Solution
Technology
Unit

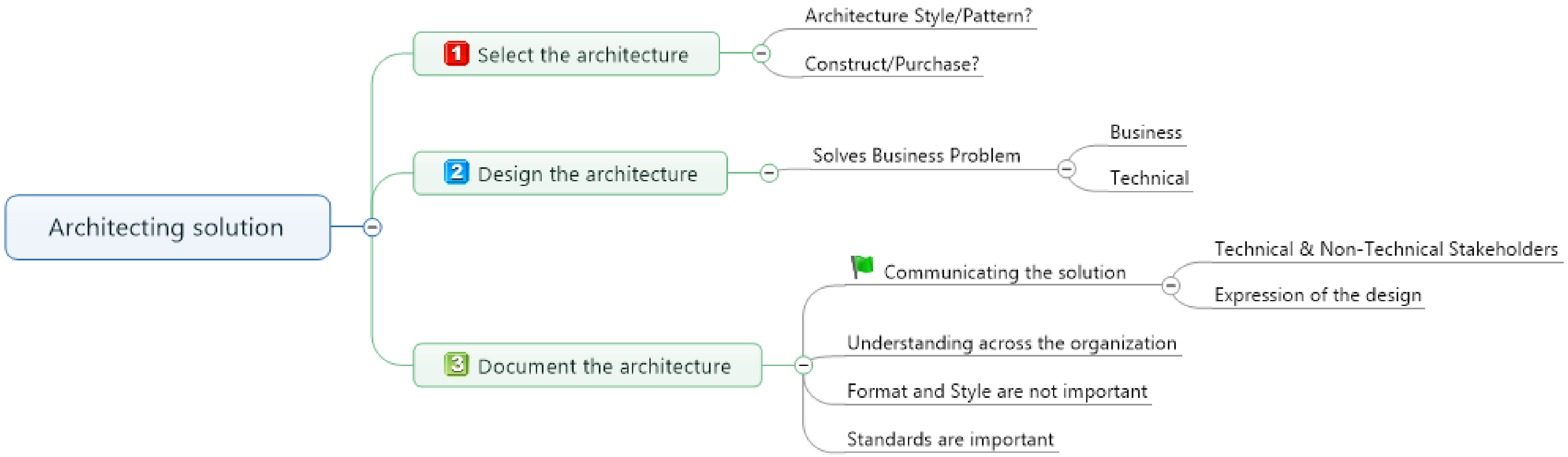
# SA – Workshop #1
# Design The Solution

Presenter: KhoaTND@fpt.com
Head of Solution & Technology Unit
FPT Software

Oct/2024

# Ask Your Question in SHY way

# Architecture Duties in 1-Page



Architecting solution

1 Select the architecture
- Architecture Style/Pattern?
- Construct/Purchase?

2 Design the architecture
- Solves Business Problem
  - Business
  - Technical

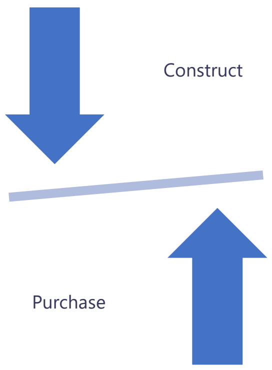
3 Document the architecture
- Communicating the solution
  - Technical & Non-Technical Stakeholders
  - Expression of the design
- Understanding across the organization
- Format and Style are not important
- Standards are important

**Style**

- Client Server
- Message Bus
- Service Oriented Architecture
- Domain Driven Design
- Layered Architecture
- Component Based
- …

