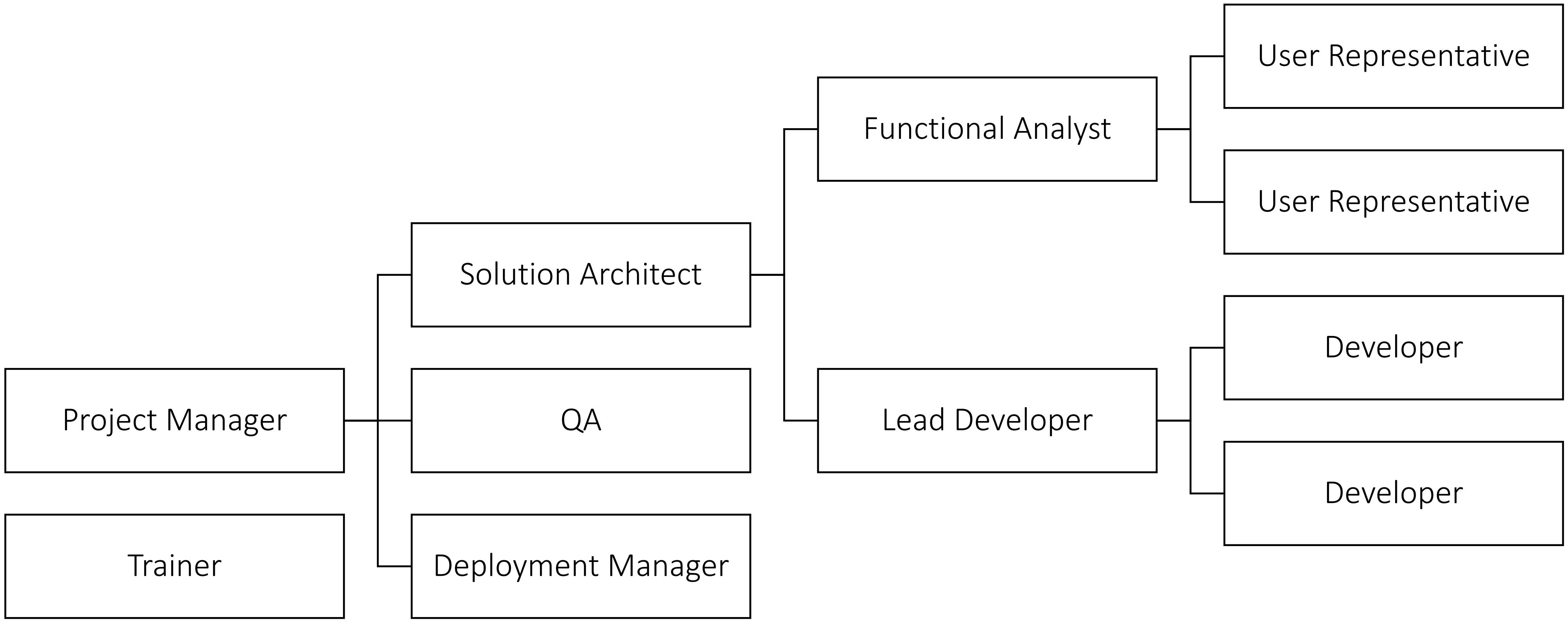
# Software Team Roles

## Principles Of Software Development

- Implementation different in each company
- Always a need to:
    - Understand business problem
    - Document non-technical business solution
    - Convert solution to technical architecture
    - Convert architecture to code
    - Manage developers
    - Test code
    - Deploy code

# Team Roles In Software Development

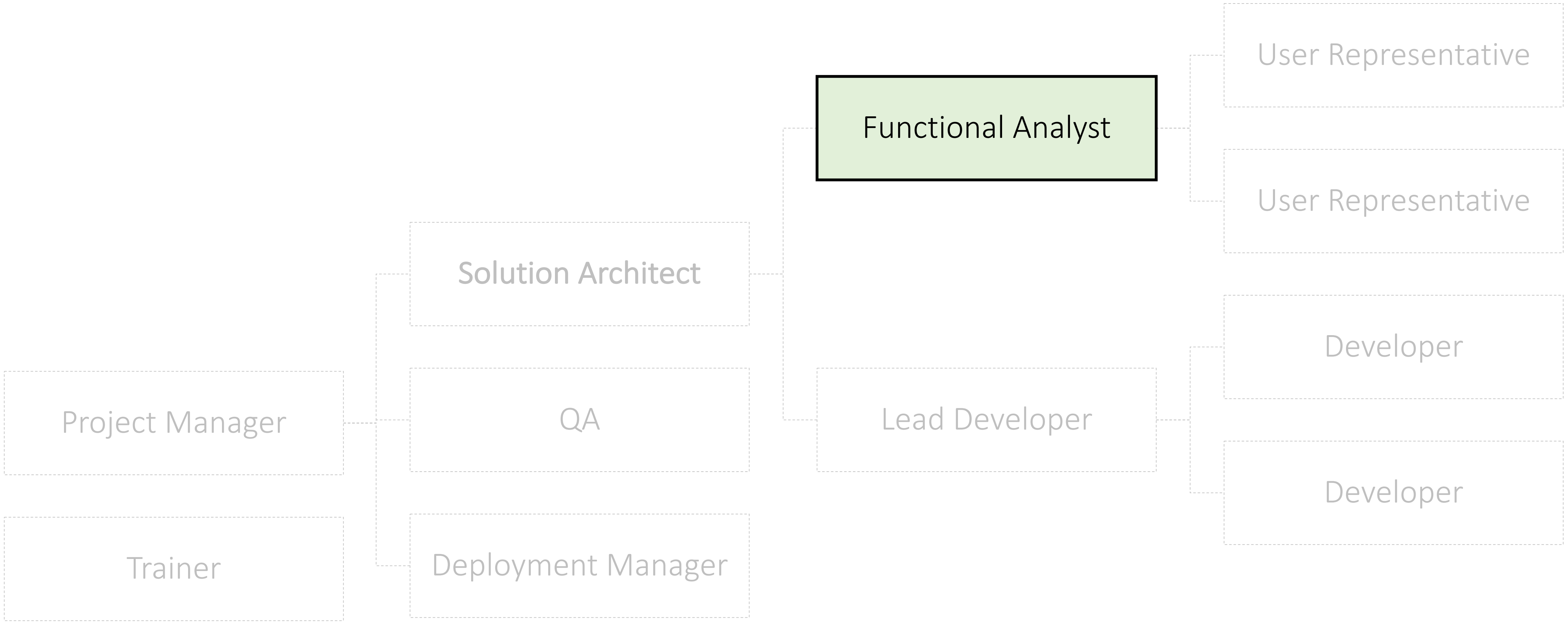


## Required Skills

- Understanding the Business
- Cross-Domain Understanding
- Multiple Perspectives
- People Skills
- Lifelong Learning

## The Functional Analyst

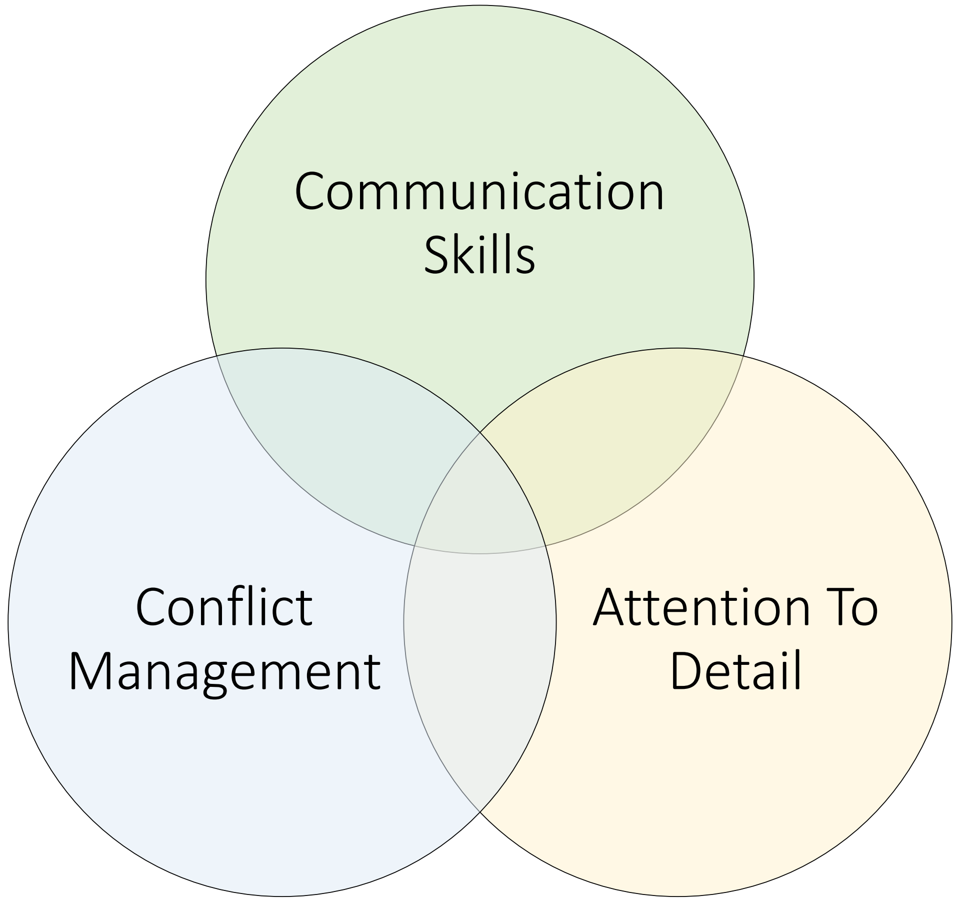
# The Functional Analyst Team Role



## Responsibilities Of The Functional Analyst

- Responsible for Functionality
- Captures, Consolidates, and Communicates information
- Constantly asks questions:
  *What do you mean?* / *How does this fit in with…?*
- Identifies and resolves conflicts
- Produces Requirements Specification

## Skills Of The Functional Analyst



## Skills Of The Functional Analyst

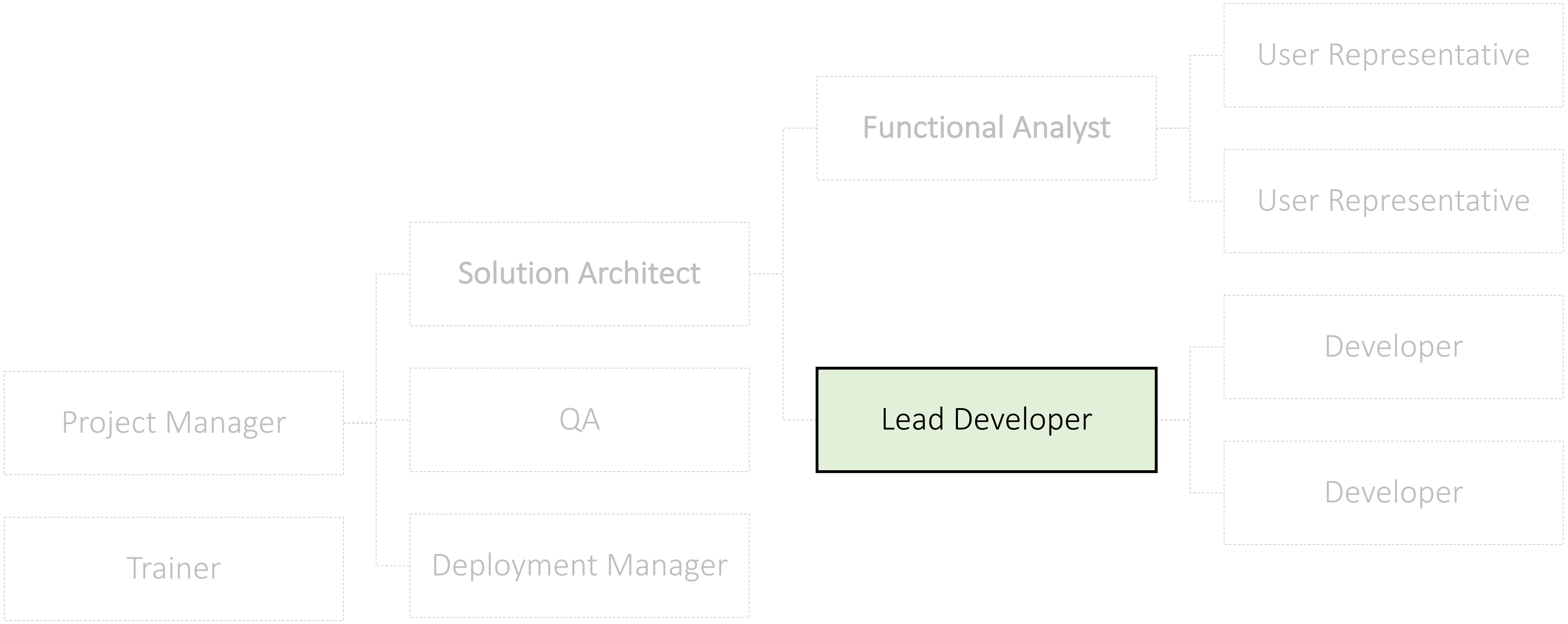
- Precise communicators
- Know when detail is necessary and when not
- Adept at dealing with differing opinions
- Great relationship skills
- Very good listeners
- Can create clear and precise documents
- Skilled in using Office tools

## Pros/Cons Of The Functional Analyst

| Pros | Cons |
|---|---|
| • Key role<br>• Lots of interactions | • Has to work with bad URs<br>• Can expect conflict<br>• Will receive blame if functionality is missing |

## The Lead Developer

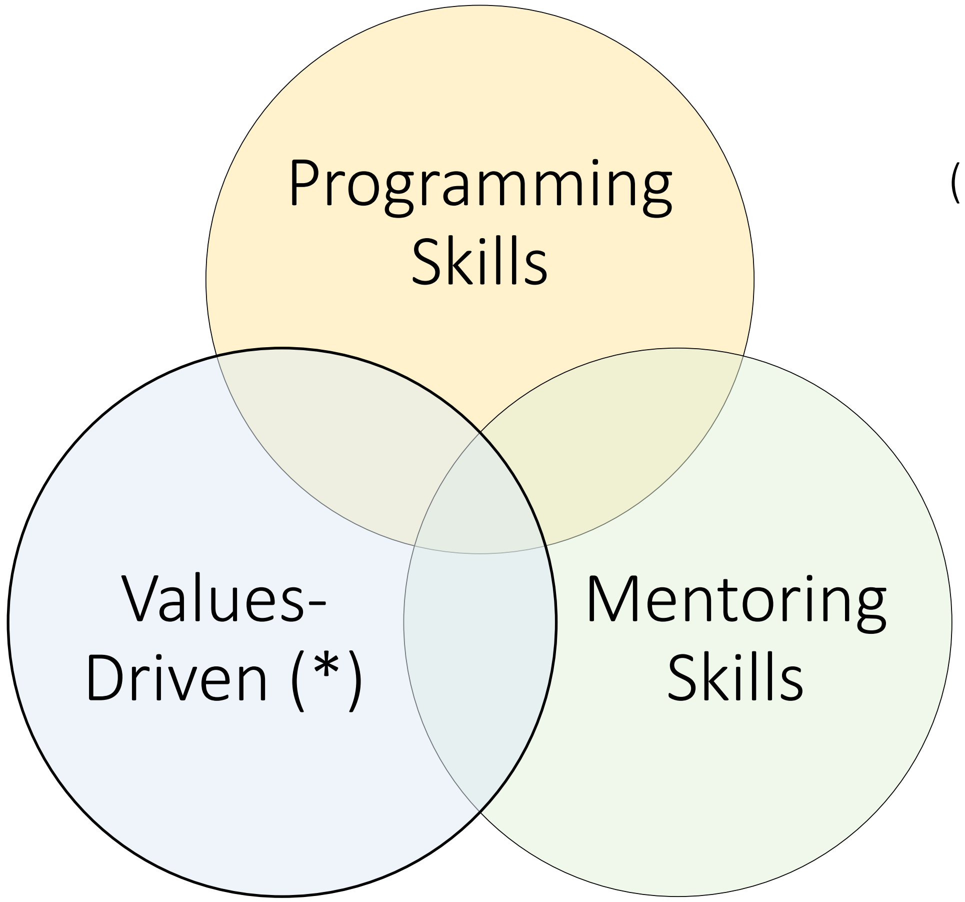
# The Lead Developer Team Role



## Responsibilities Of The Lead Developer

- Leads and mentors developers
- Assigns tasks to developers
- Details and partitions work
- Ensures that all developers are successful

# Skills Of The Lead Developer



(*) Precision
Elegance
Reusability
Efficiency

## Skills Of The Lead Developer

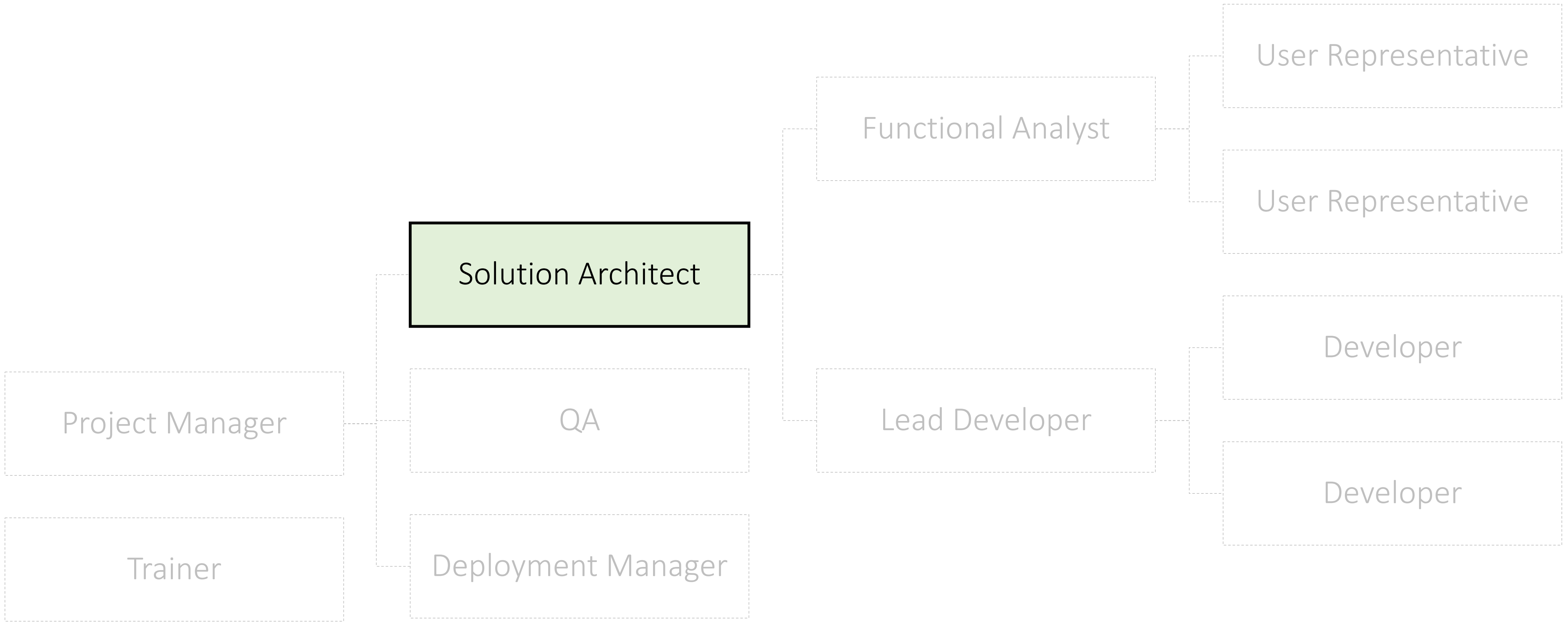
- Grows out of Developer role
- Requires great relationship with Architect
- Wide knowledge of Libraries/Tools/Techniques
- Adept at creating technical specifications
- Adept at build & configuration management
- Adept at debugging, post-mortem log inspection, etc.
- Can create own tools if needed

# Pros/Cons Of The Lead Developer

| Pros | Cons |
|---|---|
| • Lead-in to SA<br>• Involves coding (optional)<br>• Can pick & choose cool tasks | • Can get squeezed between SA and Ds<br>• Lifelong learning required<br>• Turns into D if the PM is weak<br>• Team might be too small<br>• Vulnerable to offshoring |

## The Solution Architect

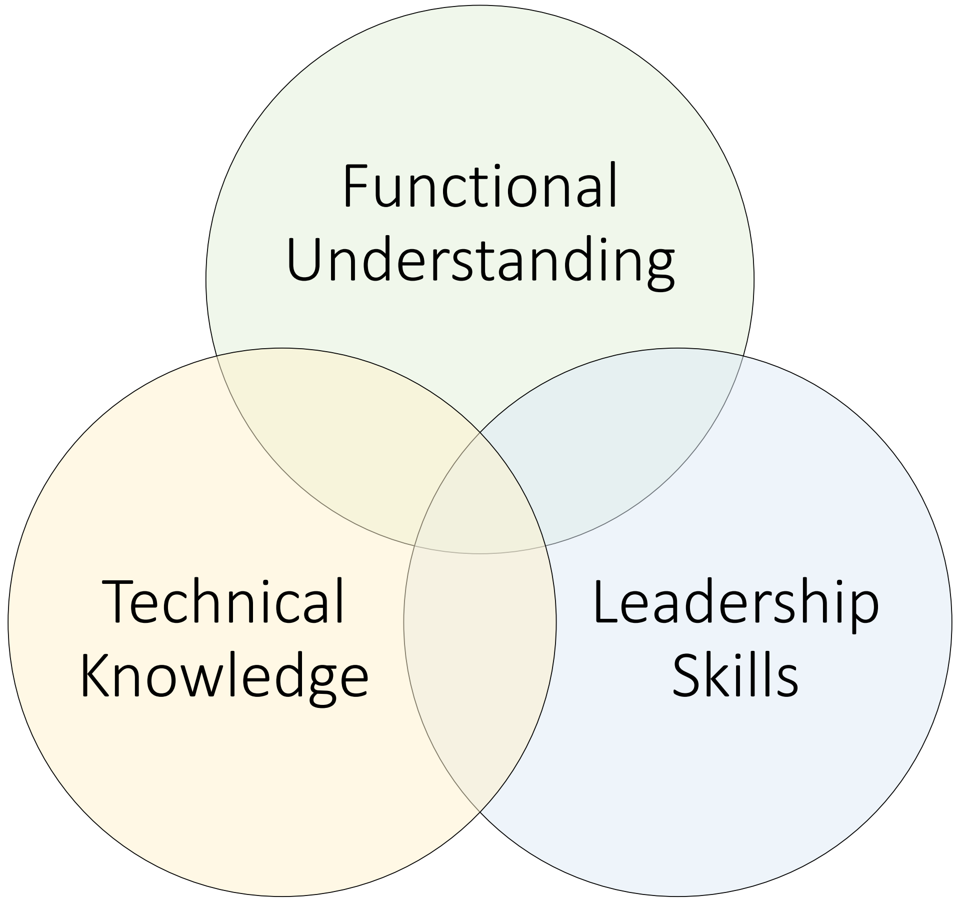
# The Solution Architect Team Role



## Responsibilities Of The Solution Architect

- Responsible for Technology
- Converts Functional Requirements to Technical Architecture
- Carefully balances Patterns/Requirements/Elegance/Concepts
- Researches Key Technologies
- Has deep understanding of Design Patterns
- Motivates and guides development team
- Ensures that the Lead Developer is successful

## Skills Of The Solution Architect



## Skills Of The Solution Architect

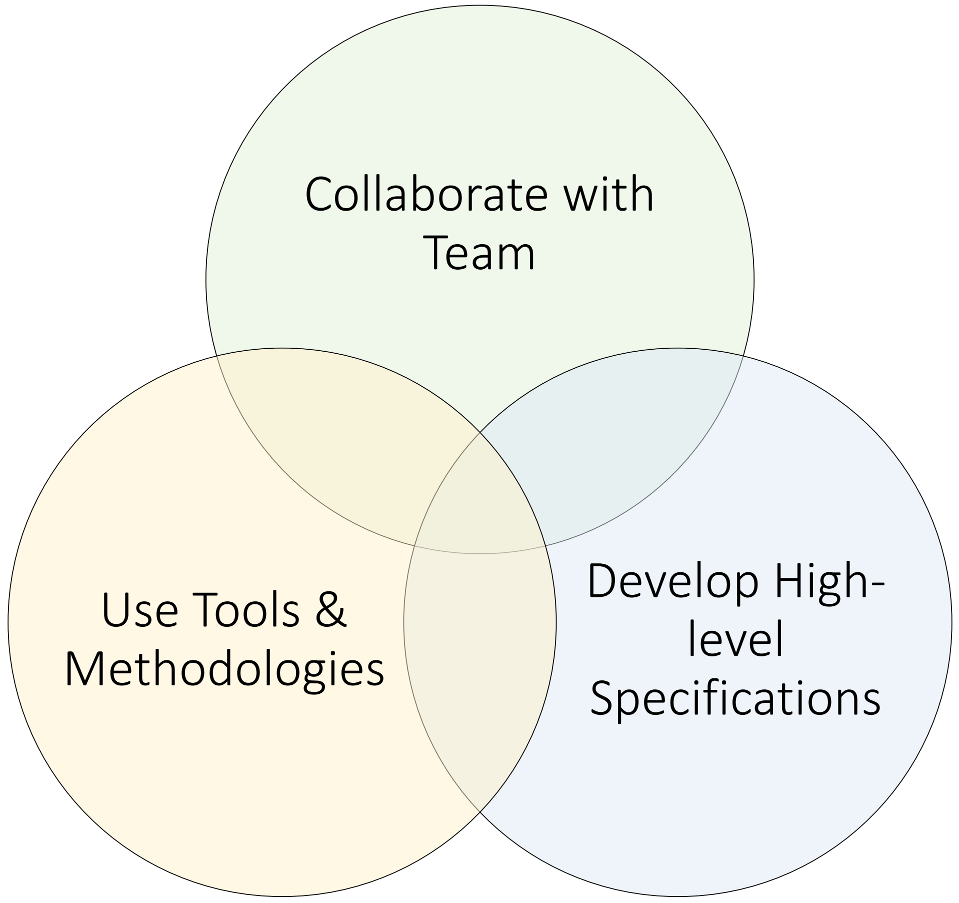
- Grows out of Lead Developer role
- Requires great relationship with Lead Developer
- Helicopter view at all times
- Deep understanding of Design Patterns
- Fluent in UML & other Design Tools
- Experience with Tools & Code Generators

## Pros/Cons Of The Solution Architect

| Pros | Cons |
|---|---|
| • High value position<br>• Great salary<br>• Visible role<br>• Lots of interactions<br>• Safe from outsourcing | • Difficult to stay up to date<br>• Difficult to get right<br>• Can receive bad requirements<br>• First in line to receive blame |

## The Architect Job Description

## Main Responsibilities Of The Architect



## Job Brief Of The Architect

*“ […] make intuitive high level decisions […] You will see the big picture and create architectural approaches for software design and implementation to guide the development team.*

*[…] strong technical background and excellent IT skills […] experienced in designing […] unified vision for software characteristics and functions.*

*[…] provide a framework for the development of a software or system that will result in high quality IT solutions.“*

## Responsibilities Of The Architect

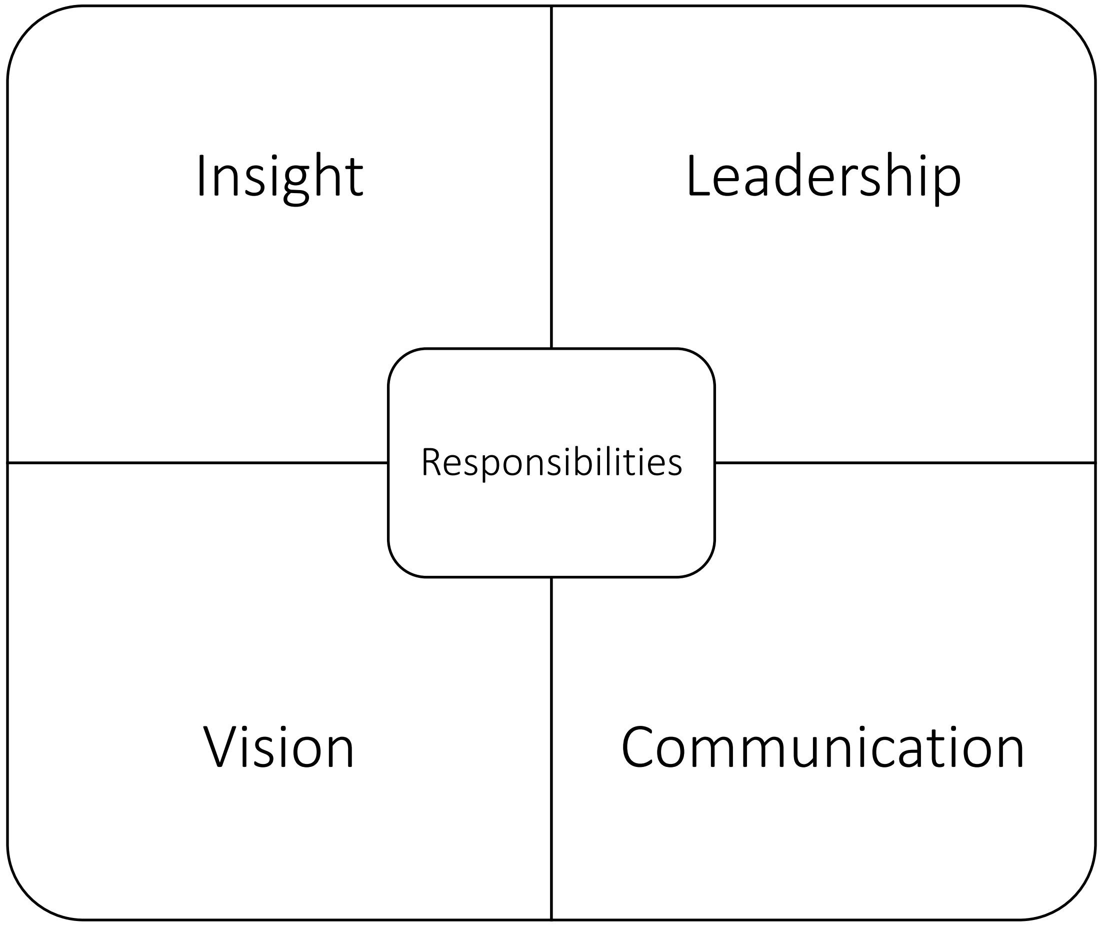
- Collaborate with team to draft functional and non-functional requirements
- Use tools and methodologies to create representations for functions and UIs
- Develop high-level product specifications
- Define all aspects of development
- Communicate all concepts and guidelines to development team
- Oversee progress of development team
- Provide technical guidance and coaching to developers and engineers
- Ensure software meets all requirements
- Approve final product before launch

## Requirements Of The Architect

- Proven experience as software architect
- Experience in software development and coding
- Excellent knowledge of software and application design and architecture
- Excellent knowledge of UML and other modeling methods
- Familiarity with UI/UX design
- Understanding of software quality assurance principles
- A technical mindset with great attention to detail
- High quality organizational and leadership skills
- Outstanding communication and presentation abilities
- MSc/MA in computer science, engineering or relevant field

## Responsibilities Of A Great Architect

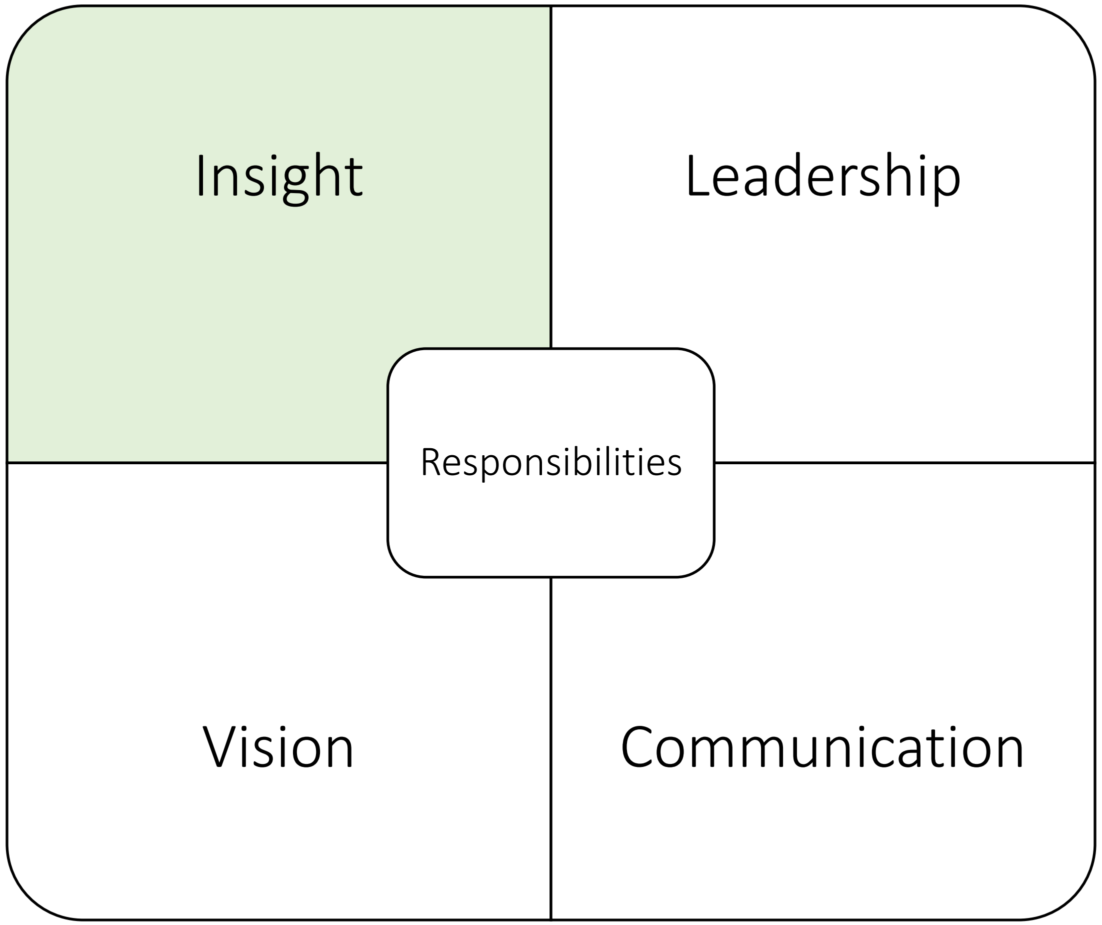
# Responsibilities Of A Great Architect



## Abstract Complexity

A great architect abstracts the complexity of a system into a manageable model.

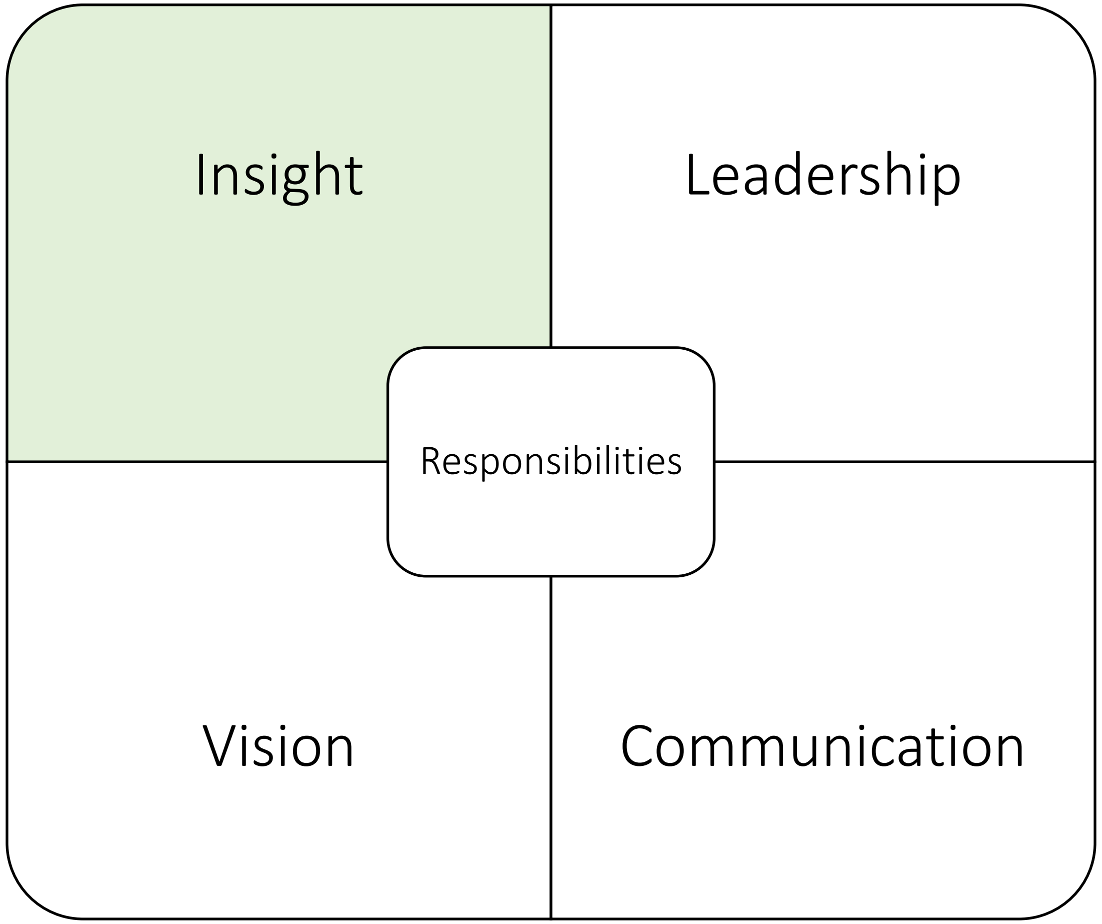
The model describes the essence of a system by exposing important details and significant constraints.



