

SE 577
Software Architecture

Architectural Styles

Acknowledgement

- Material from several sources including:
 - D. Garlan and M. Shaw, "An Introduction to Software Architecture", CMU-CS-94-166
 - R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory and Practice*, Wiley.
 - Farshidi, Jansen, and van der Wolf. Capturing software architecture knowledge for pattern-driven design. *Journal of Systems and Software*, 2020
 - Erder, Pureur, and Woods. Continuous Architecture in Practice
 - Richards and Ford. Fundamentals of Software Architecture
 - Carvantes and Kazman. Designing Software Architectures

Software Architecture - Ingredients

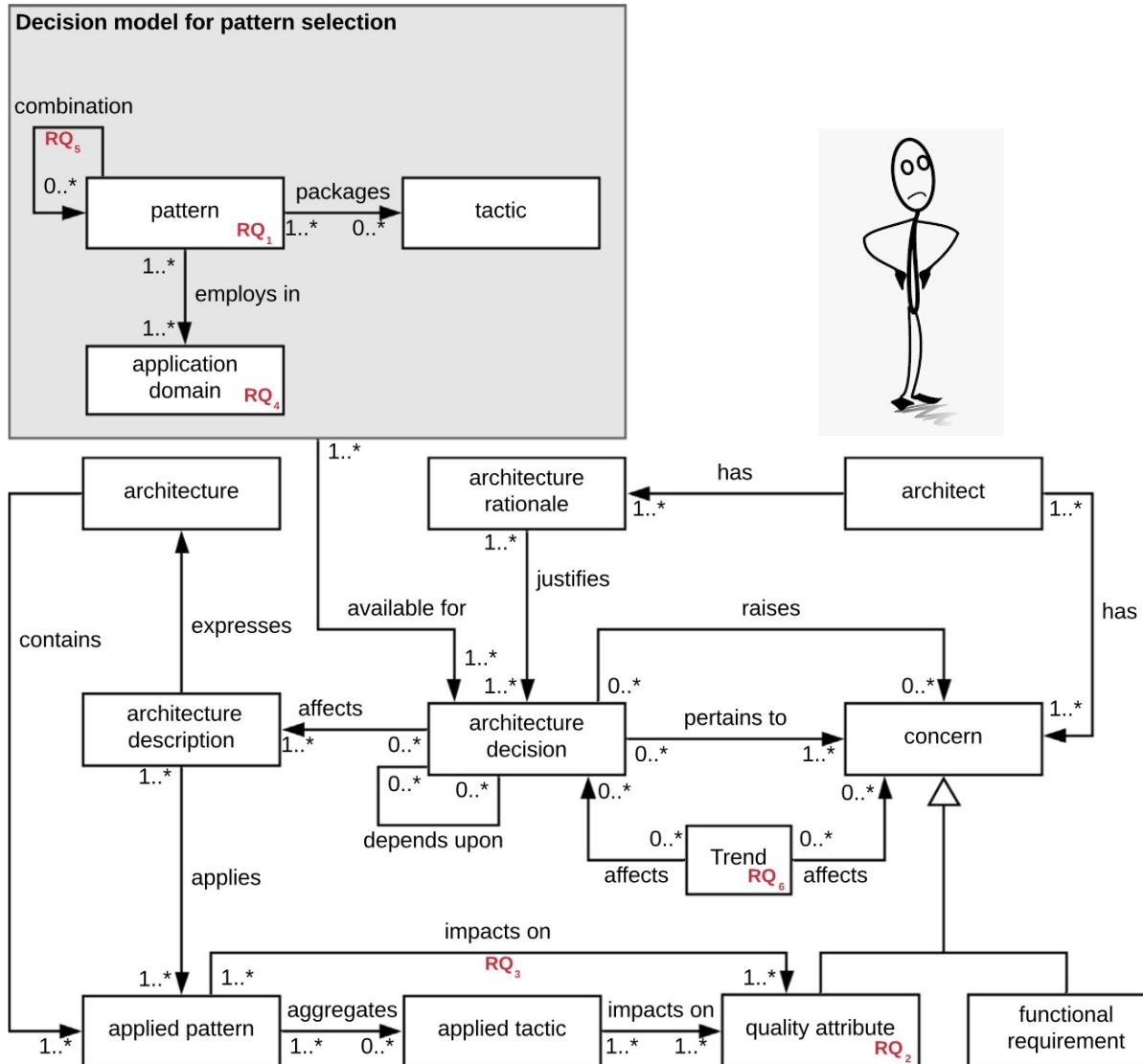
Most definitions of software architecture generally talk about

- ◆ 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.

Architecture as an Algorithm

<https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:42020:ed-1:v1:en>

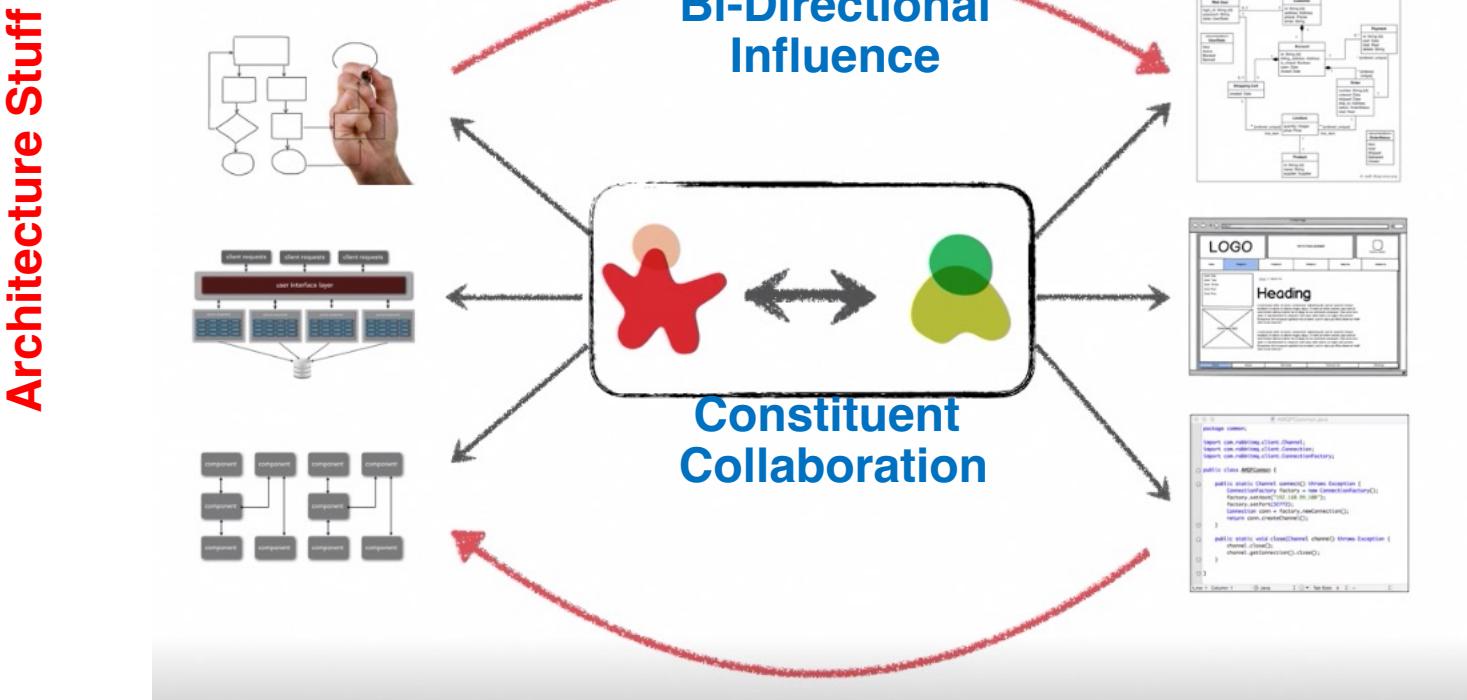


Term	Definition	Link
Software architecture	Software architecture is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.	1 2
Pattern	universal and reusable solutions to commonly occurring problems in software architecture.	1
Tactic	design decisions that improve individual quality attribute concerns	1
Quality	The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value.	1
Architect	person, team, or organization responsible for systems architecture	1
Rationale	captures the knowledge and reasoning that justify the resulting design, and its primary goal is to support designers by providing means to record and communicate the argumentation and reasoning behind the design process.	1
Decision	A decision is consisting of a restructuring effect on the components and connectors that make up the software architecture, design rules imposed on the architecture and resulting system as a consequence, design constraints imposed on the architecture, and a rationale explaining the reasoning behind the decision.	1
Functional requirement	condition or capability that must be met or possessed by a system, system component, product, or service to satisfy an agreement, standard, specification, or other formally imposed documents	1
Concern	is any interest in the system. The term is derived from the phrase “separation of concerns” as in Software Engineering. One or more stakeholders may hold a concern. Concerns involve system considerations such as performance, reliability, security, availability, and scalability.	1

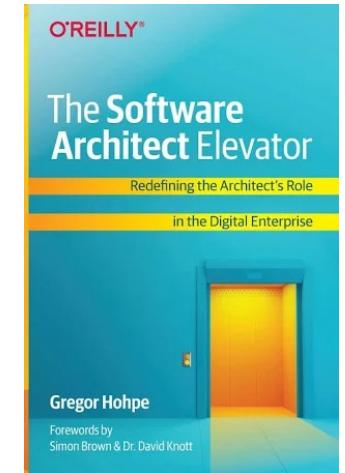
We can also look at this non- hierarchical... The relation between the architecture and design spaces

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals

where do you draw the line between
architecture and design/development?



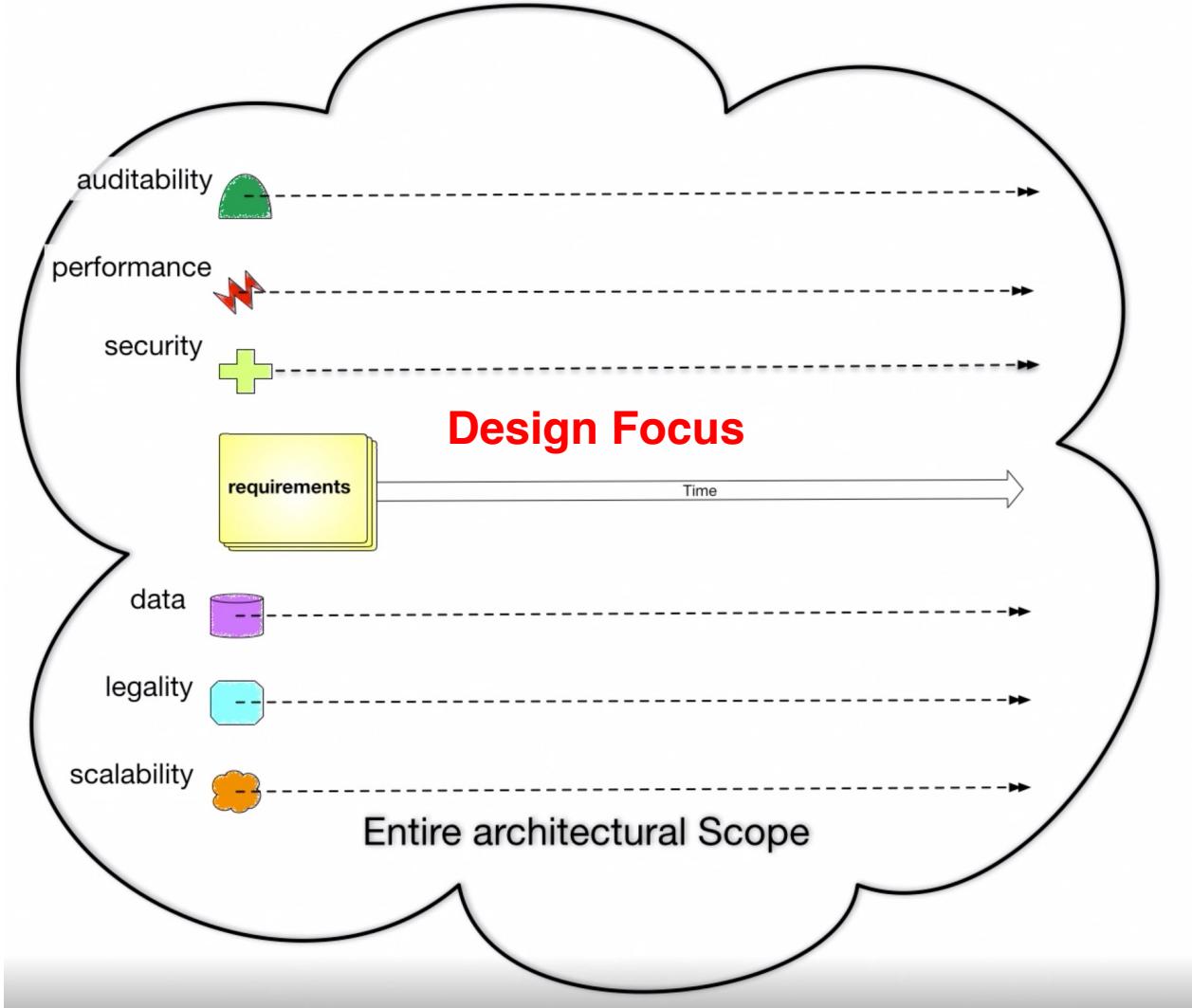
Design Stuff



Architecture has a broader scope than design

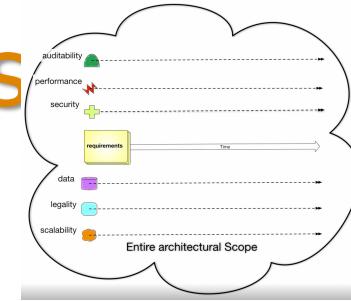
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals

Broader Architecture Focus



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](#)

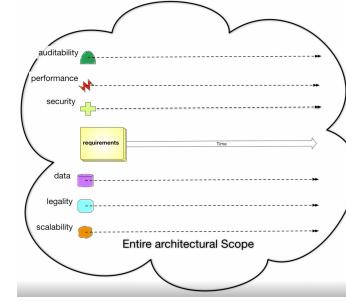
- [mobility](#)
- [modifiability](#)
- [modularity](#)
- [operability](#)
- [orthogonality](#)
- [portability](#)
- [precision](#)
- [predictability](#)
- [process capabilities](#)
- [producibility](#)
- [provability](#)
- [recoverability](#)
- [relevance](#)
- [reliability](#)
- [repeatability](#)
- [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

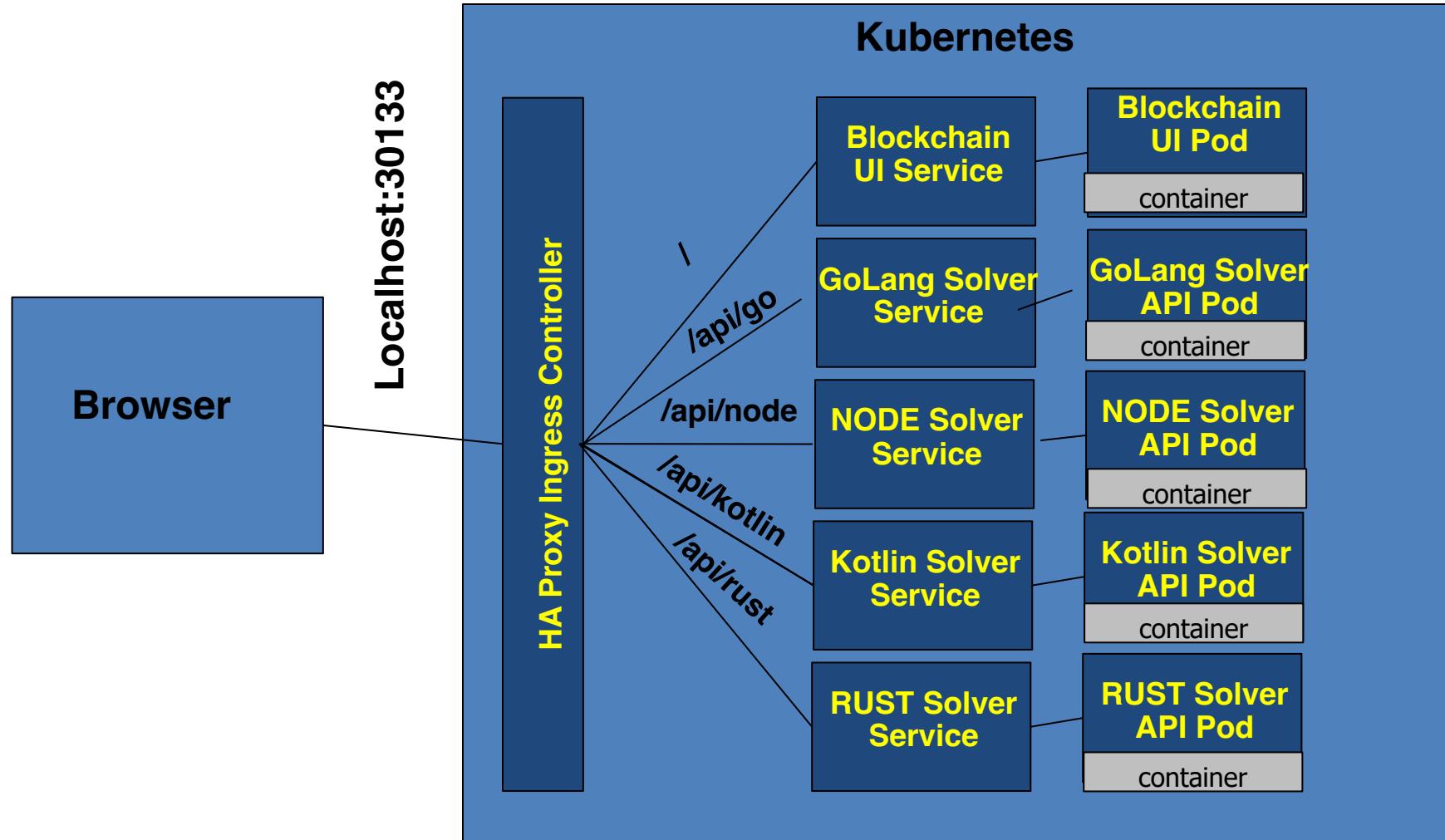


Although there are many quality attributes, certain ones tend to appear again and again that impact the design and architecture of systems

