University of
South Australia

INFS 2044

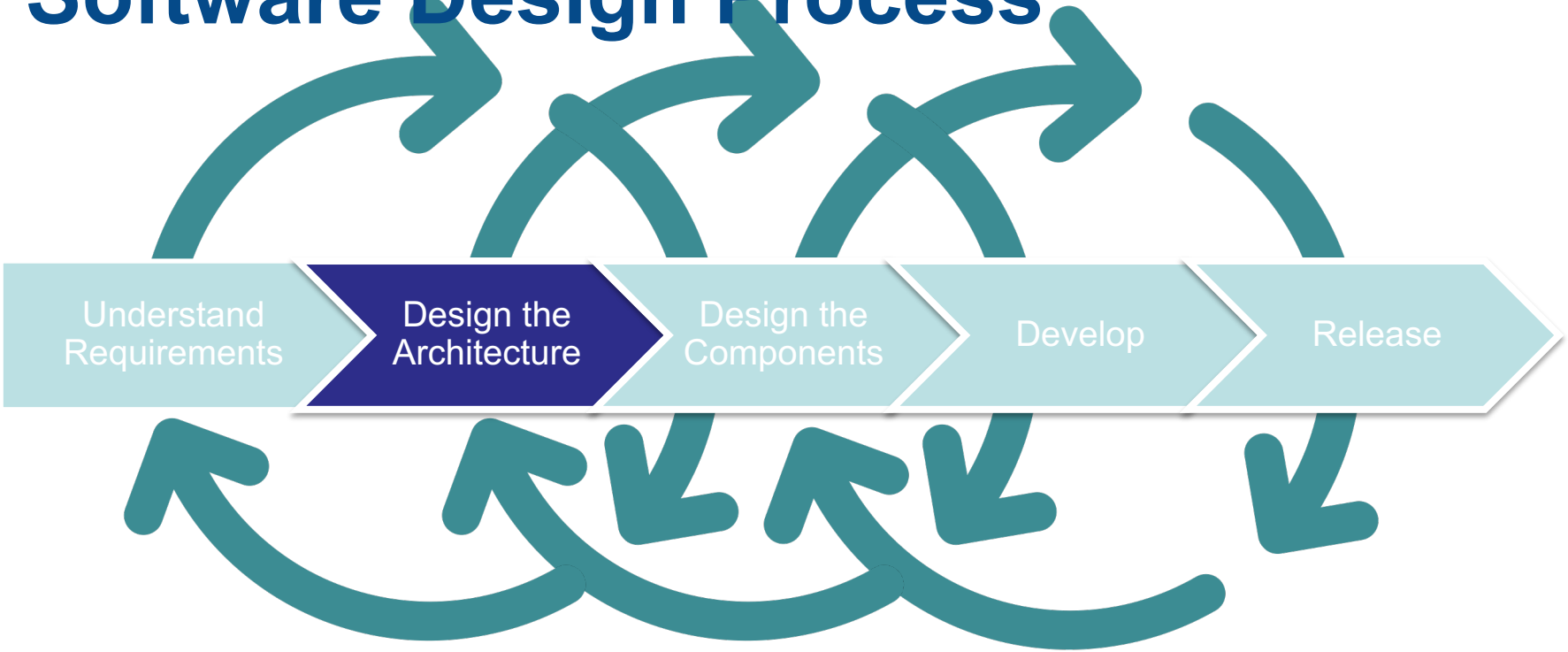
Week 2

Architecture and Decomposition

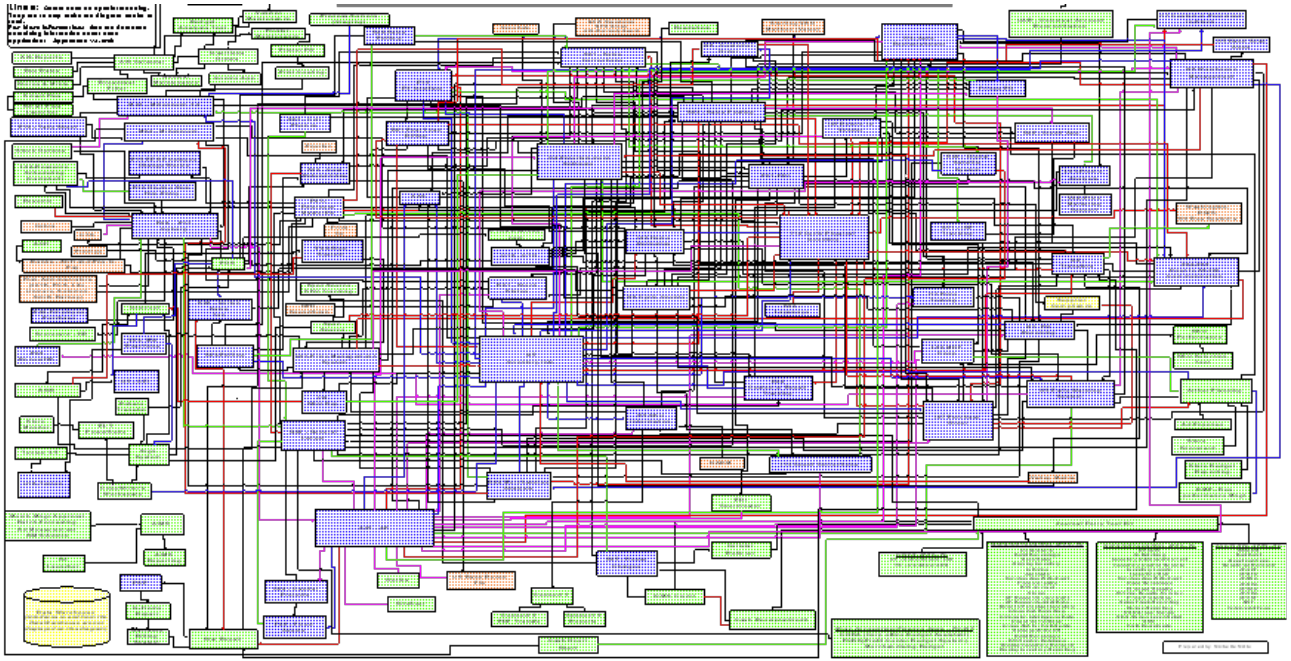
Learning Objectives

- Explain architectural concepts that influence system design (CO1, CO2)
- Understand component design principles (CO3)
- Communicate software architecture through diagrams (CO3, CO6)

