# CS 575: Software Design

*Introduction to Software Architecture*

# Software Architecture and Design Topics

- Why pay attention to software design
- Rendering architecture and design
- Software architecture definitions and context
- Why do we design & principles of good design
- Wicked Problems
- Architecture Styles and Patterns
- Refactoring
- Design properties

# Why Software Design

**The probability of creating a successful(non-trivial) software system in the absence of design is unlikely**

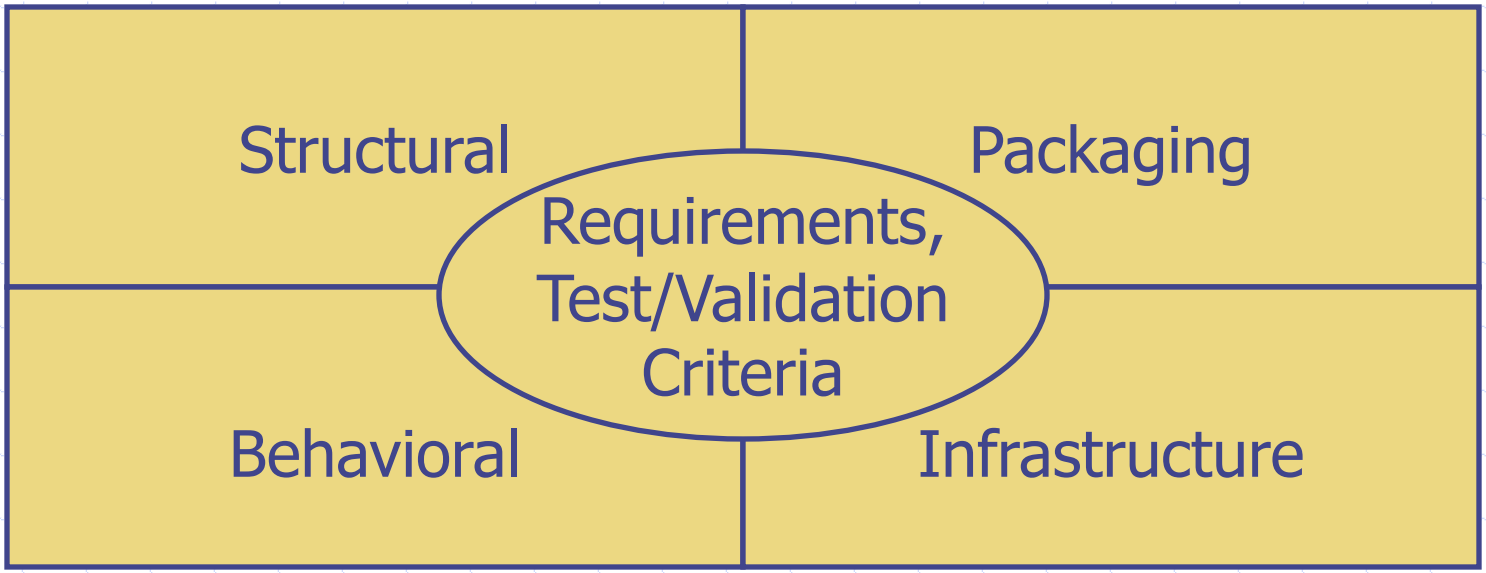
- Technical Challenges
- Non-Functional Challenges
- Sponsor/Constituent Constraint Challenges

# Software Architecture/Design Use Cases

- **Greenfield**: Guiding Net-New efforts
- **Brownfield:** Extending / enhancing existing systems
- **Legacy:** Understanding constraints of existing systems on newer brown- or green-field efforts

# Software Design

**A software design is a precise description of a system, using a variety of different perspectives…**



# Expressing A Software Design

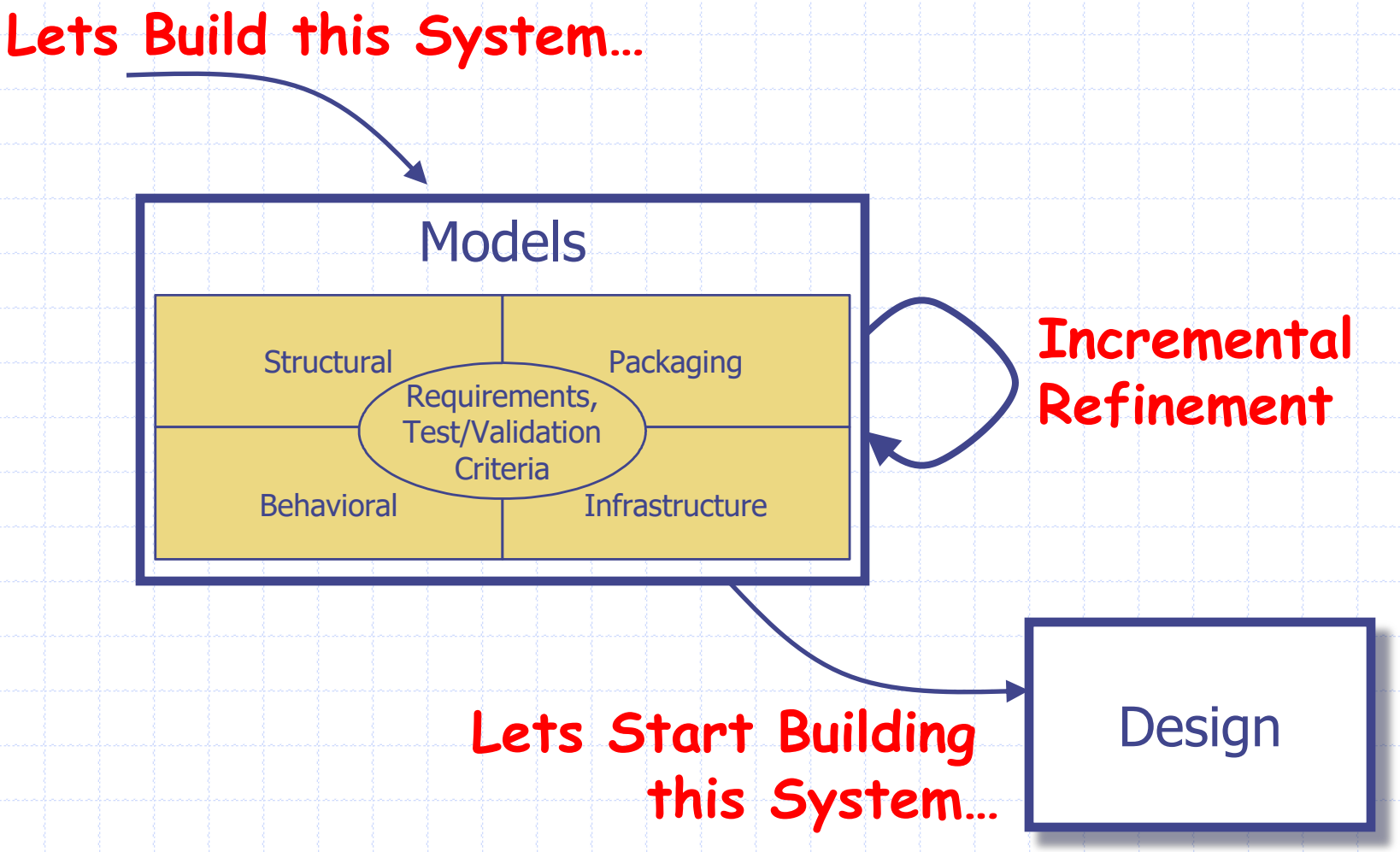
**Software Designs are complicated, therefore, they must be modeled…**

- Similar to an architects blueprint
- A model is an abstraction of the underlying problem
- Designs should be modeled, and expressed as a series of views
- Helpful to use models – UML, line/box, C4

# Modeling as a Design Technique

- Designs are too complicated to develop from scratch
- Good designs tend to be build using models…
  - 1) Abstract different views of the system
  - 2) Build models using precise notations (e.g., UML)
  - 3) Verify that the models satisfy the requirements
  - 4) Gradually add details to transform the models into the design
- And such models can be derived from proven architecture patterns

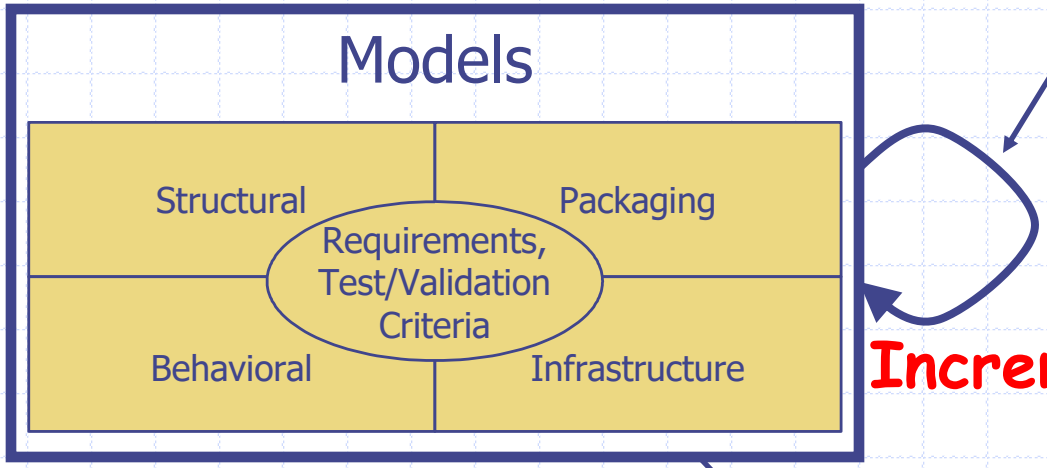
# Modeling as a Design Technique



# Modeling as a Design Technique – Reality is we rarely start from scratch these days

Lets Build this System…

Frameworks,
Patterns,
Templates, etc.

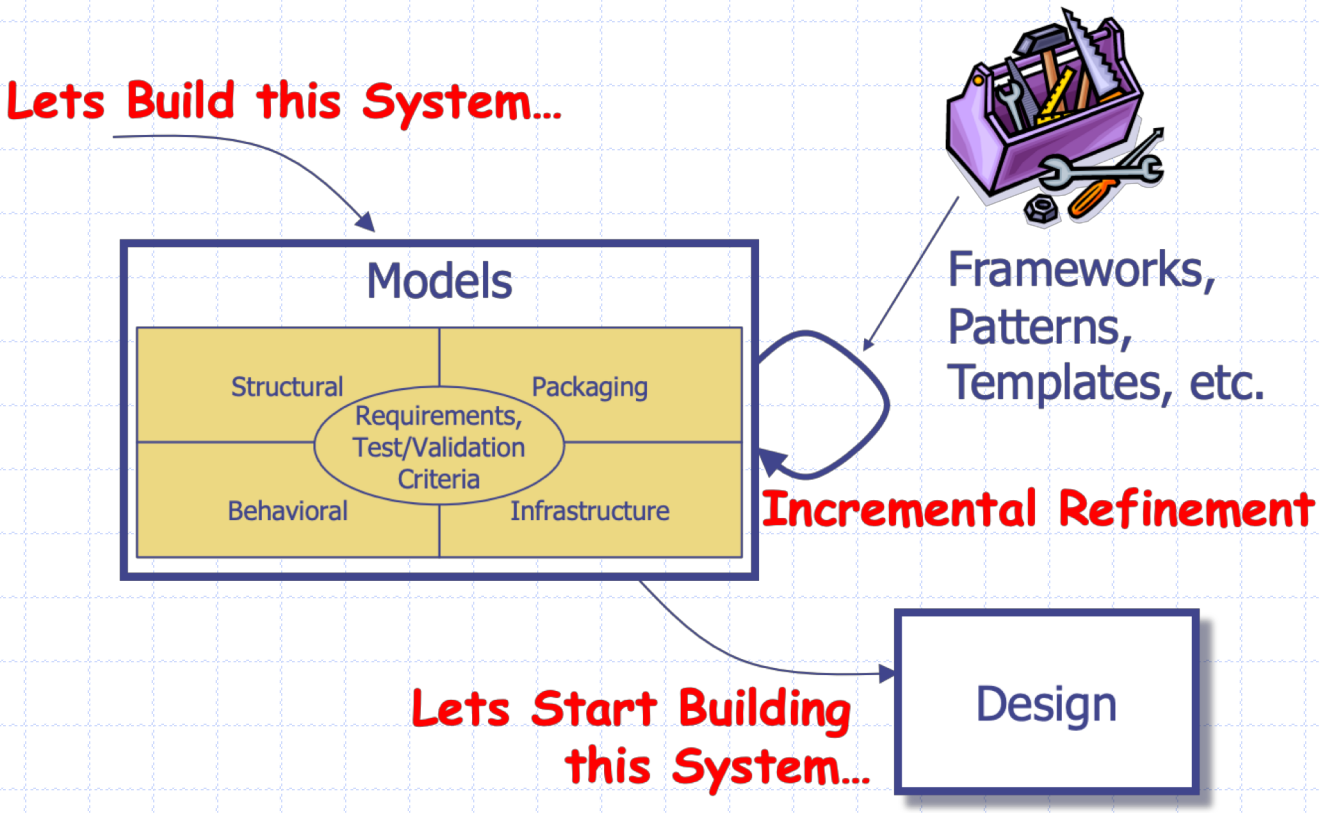



Incremental Refinement

Lets Start Building this System…

Design

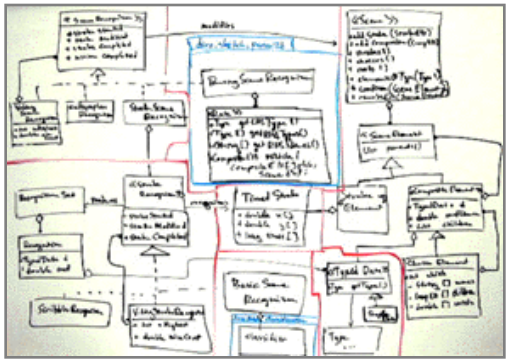
# Modeling as a Design Technique – Reality is we rarely start from scratch these days




BUT, one principal that we will explore is to defer as many design decisions to the last possible moment – avoid big "upfront" design

# Modeling Designs (UML)…

## Design



**Very Complicated To Understand**

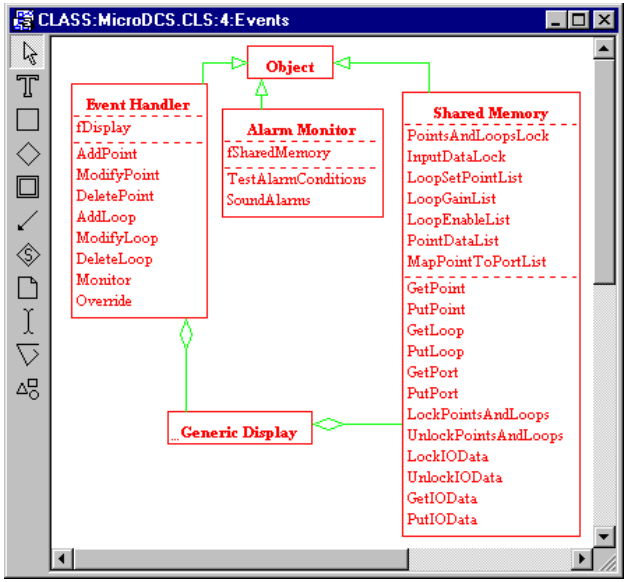
## Designing With Models

Model Repository

ENTERPRISE ARCHITECT

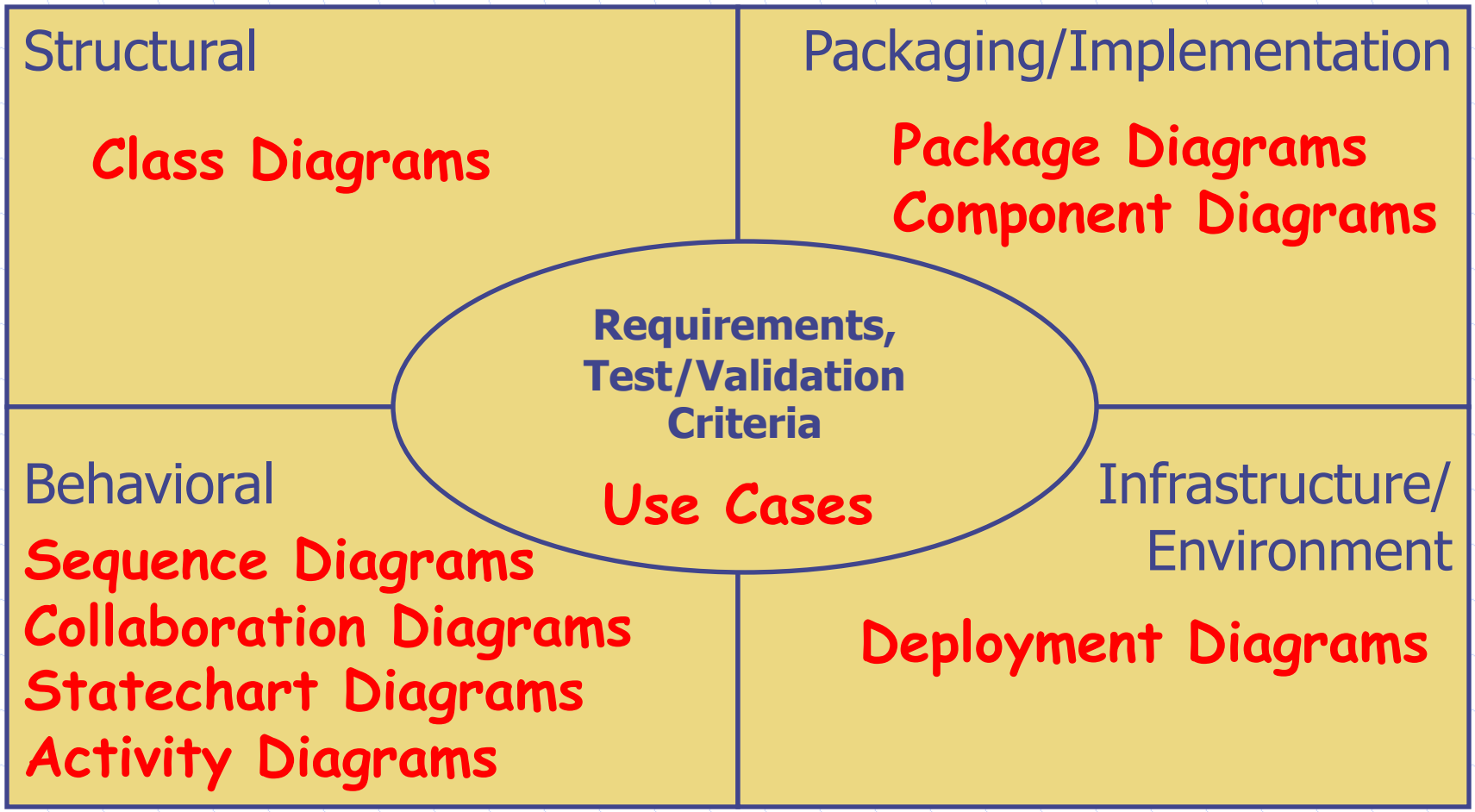
SPARX SYSTEMS

Modeling Tool



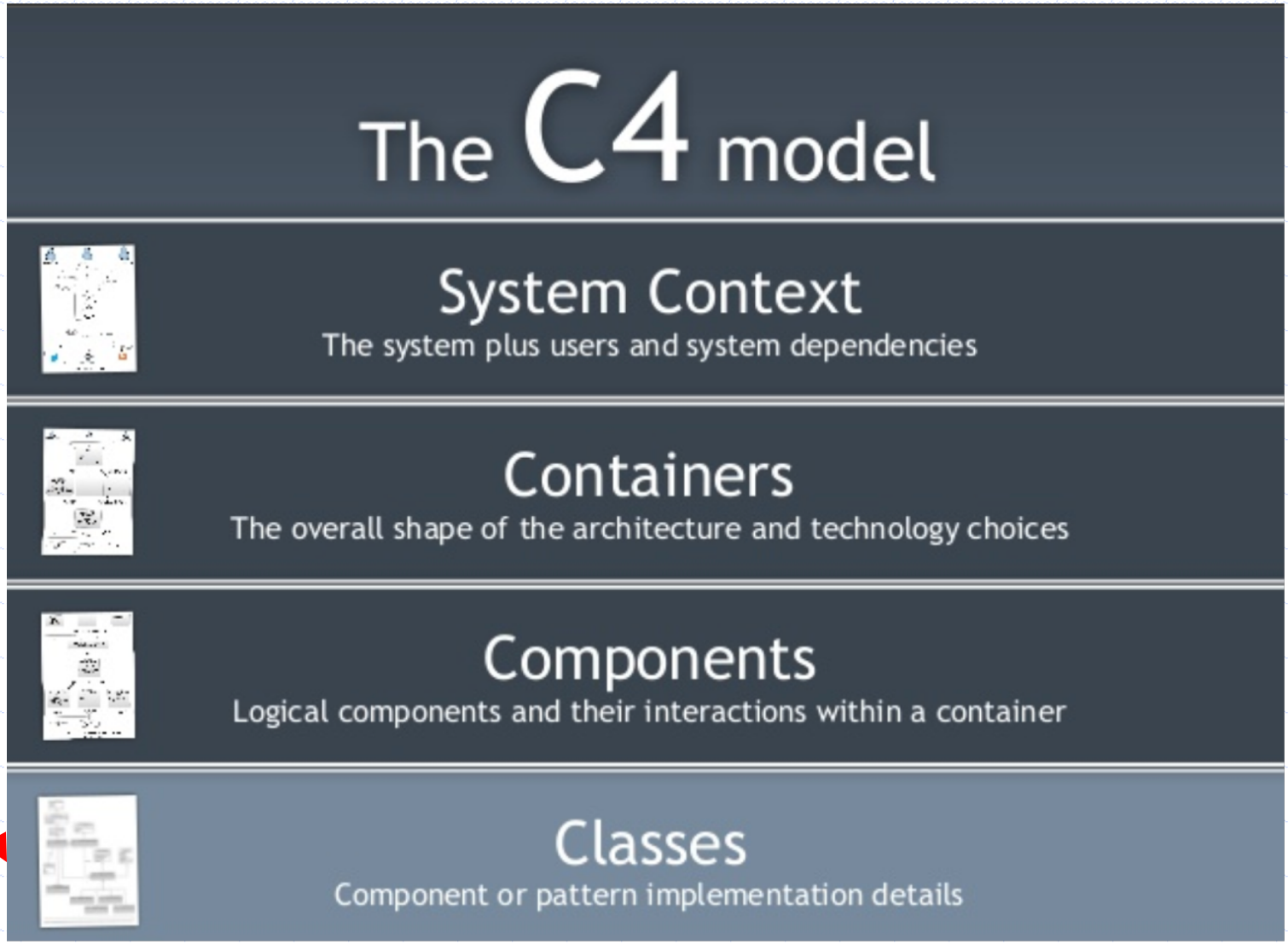

Visualization

# UML – A modeling Notation for Design

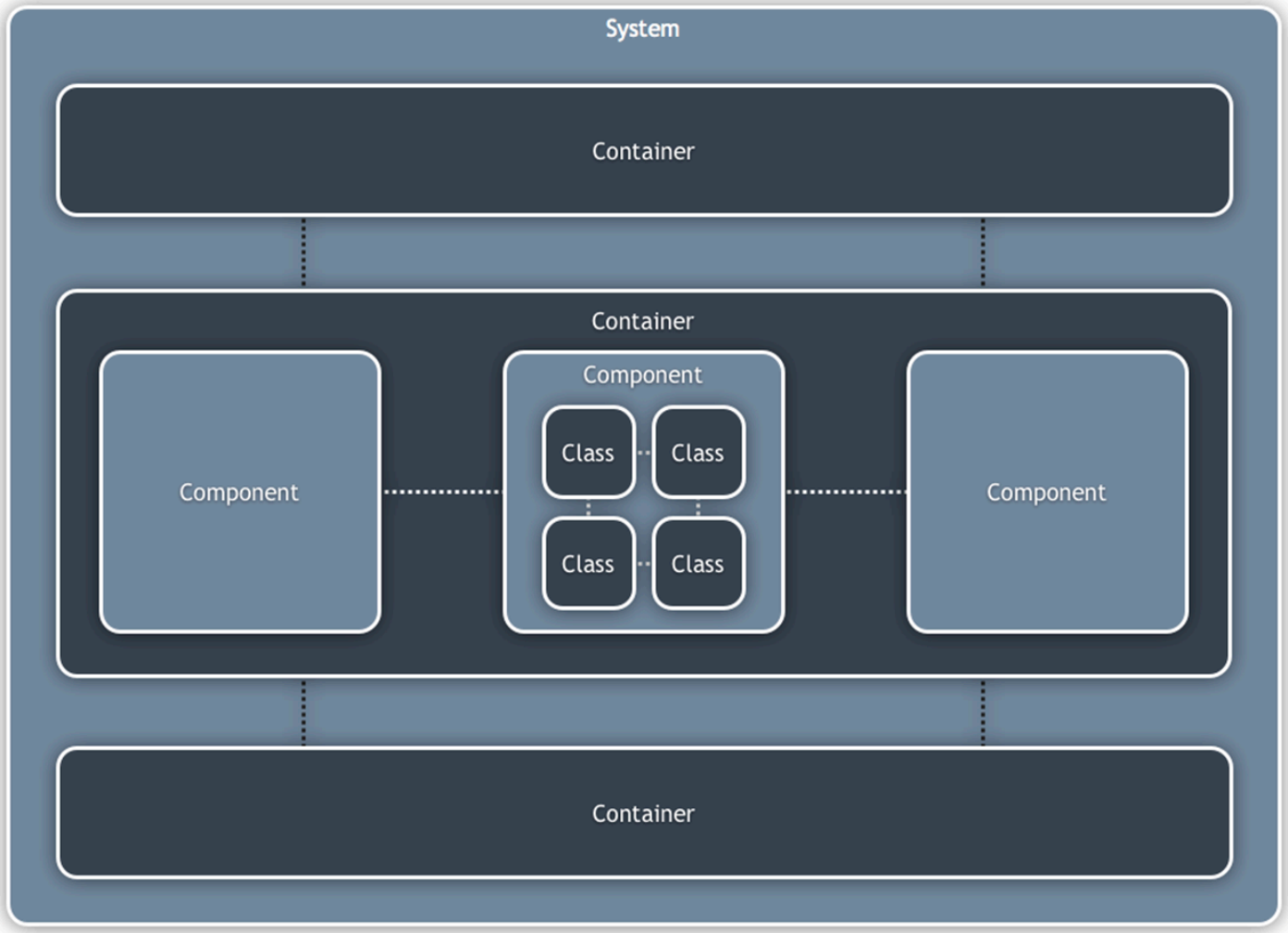


# Alternatives to Model Designs…

This is optional



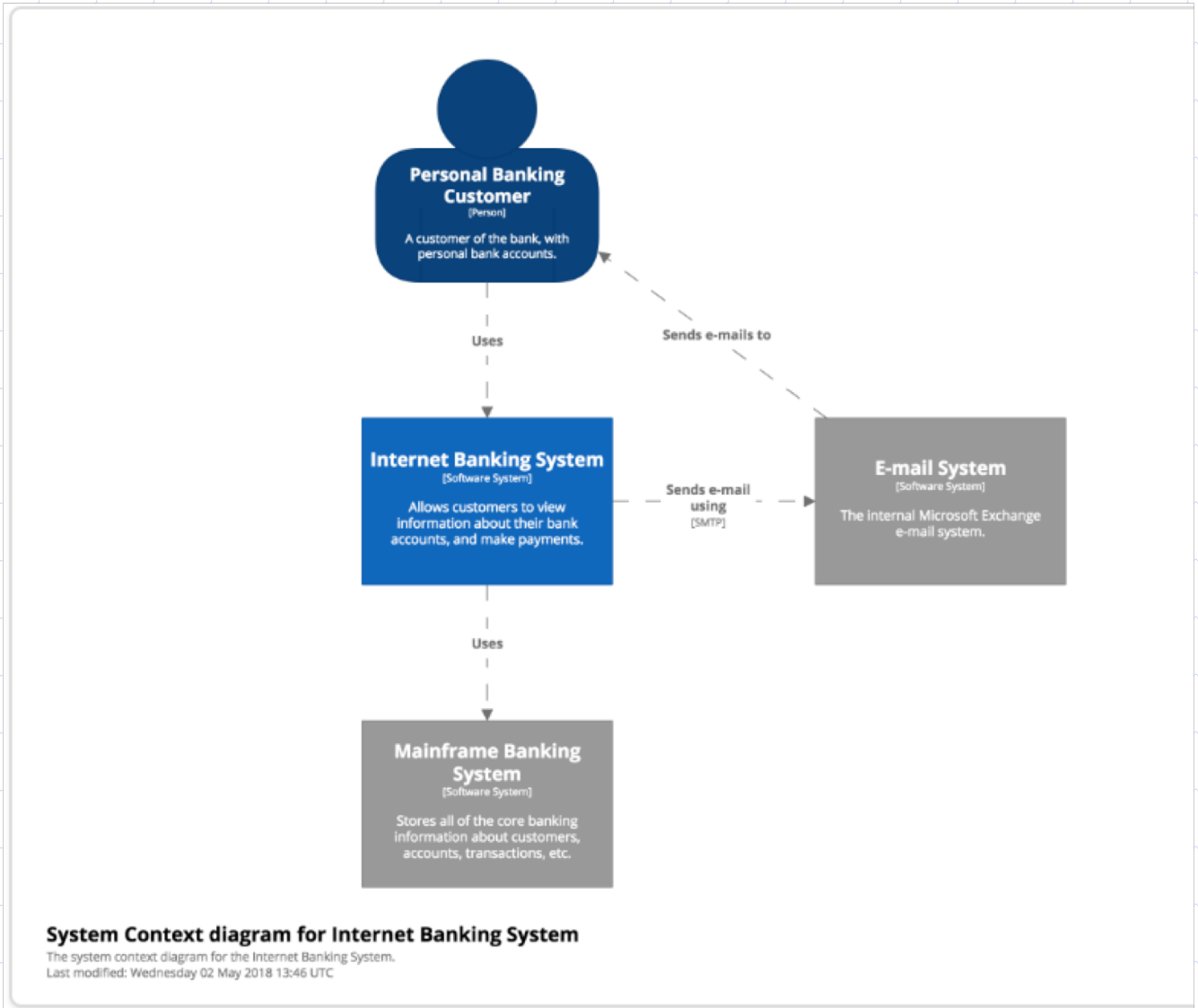
# The key attributes in C4 are related to each other