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

++ FEASIBILITY ++

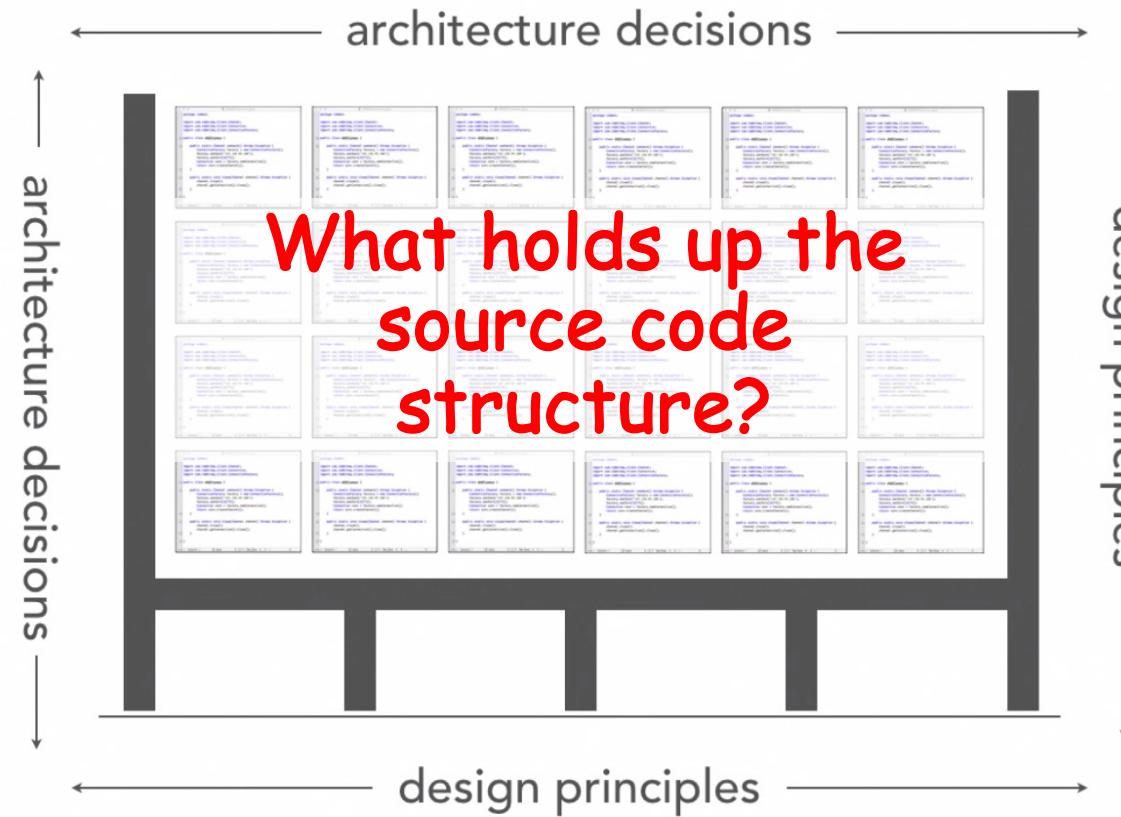
Blockchain Demo – Quality Attributes and Architecture & Design



The structure of the source code should not be determined arbitrarily

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals

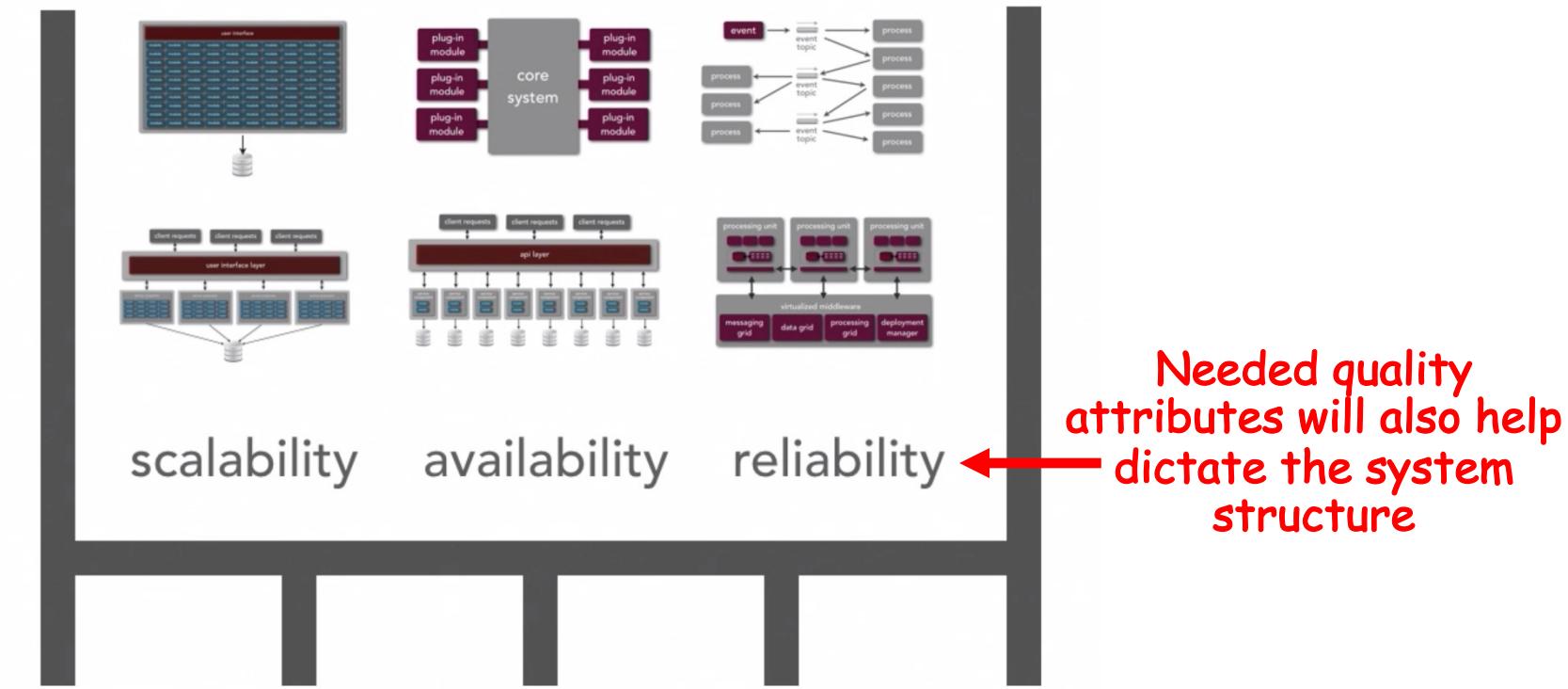
what is architecture?



Starting with understanding the structure of the system

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals

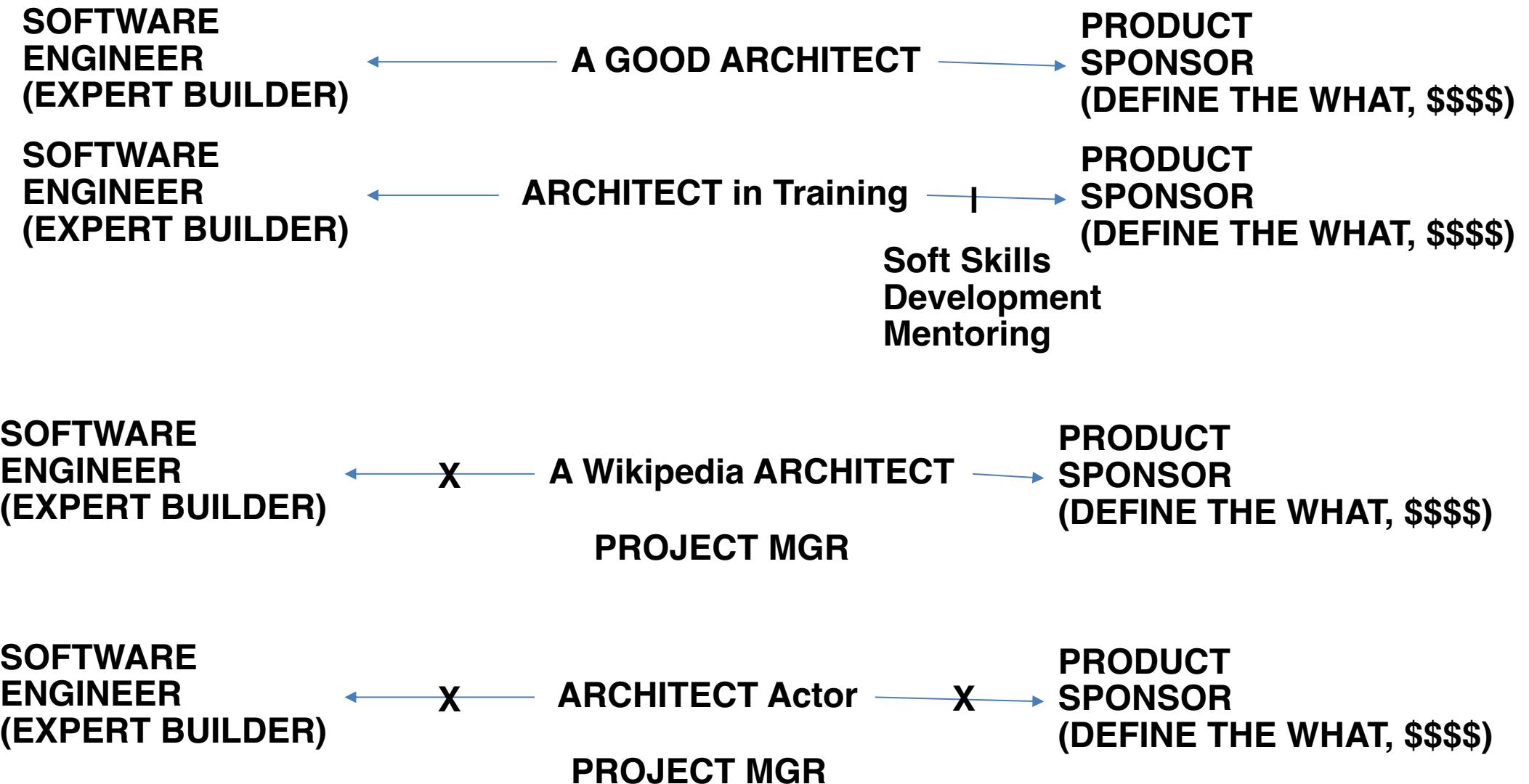
We can use architecture styles and patterns to structure the system



Back to Architecture Basics

- When I introduced 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?

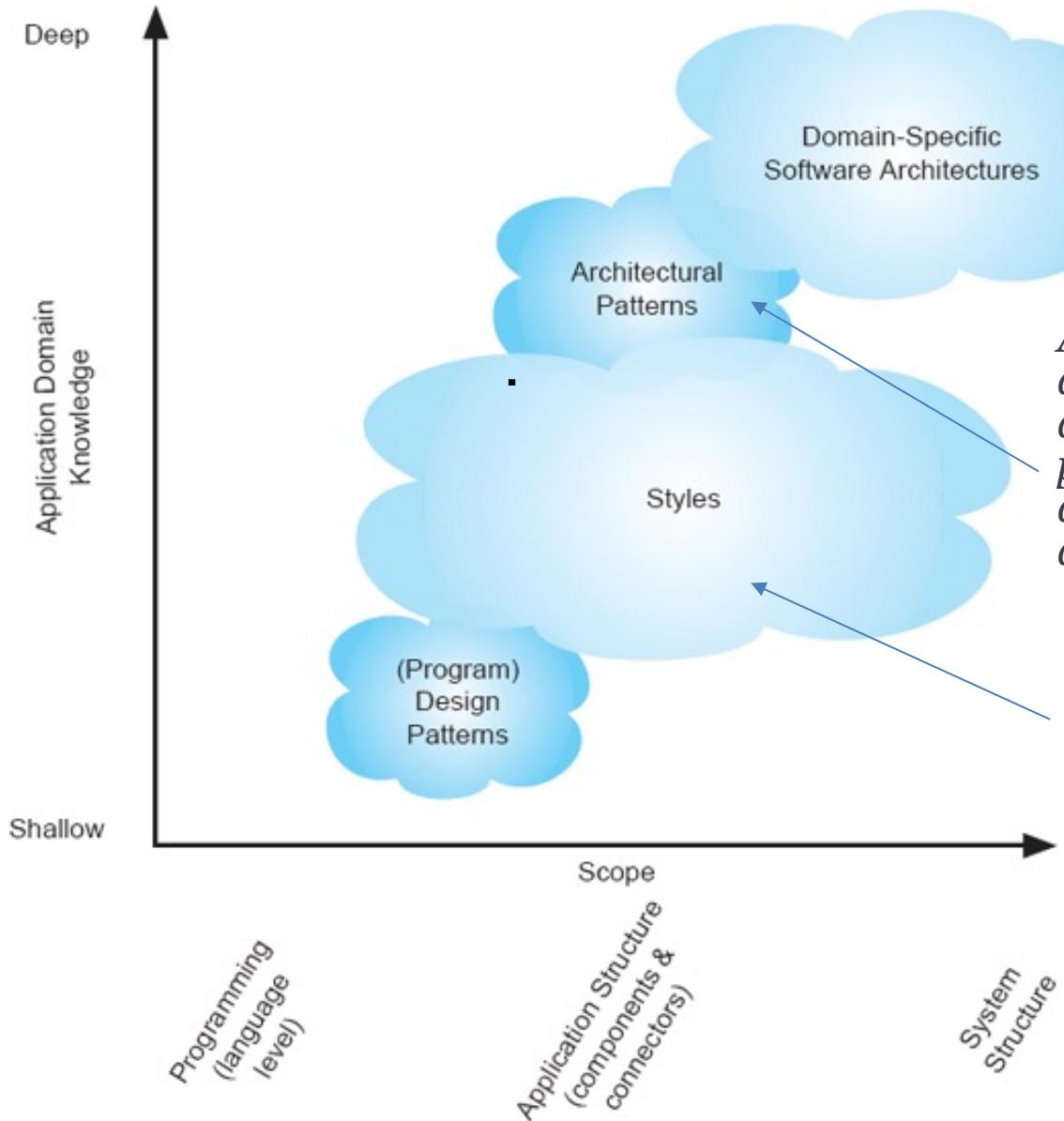
DRAFT – Types of Architects



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
 - Example: Model-View-Controller (MVC)
- Design Patterns
 - Focus is on the organization of a block of code

Software Architecture Landscape



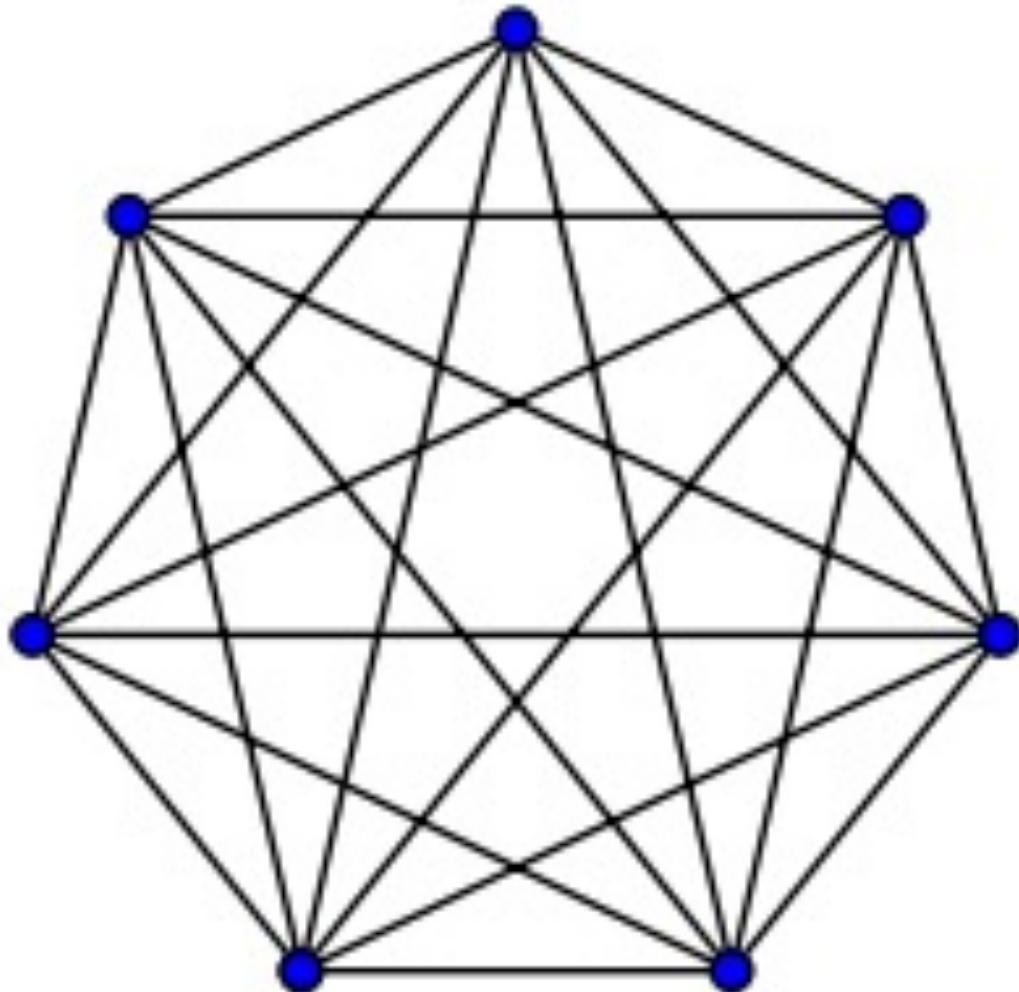
Also known as a *reference architecture*, is the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation

An architectural pattern is a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears.