## Understand Tradeoffs

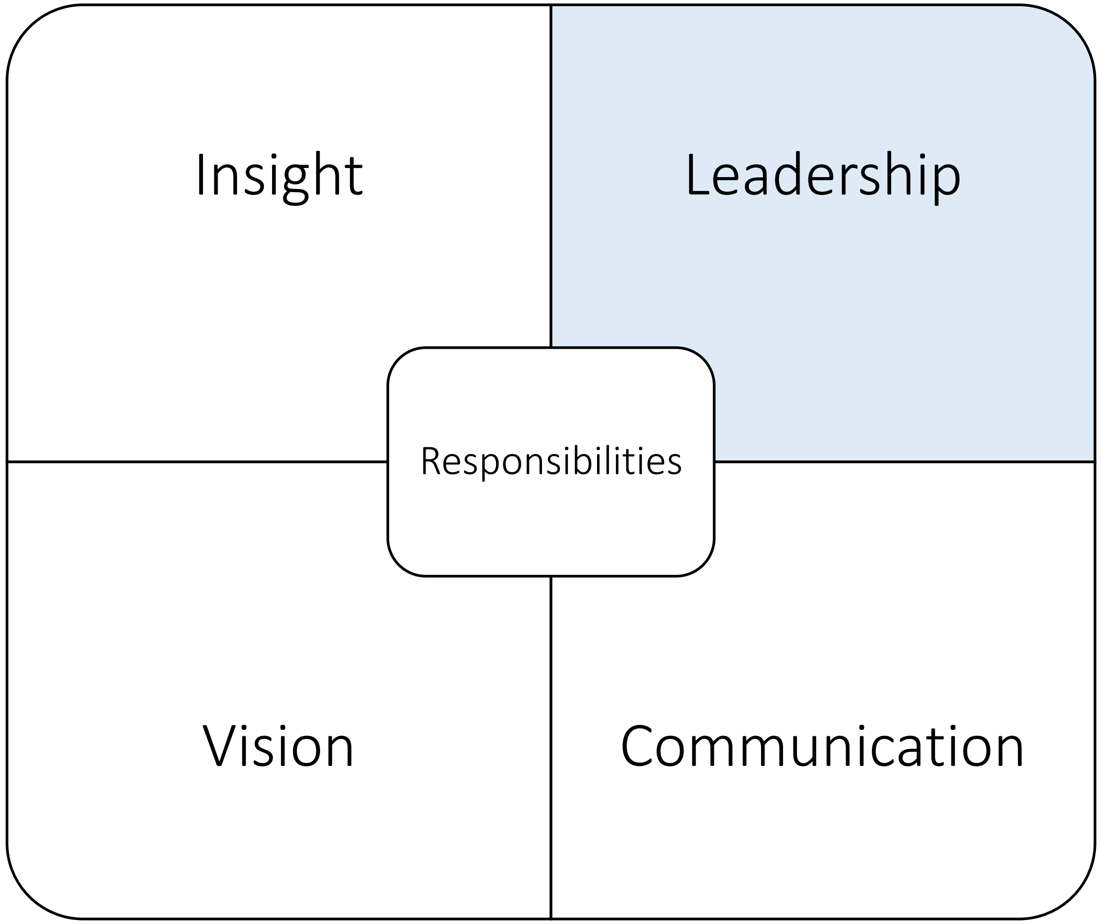
A great architect makes critical decisions in terms of implementation, operations, and maintenance. These decisions must be backed up by an understanding and evaluation of alternative options.

These decisions result in tradeoffs that must be well documented and understood by others.



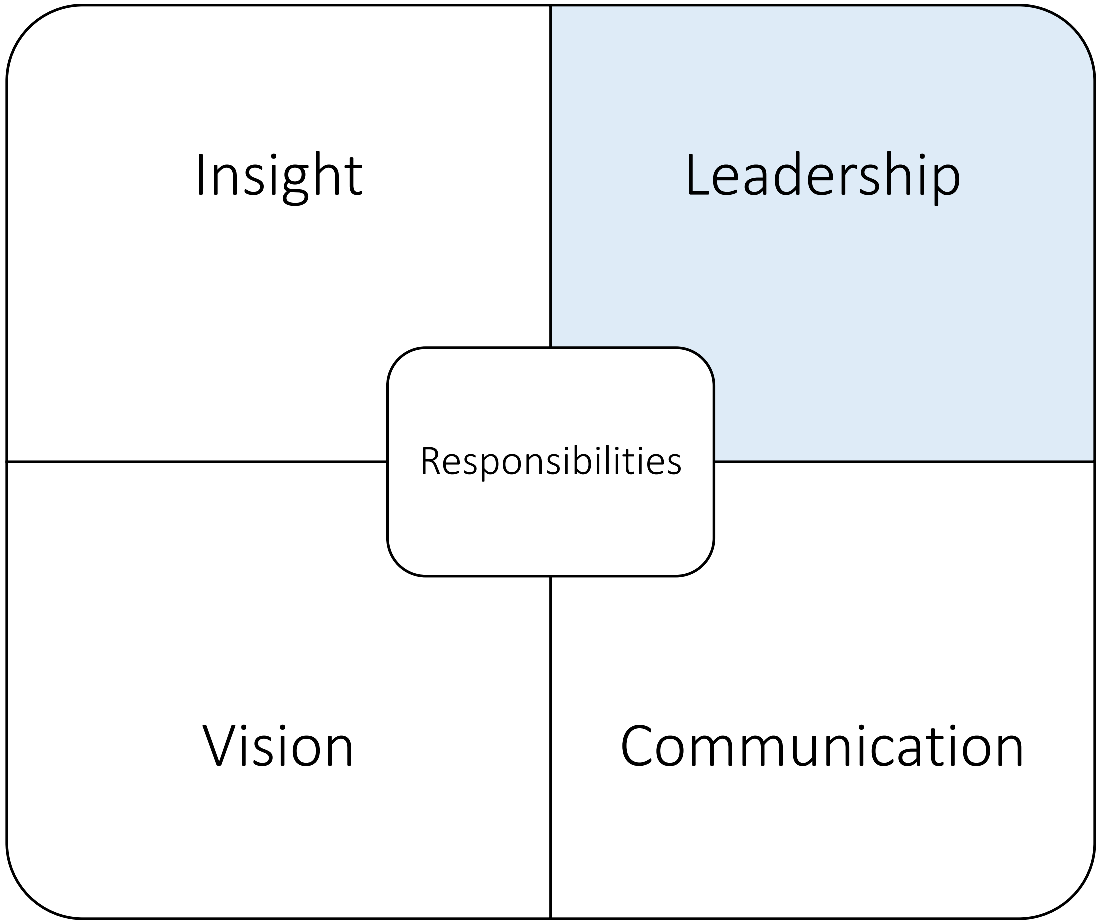
## Maintain Control

A great architect maintains control over the architecture lifecycle by continuously monitoring that the implementation adheres to the chosen architecture.



# Stay On Course

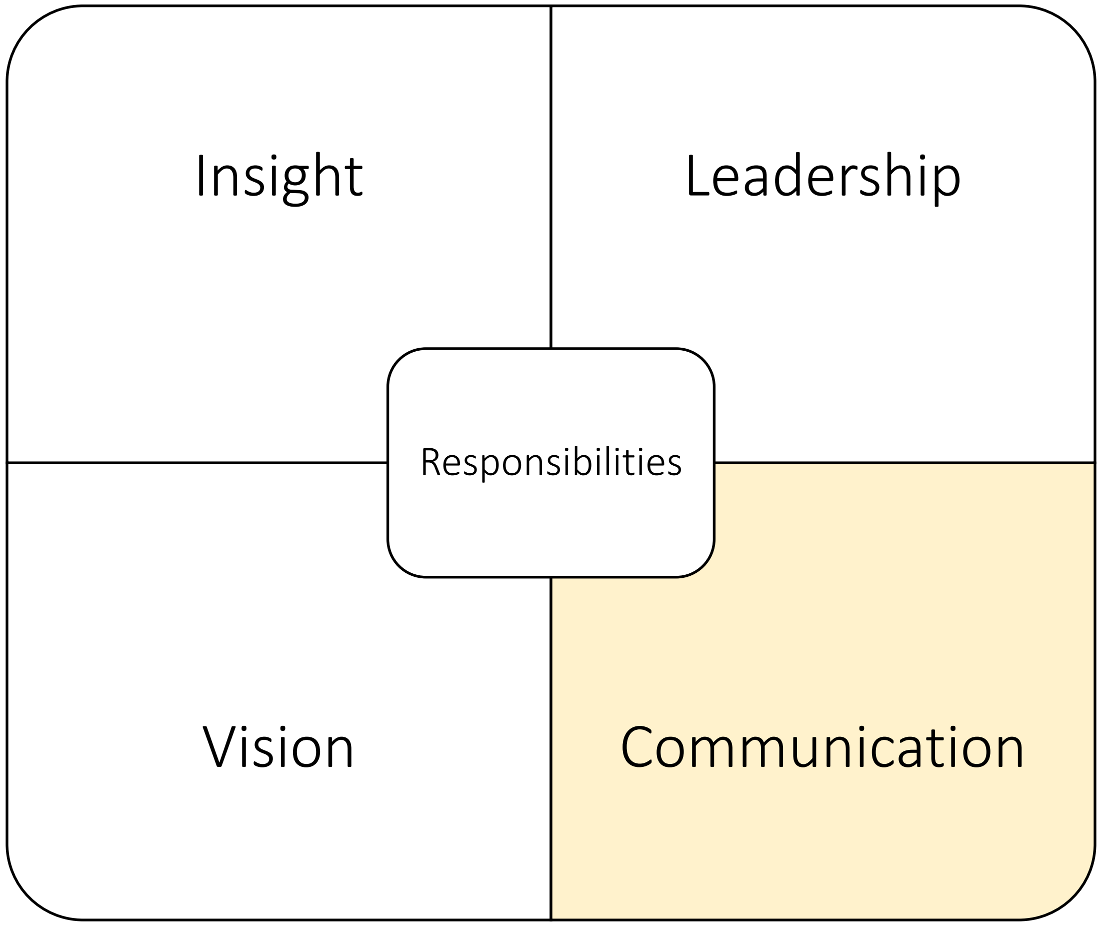
A great architect stays on course in line with the long term vision. When confronted with scope creep, the architect must know when to say no to some requests in order to say yes to others.



## Explain The Benefits

A great architect works closely with executives to explain the benefits and justify the investment in the chosen software architecture.

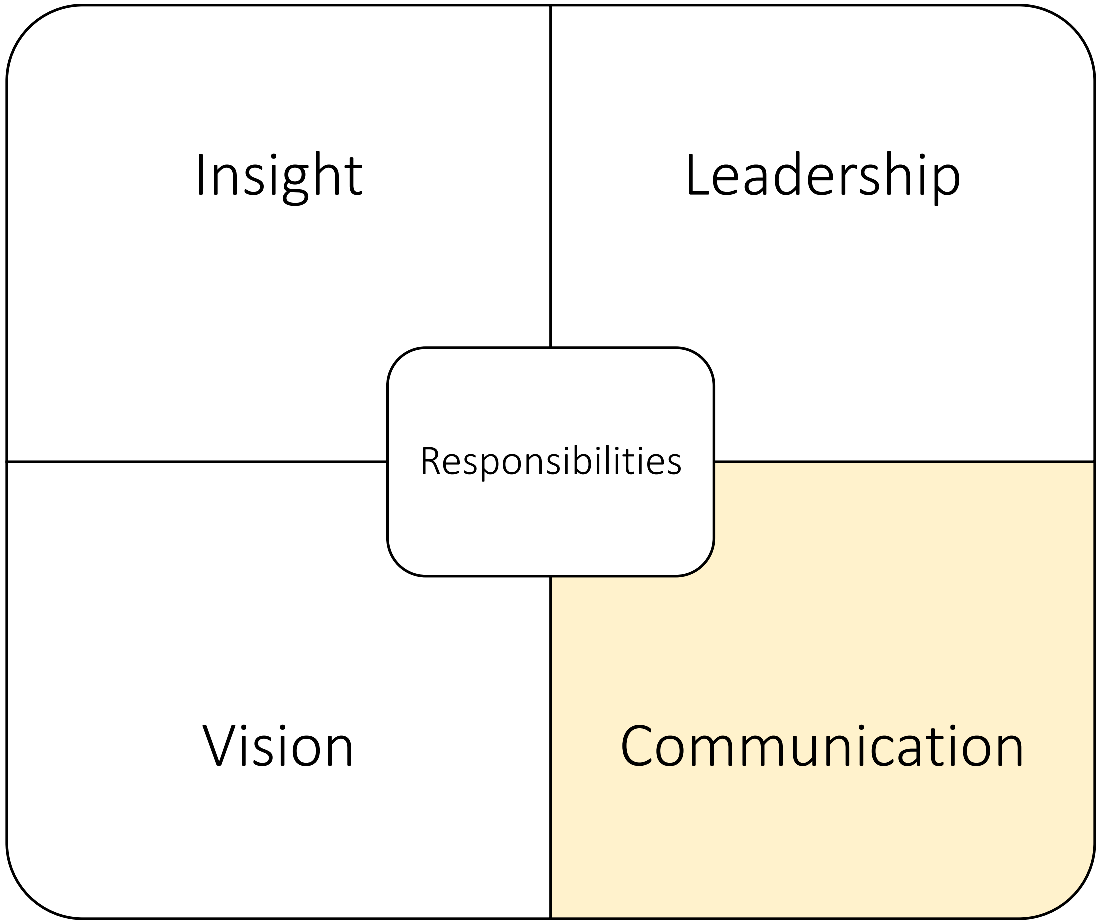
He or she must deliver results that have an impact on the bottom line.



## Inspire Stakeholders

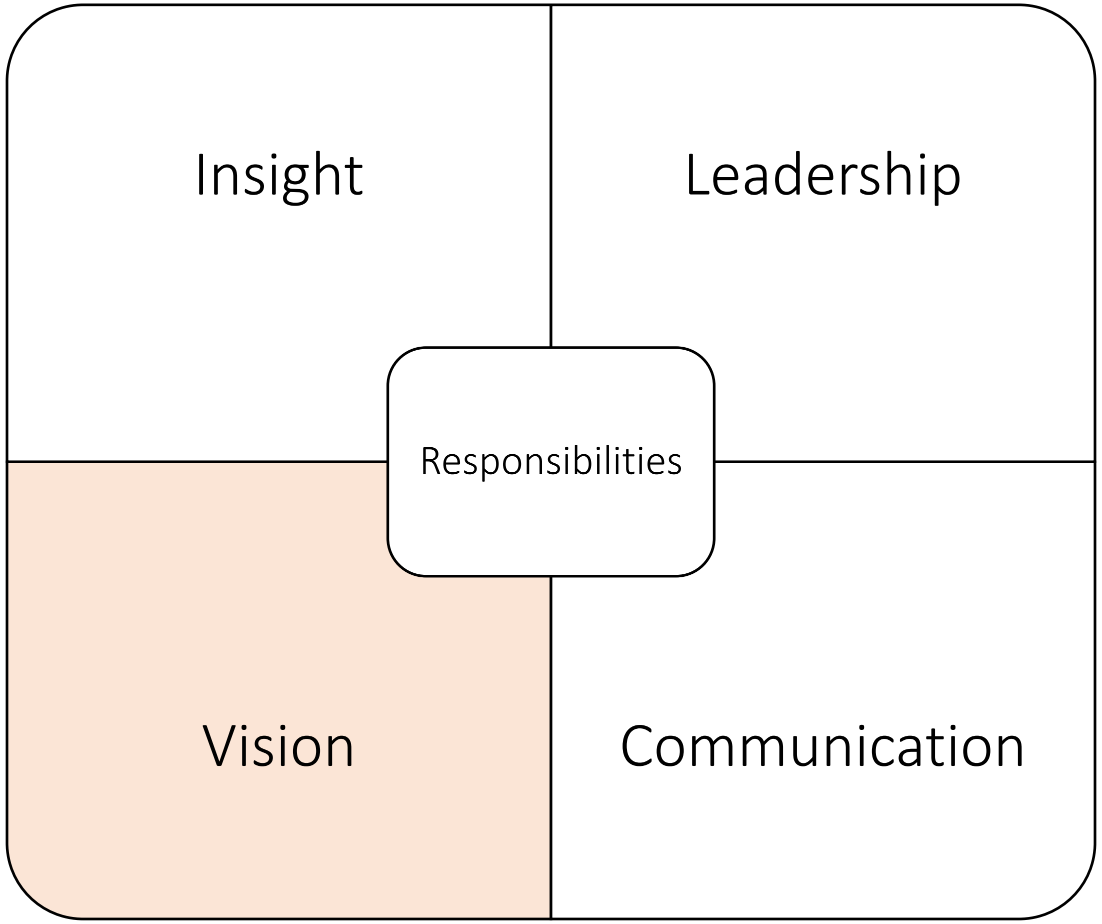
A great architect inspires, mentors, and educates the team about the solution architecture.

All stakeholders must be able to understand, evaluate, and reason about the software architecture.



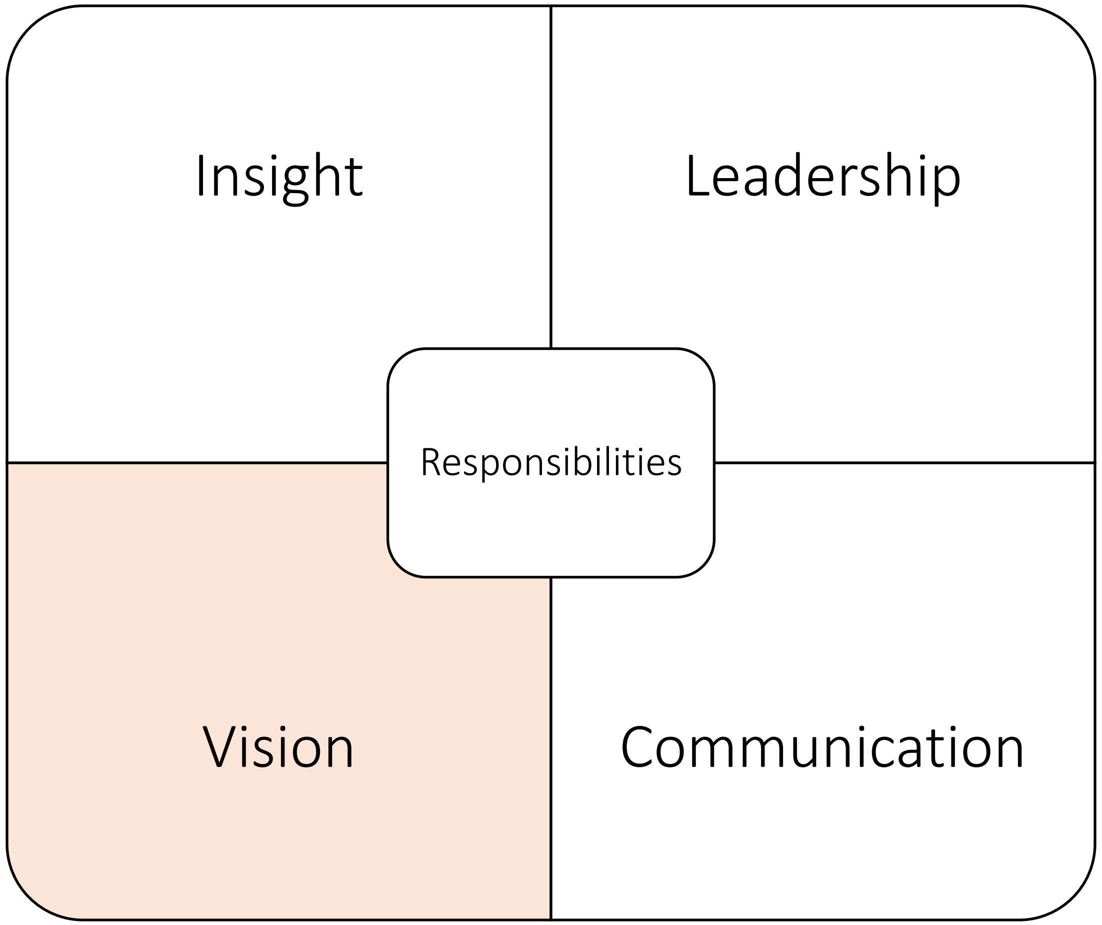
## Focus On The Big Picture

A great architect has a holistic view and always sees the big picture to understand how the software system works as a whole.



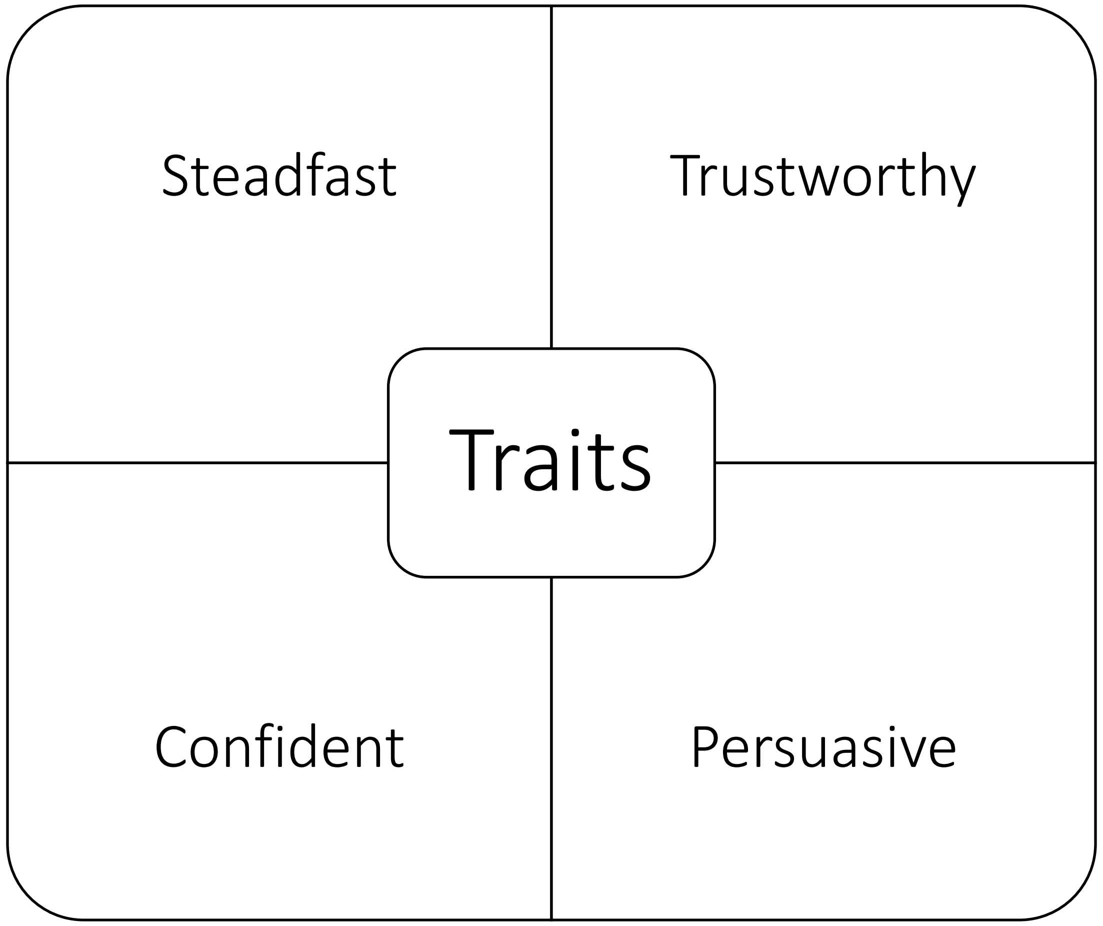
# Act As Change Agent

A great architect acts as an agent of change in organizations where process maturity is not sufficient for creating and maintaining the architecture.



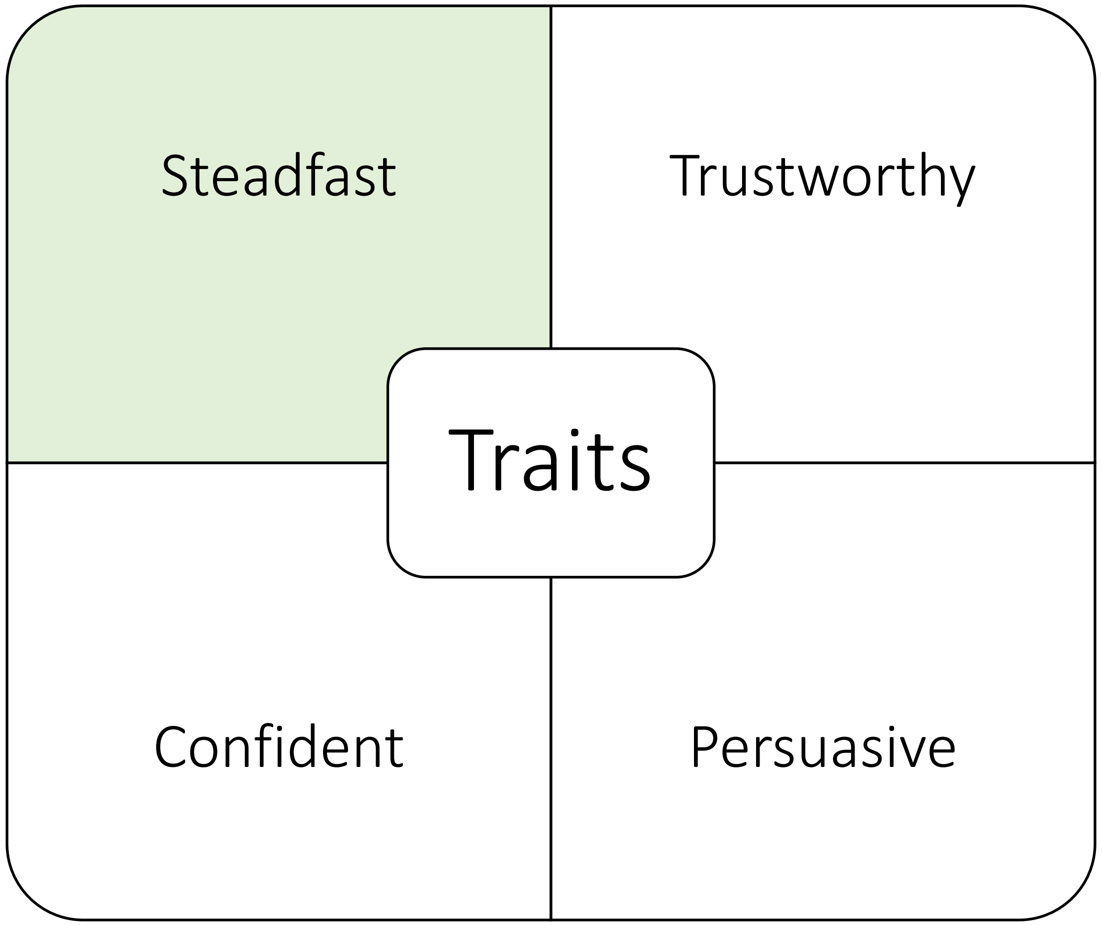
## Personality Traits Of A Great Architect

# Personality Traits Of A Great Architect



## Steadfast

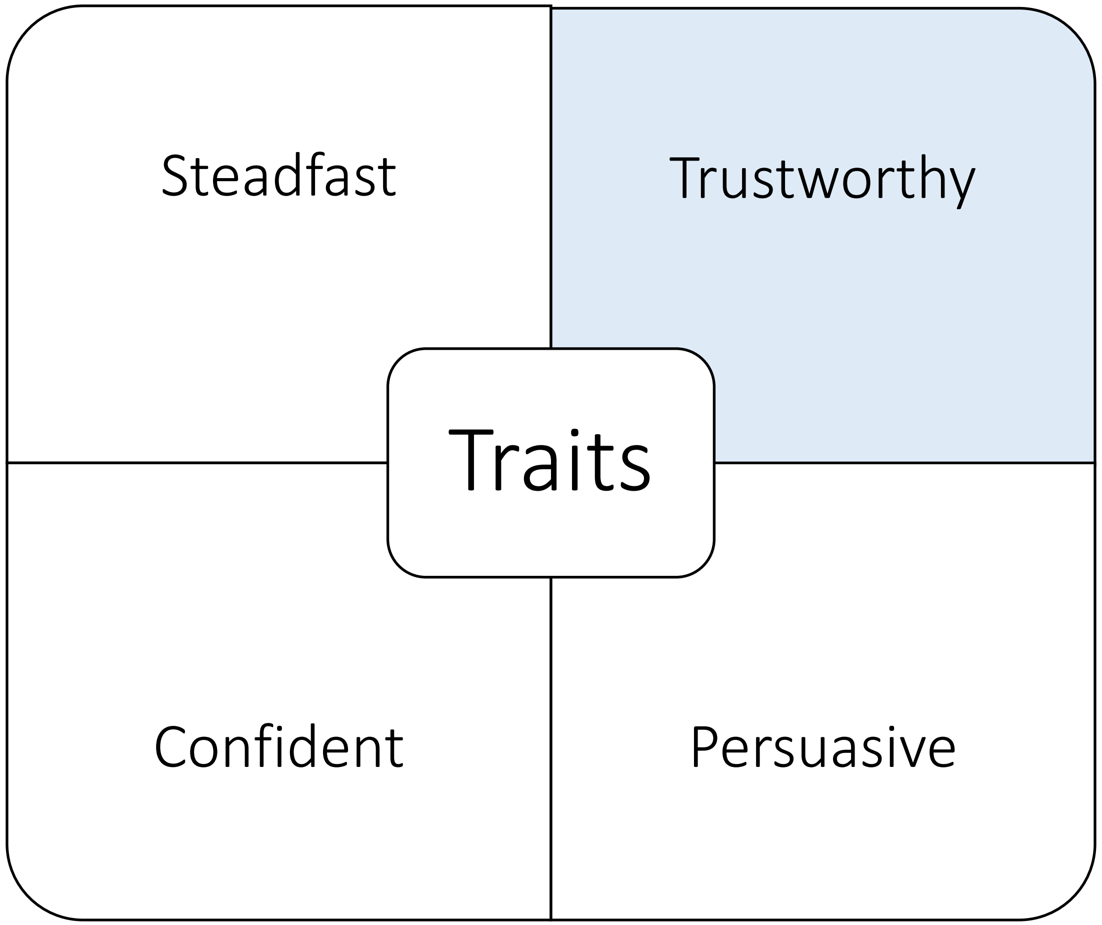
The only thing constant in Software Development is change itself. A great architect must be patient and resilient to adapt to the way stakeholders operate.



## Trustworthy

A great architect conveys a sense of credibility and trust, and must be perceived as successful.

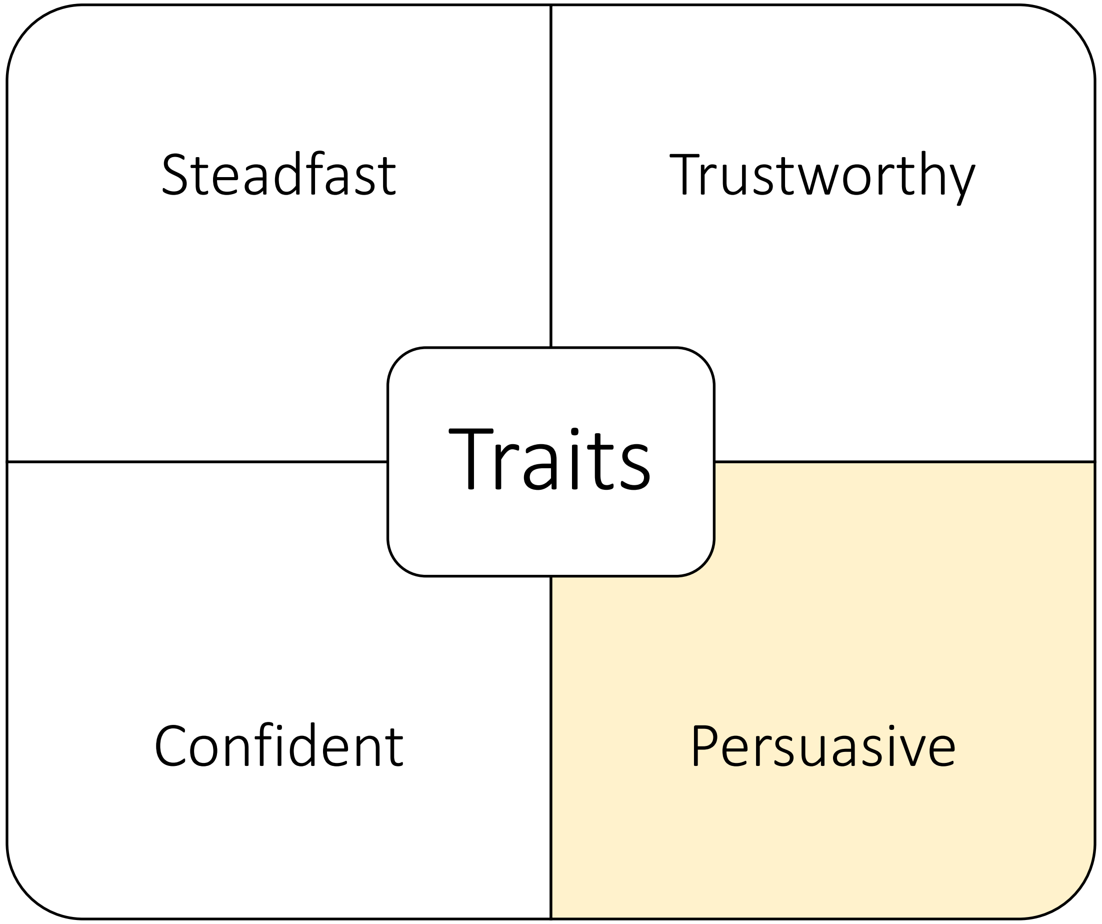
An architect can attain such status by prior successful experience, formal training in the field, and by his or her ability to deliver successful and relevant results.



## Persuasive

A great architect is a skilled and diplomatic negotiator. Principled negotiation is the tactic of choice for an architect to seek mutual cooperation with the project stakeholders.