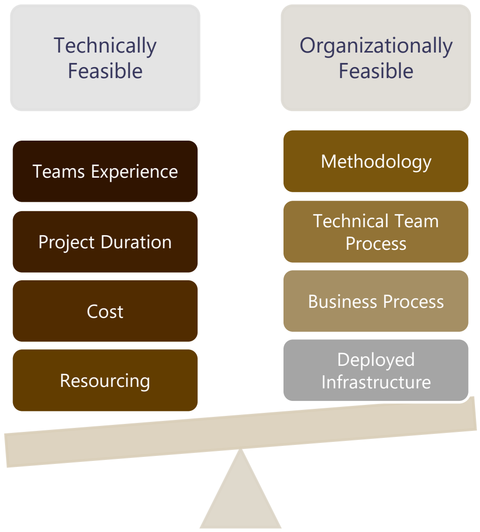
**Pattern**

- MVC
- Publish/Subscribe
- Request/reply
- Peer-to-peer
- …



Construct

Purchase



Technically Feasible

- Teams Experience
- Project Duration
- Cost
- Resourcing

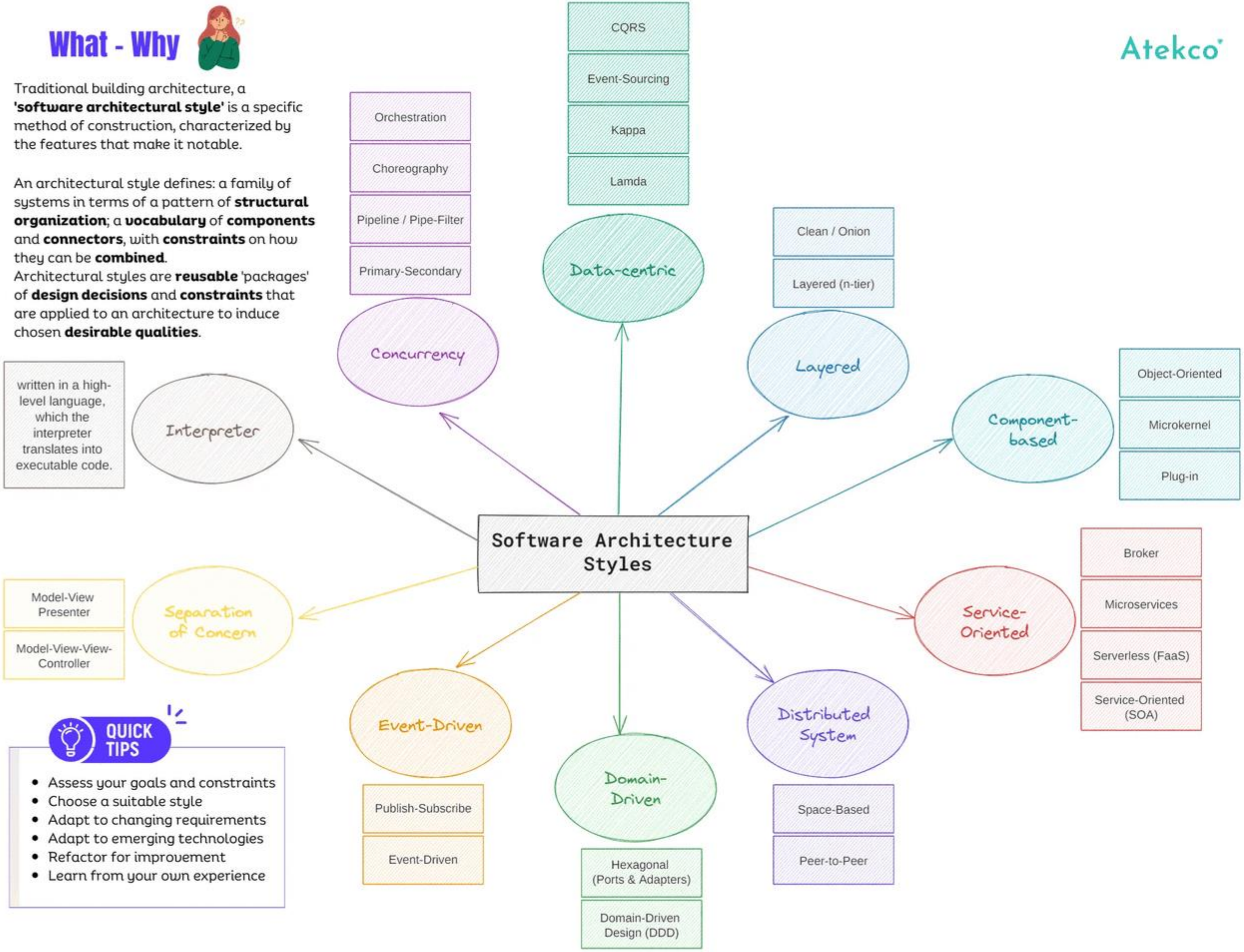
Organizationally Feasible

- Methodology
- Technical Team Process
- Business Process
- Deployed Infrastructure

# What - Why

Traditional building architecture, a **'software architectural style'** is a specific method of construction, characterized by the features that make it notable.

An architectural style defines: a family of systems in terms of a pattern of **structural organization**; a **vocabulary** of **components** and **connectors**, with **constraints** on how they can be **combined**.
Architectural styles are **reusable** 'packages' of **design decisions** and **constraints** that are applied to an architecture to induce chosen **desirable qualities**.



Software Architecture Styles

Interpreter

written in a high-level language, which the interpreter translates into executable code.

Concurrency

Orchestration

Choreography

Pipeline / Pipe-Filter

Primary-Secondary

Data-centric

CQRS

Event-Sourcing

Kappa

Lamda

Layered

Clean / Onion

Layered (n-tier)

Component-based

Object-Oriented

Microkernel

Plug-in

Service-Oriented

Broker

Microservices

Serverless (FaaS)

Service-Oriented (SOA)

Distributed System

Space-Based

Peer-to-Peer

Domain-Driven

Hexagonal (Ports & Adapters)

Domain-Driven Design (DDD)

Event-Driven

Publish-Subscribe

Event-Driven

Separation of Concern

Model-View Presenter

Model-View-View-Controller

## QUICK TIPS

- Assess your goals and constraints
- Choose a suitable style
- Adapt to changing requirements
- Adapt to emerging technologies
- Refactor for improvement
- Learn from your own experience

# Know your requirements

System requirements can be categorized as:

### Functional Requirements

- These requirements state what the system must do, how it must behave or react to run-time stimuli.