# C4 Examples

## Context

**Ref: Simon Brown**
**https://c4model.com/**



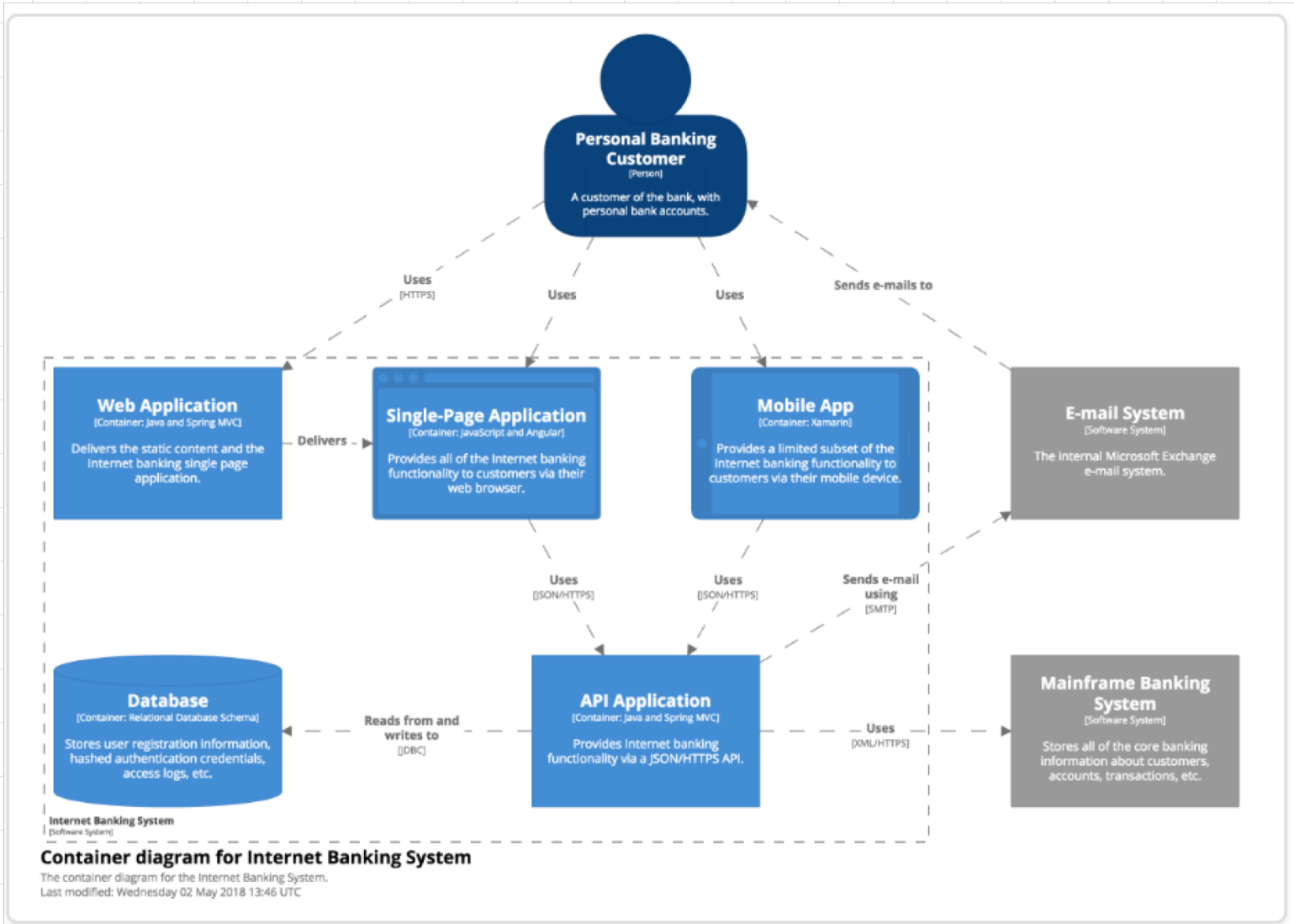

**System Context diagram for Internet Banking System**

The system context diagram for the Internet Banking System.
Last modified: Wednesday 02 May 2018 13:46 UTC

# C4 Examples

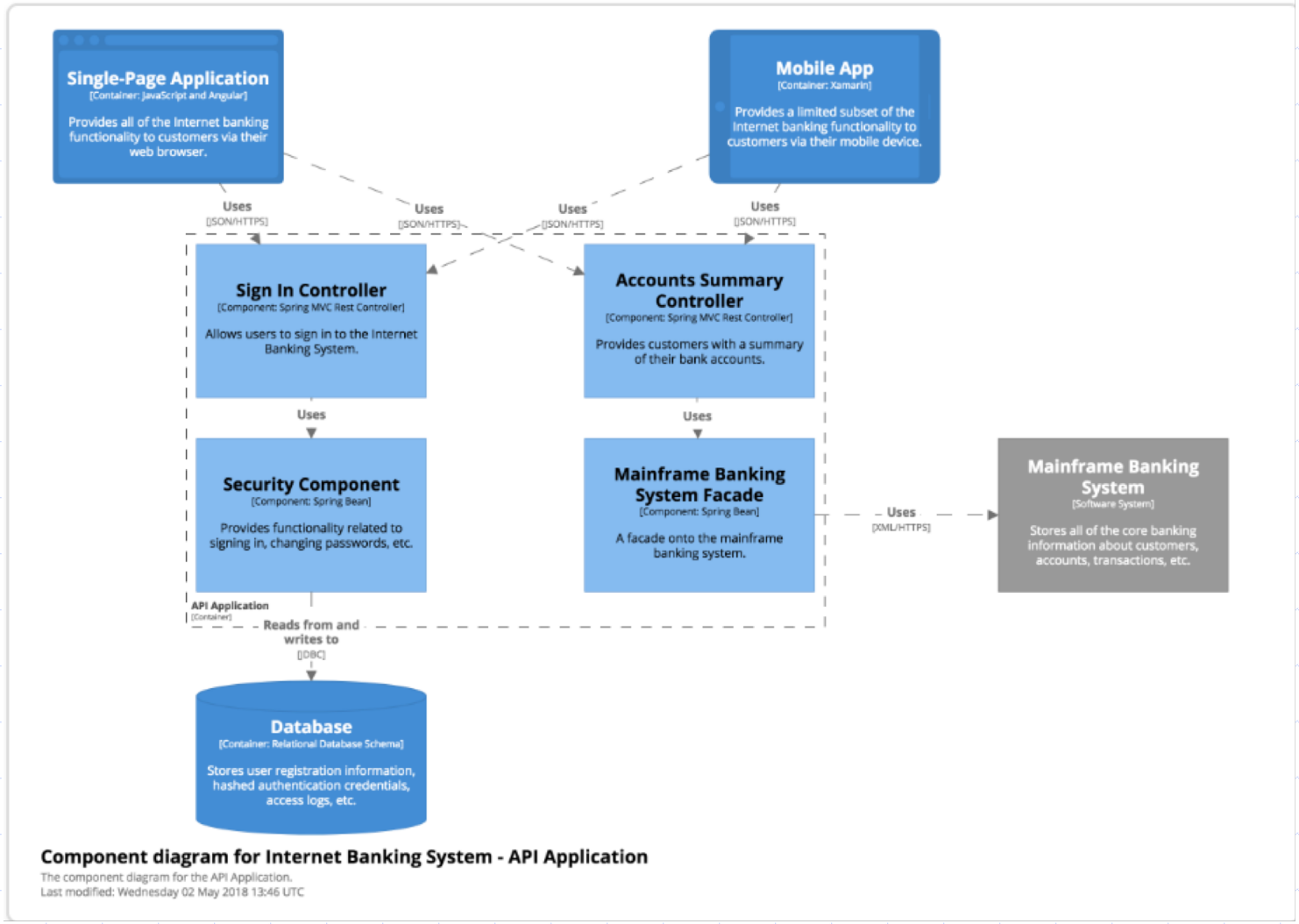
Ref: Simon Brown
https://c4model.com/

## Container



# C4 Examples
## Components

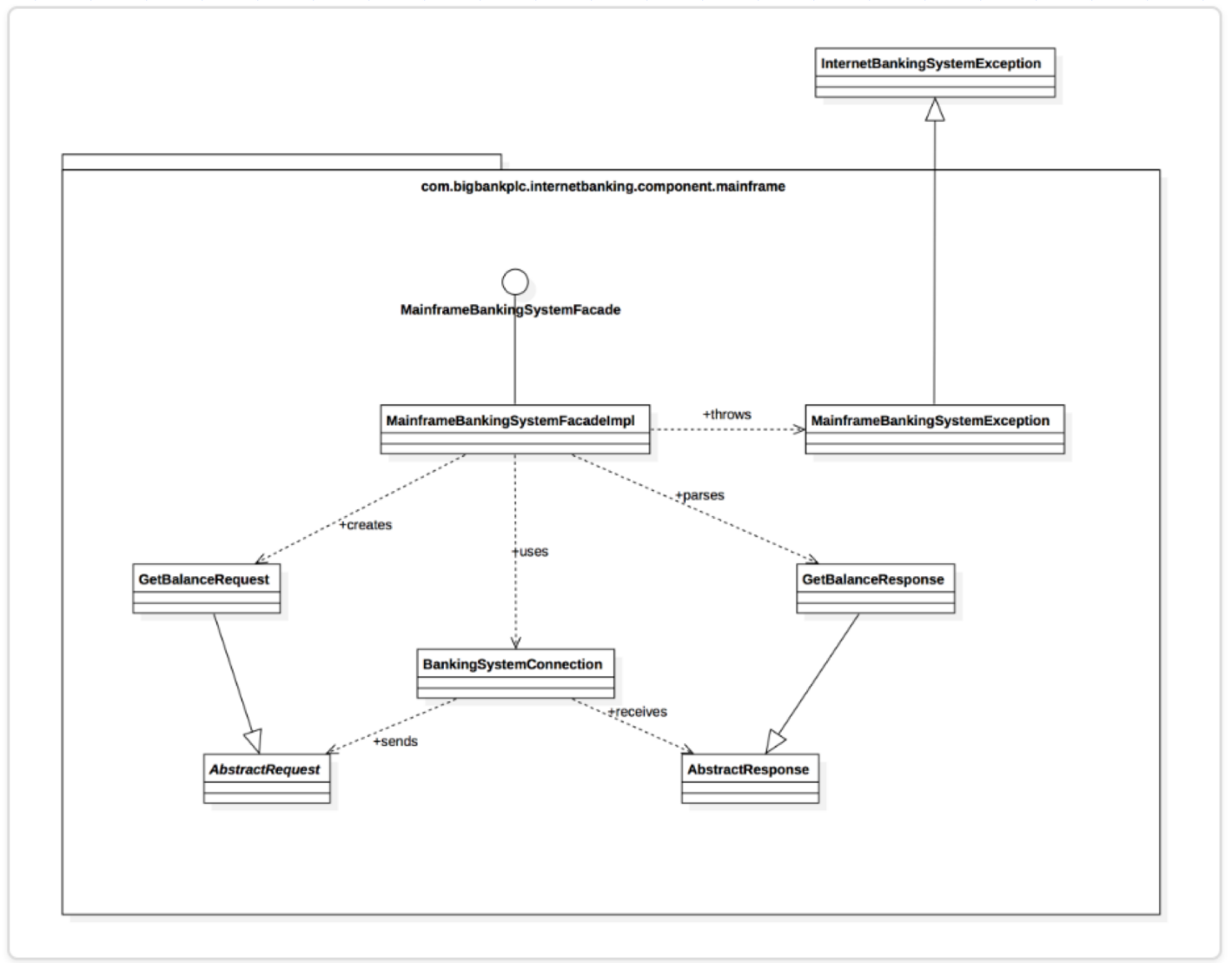
**Ref: Simon Brown**
**https://c4model.com/**



# C4 Examples

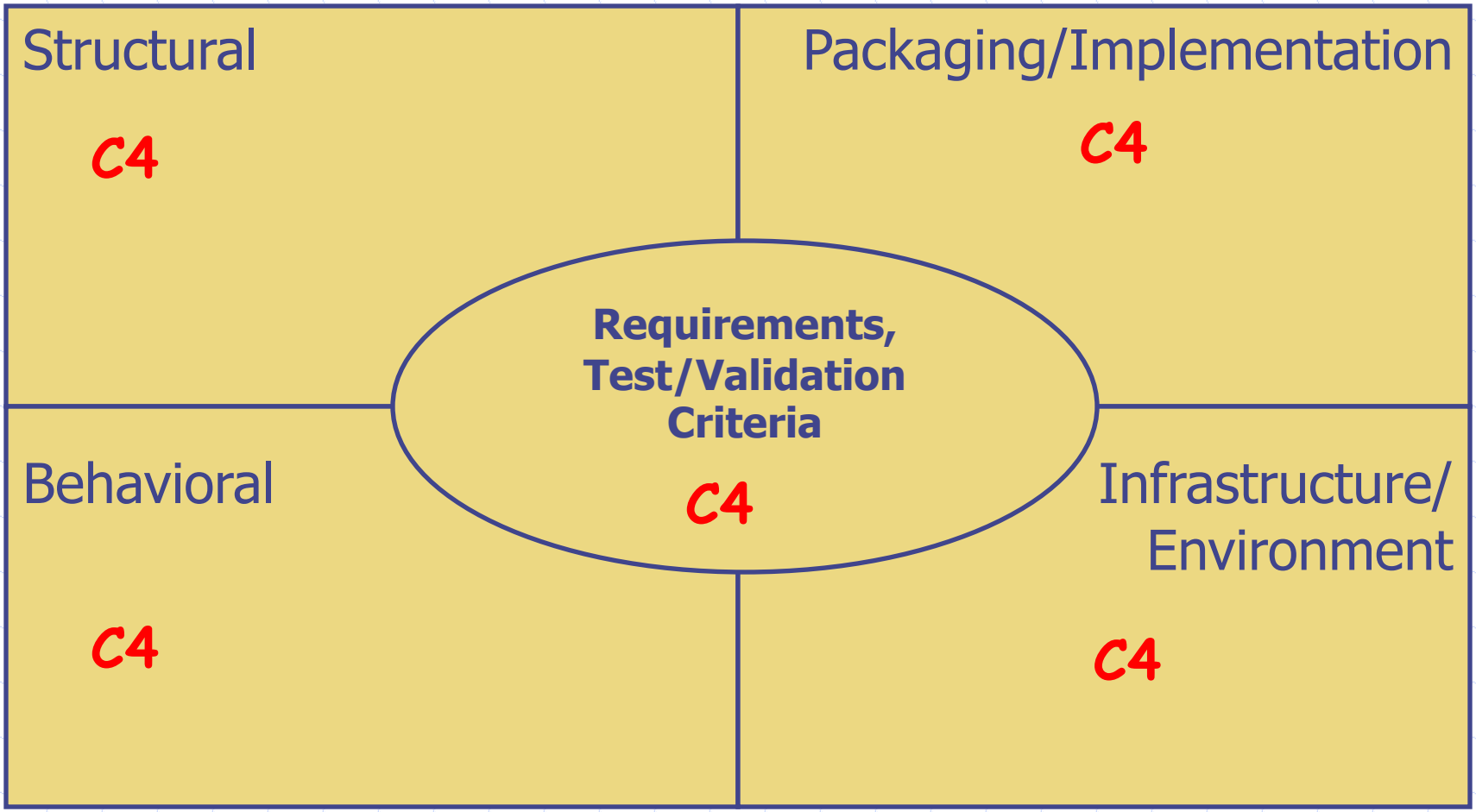
## Code (Optional)

**Ref: Simon Brown**
**https://c4model.com/**



# Modeling with C4

Ref: Simon Brown – See Supplementary Models
https://c4model.com/




Each type of C4 diagram can be focused on any aspects of the 4+1 model

# Architecture Use Cases

**There are different techniques to perform software architecture analysis…**

- Reference Architecture
- Solution Architecture
- Software Architecture

**As well as a diversity of use cases where this information can be useful…**

- New Product Definition
- Monitoring design quality (trends)
- Redocumenting existing systems

# Architecture Use Cases

**When thinking about a reference-, solution-, or software architecture focus on:**

- Who is the constituent – aka, who am I trying to talk to or influence
- What do I want them to understand – aka, why am I talking about architecture
- What are the benefits, constraints, or tradeoffs associated with the architecture I am describing

**We will look at documentation approaches later, but I'm not necessarily a big fan of standard notations – more on that later.**

# Defining Software Architecture

- **According to Shaw and Garlan…**

  *The Software Architecture of a system consists of a description of the system elements, interactions between the system elements, patterns that guide the system elements, and constraints on the relationships between system elements.*

  - Its a more abstract view of the design
  - Its helpful for communication and complexity management

# Software Architecture

- **The IEEE Definition…**

*"the highest level concept of a system in its environment. The architecture of a software system (at a given point in time) is its organization or structure of significant components interacting through interfaces, those components being composed of successively smaller components and interfaces."*

Ref: M. Fowler – Who Needs an Architect
http://files.catwell.info/misc/mirror/2003-martin-fowler-who-needs-an-architect.pdf

# Software Architecture

- **Another Definition…**

*"In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called 'architecture.' This understanding includes how the system is divided into components and how the components interact through interfaces. These components are usually composed of smaller components, but the architecture only includes the components and interfaces that are understood by all the developers."*

Ref: M. Fowler – Who Needs an Architect
http://files.catwell.info/misc/mirror/2003-martin-fowler-who-needs-an-architect.pdf

# Software Architecture

- **SEI Definition…**

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

*[Bass, Clements & Kazman, "Software Architecture in Practice", 1998]*

Problem: There is no standard definition – see http://www.sei.cmu.edu/architecture/definitions.html

# Software Architecture

- **My favorite…**

  *Architecture is about the important stuff. Whatever that is…. Ralph Johnson*

More on identifying the important stuff later, as what is important is largely system specific

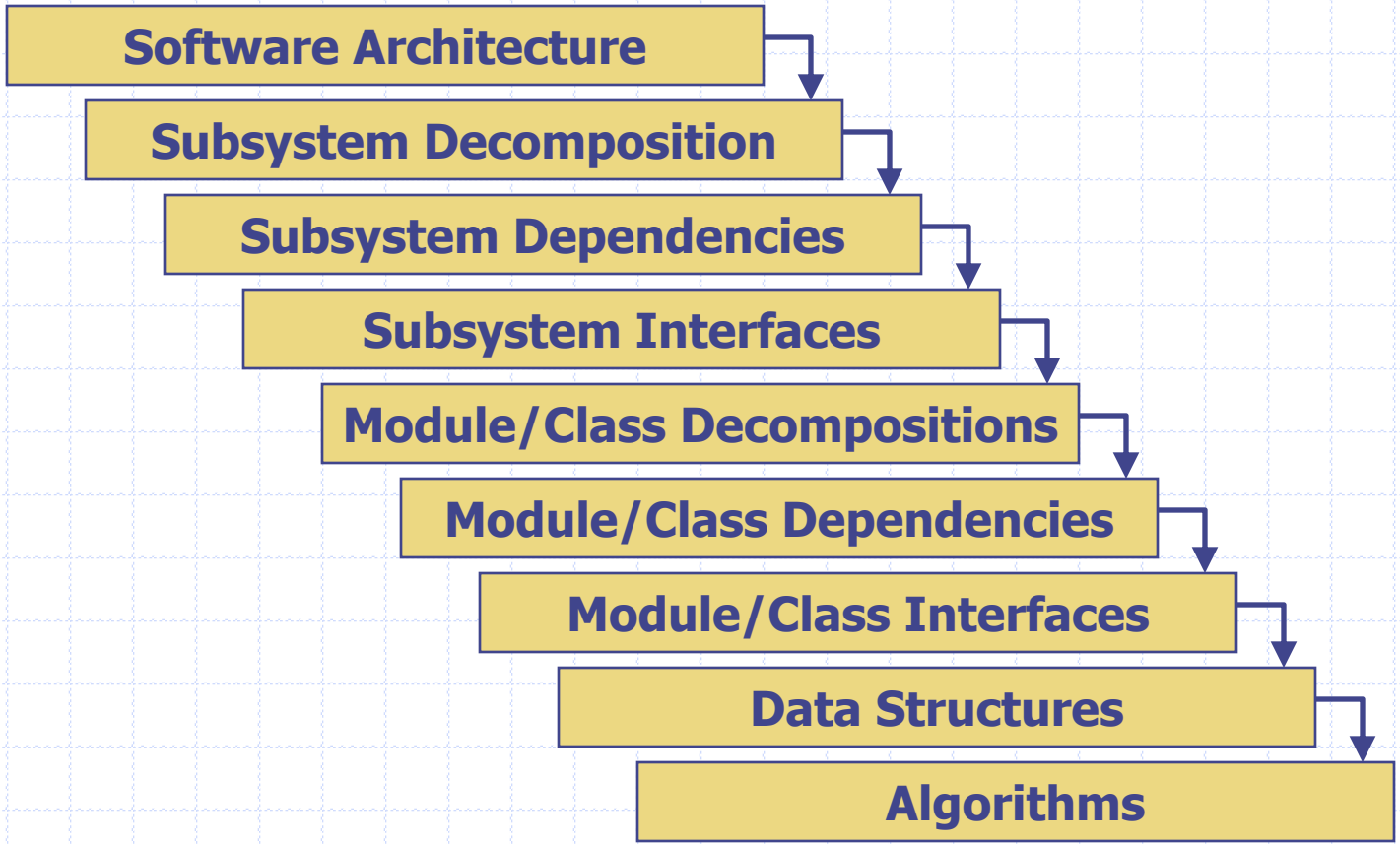
# Software Architecture - Summary

**All of the definitions of software architecture generally talk about**

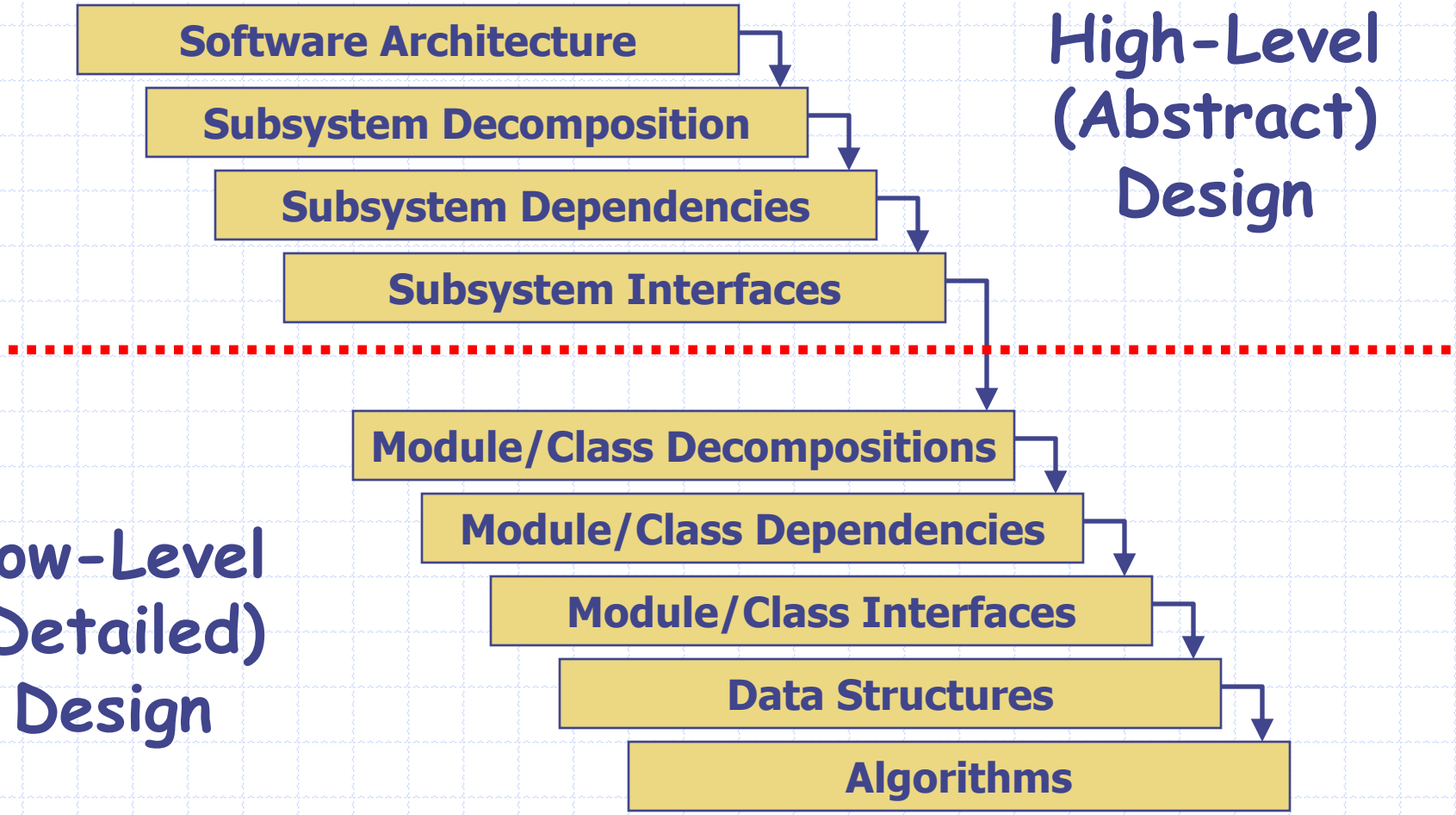
- What are the important "architectural" components
- What are the connectors between the architectural components
- What patterns, constraints, or decisions govern the restrictions on connecting the architectural components.