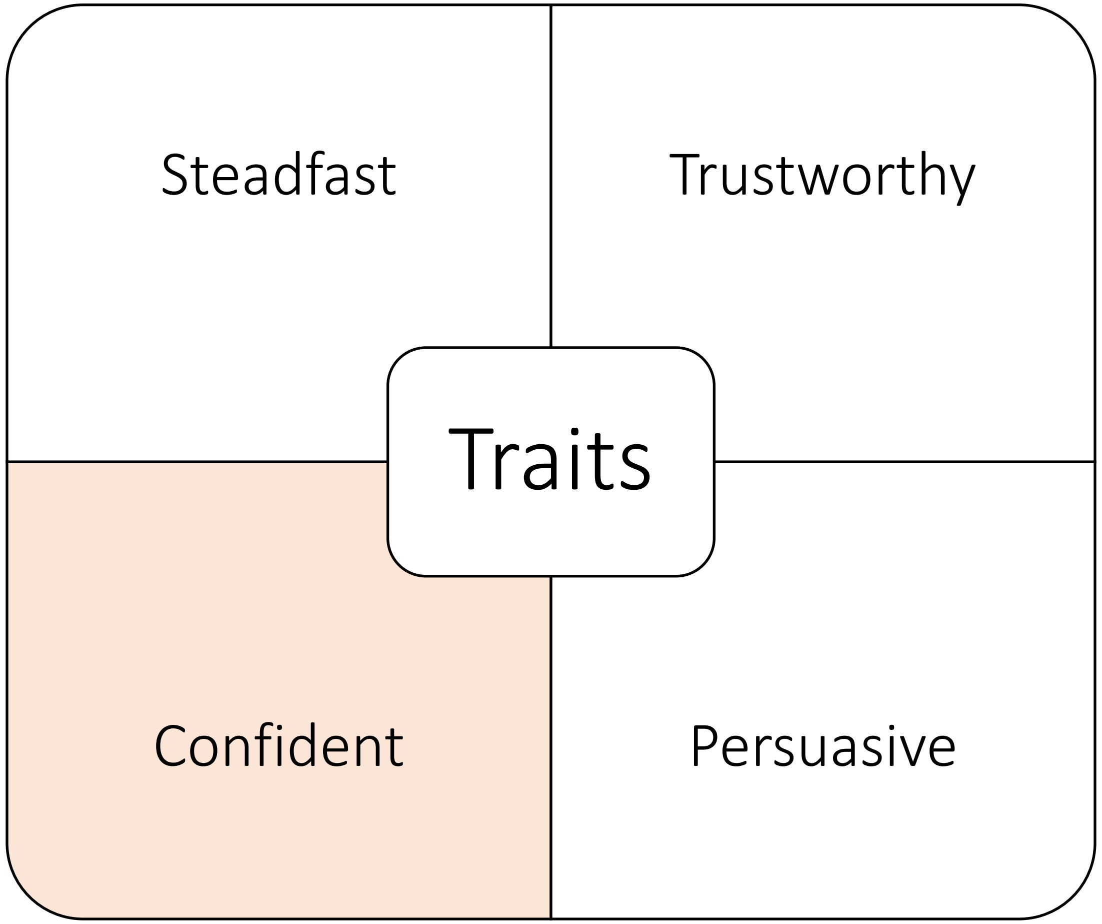
An architect will be expected to deliver better, faster, and cheaper, but must negotiate to decide which two our of three aspects will be considered first and under what conditions.



## Confident

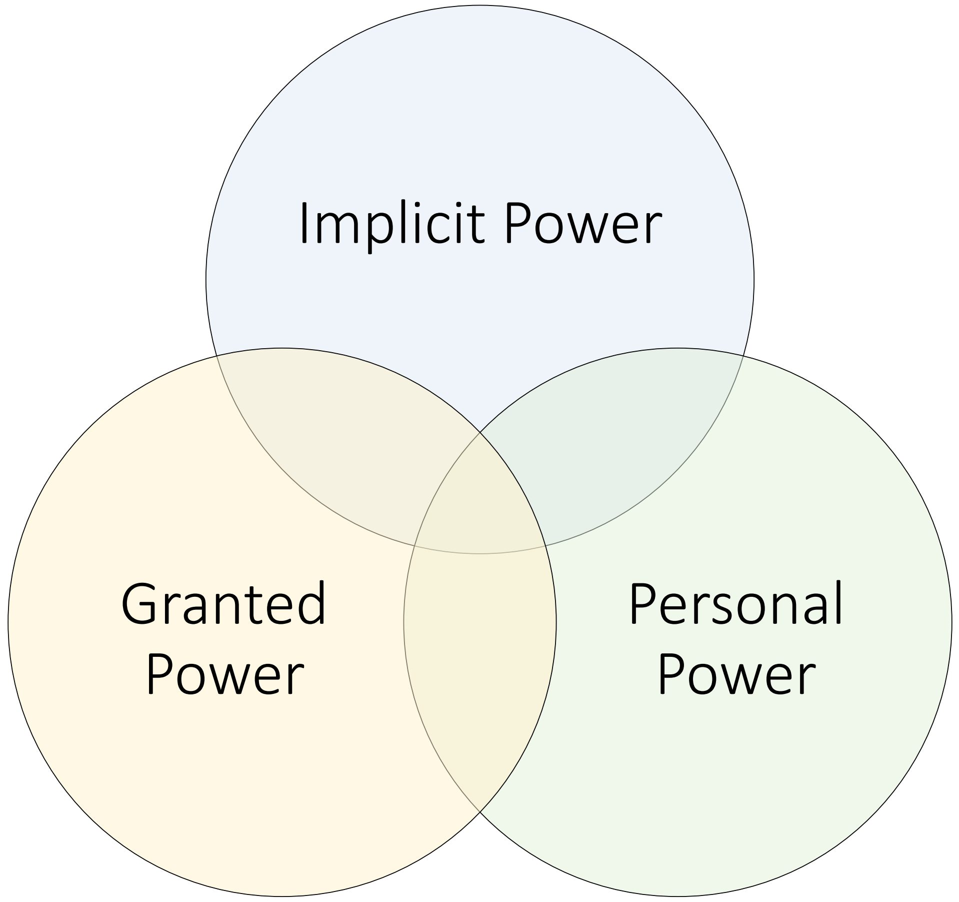
In a leadership position, attitude is everything. A great architect believes in his or her ability to perform.

An architect must have a passion for success.

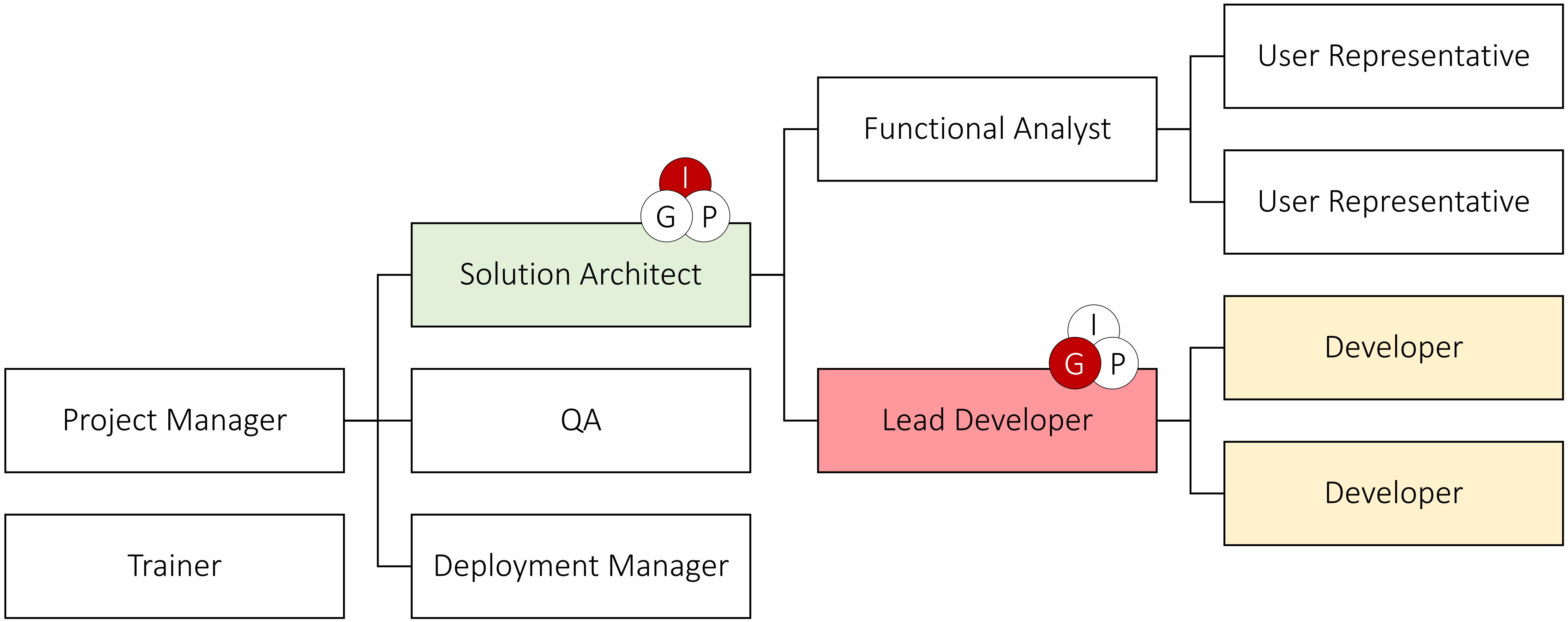


## Pitfalls For New Architects

## The Three Types Of Power



# Lead Developer Loses Trust



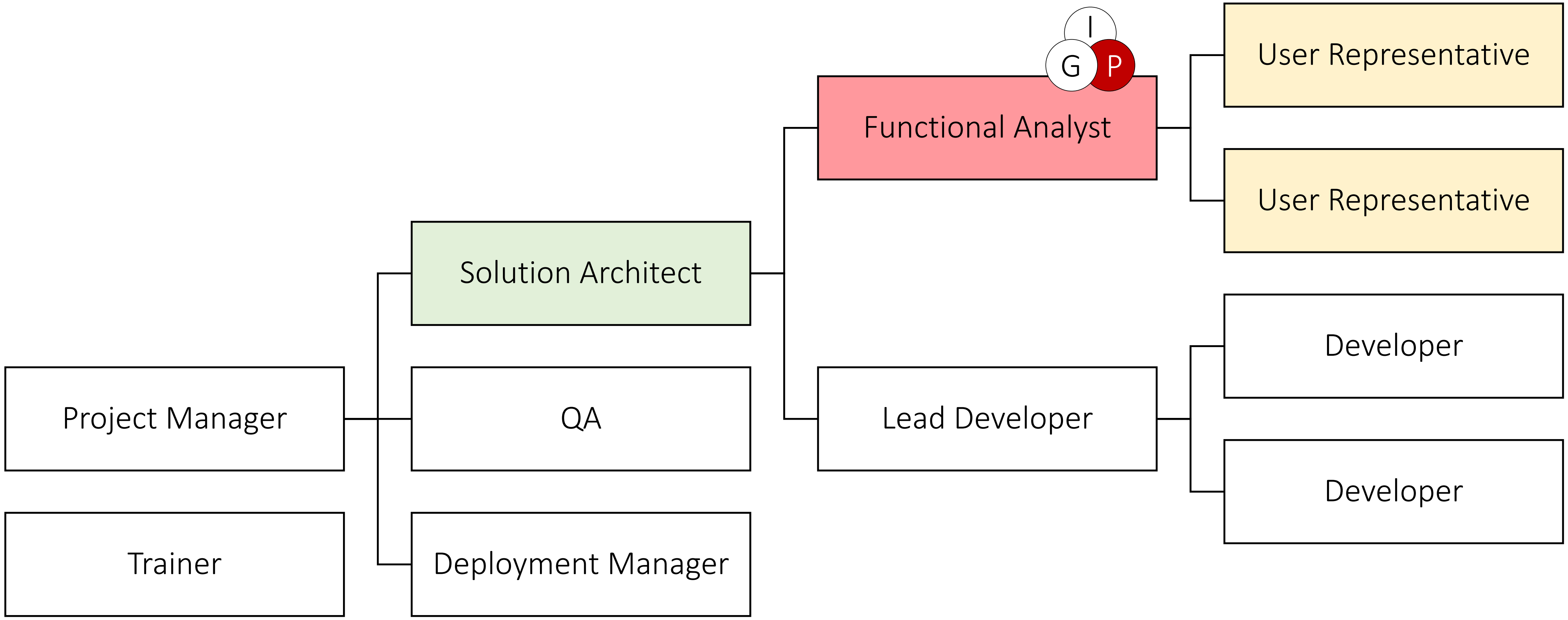
## Lead Developer Loses Trust

Endless discussions, implementation starts to deviate from architecture.

**Remember:** *your job is to make your LD succeed!*

- Empower your LD
- Always present an united front
- Be open to feedback
- Change architecture if needed

# Functional Requirements Are Invalid

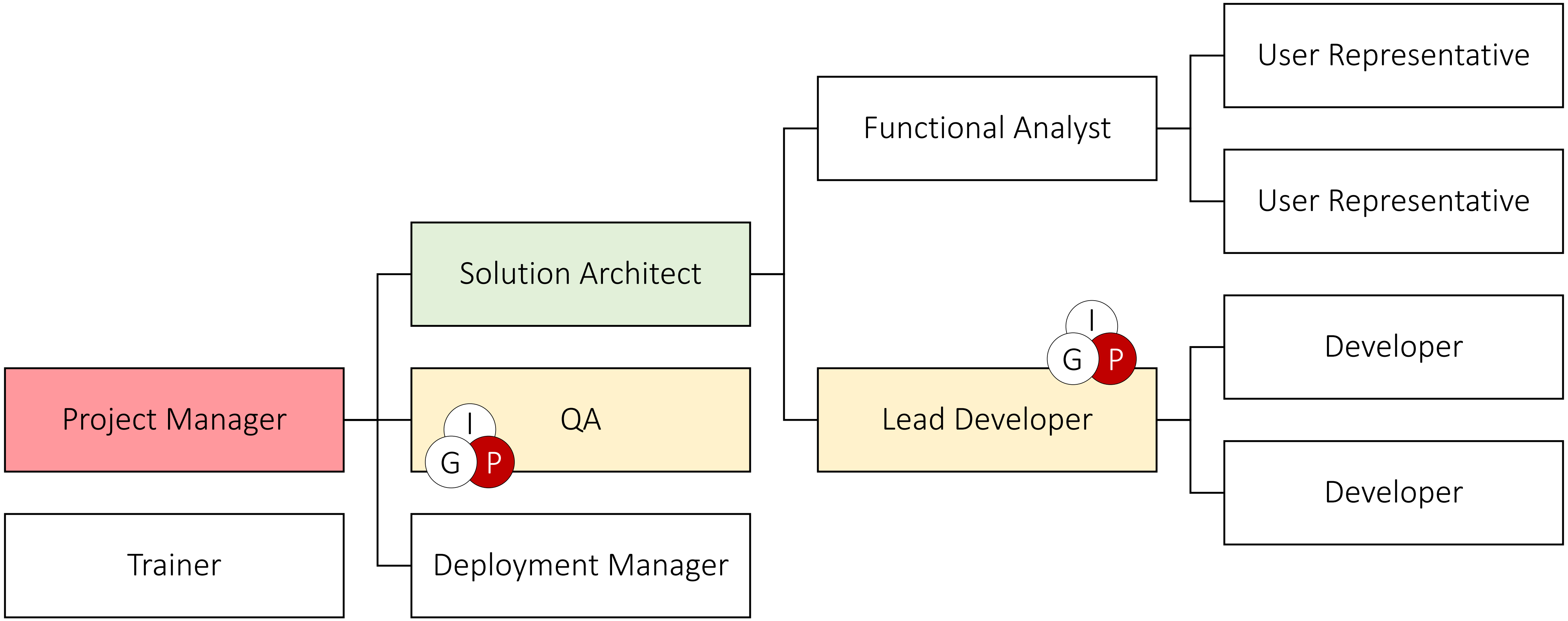


## Functional Requirements Are Invalid

For some reason the FA has no engagement with URs, and passes invalid specs to architect. **Beware:** *This will come to bite you!*

- Find the reason
- Try to restore the personal power of the FA
- Escalate up if needed

# The Planning Is Too Optimistic



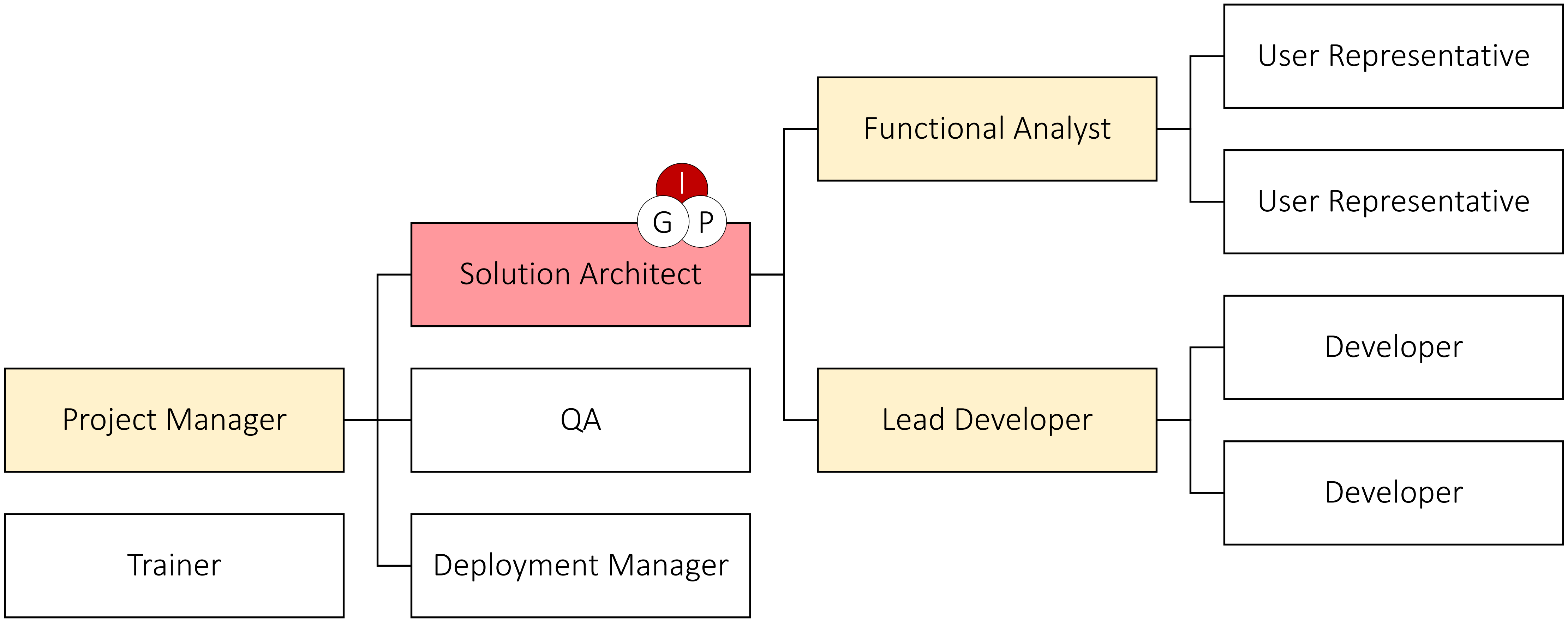
## The Planning Is Too Optimistic

The PM operates in isolation, far away from the dev team. This puts pressure on the LD and QA. Both feel disempowered.

**Remember:** *your job is to make your LD succeed!*

- Bring the PM in on the dev team
- Find the reason: optimistic PM, naïve LD, or slow devs?
- Coach LD / Restructure devs / Readjust schedule / Alter planning

## The Architect Role Holds No Power



## The Architect Role Holds No Power

The organization doesn’t understand the architect role. SA is not at the center of decision making. Implicit role power is missing.

Beware: *the project will probably fail!*

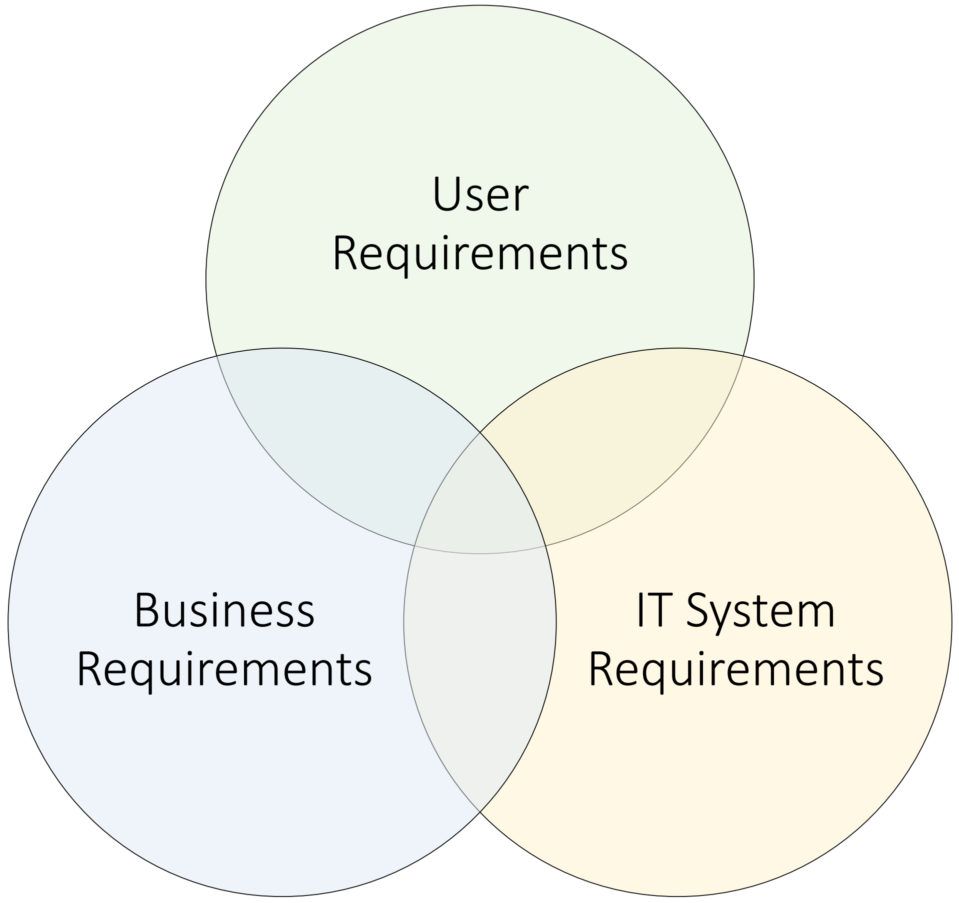
- Educate the C-section, be a change-maker
- Team up with PM, QA, FA, and LD
- Become the owner of the entire process
- Leave if the organization is unwilling to change

## What Is Software Architecture?

## What Is Software Architecture?

Software architecture is a structured solution that meets all requirements, while optimizing common quality attributes such as performance, security, and manageability.

# Software Architecture Requirements



## What Composes An Architecture?

- The structural elements and interfaces composing the system
- How these elements behave in collaboration
- The composition of elements into larger subsystems
- The architectural style that guides this composition

Also covers functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and aesthetic concerns

## What Are Software Architecture Goals?

- Document high-level structure
- Do not go into implementation details
- Minimize complexity
- Address all requirements
- Be compatible with all use cases and scenarios

## Software Architecture Design Tips

- Build to change instead of building to last
- Use models, but only to analyze and reduce risk
- Use visualizations to communicate and collaborate
- Identify and research critical points of failure

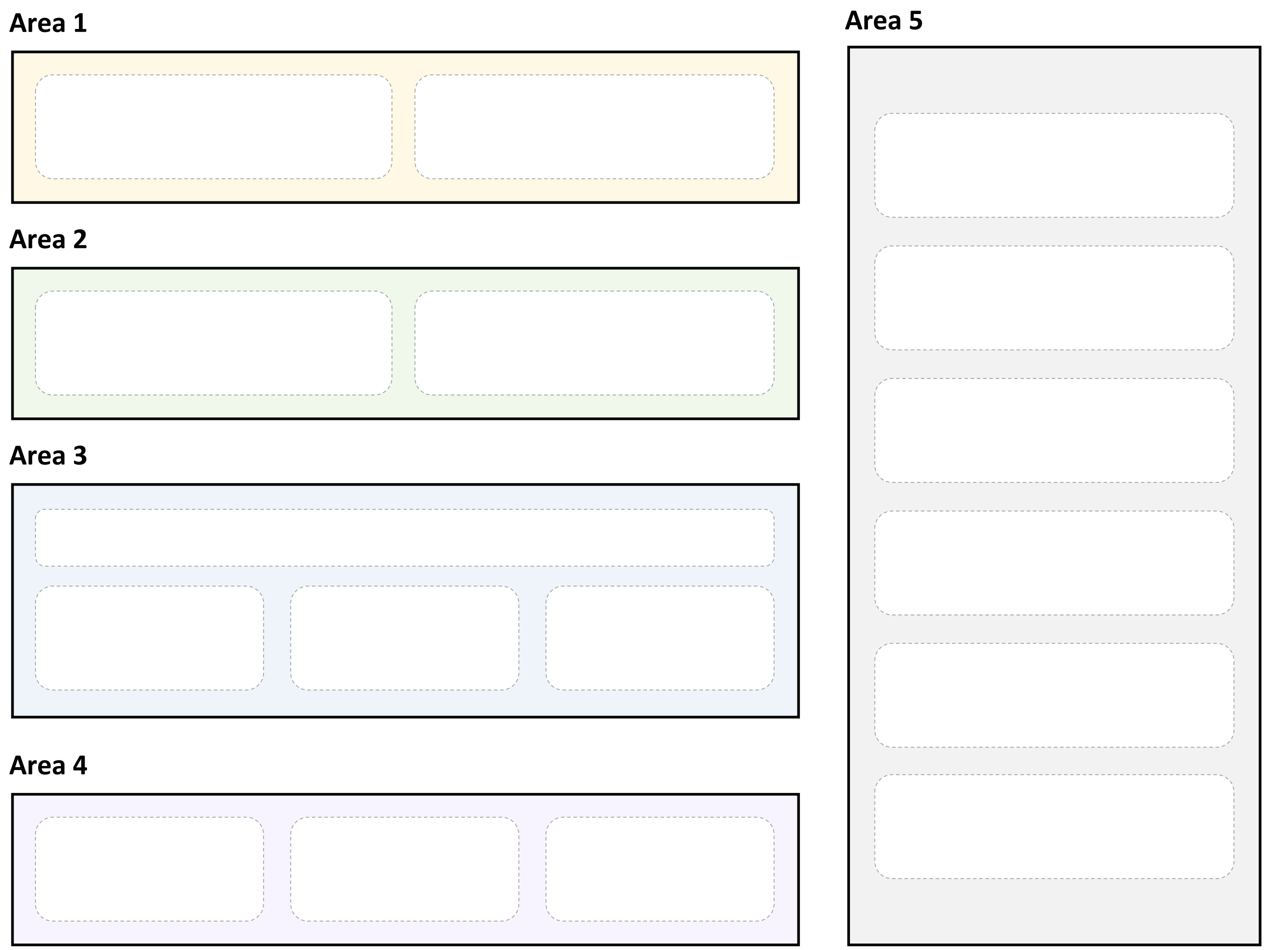
## Key Principles Of Software Architectures

## Software Architecture Goal

*The goal of a software architect is to minimize complexity.*

This can be accomplished by <u>separating</u> the design into different areas of concern

# Areas Of Concern



## Key Principles

1. Separation of concerns
2. Single responsibility principle
3. Principle of least knowledge
4. Don’t repeat yourself
5. Minimize upfront design

## Guidelines For Software Architectures

## General Guidelines

- Use consistent patterns in each layer
- Do not duplicate functionality
- Prefer composition over inheritance
- Establish a code convention

## Layer Guidelines

- Separate areas of concern
- Define communication between layers
- Use abstraction to loosely couple layers
- Don’t mix different types of components in a layer
- Use a consistent data format within a layer

## Component Guidelines

- No component should rely on the internals of another
- Do not mix roles in a single components
- Define clear contracts for components
- Abstract system wide components away from other layers

## Introduction To UML

## What Is UML?

*“A general-purpose, developmental, modeling language [...] that is intended to provide a standard way to visualize the design of a system”*

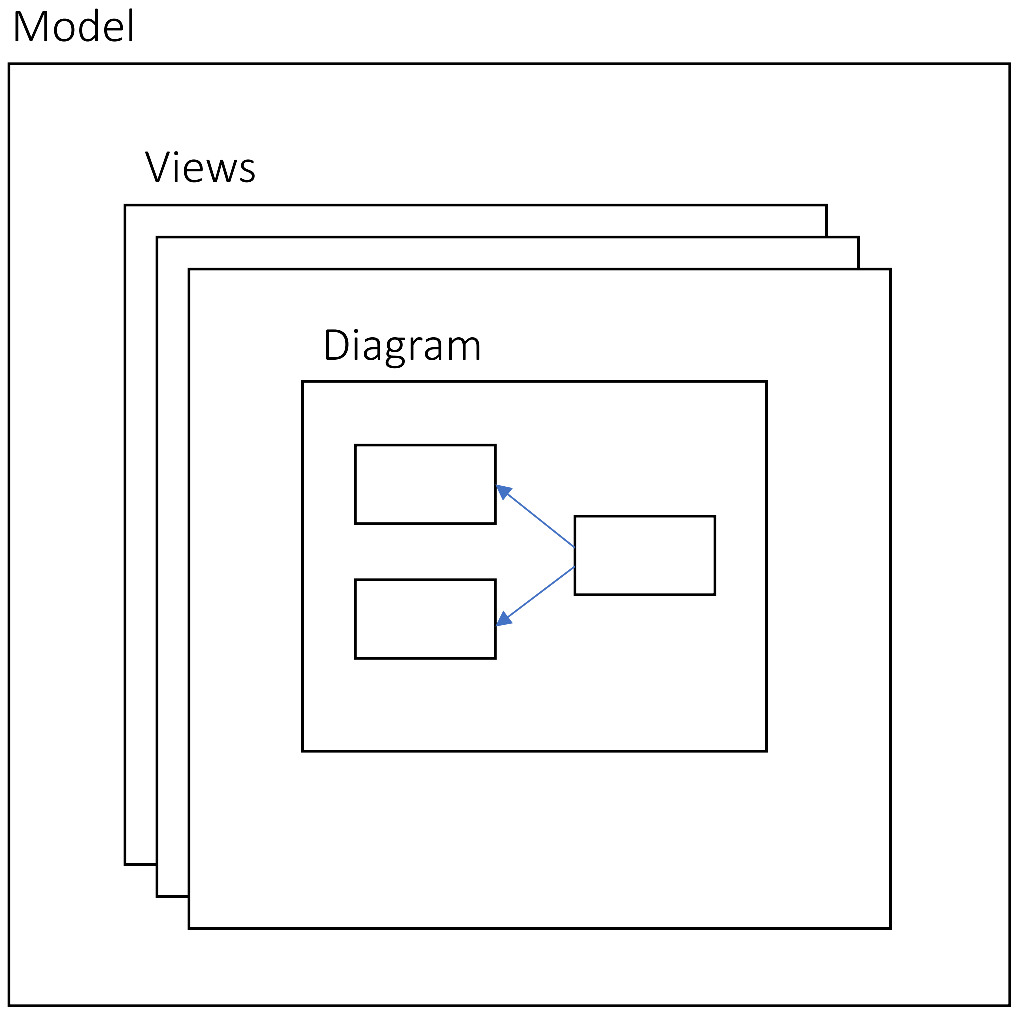
- Wikipedia

## UML Attributes

- Visual
- Abstract
- Descriptive
- Standard
- Supports Code Generation
- Supports Reverse Engineering

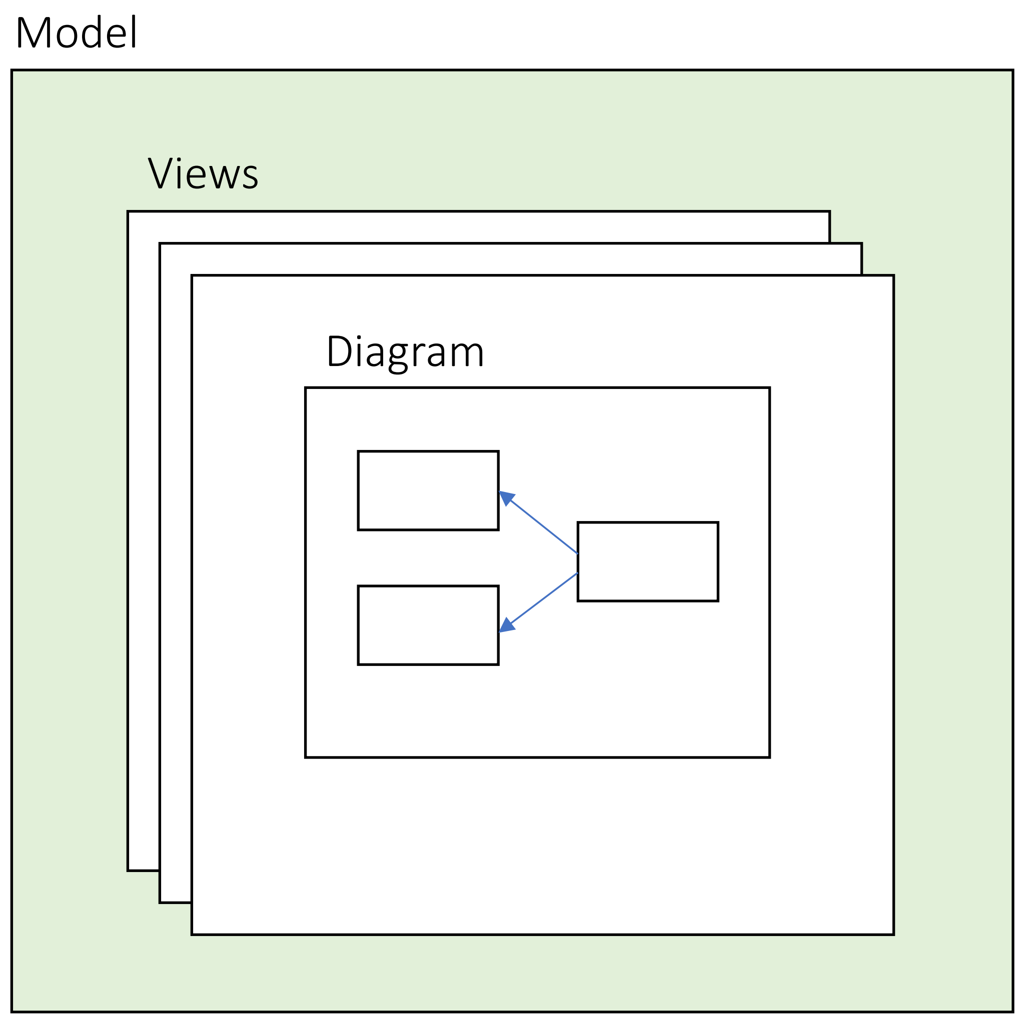
# UML Design Elements

- Models
- Views
- Diagrams



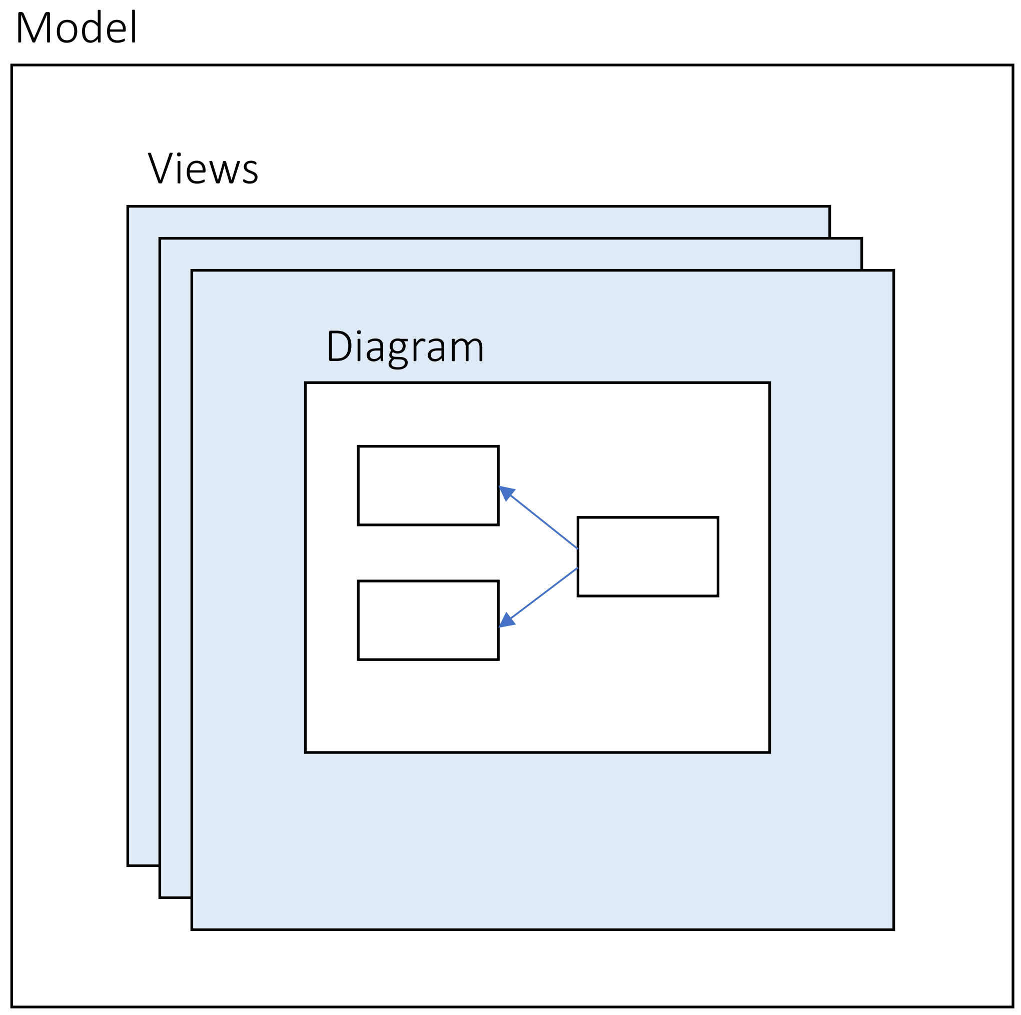
# UML Models

- Business System Model
- IT System Model



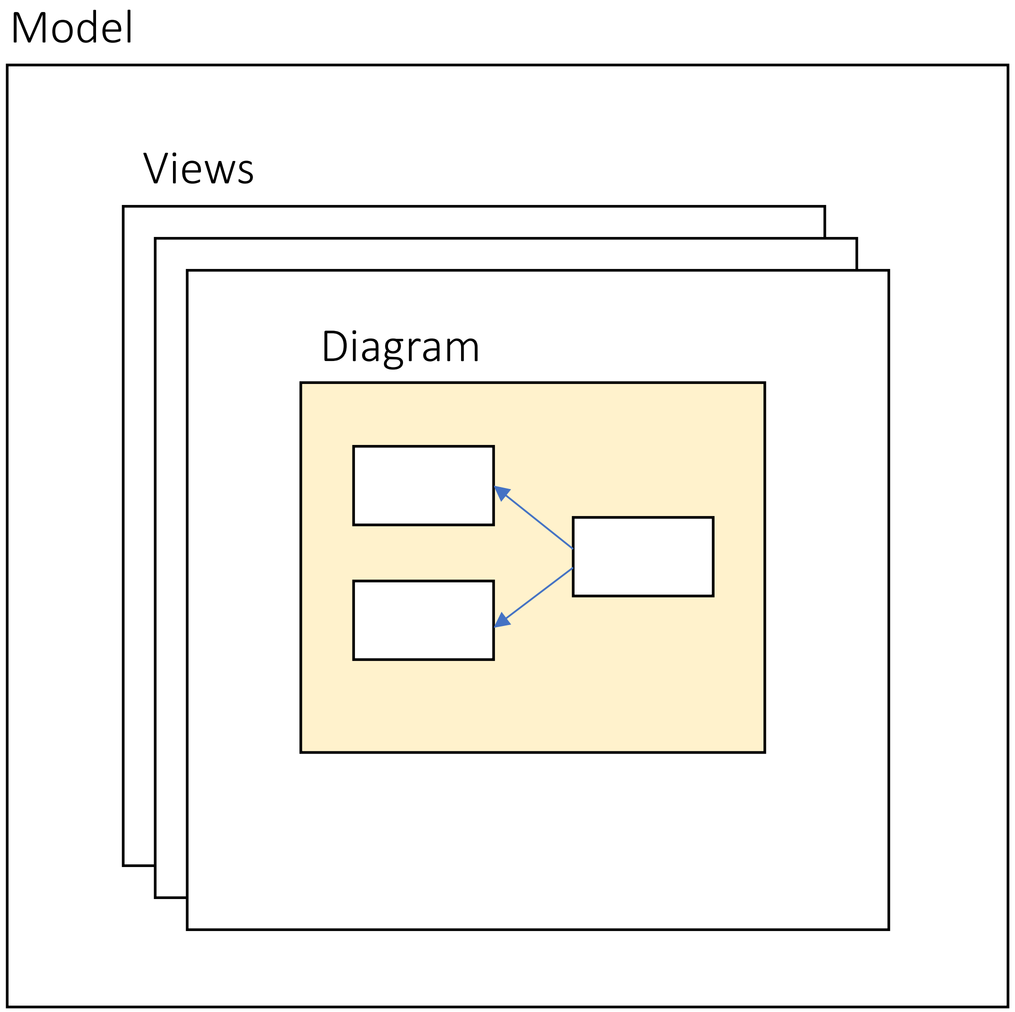
## UML Views

- Business System:
    - External View
    - Internal View
- IT System:
    - Static View
    - Dynamic View