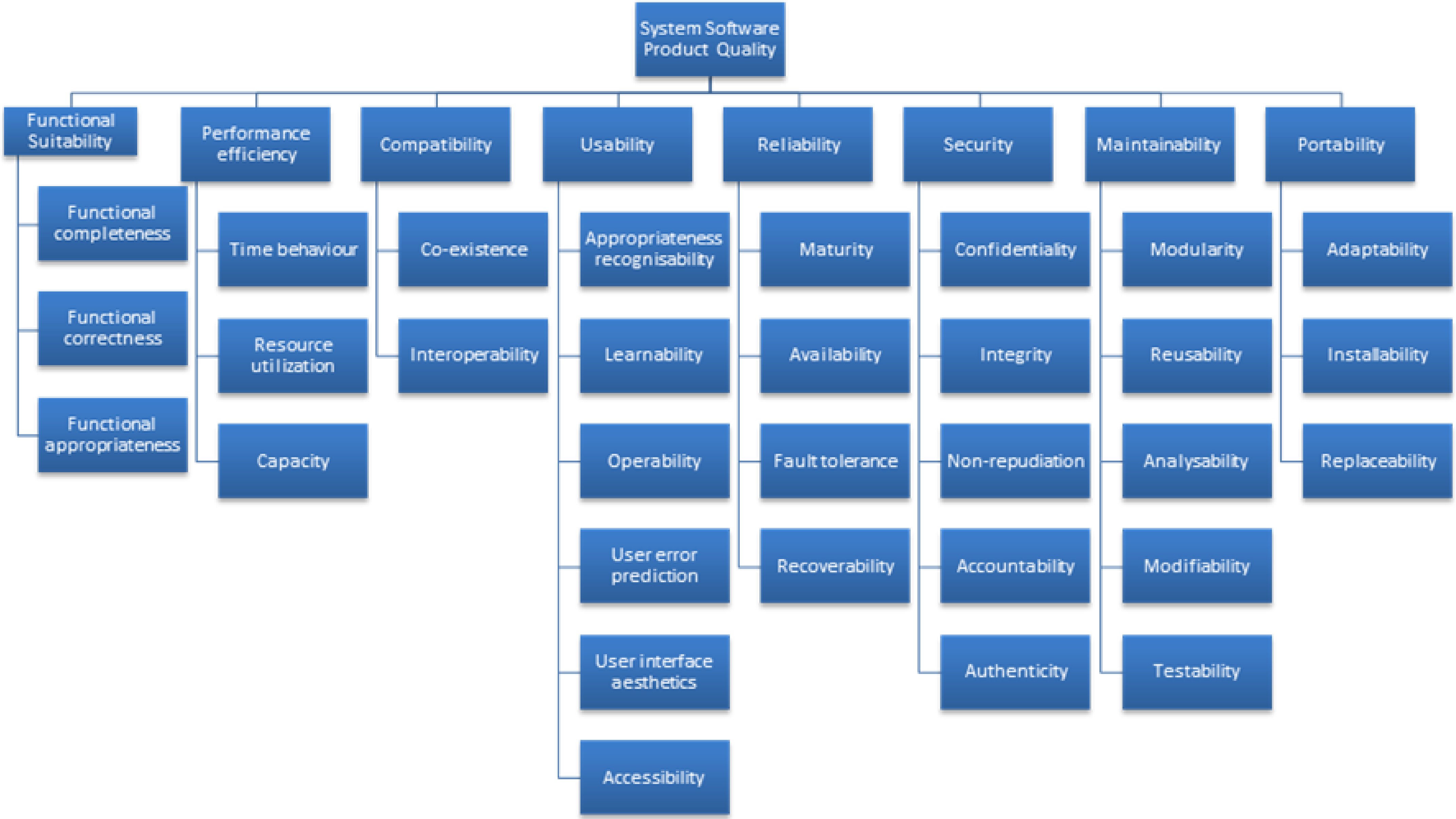
### Quality Attribute Requirements

- These requirements annotate (qualify) functional requirements
- Qualification might be how fast the function must be performed, how resilient it must be to erroneous input, how easy the function is to learn, etc

### Constraints

- A constraint is a **design decision with zero degrees of freedom**. That is, it's a design decision that has already been made for you

# ISO/IEC FCD 25010 - Product Quality Standard



- System Software Product Quality
  - Functional Suitability
    - Functional completeness
    - Functional correctness
    - Functional appropriateness
  - Performance efficiency
    - Time behaviour
    - Resource utilization
    - Capacity
  - Compatibility
    - Co-existence
    - Interoperability
  - Usability
    - Appropriateness recognisability
    - Learnability
    - Operability
    - User error prediction
    - User interface aesthetics
    - Accessibility
  - Reliability
    - Maturity
    - Availability
    - Fault tolerance
    - Recoverability
  - Security
    - Confidentiality
    - Integrity
    - Non-repudiation
    - Accountability
    - Authenticity
  - Maintainability
    - Modularity
    - Reusability
    - Analysability
    - Modifiability
    - Testability
  - Portability
    - Adaptability
    - Installability
    - Replaceability

# Approach to Architectural Design - Top-down

- Traditional approach
- Break down the system into a series of components
- Begins at the highest level of detail
- Performed iteratively → Series of sequential decomposition exercises
- Series of black boxes, interfaces and relationships → Basis for implementation choices
- Common in the enterprise
- Most effective when the problem domain is well understood
- Architect focuses on the larger issues up front

## Benefits

- Effective on both large and small projects
- Provides a logical and systematic approach
- Lends itself to system partitioning
- Helps to reduces size, scope and complexity of each module
- Works for both functional and object oriented design

## Drawbacks

- Requires an in depth understanding of problem domain
- Partitioning doesn’t facilitate reuse
- Sometimes leads to ivory tower architecture
- Design flaws can sometimes ripple up to the highest layers

# Approach to Architectural Design - Bottom-up

- Process of defining the system in small parts
- Like assembling Legos
- Typically encountered when using an agile development methodology
- More common than top-down in small project?
- Is there an architecture when you chose bottom-up?
- As the project progresses the architecture really does emerge…eventually