**Think about it, not all ways to connect architecture components together are equally good.**

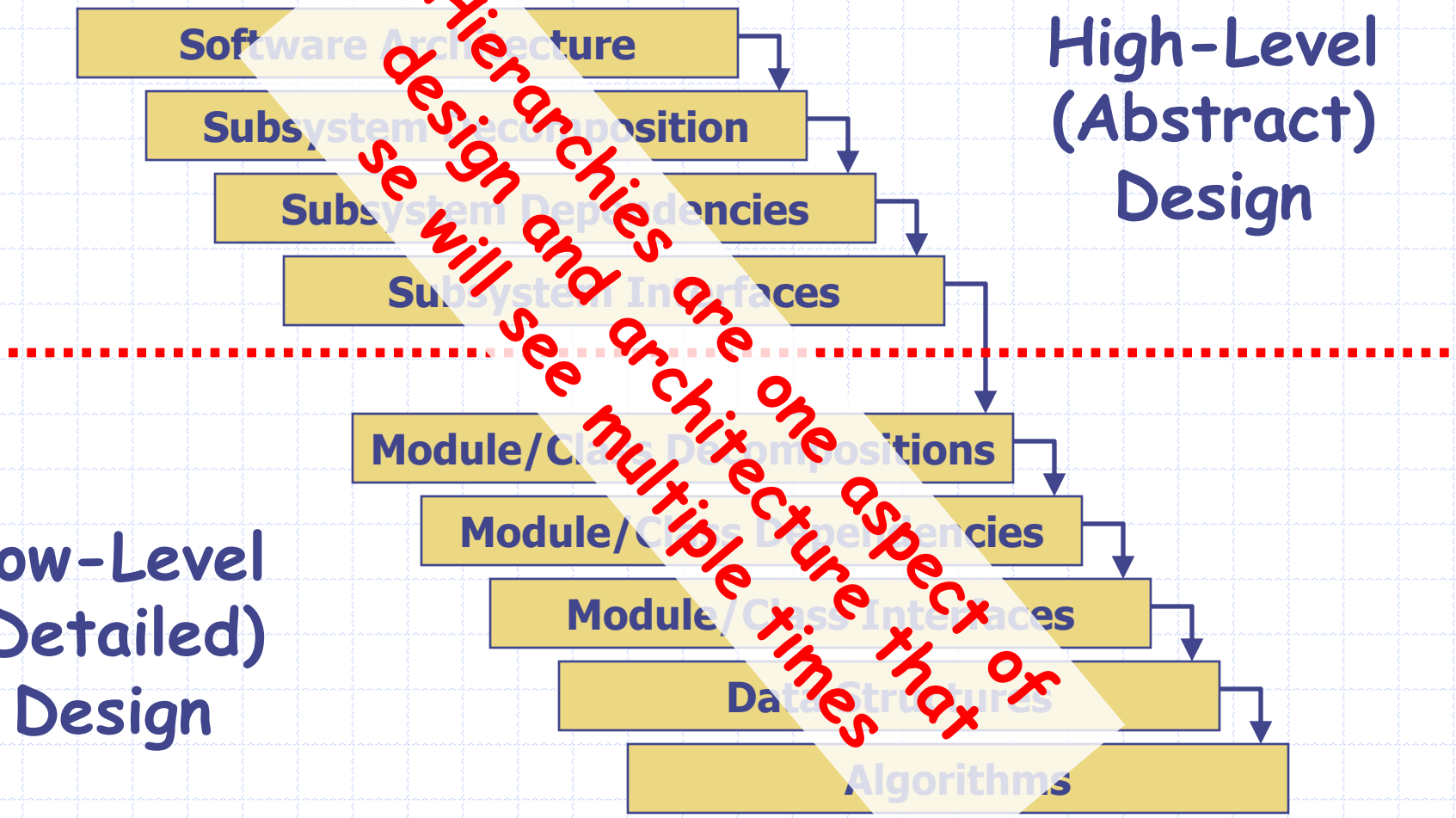
# The Software Architecture "Stack"



# The Software Architecture "Stack"

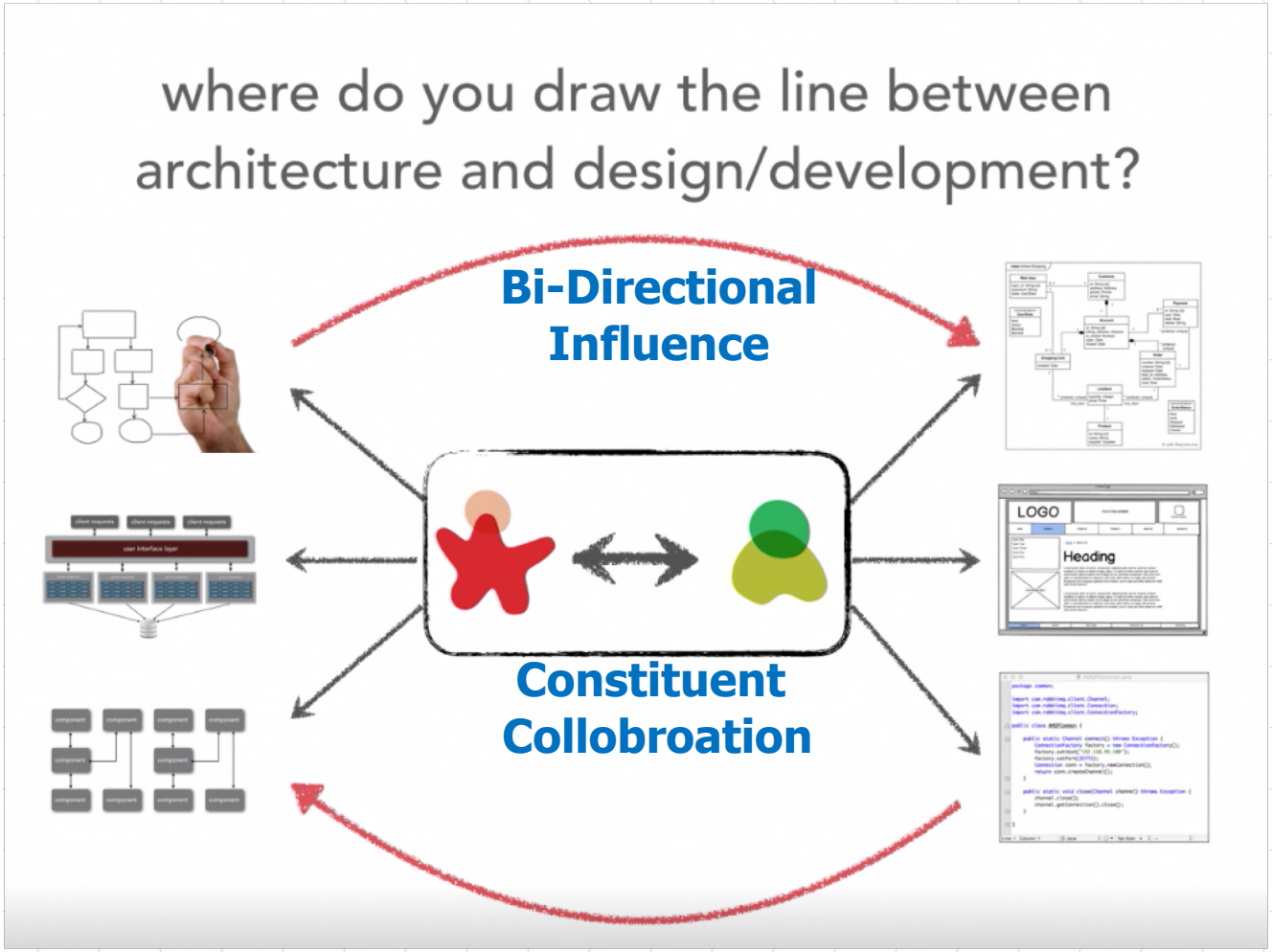


# The Software Architecture "Stack"



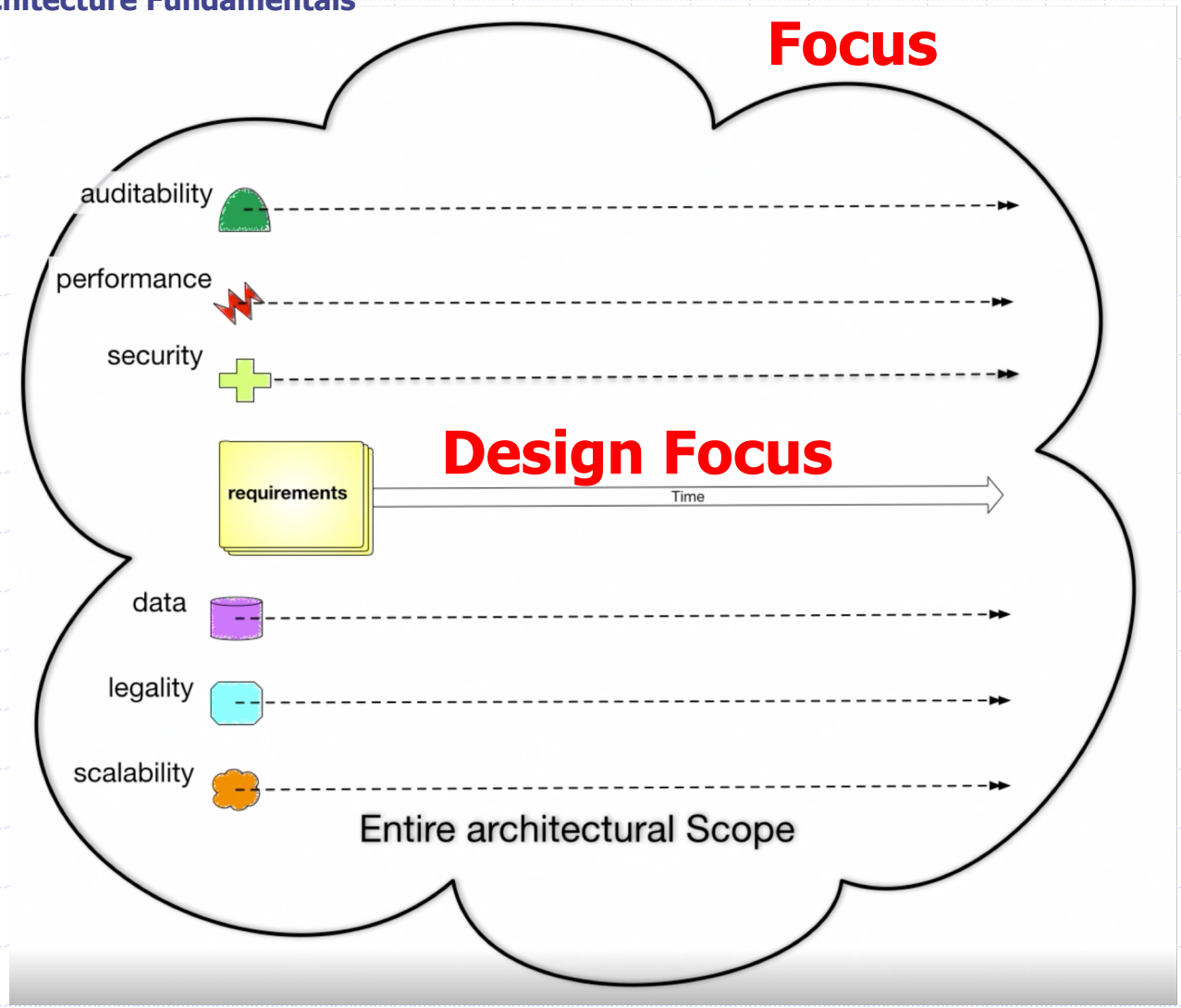
# We can also look at this non- hierarchical… The relation between the architecture and design spaces

**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**



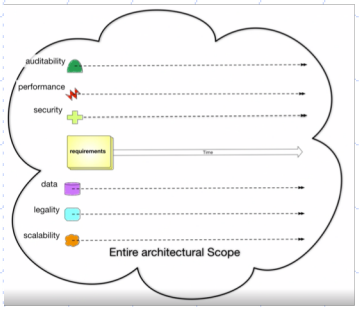
# Architecture has a broader scope than design

**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**



# System Quality Attributes

**Ref: https://en.wikipedia.org/wiki/List_of_system_quality_attributes**

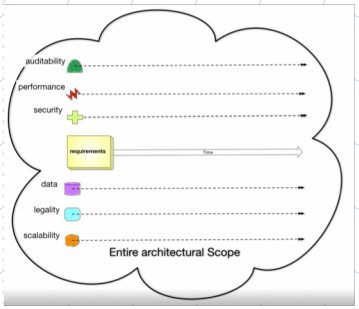



- accessibility
- accountability
- accuracy
- adaptability
- administrability
- affordability
- agility
- auditability
- autonomy
- availability
- compatibility
- composability
- configurability
- correctness
- credibility
- customizability
- debugability
- degradability
- determinability
- demonstrability
- dependability
- deployability
- discoverability [Erl]
- distributability
- durability
- effectiveness
- efficiency
- evolvability
- extensibility
- failure transparency
- fault-tolerance
- fidelity
- flexibility
- inspectability
- installability
- integrity
- interchangeability
- interoperability [Erl]
- learnability
- maintainability
- manageability
- mobility
- modifiability
- modularity
- operability
- orthogonality
- portability
- precision
- predictability
- process capabilities
- producibility
- provability
- recoverability
- relevance
- reliability
- repeatability
- reproducibility
- resilience
- responsiveness
- reusability [Erl]
- robustness
- safety
- scalability
- seamlessness
- self-sustainability
- serviceability securability
- simplicity
- stability
- standards compliance
- survivability
- sustainability
- tailorability
- testability
- timeliness
- traceability
- transparency
- ubiquity
- understandability
- upgradability
- vulnerability
- usability

These attributes and the tradeoffs associated with them influence architecture and design decisions

# System Quality Attributes

**Ref: https://en.wikipedia.org/wiki/List_of_system_quality_attributes**



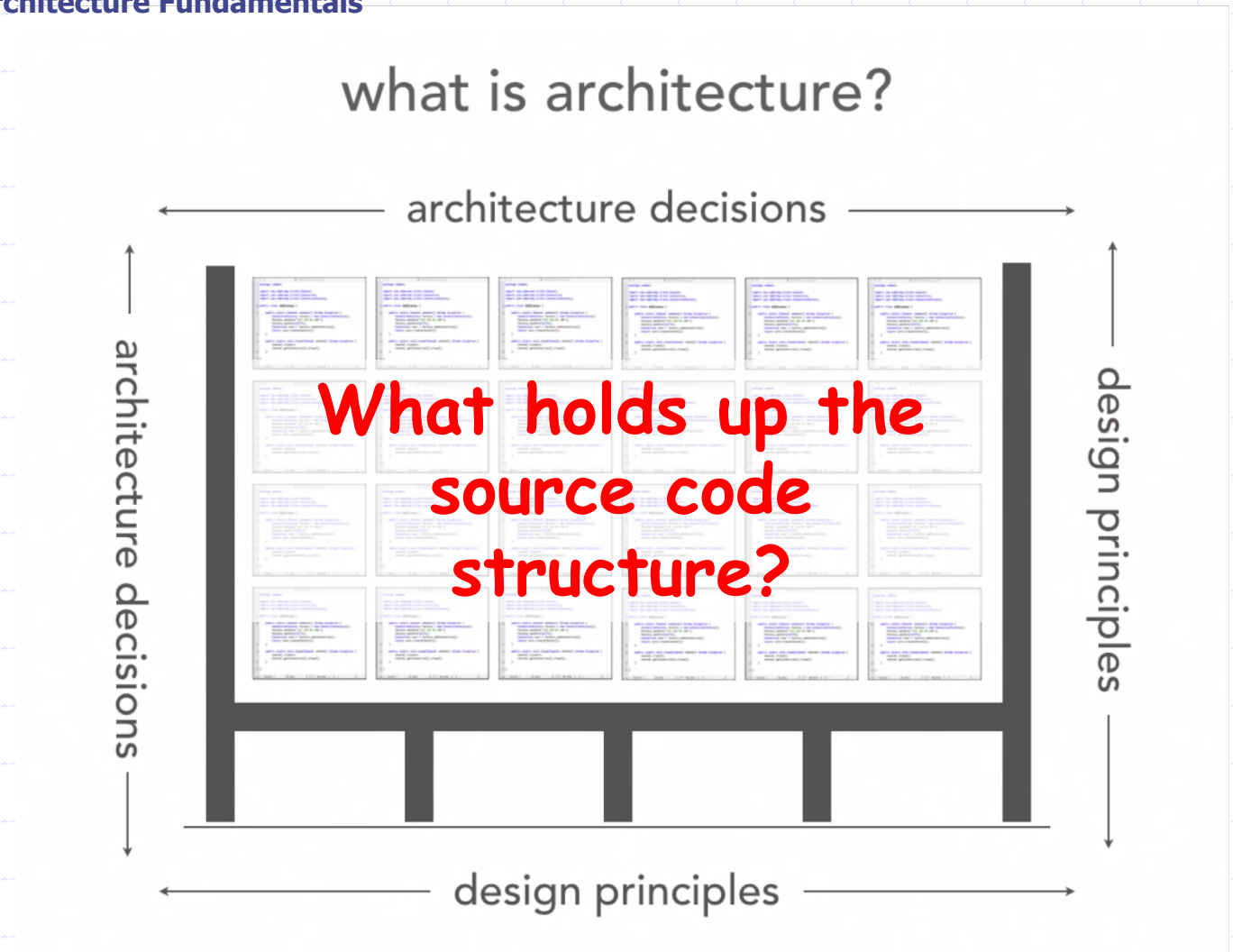

Although there are many quality attributes, certain ones tend to appear again and again that impact the design and architecture of systems

- Modularity
- Maintainability
- Testability
- Availability
- Deployability
- Reliability
- Scalability
- Evolvability
- Affordability

++ FEASIBILITY ++

# The structure of the source code should not be determined arbitrarily

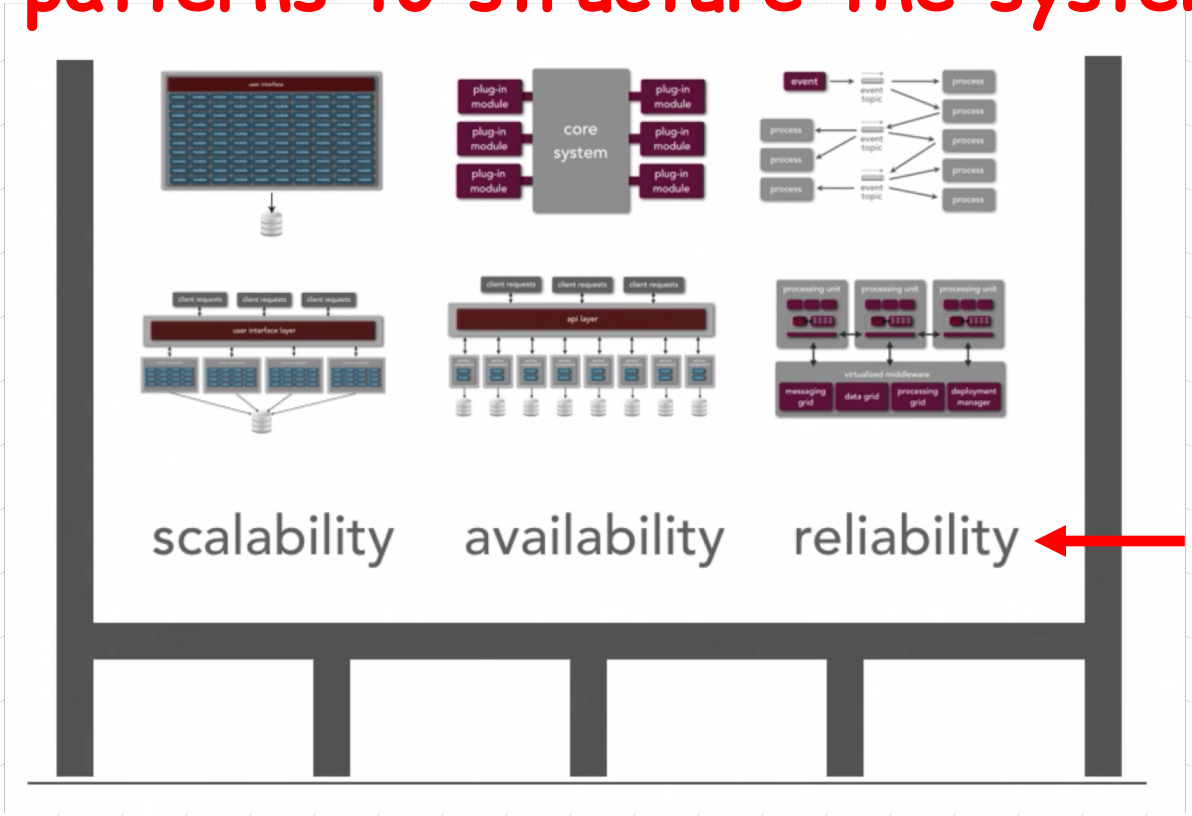
**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**



# Starting with understanding the structure of the system

**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**

**We can use architecture styles and patterns to structure the system**




**Needed quality attributes will also help dictate the system structure**

# Sidebar – Architecture Style versus Architecture Pattern?

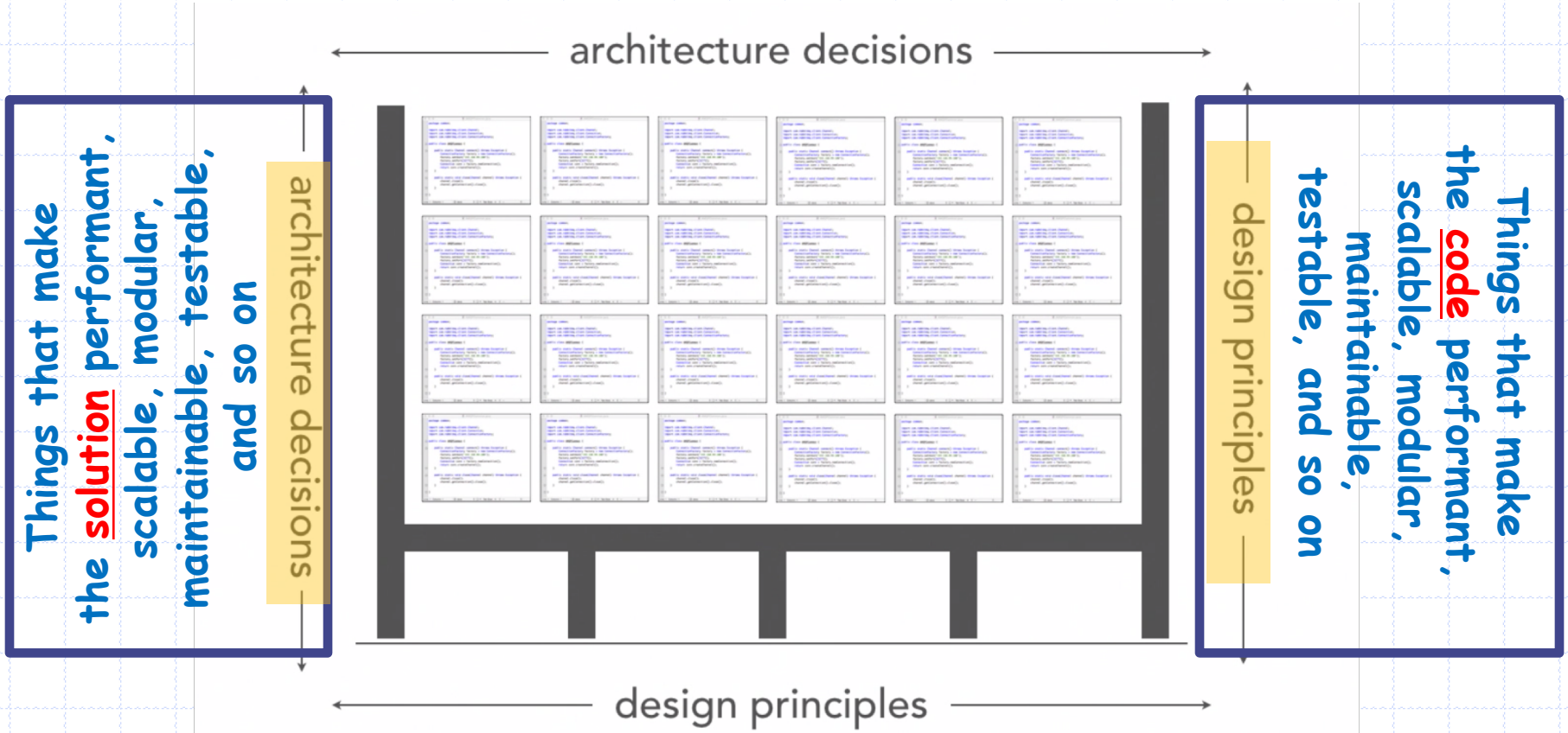
**We will look at the difference between architecture styles and patterns later…**

**While we can form a general understanding of their differences…**

**In practice, they can generally be used interchangeably**

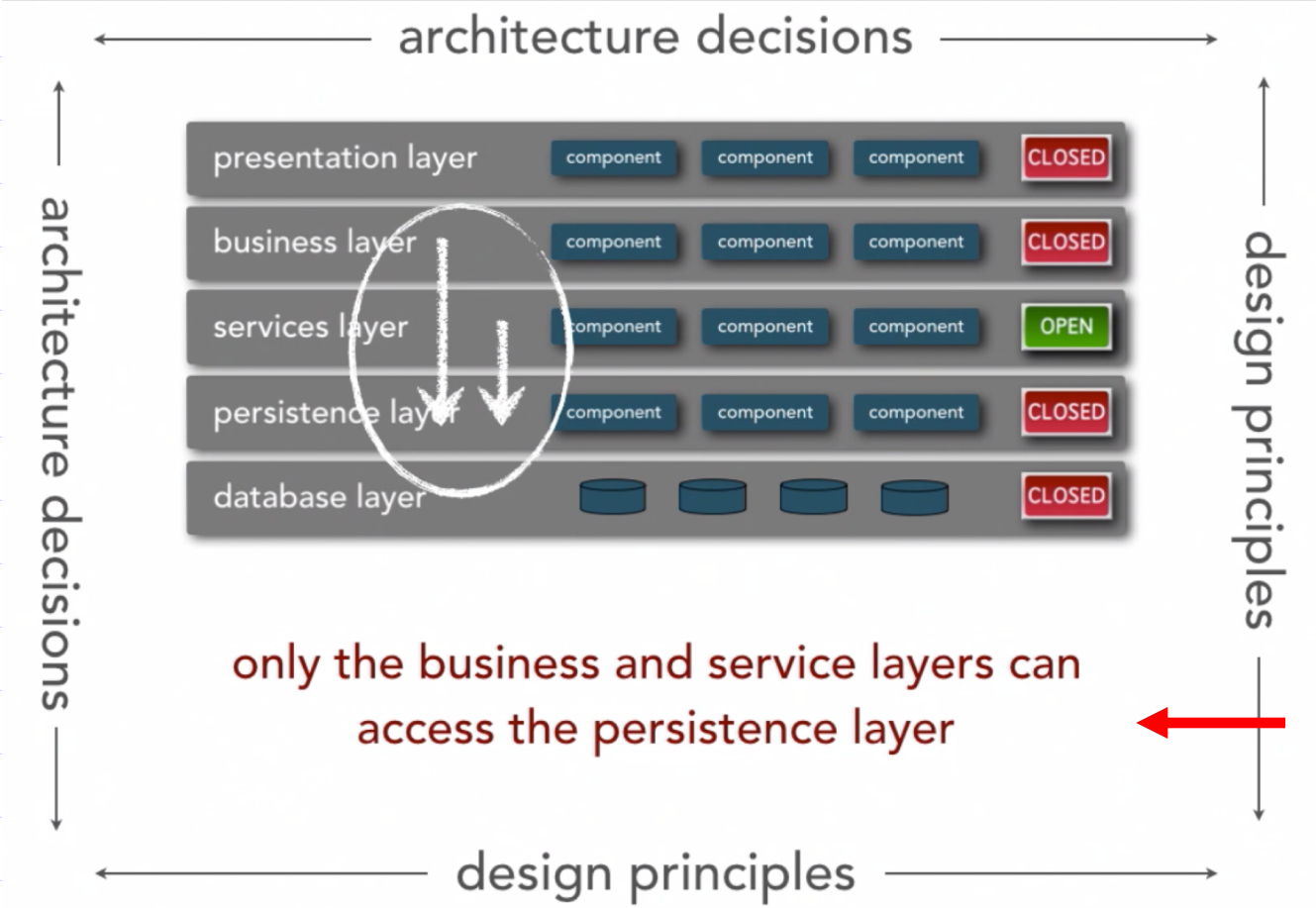
# Architecture decisions and design principles also influence the system structure

**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**



# Architecture principle example

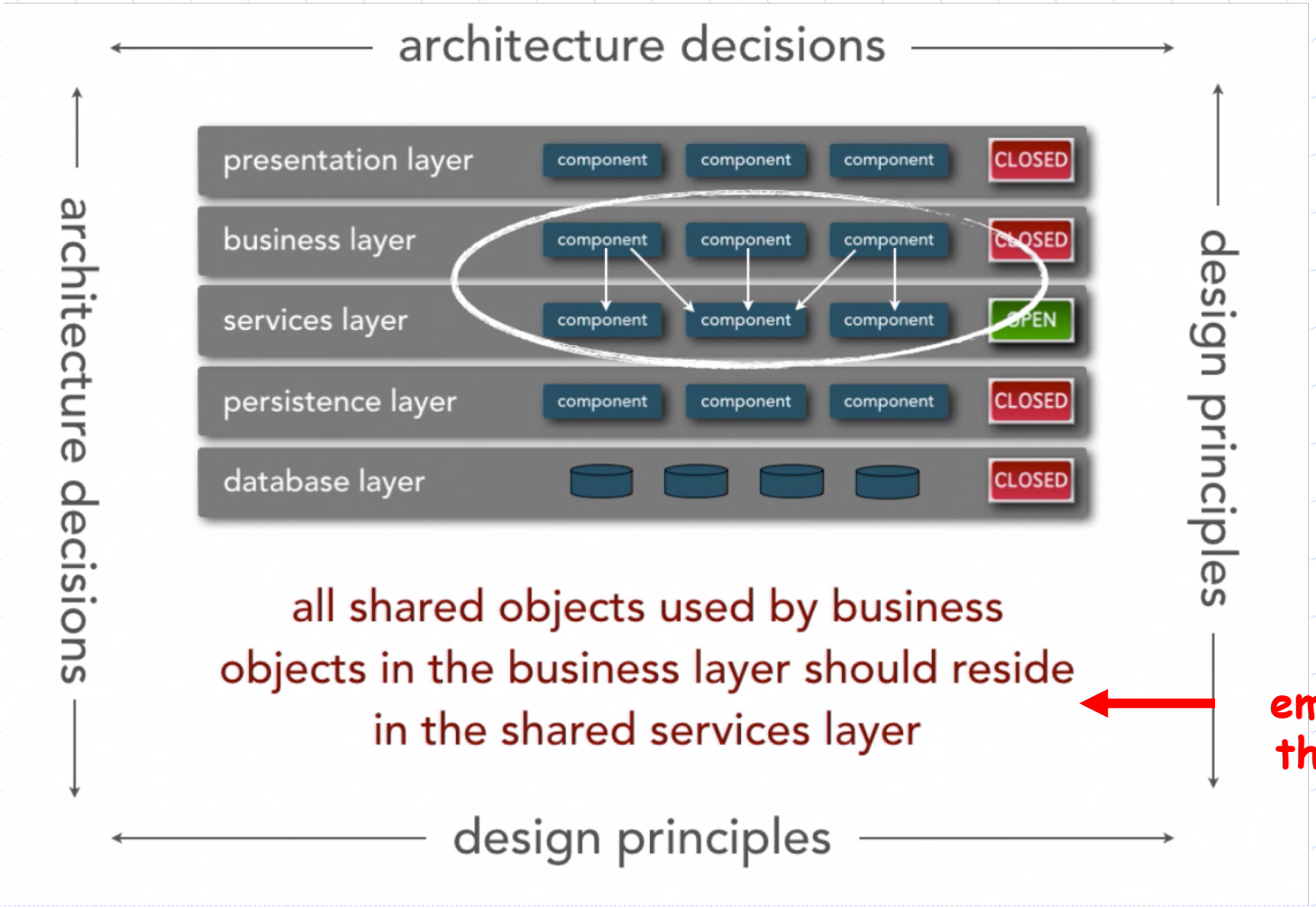
**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**




