment Declined
Credit Card Expired
Web Site
Financial System

EDA Charcterstics

- Broadcast Communications
  - Participating systems broadcast events to any interested party. More than one party can listen to the event and process it.
- Timeliness
  - Systems publish events as they occur instead of storing them locally and waiting for the processing cycle, such as a nightly batch cycle
- Asynchrony
  - The publishing system does not wait for the receiving system(s) to process the event(s)

- Fine Grained Events
  - Applications tend to publish individual events as opposed to a single aggregated event
- Ontology
  - The overall system defines a arrangements to classify events, typically in some form of hierarchy. Receiving systems can often express interest in individual events or categories of events
- Complex Events Processing:
  - The system understands and monitors the relationships between events, for example event aggregation (a pattern of events implies a higher-level event) or causality (one event is caused by another).

EDA – Good Bye Call-Stack

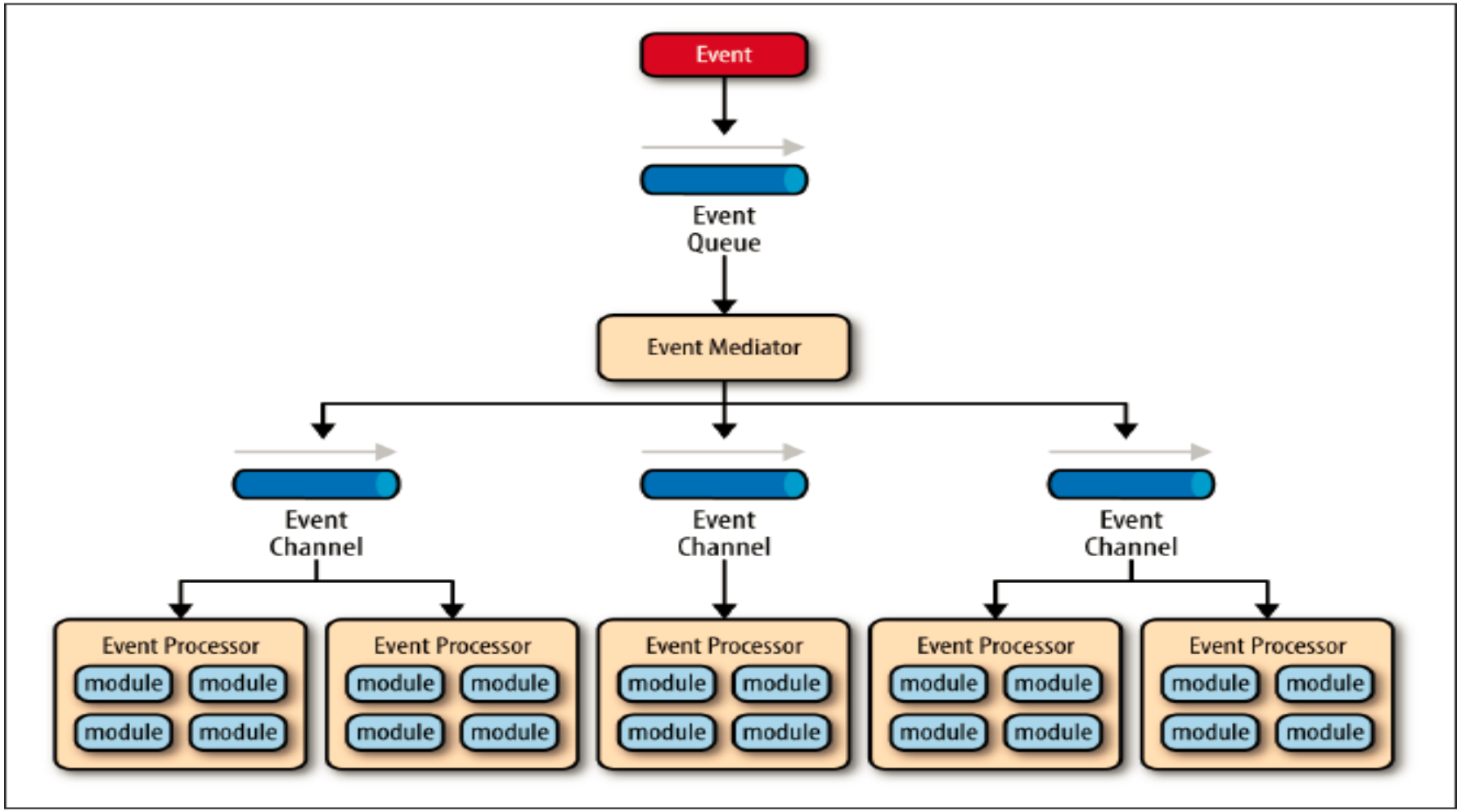
- Call stack based interaction allows one method to invoke another, wait for the results, and then continue with the next instruction
- Three Main Features
  - coordination
  - continuation
  - context

