Architectural Patterns

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

Architectural patterns are similar to DSSAs but applied "at a lower level" and within a much narrower scope.

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

Architectural patterns are similar to DSSAs but applied "at a lower level" and within a much narrower scope.

Domain-Specific Software Architecture

Domain-Specific Software Architectures (DSSA) is an assemblage of software components

- specialized for a particular type of task (domain),
- generalized for effective use across that domain, and
- composed in a standardized structure (topology) effective for building successful applications.

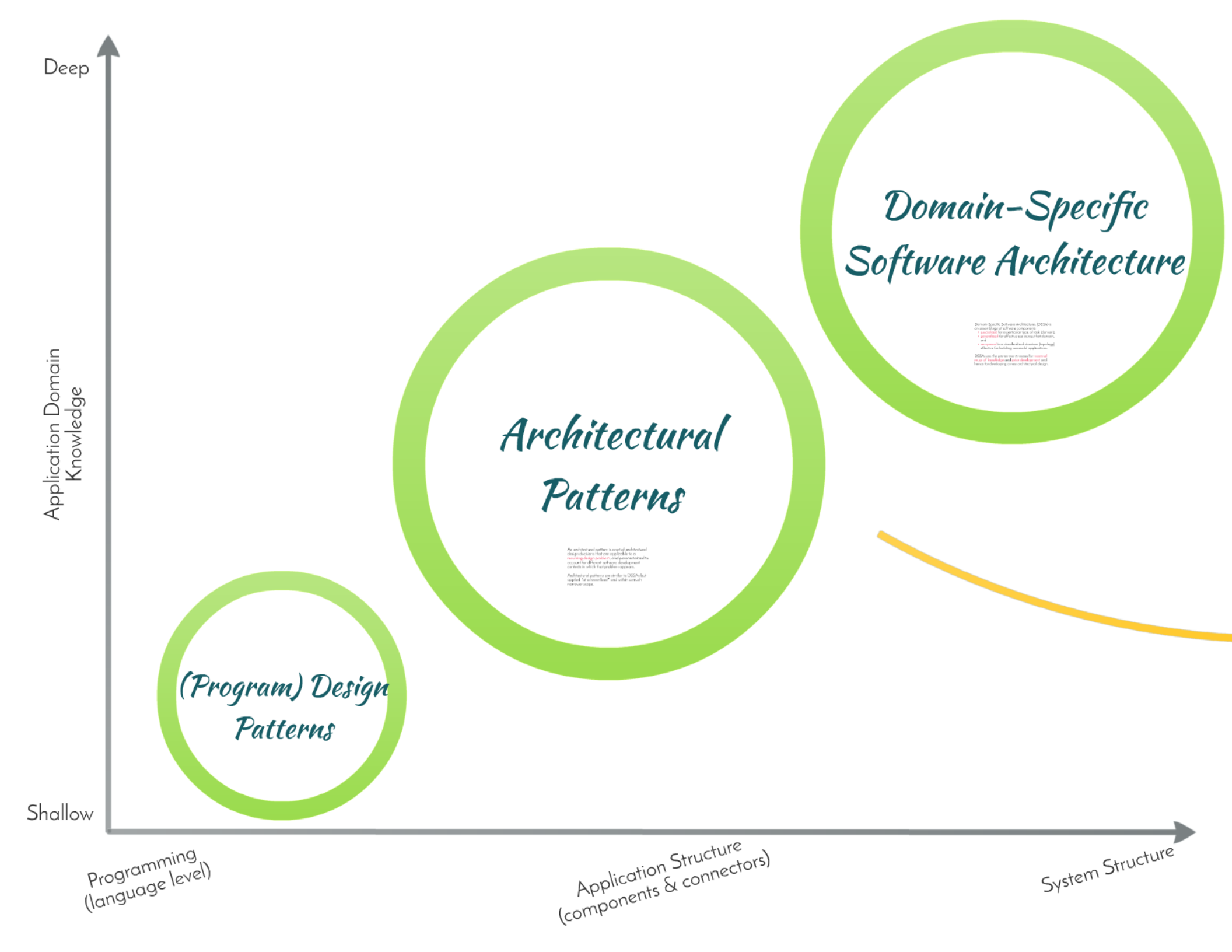
DSSAs are the pre-eminent means for maximal reuse of knowledge and prior development and hence for developing a new architectural design.