Software Design Process



The Complexity Hairball



https://www.informatica.com/potential-at-work/whose-fault-is-the-integration-hairball-your-chance-to-be-a-hairball-hero.html

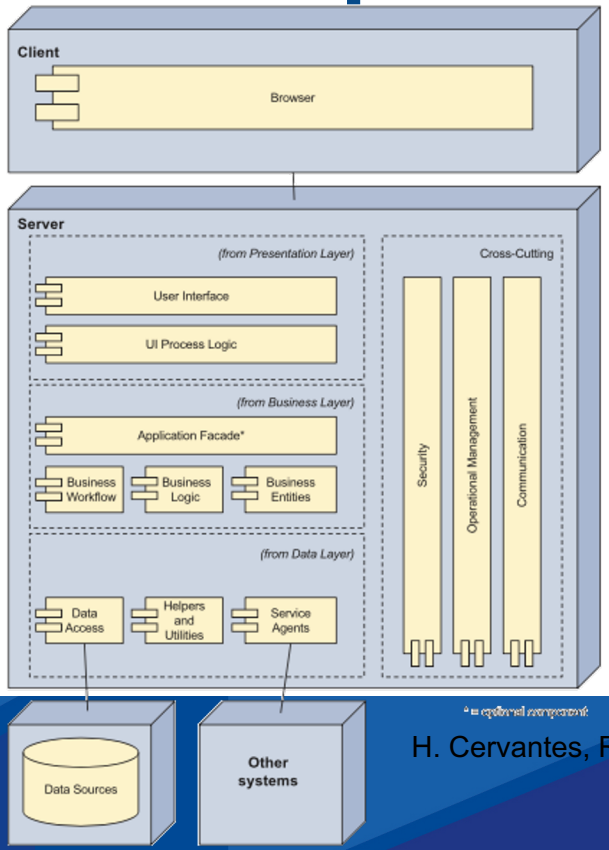
Remove Complexity

- Uncontrolled dependencies are bad
- Mitigate complexity by carefully designing components and their interactions
- Encapsulate changes

Software Architecture

- **Software architecture** is the design and structure of a software system
- **Component:** a building block of the software system with a well-defined interface and identified dependencies
- **Decomposition** is the act of identifying components

Architecture Example



H. Cervantes, R. Kazman (2016) Designing Software Architectures

Architectural Analysis

- Understanding the environment
- Determining the requirements for the system based on
  - What the system will do (the functional requirements)
  - Non-functional requirements
  - Development-time non-functional requirements
  - Business requirements & environment

Why? – De-risk

- Reduce the risk of missing something centrally important
- Identify variation points and probable evolution points
- Avoid applying excessive effort to low priority issues

What? – Considerations

- **Application Type**
  - Desktop, mobile, web app, service app, …
- **Deployment Strategy**
  - Physical placement of components, communication channels, …
- **Technologies**
  - Policies, infrastructure limitations, resources and skills
- **Quality Attributes**
  - Security, performance, availability, and usability
- **Crosscutting Concerns**
  - Monitoring & logging, authentication & authorization, exception management, communication protocols, …
- Cost, License, Support, Maturity, Integration, Size, Learning Curve, Popularity, …

When? – Early

- Architectural issues need to be identified and resolved in early development work
- Architectural thinking is required to estimate project cost, duration, schedule, and quality
- Delaying decisions may can incur very high cost
- The major design decisions are made early on
  - Decisions that are of lesser importance can be deferred until later

How? – Software Architecture Design

1. Find core use cases
2. Create a design to satisfy the core use cases
    a. Define components
    b. Define their interactions
3. Validate the architecture
4. Design and implement each component

Design to Requirements?

- “FR01. The system shall manage financial trades.”
- “NFR02. The system shall be maintainable.”

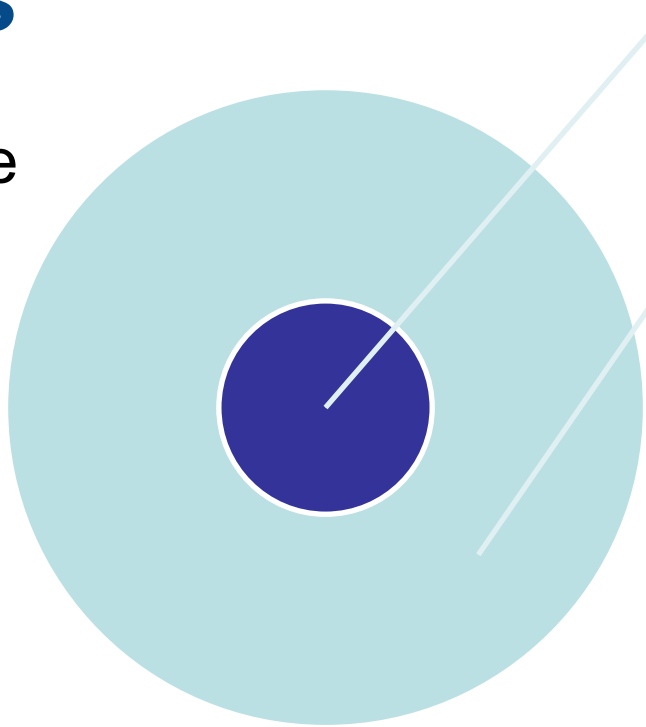
We must capture the **behaviour** and potential **changes**

Use Cases

- UC1 Feed the Family (workdays): Parent gets home after work, takes ready meal from freezer, cooks it in the microwave, meanwhile others arrive, all eat dinner together.
- UC2 Feed the Family (weekend): Parent prepares ingredient, cooks them on the stove, then all eat dinner together.
- …