Reduces coupling by only allowing components to communicate to adjacent layers or within the same layer

# Another architecture principal example…

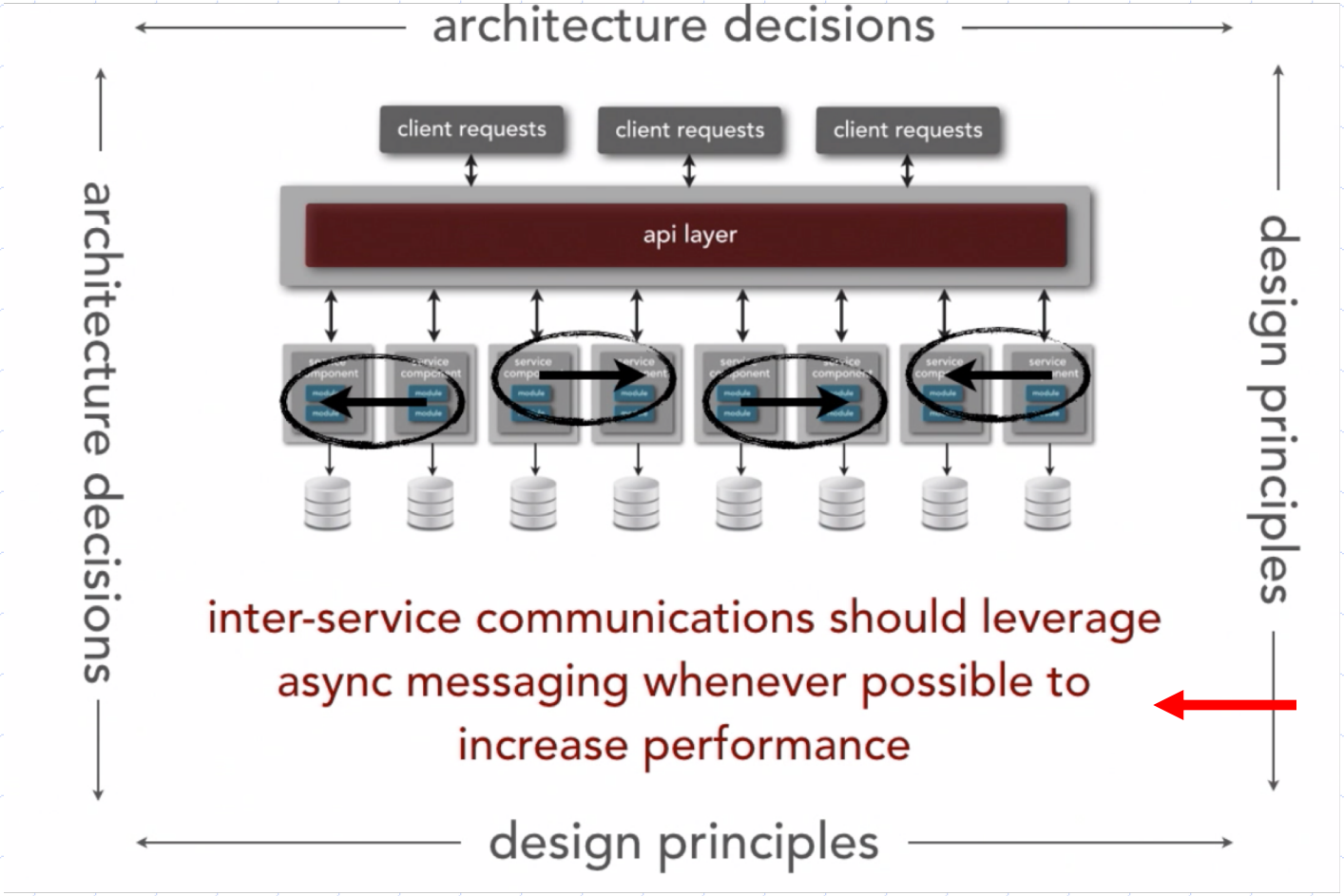
**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**




**Implies reuse emphasis will be in the services layer**

# Design principles can impact many platform and technology aspects

**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**




**A design decision that has been shown to have good performance characteristics… Tradeoff is introduction of complexity**

# Why do we design…

**A software design is not necessary for trivial systems, but for large systems a design is essential**

- Manage complexity
- Validation of delivered software
- Simplify future maintenance
- A mechanism for communication between domain experts and technical professionals
- Enables Visualization
- Enables project team members to work concurrently
  - Partitioning the work effort with limited overlap
  - Example:  Concurrently developing test cases while the code is being development

# Why is design so hard…

- Software design can't be taught, but principles of good design can
- There are degrees of good and bad design, but its hard to say if a design is correct or not
- The underlying assumptions and requirements that support the design change
- A design is like wine, it takes a long time to see if it is good or not

# The Process of Design

- Definition:
  - *Design* is a problem-solving process whose objective is to find and describe a way:
    - To implement the system's *functional requirements*...
    - While respecting the constraints imposed by the *quality, platform and process requirements...*
      - including the budget
    - And while adhering to general principles of *good quality*

# Design as a series of decisions

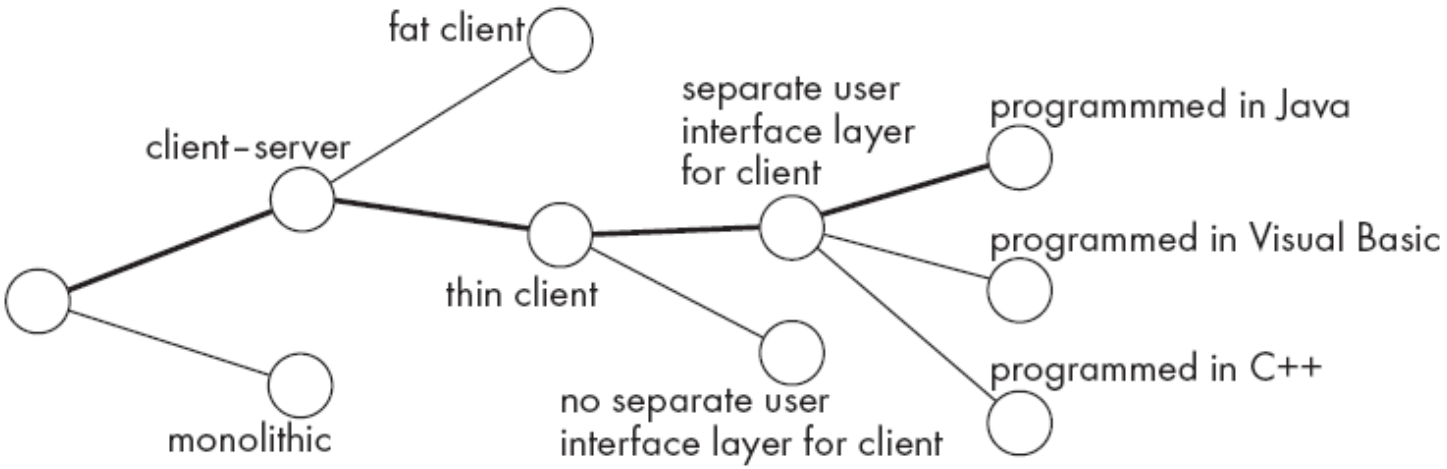
- A designer is faced with a series of *design issues*
  - These are sub-problems of the overall design problem.
  - Each issue normally has several alternative solutions:
    - design *options*.
  - The designer makes a *design decision* to resolve each issue.
    - This process involves choosing the best option from among the alternatives.

# Making decisions

- To make each design decision, the software engineer uses:
  - Knowledge of
    - the requirements
    - the design as created so far
    - the technology available
    - software design principles and ‘best practices’
    - what has worked well in the past

# Design space

- The space of possible designs that could be achieved by choosing different sets of alternatives is often called the *design space*
  - For example:



# Component

- Any piece of software or hardware that has a clear role.
  - A component can be isolated, allowing you to replace it with a different component that has equivalent functionality.
  - Many components are designed to be reusable.
  - Conversely, others perform special-purpose functions.

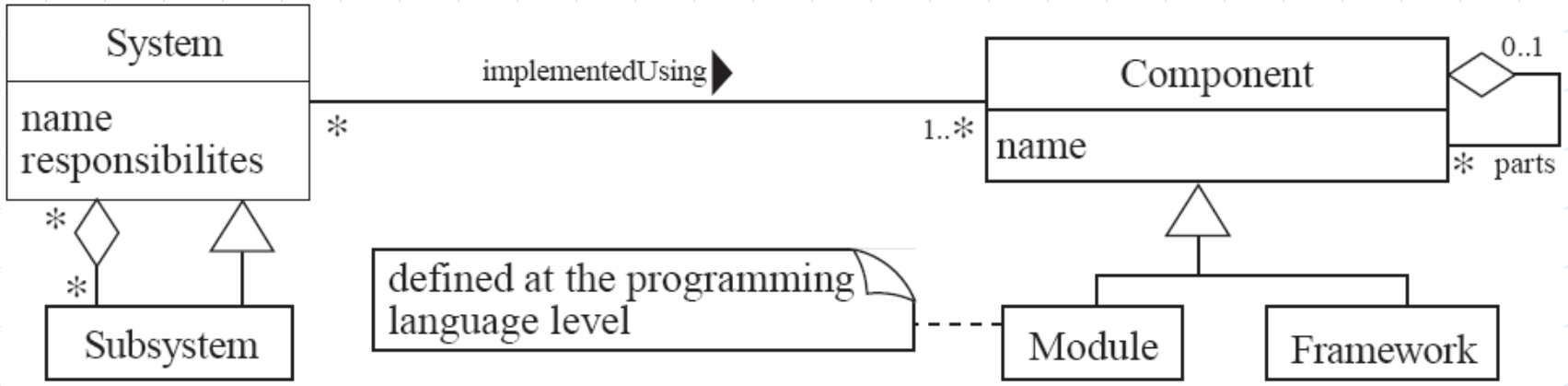
# Module

- A component that is defined at the programming language level
  - For example, methods, classes and packages are modules in Java.

# System

- A logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software or both.
  - A system can have a specification which is then implemented by a collection of components.
  - A system *continues to exist, even if its components are changed* or replaced.
  - The goal of requirements analysis is to determine the responsibilities of a system.
  - **Subsystem**:
    - A system that is part of a larger system, and which has a definite interface

# UML diagram of system parts



# Different aspects of design

- *Software architecture* :
  - The division into subsystems and components,
    - How these will be connected.
    - How they will interact.
    - Their interfaces.
- *Class design*:
  - The various features of classes.
- *User interface design*
- *Algorithm design*:
  - The design of computational mechanisms.
- *Protocol design*:
  - The design of communications protocol.

# 9.2 Principles Leading to Good Design

- Overall *goals* of good design:
  - Increasing profit / reducing cost / increasing revenue
  - Ensuring that we actually conform with the requirements
  - Accelerating development
  - Increasing qualities such as
    - Usability
    - Efficiency
    - Reliability
    - Maintainability
    - Reusability

# Design Principle 1: Divide and conquer

- Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things
  - Separate people can work on each part.
  - An individual software engineer can specialize.
  - Each individual component is smaller, and therefore easier to understand.
  - Parts can be replaced or changed without having to replace or extensively change other parts.

# Ways of dividing a software system

- A distributed system is divided up into clients and servers
- A system is divided up into subsystems
- A subsystem can be divided up into one or more packages
- A package is divided up into classes
- A class is divided up into methods
- Try to promote hierarchies

# Design Principle 2: Increase cohesion where possible

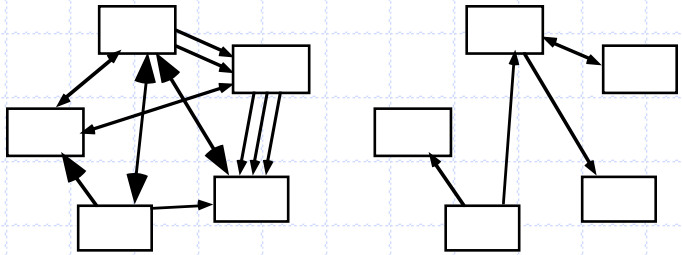
- A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things
  - This makes the system as a whole easier to understand and change
  - Type of cohesion:
    - Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

# Design Principle 2: Increase cohesion where possible

| | |
|---|---|
| Functional | Keeps all code that computes a particular result together |
| Layer | Facilities for accessing related services are kept together. Hierarchical, you can access layers below but not above |
| Communicational | Modules that access or manipulate certain data are kept together |
| Sequential and Procedural | Procedures where one provides input to the next are kept together in a common module. Also procedures that are used one after the next are kept together |
| Temporal | Operations that are performed at the same phase of execution are kept together (e.g., Initialization) |
| Utility | Keep related semantic things together, e.g., java.lang.math |

# Design Principle 3: Reduce coupling where possible

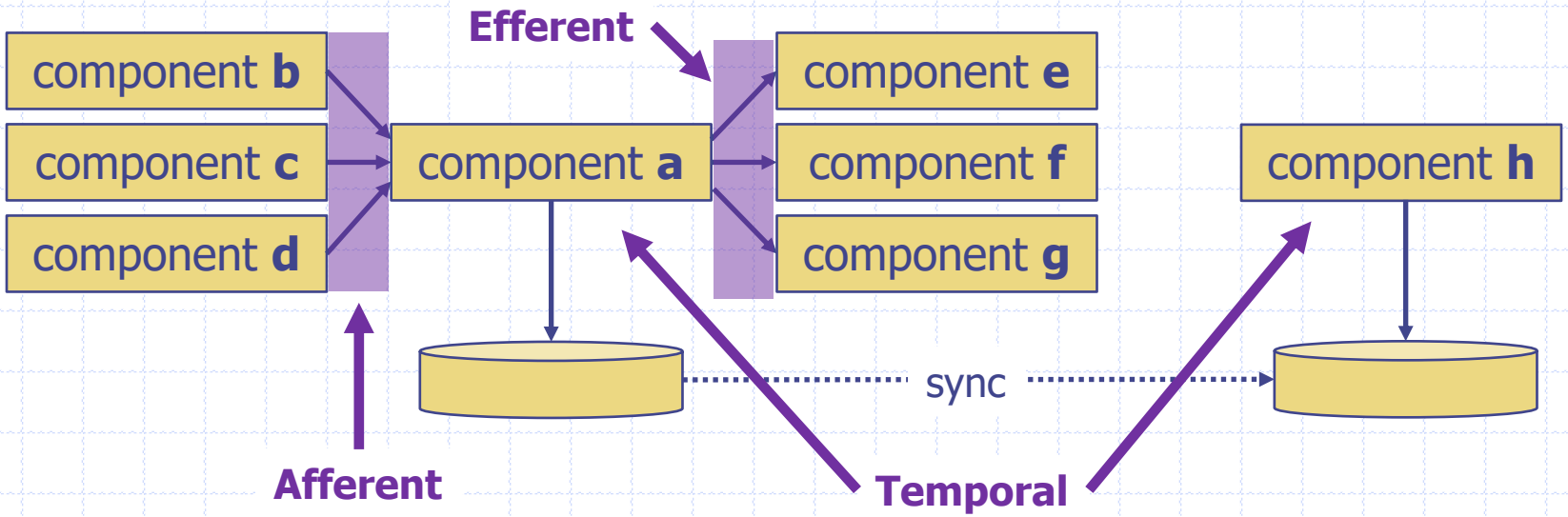
- *Coupling* occurs when there are *interdependencies* between one module and another
  - When interdependencies exist, changes in one place will require changes somewhere else.
  - A network of interdependencies makes it hard to see at a glance how some component works.
  - Type of coupling:
    - Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External



# Design Principle 2: Reduce coupling where possible – Code Level Things

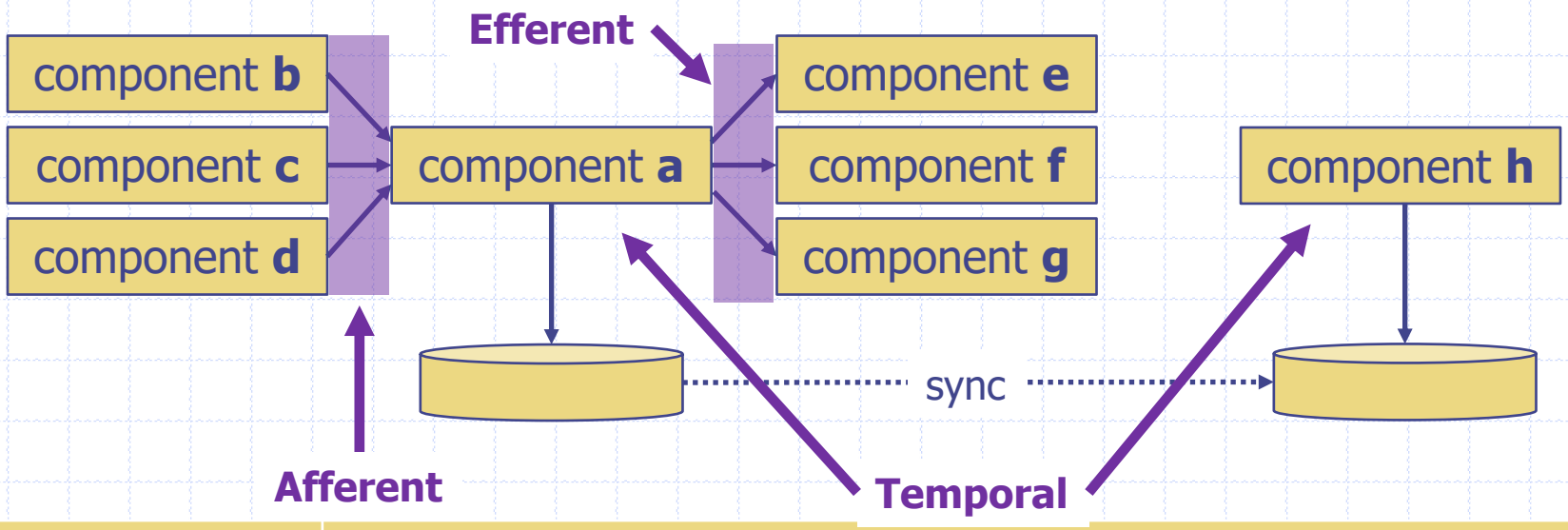
| | |
|---|---|
| Content | One module modifies internal data of another module |
| Common | Coupling based on a global variable |
| Control | Parameter is passed as an argument that controls the behavior of the callee, e.g, ("CustomerArrived" passed as an arg that is then processed by a switch statement |
| Stamp | Using custom types as arguments to a method or function. Couples class dependencies – very common in OO languages |
| Routine | When one method calls a method in another class or package |
| Type use | When one module uses a type defined in another module |
| Inclusion / Import | When one module imports another module |
| External | When a module has an external dependency on things like an operating system, shared library or hardware |

# Design Principle 2: Reduce coupling where possible – Architecture Things




| | |
|---|---|
| Afferent | How multiple other components depend on a target component |
| Efferent | How dependent a component is on other components |
| Temporal | Coupling introduced by non-static timing dependencies |

# Design Principle 2: Reduce coupling where possible – Architecture Things




| | |
|---|---|
| Pathologic | One component depends on the inner workings of another |
| External | Multiple components share an externally imposed protocol or data format |
| Control | One component passes information on another component on what to do |
| Data | Components are bound to the same data context |

# Design Principle 4: Increase abstraction

- Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity
  - A good abstraction is said to provide *information hiding*
  - Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details
  - Examples of abstractions:
    - Classes
    - UML associations
    - Interfaces
    - State machines
    - Domain specific languages (DSLs)

# Design Principle 5: Increase reusability where possible

- Design the various aspects of your system so that they can be used again in other contexts
  - Generalize your design as much as possible
  - Follow the preceding three design principles
  - Design your system to contain hooks
  - Simplify your design as much as possible

# Design Principle 6: Reuse where possible

- Design with reuse is complementary to design for reusability
  - Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
    - *Cloning* should not be seen as a form of reuse

# Design Principle 7: Design for flexibility

- Actively anticipate changes that a design may have to undergo in the future, and prepare for them
  - Reduce coupling and increase cohesion
  - Create abstractions
  - Do not hard-code anything
  - Leave all options open
    - Do not restrict the options of people who have to modify the system later
  - Use reusable code and make code reusable

# Design Principle 8: Anticipate obsolescence

- Plan for changes in the technology or environment so the software will continue to run or can be easily changed
  - Avoid using early releases of technology
  - Avoid using software libraries that are specific to particular environments
  - Avoid using undocumented features or little-used features of software libraries
  - Avoid using software or special hardware from companies that are less likely to provide long-term support
  - Use standard languages and technologies that are supported by multiple vendors

# Design Principle 9: Design for Portability

- Have the software run on as many platforms as possible
  - Avoid the use of facilities that are specific to one particular environment
  - E.g. a library only available in Microsoft Windows

# Design Principle 10: Design for Testability

- Take steps to make testing easier
  - Design a program to automatically test the software
    - Discussed more in Chapter 10
    - Ensure that all the functionality of the code can by driven by an external program, bypassing a graphical user interface
  - In Java, you can create a main() method in each class in order to exercise the other methods
  - Use Junit or similar frameworks

# Design Principle 11: Design defensively

- Never trust how others will try to use a component you are designing
  - Handle all cases where other code might attempt to use your component inappropriately
  - Check that all of the inputs to your component are valid: the *preconditions*
    - Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking

# Example: SOLID Design Principles

From WikiPedia - http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

| Initial | Principle |
|---|---|
| S | **Single responsibility principle**<br>a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class) |
| O | **Open/closed principle**<br>"software entities … should be open for extension, but closed for modification." |
| L | **Liskov substitution principle**<br>"objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract. |
| I | **interface segregation principle**<br>"many client-specific interfaces are better than one general-purpose interface." |
| D | **Dependency inversion principle**<br>one should "Depend upon Abstractions. Do not depend upon concretions."<br>Dependency injection is one method of following this principle. |

# Example: Functional Programming

| Principle |
| --- |
| **Immutability**<br>Values can be calculated and determined once, from there they cannot be altered. There are novel persistent data structures to make this efficient. |
| **(Pure) Functions**<br>Functions accept parameters and return values, and can be chained together. |
| **Function are a First Class Citizen**<br>Functions can be accepted as parameters, and returned as a result from another function. Functions can be assigned to variables, and executed in either a greedy and/or lazy manner. |

- ✓ Abstraction
- ✓ Reusability
- ✓ Flexibility
- ✓ Testability

# Is Software Design a Wicked Problem…

- Types of Problems:
  - Simple problems: Both the problem and the solution are known
  - Complex problems: The problem is known but the solution is not.
  - Wicked Problem: Neither the problem or solution is known

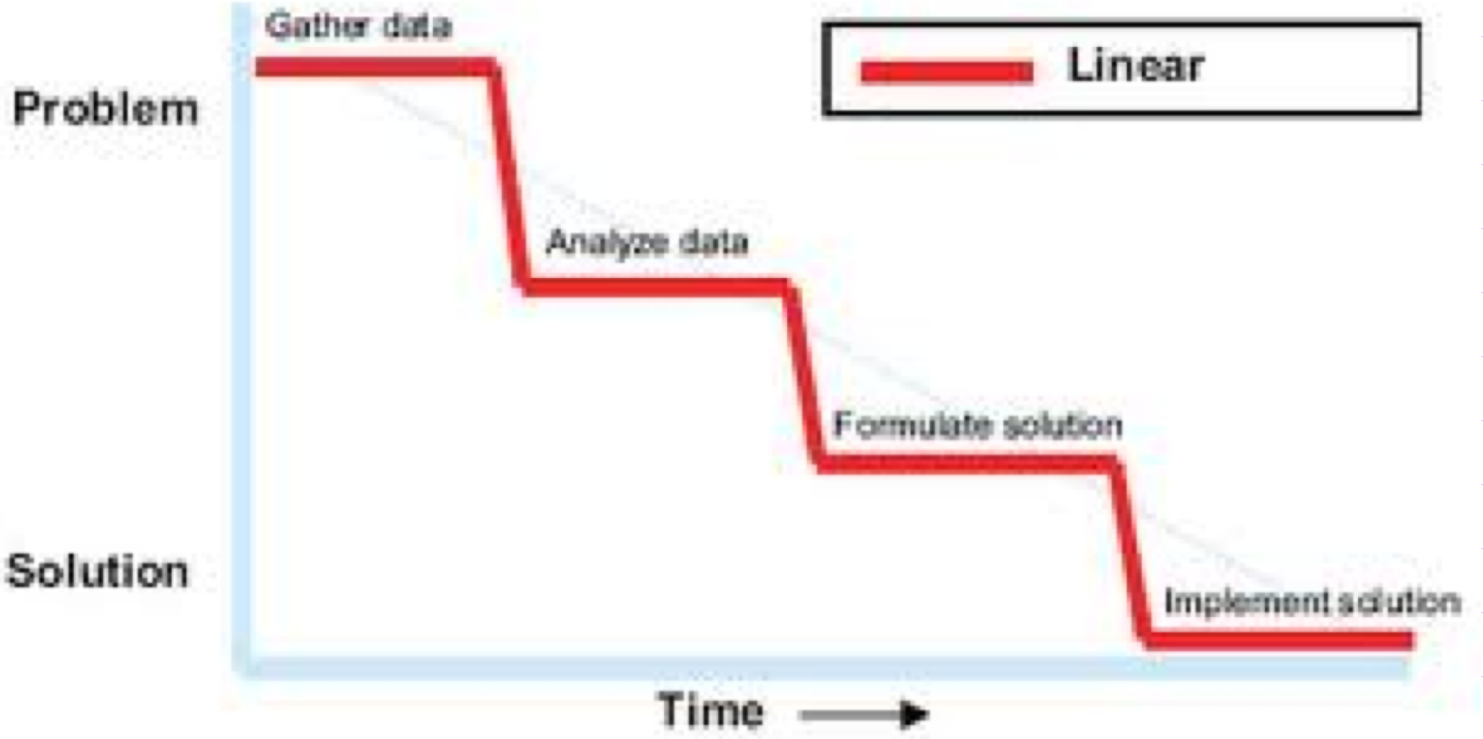
So is software design a Wicked Problem?