## UML Diagram

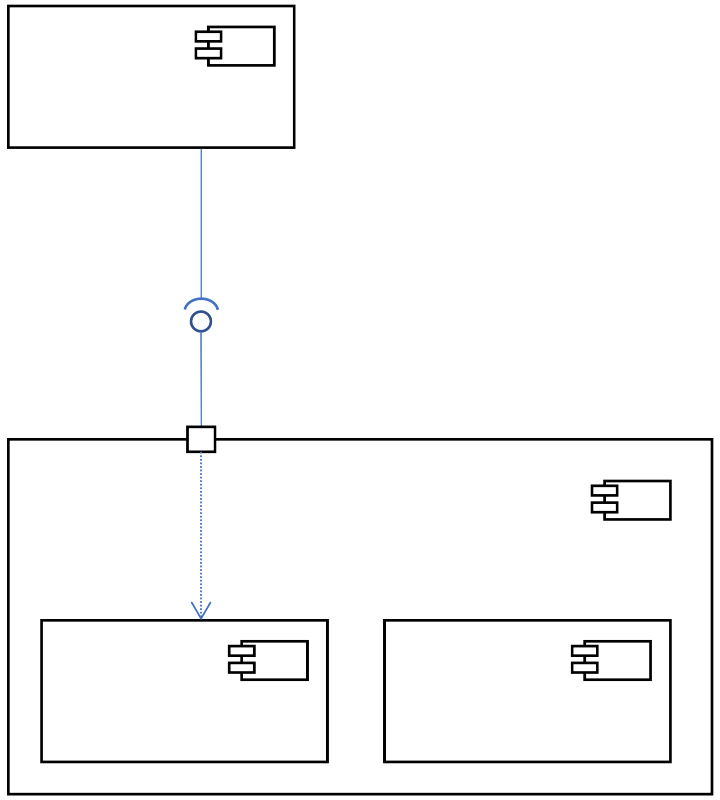
- Component Diagram
- Class Diagram
- Sequence Diagram
- State Diagram
- Activity Diagram
- Layer Diagram
- Use Case Diagram
- …



## Seven Popular UML Diagrams

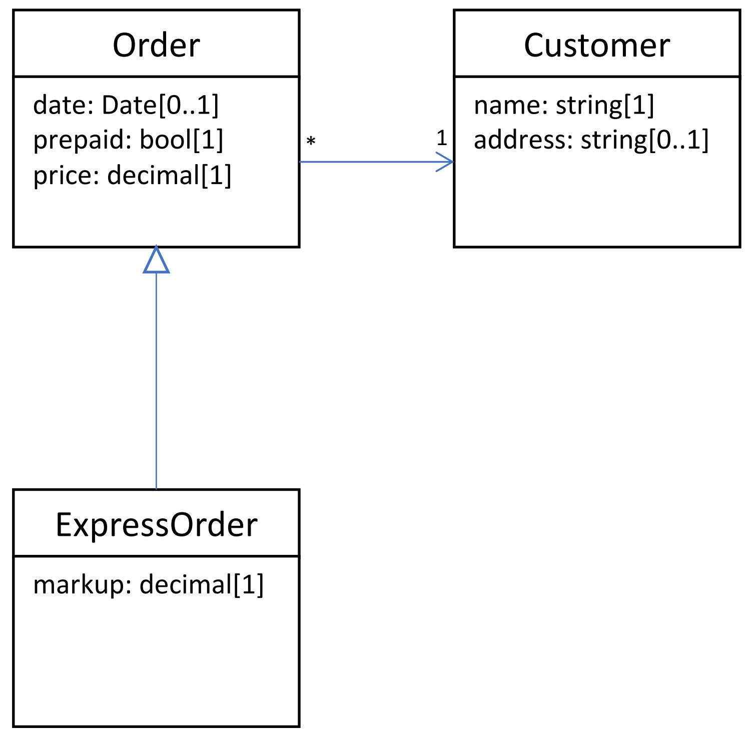
## The Component Diagram

- Shows components
- Shows implemented and required interfaces
- Components can be nested



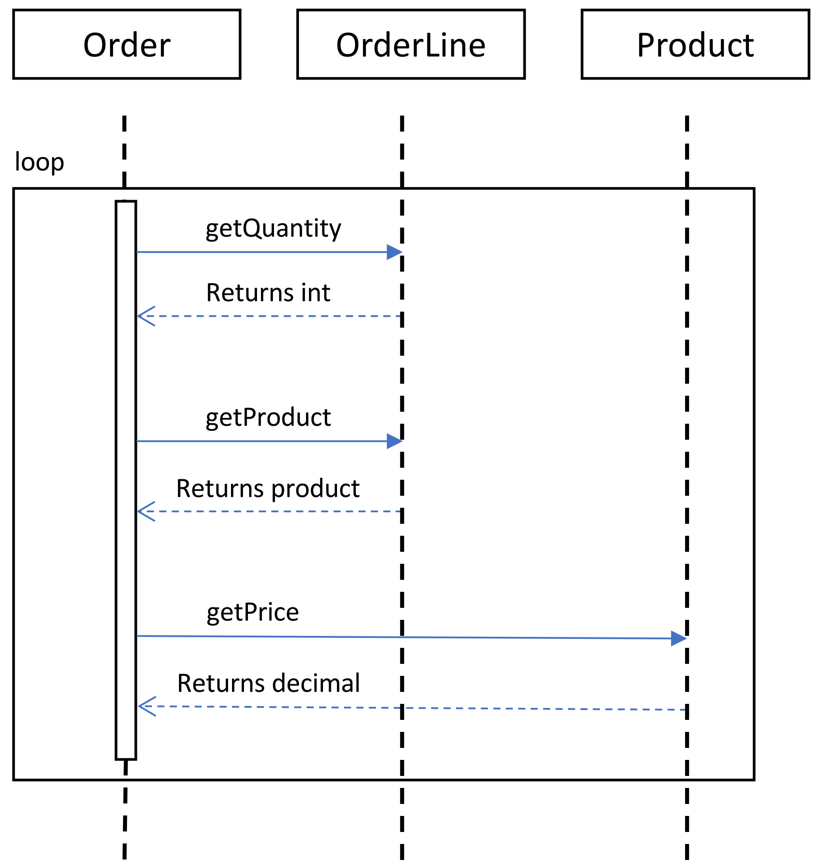
## The Class Diagram

- Shows classes
- Shows methods and fields
- Shows associations, generalizations, and cardinality



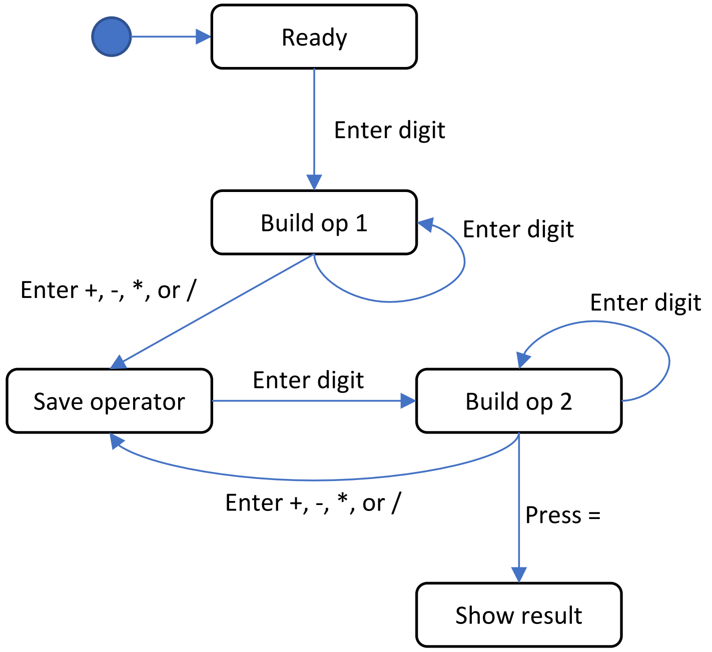
## The Sequence Diagram

- Shows call sequence
- Shows calling class, called method, and return data type
- Can depict loops



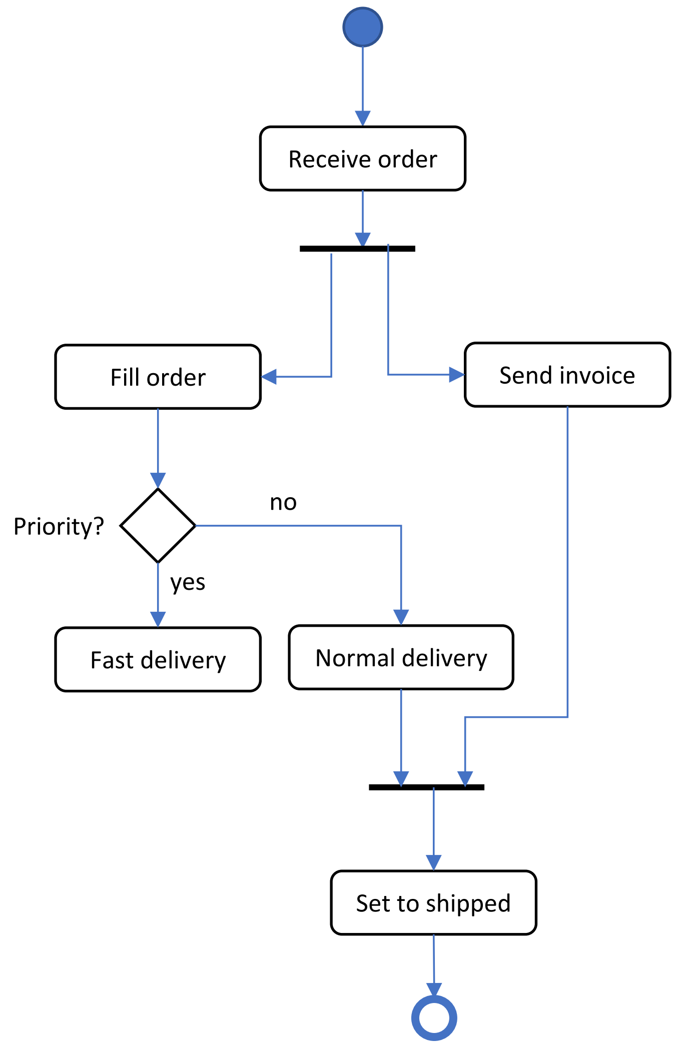
# The State Diagram

- Shows states or activities
- Shows allowed transitions
- Can be nested
- Can depict internal activities



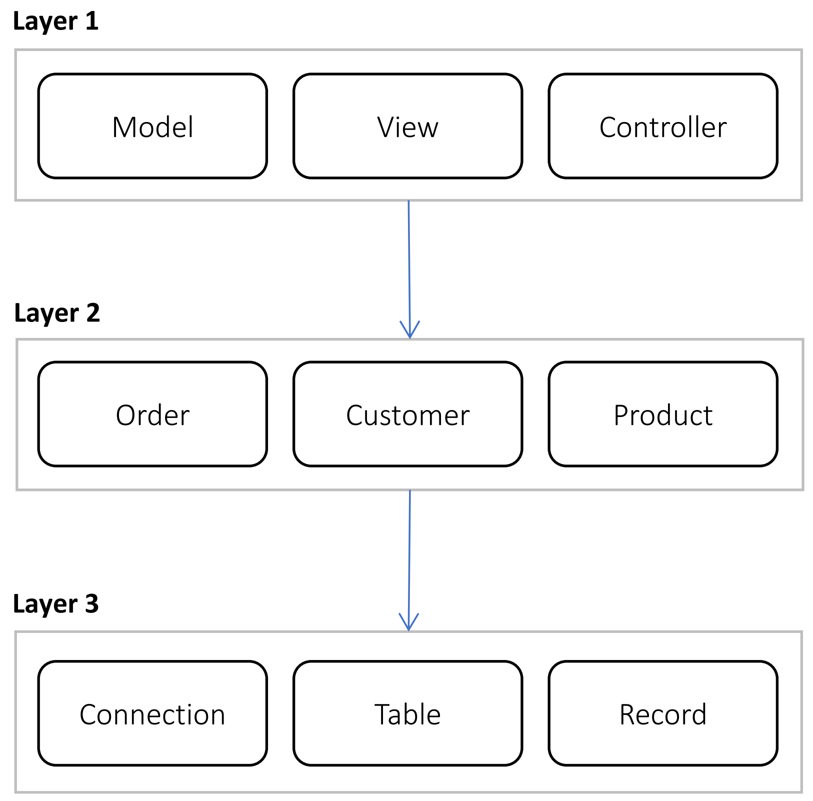
## The Activity Diagram

- Shows process or workflow
- Can be nested
- Can show concurrent actions
- Can have swim lanes



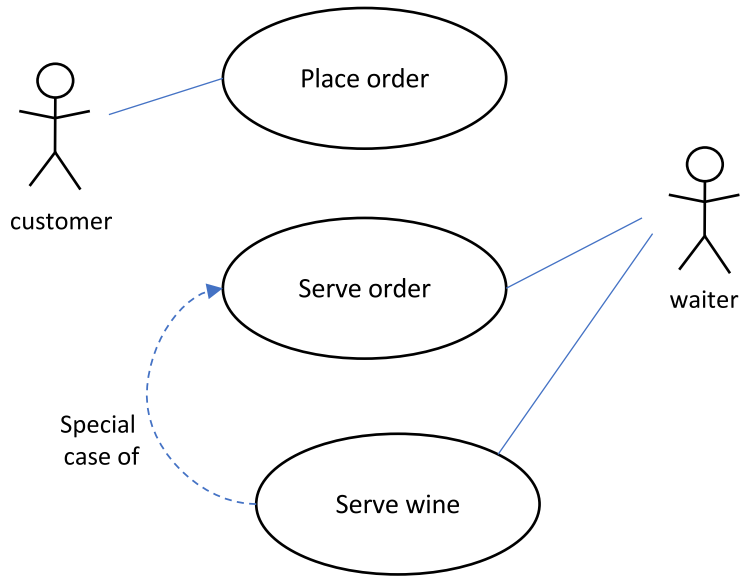
# The Layer Diagram

- Non-standard, invented by MS
- Shows areas of concern
- Shows references between areas
- Can be validated



# The Use Case Diagram

- Shows actors
- Shows use cases
- Binds actors to use cases
- Can depict generalizations



## Designing Solution Architectures With UML

## UML Diagrams In Architectures

| Solution Architecture Element | UML Diagram |
|---|---|
| Functional Requirement | Use Case Diagram |
| Structural Elements, Composition | Class Diagram<br>Component Diagram |
| Structural Elements, Collaboration | Sequence Diagram<br>Activity Diagram<br>State Diagram |
| Areas Of Concern | Layer Diagram |

## UML Design Strategies

- UML As Sketch
- UML As Blueprint
  - **Forward Engineering**: use diagram to generate code
  - **Reverse Engineering**: build diagram from existing code
- UML As Validation
  - Validate implementation against diagram

## UML In Visual Studio Ultimate 2017

- Supports Component, Class, Sequence, Activity, Layer, and Use Case Diagrams
- Under version control, machine-readable, uses T4 for code gen
- Forward engineering supported for Class Diagrams
- Reverse engineering supported for Class, Sequence Diagrams
- Validation supported for Layer Diagrams

# Missing In Visual Studio Ultimate 2017

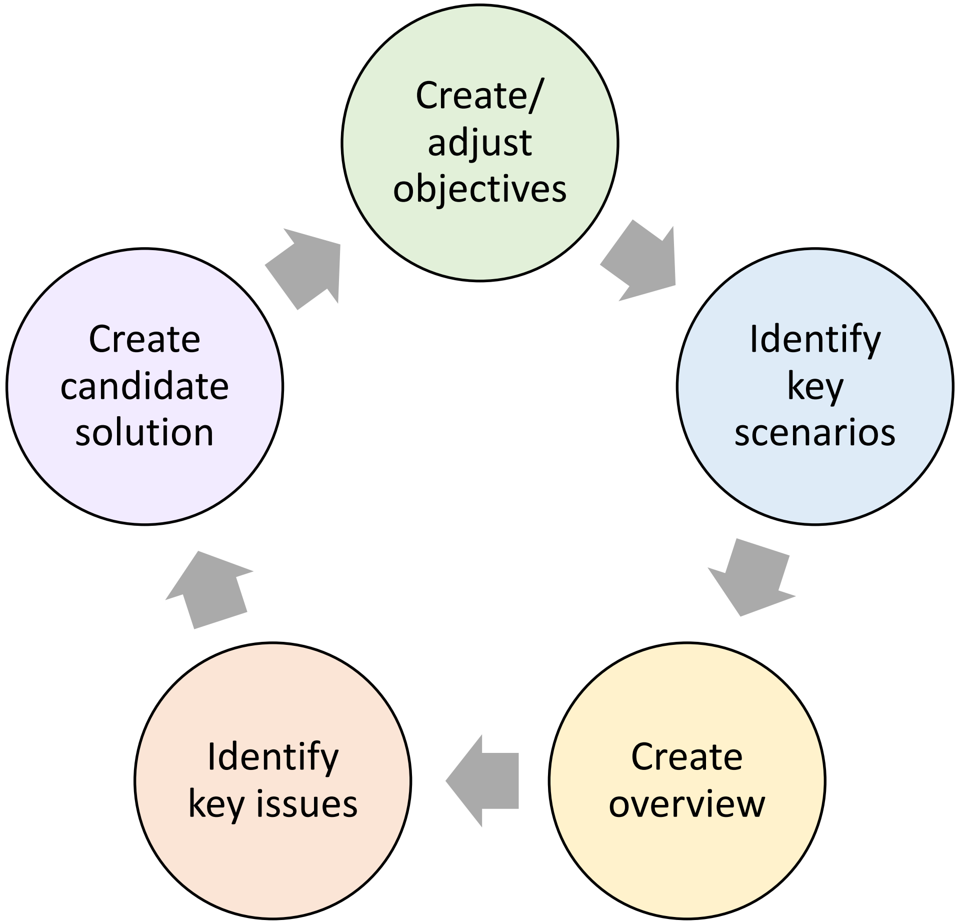
- Support for State Diagrams
- Forward engineering Layer, Sequence, Activity(*) Diagrams
- Reverse engineering Layer Diagrams
- Validating Class, Sequence Diagrams

(*) can be emulated with Workflow Projects

… but you can build it yourself with extensions and T4!

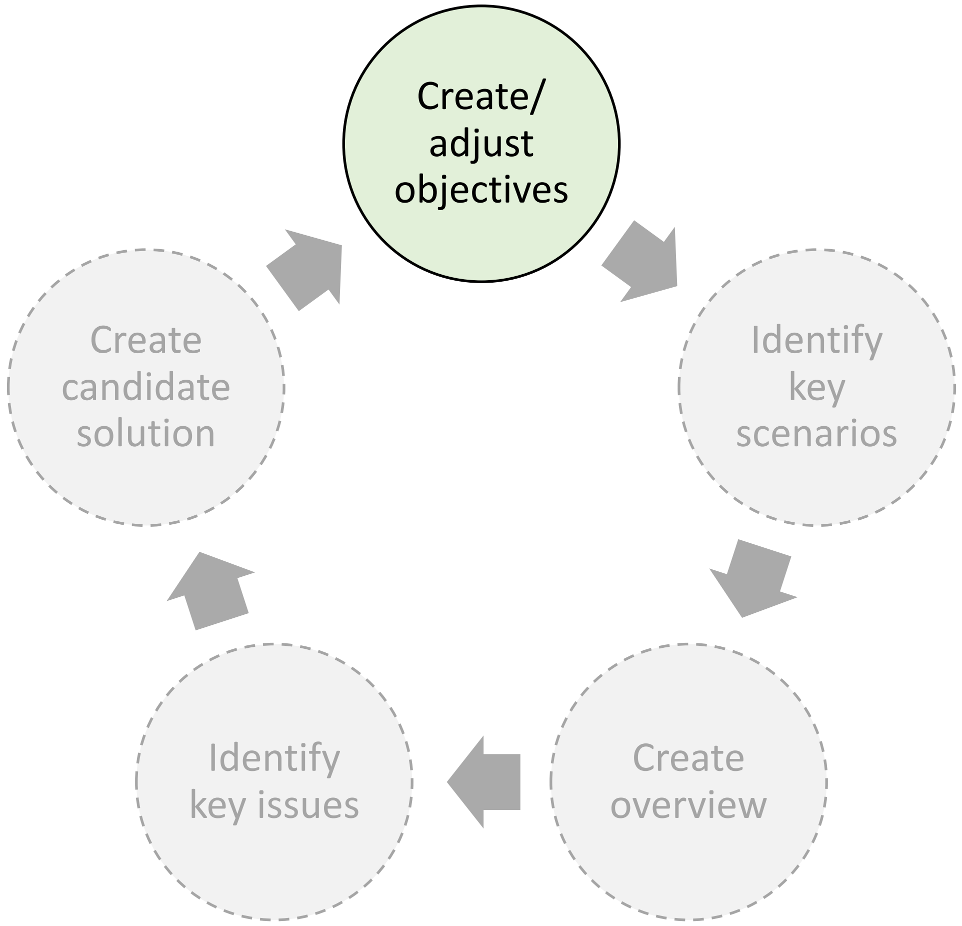
## The Process For Designing Architectures

## The Process For Designing Architectures



## Create Or Adjust Objectives

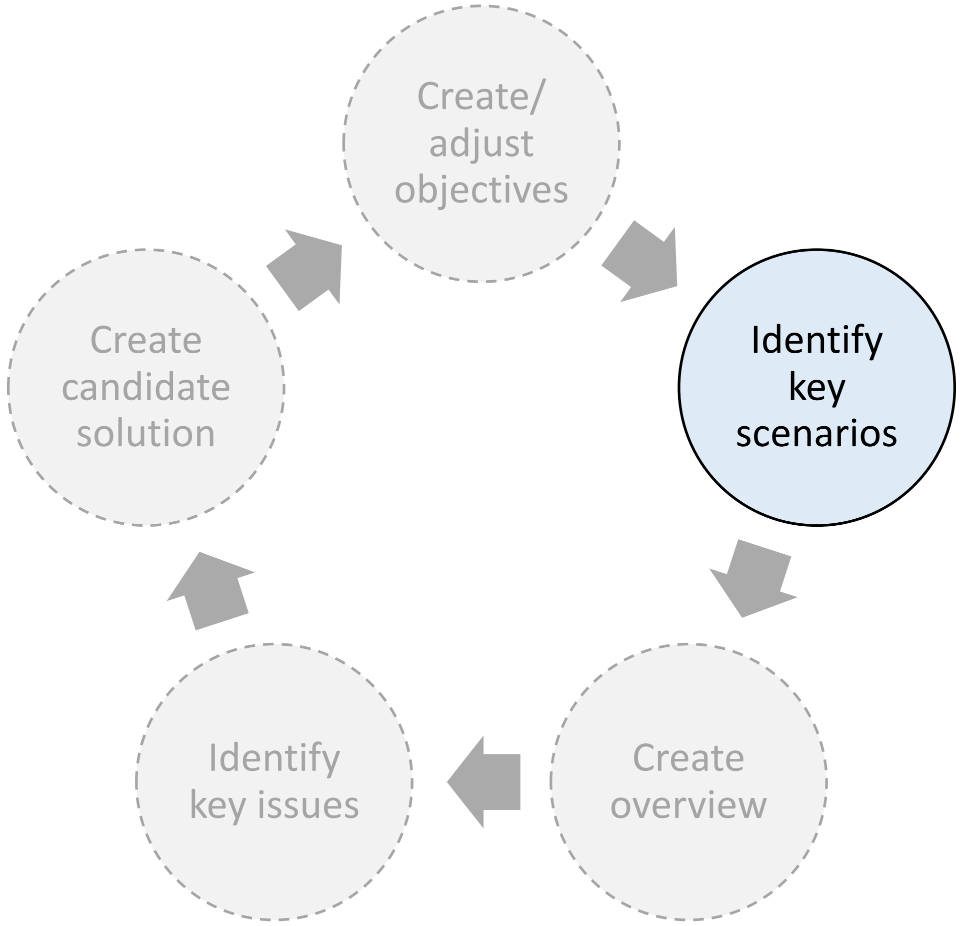
- Identify scope of architecture
- Estimate time to spend
- Identify audience
- Identify technical-, usage- and deployment constraints



## Identify Key Scenarios

- Key scenarios:
  - Significant unknown/risk
  - Significant use case
  - Intersection of quality/function
  - Tradeoff between attributes
- Significant use cases:
  - Business-critical
  - High impact

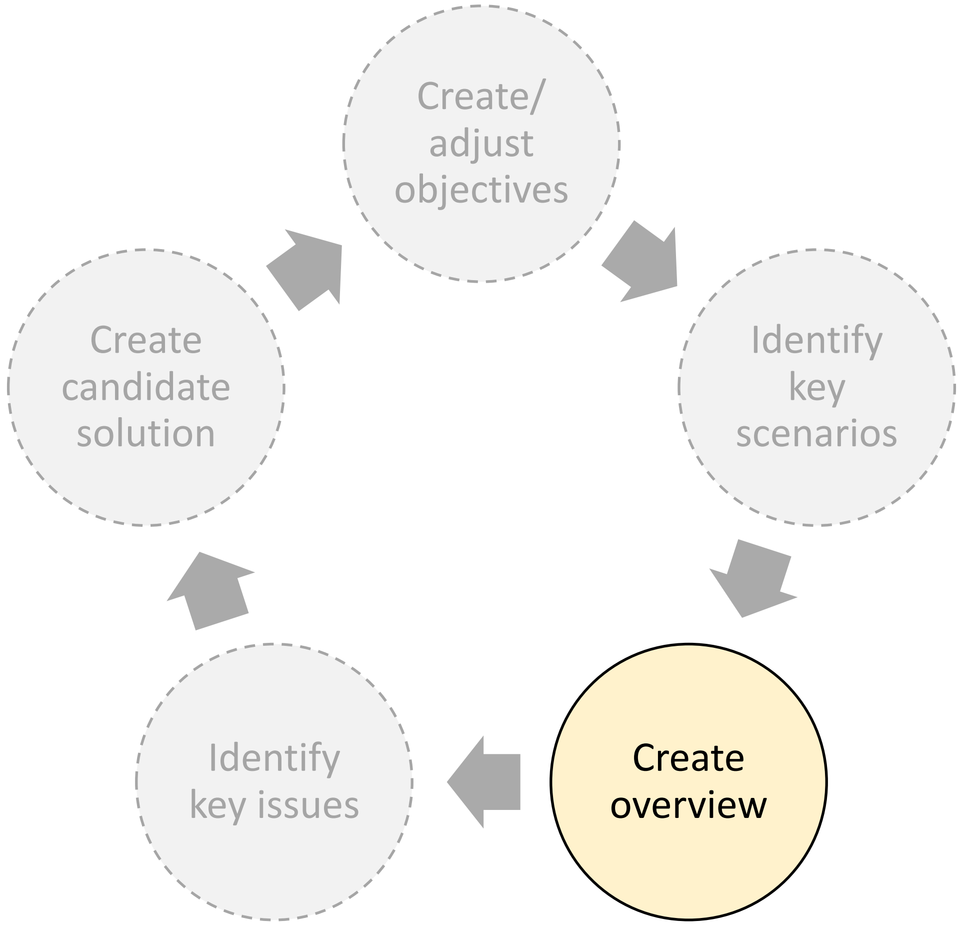
Create Use Case Diagrams



## Create Application Overview

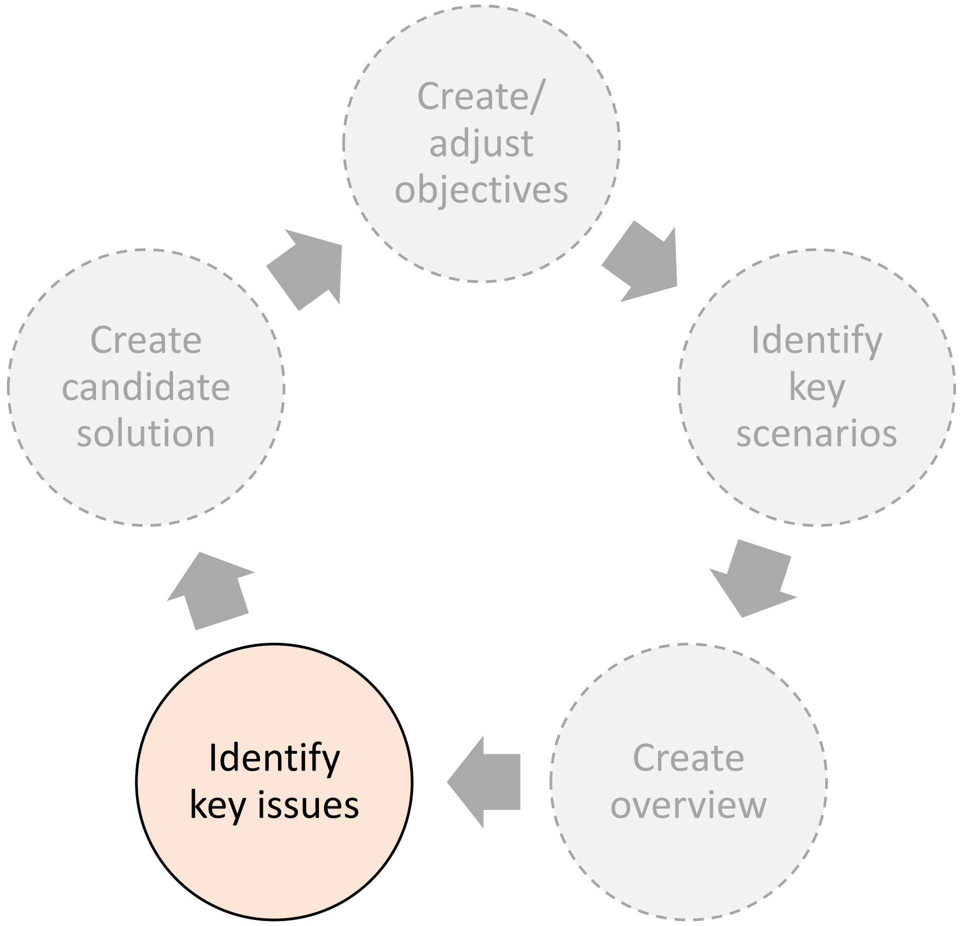
- Determine application type
- Identify deployment constraints
- Identify architecture pattern
- Determine technologies

Create Layer Diagram



# Identify Key Issues

- Quality attributes:
  - System/Run-time/Design/User
- System wide concerns:
  - Authentication & authorization
  - Caching
  - Communication
  - Configuration management
  - Logging & exception management
  - Validation

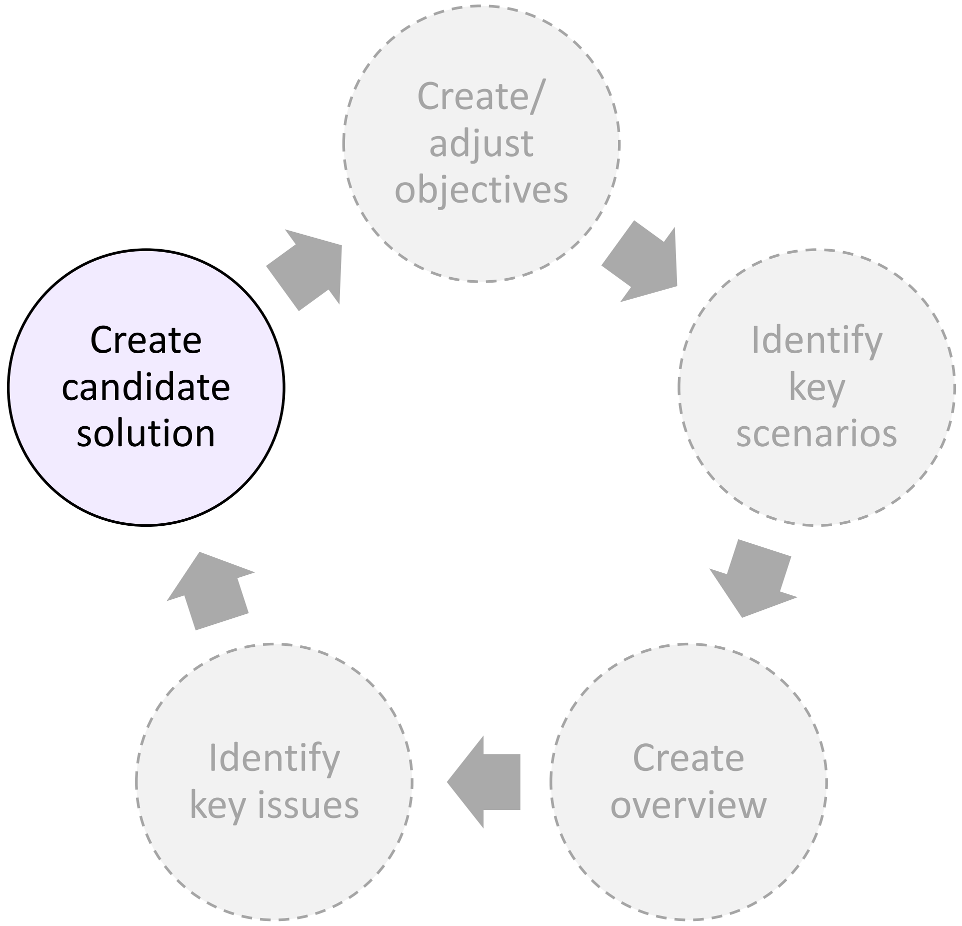


## Create Candidate Solution

- Create a Baseline architecture
- Create a Candidate architecture
- Develop Architectural Spikes