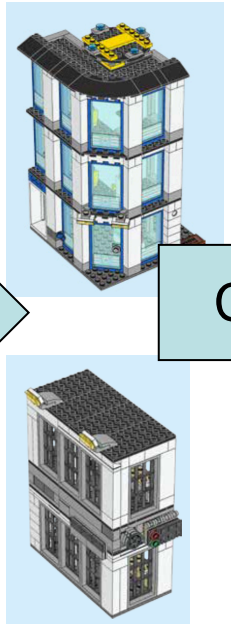
Core Use Cases

- Represent the essence of the business of the system
- Stable

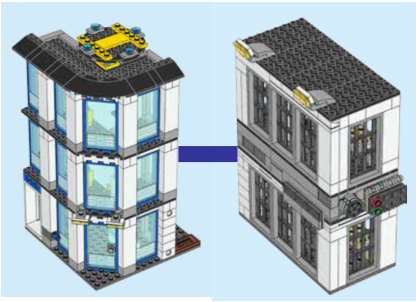


Architecture Approach

Decomposition



Composition



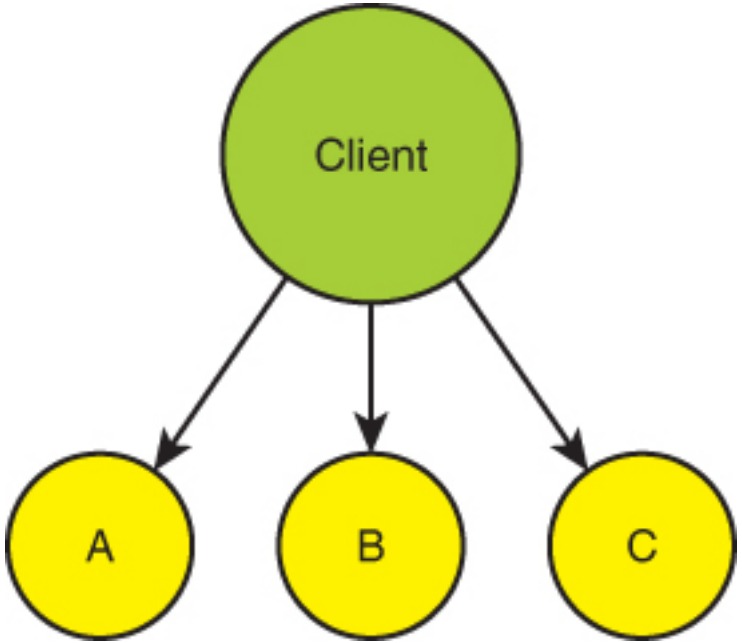
Decomposition

- Functional decomposition
- Domain decomposition
- **Volatility-based decomposition**

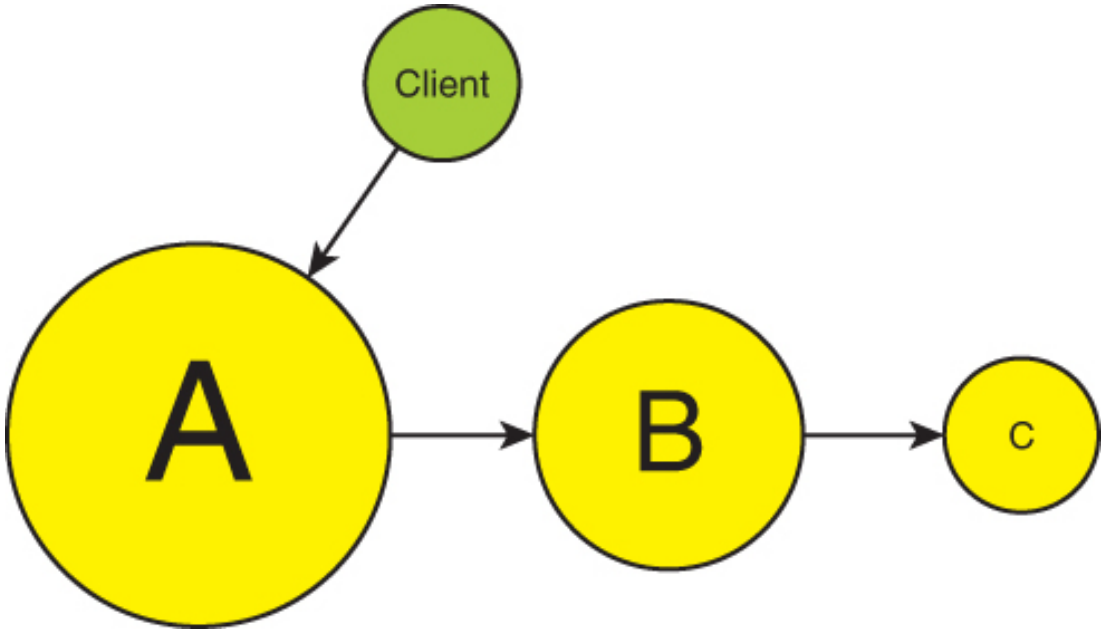
https://www.irishjobs.ie/careeradvice/wp-content/uploads/Problem-solving-skill.png

Functional Decomposition House

Complexity: Client Bloat



Complexity: Service Bloat & Coupling



Impact of Changes in Func. Decomp.

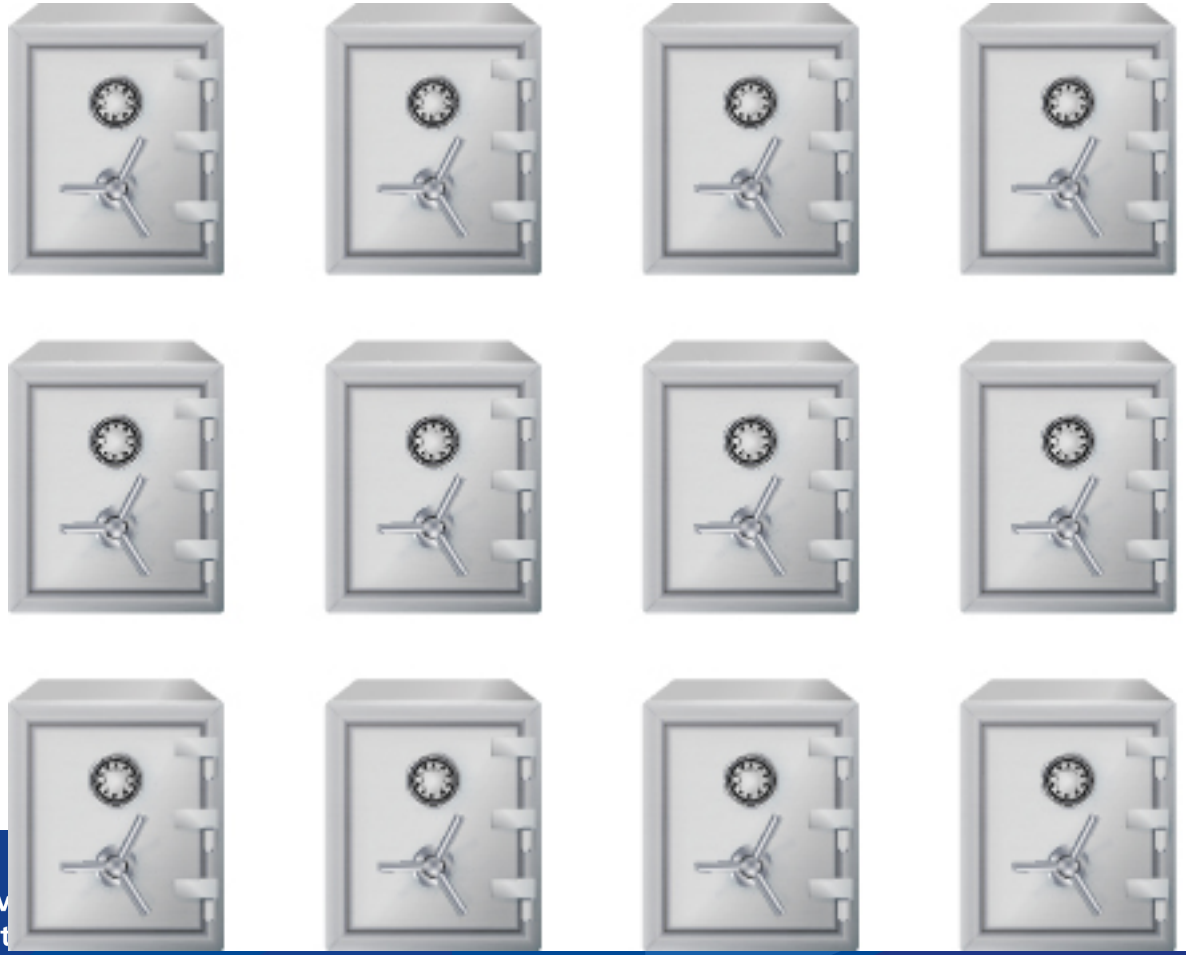
- Good for understanding requirements
- Poor for software design
- Change affects multiple components