# What is a Wicked Problem?

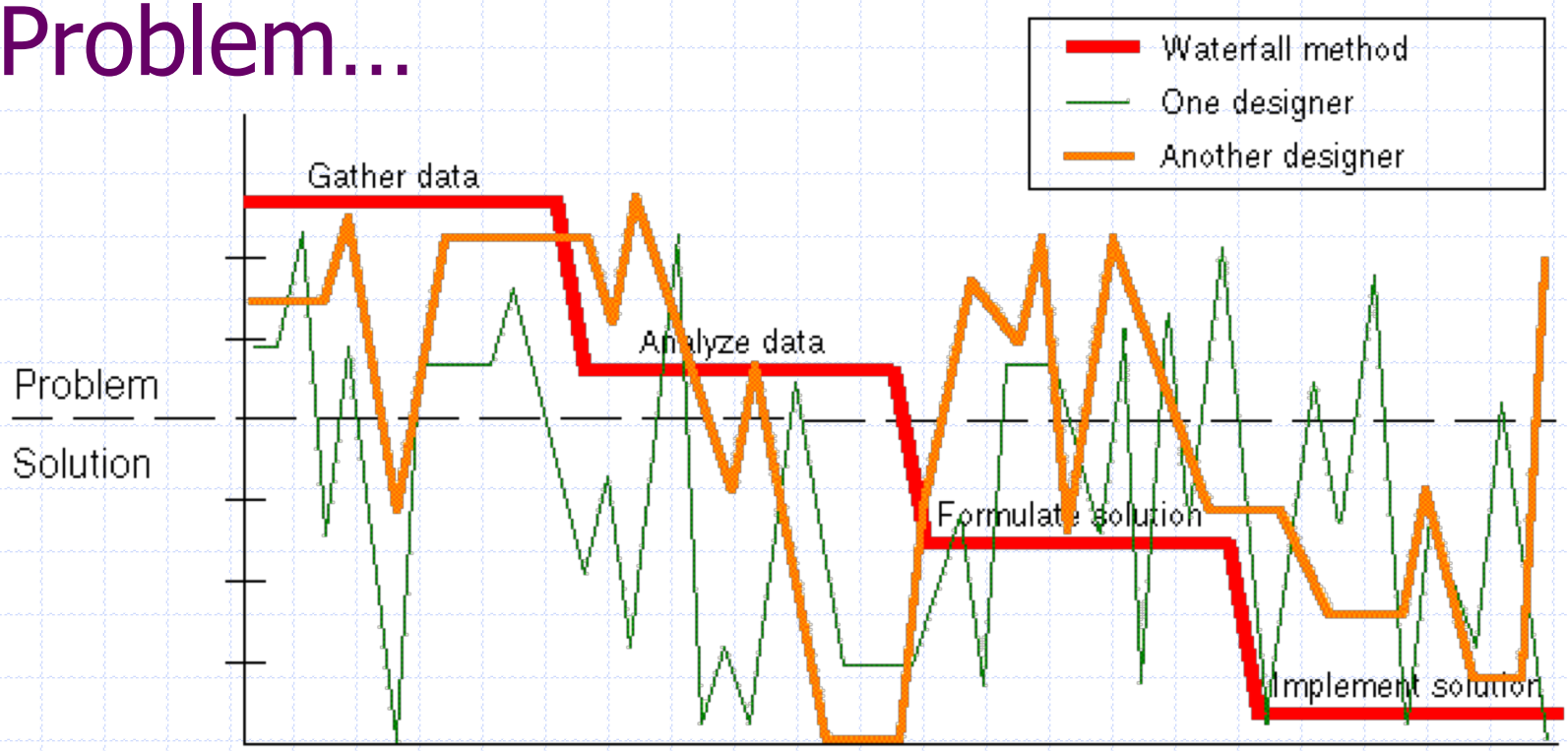
- Characteristics of Wicked Problems:
  - The understanding of the problem is driven by the formulation of a (proposed) solution
  - There is no stopping rule – you will never know when the perfect solution is defined
  - You can only tell that a solution "is good enough"
  - You can only tell that a solution is successful after it is built

# Is Software Design a Wicked Problem…

The intuitive way to solve a problem is to follow an orderly flow from problem definition to solution formulation



# Is Software Design a Wicked Problem…




When faced with complex problems humans tend to jump from the problem to the solution space, using their intuition in the problem domain, do validation in the solution domain, that then drives further intuition…

# Dealing with Wicked Problems

- Techniques to deal with Wicked Problems:
  - Lock down the problem domain
  - Specify objective measures to measure the success of a solution
  - Give up on the notion that there is a "perfect" solution, and focus on what would be a "good enough solution"
  - Balance what is needed with what can be built given the constraints of solving the problem

# Complex/Wicked Problems in Software Engineering

- Software Development Methodologies help manage the complexity of designing and building complex systems
- Brings structure to answering questions such as:
  - What does the software need to do
  - How long will the software take to construct
  - How much investment will constructing the software take
  - What are the technical constraints associated with building the software
  - And So On…

# Wicked Problems and Ultra-Large Scale Systems

- In 2006, SEI Introduced the term Ultra-Large Scale Systems, these systems:
  - Have decentralized data, development, evolution and operational control
  - Address inherently conflicting, unknowable, and diverse requirements
  - Evolve continuously while it is operating, with different capabilities being deployed and removed
  - Contain heterogeneous, inconsistent and changing elements
  - Erode the people system boundary. People will not just be users, but elements of the system and affecting its overall emergent behavior.
  - Encounter failure as the norm, rather than the exception, with it being extremely unlikely that all components are functioning at any one time
  - Require new paradigms for acquisition and policy, and new methods for control

# Characteristics of Ultra-Large Scale Systems

- A ULS System has unprecedented scale in some of these dimensions:
  - lines of code
  - amount of data stored, accessed, manipulated, and refined
  - number of connections and interdependencies
  - number of hardware elements
  - number of computational elements
  - number of system purposes and user perception of these purposes
  - number of routine processes, interactions, and "emergent behaviors"
  - number of (overlapping) policy domains and enforceable mechanisms
  - number of people involved in some way

# Designing beyond Human Ability

**https://dreamsongs.com/Files/DesignBeyondHumanAbilitiesSimp.pdf**

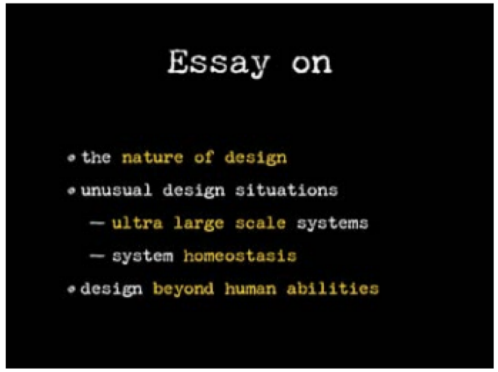
## Design Beyond Human Abilities

Richard P. Gabriel

*On November 16, 2007, I presented a talk on Design as my Wei Lun Public Lecture at the Chinese University of Hong Kong. This is a revised and reimagined transcript of that talk.*

This talk is an essay on design. In the 16th century, Michel de Montaigne invented a new genre of writing he called an *essai*, which in modern French translates to *attempt*. Since then, the best essays have been explorations by an author of a topic or question, perhaps or probably without a definitive conclusion. Certainly there can be no theme or conclusion stated at the outset, repeated several times, and supported throughout, because a true essay takes the reader on the journey of discovery that the author has or is experiencing.

This essay—on design—is based on my reflections on work I've done over the past 3 years. Some of that work has been on looking at what constitutes an "ultra large scale software system" and on researching how to keep a software system operating in the face of internal and external errors and unexpected conditions.

# Back to Architecture

- The previous definitions presented focus on the vocabulary of architecture components
  - Terms used: Components, Connectors, Structures, and so on.
- I like to think about architecture in terms of laying out the foundation for design
  - What **design decisions** do we need to make, and/or what **structures do we need to document** in order to realize all of the constraints imposed on the system
  - Constraints typically come from the non-functional requirements – time to market, budget, technology standards, skillsets, etc
- Good architecture makes important decisions early and defers less-important decisions to later
  - Do we really need to pick the database technology up front?
  - How much should the fact that the system is web-based influence the overall design, can we abstract this for now and specify it later?

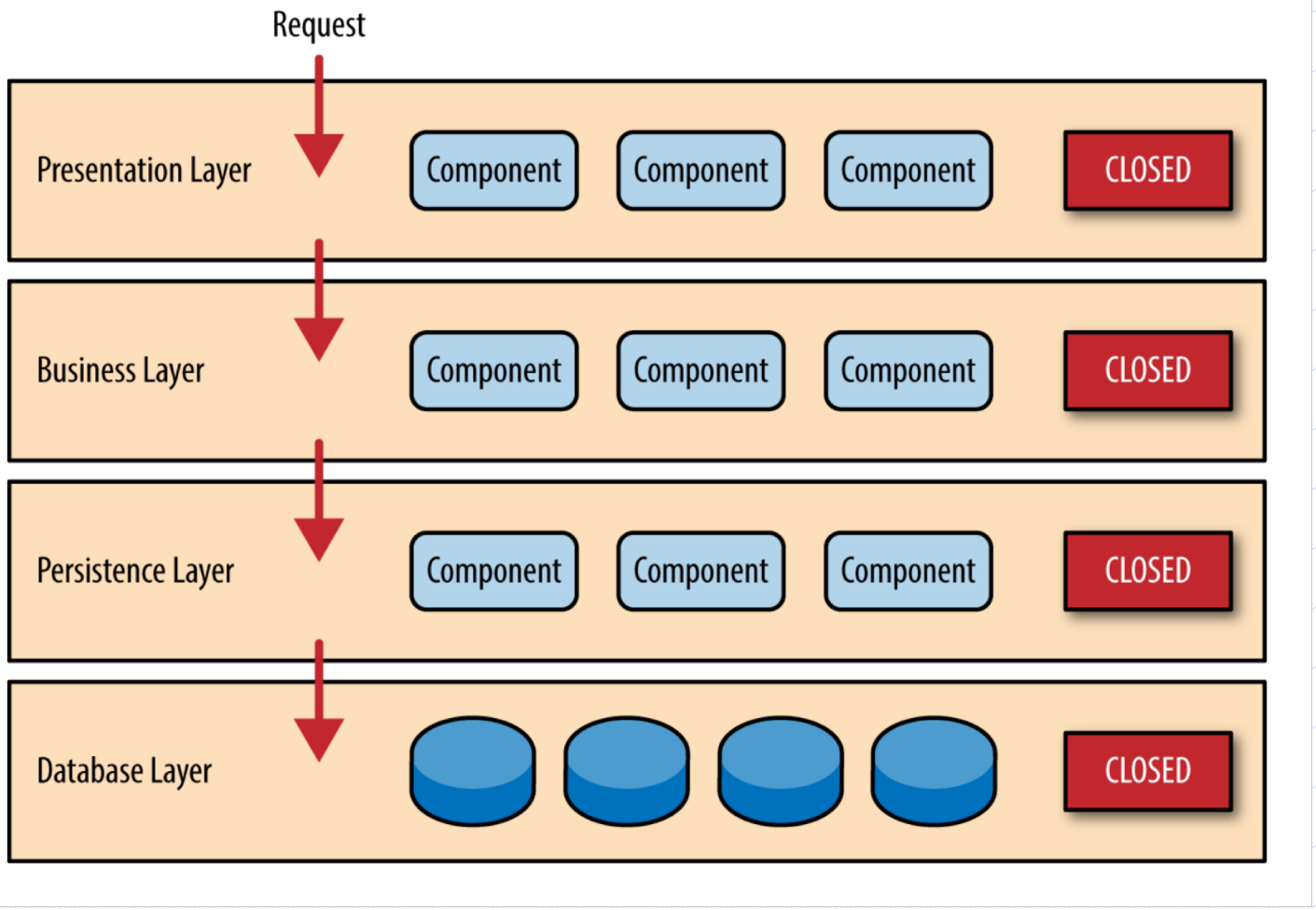
# Architecture and Design Patterns

- Architecture Style
  - Defines the **vocabulary** of the components and connectors in a software architecture and the **constraints** on how they can be combined.
  - Example: Pipe and Filter
- Architecture Patterns
  - Focus is on the organization of the overall system
  - An instance of an architecture style
- Design Patterns
  - Focus is on the organization of a block of code

# Architecture Styles

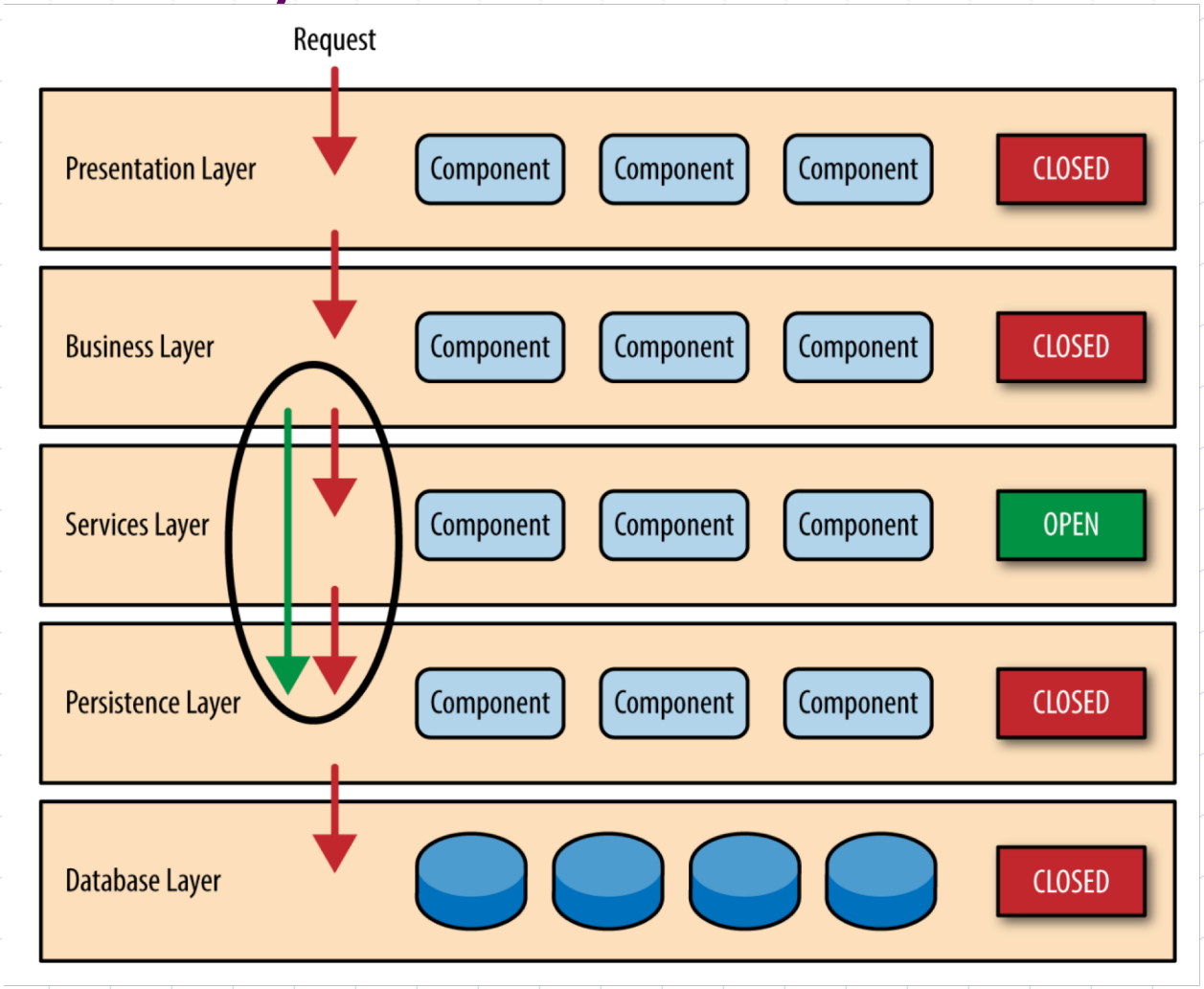
- Architecture Styles **Define** and **Constrain** the:
  - The **components** of the solution
    - What are the major computational units of the solution
    - Eg., Security, Data Access, Protocol Handling, …
  - The **connectors** of the solution
    - How are the interactions between the components realized
    - Eg., procedure calls, network calls, events, broadcasts, …
  - The **properties** of the solution
    - What are the quality attributes and non-functional requirements
    - Eg., pre/post conditions, availability specifications, interface semantics, …

# Layered Style




Basic Principle is that you can only interact in one direction with an adjacent layer

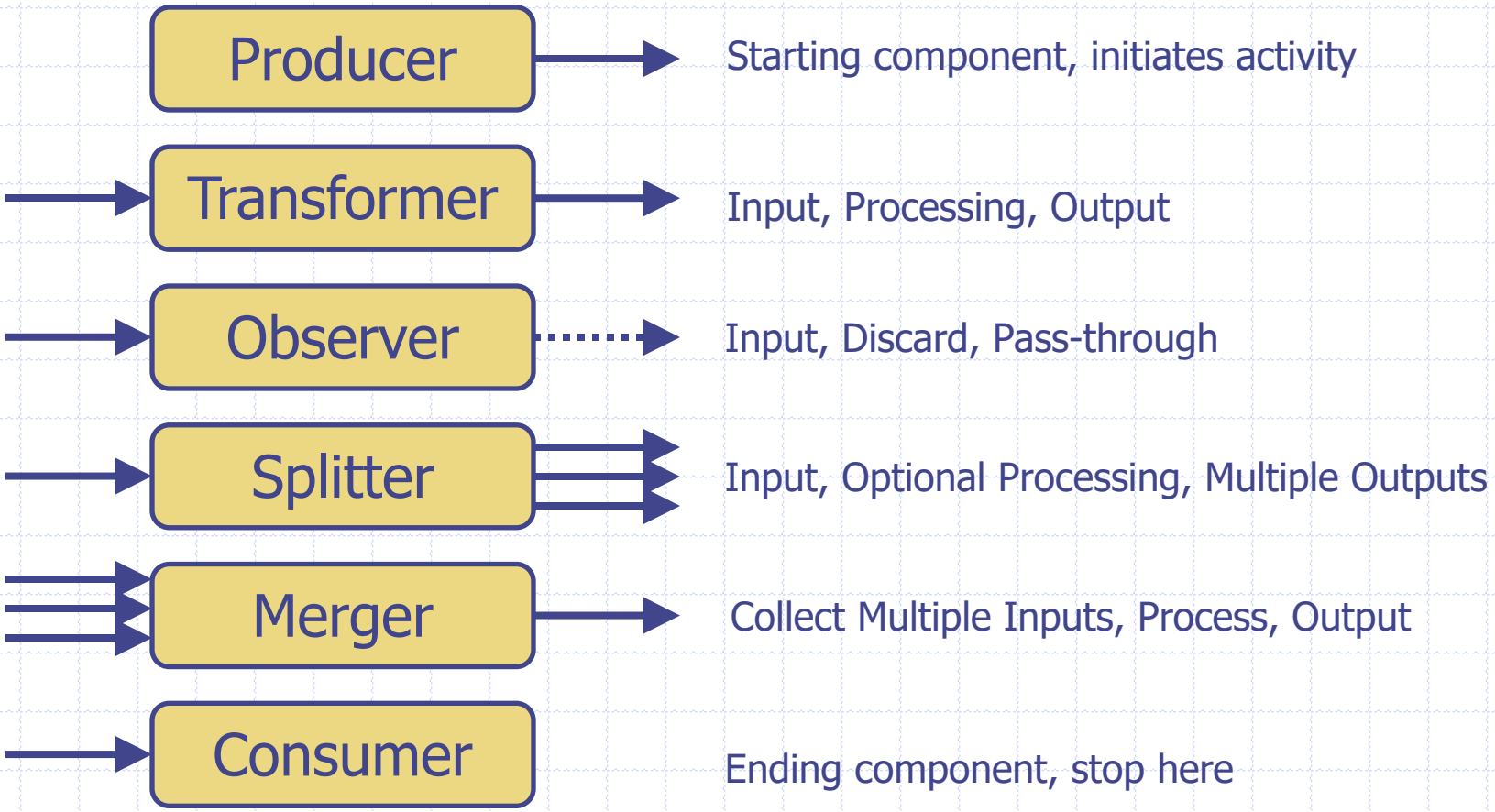
# Layered Style




Sometimes Layers are Open To Bypassing

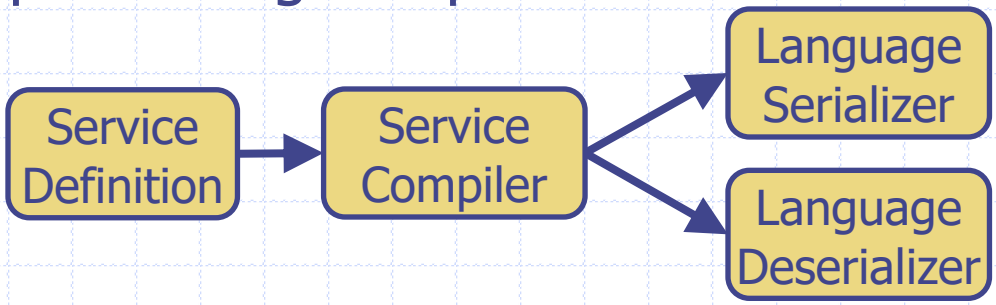
# Pipe and Filter

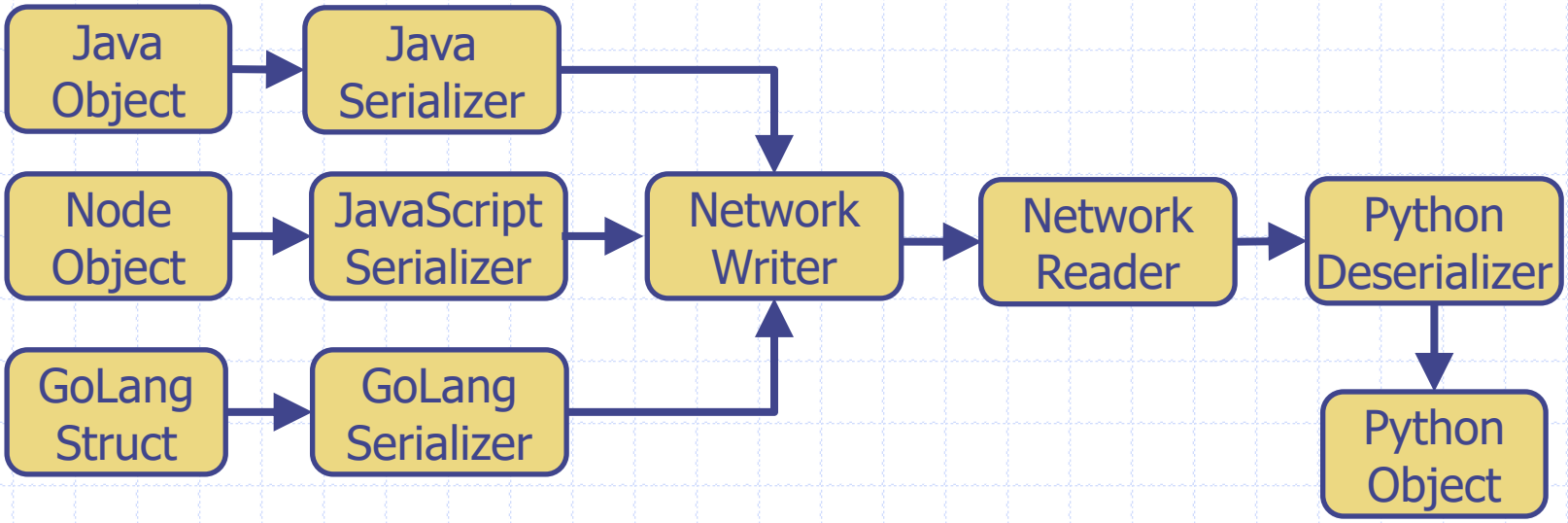
The pipe & filter architecture style is used to wire together processing components



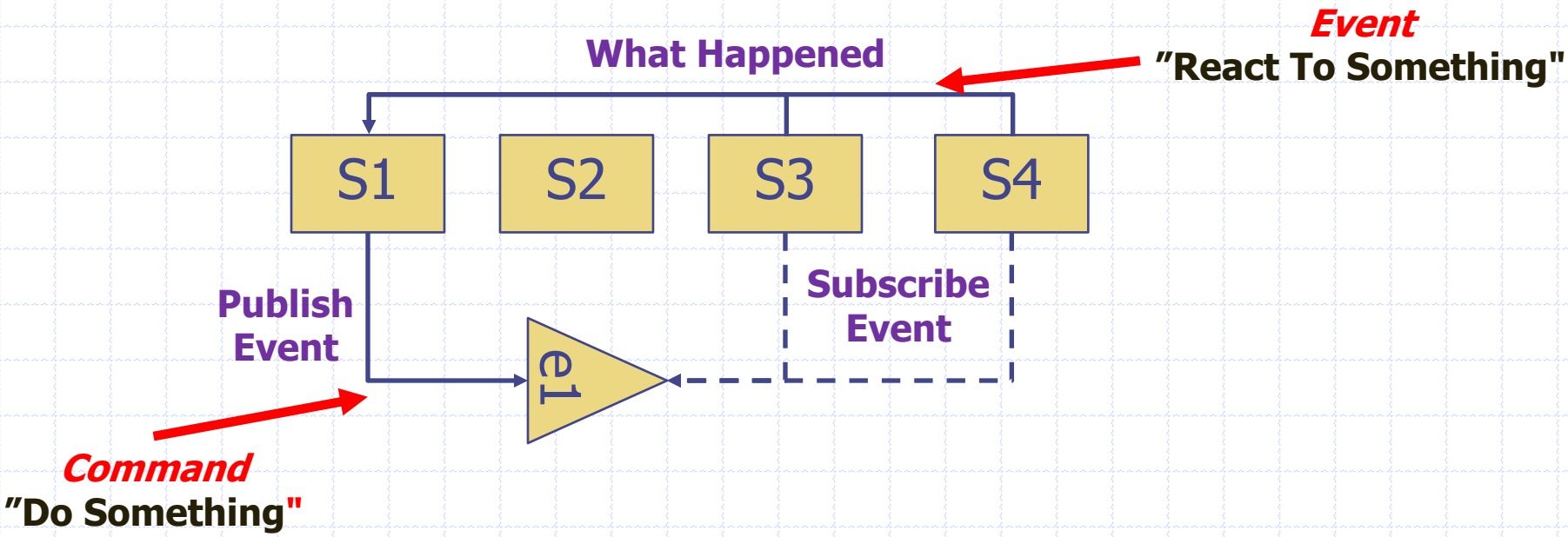
# Pipe and Filter

The pipe & filter architecture style is used to wire together processing components



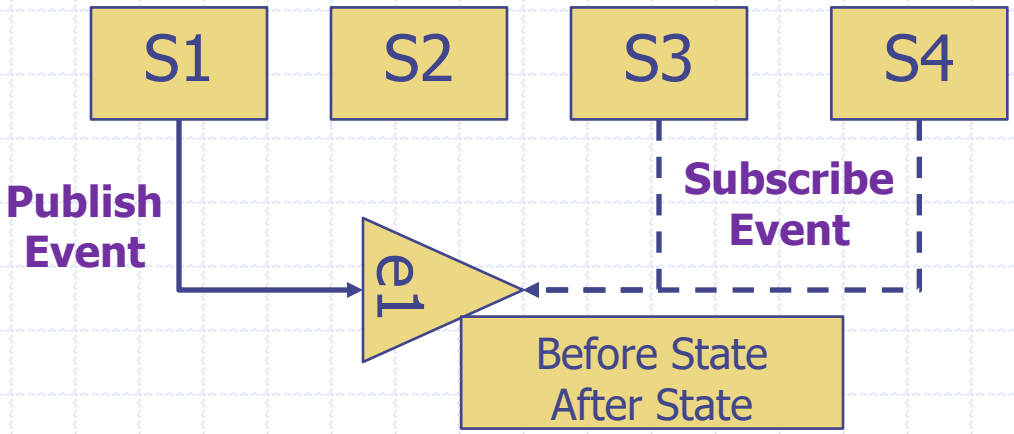
# Event Architectures – Event Notification




Event Notification
Loosen coupling with pub/sub patterns
Interest is only in if the event has occurred

Source: The Many Meanings of Event-Driven Architecture – Martin Fowler

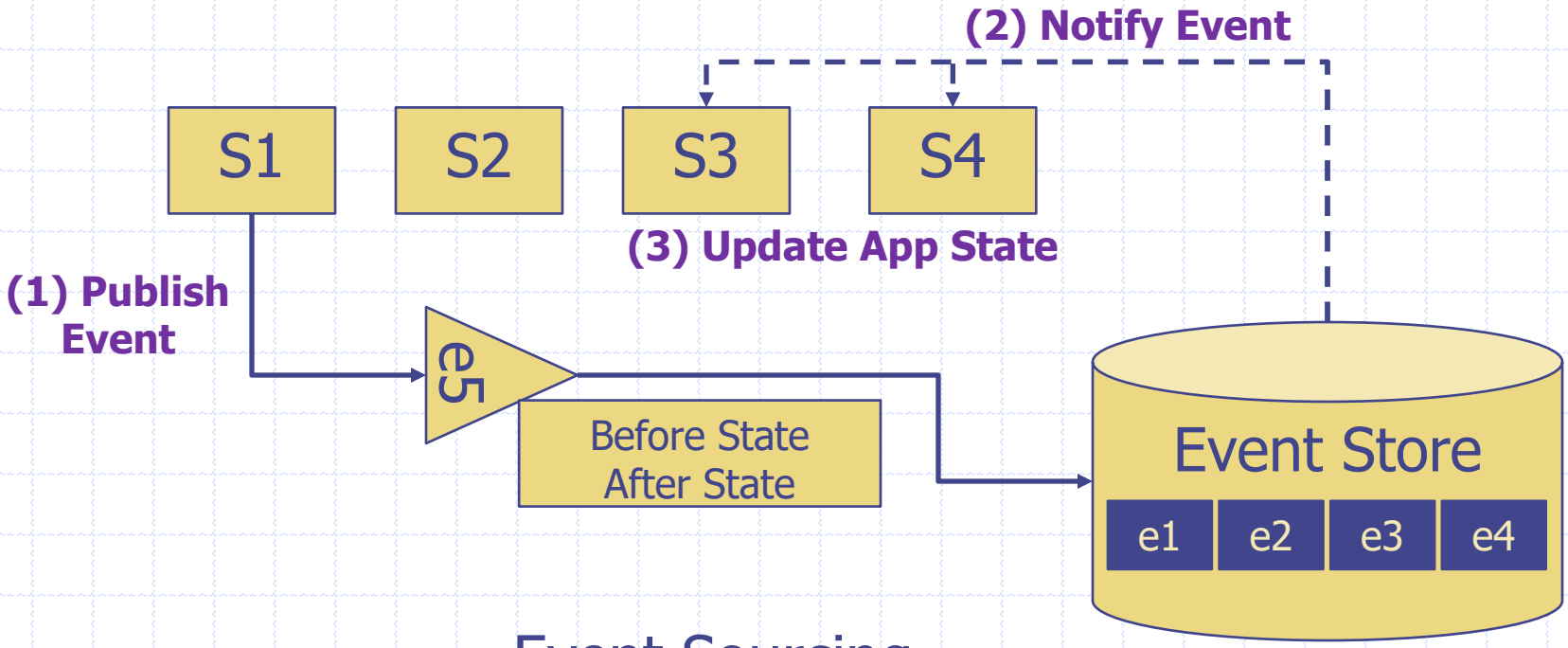
# Event and Streaming Architectures




Event Carried State Transfer
Loosen coupling with pub/sub patterns
Eventual Consistency