## Advantages

- Allows a team to begin coding and testing early
- Simplicity
- Promotes code reuse
- Promotes the use of continuous integration and unit testing

## Disadvantages

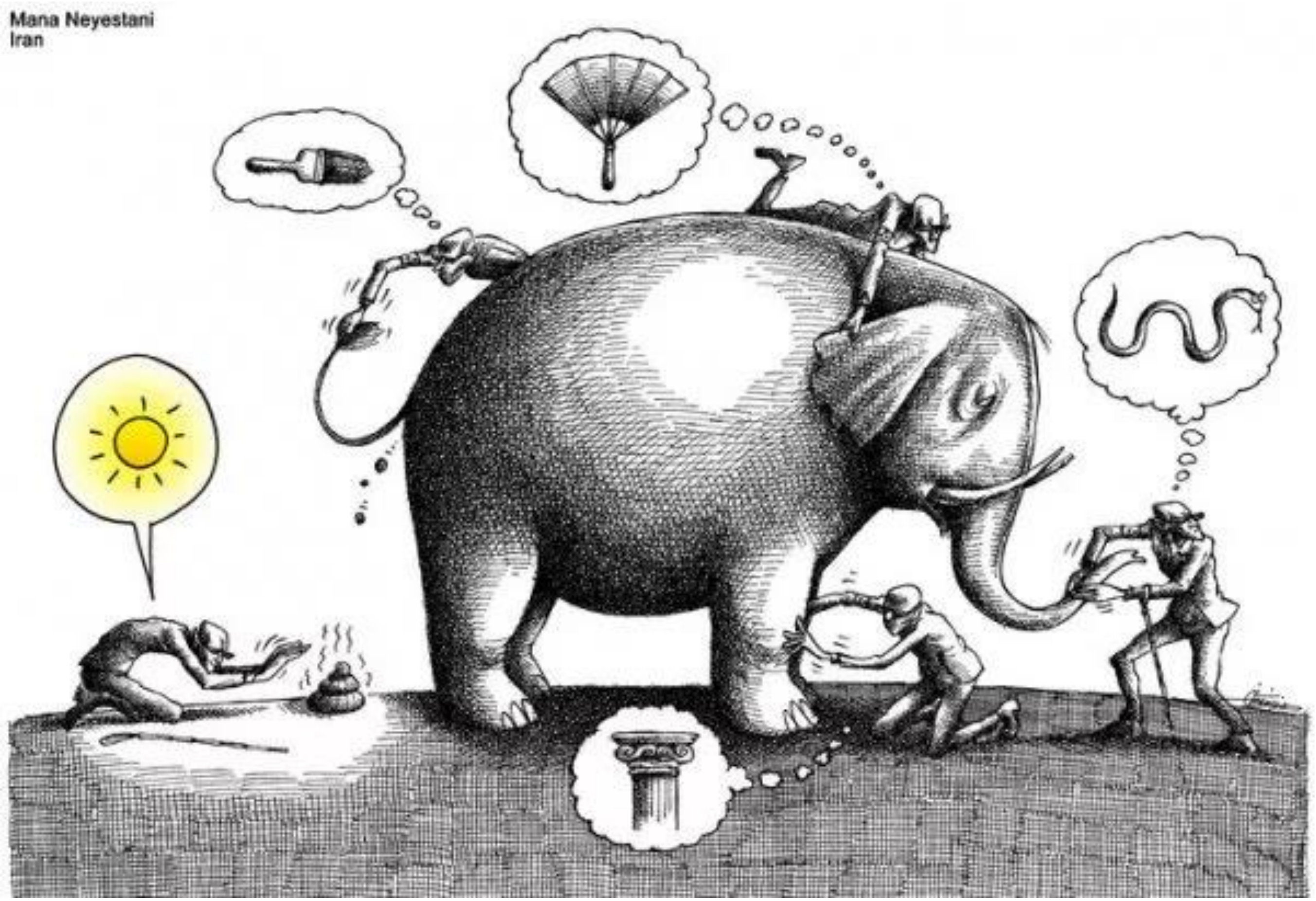
- Can become difficult to maintain
- Benefits of code reuse are eliminated or at least delayed as the team grows
- Design flaws can ripple throughout the entire solution

# Communication Your Design

- **How do we know we are meeting these objectives?**
  - Ask yourself two questions
    - Q1: Does this document provide value?
    - Q2: Does this level of detail communicate enough?
  - Second question tells us when to stop
    - Is there enough detail for our business users to understand how we are meeting their needs?
    - Is there enough detail for our development team to build a solution?
  - When both of these questions are answered then you have provided enough detail