Domain-Specific Software Architectures (DSSA) is an assemblage of software components

- specialized for a particular type of task (domain),
- generalized for effective use across that domain, and
- composed in a standardized structure (topology) effective for building successful applications.

DSSAs are the pre-eminent means for maximal reuse of knowledge and prior development and hence for developing a new architectural design.

(Program) Design Patterns


Deep
Application Domain Knowledge
Shallow
Programming (language level)
Application Structure (components & connectors)
System Structure
(Program) Design Patterns
Architectural Patterns
Domain-Specific Software Architecture

Layered Pattern

Pattern

ttern

Sh

Architectural Patterns

Architecture pattern
- is a package of design decisions that is found repeatedly in practice,
- has known properties that permit reuse, and
- describes a class of architectures.

Architecture pattern establishes a relationship between:
- A context: A recurring, common situation in the world that gives rise to a problem.
- A problem: The problem, appropriately generalized, that arises in the given context.
- A solution: A successful architecture resolution to the problem, appropriately abstracted.

Patterns are found in practice
- One does not invent patterns, one discovers them.
- There will never be a complete list of patterns.

del-View-Controller Pattern

Publish-Subscr Pattern

Architecture pattern

- is a package of design decisions that is found repeatedly in practice,
- has known properties that permit reuse, and
- describes a class of architectures.

Architecture pattern establishes a relationship between:

- A context: A recurring, common situation in the world that gives rise to a problem.
- A problem: The problem, appropriately generalized, that arises in the given context.
- A solution: A successful architecture resolution to the problem, appropriately abstracted.

Patterns are found in practice

- One does not invent patterns, one discovers them.
- There will never be a complete list of patterns.

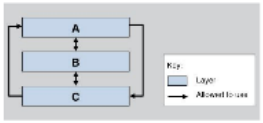

Layered Pattern Solution

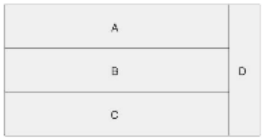
The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers. The pattern is usually shown graphically by stacking boxes representing layers on top of each other.

Layer, a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides.

Allowed to use, which is a specialization of a more generic *depends-on* relation. The design should define what the layer usage rules are (e.g., "a layer is allowed to use any lower layer" or "a layer is allowed to use only the layer immediately below it") and any allowable exceptions.

- Every piece of software is allocated to exactly one layer.
- There are at least two layers (but usually there are three or more).
- The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).
- The addition of layers adds up-front cost and complexity to a system.
- Layers contribute a performance penalty.