Call Stack Assumptions

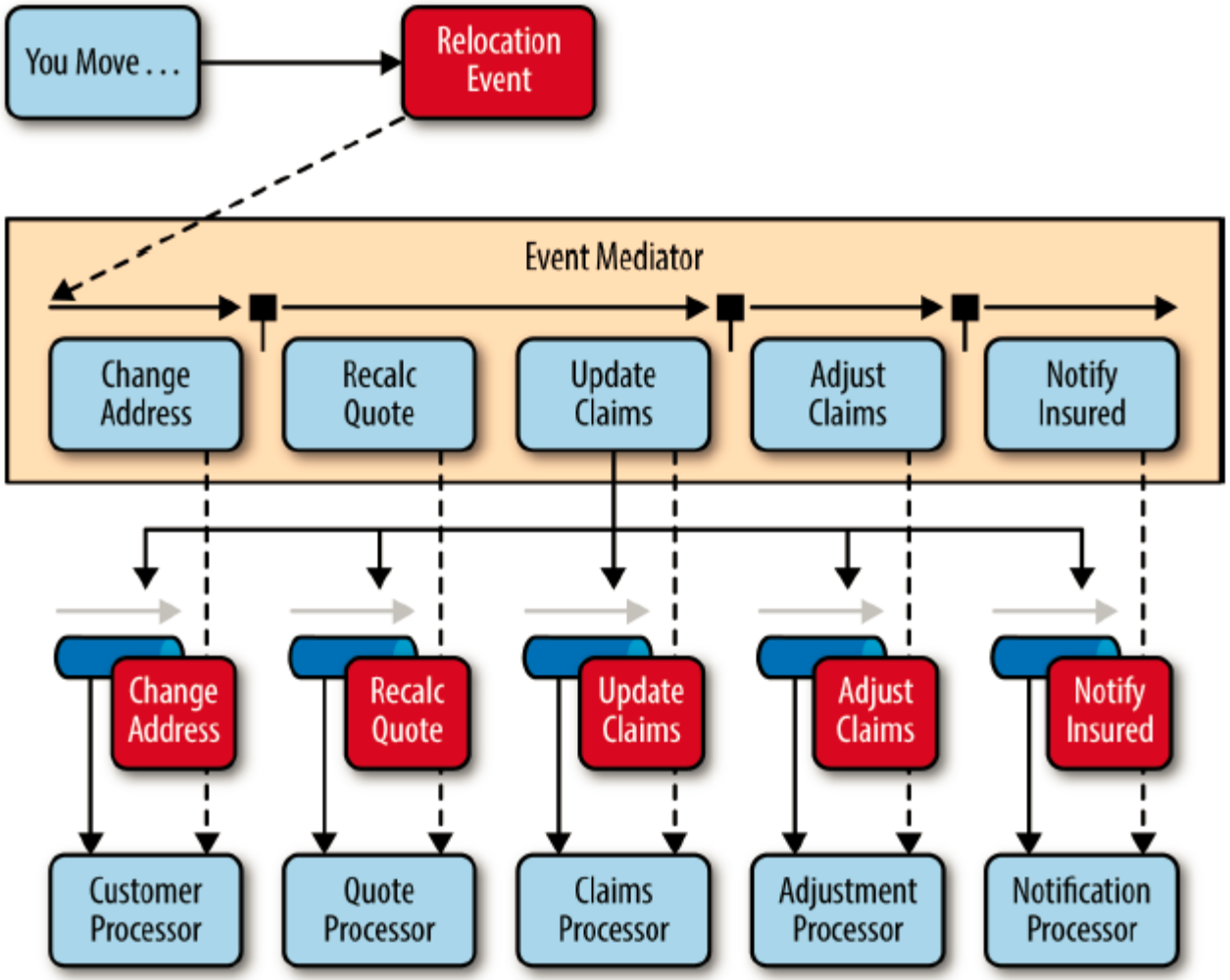
- One thing happens at a time
- We know what should happen in what order
- We know who can provide a needed function
- Execution happens in a single virtual machine