Create Activity, Sequence, State, and Component Diagrams

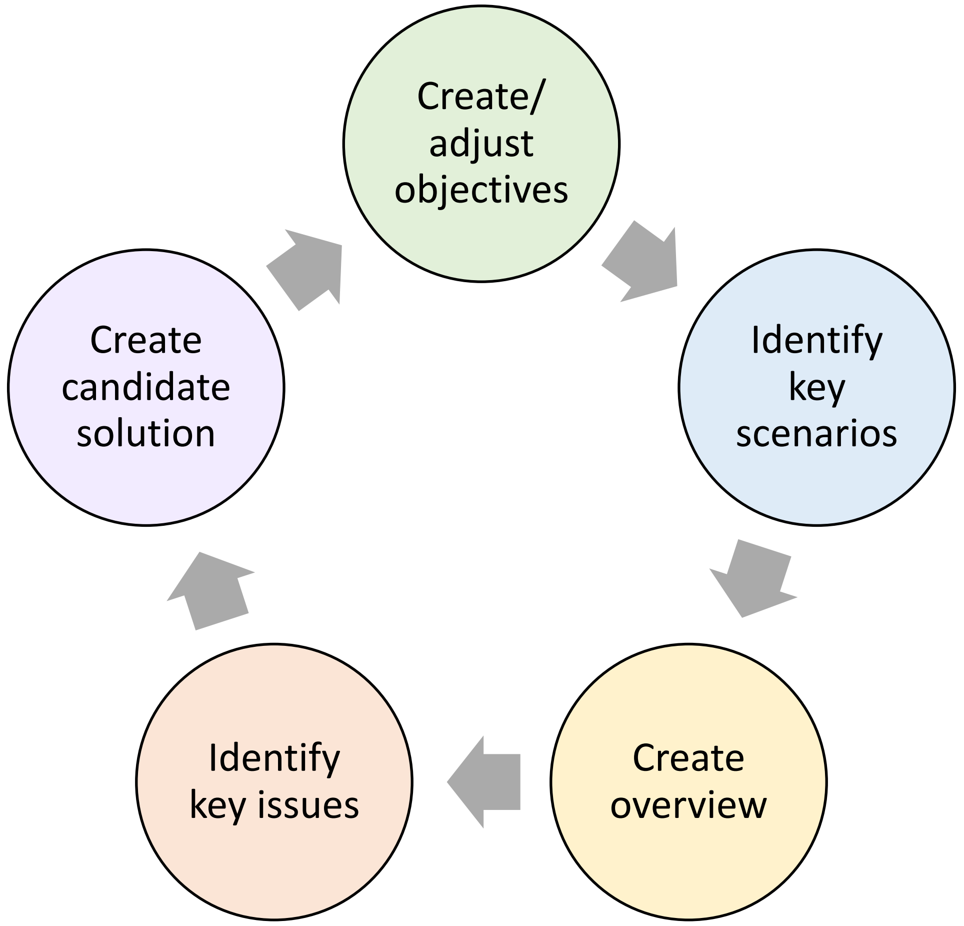
Create Class Diagram if needed



## During Each Cycle…

- Do not introduce new risk
- Mitigate more risk than baseline
- Meet additional requirements
- Enable more key scenarios
- Address more key issues

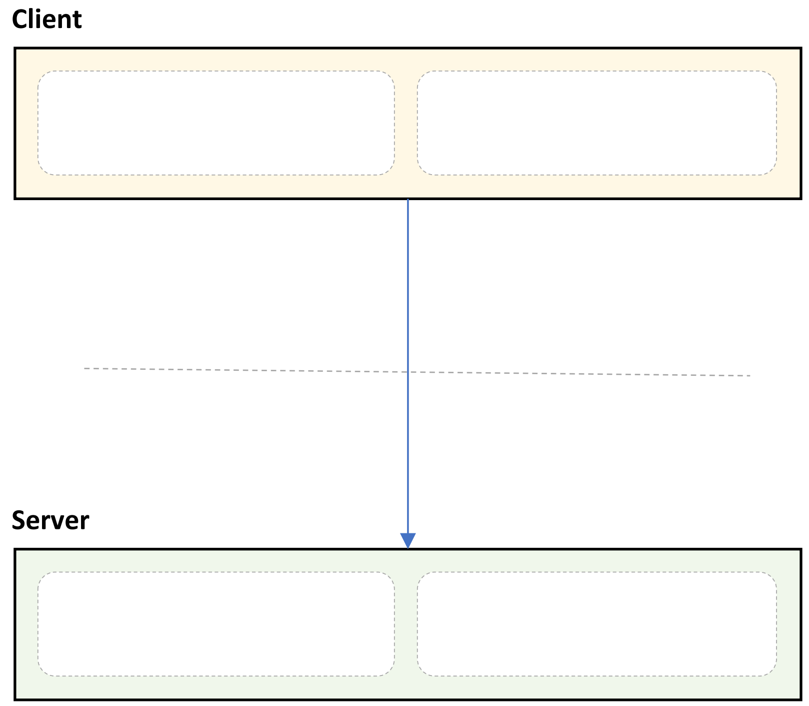
Start communicating architecture when you exceed 50% coverage



## Layered Architecture Patterns

# Client/Server Pattern

- Distinct client and server
- Separated by network
- Communication protocol
- Many clients, one server



## Client/Server Pattern

### Pros

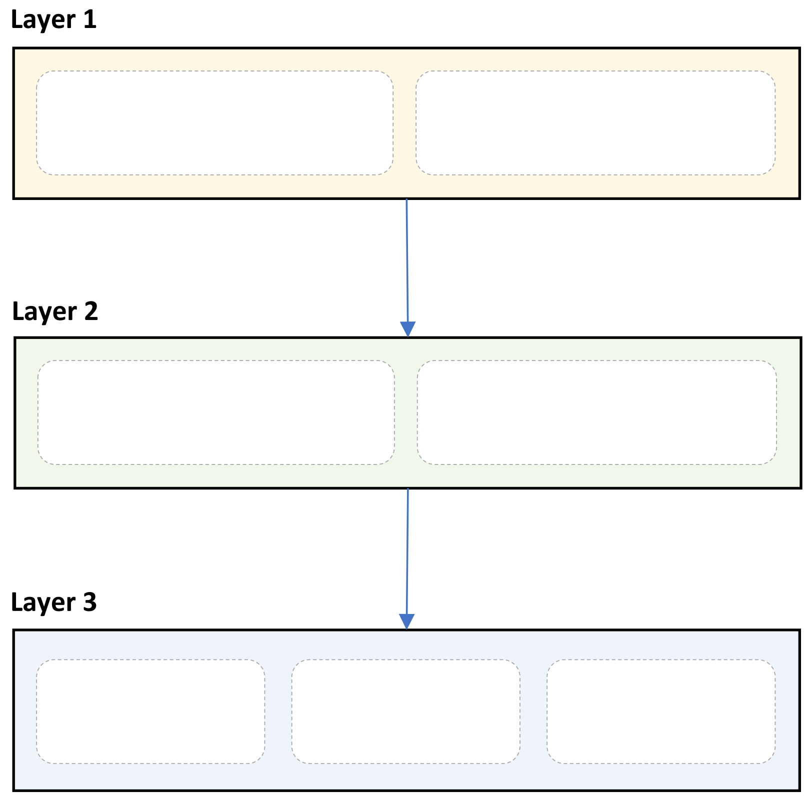
- Very secure
- Simple communication
- Centralized control
- Easy to manage

### Cons

- Requires network
- Difficult to scale out
- Single point of failure
- Hard to debug

## Layered Pattern

- Strict areas of concern
- Layers may only communicate with peers above/below

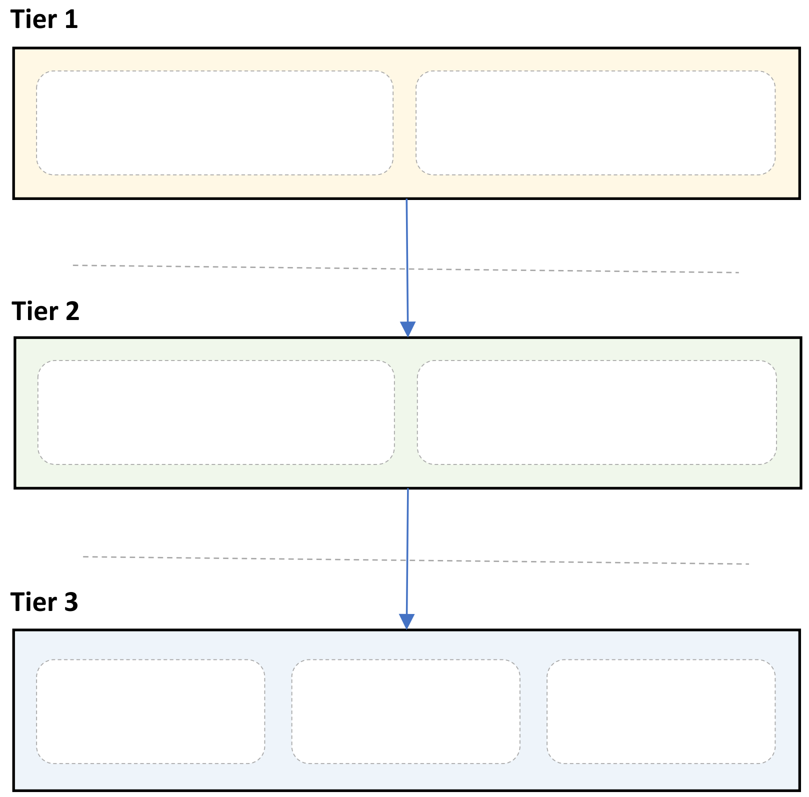


## Layered Pattern

| Pros | Cons |
|---|---|
| • High abstraction<br>• High isolation<br>• Structured communication<br>• Easy to scale out | • Deep call chains<br>• Can hide complexity<br>• May harm performance<br>• Lowest layer must cover all use cases |

# N-Tier Pattern

- Layers deployed to servers
- Usually 3 tiers
- Communication between layers uses network



## N-Tier Pattern

### Pros

- High abstraction
- High isolation
- Structured communication
- Easy to scale out

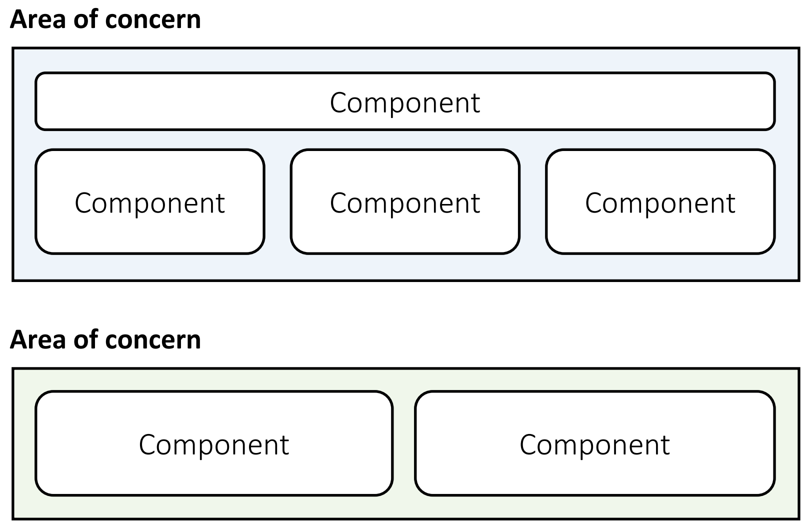
### Cons

- Network = point of failure
- Network may be slow
- Coarse interfaces
- Hard to debug

## Structural Architecture Patterns

# Component-based Pattern

- Components are modular building blocks of software
- Grouped into areas of concern
- Clearly described interfaces
- Containers provide additional services

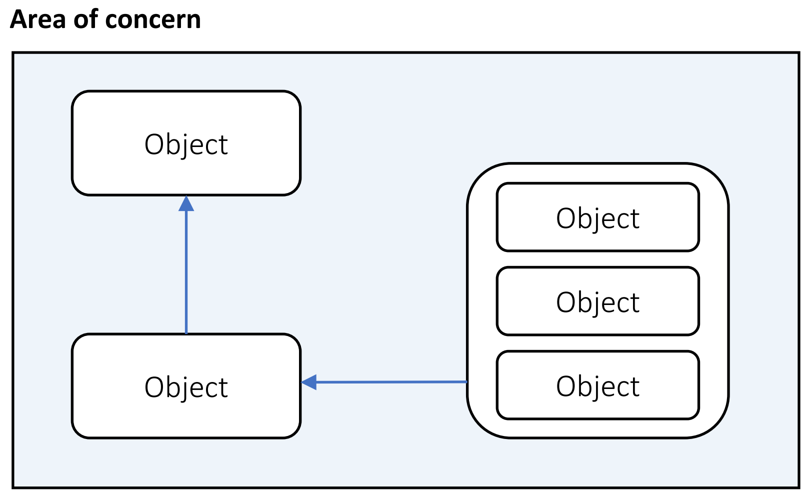


## Component-based Pattern

| Pros | Cons |
|---|---|
| • Easy deployment<br>• Allows 3rd parties<br>• Promotes modularity<br>• Few unanticipated interactions | • Coarse building blocks<br>• Can be expensive<br>• Initialization may be slow<br>• Harder to develop & maintain |

## Object-Oriented Pattern

- Uses Classes
- Grouped into areas of concern
- Describes public members
- Uses inheritance, composition, aggregation, and associations



## Object-Oriented Pattern

### Pros

- Easy to understand
- Promotes reuse
- Easy to test & debug
- Highly cohesive

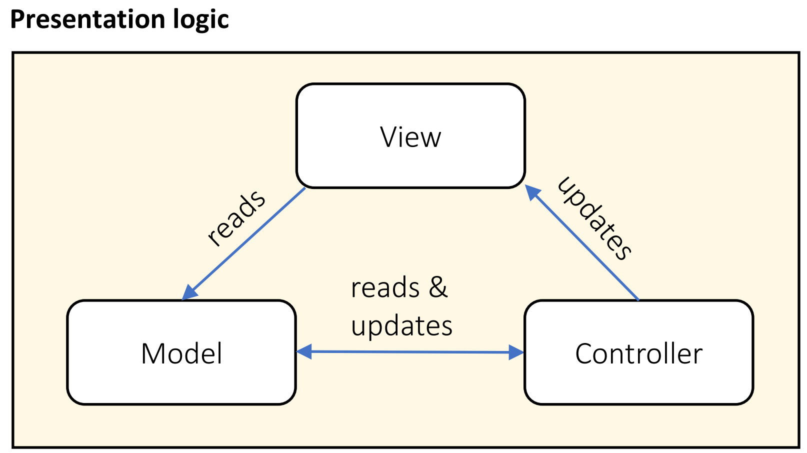
### Cons

- Inheritance hard to get right
- Many unanticipated communications
- Too detailed

## Presentation Architecture Patterns

# MVC Pattern

- Designed for presentation layers
- View handles output
- Model handles data
- Controller handles interaction



## MVC Pattern

### Pros

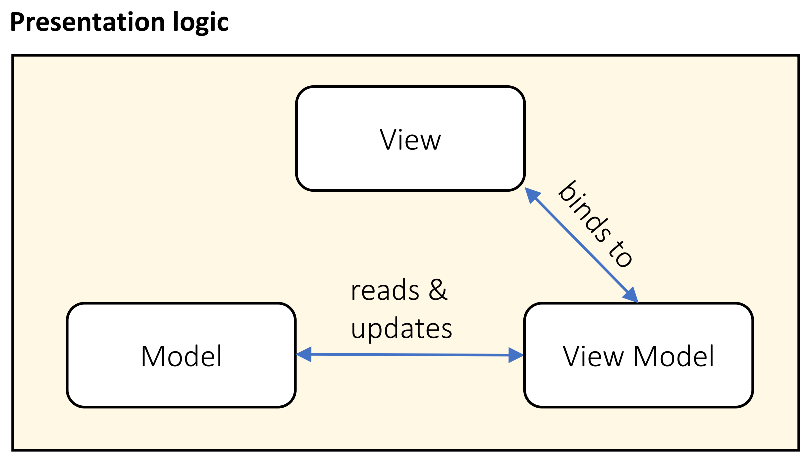
- Strict separation of concerns
- Scales well

### Cons

- High overhead
- Scattered code
- Hard to data-bind

## MVVM Pattern

- Derived from MVC pattern
- View handles output
- Model handles data
- View Model is binding source



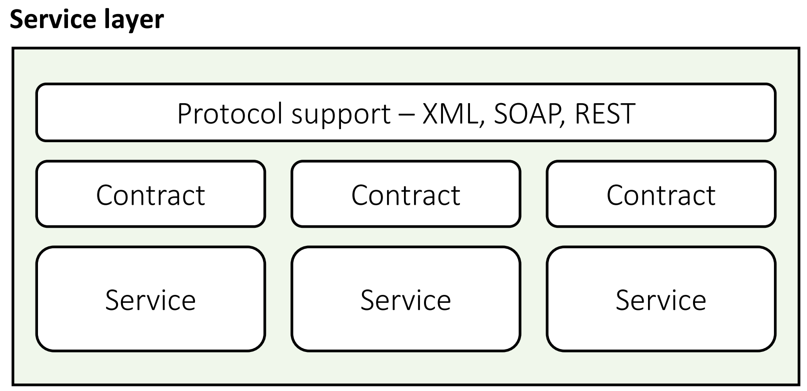
## MVVM Pattern

| Pros | Cons |
|---|---|
| • Strict separation of concerns<br>• Scales well<br>• Easy to data-bind | • High overhead<br>• Scattered code<br>• Controller not separated |

## Service Architecture Patterns

# Service-Oriented Pattern

- Discrete business services
- Use network to communicate
- HTTP, XML, SOAP, Binary, …



## Service-Oriented Pattern

### Pros

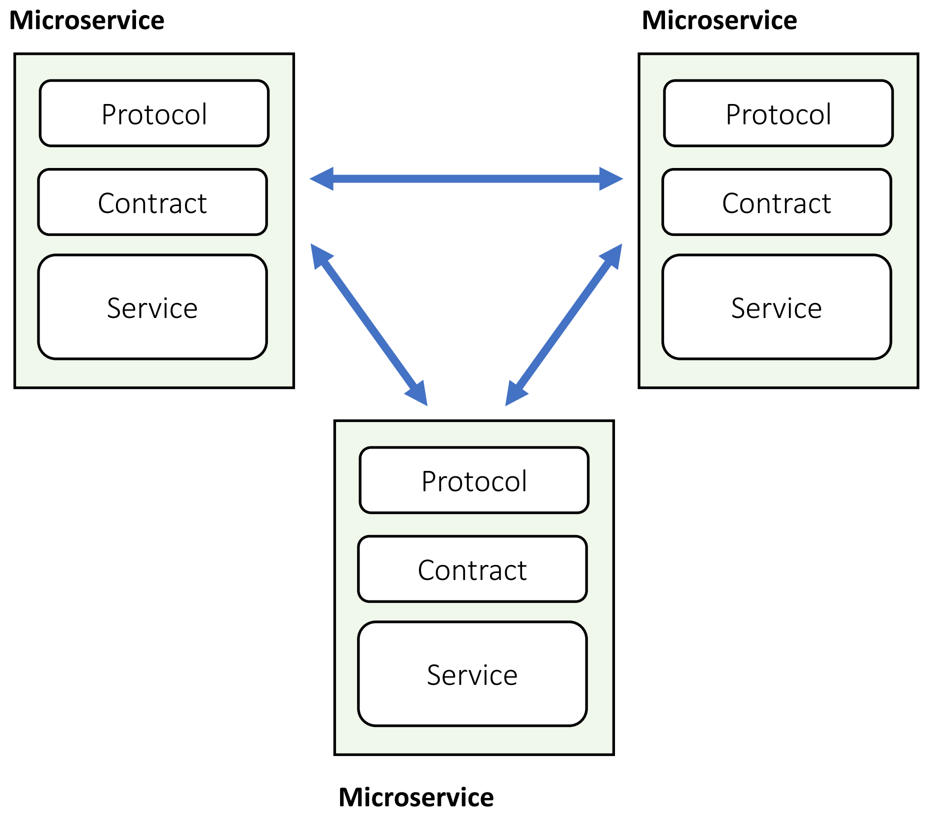
- Business domain alignment
- High abstraction
- Discoverable, resilient
- Allows 3rd parties
- Cross-platform

### Cons

- Clients must handle slow, offline network
- Coarse interfaces
- May harm performance
- Security issues

## Microservice Pattern

- Services calling services
- Can use fast private network
- Deployed on multiple servers

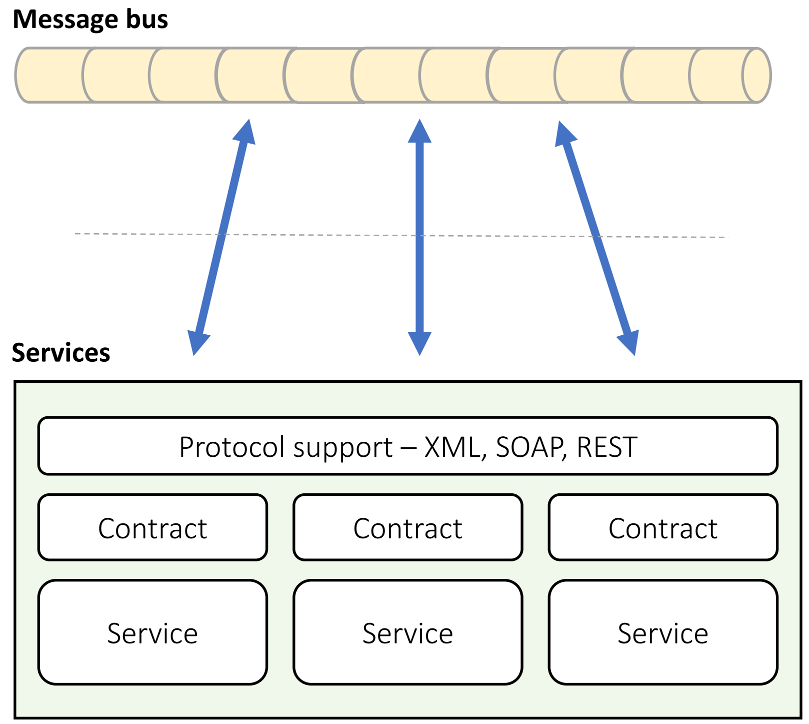


## Microservice Pattern

| Pros | Cons |
|---|---|
| • Modular<br>• Reduced abstraction<br>• Discoverable, resilient<br>• Can use fast network<br>• Less coarse interfaces | • Must cope with slow, offline network<br>• More unanticipated communications<br>• Hard to do transactions<br>• Hard to test, debug, deploy |

# Message Bus Pattern

- Services connected to shared data bus
- Uses messages for communication
- Supports discovery, failover



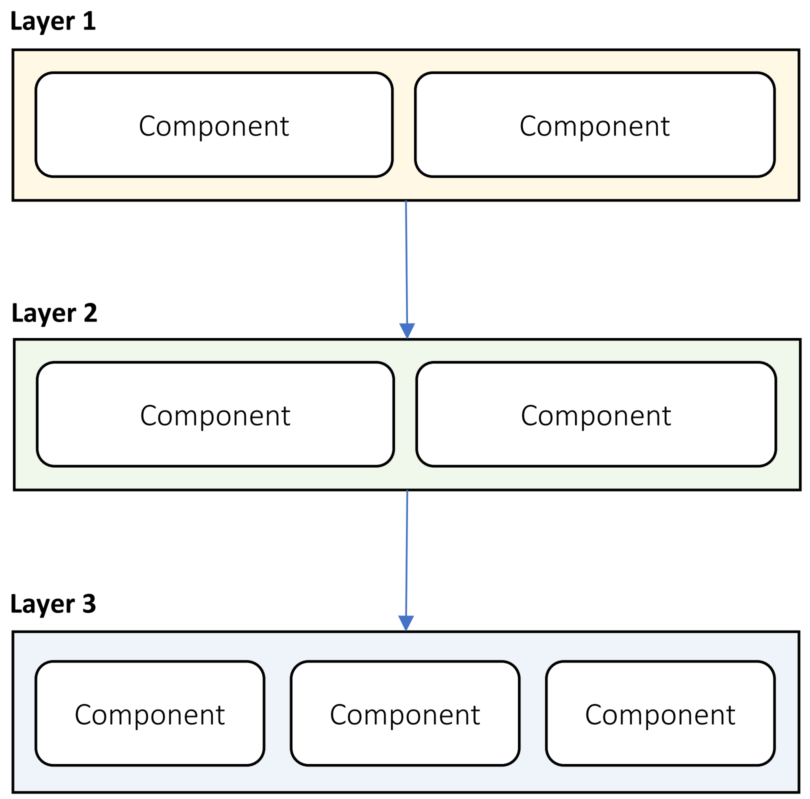
## Message Bus Pattern

| Pros | Cons |
|---|---|
| • Easy to extend<br>• Simple communication<br>• Very flexible<br>• Easy to scale<br>• Easy discovery, failover | • Bus = single point of failure<br>• Coarse communications<br>• Can be slow<br>• Hard to test, debug |

## Hybrid Architecture Patterns

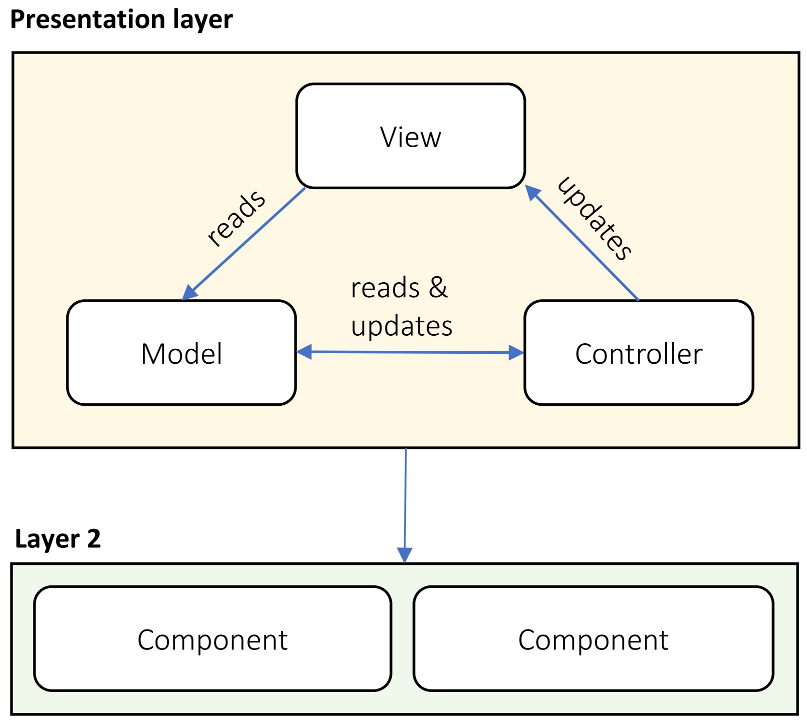
## Components In Layers

- Layers containing components
- Benefits of components…
- Plus: structured communication, isolation, easy scaling & testing



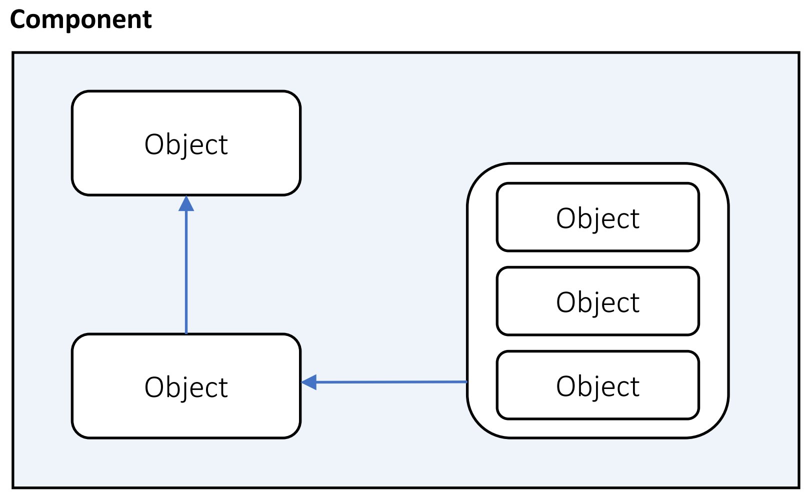
# MVC In Presentation Layer

- Presentation layer with MVC
- Benefits of layers…
- Plus: separation of concerns, presentation code scalability



## Objects In Components

- Components contain objects
- Benefits of components…
- Plus: cohesion & extensibility

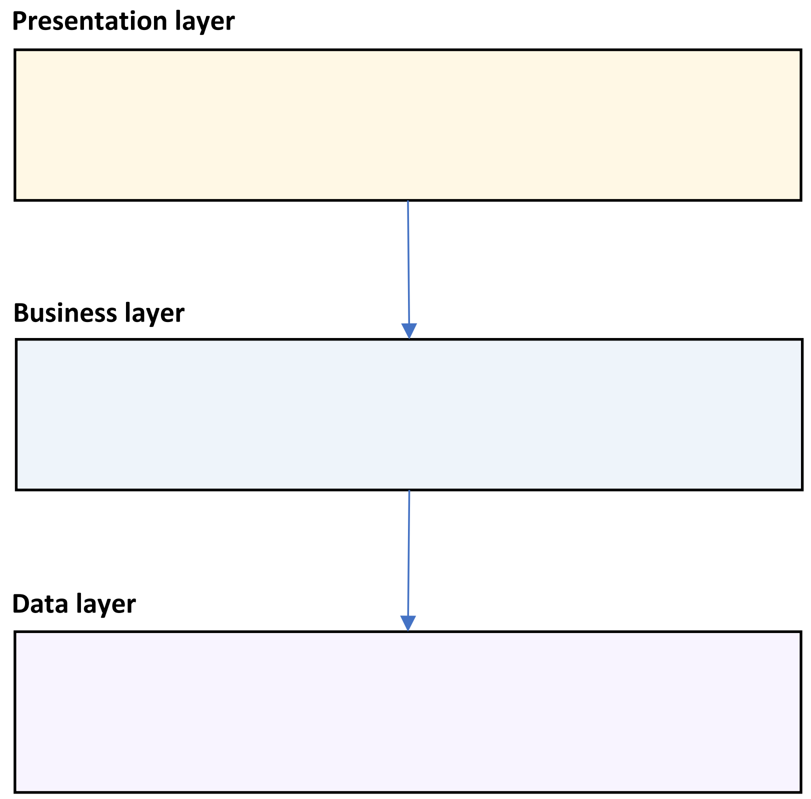


## Choosing The Right Patterns

## The Layered Pattern

- Good starting point
- Create 3 layers:
  - Presentation
  - Business
  - Data

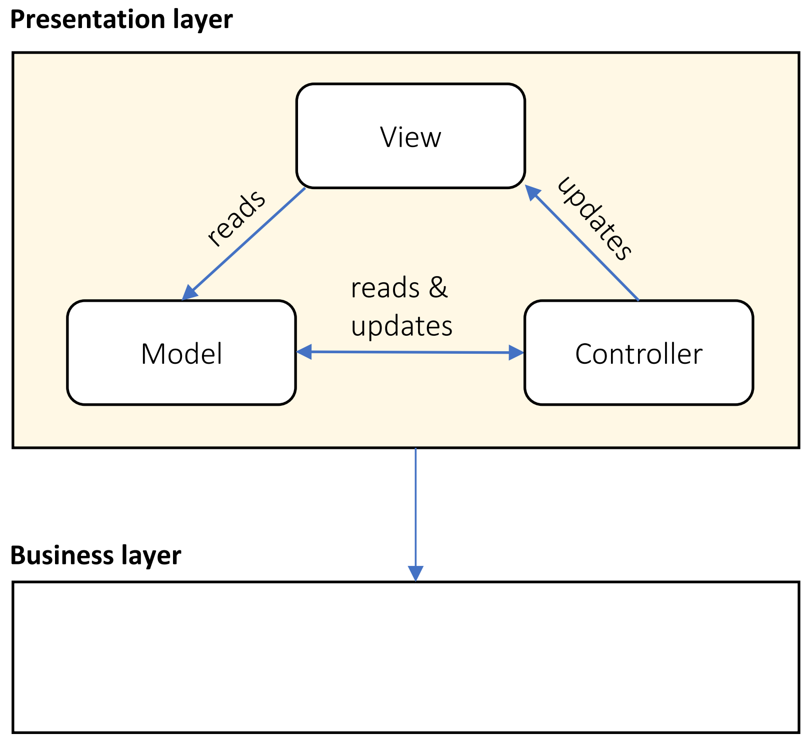
Add a Service layer if you plan to expose an API.