https://i.ebayimg.com/images/g/rE4AAOSw-0xYgAyO/s-l500.jpg

Volatility-based Decomposition

1. Identify areas of potential change and encapsulate those into services or system building blocks.

2. Implement the required behaviour as the interaction between the encapsulated areas of volatility.



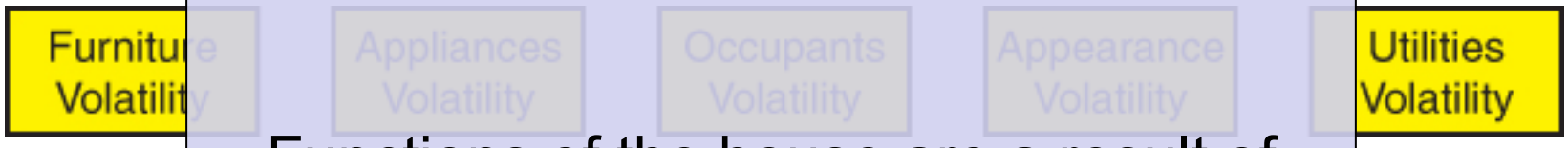
Volatile or Variable?

- Not everything that is variable is also volatile.

- **Volatile change**
  - Open-ended
  - Expensive to contain (unless encapsulated)
- **Variable change**
  - Controlled
  - Handled by logic in the code

Volatile House

- Same house over time



- Across houses at the same time

Functions of the house are a result of **integration** among the components

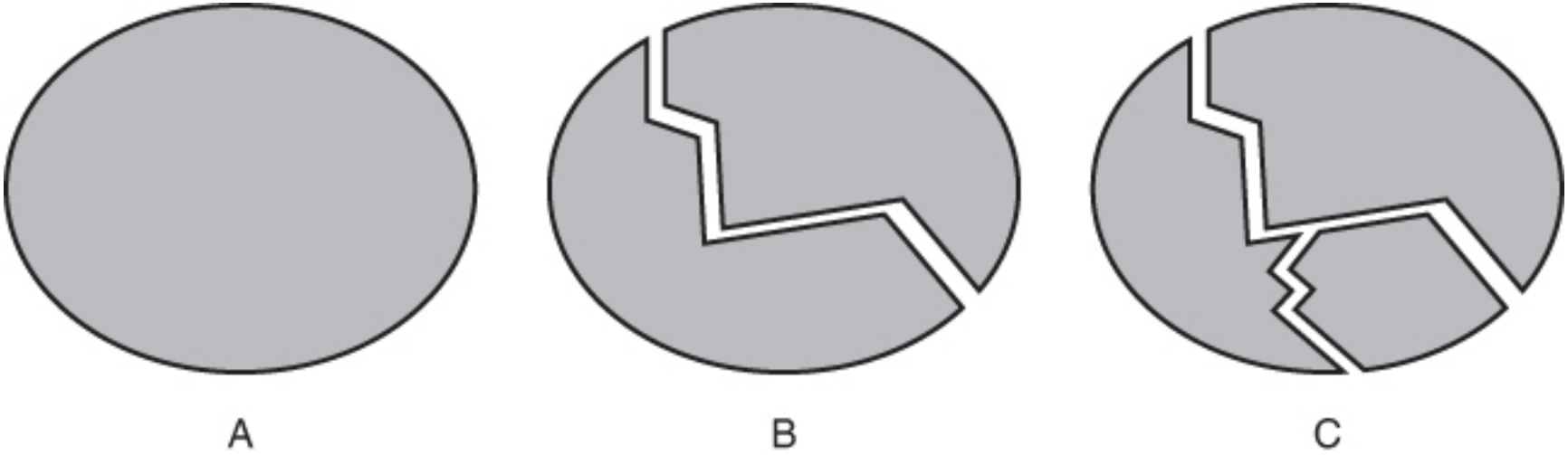
Volatility Lists

- User
- Client application
- Security
- Notification
- Storage
- Connection & Synchronisation
- Duration and device
- Workflow
- Locale
- Regulations
- …

Identifying Volatilities

- Discussion with clients
- Speculative Design
- Design for competitors

Design Factoring



Defining Components

- Map volatile areas to components
  - Usually not one-to-one
  - A component can encapsulate multiple volatilities
  - Some volatilities map to operational concepts
  - Some volatilities map to third-party services
- Start with the simple and easy decisions

Composable Design

- Identify the smallest set of components that you can put together to satisfy all the core use cases
  - the regular use cases simply represent a different interaction between the components, not a different decomposition
- When the requirements change, the decomposition does not change!