An architectural style is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system.

R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory and Practice*, Wiley.

Remember every system has an architecture – Our first architecture pattern

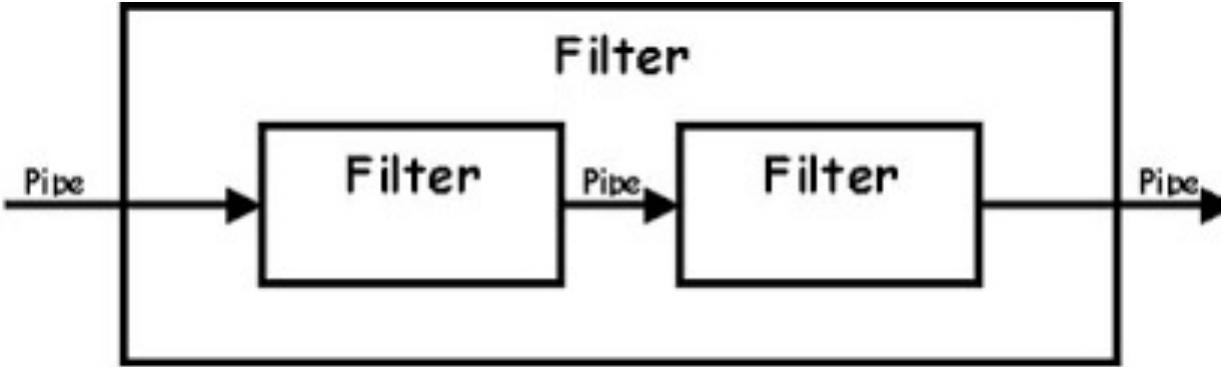


Lets call this the “ball of mud” architecture pattern – it meets the definition

Constraints of this pattern – every component is allowed to directly talk to every other component – in other words – there are no constraints

Thus, introducing a new component, allows for an additional $n-1$ connections where n is the total number of components

Another Architecture Pattern – this one a little more realistic



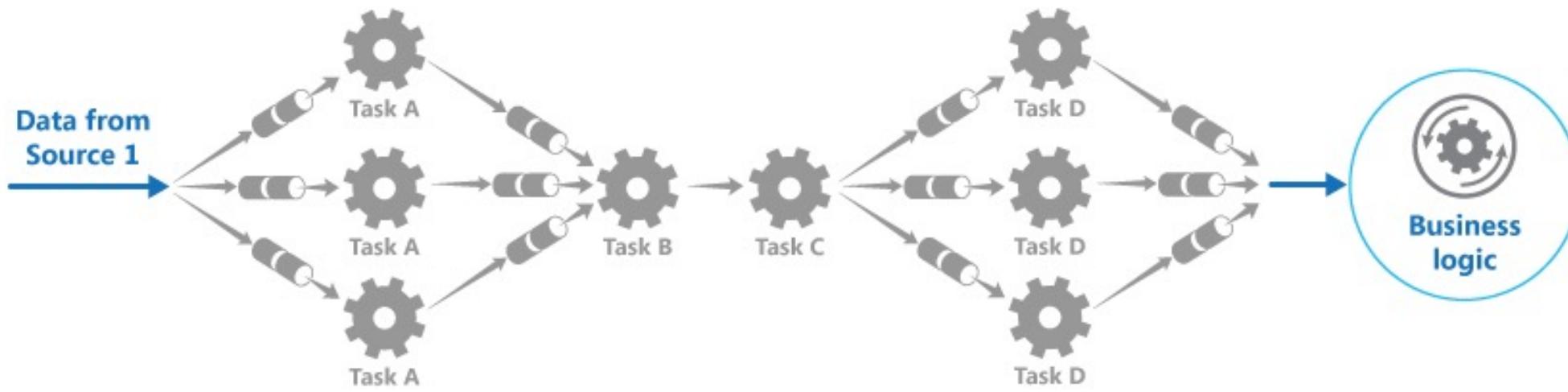
The Pipe-and-Filter is a classic architecture pattern – it's actually the basis for some of the goodness in functional programming and other design patterns such as the builder pattern

Components are the filters – all inputs are provided by pipes, all transformations are done by filters.

Filters are stateless, all transformations are based on pipe inputs

The filters can ingest inputs and produce outputs atomically or via streaming

Another Architecture Pattern – this one a little more realistic



The Pipe-and-Filter is also powerful because there can be special filters that split data and other filters that coordinate merging of data

This enables large scale and concurrency

Pipe and Filter is very useful in the real world

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

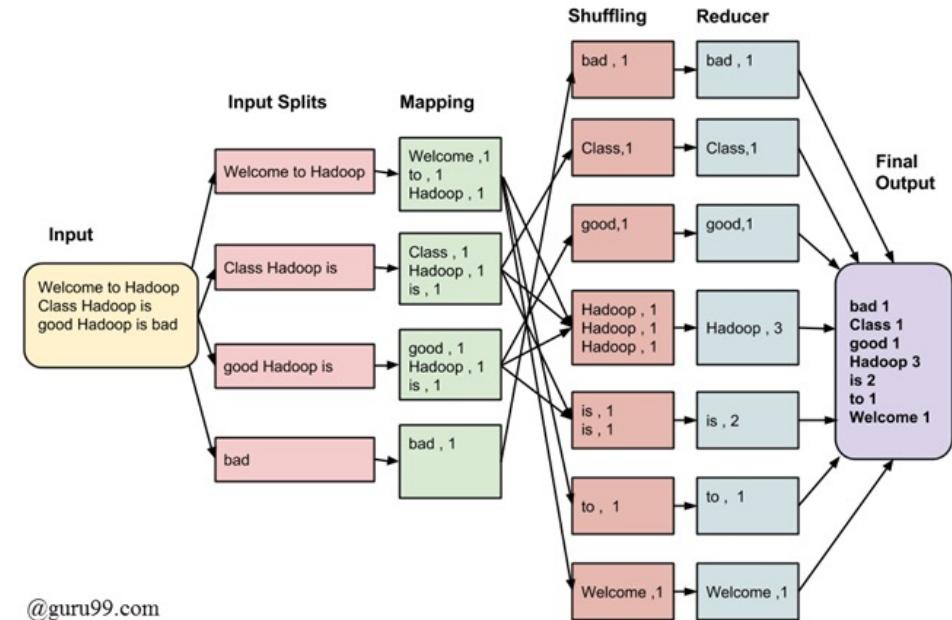
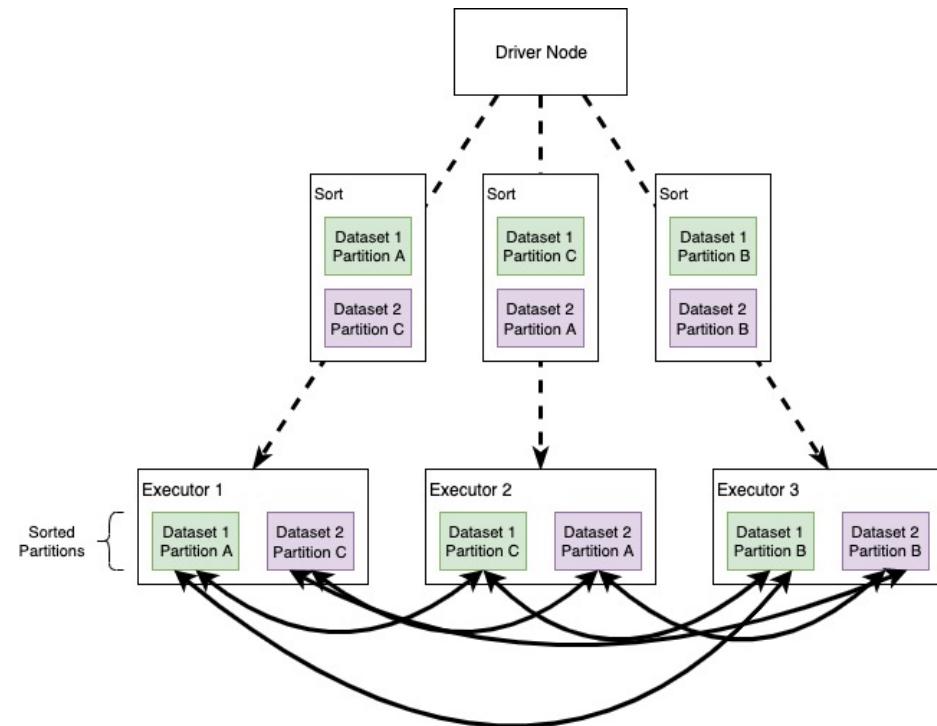
given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

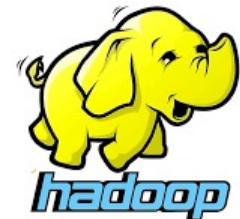
The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined

This is the seminal paper by google on MapReduce – basically an at scale implementation of pipe-and-filter

Pipe-and-Filter used as at scale data pipelines



@guru99.com



Pipe-and-Filter – Functional Programming

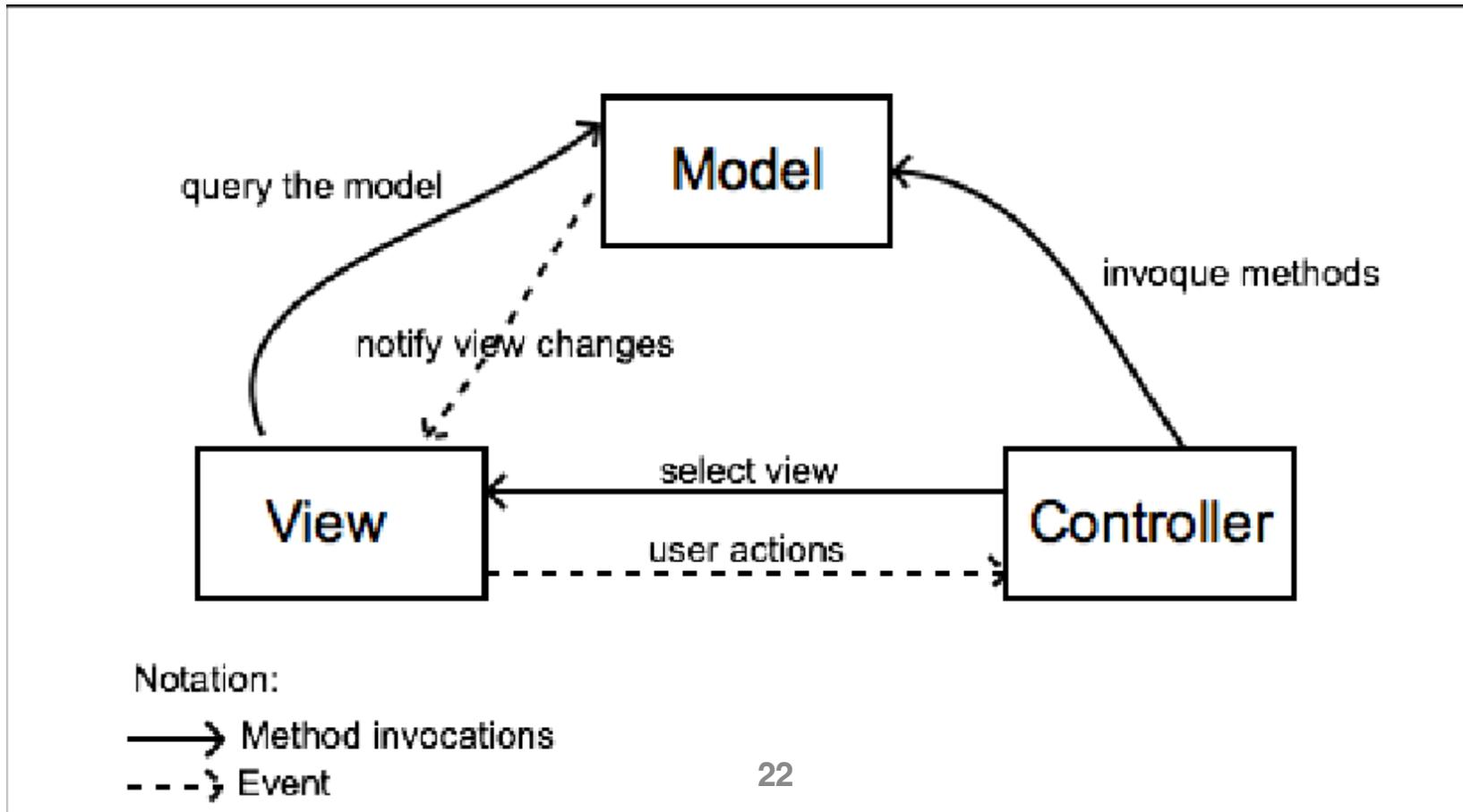
Demo at: <https://www.learnrxjs.io/learn-rxjs/recipes/type-ahead>

```
11 const getContinents = keys =>
12   [
13     'africa',
14     'antarctica',
15     'asia',
16     'australia',
17     'europe',
18     'north america',
19     'south america'
20   ].filter(e => e.toLowerCase().indexOf(keys.toLowerCase()) > -1);
21
22 const fakeContinentsRequest = keys =>
23   of(getContinents(keys)).pipe(
24     tap(_ => console.log(`API CALL at ${new Date()}`))
25   );
26
27 fromEvent(document.getElementById('type-ahead'), 'keyup')
28   .pipe(
29     debounceTime(200),
30     map((e: any) => e.target.value),
31     distinctUntilChanged(),
32     switchMap(fakeContinentsRequest),
33     tap(c => (document.getElementById('output').innerText = c.join('\n')))
34   )
35   .subscribe();
```

An example of an architecture “style” – MVC (Model/View/Controller)

MVC is a more realistic example of an architecture style

- View is passive, reacts to changes – updates data based on model events, updates presentation based on controller reacting to user behavior
- Application state fully managed by model



Sidebar – Architecture Style versus Architecture Pattern?

If you study classic references for software architecture you will see that architecture styles and patterns are considered different things...

While we can form a general understanding of their differences based on abstraction (e.g., pipe/filter vs MVC)...