Mediator Topology

- This architecture is adopted when events that have **multiple steps** and require some level of **orchestration** to process the event .
- Players
  - Event Queues
    - Message Queue, Service Endpoint
    - Used to transport the event to the event mediator
  - Event Mediator
    - Orchestrate the Events
    - Initial Events
    - Mediators are implemented using
      - Mule ESB, Camel Integration Framework ,BPEL, JBPM, Spring Integration etcb
  - Event Channels
  - Event Processors
    - execute specific business logic to process the event



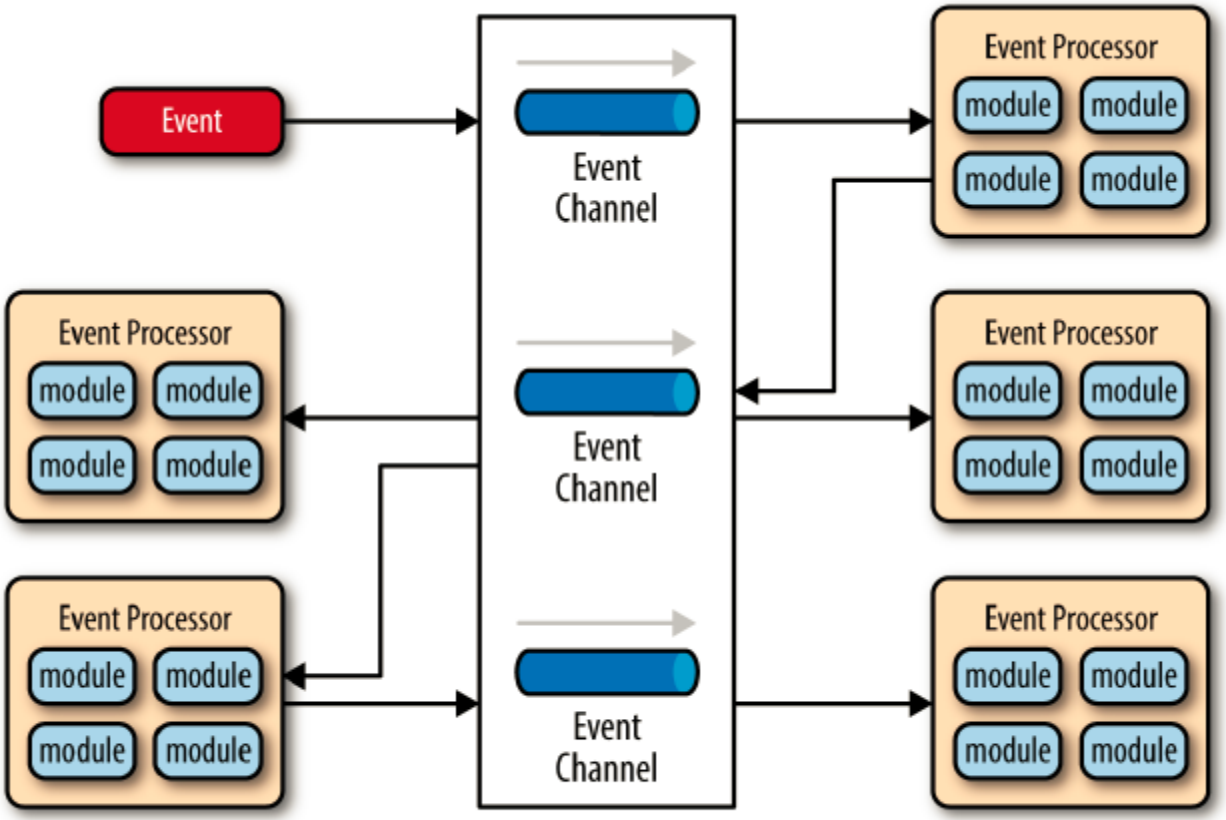
Event-driven architecture mediator topology