Component Design Principles

- Common Closure Principle (CCP)
- Common Reuse Principle (CRP)

Common Closure Principle (CCP)

- A component should not have multiple reasons to change.
  - gather together in one place all the classes that are likely to change for the same reasons
  - a change in requirements will likely affect few components

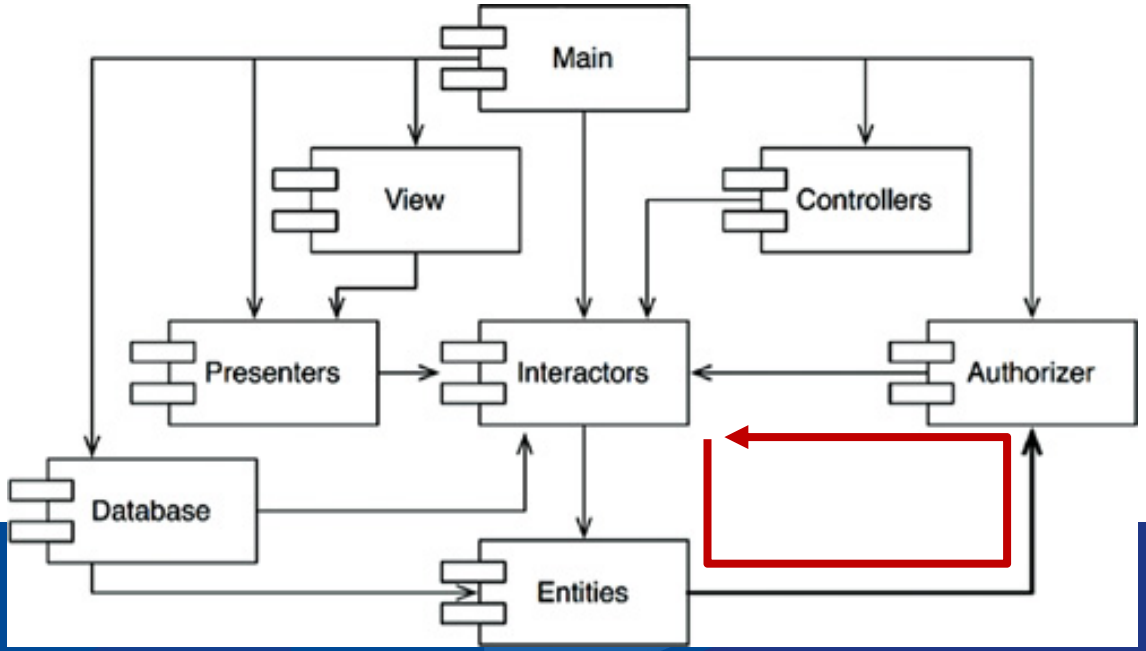
Common Reuse Principle (CRP)

- Code and services that tend to be reused together belong in the same component.
  - Reuse introduces a dependencies
  - Code that is not tightly bound to each other should not be in the same component

- "*Don't depend on things you don't need*."