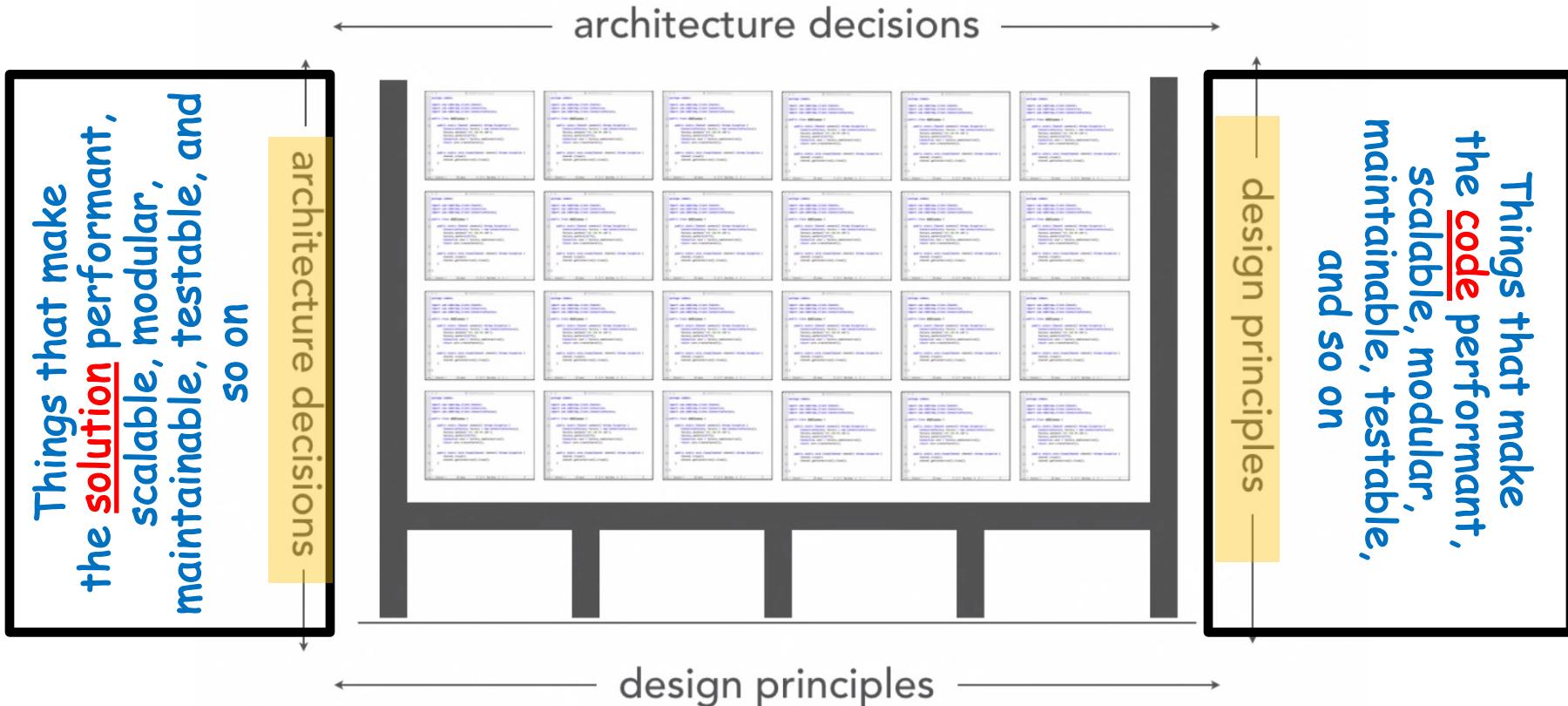
In practice, they can generally be used interchangeably – I'll generally just call them architecture styles

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, ...

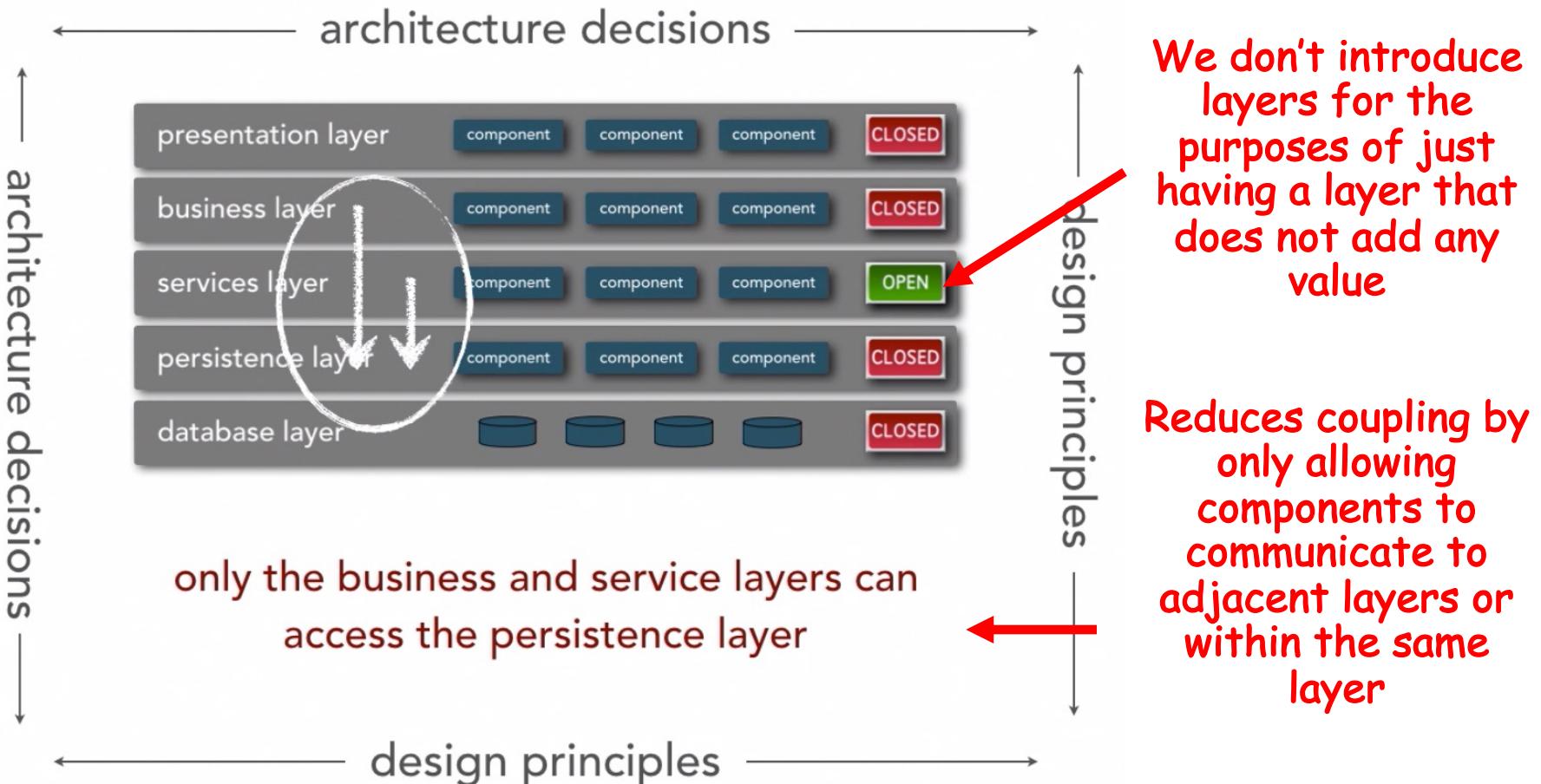
Architecture decisions and design principles also influence the system structure

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



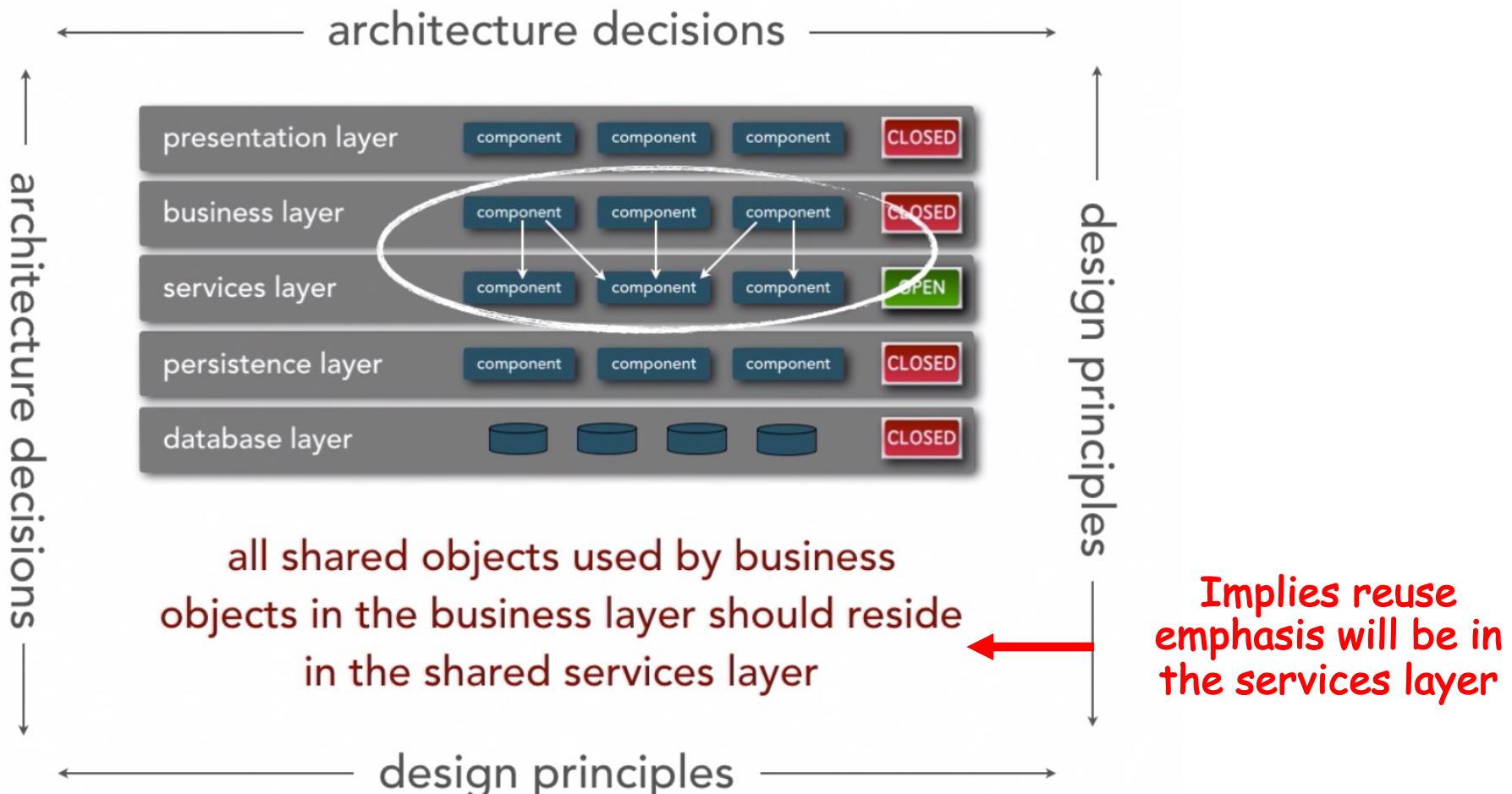
Architecture principal example

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



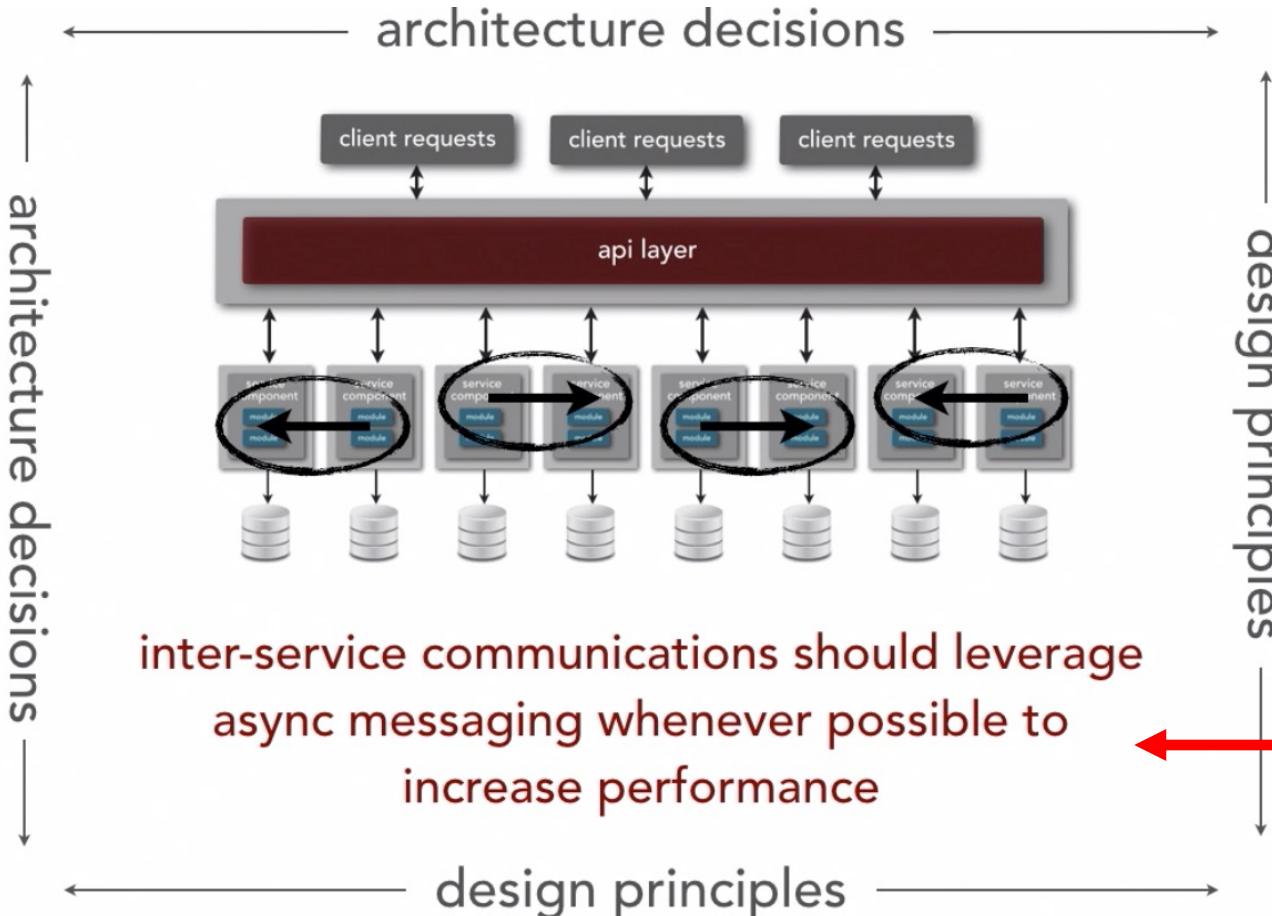
Another architecture principal example...

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Design principles can impact many platform and technology aspects

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals

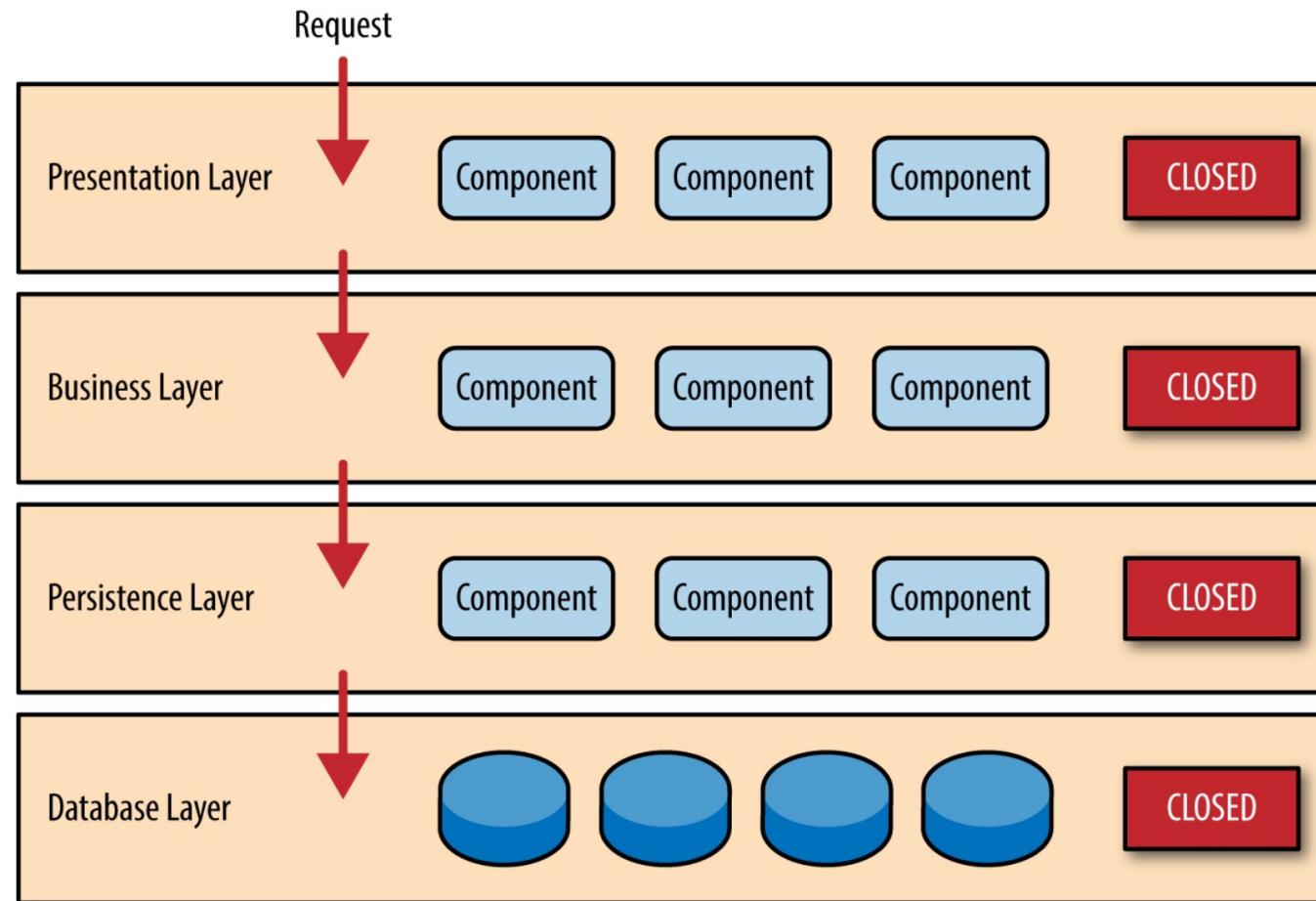


A design decision
that has been
shown to have good
performance
characteristics...
Tradeoff is
introduction of
complexity