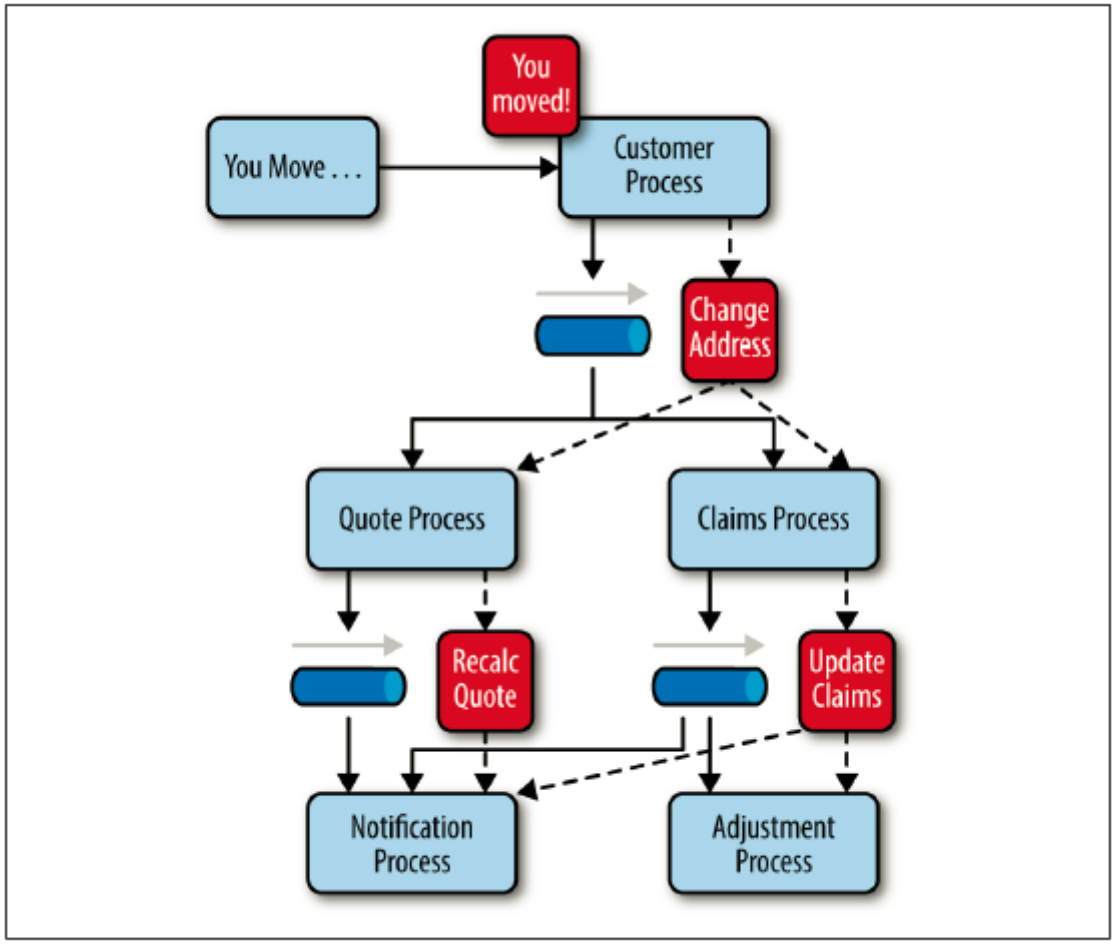
Mediator topology example

Broker Topology

- There is no central event mediator
- The message flow is distributed across the event processor components in a chain-like fashion through a lightweight **message broker**
- Adapted when there is a simple event processing flow and no orchestration of event is required.
- Architectural components
  - Broker (ActiveMq or HornetQ etc)
    - centralized or federated
    - contains all of the event channels that are used within the event flow
      - Message Queue or Topics or Both
  - Event Processor
    - Contains Business Logic