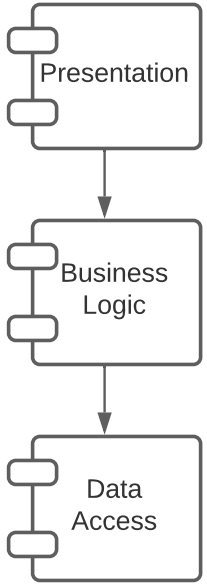
Acyclic Dependency Principle

- *Allow no cycles in the component dependency graph.*

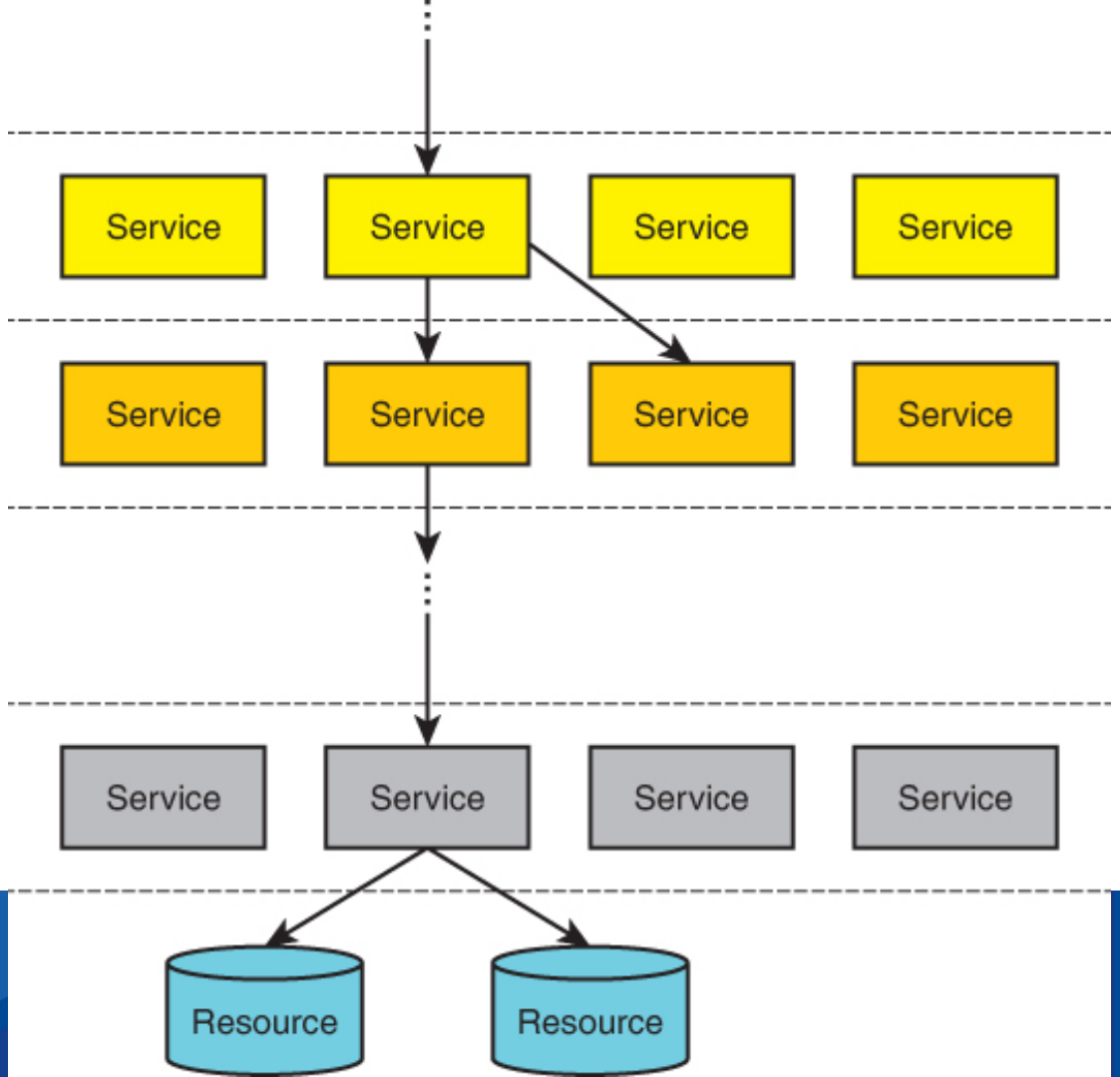


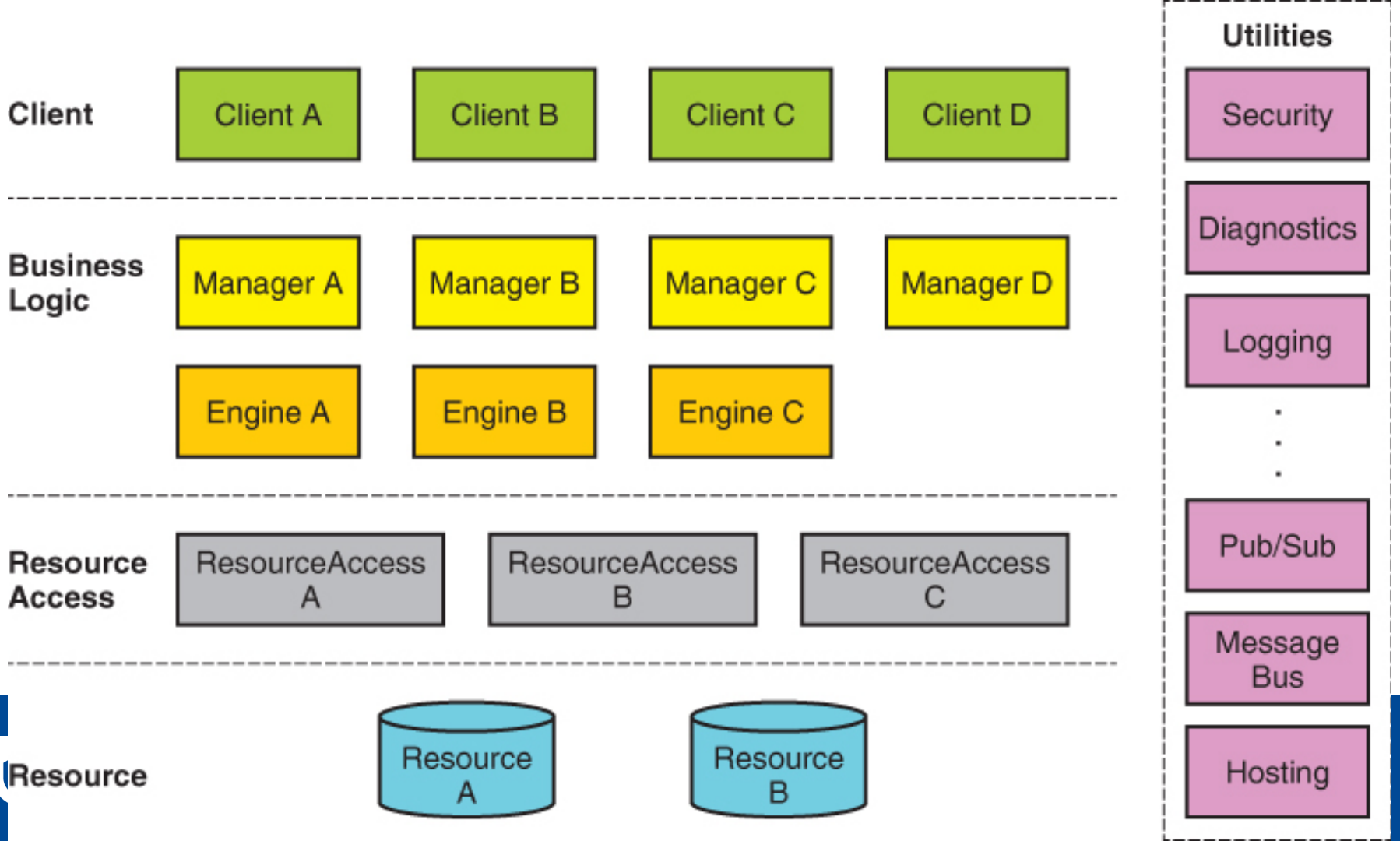
Stable Dependency Principle

- *Depend in the direction of stability.*
  - Some of these components are *designed* to be volatile. We *expect* them to change.
  - Volatile components should not be depended on by a component that is difficult to change