***Remembers: ADR ?***

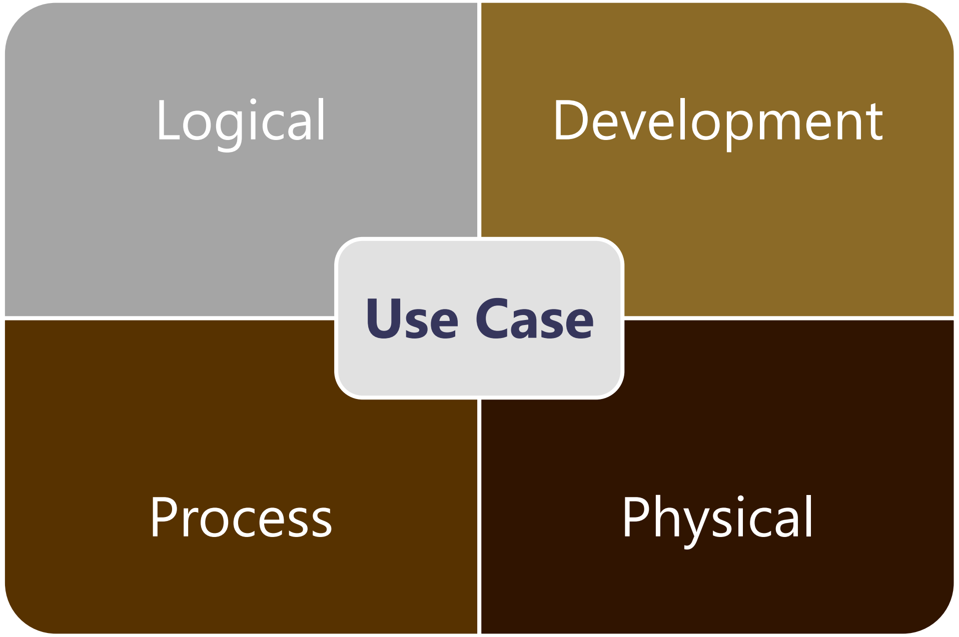
# 4+1 Views



## What are views?

- A window into the architecture
- Single viewpoint targeted to a particular audience
- No single view
- No set number and types of views
- The architecture is comprised of all the views
- Several well-known approaches rely on of views
  - 4+1 architectural view model
  - Views & Beyond

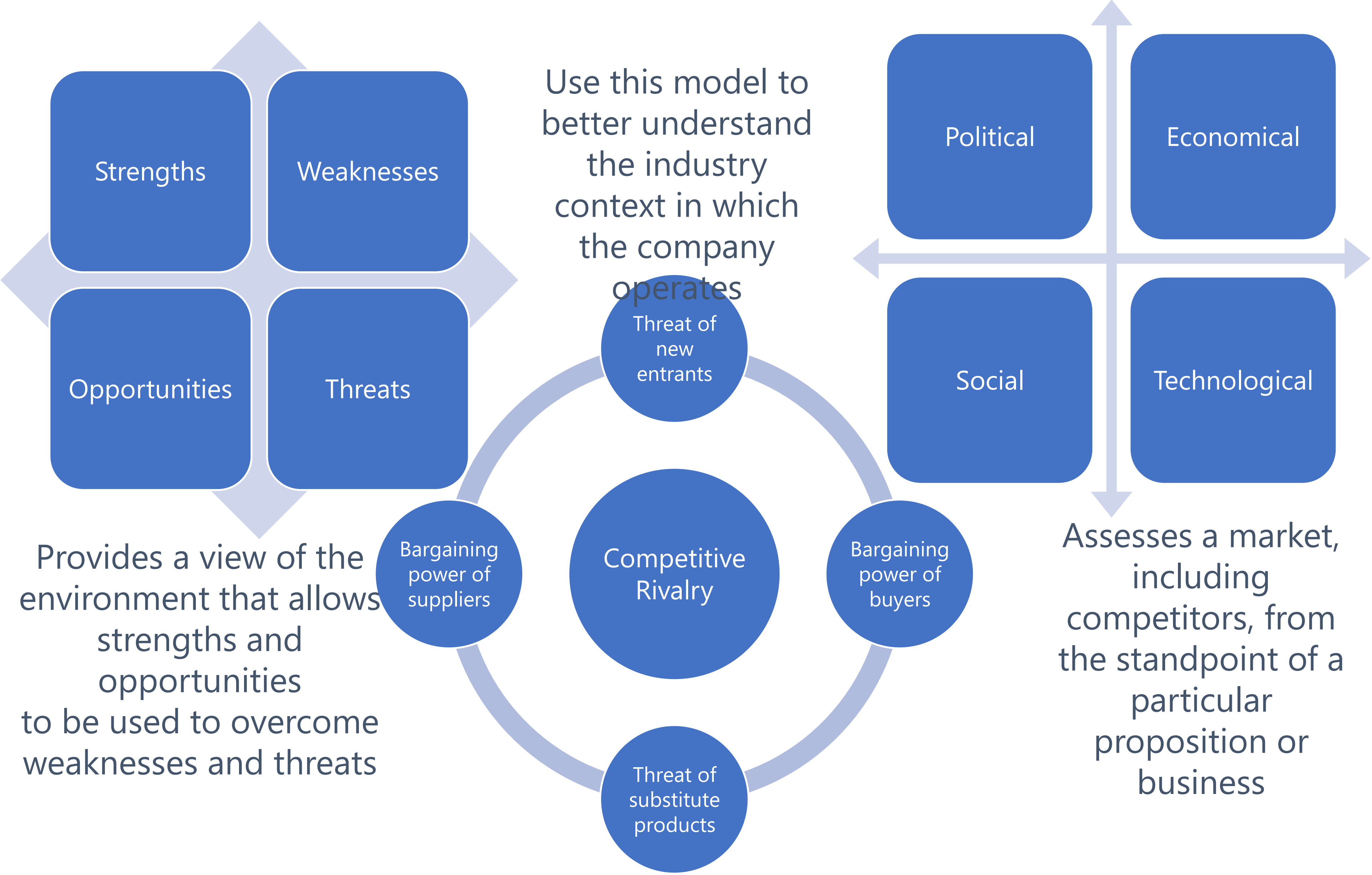
# 4+1 Views Spec



- **Use-Case**
  - Ties all of the other views together
  - User requirements
  - System functionality
  - Internal and external actors
  - Represented using use UML case diagrams
- **Logical**
  - End user functionality viewpoint
  - Structures of the architecture that implement functional requirements
  - Classes and their relationships
  - Represented using UML class diagrams