Event-driven architecture broker topology



Broker topology example

Considerations

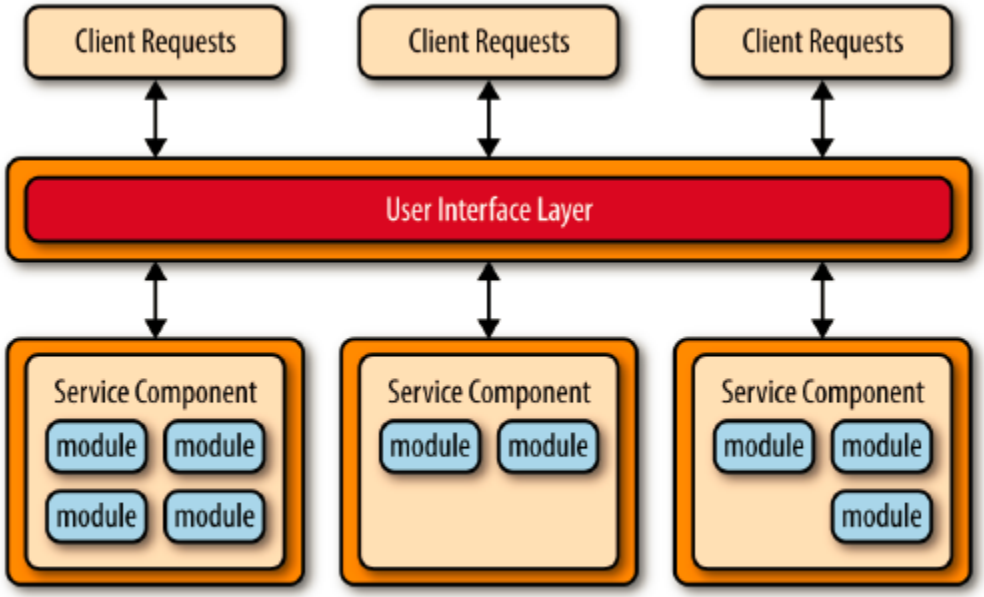
- Relatively complex pattern to implement
  - primarily due to its asynchronous distributed nature
- Distributed architecture issues
  - Remote Process Availability
  - Lack of responsiveness
  - Broker reconnection logic in the event of failure
- Lack of atomic transactions
  - event processor components are highly decoupled and distributed, it is very difficult to maintain a transactional unit of work across them
- Difficult in creation, maintenance, and governance of the event-processor component contracts
  - data values and data format being passed to the event processor

Pattern Analysis

Criteria	*Rating*	Analysis
Agility	High	Since event-processor components are single-purpose and completely decoupled from other event processor components, changes are generally isolated to one or a few event processors and can be made quickly without impacting other components.
Ease of deployment	High	Overall this pattern is relatively easy to deploy due to the decoupled nature of the event-processor components. The broker topology tends to be easier to deploy than the mediator topology, primarily because the event mediator component is somewhat tightly coupled to the event processors: a change in an event processor component might also require a change in the event mediator, requiring both to be deployed for any given change
Testability	Low	While individual unit testing is not overly difficult, it does require some sort of specialized testing client or testing tool to generate events. Testing is also complicated by the asynchronous nature of this pattern.
Performance	High	While it is certainly possible to implement an event driven architecture that does not perform well due to all the messaging infrastructure involved, in general, the pattern achieves high performance through its asynchronous capabilities; in other words, the ability to perform decoupled, parallel asynchronous operations outweighs the cost of queuing and dequeing messages.
Scalability	High	Scalability is naturally achieved in this pattern through highly independent and decoupled event processors. Each event processor can be scaled separately, allowing for fine-grained scalability.
Ease of development	Low	Development can be somewhat complicated due to the asynchronous nature of the pattern as well as contract creation and the need for more advanced error handling conditions within the code for unresponsive event processors and failed brokers.