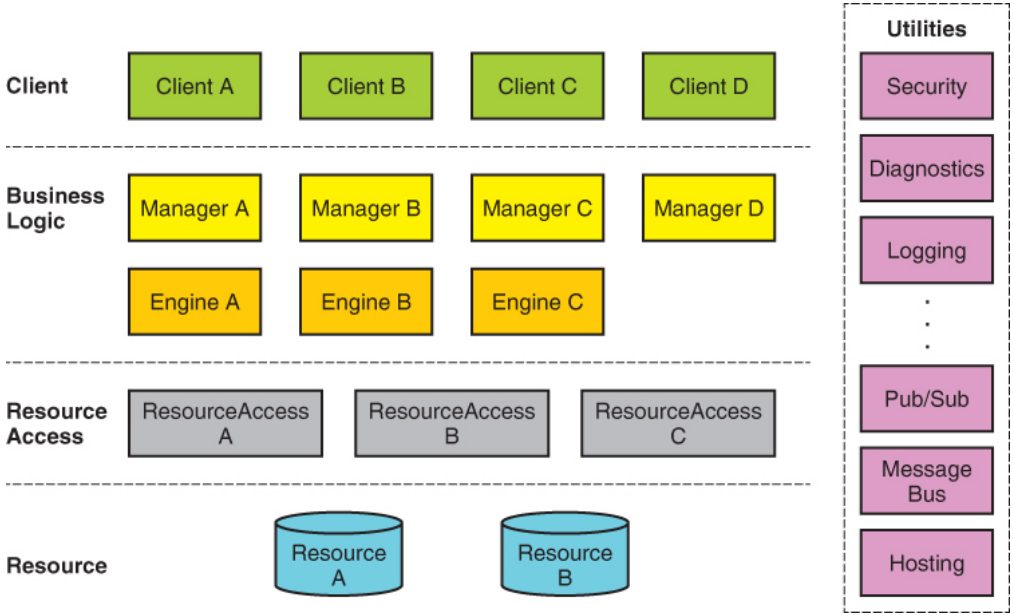
Layers

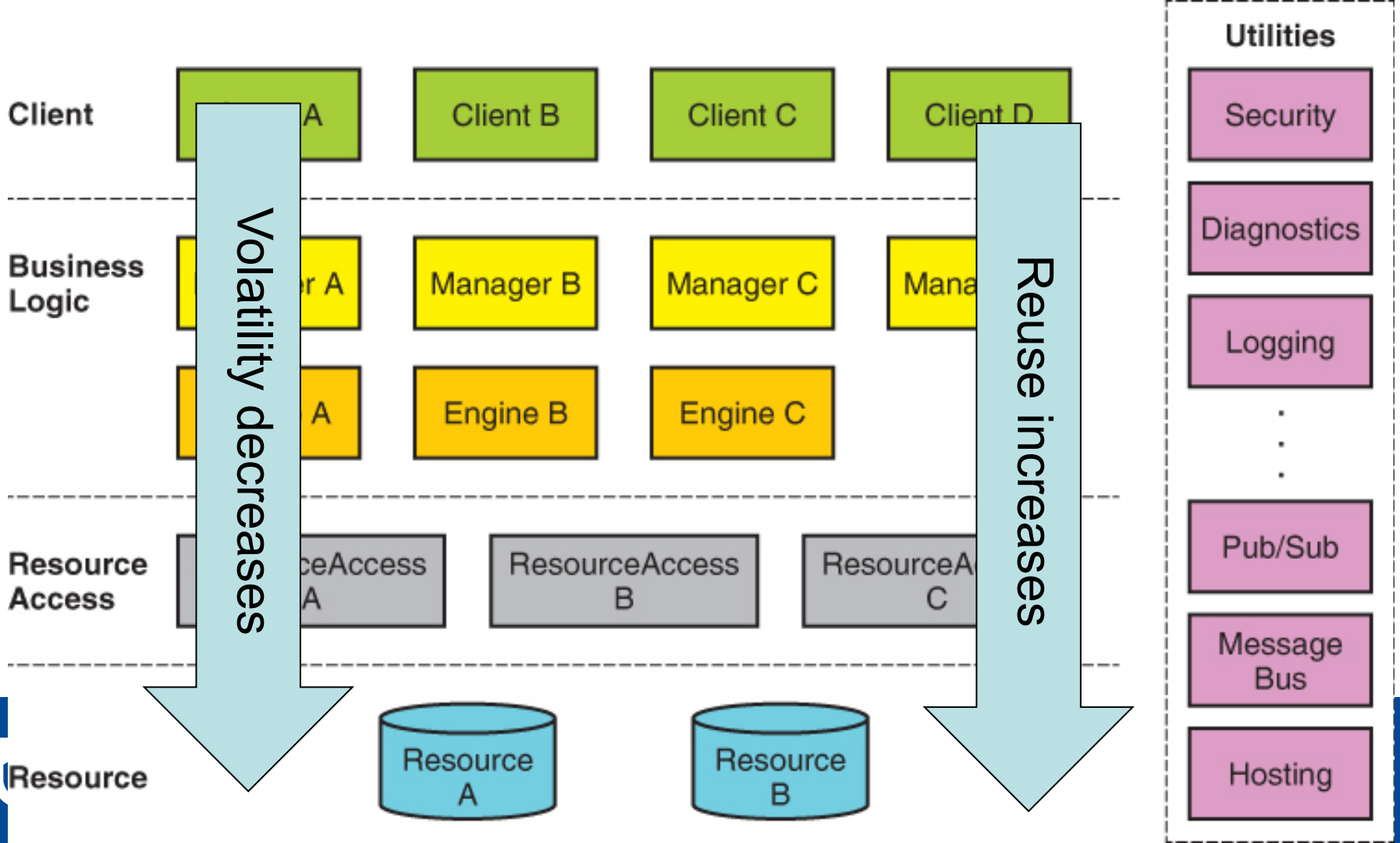

Service
Service
Service
Service
Service
Service
Service
Service
Service
Service
Service
Service
Resource
Resource


Client
Client A
Client B
Client C
Client D
Business Logic
Manager A
Manager B
Manager C
Manager D
Engine A
Engine B
Engine C
Resource Access
ResourceAccess A
ResourceAccess B
ResourceAccess C
Resource
Resource A
Resource B
Utilities
Security
Diagnostics
Logging
Pub/Sub
Message Bus
Hosting