**Source: The Many Meanings of Event-Driven Architecture – Martin Fowler**

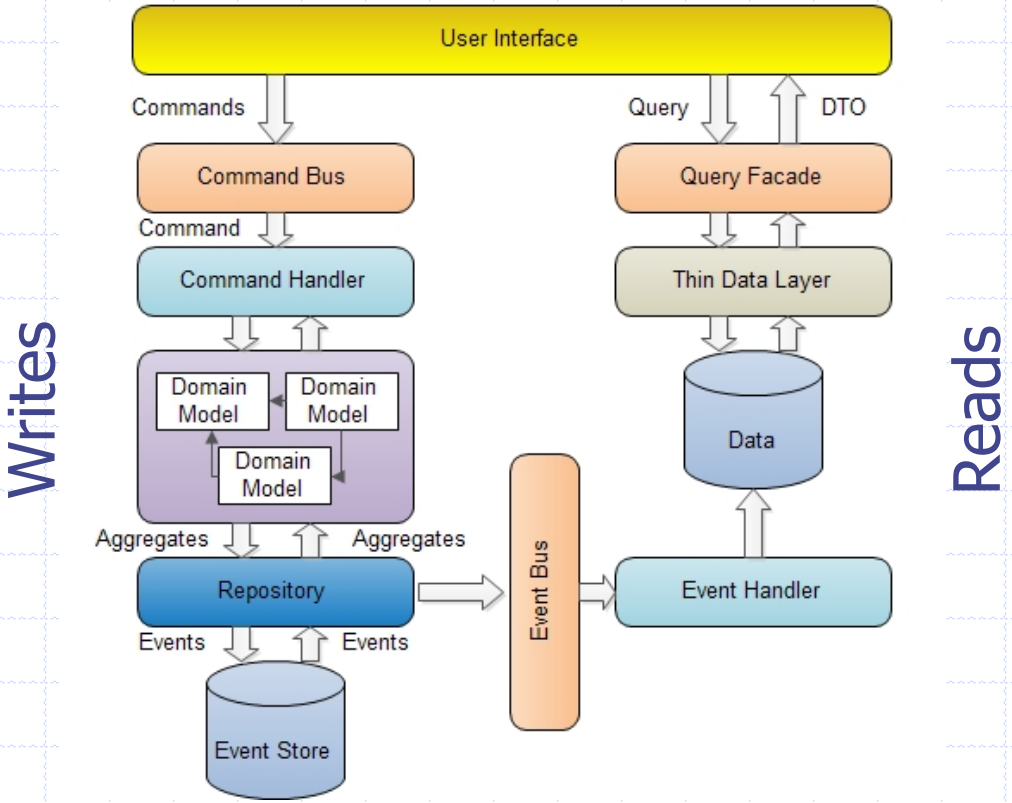
# Event – Event Carried State Transfer




Event Sourcing
Event Store Maintains State Over Time
Loosens Coupling

**Source: The Many Meanings of Event-Driven Architecture – Martin Fowler**

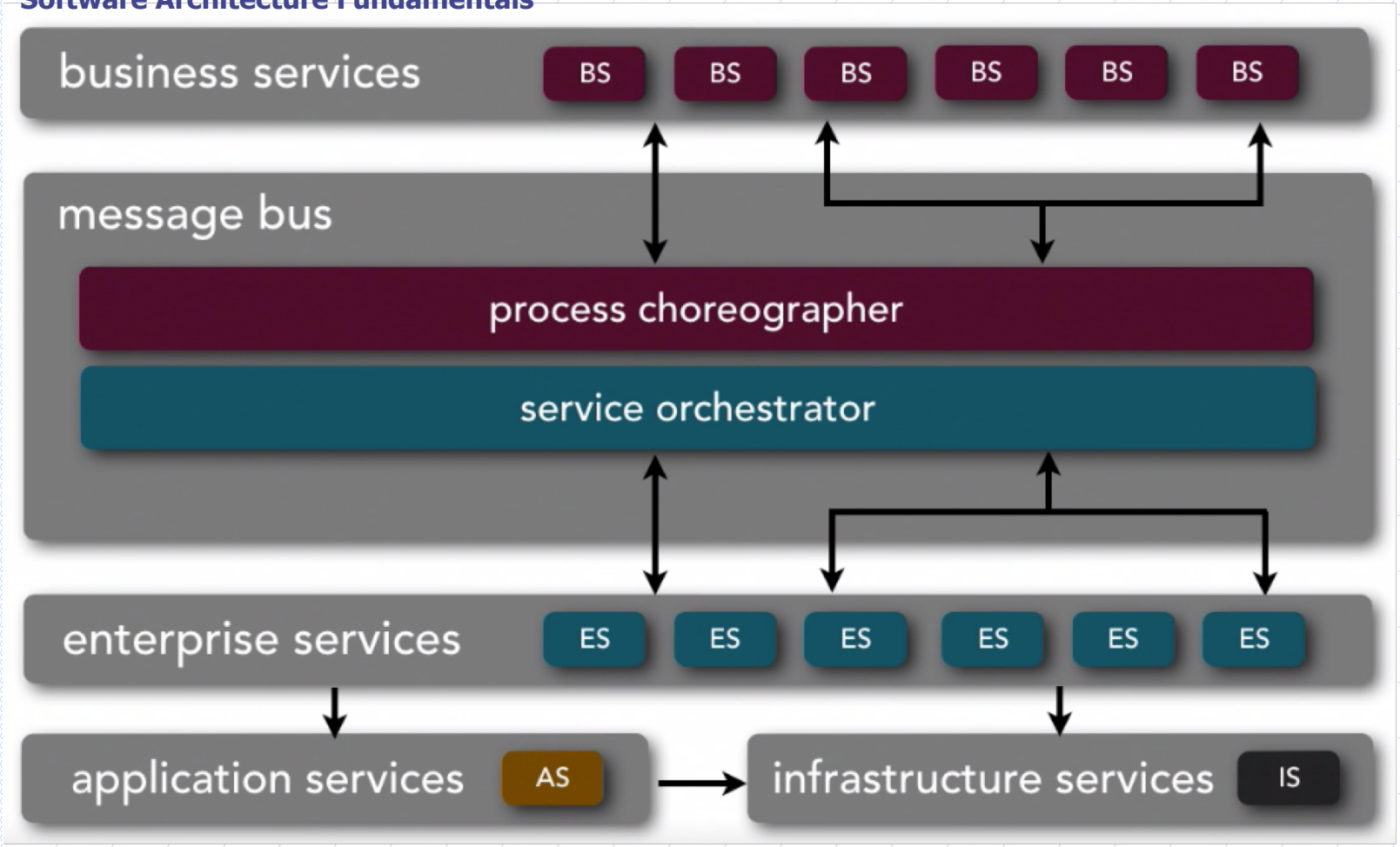
# Event – Command Query Responsibility Seperation




Scalability and Consistency via Separate Handling of Reads and Writes (CQRS)

# Service Oriented Architecture Style

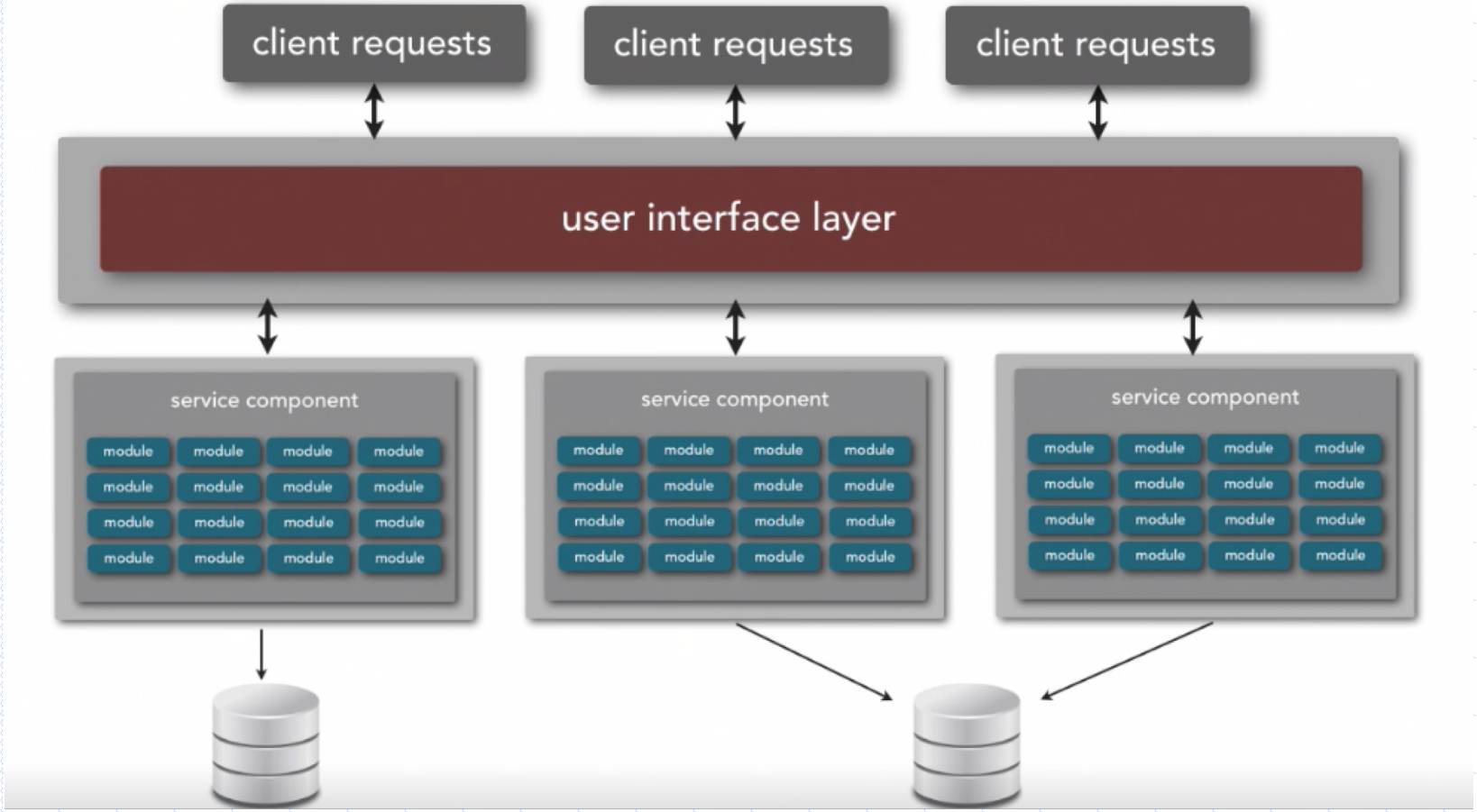
**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**




Thoughts? Good Parts? Bad Parts?

# Service Based Style

**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**




Works Well To Partition Functionality

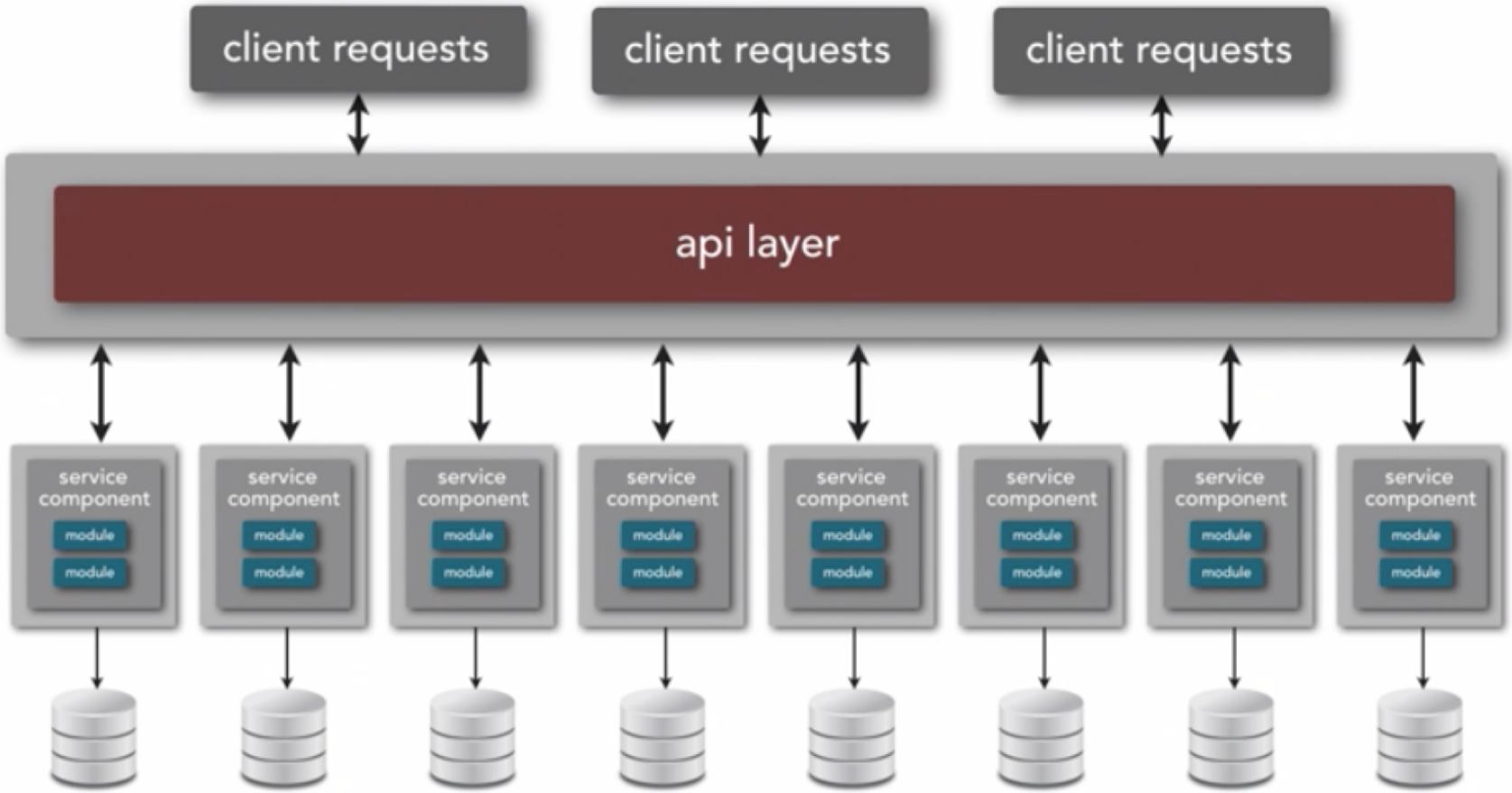
# Service Based Style - Example

**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**

TBD

# Microservice Architecture

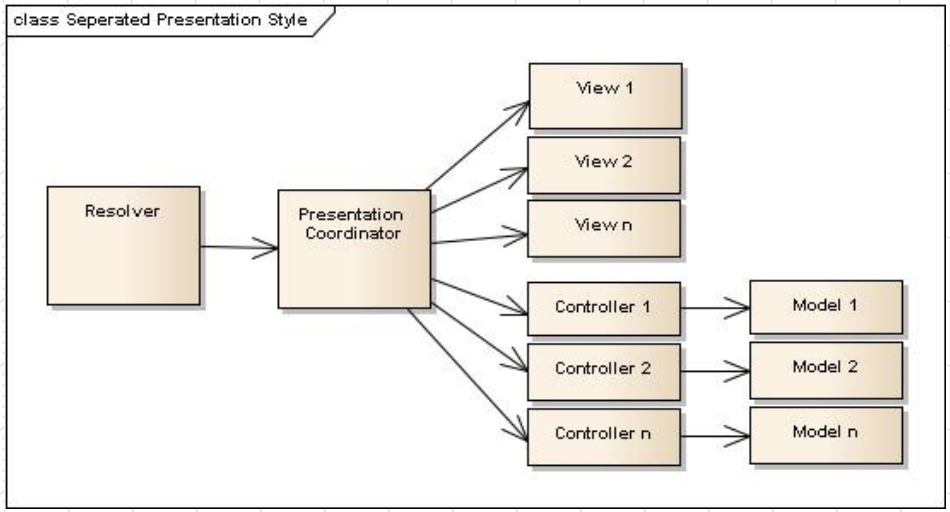
**Ref: Neil Ford & Mark Richards**
**Software Architecture Fundamentals**




Basic Idea – SHARE NOTHING

# Lets Look at an Example of Putting these things into Practice

# Example Architecture Style: Layered Style

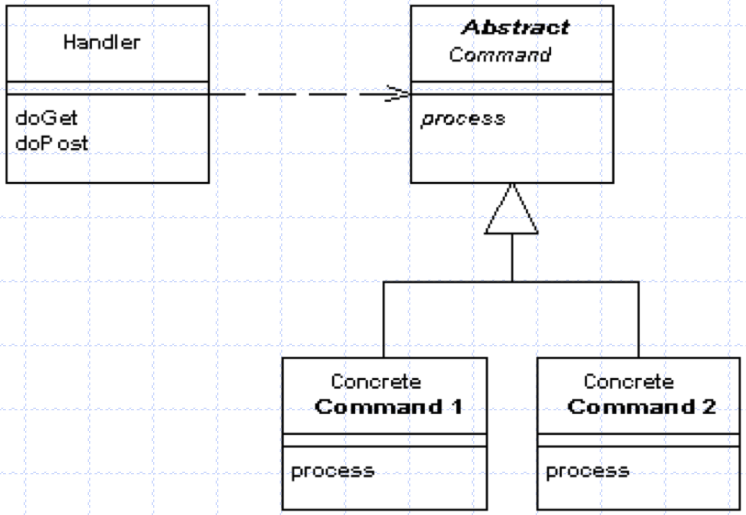



- Suitable for applications that involve distinct classes of services that can be **organized hierarchically**.
- Each layer provides service to the layer above it and serves as a client to the layer below it.
- Only carefully selected procedures from the inner layers are made available (exported) to their adjacent outer layer.

# Architecture Patterns

- Architecture Patterns can be thought of an **instance** of an architecture style that defines the specific components, connectors, and properties used to describe the overall architecture of a (sub)system
- Generally architecture patterns have a generic description and one or more specializations

# Example Architecture Pattern: Generic Front Controller Pattern

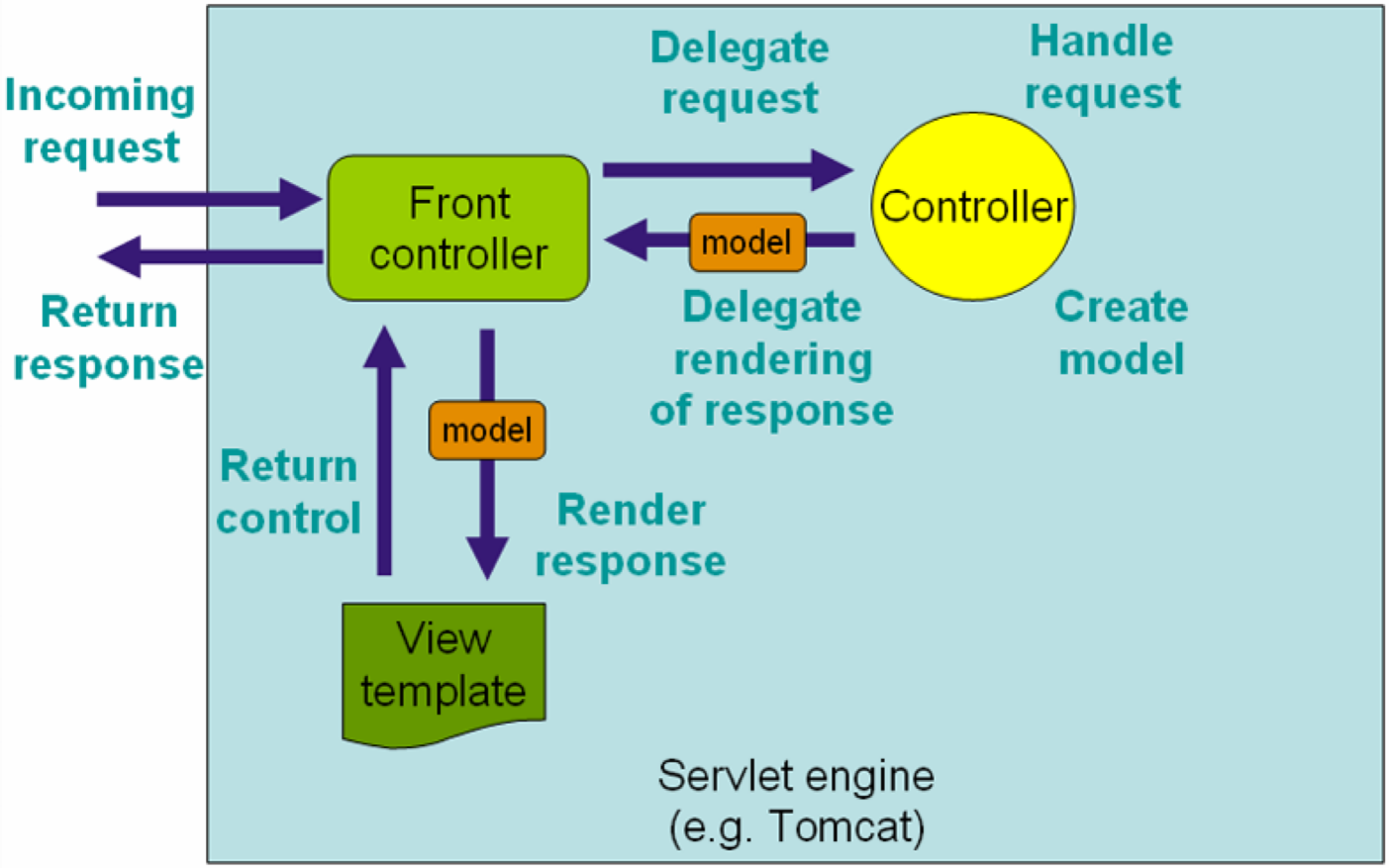



**From: http://www.martinfowler.com/eaaCatalog/frontController.html**

In a complex Web site there are many similar things you need to do when handling a request. These things include security, internationalization, and providing particular views for certain users. If the input controller behavior is scattered across multiple objects, much of this behavior can end up duplicated. Also, it's difficult to change behavior at runtime.

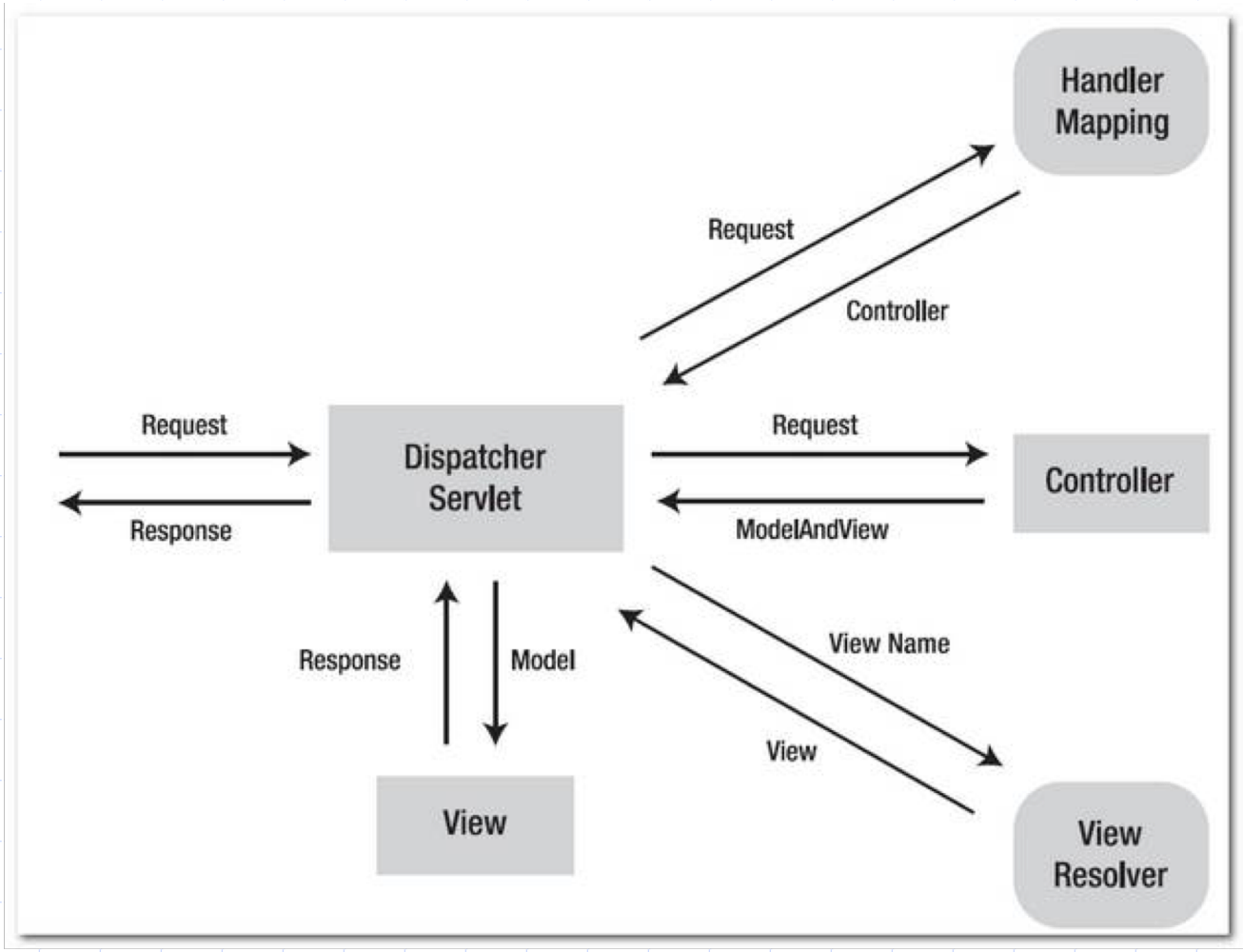
The Front Controller consolidates all request handling by channeling requests through a single handler object. This object can carry out common behavior, which can be modified at runtime with decorators. The handler then dispatches to command objects for behavior particular to a request.