Micro-Services Pattern

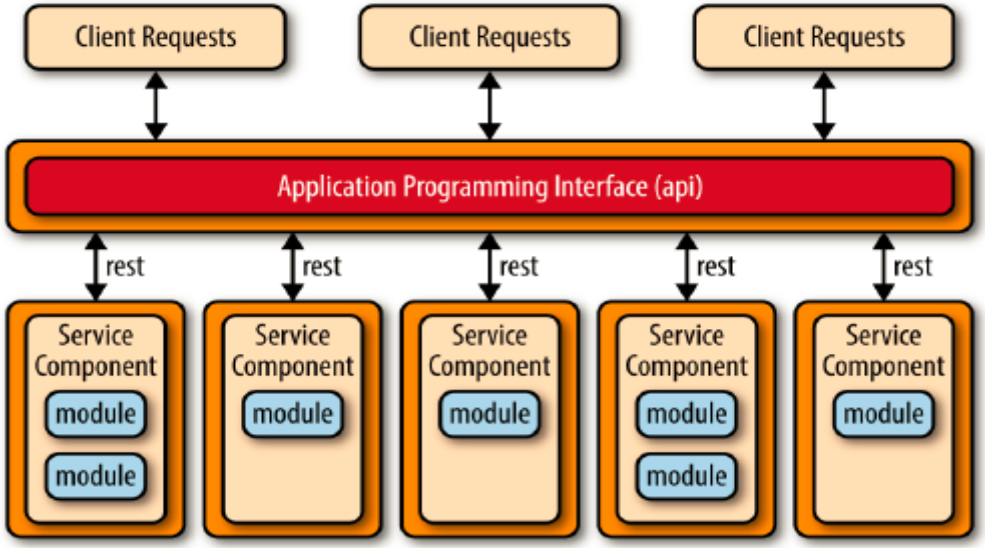
- Core Concepts
  - Separately Deployed Units
  - Service Component
    - Contain One or More Modules
  - Distributed Architecture



Basic Microservices architecture pattern

Pattern Topologies

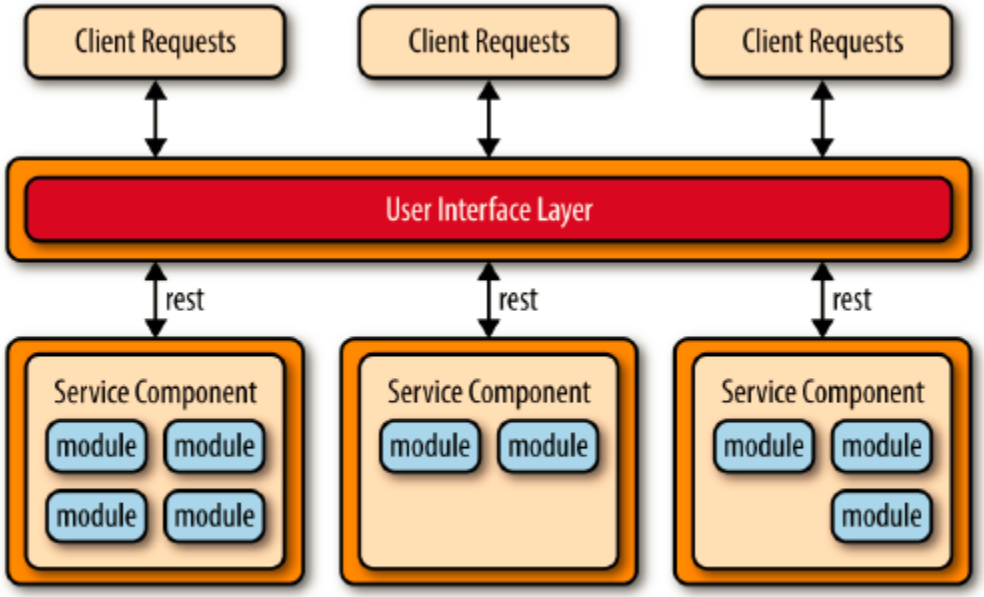
- *API REST-based topology*
  - useful for websites that expose small, self-contained individual services through some sort of API
  - Fine-grained service Components are typically accessed using a REST-based interface implemented through a separately deployed web-based API layer