Back to Architecture

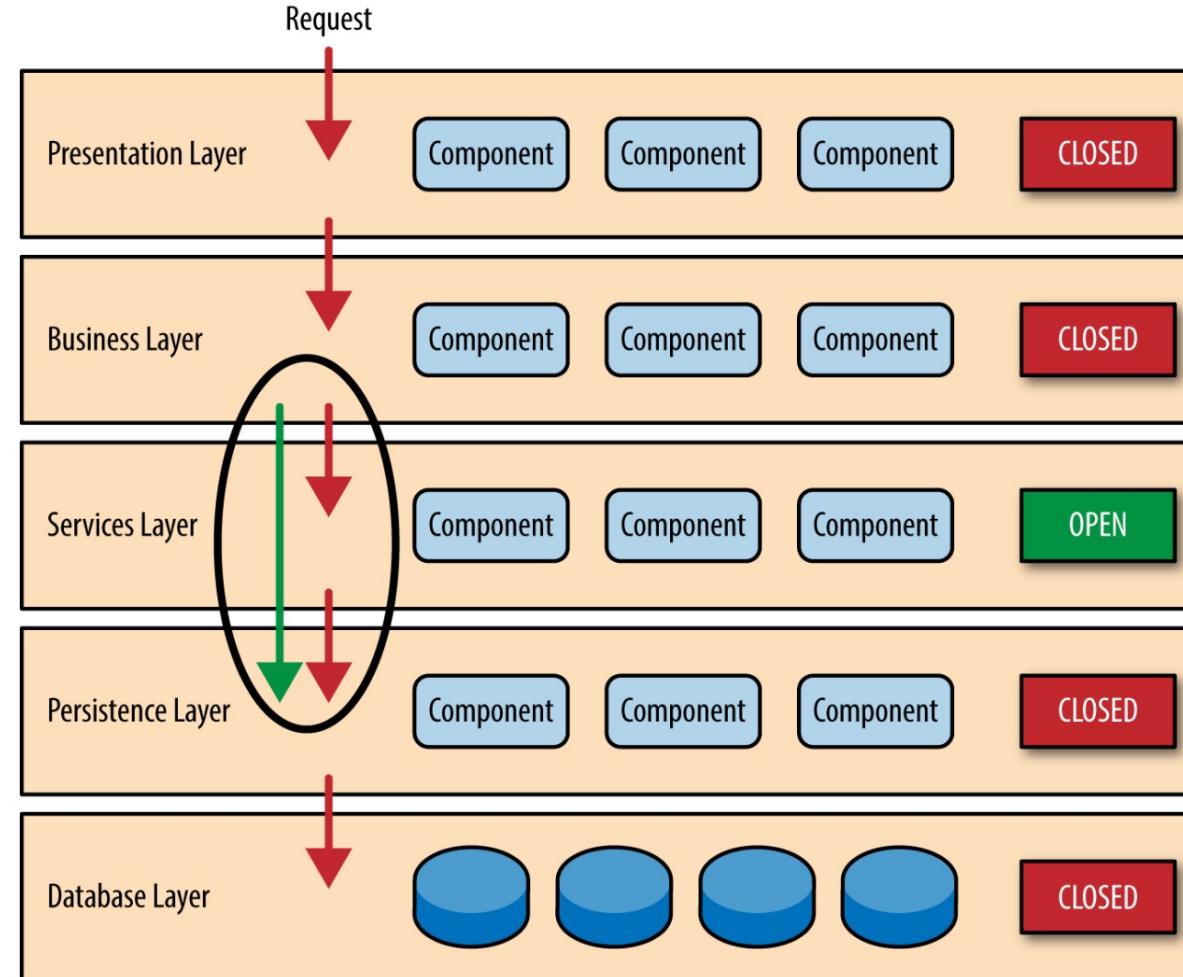
- The previous definitions presented focus on the vocabulary of architecture components
 - Terms used: Components, Connectors, Structures, and so on.
- I like to think about architecture in terms of laying out the foundation for design
 - What **design decisions** do we need to make, and/or what **structures 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?

Layered Style



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

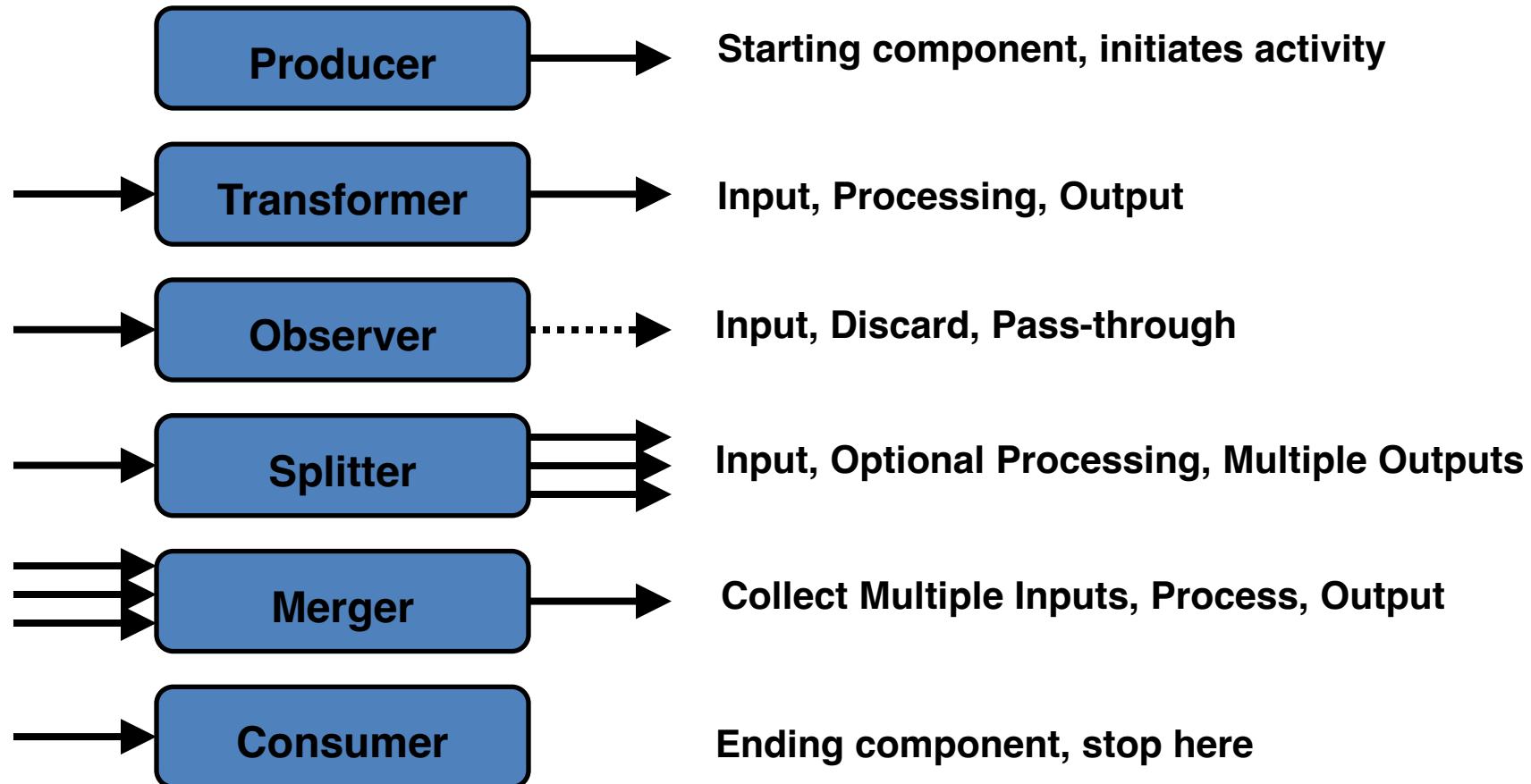
Layered Style



Sometimes Layers are Open To Bypassing

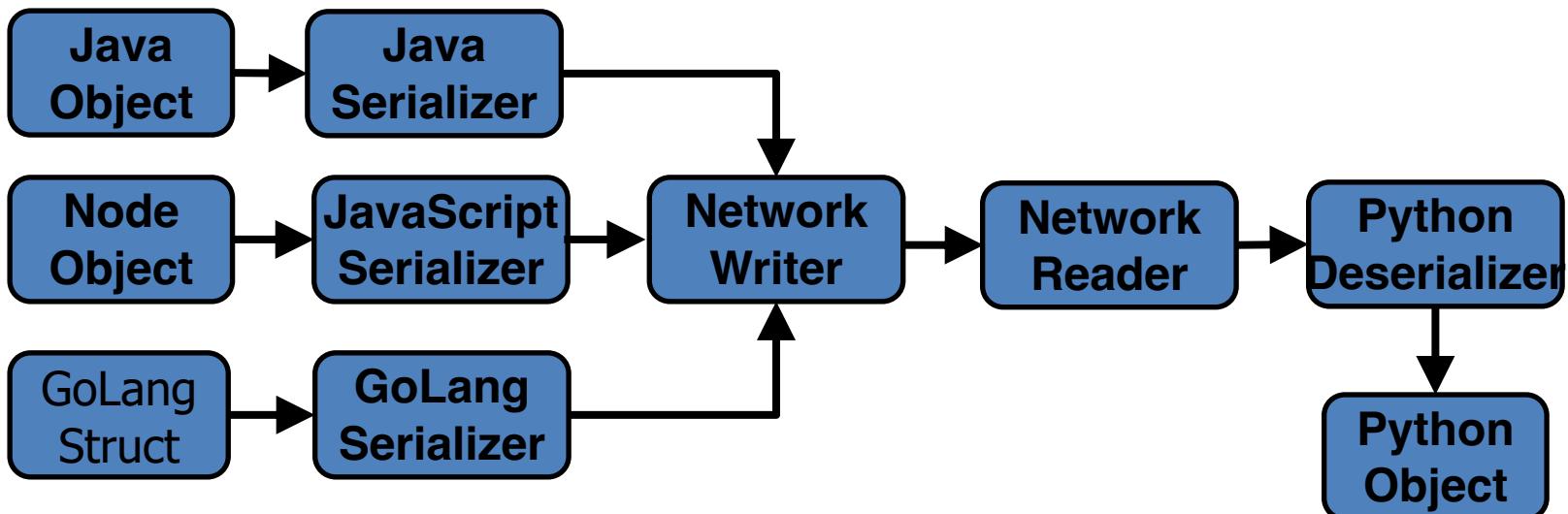
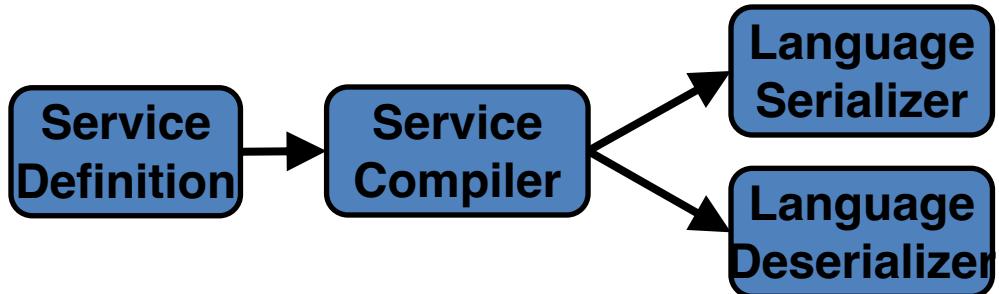
Pipe and Filter

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

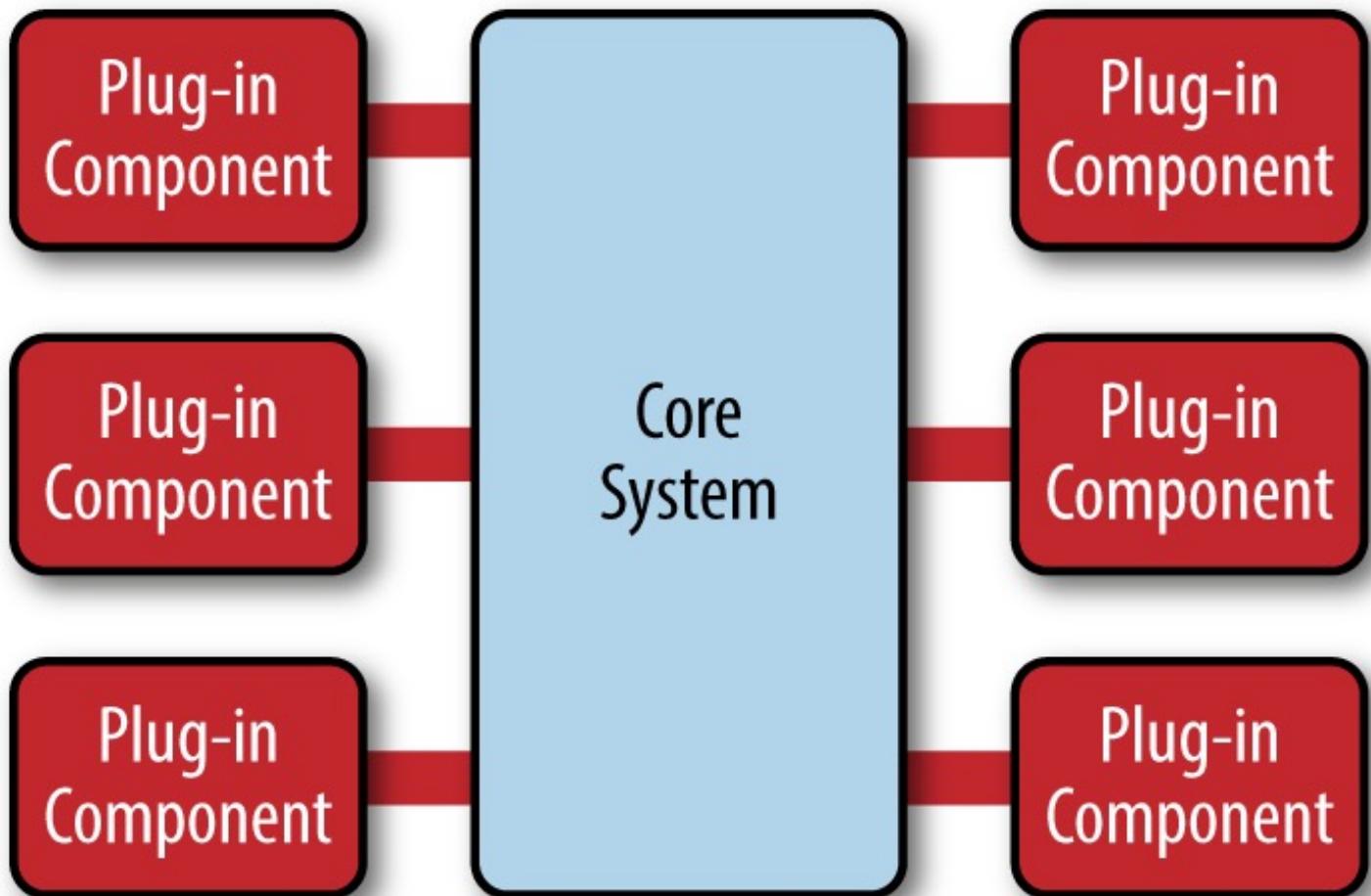


Pipe and Filter

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



Microkernel Architecture Style



A common architecture style for Software as a Service solutions

The “Core System” provides the value added functionality out of the box

The “Plug In Component” can be other purchased software specific to an industry vertical (e.g., healthcare, telecom, etc), or custom code taylored to a specific need.