- **Development**
  - Structure & organizational viewpoint
  - Modules are organized
  - Module interaction
  - Represented with a UML package and component diagram
- **Process**
  - Run-time viewpoint
  - Performance
  - Reliability
  - Scalability
  - Interaction and communication
  - Represented using UML activity diagrams
- **Physical**
  - Infrastructure viewpoint
  - Deployment
  - Communications between physical tiers
  - Represented with a UML deployment diagram

# SA to Advisory



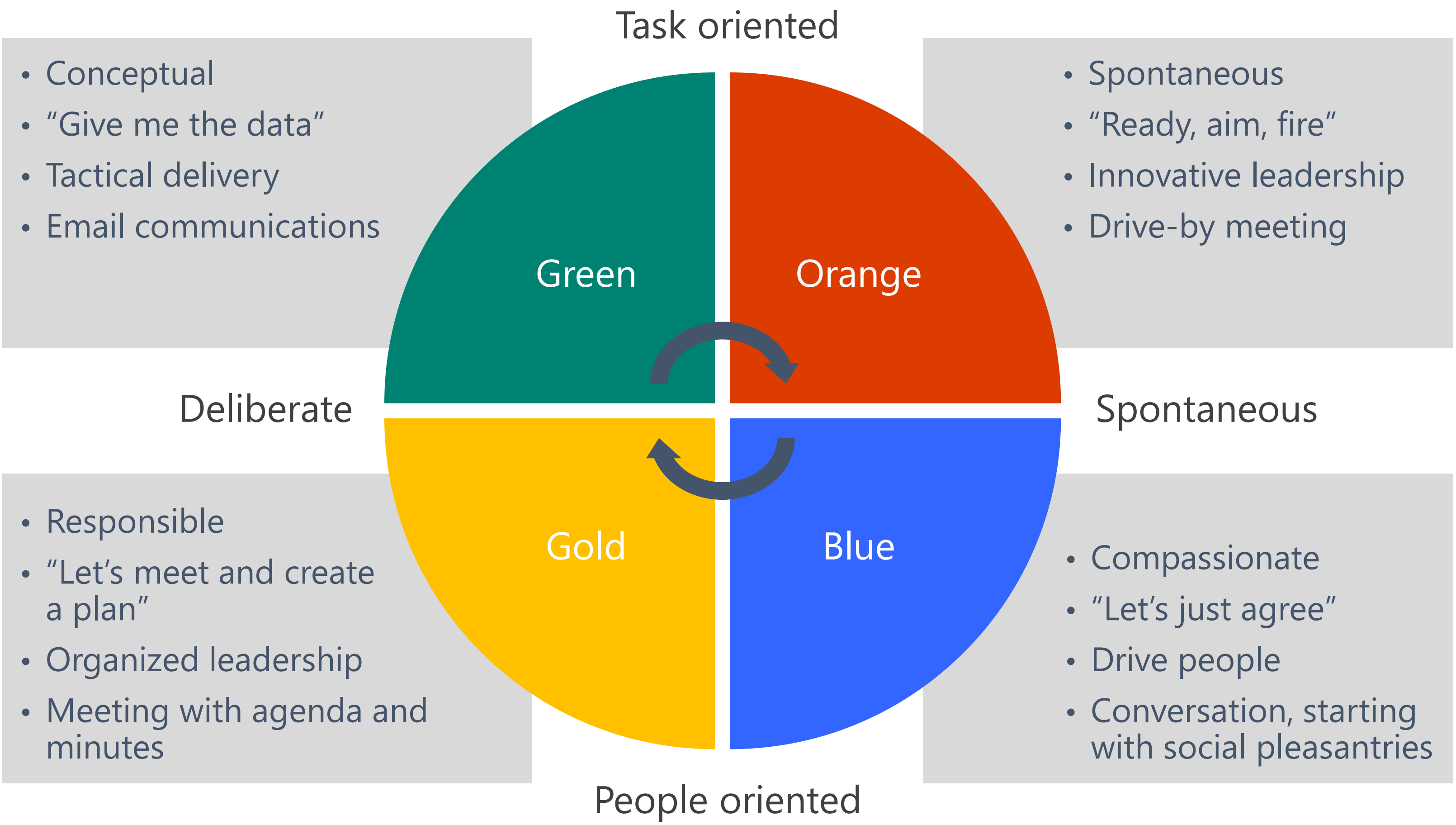
Provides a view of the environment that allows strengths and opportunities
to be used to overcome weaknesses and threats

Use this model to better understand the industry context in which the company operates

Assesses a market, including competitors, from the standpoint of a particular proposition or business