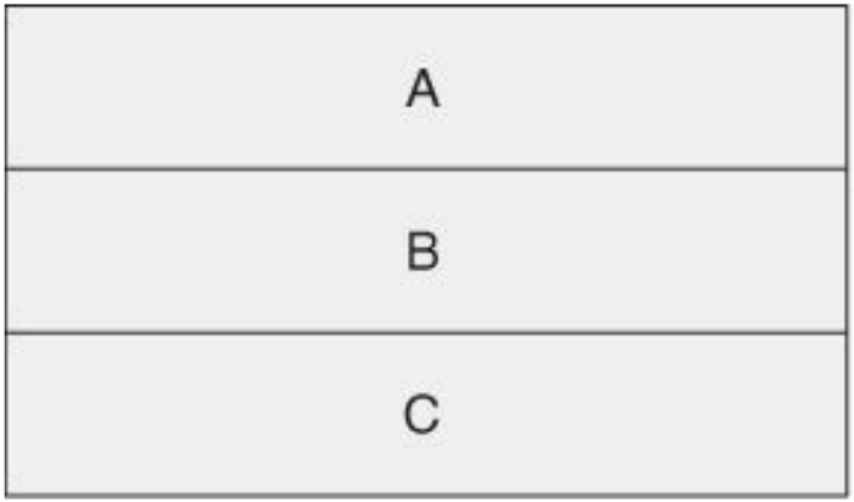


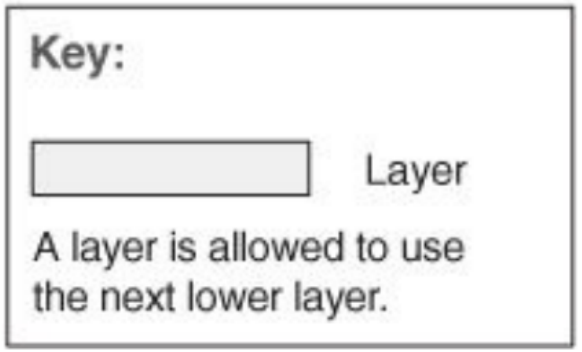
Layered Pattern

Layered Pattern Solution

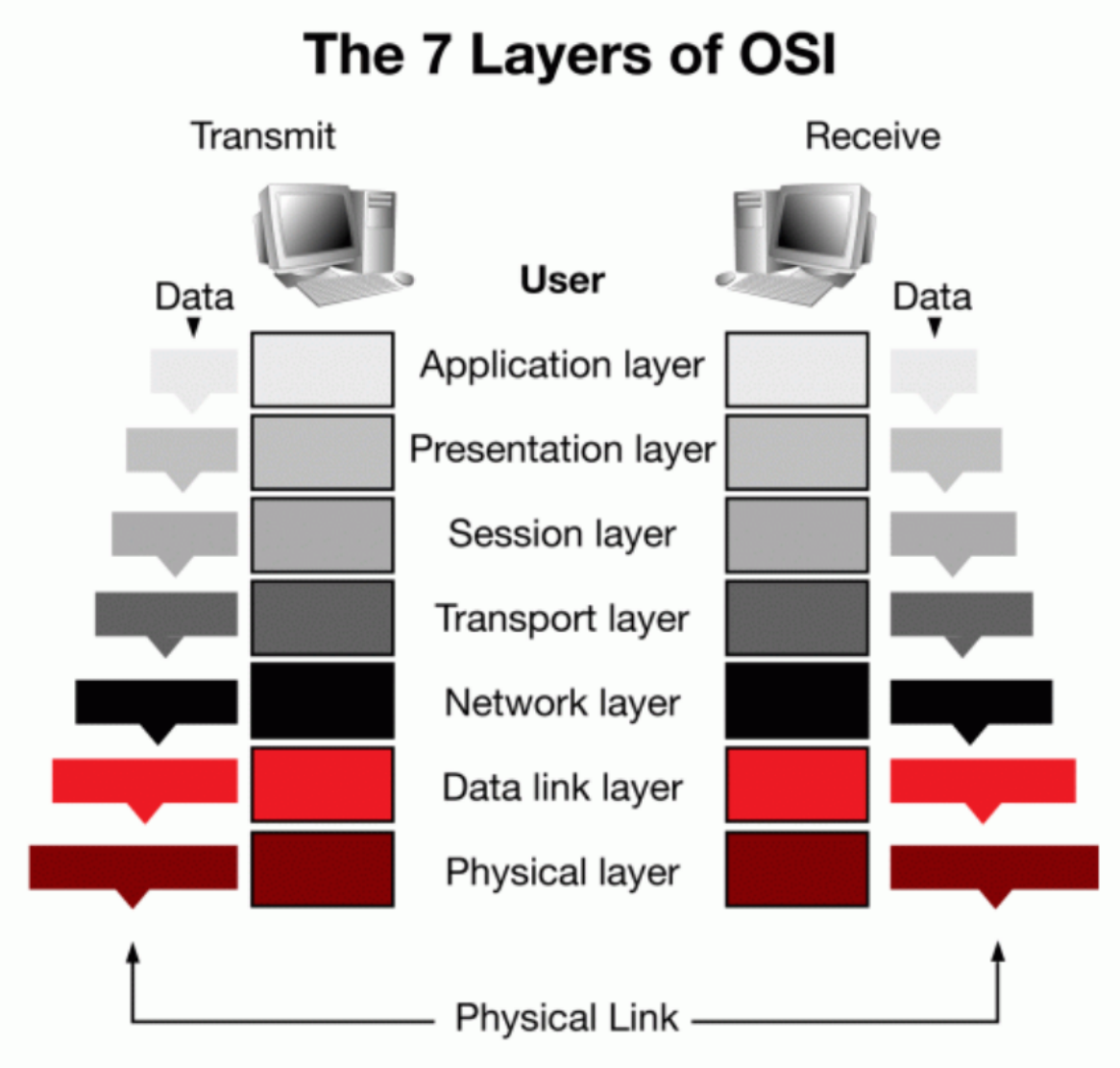
Overview	The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers. The pattern is usually shown graphically by stacking boxes representing layers on top of each other.
Elements	*Layer*, a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides.
Relations	*Allowed to use*, which is a specialization of a more generic *depends-on* relation. The design should define what the layer usage rules are (e.g., "a layer is allowed to use any lower layer" or "a layer is allowed to use only the layer immediately below it") and any allowable exceptions.
Constraints	▪ Every piece of software is allocated to exactly one layer. ▪ There are at least two layers (but usually there are three or more). ▪ The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).
Weaknesses	▪ The addition of layers adds up-front cost and complexity to a system. ▪ Layers contribute a performance penalty.