Often-times the core system will prove an API or SDK to enable the plug-in ecosystem

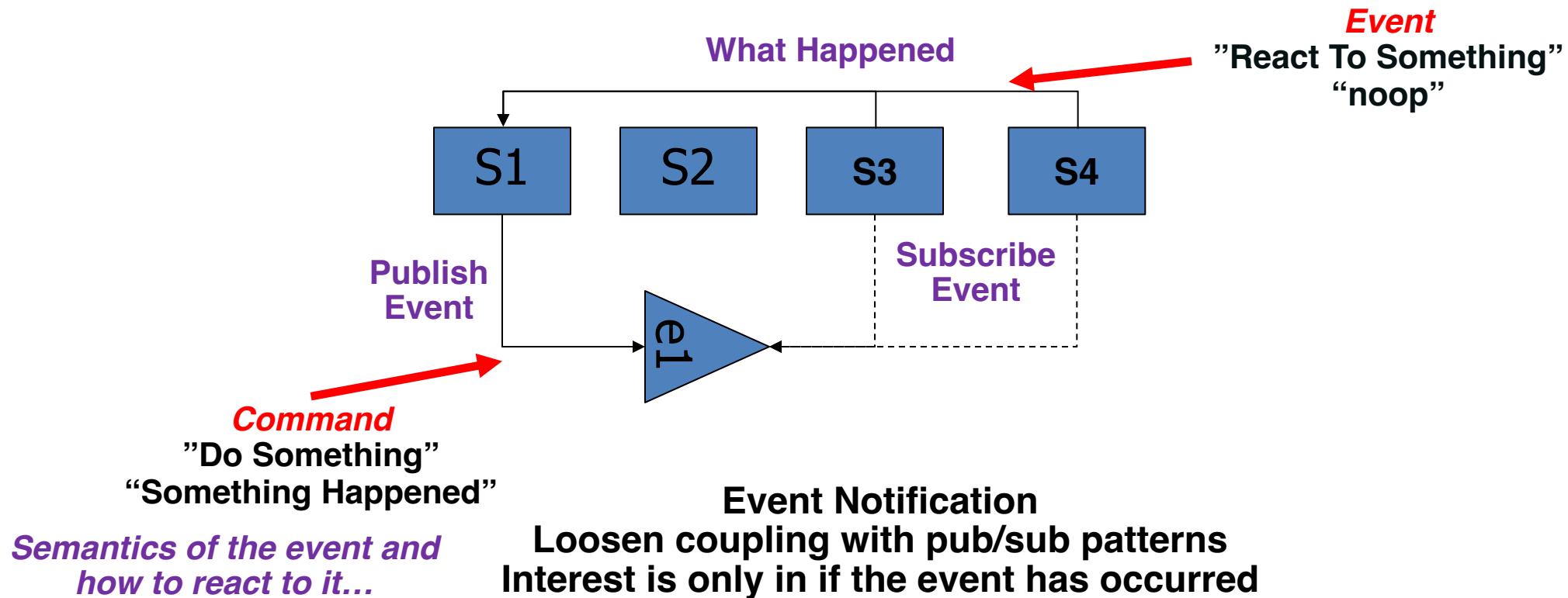
Microkernel Architecture Style - Examples



Visual Studio Code

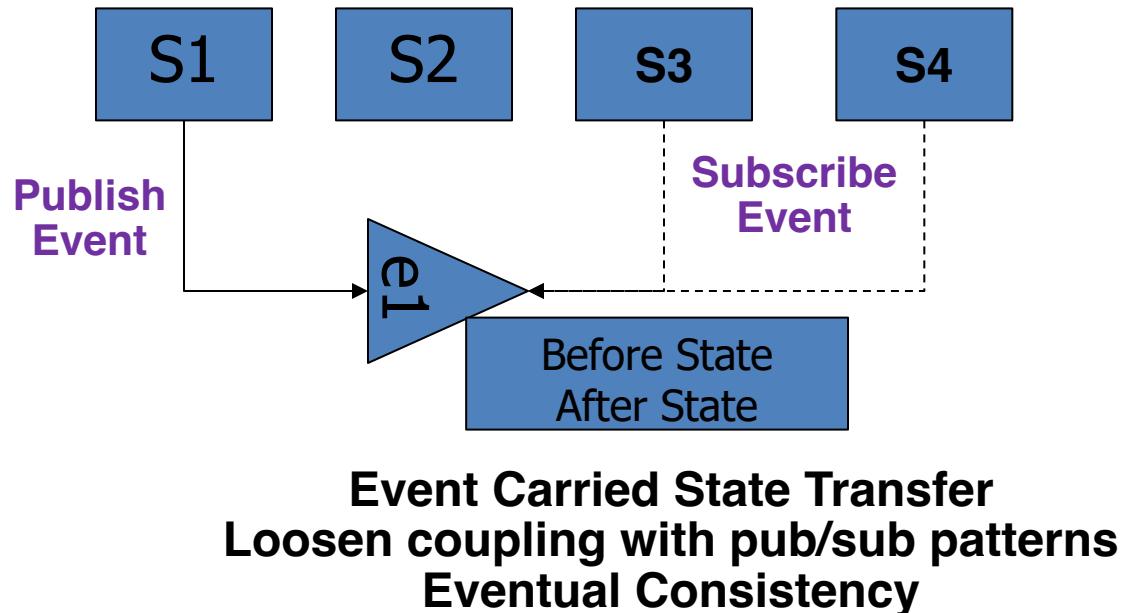


Event Architectures – Event Notification



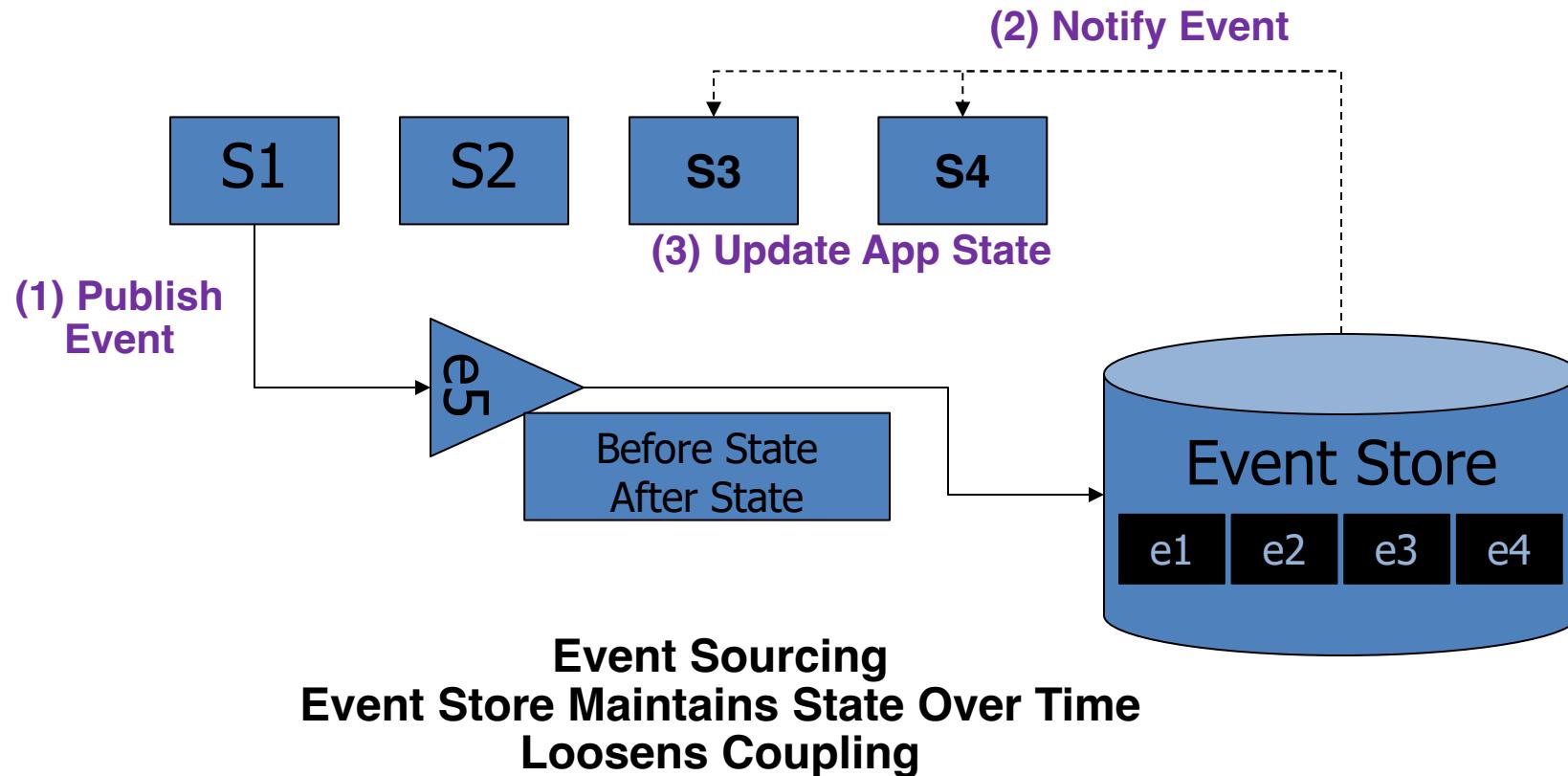
Source: The Many Meanings of Event-Driven Architecture – Martin Fowler

Event and Streaming Architectures



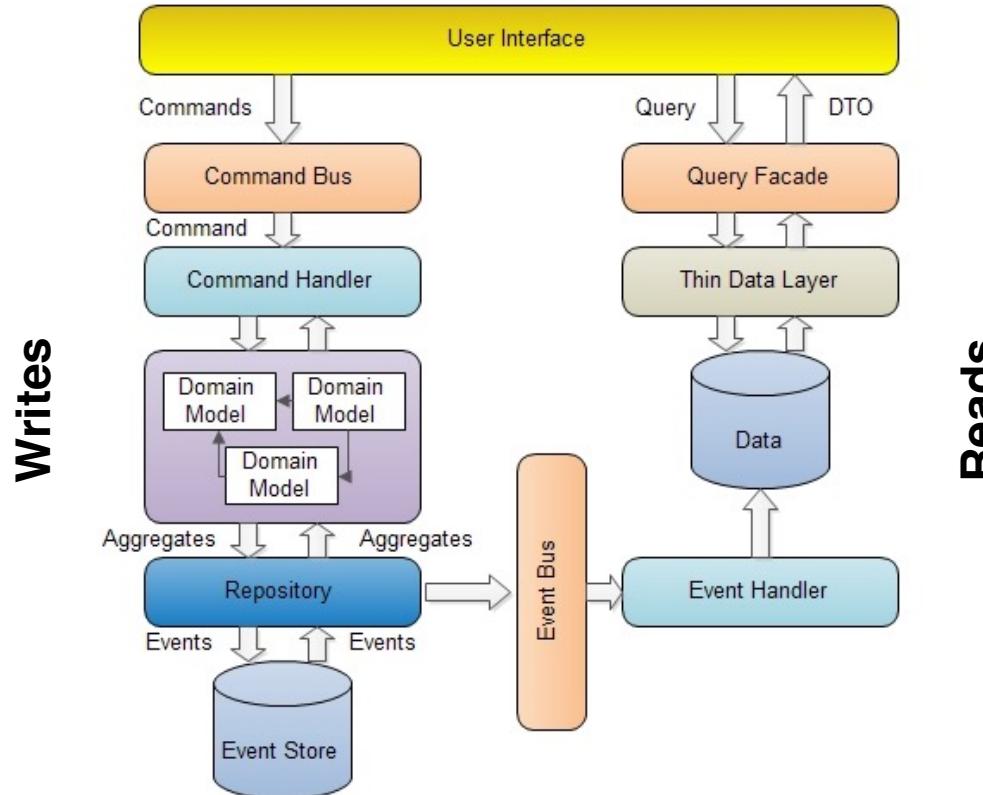
Source: The Many Meanings of Event-Driven Architecture – Martin Fowler

Event – Event Carried State Transfer



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

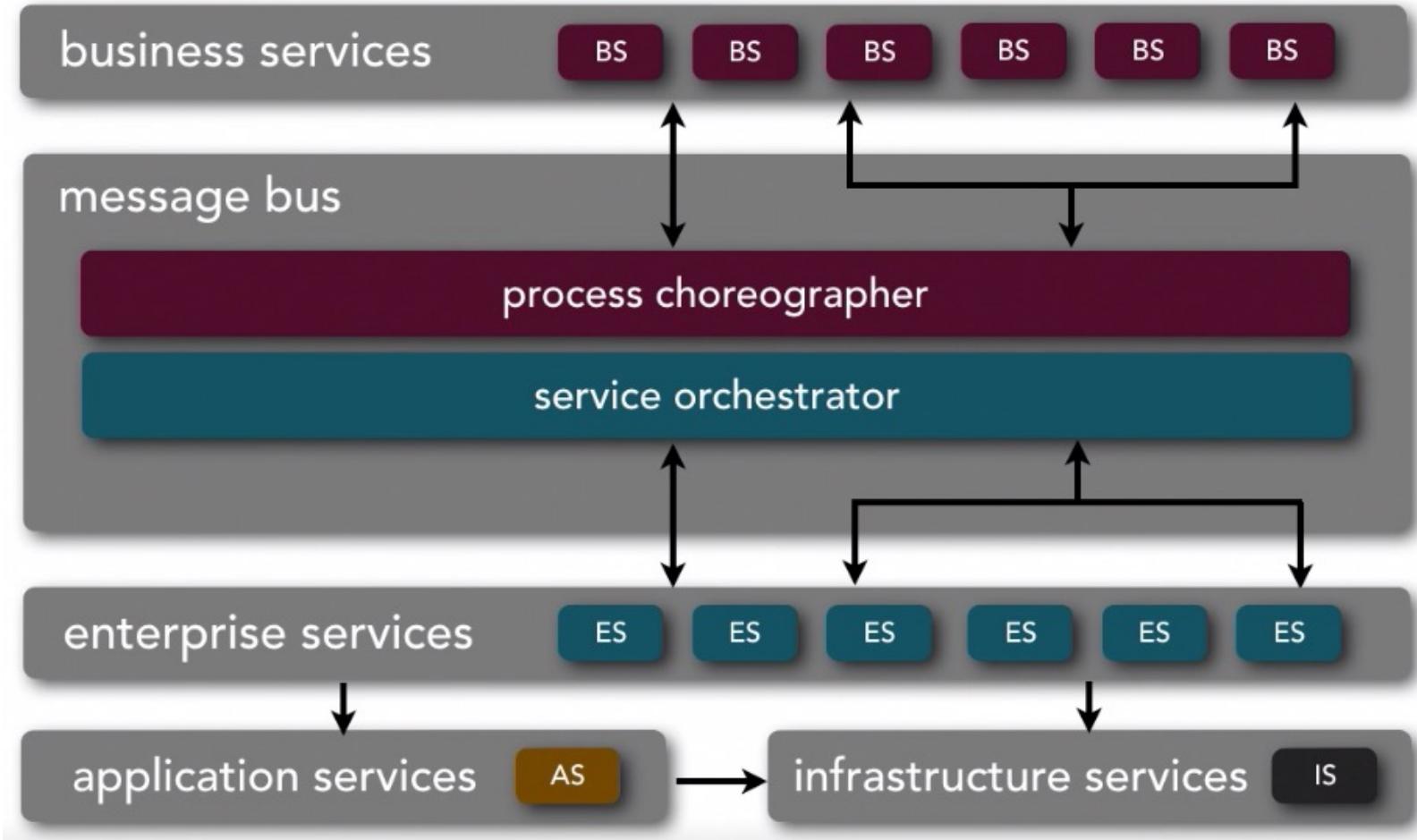
Event – Command Query Responsibility Separation



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

Service Oriented Architecture Style

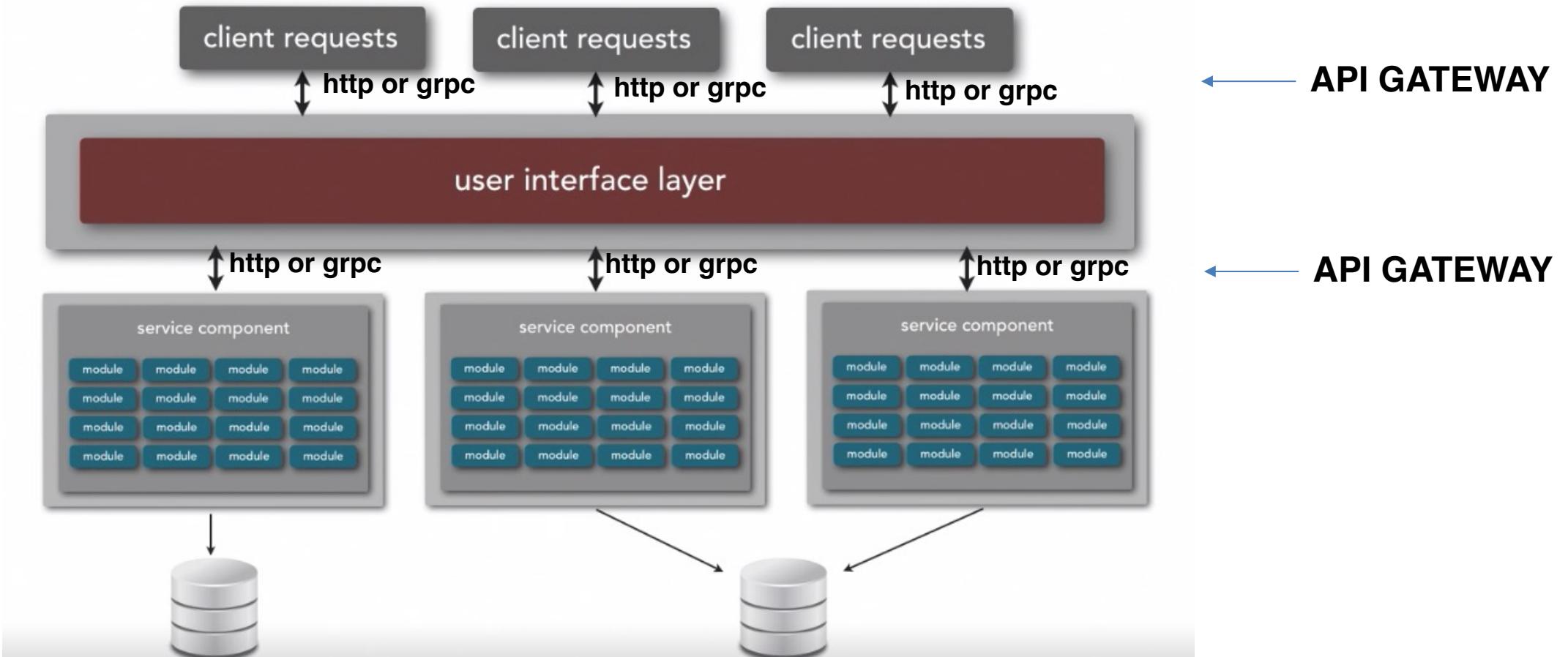
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Thoughts? Good Parts? Bad Parts?