API REST-based topology

Application REST-based topology

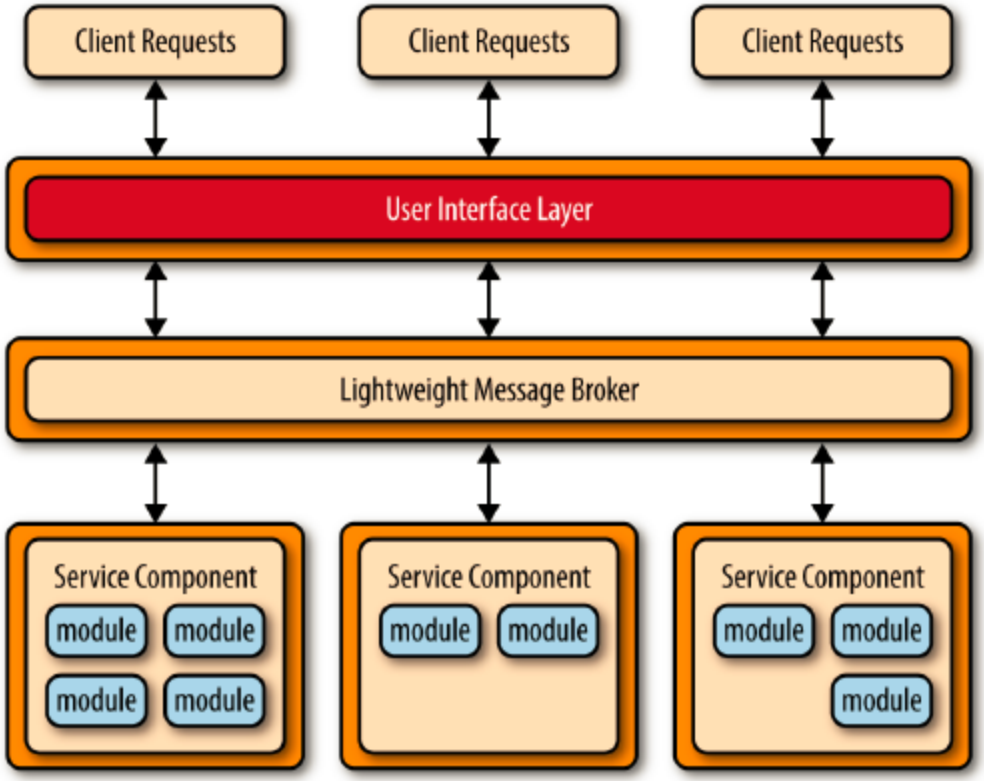
- client requests are received through traditional web-based or fat-client business application screens rather than through a simple API layer
- Service components tend to be larger, more coarse-grained, and represent a small portion of the overall business application rather than fine-grained



Application REST-based topology

Centralized Messaging Topology

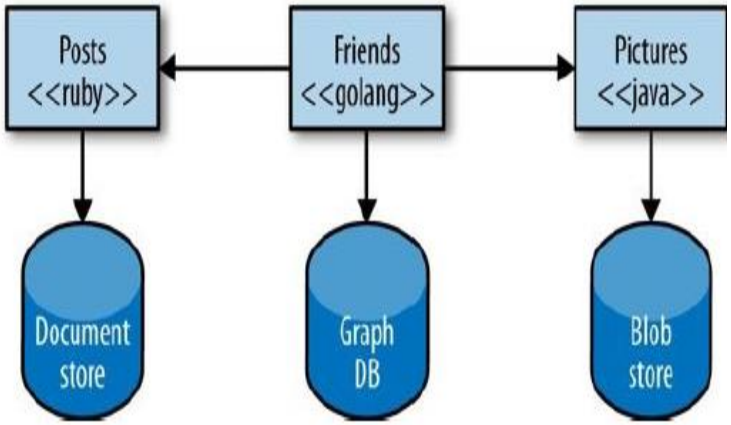
- This topology uses a lightweight centralized message , etc.)
  - Does not per
    - orchestrat
  - lightweight t
  - Use this topo requiring more sophisticatec terface and the service c
    - advance
    - asynch
    - monito
    - error h
    - Single p
- and bette calability