Stack-of-boxes Notation

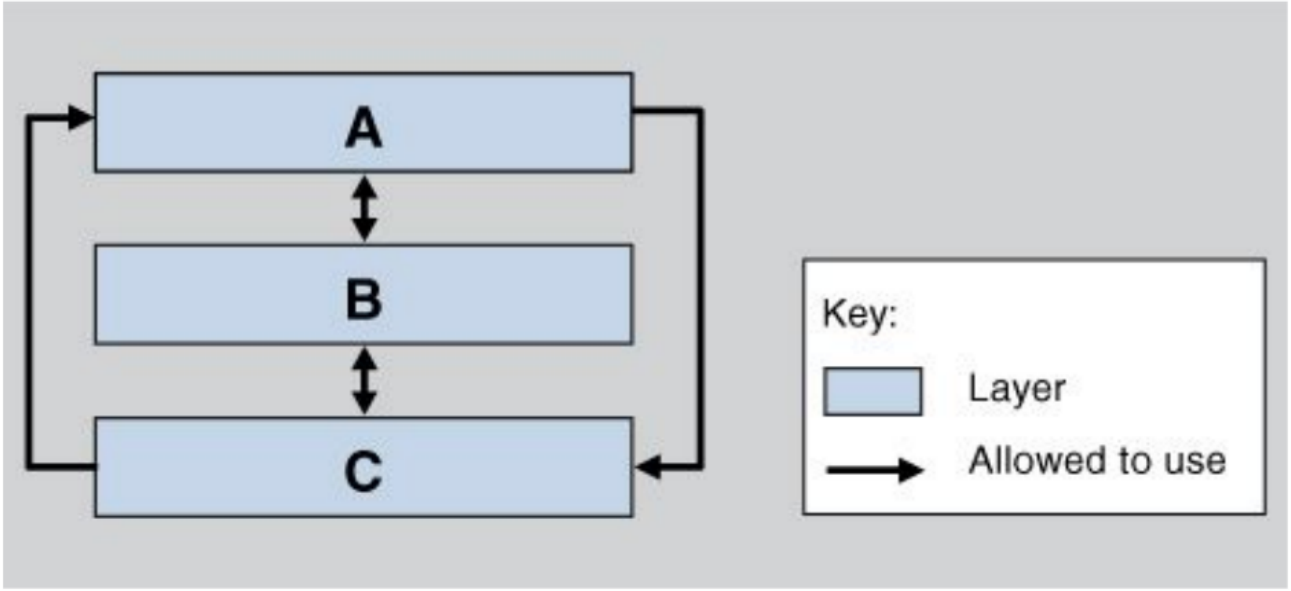


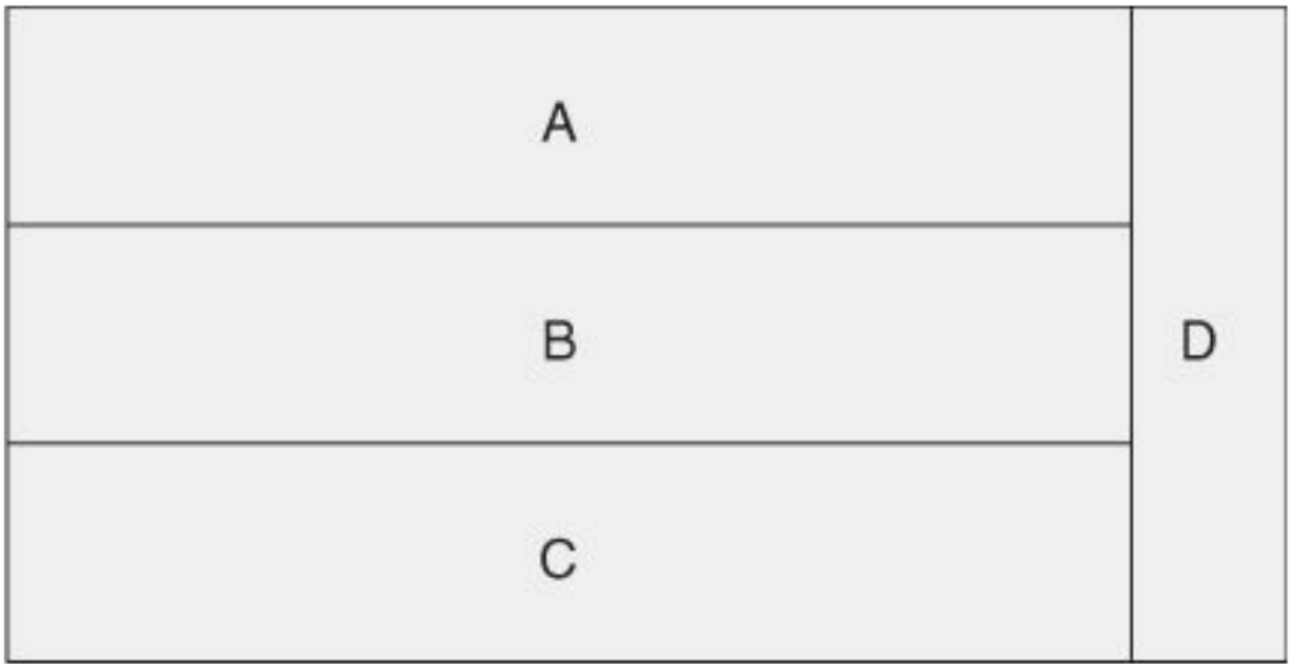


Layered Pattern Based System



Layered Pattern Variants



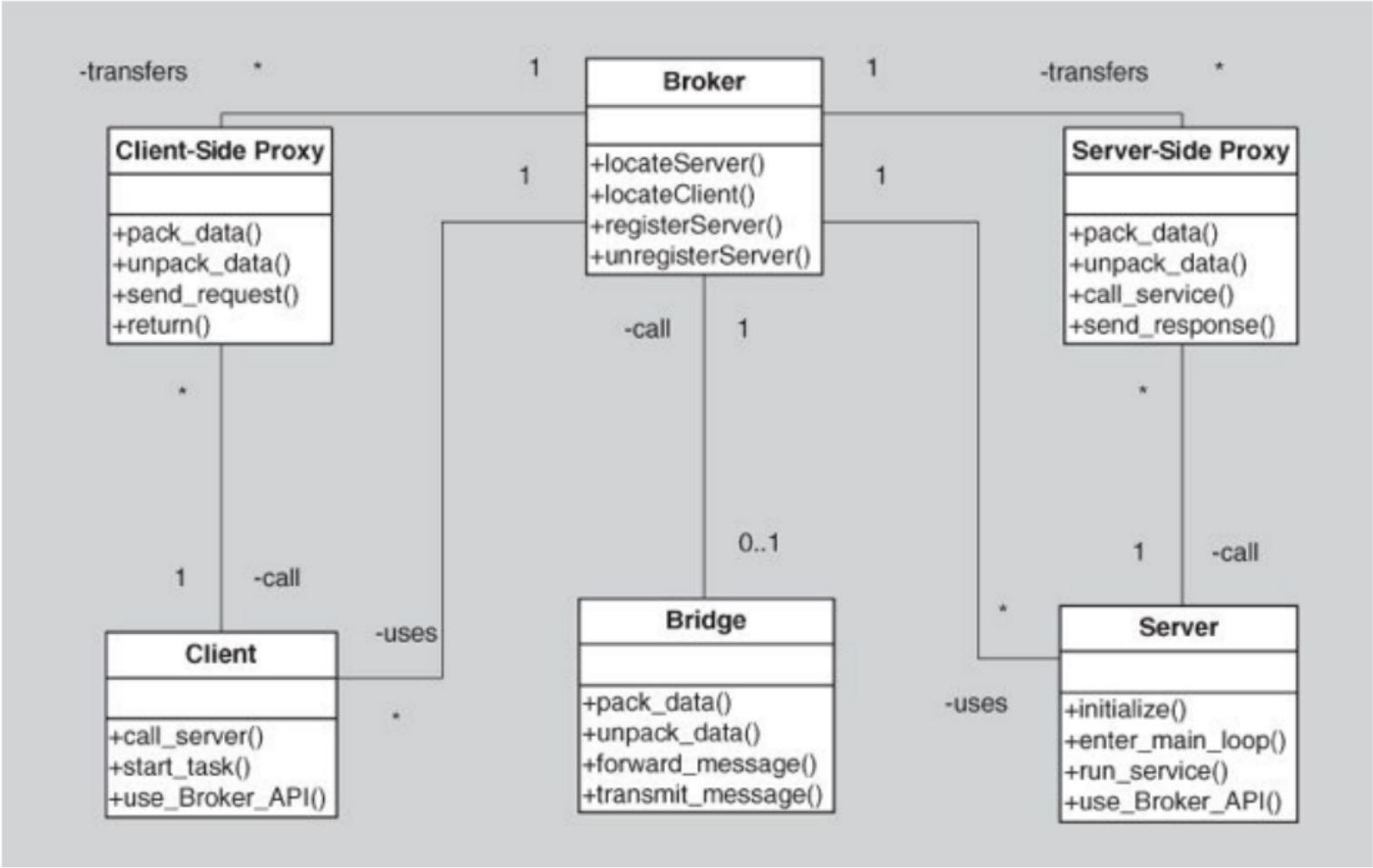


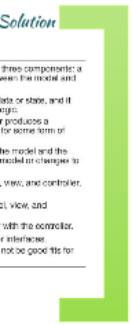
Broker Pattern

Broker Pattern Solution

Overview	The broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers.
Elements	*Client,* a requester of services *Server,* a provider of services *Broker,* an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client *Client-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages *Server-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
Relations	The *attachment* relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.
Constraints	The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy).
Weaknesses	Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck. The broker can be a single point of failure. A broker adds up-front complexity. A broker may be a target for security attacks. A broker may be difficult to test.