Service Based Style

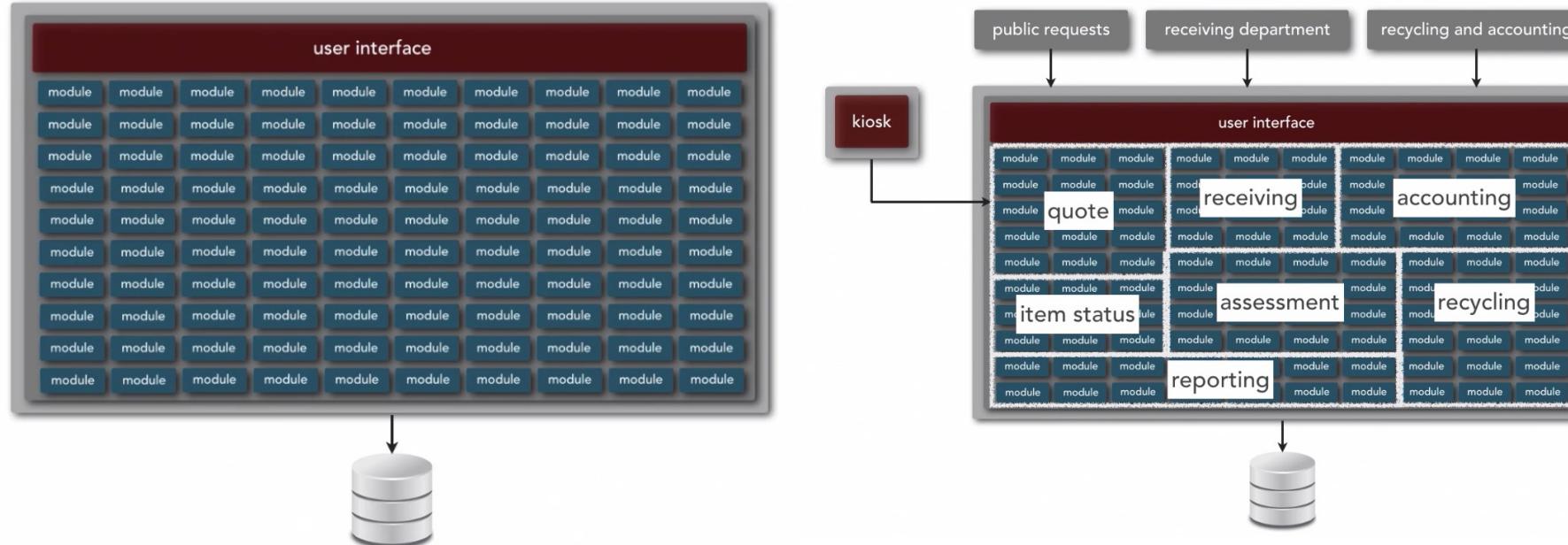
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Works Well To Partition Functionality

Service Based Style - Example

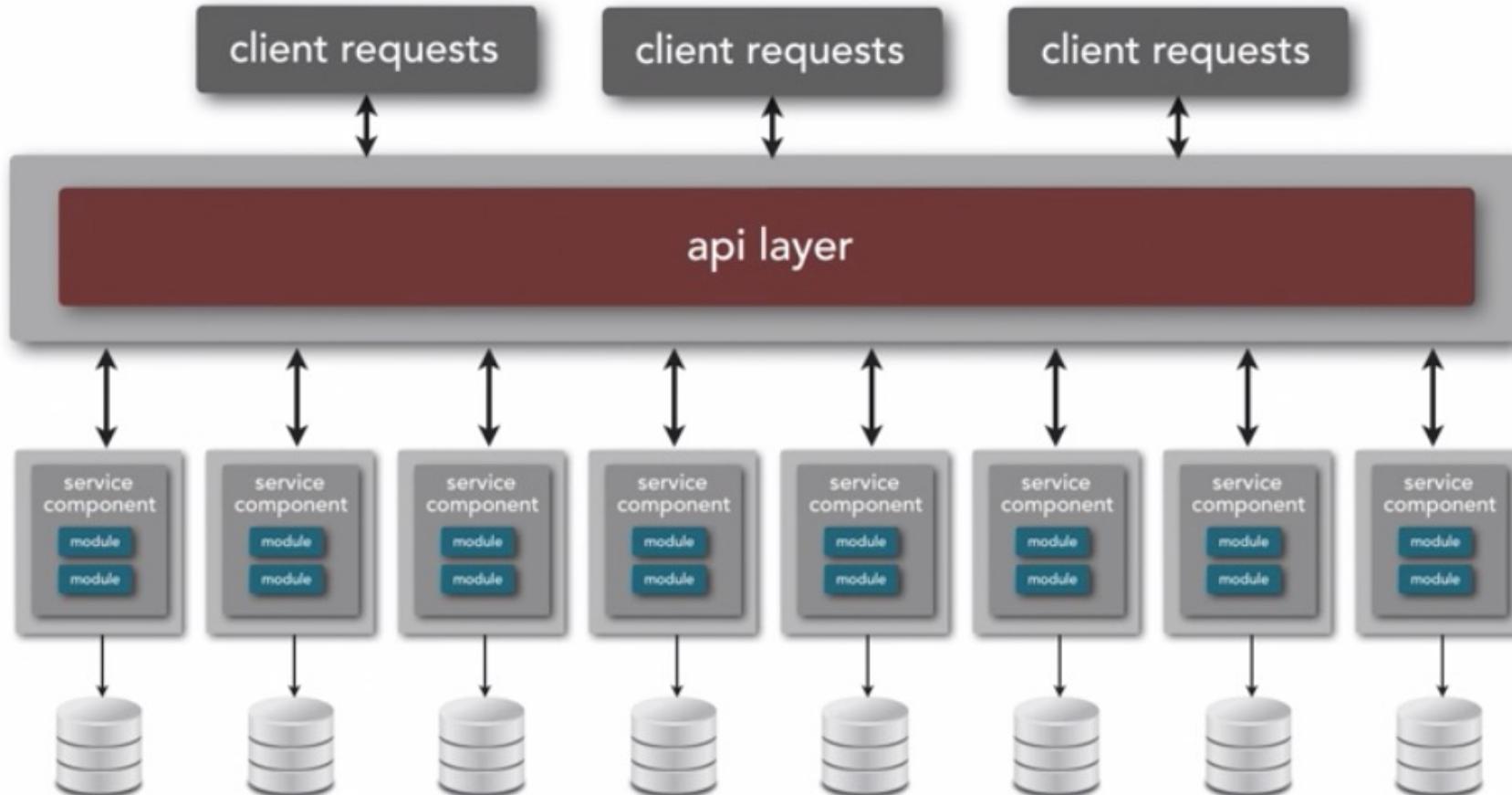
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Generally Good To Decompose a Monolithic Application

Microservice Architecture

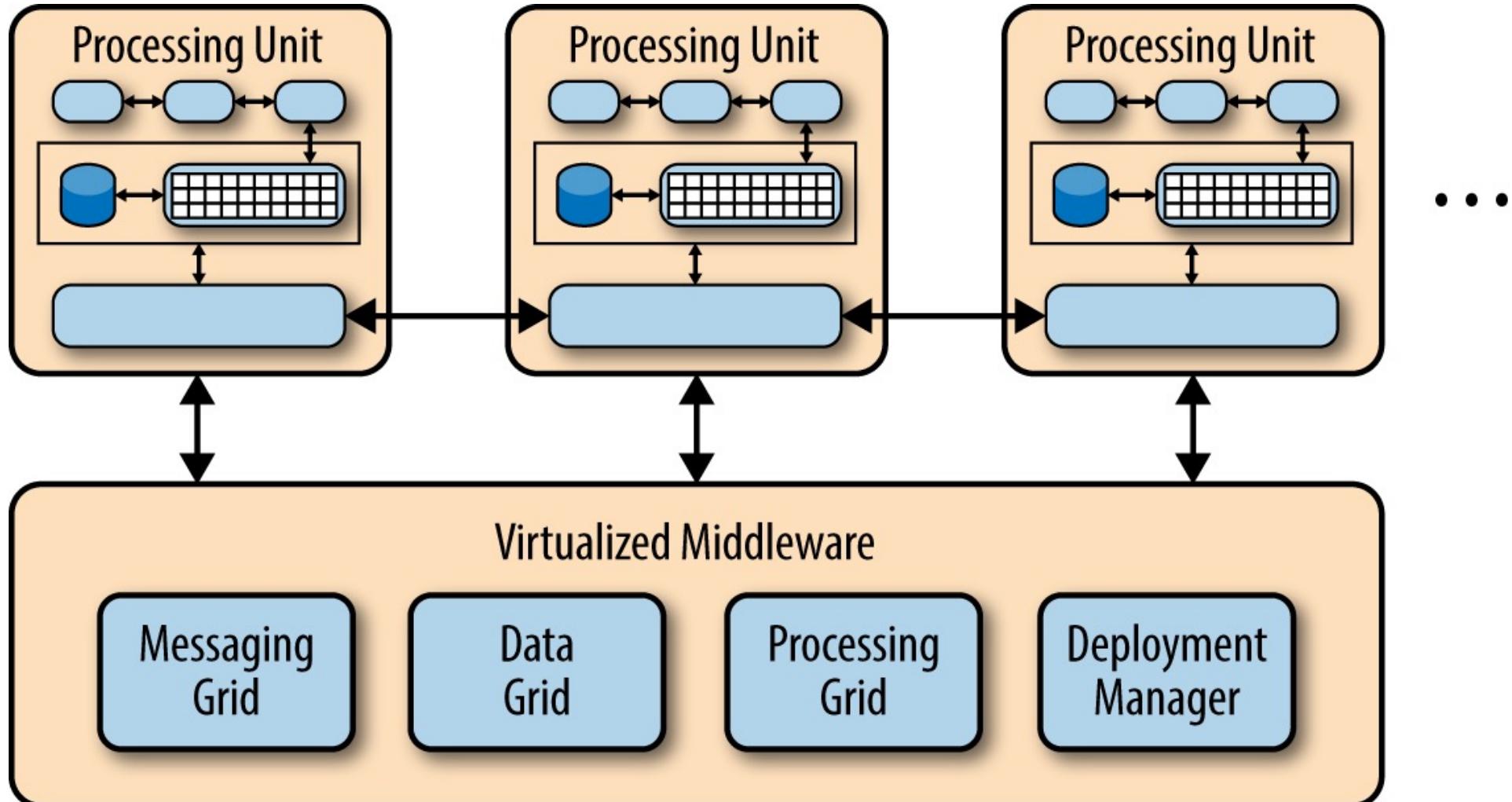
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Basic Idea – SHARE NOTHING
43

Space-Based Architecture (aka Cloud)

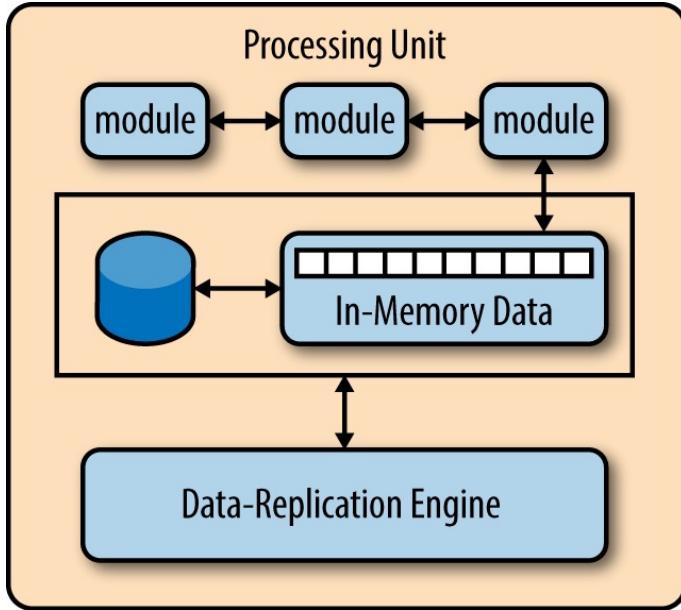
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



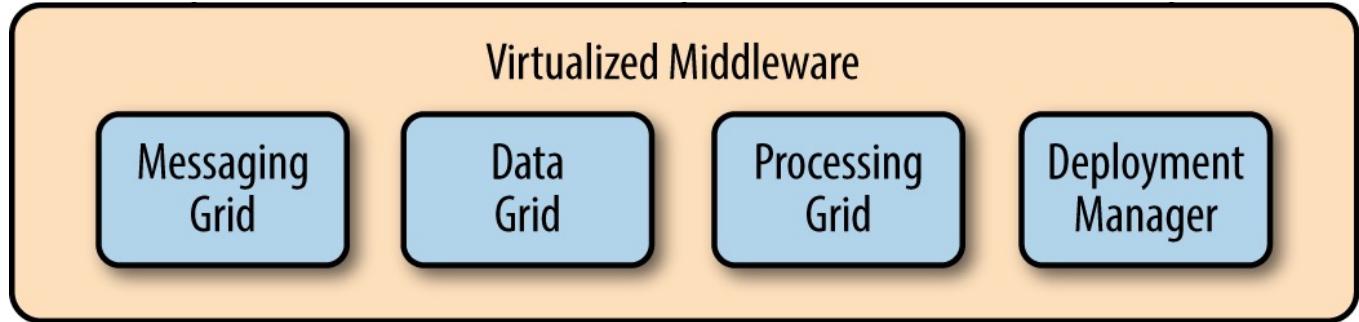
Basic Idea – Minimize Factors that Limit Scaling

Space-Based Architecture- Key Components

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



The processing unit provides a virtualized environment that enables modules to have everything that they need appear to be local