# Know your audience

A personality test shows perspective and communication style for stakeholders



Task oriented

- Conceptual
- “Give me the data”
- Tactical delivery
- Email communications

- Spontaneous
- “Ready, aim, fire”
- Innovative leadership
- Drive-by meeting

Green

Orange

Deliberate

Spontaneous

Gold

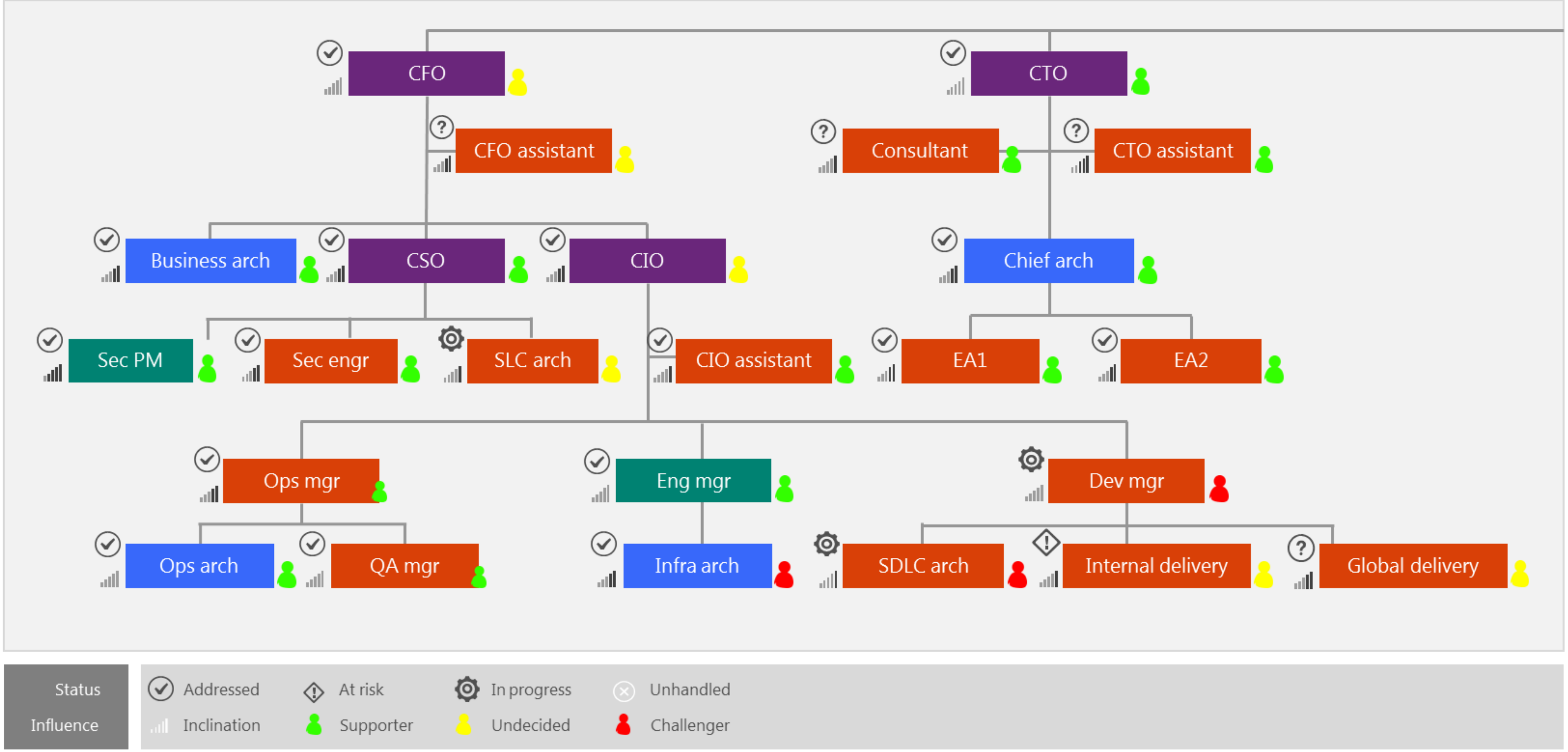
Blue

- Responsible
- “Let’s meet and create a plan”
- Organized leadership
- Meeting with agenda and minutes

- Compassionate
- “Let’s just agree”
- Drive people
- Conversation, starting with social pleasantries

People oriented

# Political Alignment Chart



CFO

CTO

CFO assistant

Consultant

CTO assistant

Business arch

CSO

CIO

Chief arch

Sec PM

Sec engr

SLC arch

CIO assistant

EA1

EA2

Ops mgr

Eng mgr

Dev mgr

Ops arch

QA mgr

Infra arch

SDLC arch

Internal delivery

Global delivery

| | | | | |
|---|---|---|---|---|
| Status | ✓ Addressed | ◇ At risk | ⚙ In progress | ⊗ Unhandled |
| Influence | Inclination | ● Supporter | ● Undecided | ● Challenger |

Note: Consider using Microsoft Visio to create a tool similar to this one.

# Remind – Your Job = decision based on principles

## Separation of concerns.

Divide your application into distinct features with as little overlap in functionality as possible. The important factor is minimization of interaction points to achieve high cohesion and low coupling. However, separating functionality at the wrong boundaries can result in high coupling and complexity between features even though the contained functionality within a feature does not significantly overlap.

## Single Responsibility principle.

Each component or module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionality.

## Principle of Least Knowledge

A component or object should not know about internal details of other components or objects.

## Don’t repeat yourself (DRY).

You should only need to specify intent in one place. For example, in terms of application design, specific functionality should be implemented in only one component; the functionality should not be duplicated in any other component.