# The Presentation Layer

- Use Components for…
  - Containers
  - Reuse
  - 3rd parties
  - Declarative rendering

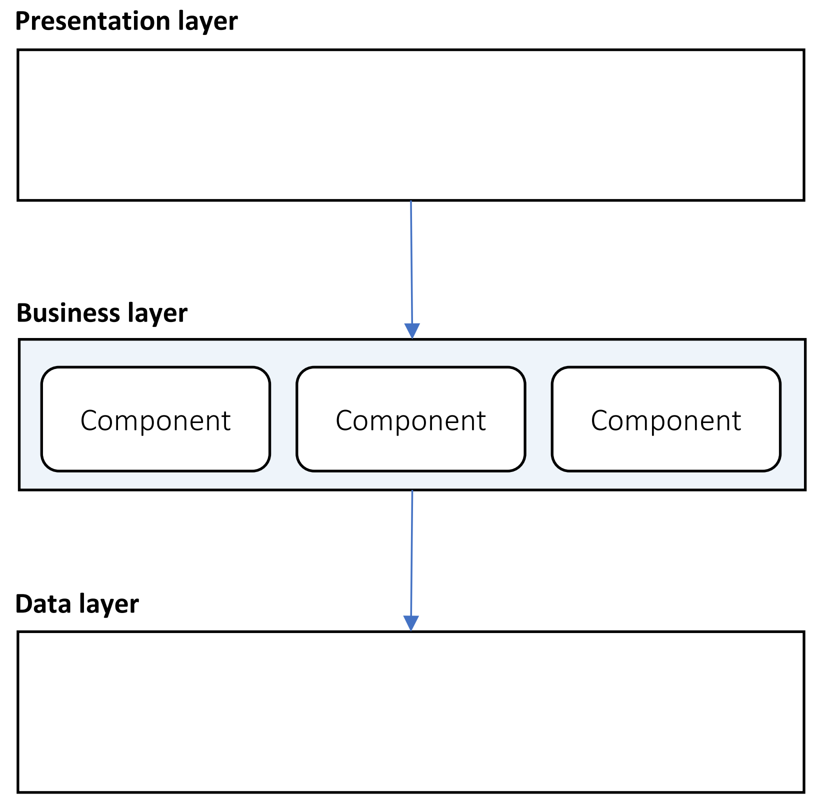
- Choose MVC for large UIs
  - … or MVVM if your technology supports databinding



# The Business Layer

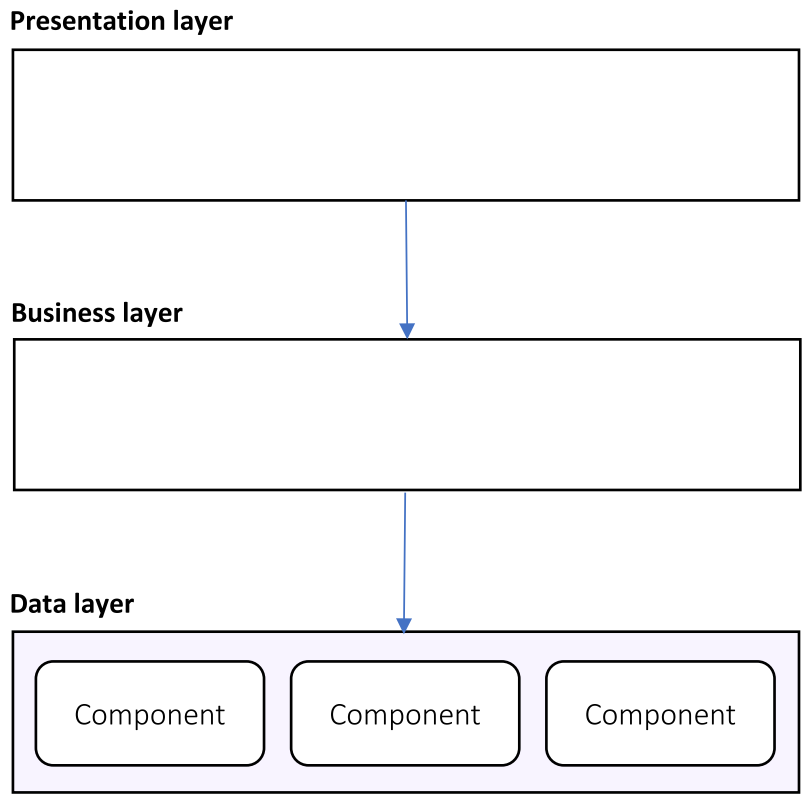
- Use Components for…
  - Modular functionality
  - Plugin support
  - Business Entity abstraction
  - Declarative configuration:
    - Visual workflows
    - Business rules

If you don't need the above, use an object-oriented architecture



# The Data Layer

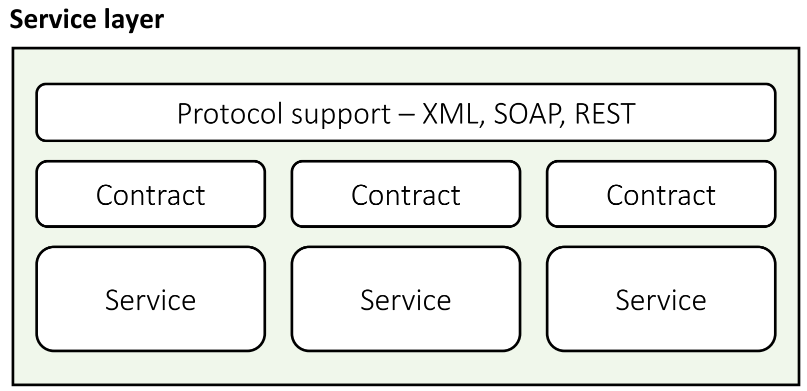
- Use Components for…
  - Handling diverse data sources
  - Increased abstraction
  - Declarative configuration



## The Service Layer

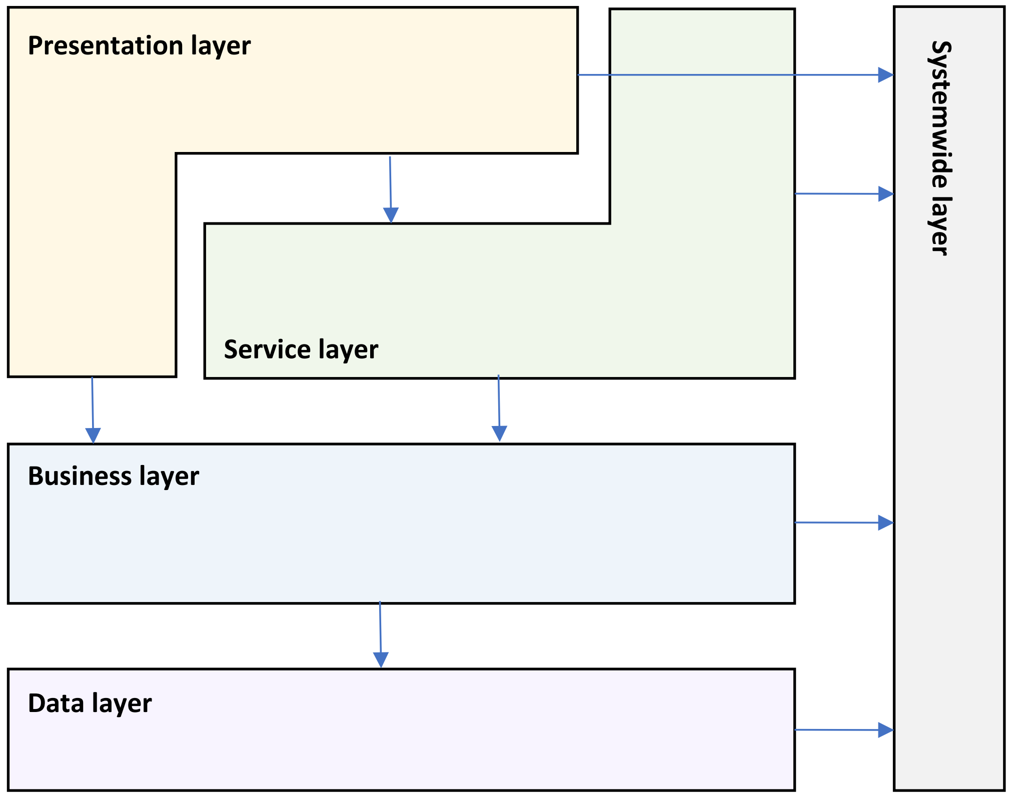
- Use Components for…
  - Contract management
  - Containers
  - Declarative configuration

- Choose Microservices if your system is composed mostly of interlocking APIs
- Choose Message Bus if all services alter state on a common message



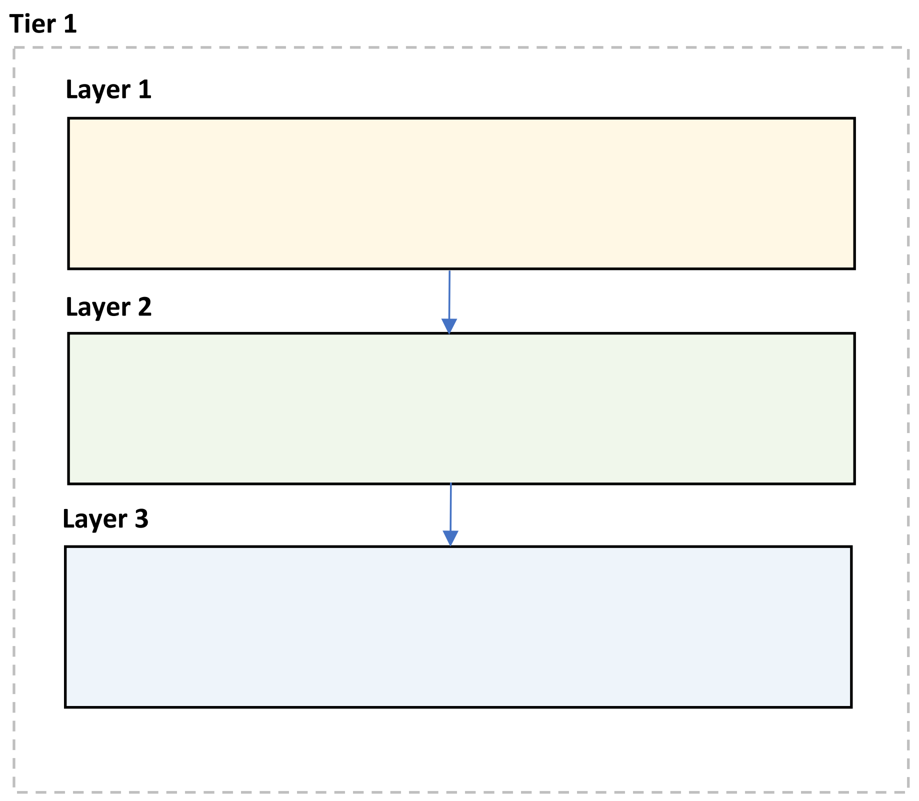
## Designing Layered Architectures

# Logical Layered Design

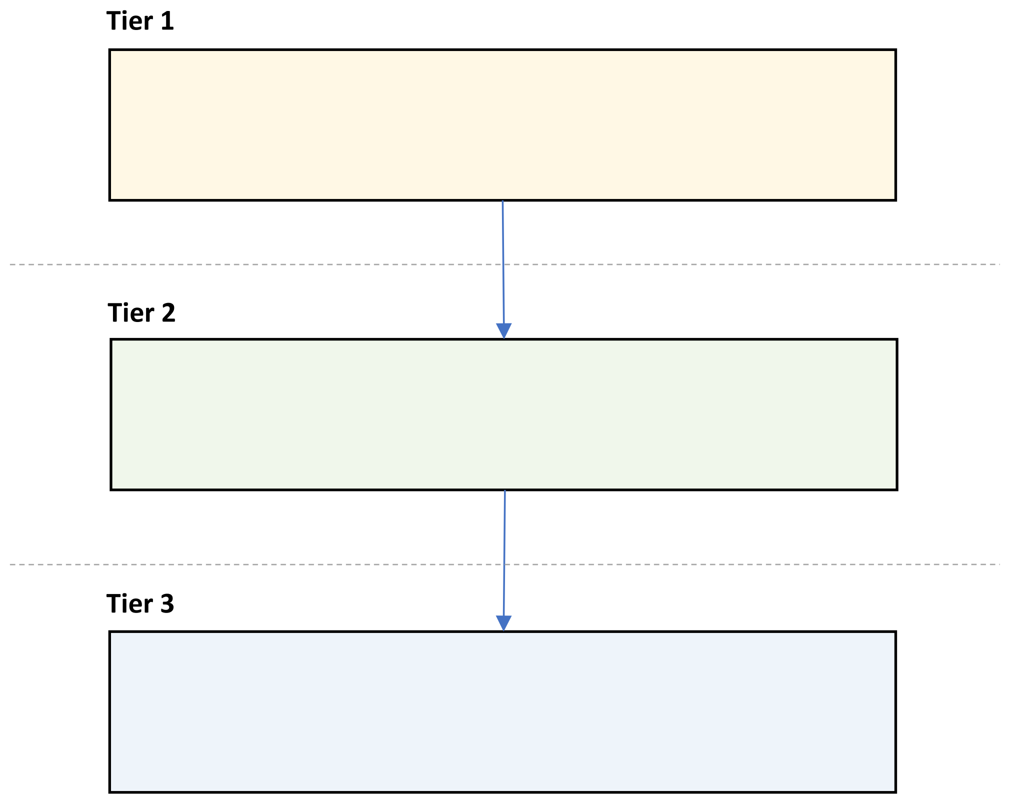


# Choose Layering Strategy

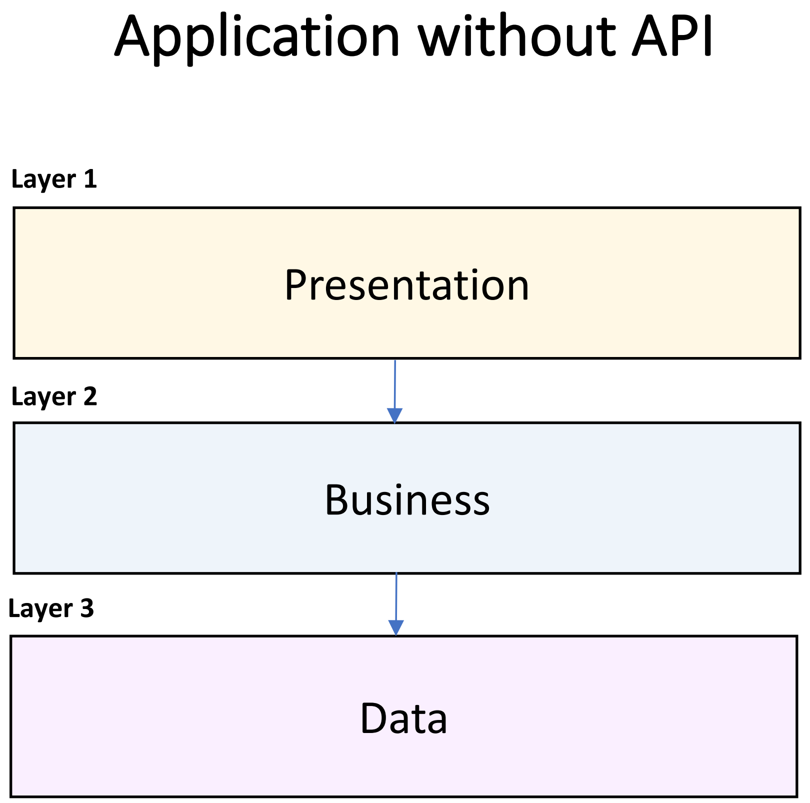
## Logical separation

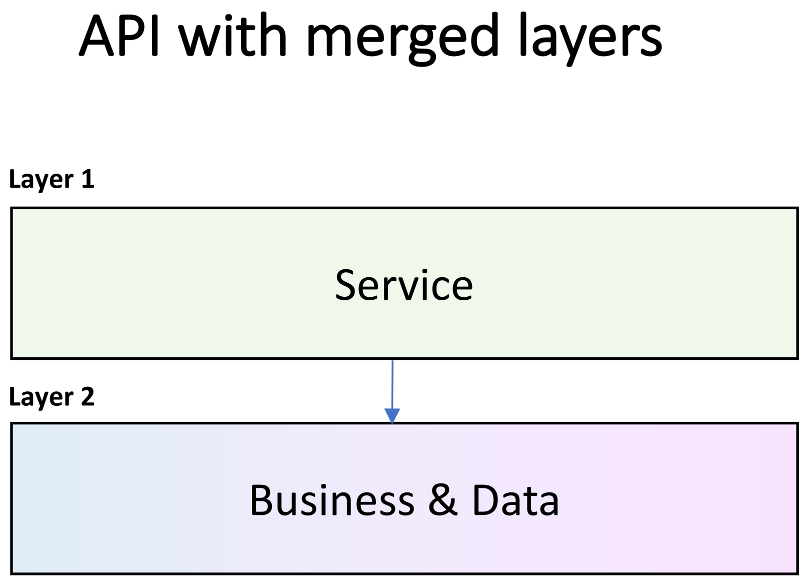


## Physical separation

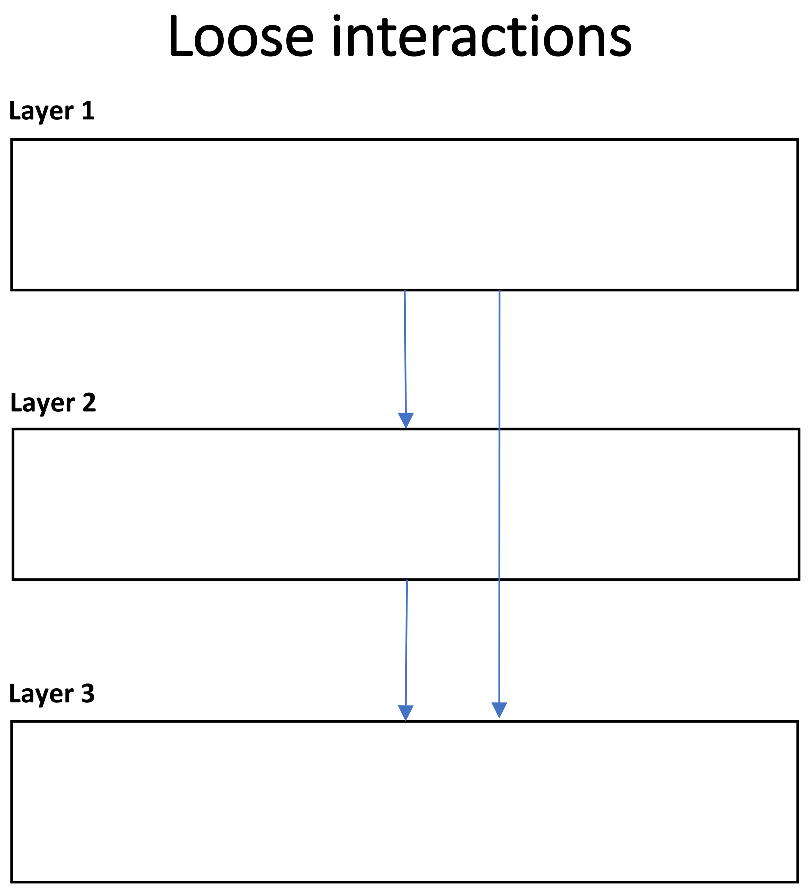


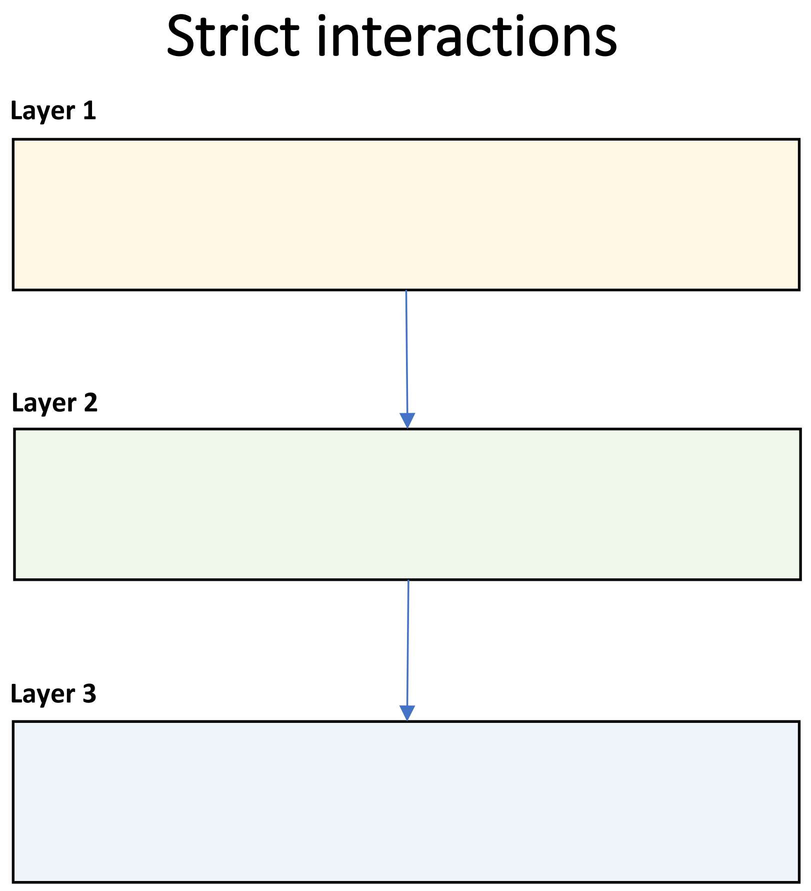
# Remove & Merge Layers



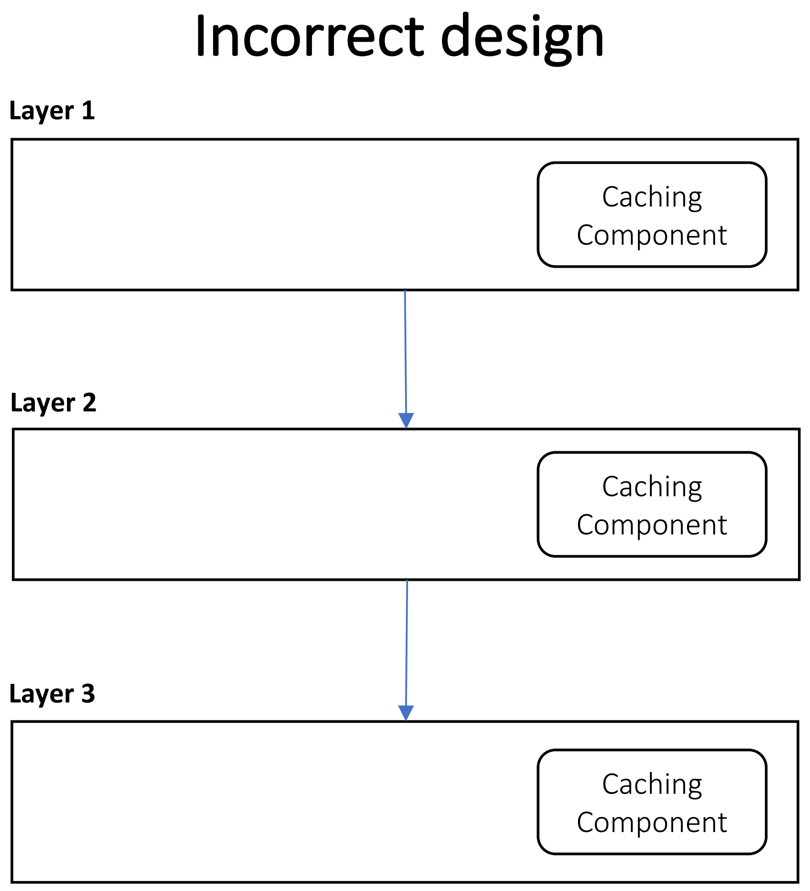


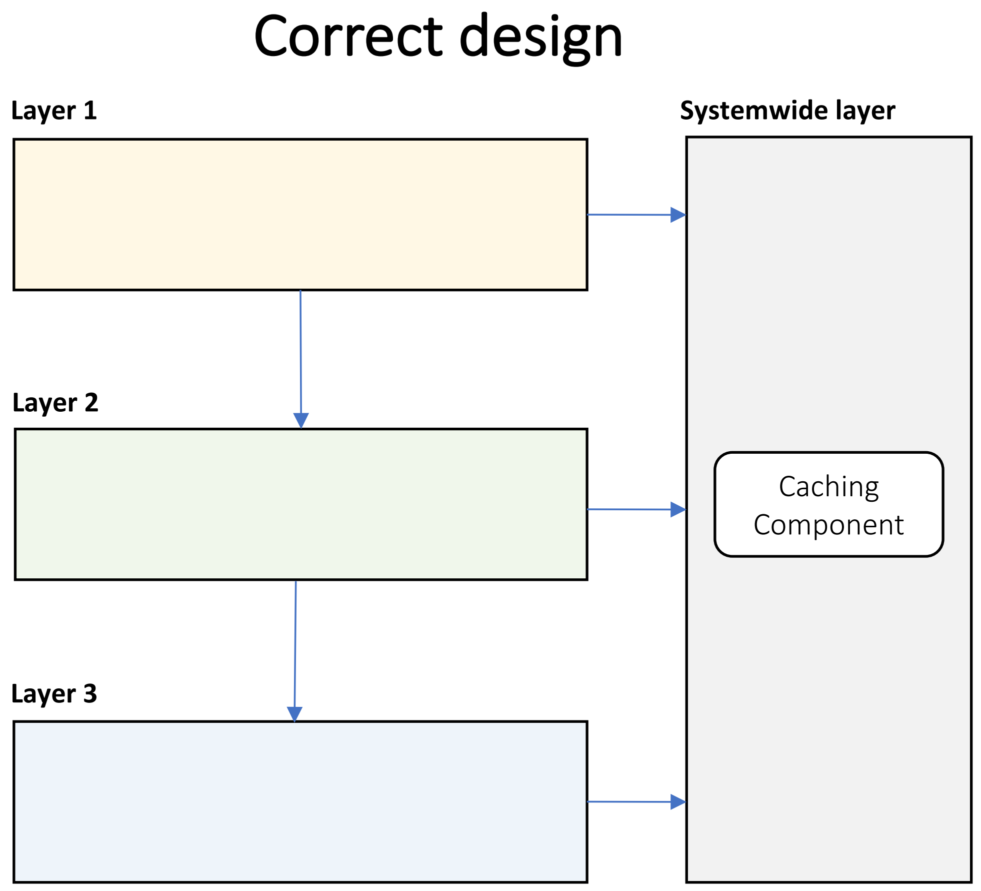
# Determine Layer Interactions





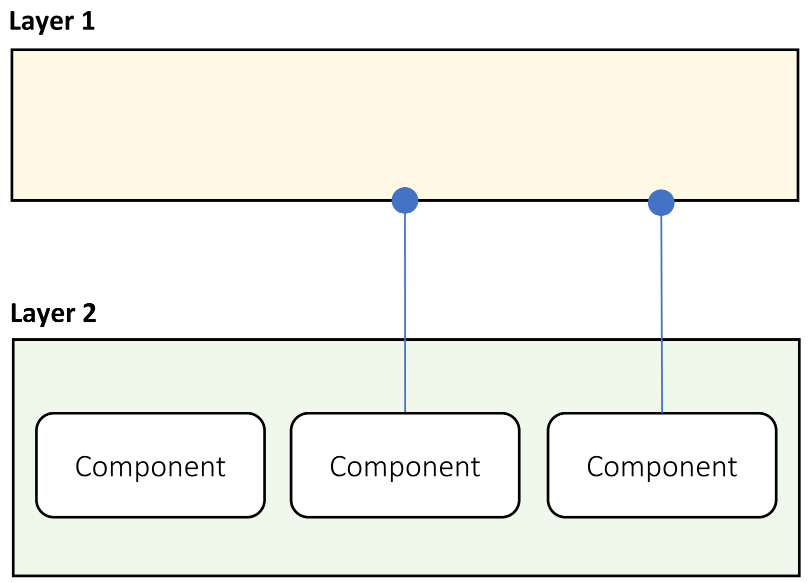
# Identify Systemwide Concerns



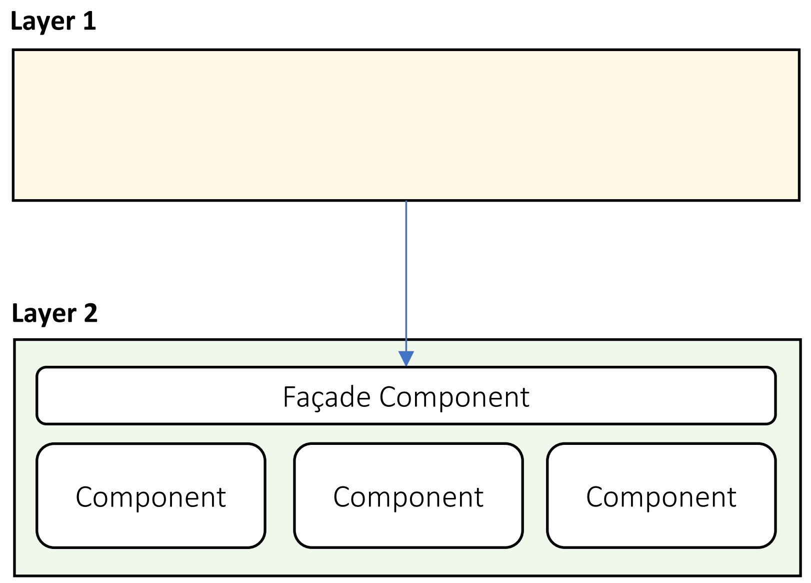


# Define Layer Interfaces

## Public interfaces



## Façade component



Alternatives: Singletons, Command Pattern, Dependency Injection, Message-based, …

## Designing Component Architectures

## General Guidelines

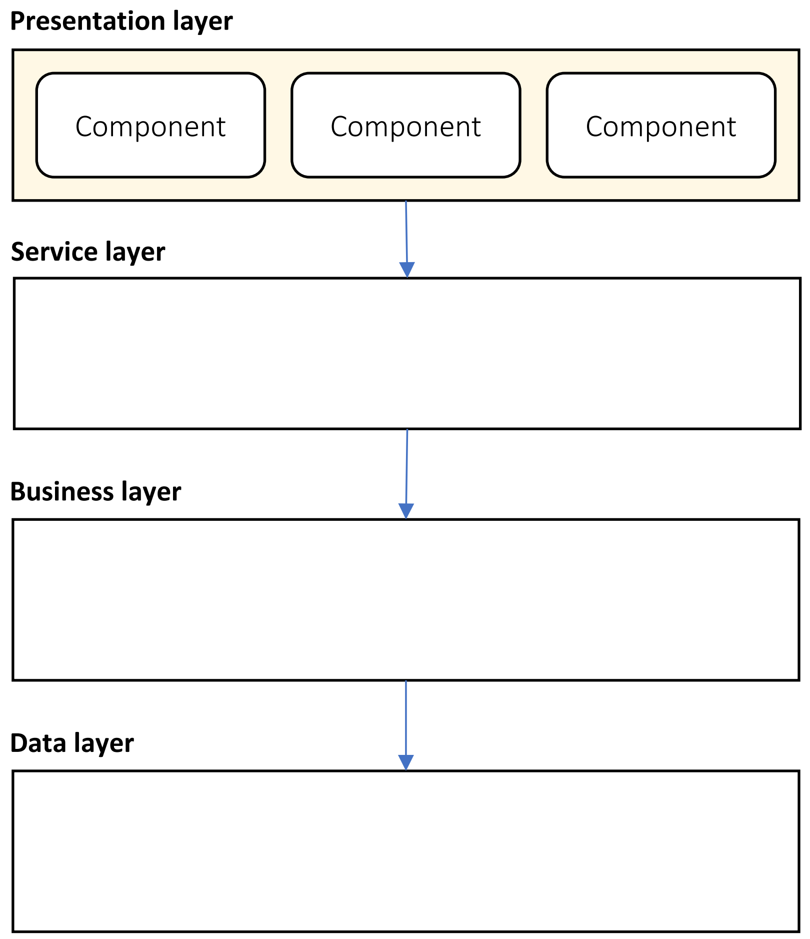
- Components are SOLID:
  - Single Responsibility
  - Open/Closed
  - Liskov Substitution
  - Interface Segregation
  - Dependency Inversion

- Highly cohesive
- No knowledge of internals of other components

# Presentation Layer Components

- UI components
- Presentation Logic
  - Views
  - Controllers
  - View Models

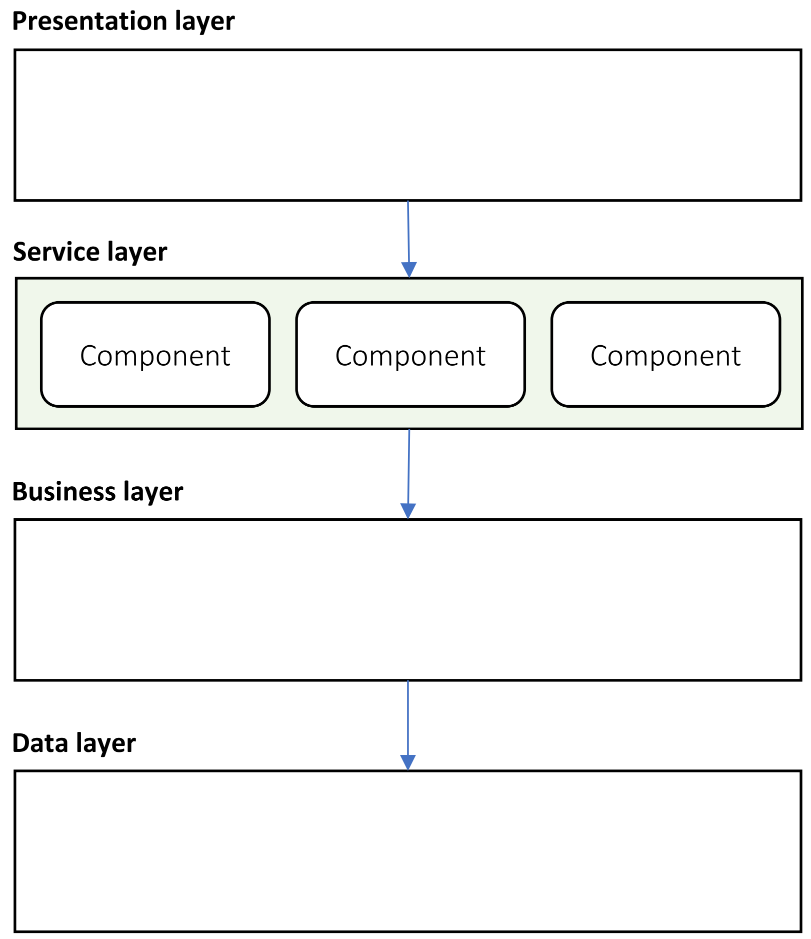
Optional: Presentation Entities



## Service Layer Components

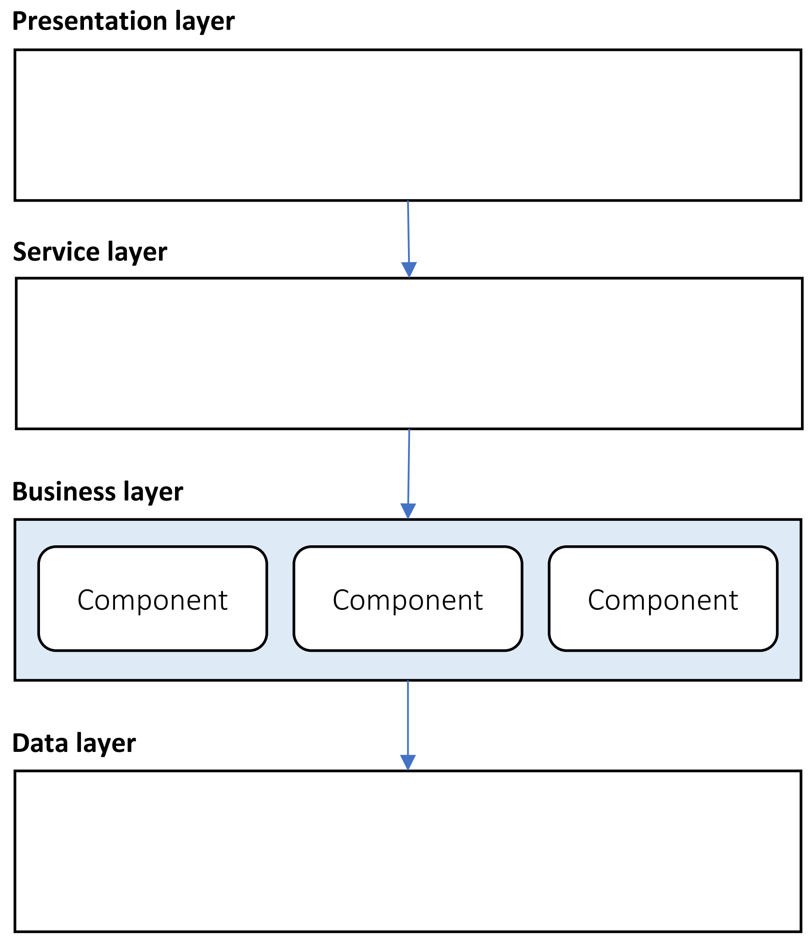
- Services
- Service Interfaces
- Message Types

Optional: Service Broker



# Business Layer Components

- Business Façade
- Business Logic
  - Components
  - Workflows
  - Business Rules
- Business Entities
  - Populated through ORM
  - or: Data Objects from data layer
- Business Events