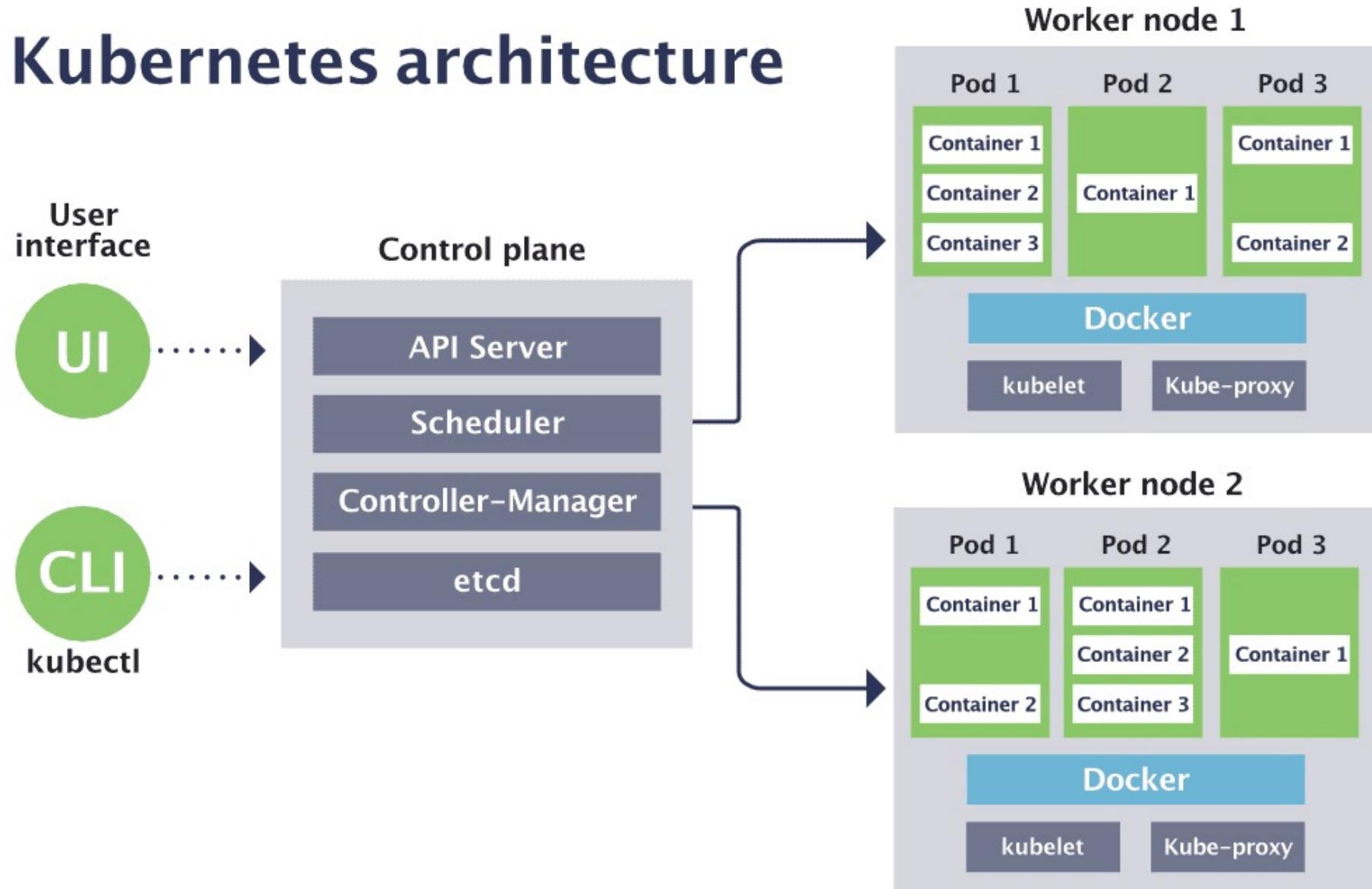


The virtual middleware connects to all processing units

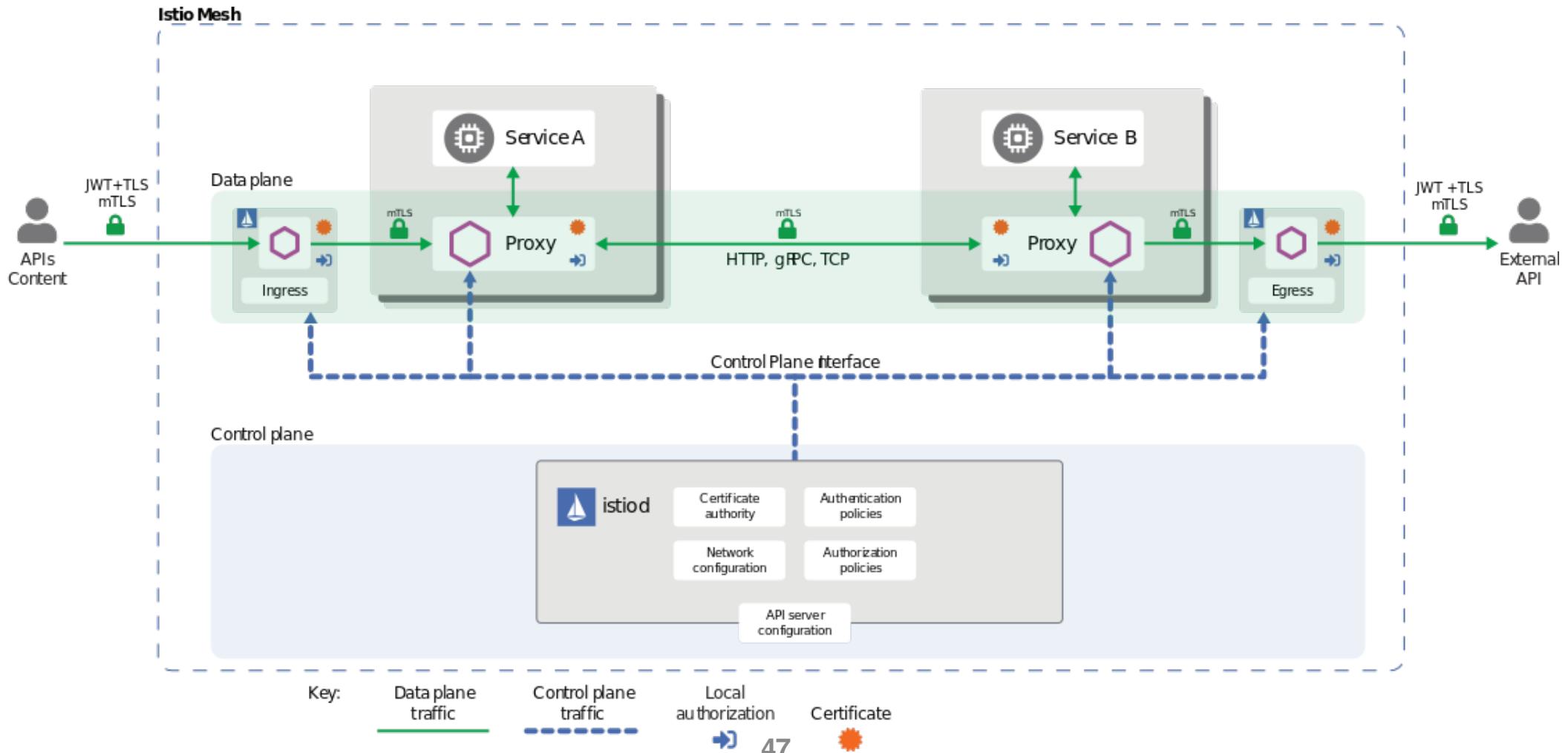
- Messaging Grid: Behaves like a smart load balancer, intelligently routing messages to processing units
- Data Grid: Works with the data replication components in each processing unit to make sure the right data is replicated to the right Processing Unit and kept coherent
- Processing Grid: Mediates and orchestrates distributed work that is executed on multiple processing units
- Deployment Manager: Ensures the right code is deployed to the right processing unit as well as handling scaleup and scale down

Space-Based Architecture- Example: Kubernetes

Kubernetes architecture



Space-Based Architecture– Example: Service Mesh on Kubernetes



Architecture is a design activity

A software design is not necessary for trivial systems, but for large systems a design based on a strong software architecture 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

A foundational understanding of software architecture patterns and styles is important, but structuring the architecture is a design activity and not a leggo-block or paint by numbers activity

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

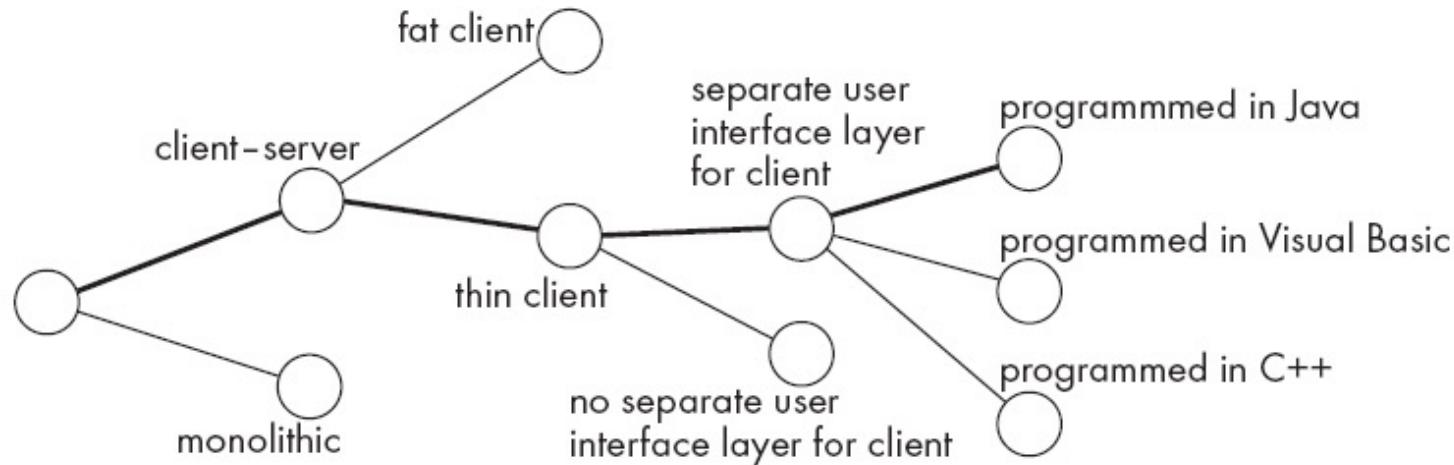
- Definition:
 - *Design* is a problem-solving process whose objective is to find and describe a way:
 - To implement the system's *functional requirements*...
 - While respecting the constraints imposed by the *quality, platform and process requirements*...
 - including the budget
 - And while adhering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

Design as a series of decisions

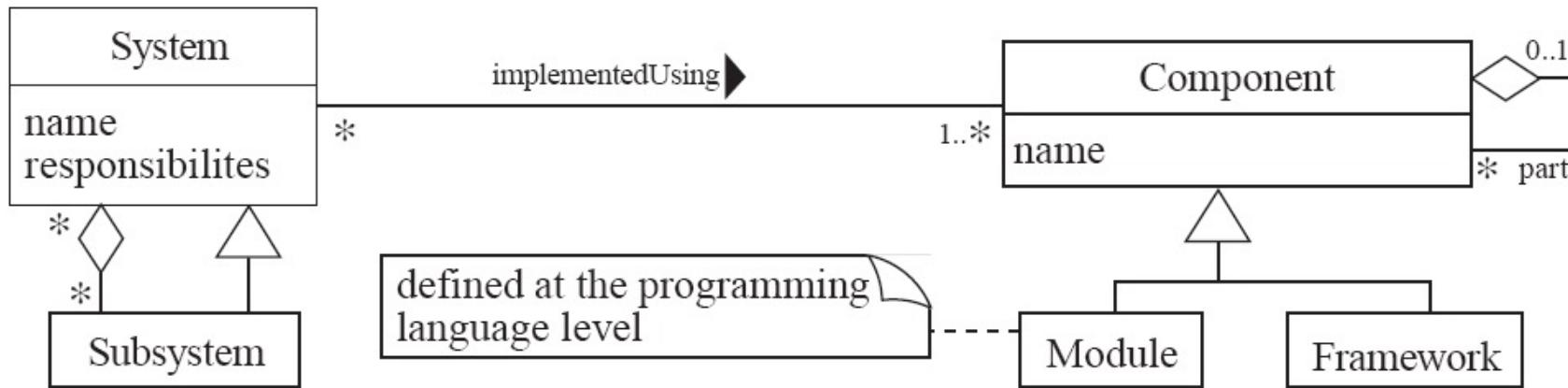
- A designer is faced with a series of *design issues*
 - These are sub-problems of the overall design problem.
 - Each issue normally has several alternative solutions:
 - design *options*.
 - The designer makes a *design decision* to resolve each issue.
 - This process involves choosing the best option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



UML diagram of system parts and their relationships



Principles Leading to Good Design

- Overall *goals* of good design:
 - Increasing profit / reducing cost / increasing revenue / learning opportunity (e.g., open source)
 - 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 things up to manage complexity

- 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

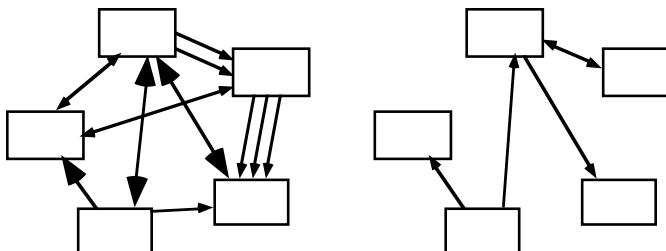
- A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things
 - This makes the system as a whole easier to understand and change
 - Type of cohesion:
 - Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

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., <code>java.lang.math</code>

Design Principle 3: Reduce coupling where possible

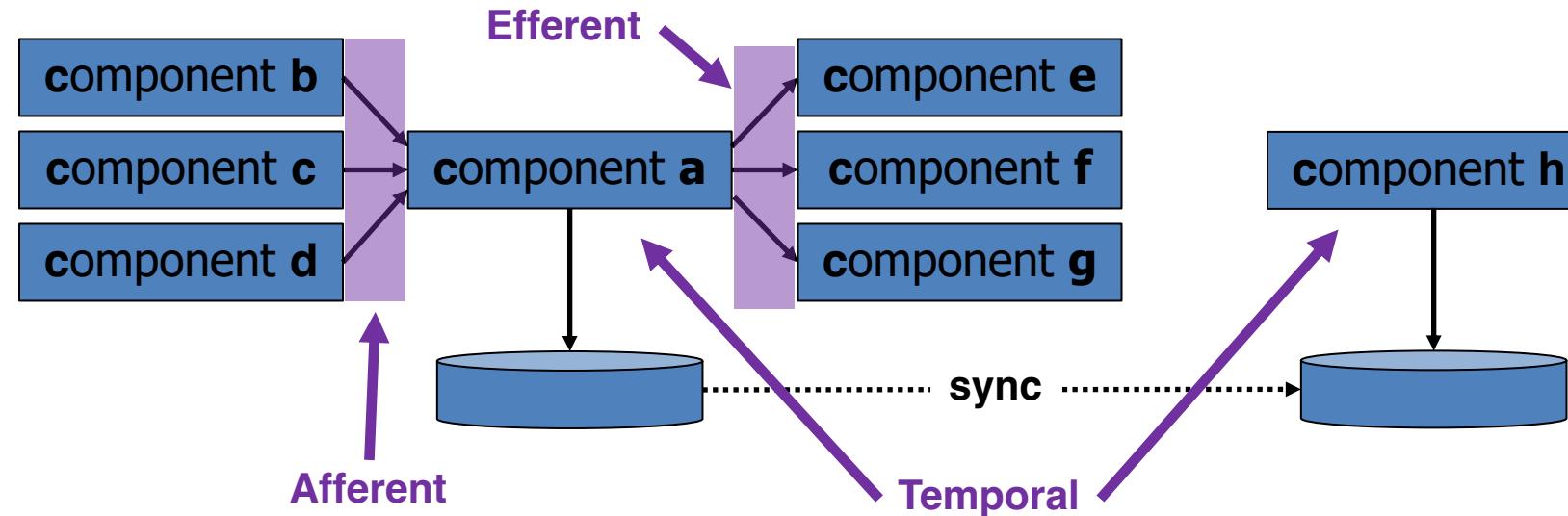
- *Coupling* occurs when there are *interdependencies* between one module and another
 - When interdependencies exist, changes in one place will require changes somewhere else.
 - A network of interdependencies makes it hard to see at a glance how some component works.
 - Type of coupling:
 - Content,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



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

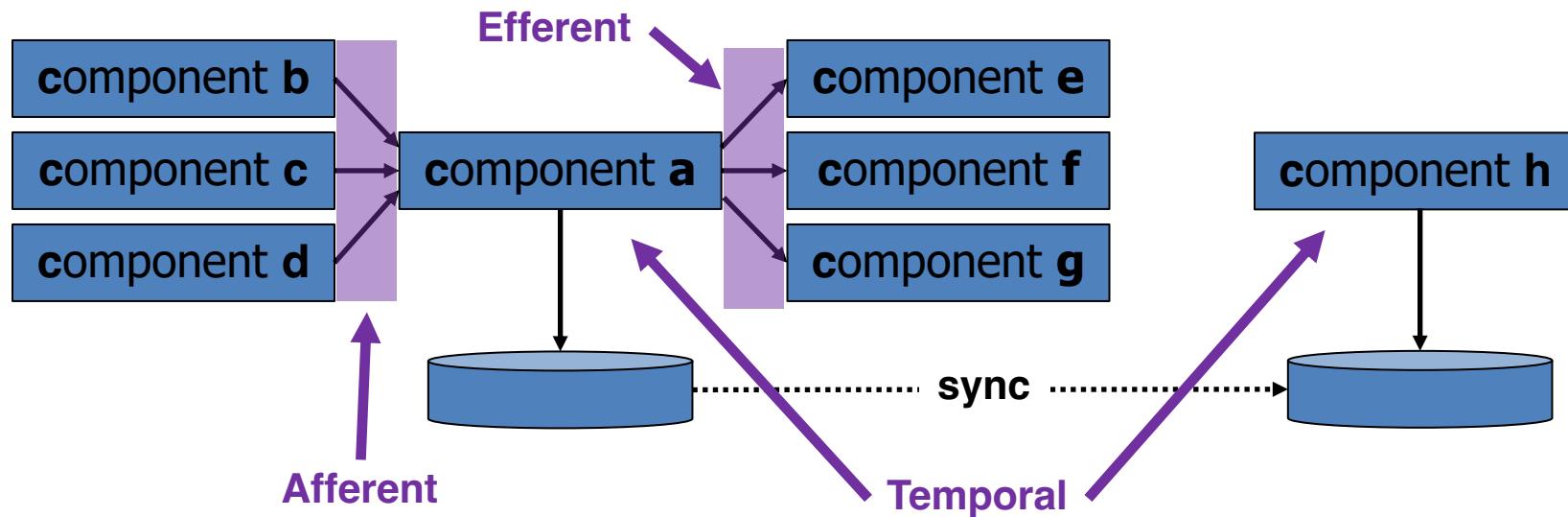
Content	One module modifies internal data of another module
Common	Coupling based on a global variable
Control	Parameter is passed as an argument that controls the behavior of the callee, e.g, ("CustomerArrived" passed as an arg that is then processed by a switch statement
Stamp	Using custom types as 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
 - Use approaches like TDD
 - Ensure that all the functionality of the code can be 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