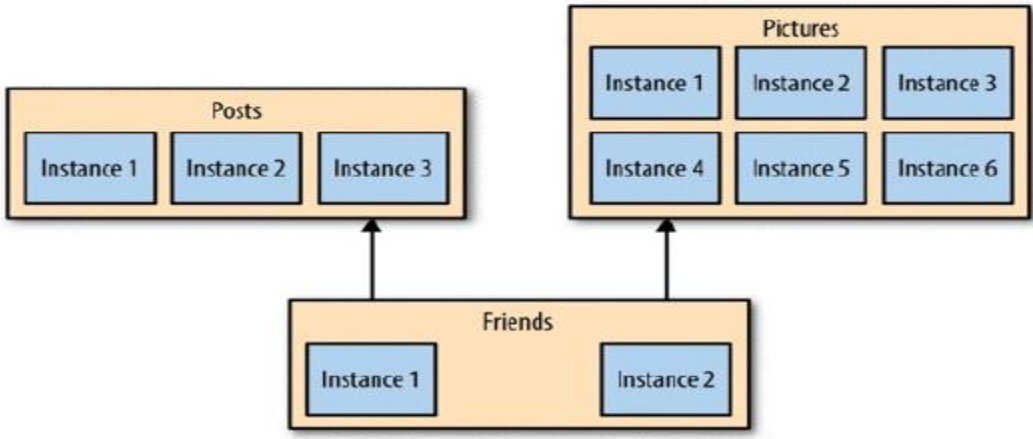
Centralized messaging topology

Consideration

- Technology Heterogeneity
- Resilience
- Scaling
- Ease of Deployment
- Organizational Alignment
- Optimizing for Replaceability



Microservices can allow you to more easily embrace different technologies



You can target scaling at just those microservices that need it

Pattern Analysis

Criteria	*Rating*	Analysis
Agility	High	Due to the notion of separately deployed units, change is generally isolated to individual service components, which allows for fast and easy deployment. Also, applications build using this pattern tend to be very loosely coupled, which also helps facilitate change..
Ease of deployment	High	Overall this pattern is relatively easy to deploy due to the decoupled nature of the service components
Testability	High	Due to the separation and isolation of business functionality into independent applications, testing can be scoped, allowing for more targeted testing efforts.
Performance	Low	While you can create applications implemented from this pattern that perform very well, overall this pattern does not naturally lend itself to high-performance applications due to the distributed nature of the microservices architecture pattern.
Scalability	High	Because the application is split into separately deployed units, each service component can be individually scaled, allowing for fine-tuned scaling of the application. For example, the admin area of a stock-trading application may not need to scale due to the low user volumes for that functionality, but the trade-placement service component may need to scale due to the high throughput needed by most trading applications for this functionality.
Ease of development	High	Because functionality is isolated into separate and distinct service components, development becomes easier due to the smaller and isolated scope. There is much less chance a developer will make a change in one service component that would affect other service components, thereby reducing the coordination needed among developers or development teams

Pattern Analysis

	Layered	Event-driven	Microkernel
Overall Agility	↓	↑	↑
Deployment	↓	↑	↑
Testability	↑	↓	↑
Performance	↓	↑	↑
Scalability	↓	↑	↓
Development	↑	↓	↓

VM

- A VM is an abstraction of physical hardware
- Each VM has a full server hardware stack from virtualized BIOS to virtualized network adapters, storage, and CPU
  - Complete Hardware Stack
- VM instantiation requires starting a full OS
- Can use different operating systems or kernels
- prime focus is on Heterogeneous OS platforms
- VM's are fat in terms of system requirements

Containers

- Container abstraction occurs at the OS level
- The user space is abstracted
- Each user shares the same OS instance including kernel, network connection, and base file system
- containers use less memory and CPU than VMs running similar workloads
  - Since a separate kernel doesn't load for each user session
- Docker, lmctfy (Let Me Contain That For You).
- Multiple copies of the same application are what you want, then you'll love containers.
- Containers save a data center or cloud provider tens-of-millions of dollars annually in power and hardware costs
- Containers gives you instant application portability.
- Docker Containers can run inside Virtual Machines
- prime focus is on applications and their dependencies

Why Containers

- Do you need to run the maximum amount of particular applications on a minimum of servers?
- Do you need instant application portability ?

The challenges of using containers

- **Security**
  - Since containers share the same kernel, admins and software vendors need to apply special care to avoid security issues from adjacent containers
- **Management**
  - container management more of an art than a science
  - capacity management
- Breaking deployments into more functional discrete parts is smart, but that means we have MORE PARTS to manage. There's an inflection point between separation of concerns and sprawl.
  - Rob Hirschfeld

VMS vs Containers