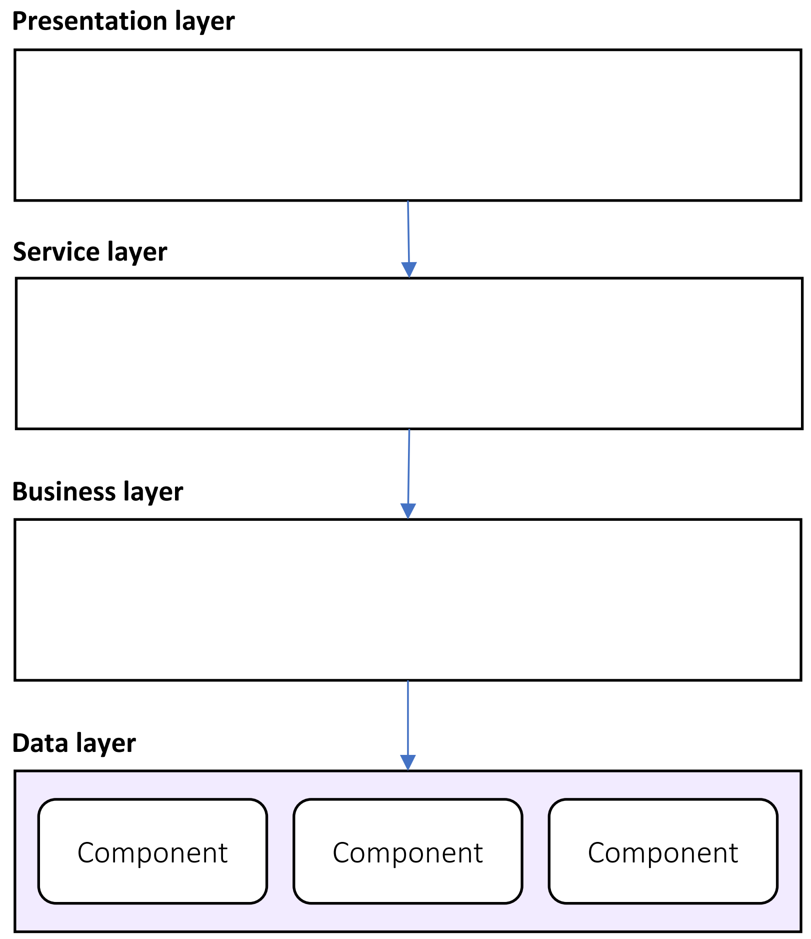


# Data Layer Components

- Data Façade
- Data Source Adapters
- Service Adapters

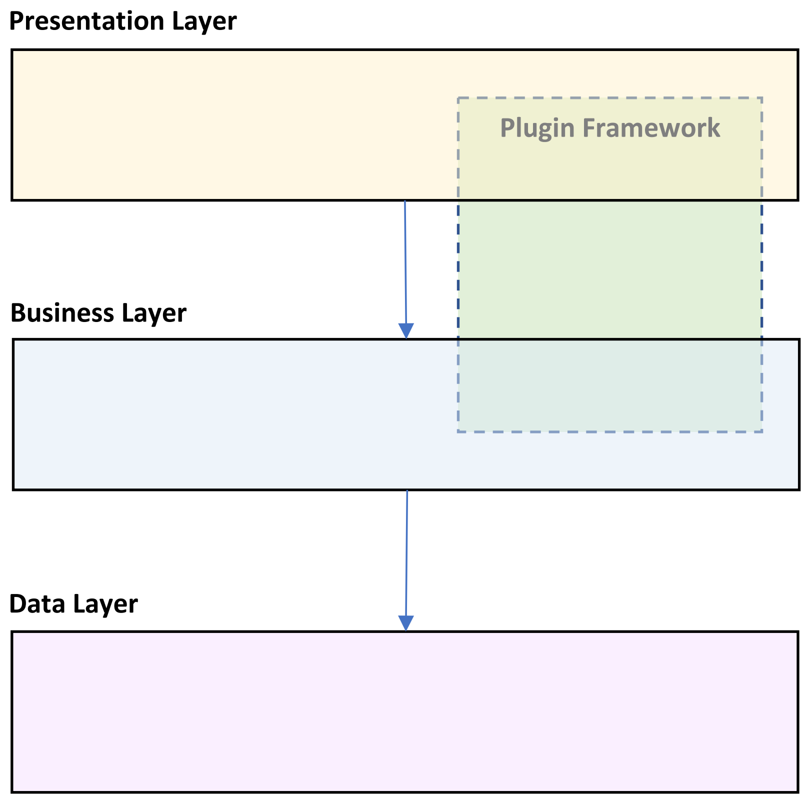
Optional:

- Data Objects
- Command & Query objects

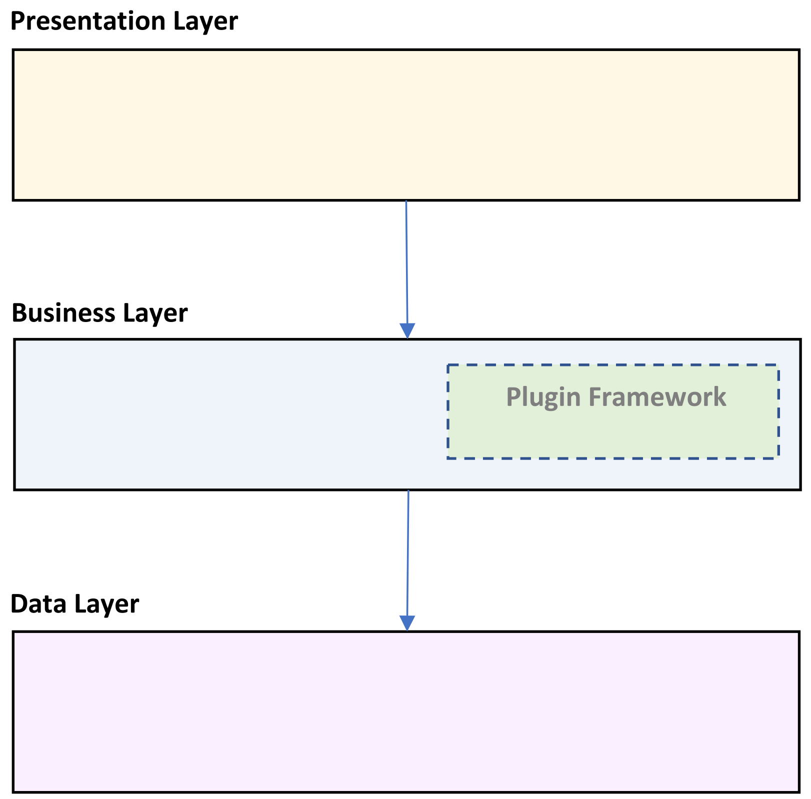


# Modular Architectures

## Plugin with UI



## Plugin without UI



## Designing Service-Oriented Architectures

## REST versus SOAP Services

### REST Service

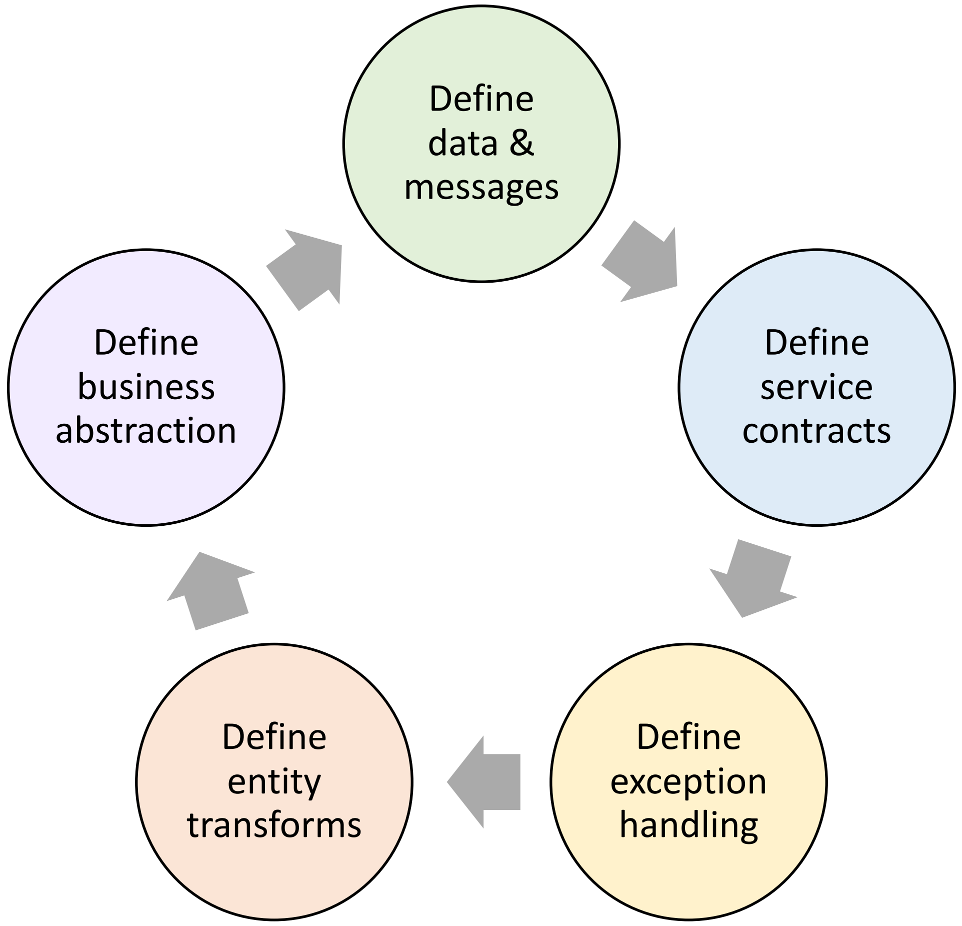
- CRUD operations
- Operates on Entities
- JSON/XML/HTML/Markdown/…
- Low overhead
- Request/Response
- No discovery & routing
- Incomplete standard

### SOAP Service

- Any operations
- Not limited to Entities
- XML
- High overhead
- Request/Response, Fire & Forget, Bi-Directional Calls
- Standard discovery & routing
- World standard

# The Service Design Process

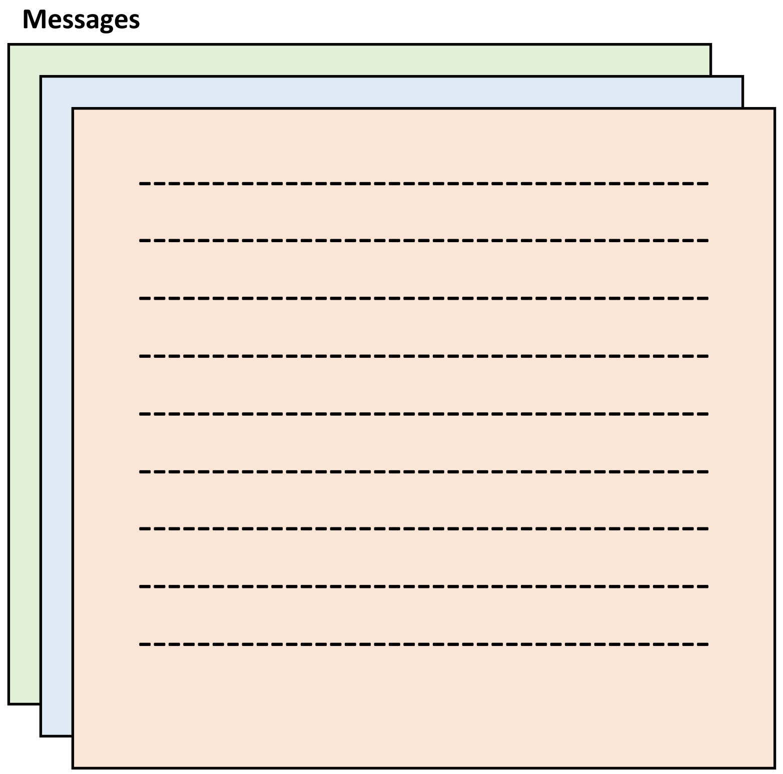
1. Define data & messages
2. Define service contracts
3. Plan exception handling
4. Define how business entities are transformed to messages
5. Define how business functions are abstracted to services



## Define Service Messages

- Request/Response, Fire & Forget, or Bi-Directional
- Command, Query, Document, Entity, Event, Message, …
- Avoid large messages
- Add expiration & diagnostic info

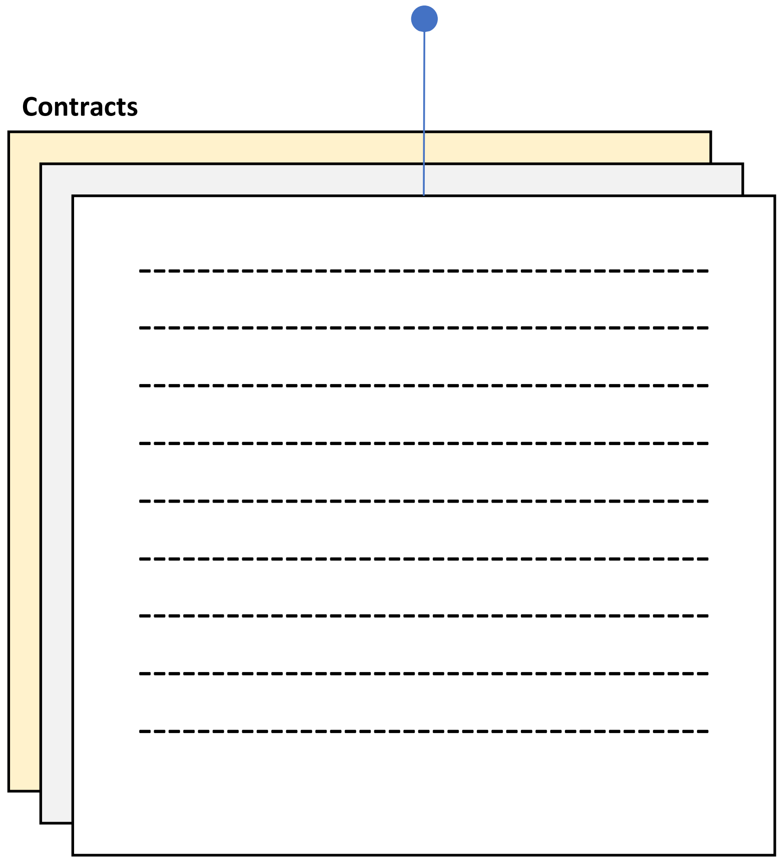
Define in Class Diagram



## Define Service Contracts

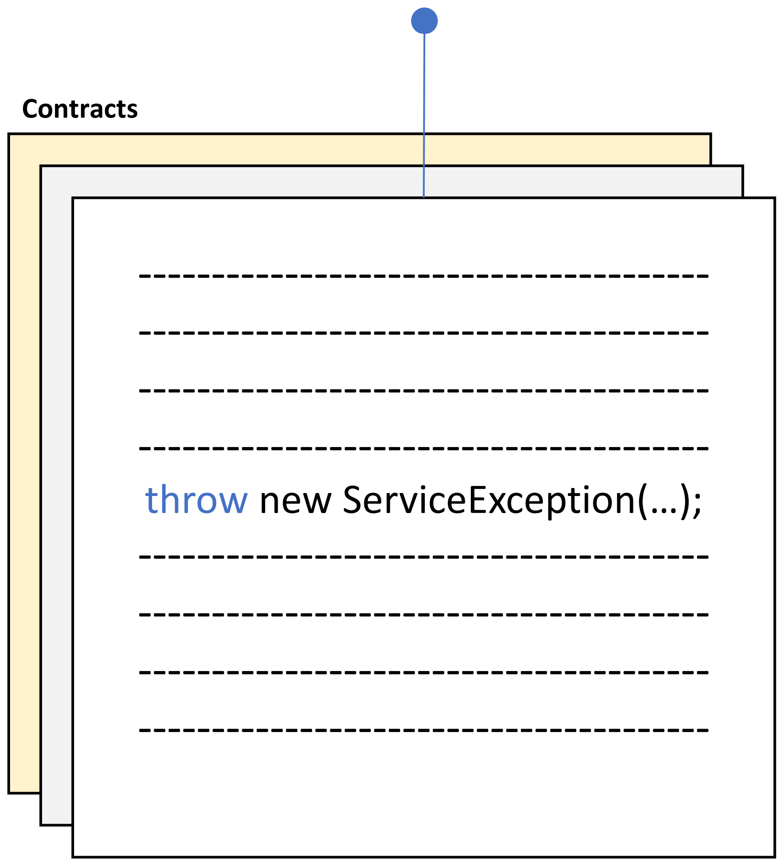
- CRUD or RPC
- Stateful/Stateless
- Transaction Management
- Handle invalid calls:
  - Timeouts
  - Duplicate calls
  - Calls out of order

Define in Component Diagram



## Plan Exception Handling

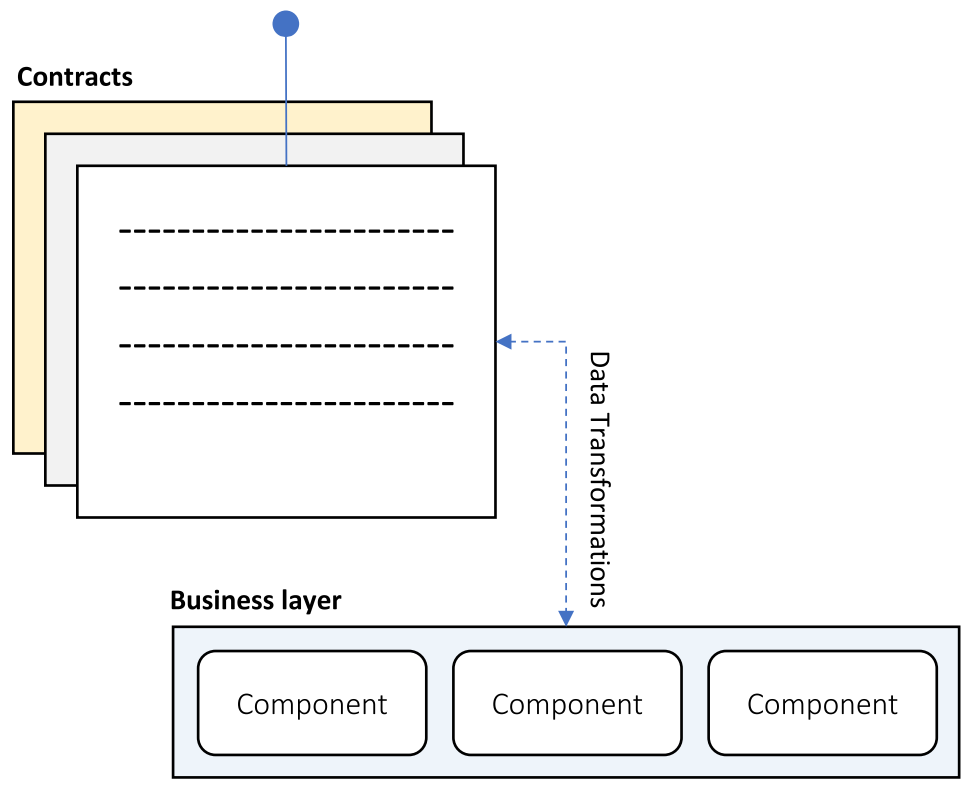
- Only catch what you can handle
- Use meaningful messages
  - Business explanation
  - Technical information
  - Retry instructions
- Return fault metadata
- Log everything
- Notify exception subscribers



## Define Business Entity Transforms

- Transform entities to messages:
  - Reference Business layer directly
  - Use an Object Mapper
  - Use an Object-Relational Mapper
  - Use a Transform Language
  - Custom-built

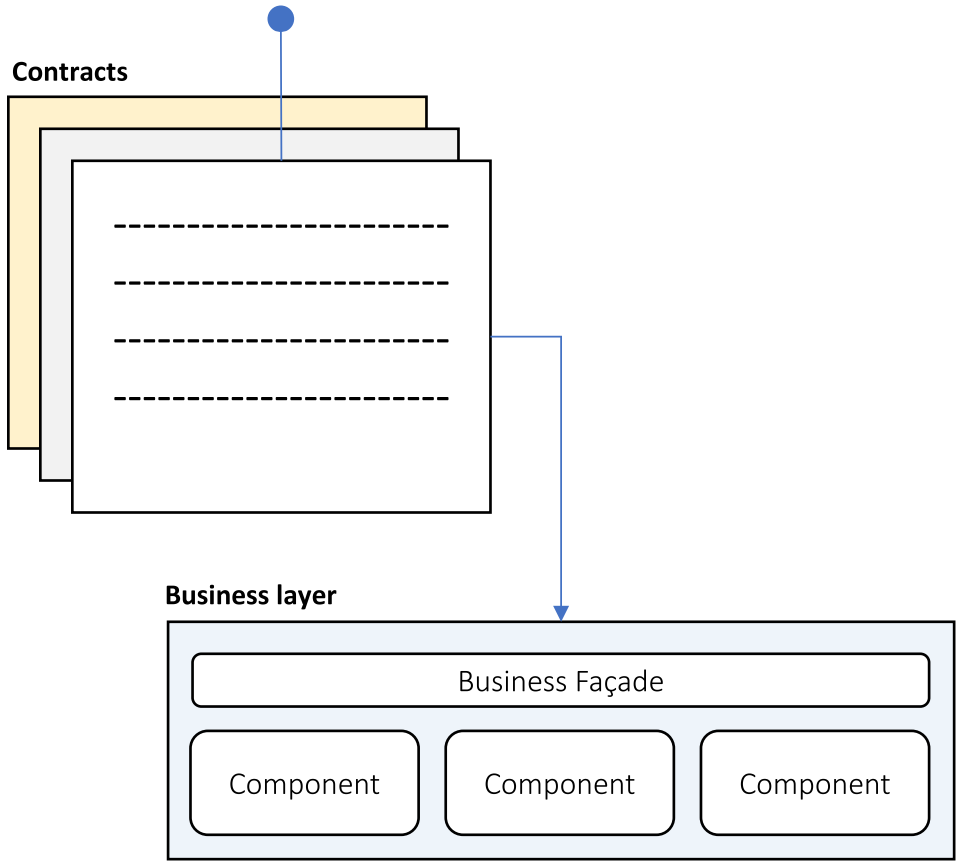
- Considerations:
  - Narrowing/Widening transforms
  - Flattening/Elevating transforms
  - Reversible transforms



# Define Business Abstraction

- Business layer abstractions:
  - Call the Façade directly
  - Call Business Components
  - (Re)Start a Workflow
  - Log a Business Event

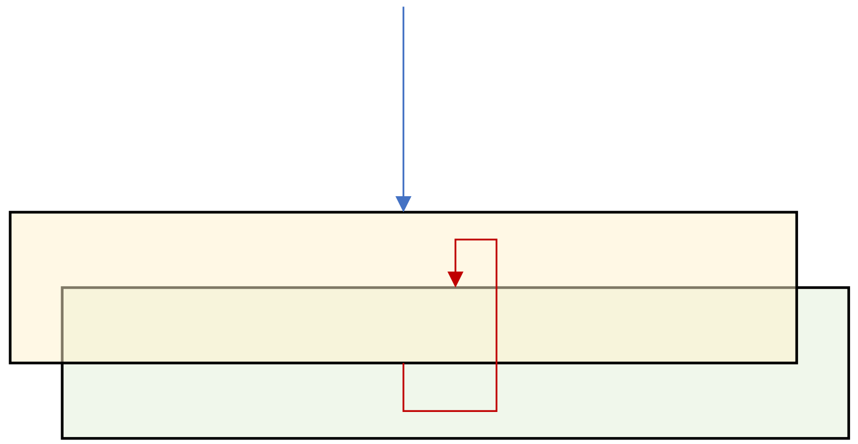
- Considerations:
  - Long-running workflows
  - State management
  - Transactions



## Design Quality Attributes

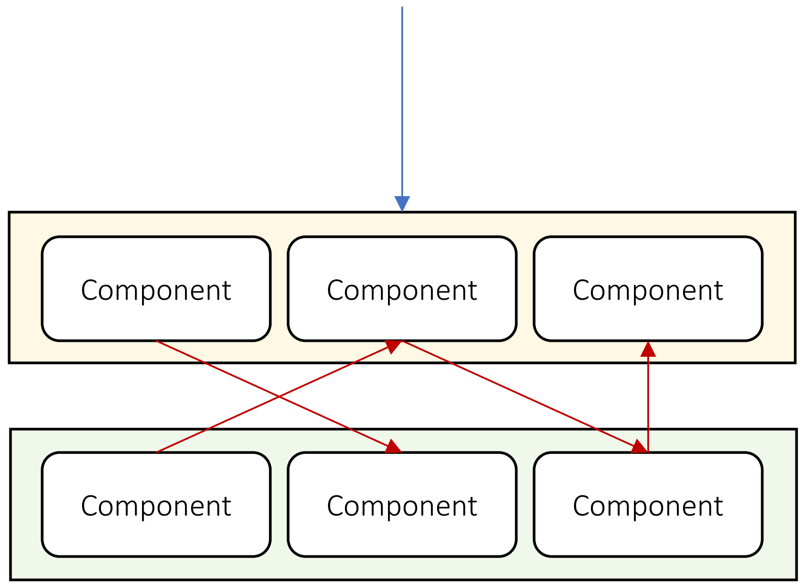
# Conceptual Integrity

- Isolated layers and components
- Application Lifecycle Management
- Healthy team collaboration
- Design and coding standards
- Break away from legacy designs:
  - Façade Pattern
  - Wrap as service
  - Rebuild from scratch



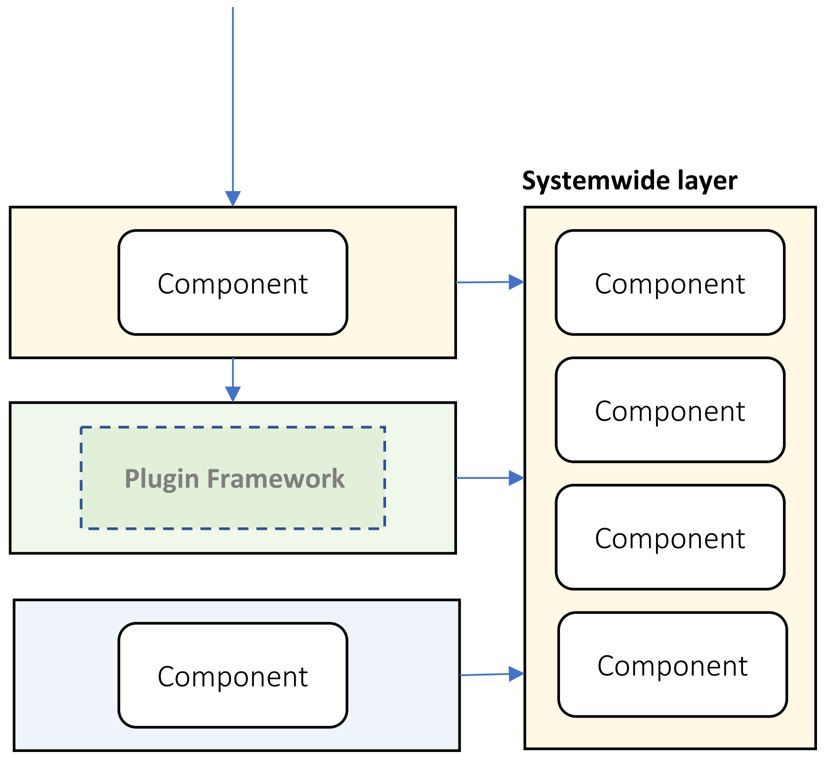
# Maintainability

- Isolated layers and components
- Structured communication
- Consider a plugin system
- Rely on platform features
- Use systemwide layer
- Add unit tests
- Documentation



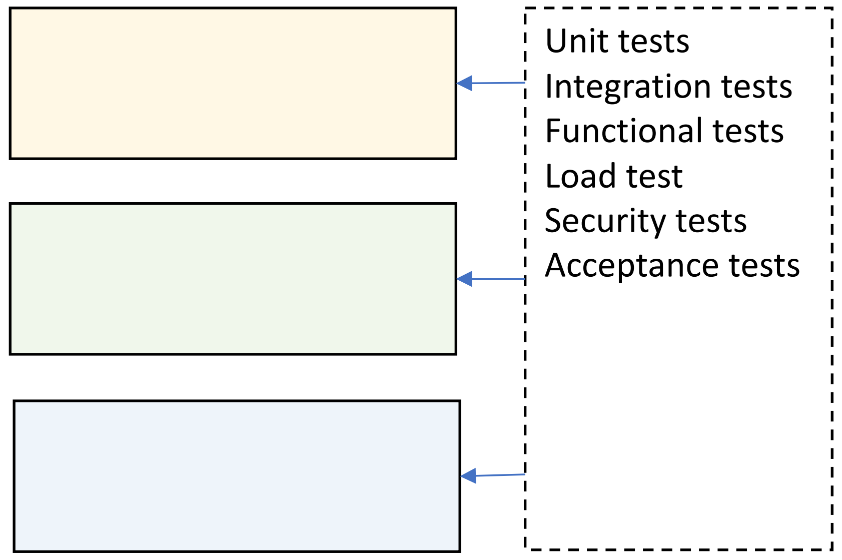
## Reusability

- Component-based architecture
- Adhere to standards
- General-purpose code
- Allow 3rd parties
- Use a plugin system
- Use systemwide layer



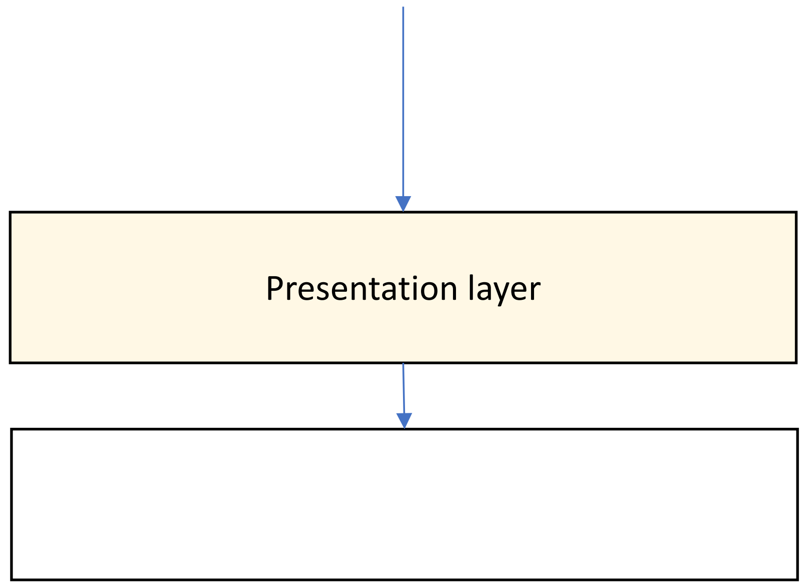
## Testability

- Design for testing
- Allow mocking
- Cover all layers
- Automate case studies
- Test:
  - Individual components
  - Entire layers
  - Collaboration between layers
  - Load, Security, …



## Usability

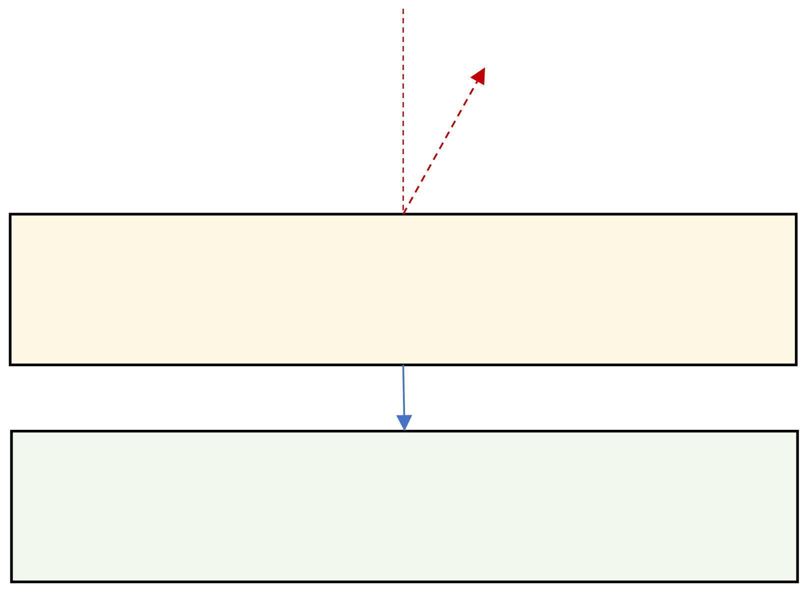
- Elegant & simple UI
- Implements all case studies in minimal number of interactions
- Clear multi-step workflows
- Intuitive feedback
- Non-technical



## Runtime Quality Attributes

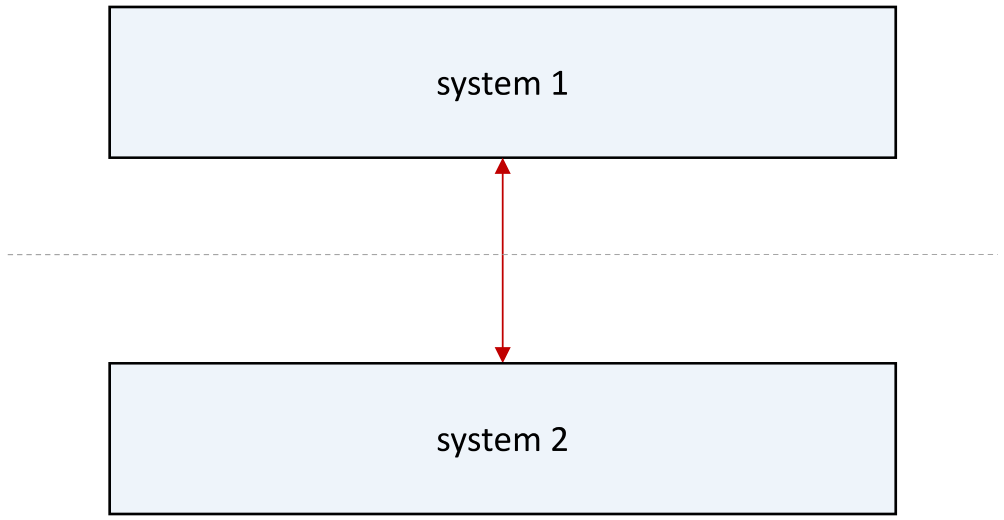
# Availability

- Tier failover
- Use rate limiter
- Short-lived resources locks
- Recover from exceptions
- Update-friendly architecture
- Handle network faults:
  - Offline support
  - Buffered proxy



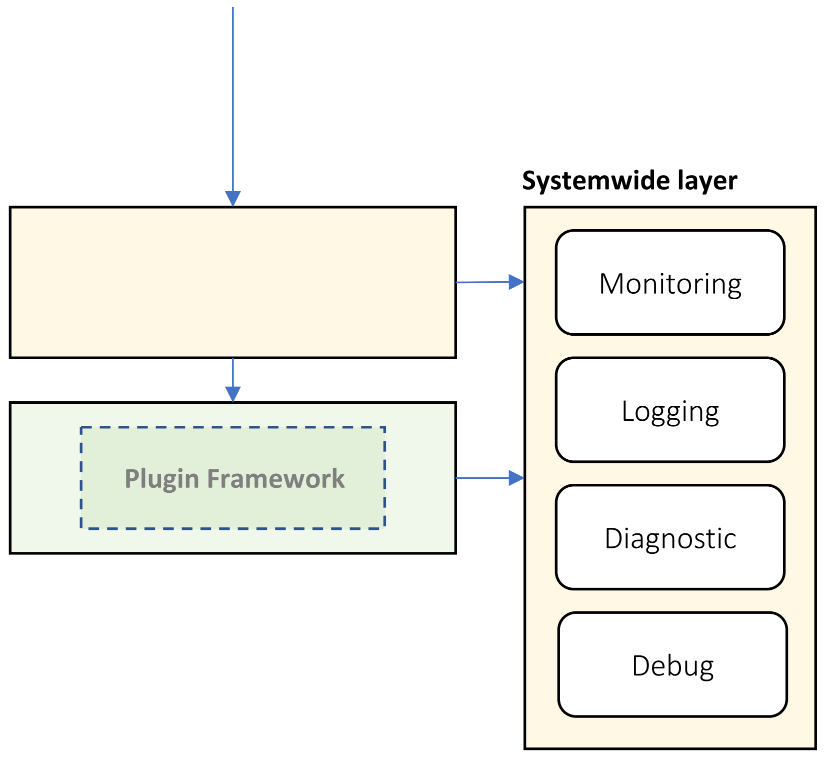
# Interoperability

- Data transformation
- Keep systems separate
- Adhere to standard:
  - SOAP
  - REST
  - XML/JSON



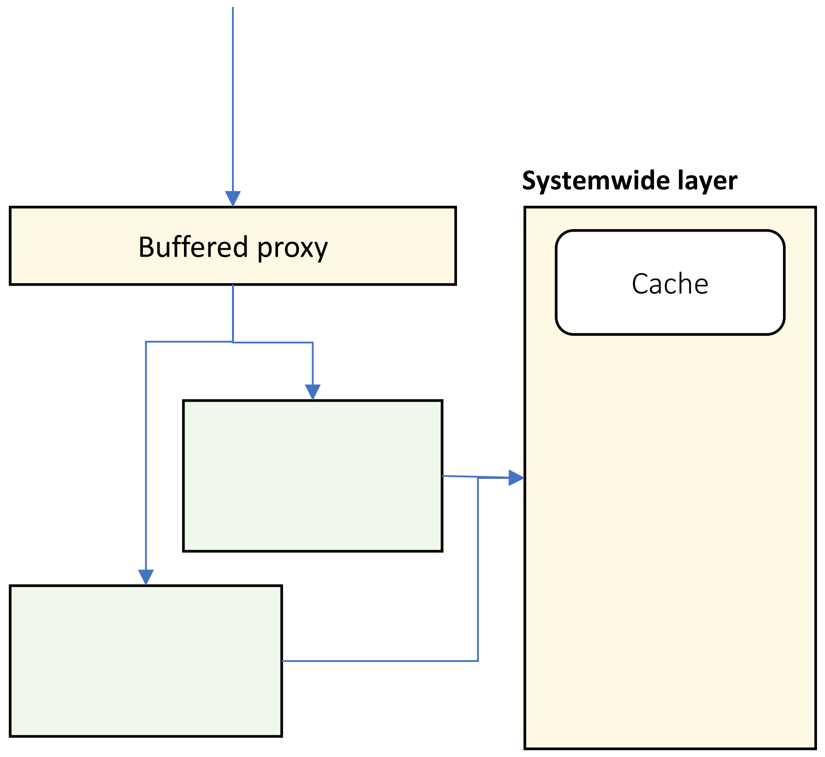
## Manageability

- Health monitoring, logging, and diagnostic tracing
- Consider a plugin system
- Declarative configuration
- Add diagnostic tools:
    - Live tracing
    - Diagnostic notifications
    - Runtime log inspection
    - In-situ debugging