# Example: Front Controller Pattern for a Web Application




**Using the Front Controller Pattern to Modularize Web Applications (e.g., Spring Framework)**

# Example Front Controller: Spring MVC Framework

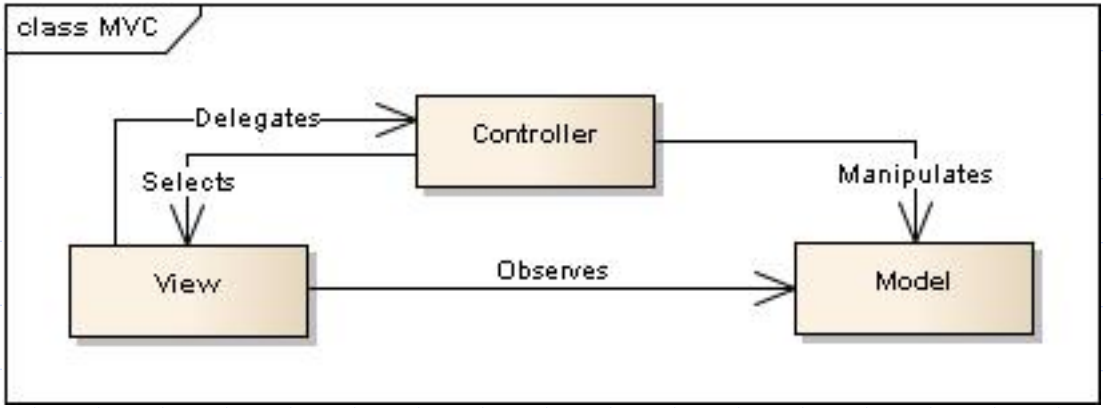


# Design Patterns

- Design Patterns can be thought of as **building blocks** of an architecture pattern that defines the specific code-level components and connectors used to implement the architecture pattern
- An architecture pattern tends to be realized through one or more design patterns

# Is our MVC pattern or its specialization Front Controller a Design Pattern?

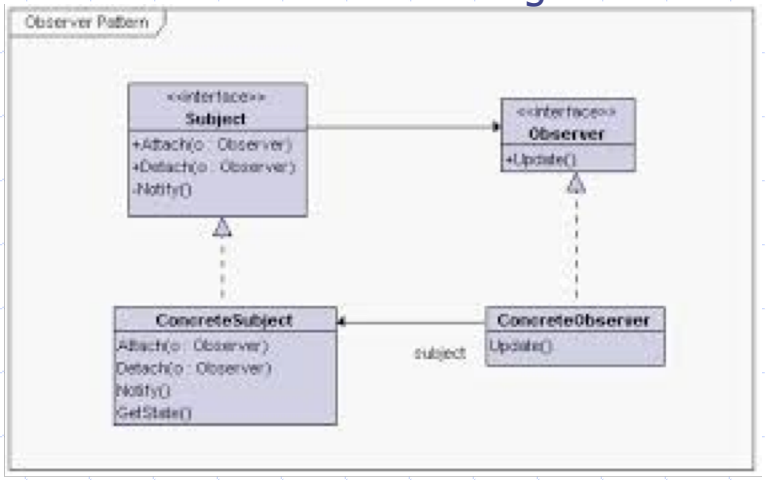
- People have different opinions, but mine is NO, MVC and its derivatives are not sufficient to structure a software design

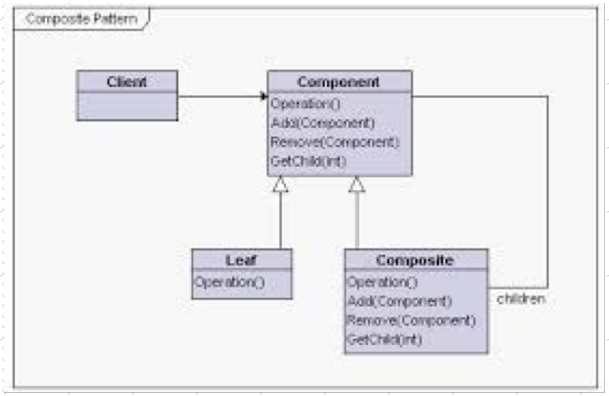


# Design Patterns for MVC

## MVC ?= Observer+Strategy+Composite Pattern

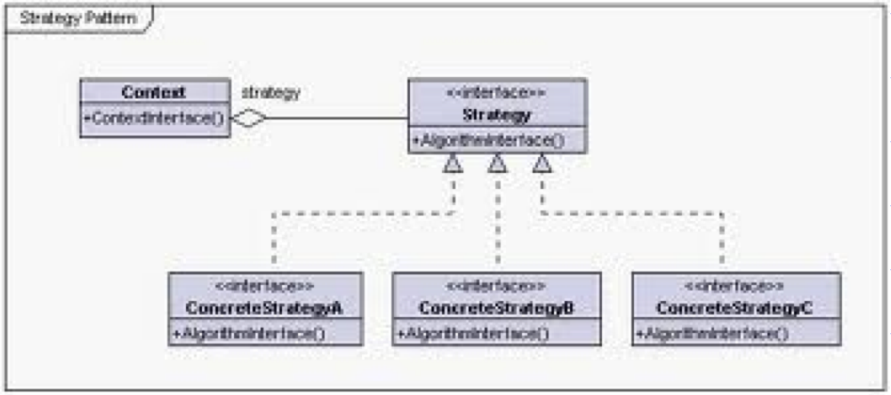
Observer: Have central object notify other objects of changes that they are interested in observing



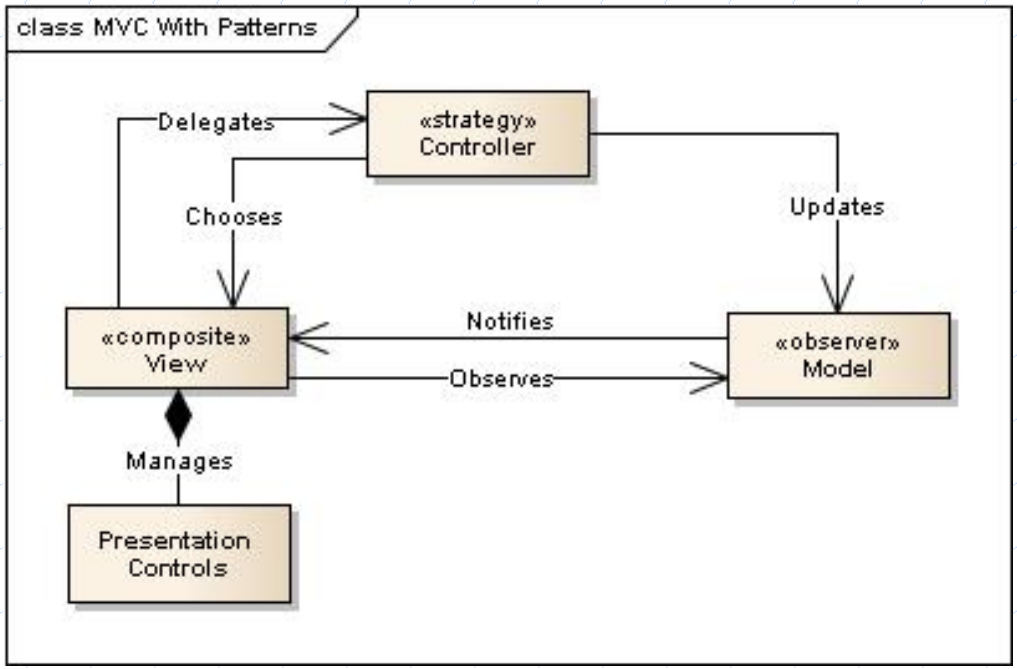

Composite: Treat tree-like structures uniformly…

Strategy: Abstraction to allow behavior to be selected at runtime



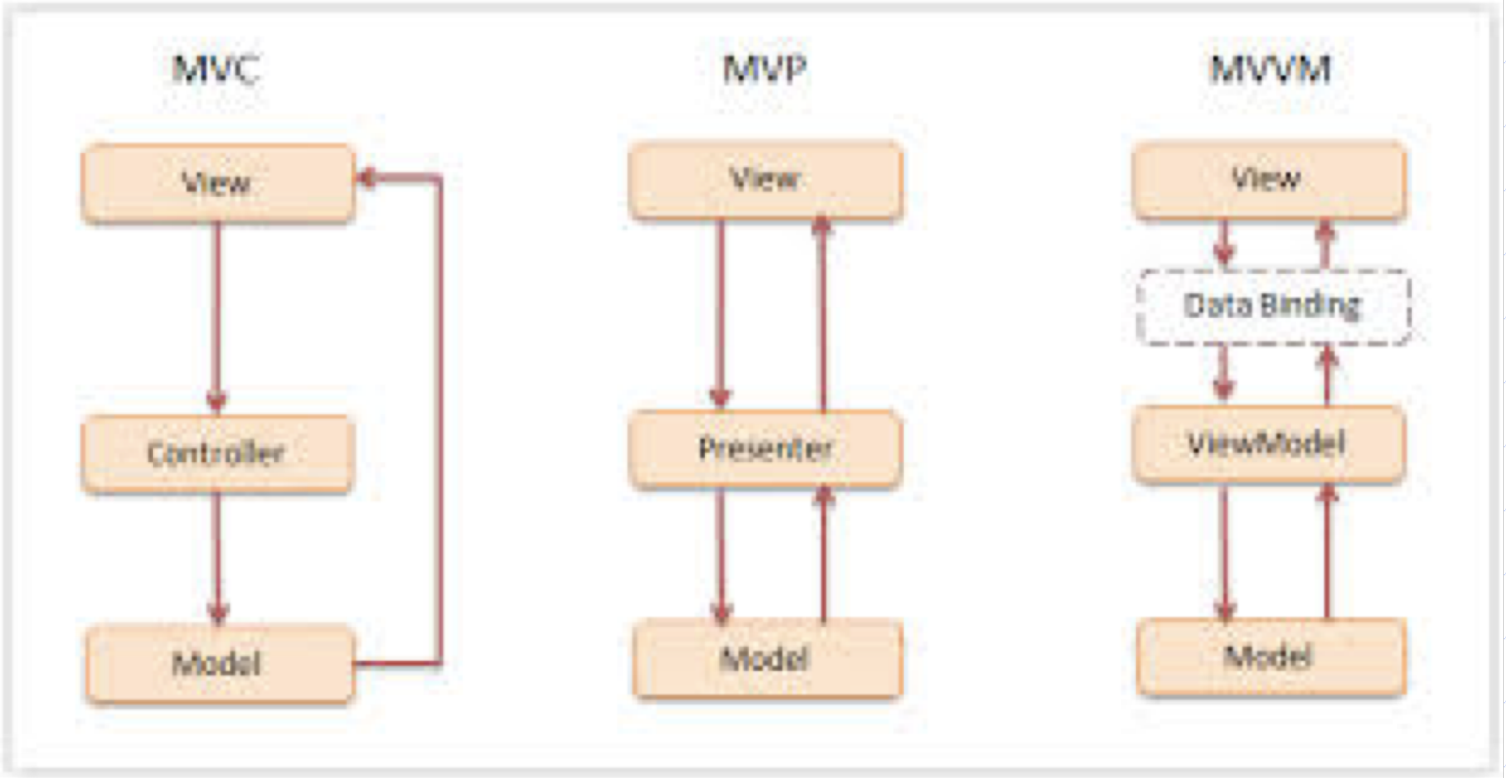
# Using Design Patterns to Realize an Architecture Pattern?

- Model uses an **Observer** pattern to notify the view of changes
- Controller uses a **Strategy** pattern to manage which view should be rendered
- View uses a **Composite** pattern to manage the hierarchy of presentation controls




Note that MVC can be implemented using other design patterns as well, this is just a sensible representation to show one way that this can be accomplished

# MVC Variations – Like Most Patterns MVC has Specializations

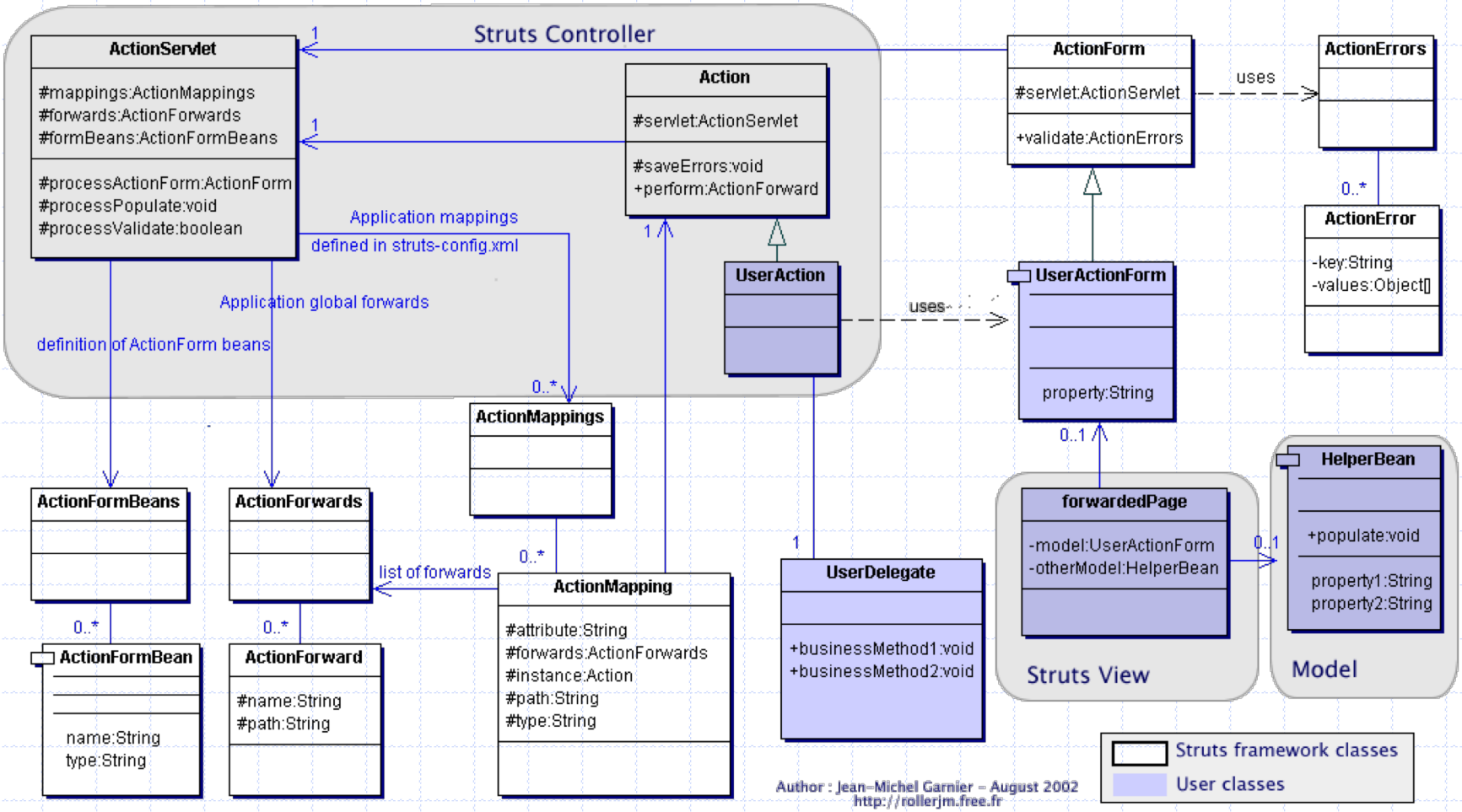



Controller facilitates Binding the model to the view

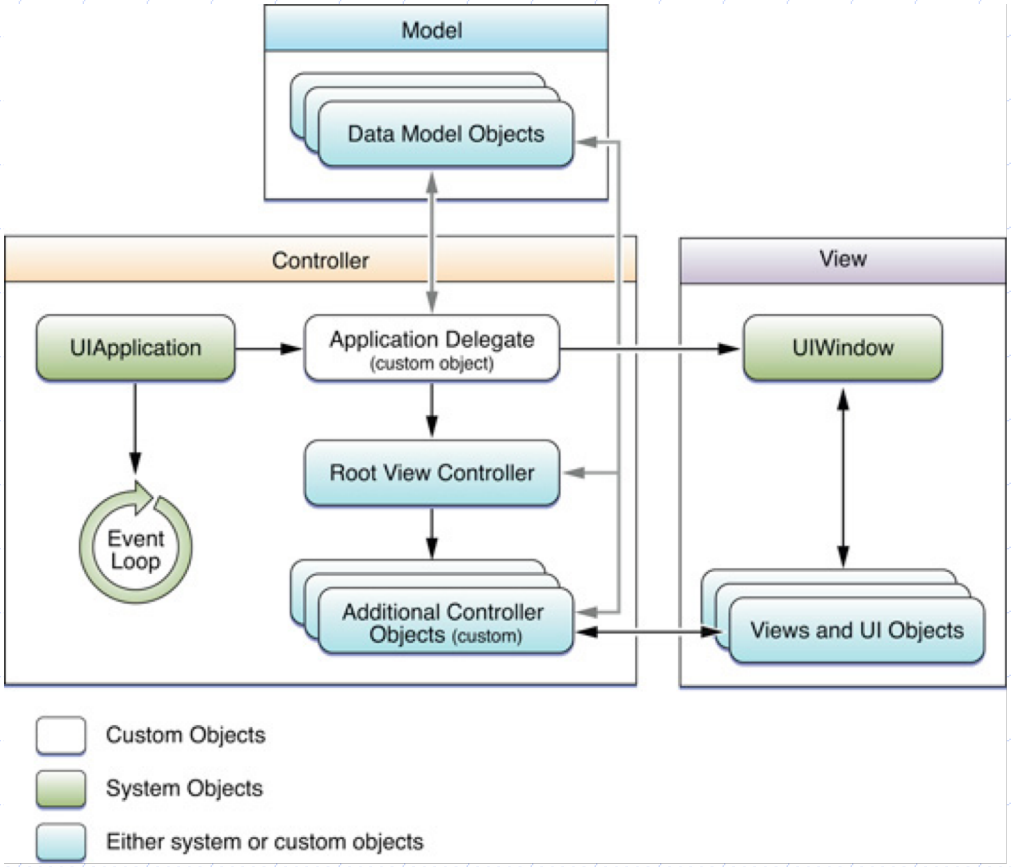
Presenter proxies the model for the view

ViewModel binds to one or more views. Generally data binding is bi-directional

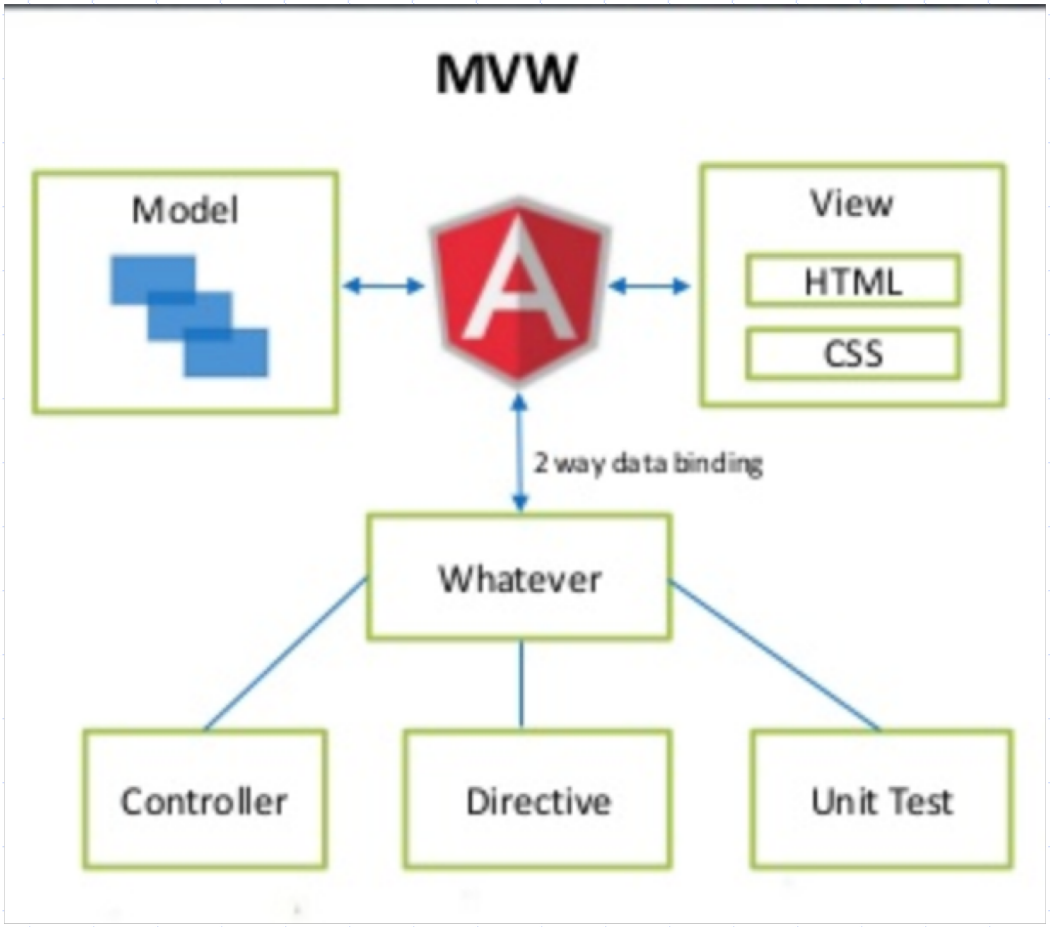
# Example: Jakarta Struts Presentation Framework – A Design of an MVC



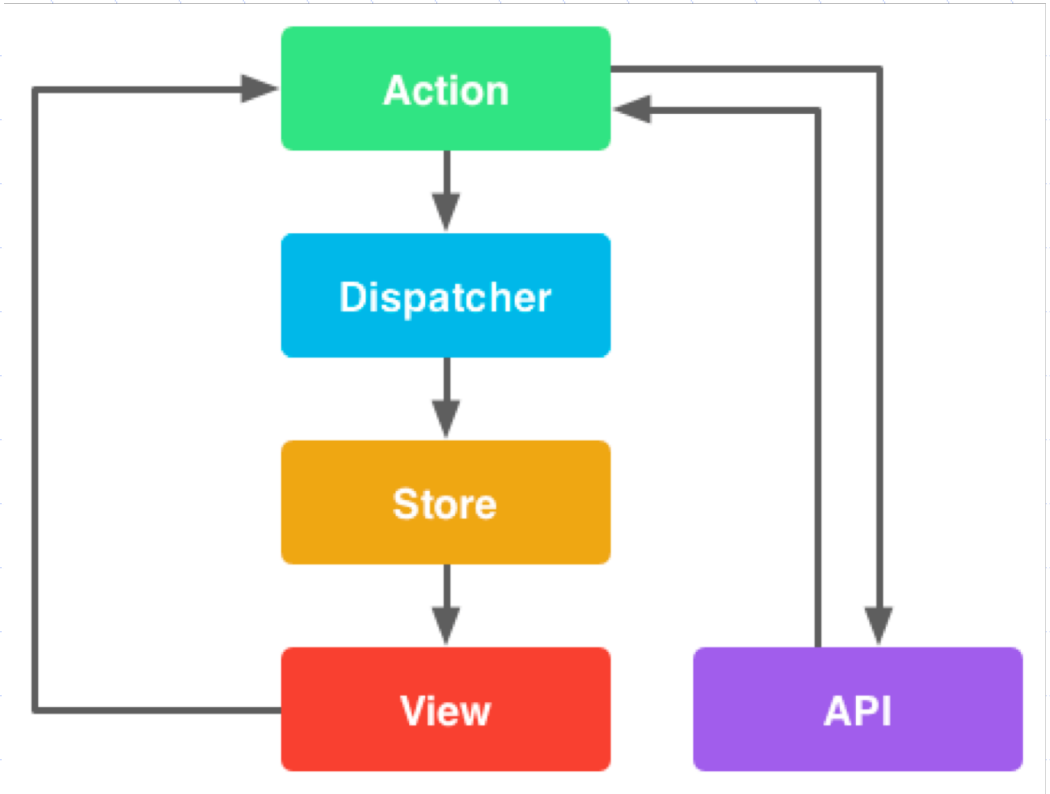
# Example: IOS Cocoa Framework MVC



# Example: Angular - MVW

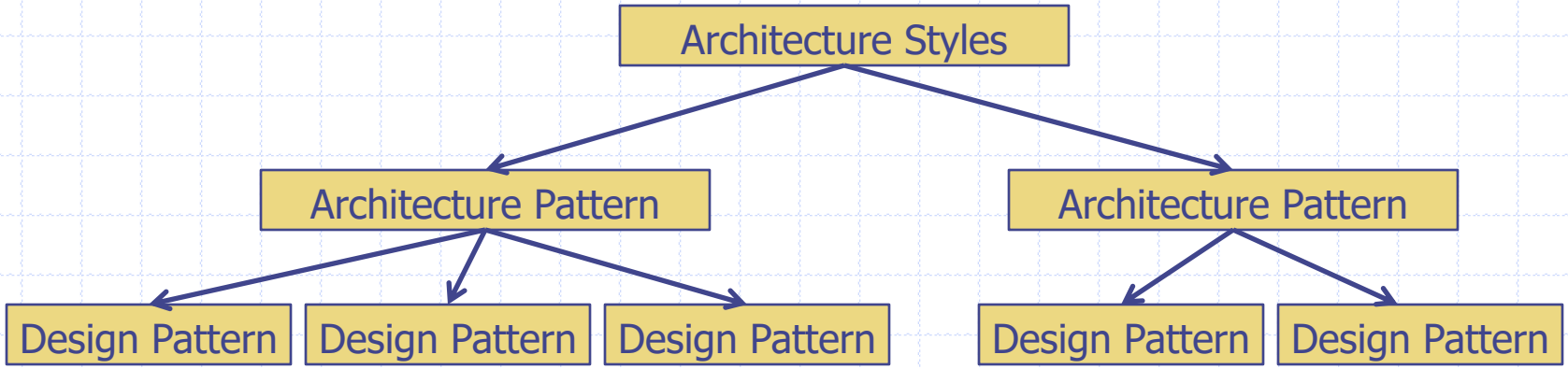


# Example: React.js Architecture



# Patterns - Summary

- Architecture styles define a vocabulary for the overall software architecture (and can be specialized)
- Architecture patterns define the components and connectors used to organize the software architecture (and can be specialized)
- Design patterns define the code-level assets and their associations.




As I mentioned earlier, conceptually combining Architecture Styles and Patterns is generally OK.

# What's good about Patterns

- They solve common problems in a "proven" way
- They tend not to be implementation specific
- They tend to be classified in a common way – context, forces, examples, etc
- They embody good design principles
  - Example: Loose Coupling

# Design Quality

- Software design “quality”, as with other ideas on quality, is an elusive concept:
- It depends on priorities of your company and the customers:
  - fastest to implement
  - easiest to implement
  - easiest to maintain, “evolve”, port
  - most efficient/reliable/robust end-product.

# Good Design Properties

- **Hierarchical**: A good design should be organized into a well-designed hierarchy of components.
- **Modular**: Separate distinct concerns (dataand processing) into distinct containers (i.e.,subsystems, modules, and/or classes). Hide implementation details and provide clean, simple interfaces for each container.

# Good Design Properties

- **Independent**: Group similar things together; limit the amount of "special knowledge" that unrelated components may share. If you change your mind about something, the impact will be *localized*.

# Good Design Properties

- **Simple Interfaces:** Endless flexibility adds complexity. Complex interfaces mean:
  - hard to understand by users and developers (*e.g.,* Unix *man* page syndrome)
  - many possible variations of use
  - inconvenient to change interface in order to eliminate “bad options”.
- You can get away with “flexible interfaces” in a low-level localized setting, but the larger the scale, the simpler the interface should be.