## Minimize upfront design.

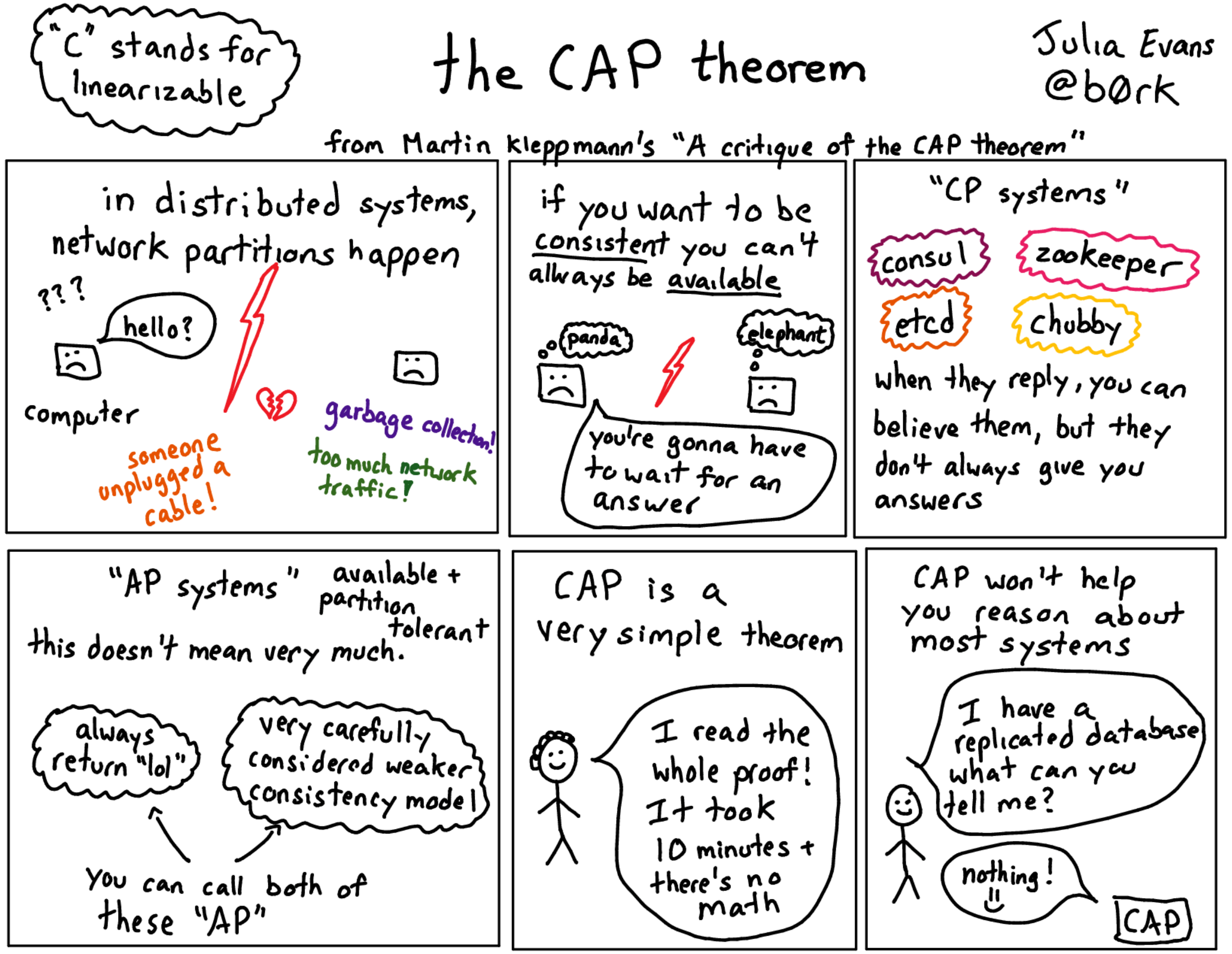
Only design what is necessary. In some cases, you may require upfront comprehensive design and testing if the cost of development or a failure in the design is very high. In other cases, especially for agile development, you can avoid big design upfront (BDUF). If your application requirements are unclear, or if there is a possibility of the design evolving over time, avoid making a large design effort prematurely. This principle is sometimes known as YAGNI ("You ain’t gonna need it").

# Focus # based on your company

- Determine the Application Type
- Determine the Deployment Strategy
- Determine the Appropriate Technologies
- Determine the Quality Attributes
- Determine the Crosscutting Concerns

# Cross-cutting ?

- Instrumentation and logging.
- Authentication.
- Authorization.
- Exception management.
- Communication.
- Caching.
- CI / CD
- Containerization
- Scale
- Tracing
- Resilience



"C" stands for linearizable
the CAP theorem
Julia Evans @b0rk
from Martin Kleppmann's "A critique of the CAP theorem"
in distributed systems, network partitions happen
???
hello?
computer
someone unplugged a cable!
garbage collection!
too much network traffic!
if you want to be consistent you can't always be available
panda
elephant
you're gonna have to wait for an answer
"CP systems"
consul
zookeeper
etcd
chubby
when they reply, you can believe them, but they don't always give you answers
"AP systems" available + partition tolerant
this doesn't mean very much.
always return "lol"
very carefully considered weaker consistency model
You can call both of these "AP"
CAP is a very simple theorem
I read the whole proof! It took 10 minutes + there's no math
CAP won't help you reason about most systems
I have a replicated database what can you tell me?
nothing!
CAP

# My Tips

## Performance

- SOC
- Tech Update
- The right stack for the right workload
- Wait is blocking ! = Synchro

## Availability

- SOC
- SPOF
- Everything fails all the times

Solution
Technology
Unit

# THANK YOU
# Visit me @
# Atekco.io