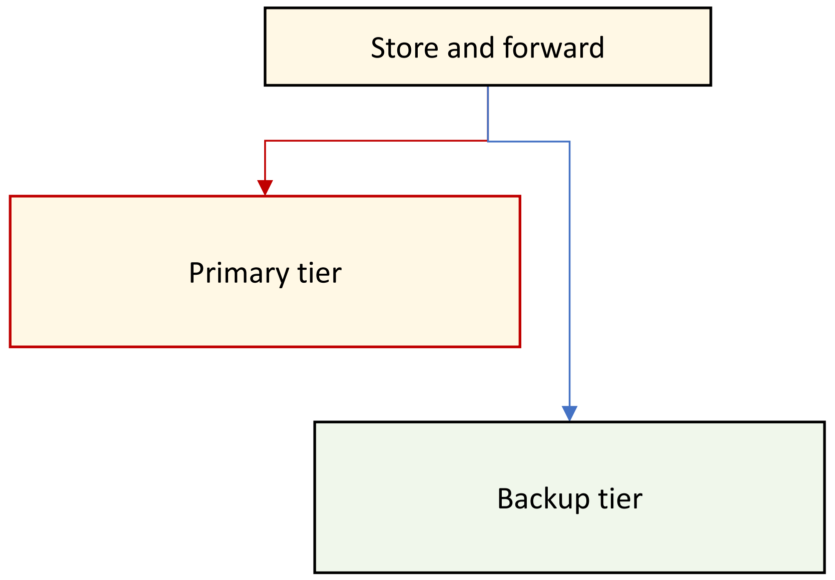
# Performance

- Buffered proxy
- Async responses
- Load-balanced tiers
- Caching
- Load tests
- Minimize throughput:
  - Rate limiting
  - Design coarse interfaces
  - Minimize cache misses



## Reliability

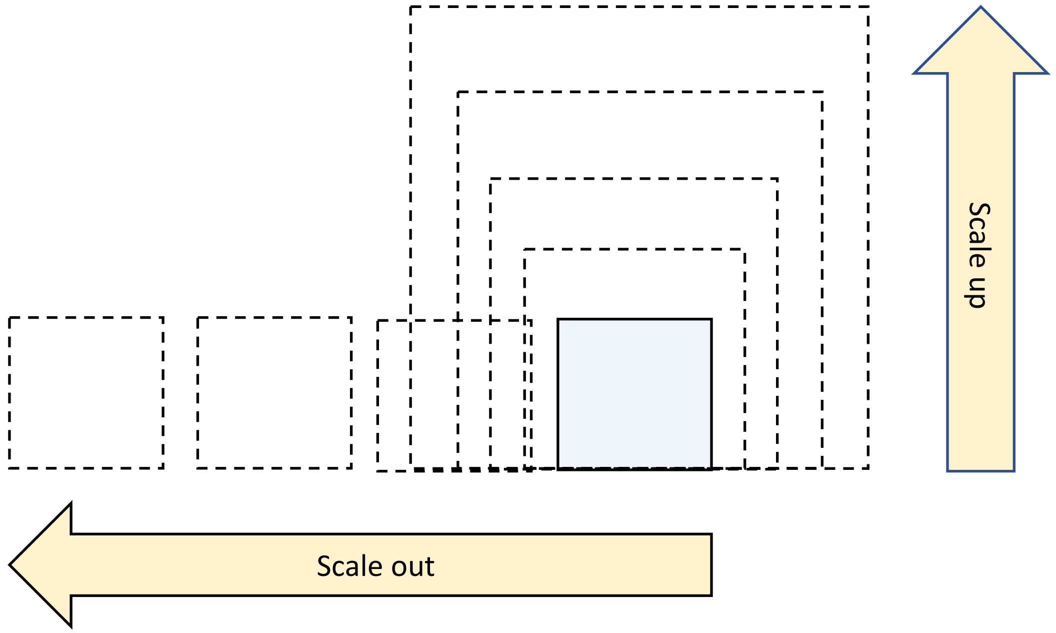
- Self-healing architecture
- Use store and forward
- Use alternative system if:
  - Primary system is offline
  - Primary system is very slow
  - Primary output is invalid
- Replay messages when external resources come back online



## Scalability

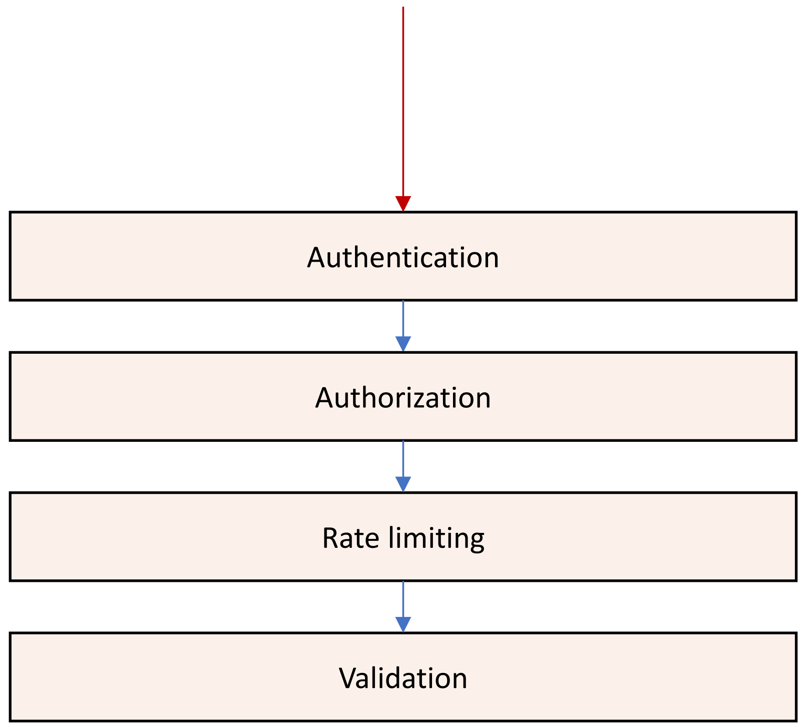
- Tier scaling:
  - Scale up
  - Scale out

- Handle load spikes:
  - Async responses
  - Store and forward
  - Allow stale data



## Security

- Authenticate & authorize clients
- Validate input & output
- Encrypt sensitive data
- Protect against:
  - Spoofing
  - Malicious input & output
  - Malicious use
  - Data theft
  - DDoS attacks

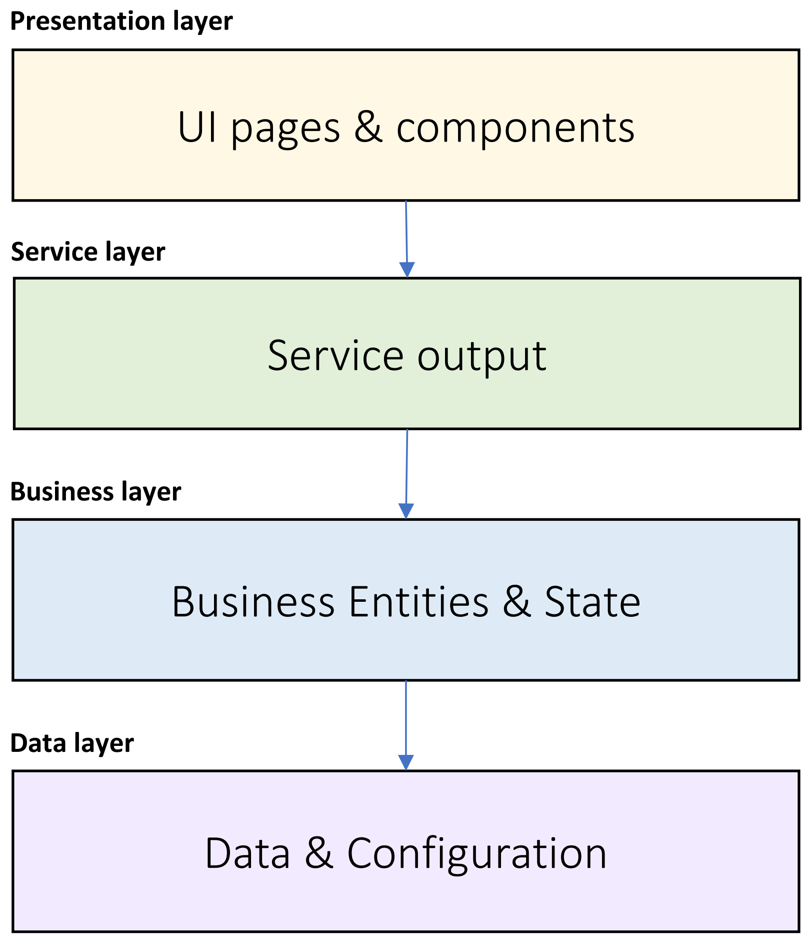


## Planning For Caching

# What To Cache?

- UI pages
- UI components
- Service output
- Business Entities
- Business State
- Data query results
- Configuration data

List data to be cached in each layer



## Where To Cache?

- Local memory
- State server
- File system
- Database

Use standard solutions like .NET Cache, Redis, Memcached, …

## How To Manage The Cache?

- Expiration strategy:
    - Time-based
    - Event-based

- Flush strategy:
    - Manual
    - Automatic:
        - Least Recently Used
        - Least Frequently Used
        - Priority

## How To Fill The Cache?

- Proactive Loading
  - Static data
  - Known update frequency
  - Known size

- Reactive Loading
  - Volatile data
  - Unknown lifetime
  - Large data volume
  - Fast caching medium

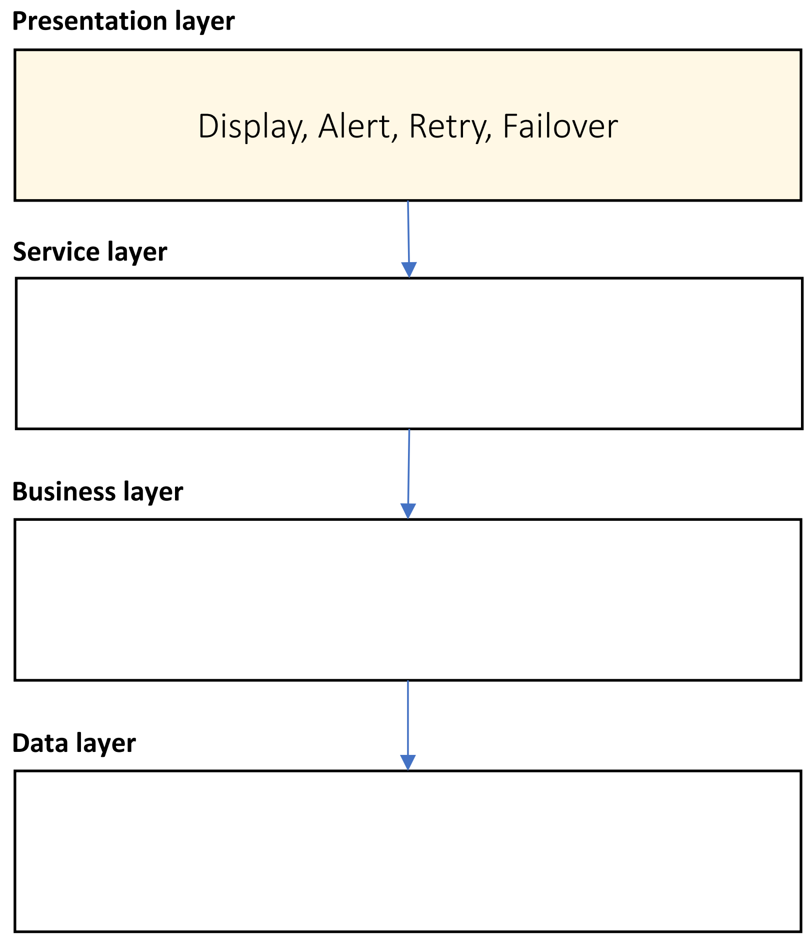
## Planning For Exceptions

## Exception Strategies

- Allow to propagate
- Catch and Re-throw
  - Logging
  - Retains stack trace
- Catch, Wrap, and Throw
  - Add metadata
  - Expose consistent exception types
- Catch and Discard

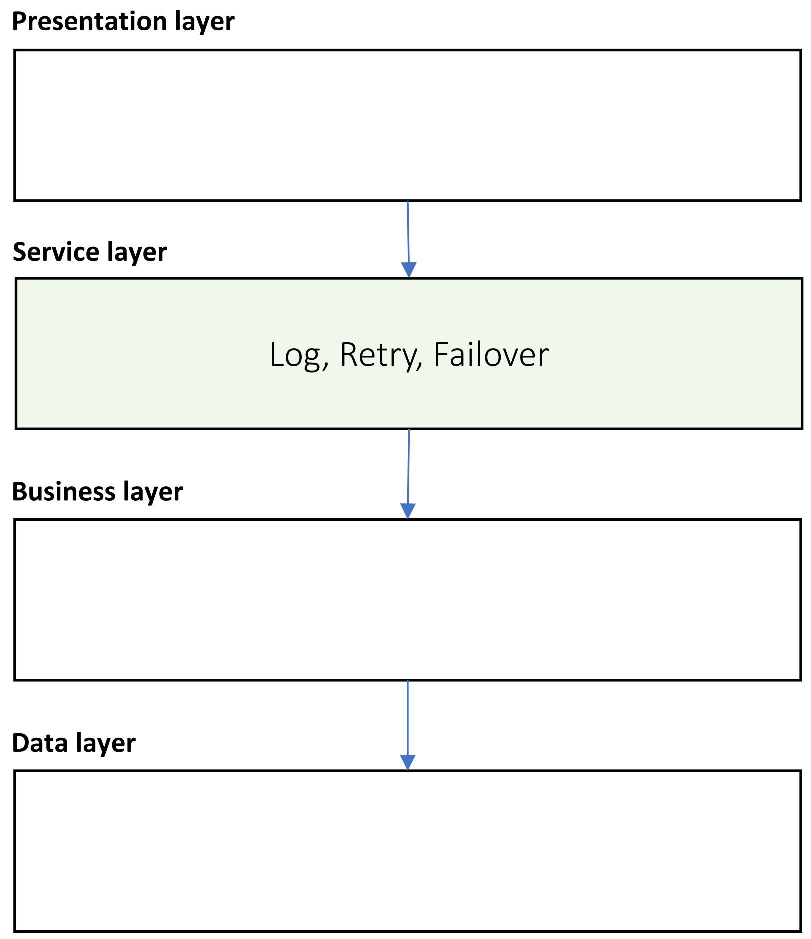
# Presentation Layer Exceptions

- Catch, Display, and Discard
- Attempt to retry
- Switch to secondary system
- Alert by Email, SMS, Slack, …
- Use meaningful messages
  - Business explanation
  - Technical information
  - Steps to resolve



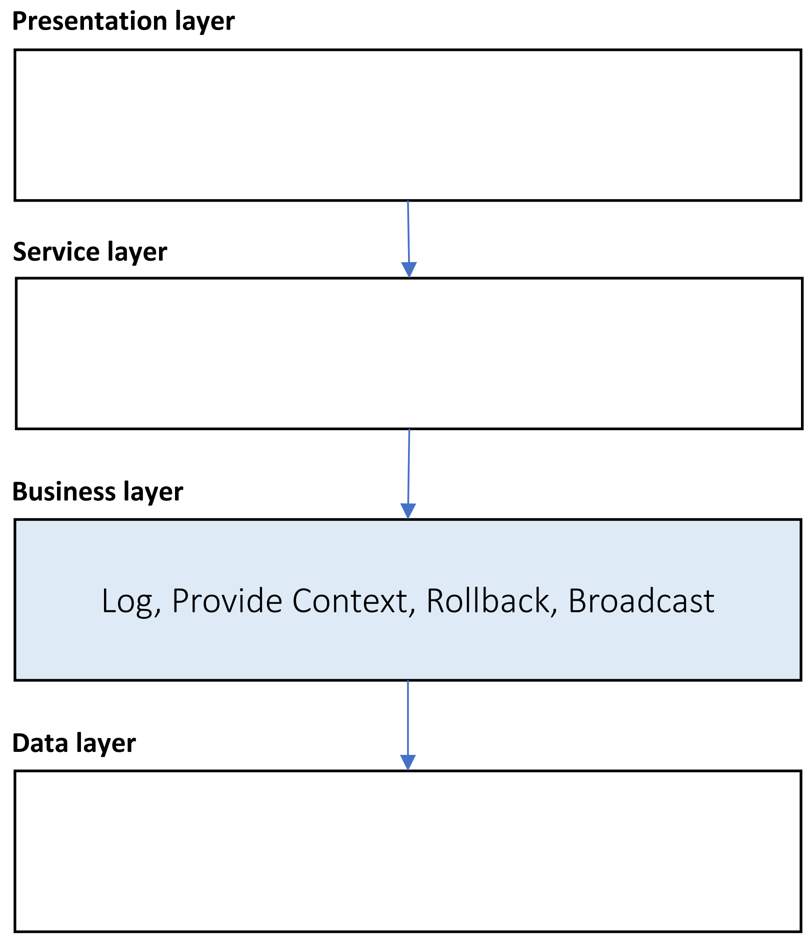
# Service Layer Exceptions

- Catch and Re-throw
- Attempt to retry
- Switch to secondary system
- Log exception and input message



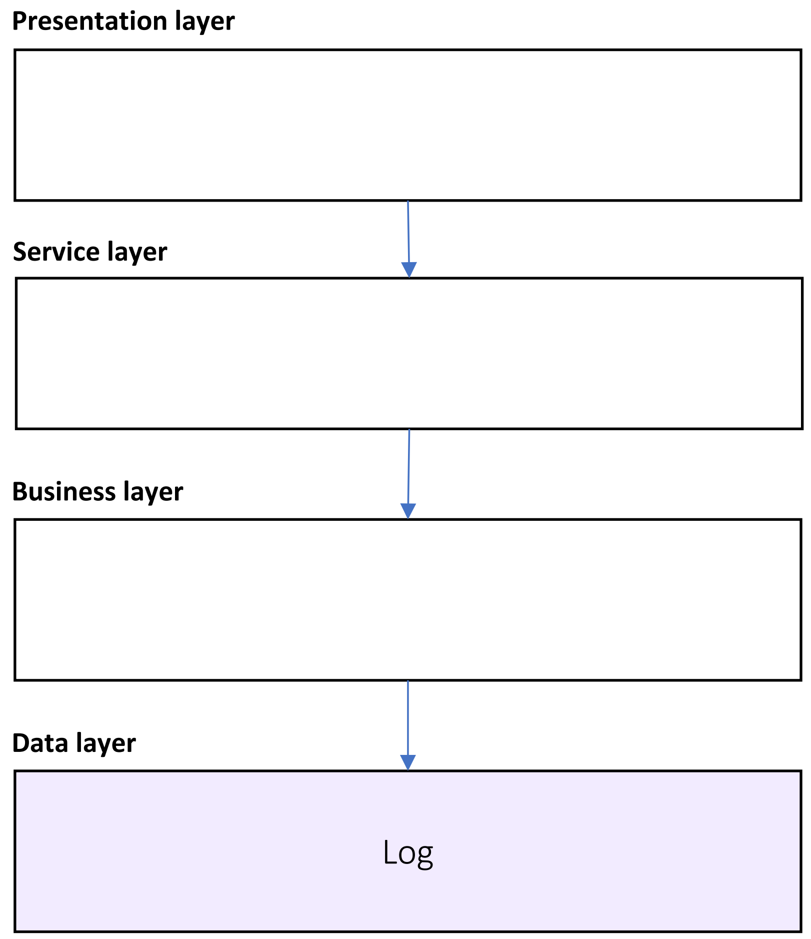
# Business Layer Exceptions

- Catch, Wrap, and Throw
- Use custom exception types
- Provide business context
- Rollback transactions
- Log exception and input args
- Broadcast to subscribers



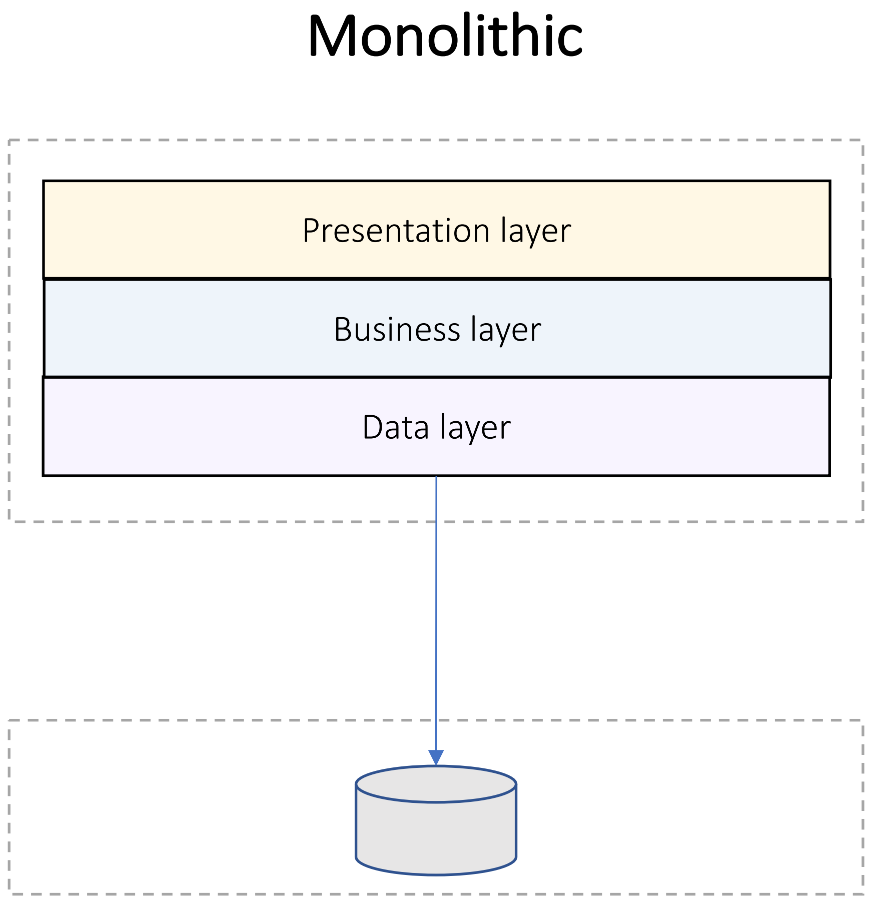
# Data Layer Exceptions

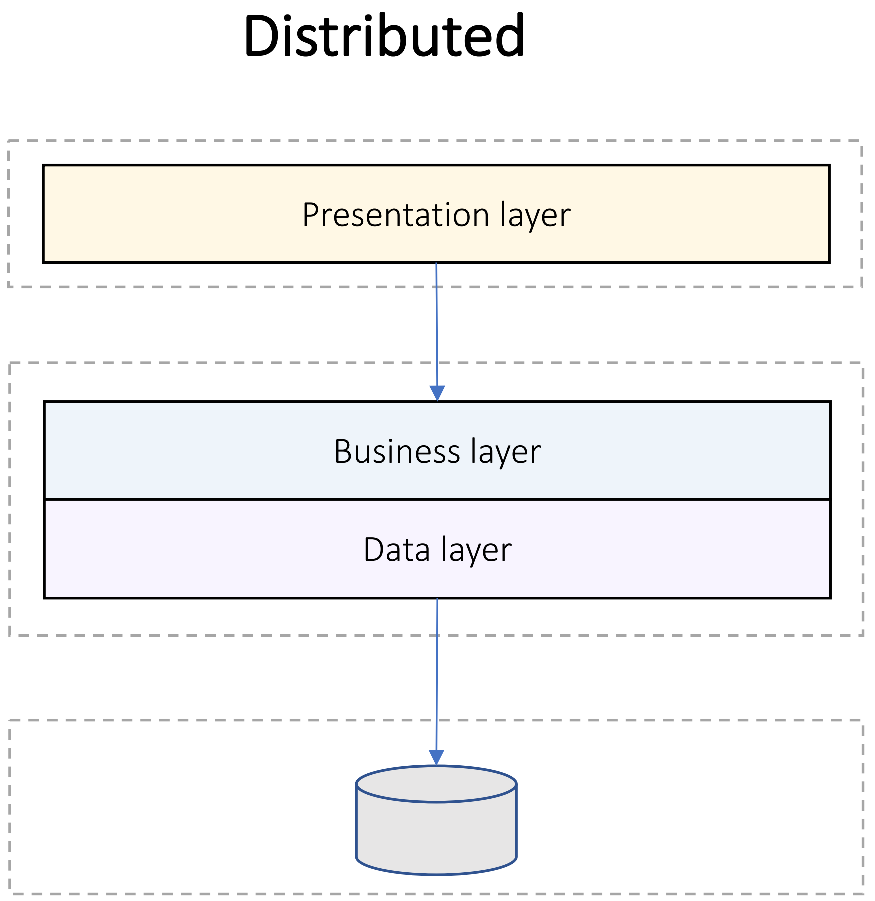
- Catch and Re-throw
- Log exception and input query



## Planning For Deployment

## Deployment Models



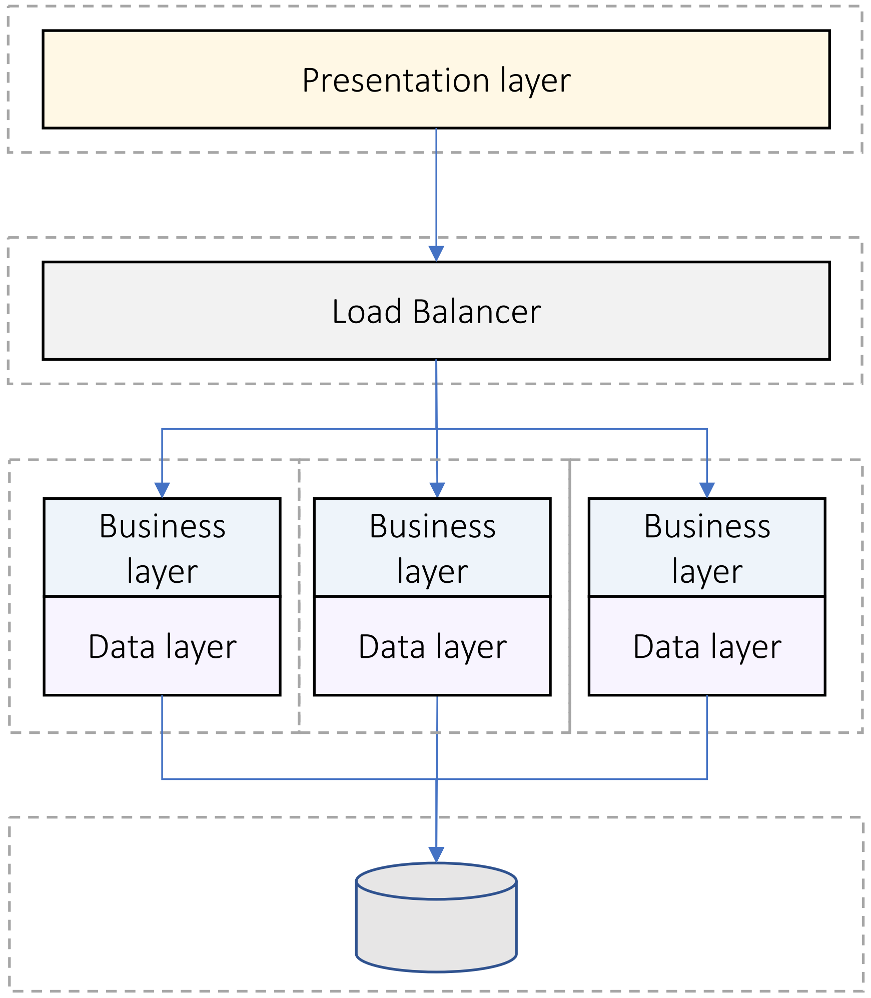


## Distributed Deployment Guidelines

- Minimize blocking calls:
  - Async calls
  - One-way calls
  - Buffering
- Use distributed transactions
- Use coarse-grained interfaces
- Manage state:
  - Stateless design – highly scalable
  - Stateful design – supports workflows but doesn’t scale
  - Shared State server

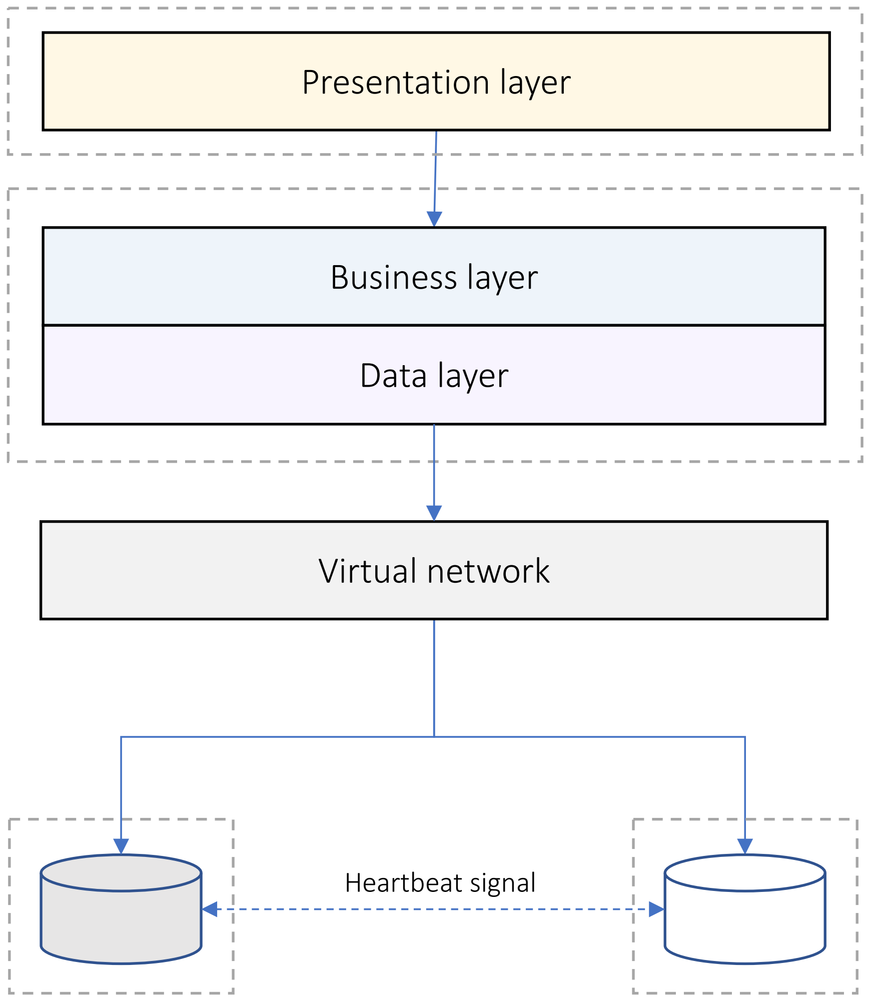
# Deploy For Performance

- Business/Data layers scale out
- Can detect failed tiers
- Stateless design preferred
- Stateful design requirements:
  - Shared State Server
  - Session Affinity



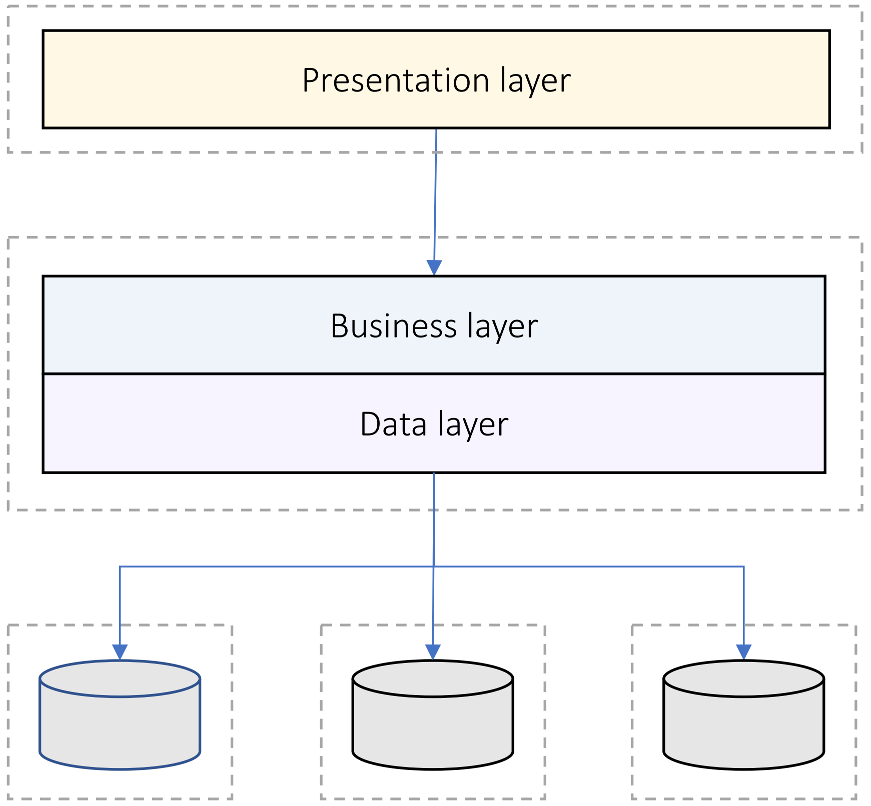
## Deploy For Reliability

- Secondary tier takes over when primary tier fails
- Requires 2x hardware
- Synchronization considerations:
  - Sync when secondary tier activates
  - Or: allow stale data



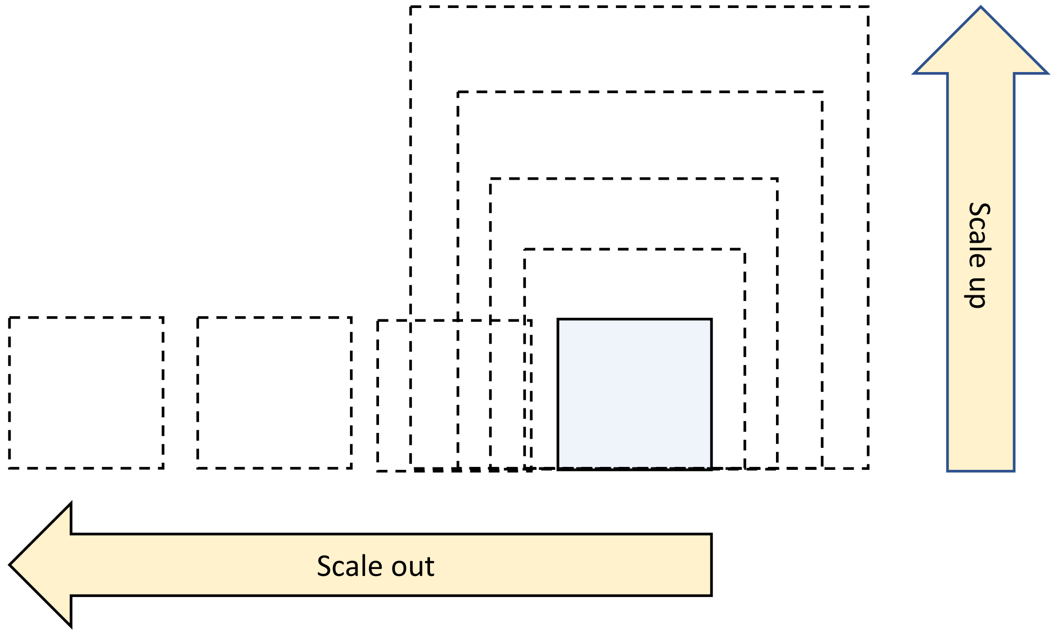
## Deploy For Scalability

- Data replicated on multiple tiers
- Replication breaks consistency and atomicity
- Consistency considerations:
  - Delayed sync in background
  - Or: allow stale data
  - Or: partition data



# Scale Up And Scale Out

- Scale Up
  - Easy with VMs
  - … but limited results
- Scale Out
  - Requires layered design
  - Requires partitioned data
  - Potentially unlimited


Scale up
Scale out