Types of Components

- Managers
- Engines
- Resource Access
- Utilities

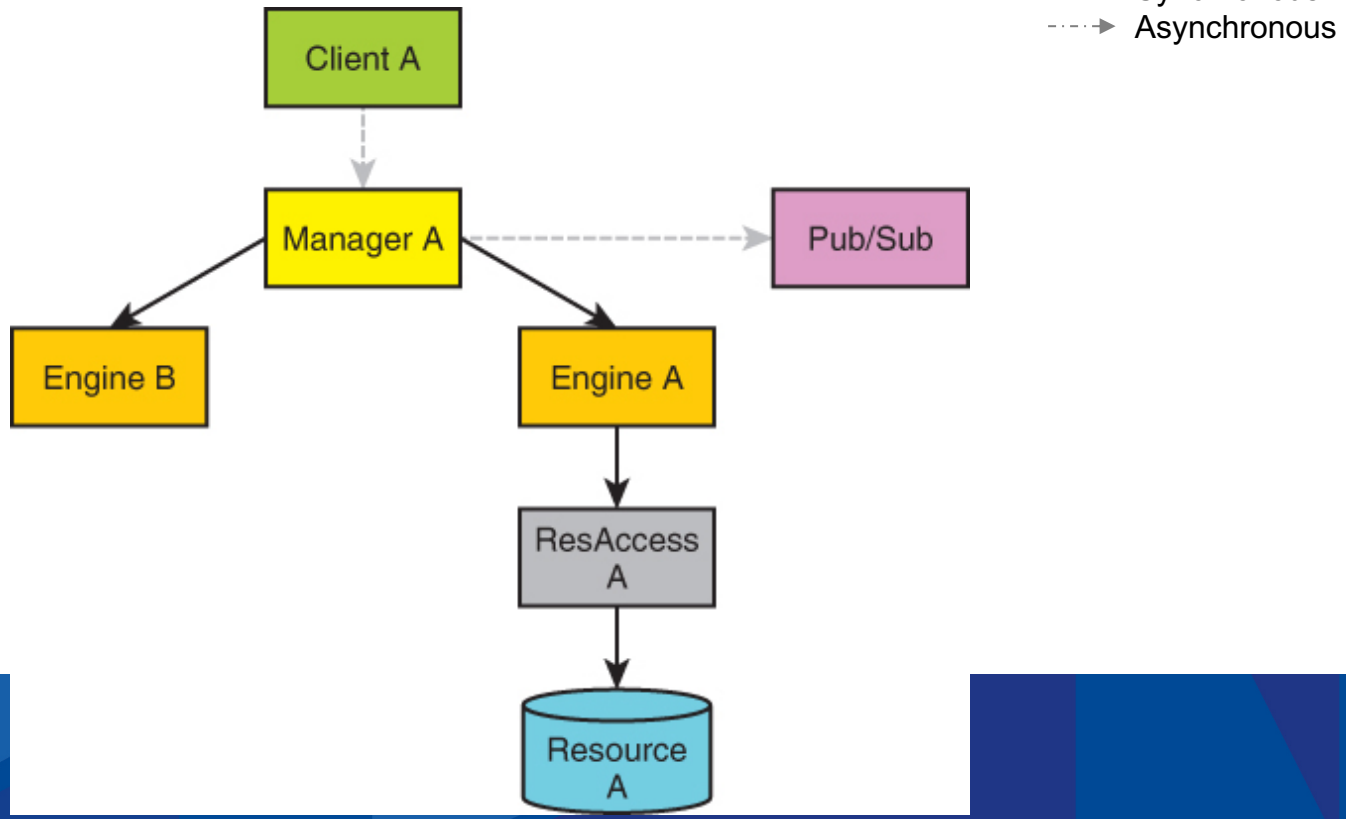



Client
Client B
Client C
Client D
Business Logic
Manager B
Manager C
Engine B
Engine C
Resource Access
ResourceAccess B
Resource
Resource A
Resource B
Volatility decreases
Reuse increases
Utilities
Security
Diagnostics
Logging
Pub/Sub
Message Bus
Hosting

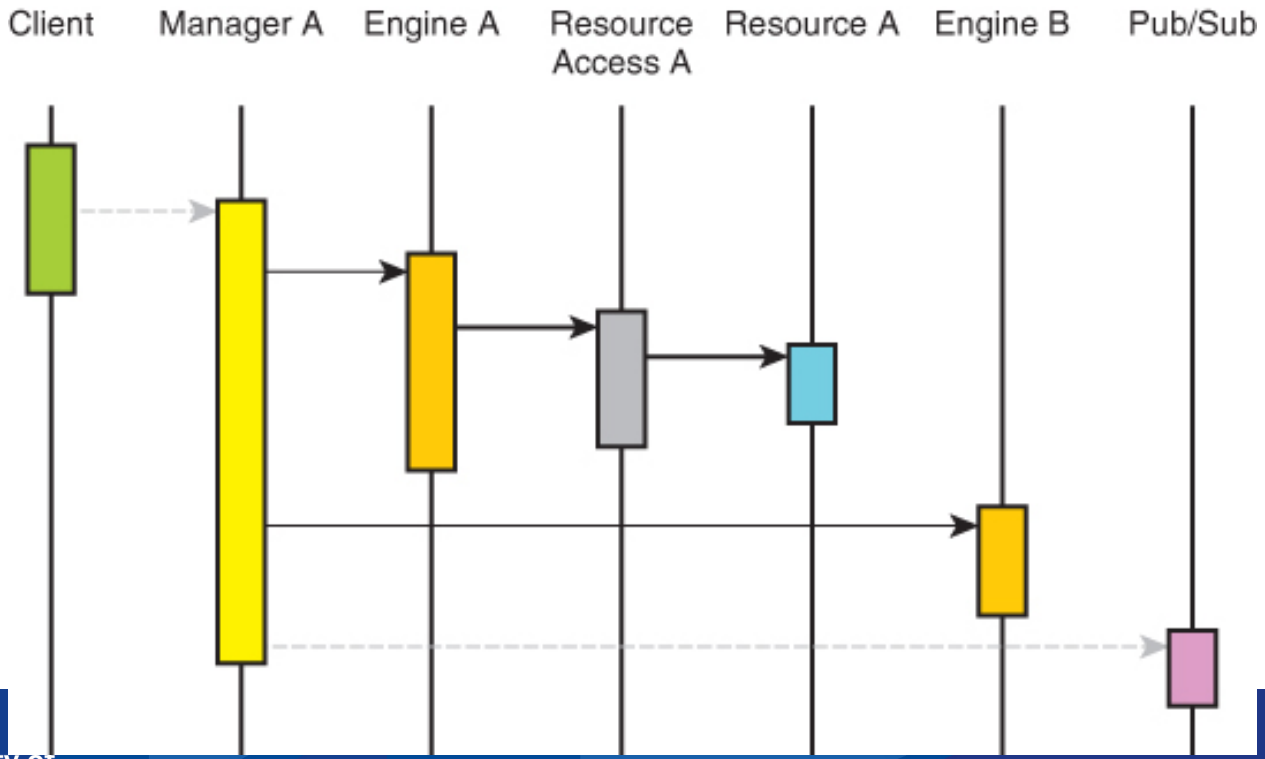
Software Architecture Design

1. Find core use cases
2. Create a design to satisfy the core use cases
    a. Define components
    b. Define their interactions
3. **Validate the architecture**
4. Design and implement each component

Communication Diagram



Sequence Diagram (informal)



Summary

- Architecture helps mitigate complexity
- Architecture Design is driven by volatility
- Design Principles capture desirable properties
- Layered architecture example
- Architecture validation on use cases

Activities this Week

- Read required readings
- Participate in Workshop 2
- Complete Quiz 2

University of
South Australia