# Summary: Software Design

- Good software designers are experienced software designers
  - Given a design, and experienced designer can tell you:
    - What's good, What's Bad, and provide suggestions for improvement
  - Patterns and Frameworks are very helpful to software designers
- Good software designs are based on good design principles

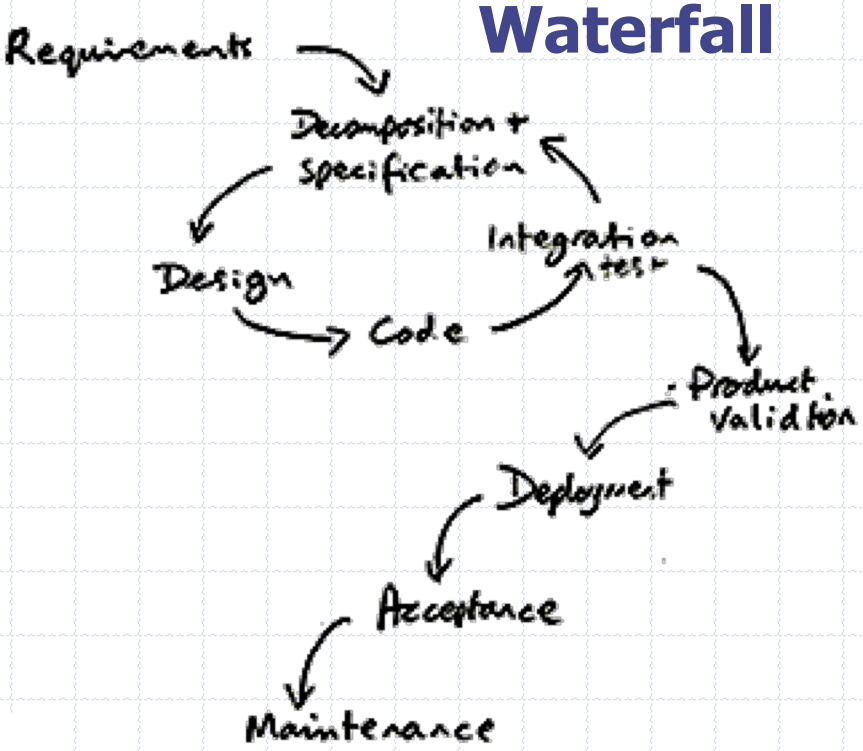
# References

- Some materials derived from Dr. Tim Leftbridge's Software Engineering Class Notes - http://www.site.uottawa.ca/~tcl/seg2105/ (specifically Chapter 9)

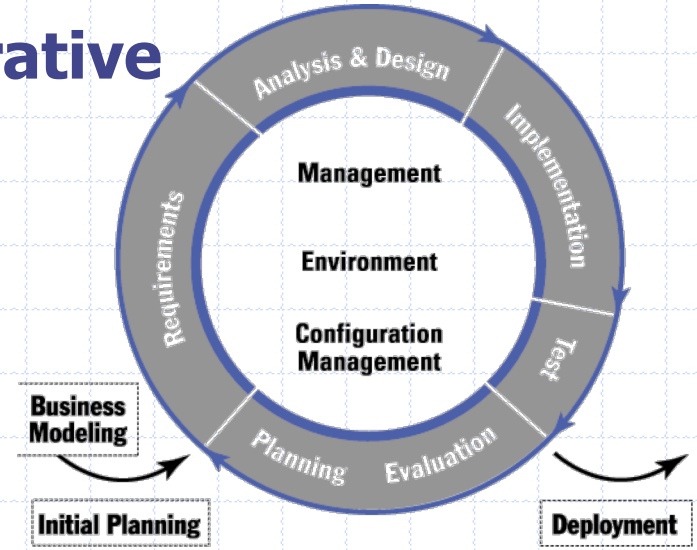
# The Slides after Here Are Likely Going to be Deleted

# x-Design Methodologies- Dealing with Complexity

**Waterfall**




**Iterative**



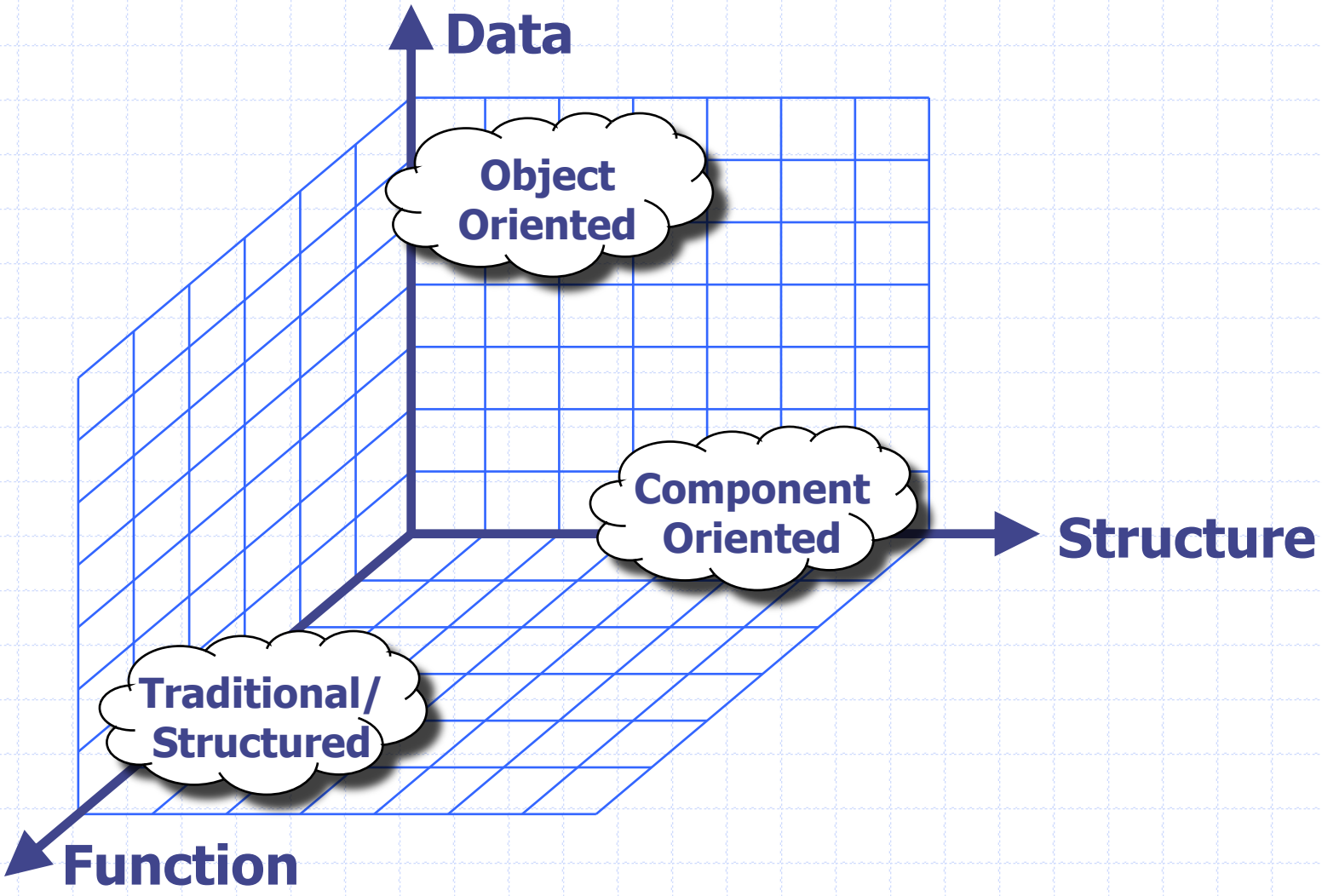


**Agile/Scrum**

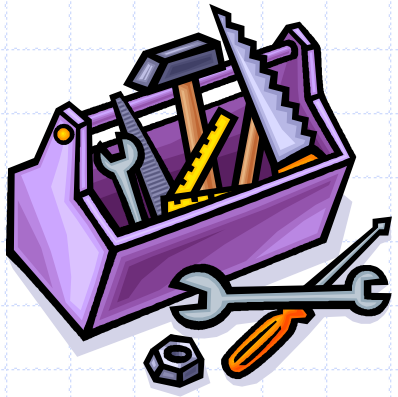
# x-Software Development Methodology

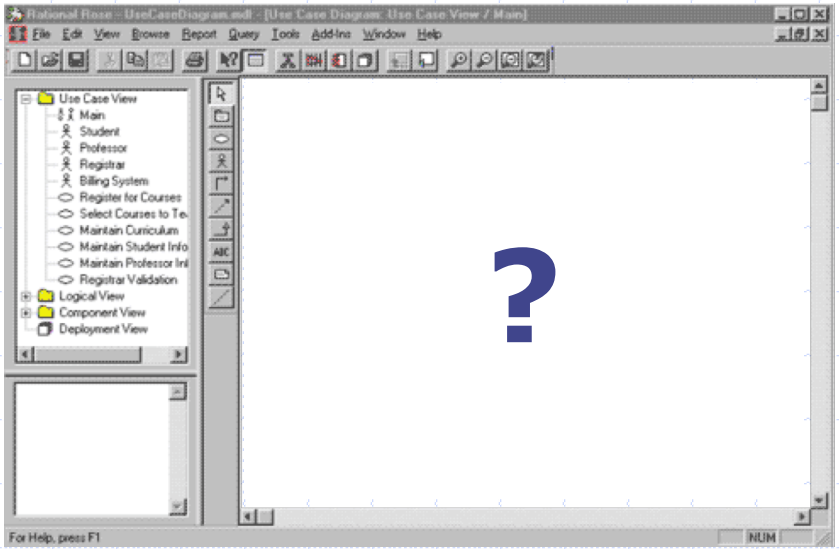
- A Methodology is:
  - A Process
    - What happens over time
    - Coupled to the Software Development Lifecycle (SDLC)
  - A Management Approach
    - What is needed to move from one step to another
- Helps with the project management aspects of a software development project

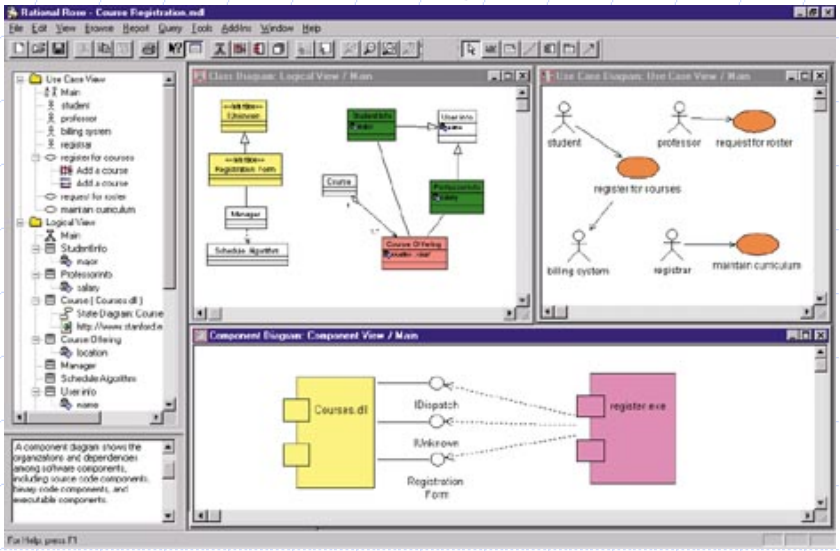
# X-Modeling Emphasis for Different Design Approaches



# X-Jumpstarting Design

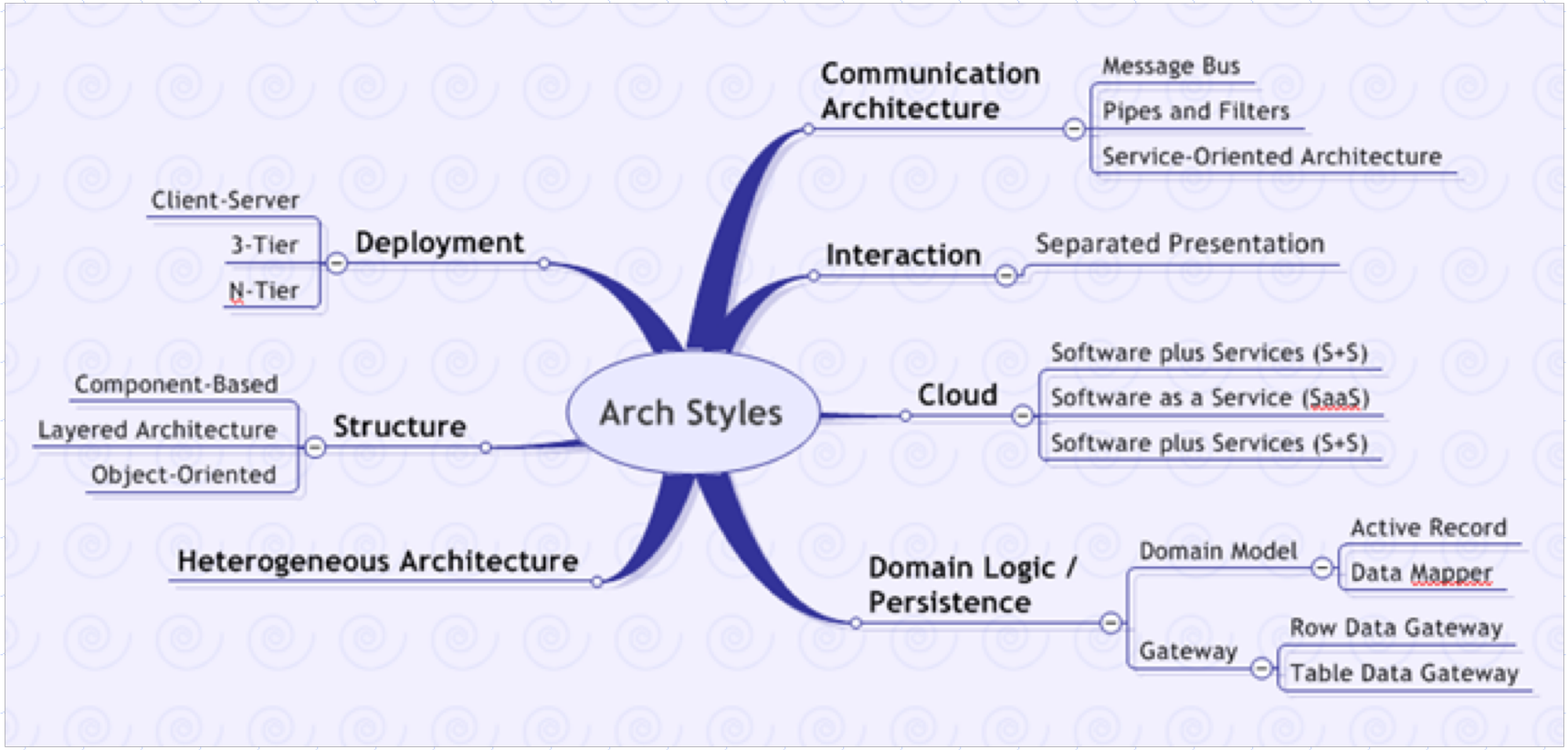




**Starting from scratch "blank screen" is tough. Components of a design toolbox – Templates, Patterns, Reference Architectures, Frameworks, and so on…**

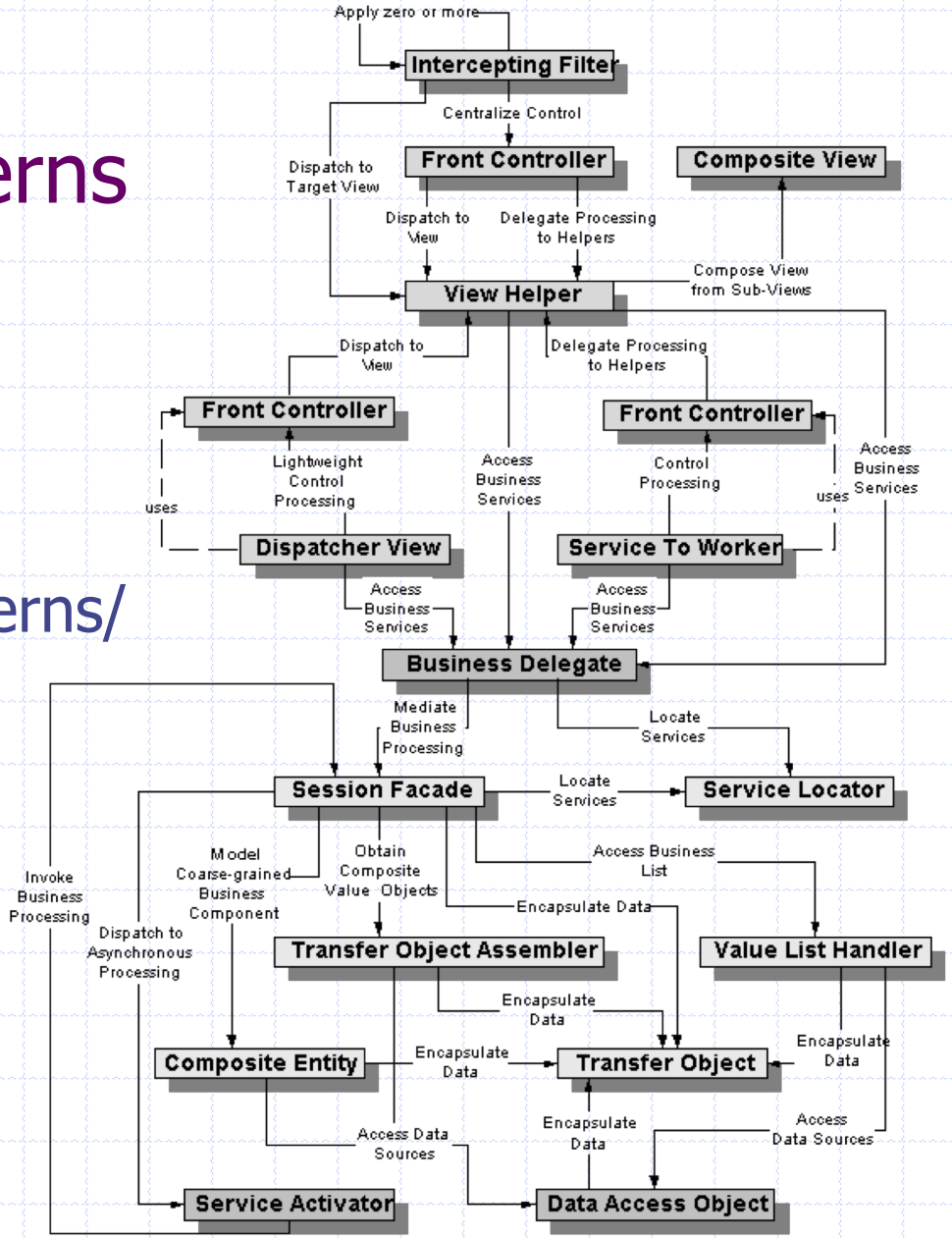
# Architecture Styles are generally categorized and can be specialized



# Example: J2EE Architecture Patterns

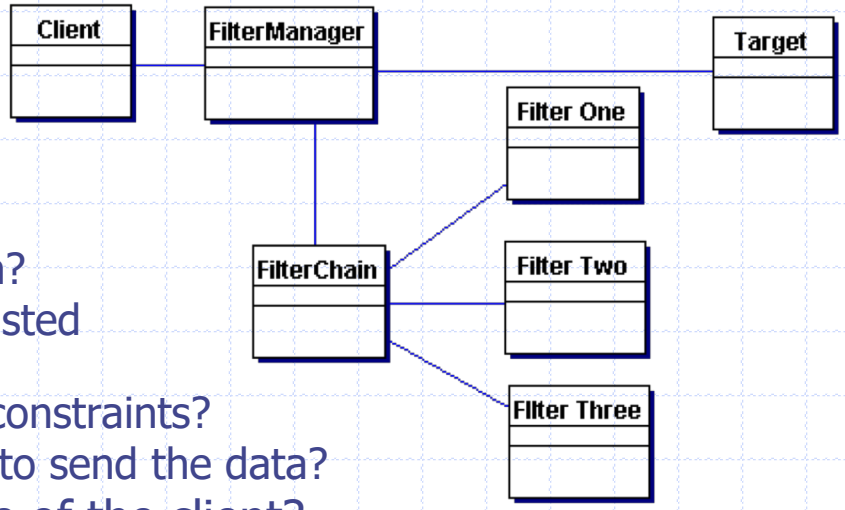
Source:

http://java.sun.com/blueprints/corej2eepatterns/Patterns/



# Example: Intercepting Filter Pattern Architectural Pattern

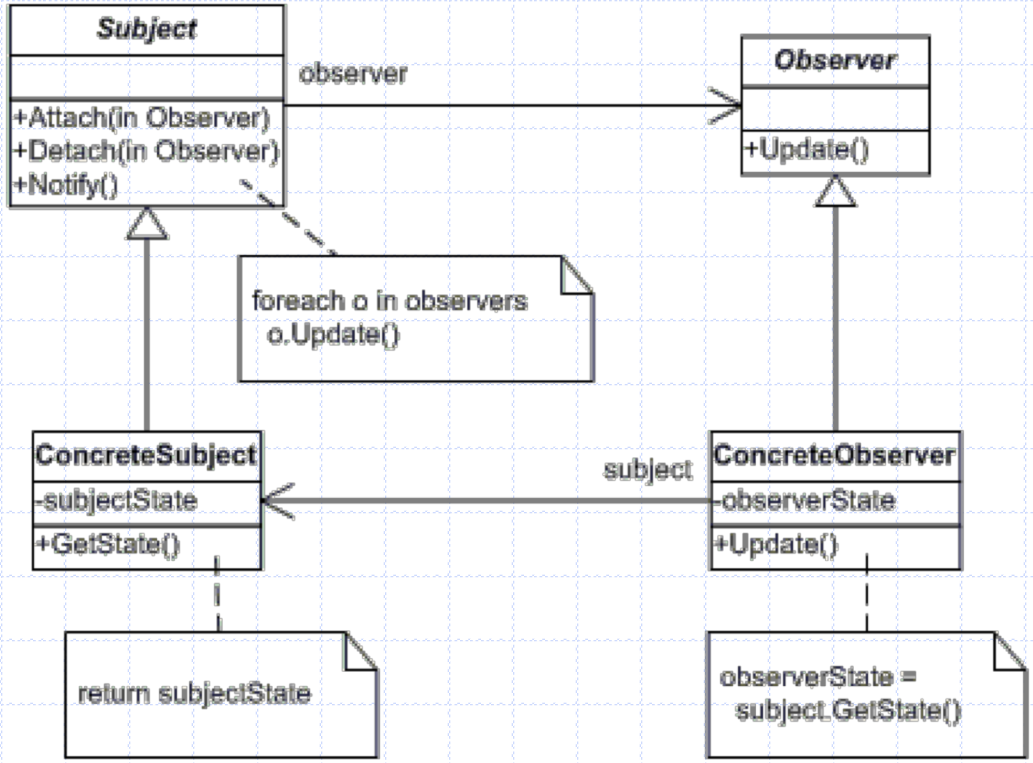
- Problem: Preprocessing and post-processing of a client Web request and response are required
  - When a request enters a Web application, it often must pass several entrance tests prior to the main processing stage:
    - Has the client been authenticated?
    - Does the client have a valid session?
    - Is the client's IP address from a trusted network?
    - Does the request path violate any constraints?
    - What encoding does the client use to send the data?
    - Do we support the browser type of the client?




- The key to solving this problem in a flexible and unobtrusive manner is to have a simple mechanism for adding and removing processing components, in which each component completes a specific filtering action.

# Design Patterns

- Example: Observer Pattern
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically



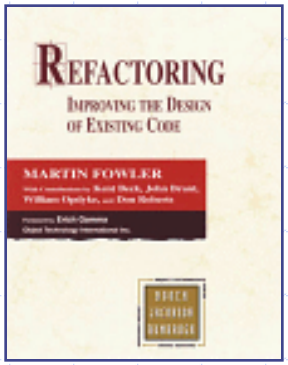

**Note: Design pattern references often include sample code in a variety of languages to illustrate how to use the pattern.**

# How to “Fix” A Software Design

- Many times the source code is the only up-to-date documentation for a software system
  - Must be able to recover the design to some extent
  - Must be able to “improve” the design where appropriate

# Improving Existing Designs - Refactoring

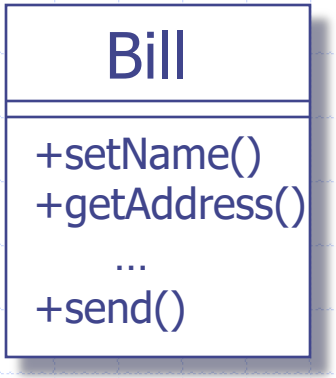
- See Martin Fowler's Book
- What is Refactoring:
  - Refactoring is a technique to restructure code in a disciplined way
  - Used to improve a system design in any number of ways
  - Pattern/Template based process
  - Automated tools exist



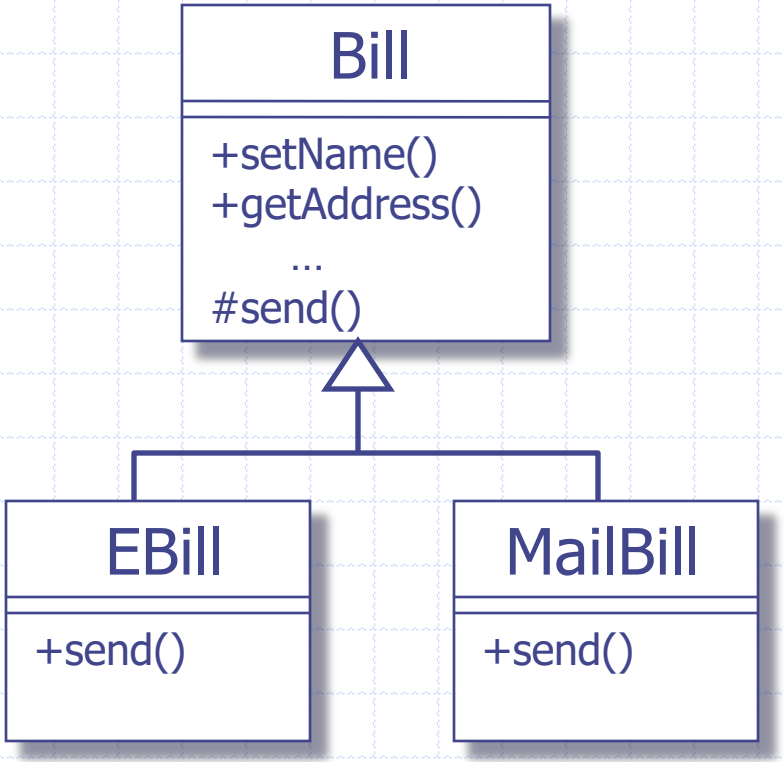

Refactoring is a behavior preserving transformation of the source code…

# Example: Refactoring – PushDown Method

**Before**




**After**




***Note: There are many ways to do this type of refactoring – this is just one example.***

# Anti-Patterns (plus Refactoring)

- AntiPatterns are Negative Solutions that present more problems than they address
- AntiPatterns are a natural extension to design patterns
- AntiPatterns bridge the gap between architectural concepts and real-world implementations.
- Understanding AntiPatterns provides the knowledge to prevent or recover from them
  - Recovery can be via Refactoring

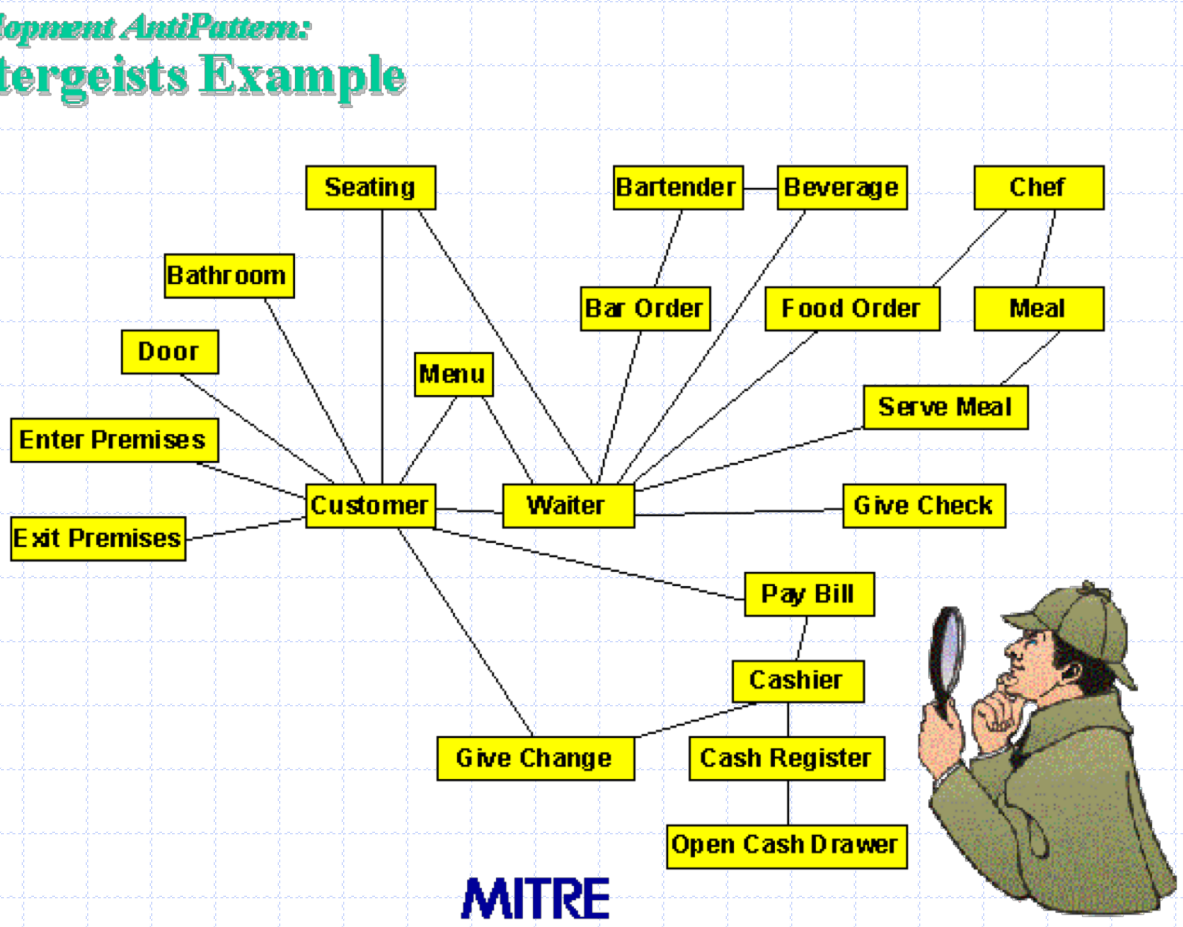
From: www.antipatterns.com

# Antipattern Example: Poltergeists

- Proliferation of classes [Riel 96]
- Spurious classes and associations
  - Stateless, short-lifecycle classes
  - Classes with few responsibilities
  - Transient associations
- Excessive complexity
- Unstable analysis and design models
- Analysis paralysis
- Divergent design and implementation
- Poor system performance
- Lack of system extensibility

From: www.antipatterns.com

# Antipattern Example: Poltergeists




From: www.antipatterns.com

# Antipattern Example: Fixing Poltergeists

- Refactor to eliminate irrelevant classes
  - Delete external classes (outside the system)
  - Delete classes with no domain relevance
- Refactor to eliminate transient "data classes"
- Refactor to eliminate "operation classes"
- Refactor other classes with short lifecycles or few responsibilities
  - Move into collaborating classes
  - Regroup into cohesive larger classes

From: www.antipatterns.com