The Broker Pattern



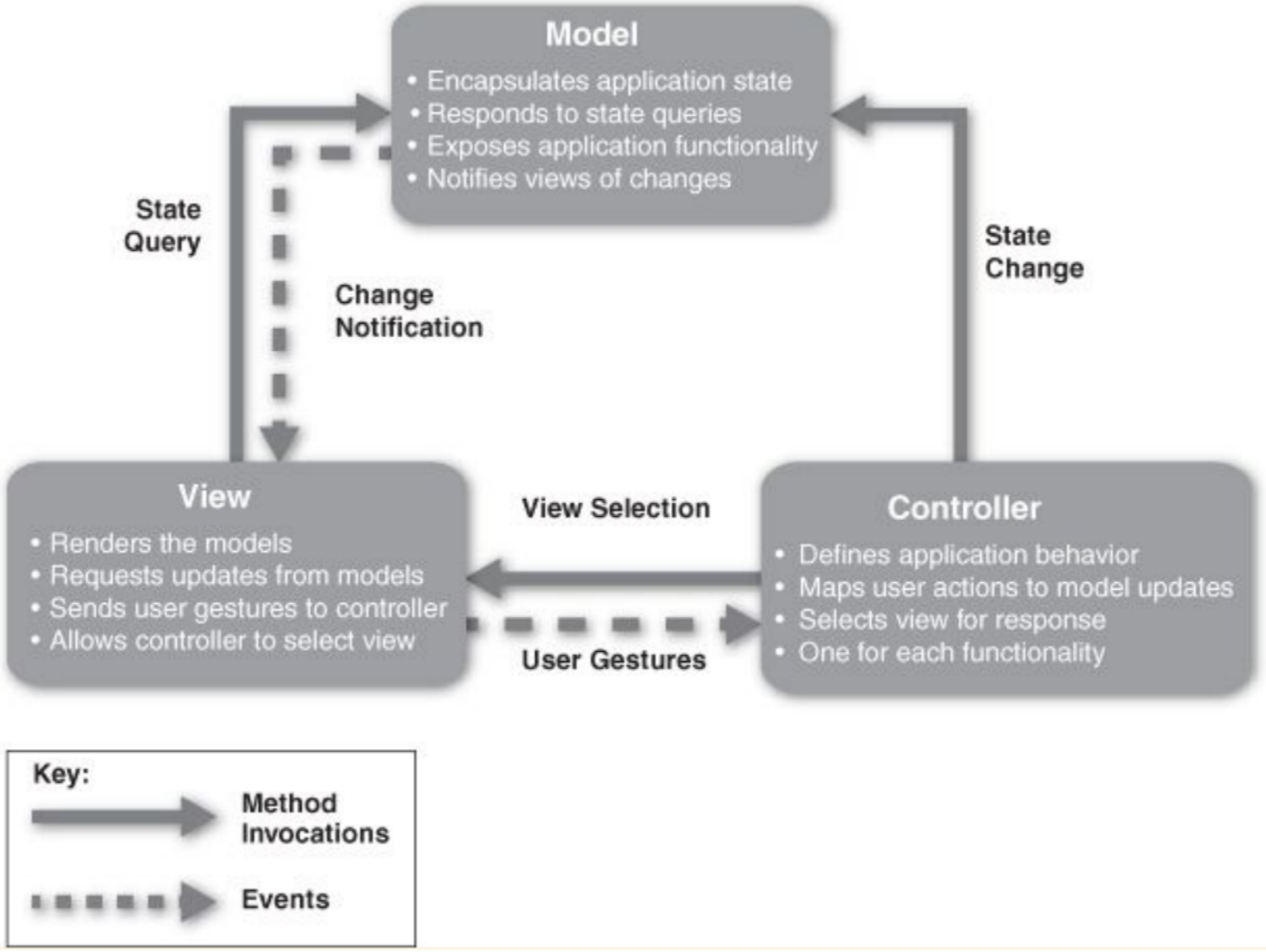


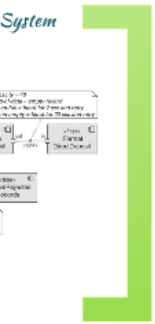
Model-View-Controller Pattern

Model-View-Controller Pattern Solution

Overview	The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.
Elements	The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic. The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both. The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.
Relations	The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes.
Constraints	There must be at least one instance each of model, view, and controller. The model component should not interact directly with the controller.
Weaknesses	The complexity may not be worth it for simple user interfaces. The model, view, and controller abstractions may not be good fits for some user interface toolkits.

Model-View-Controller Pattern



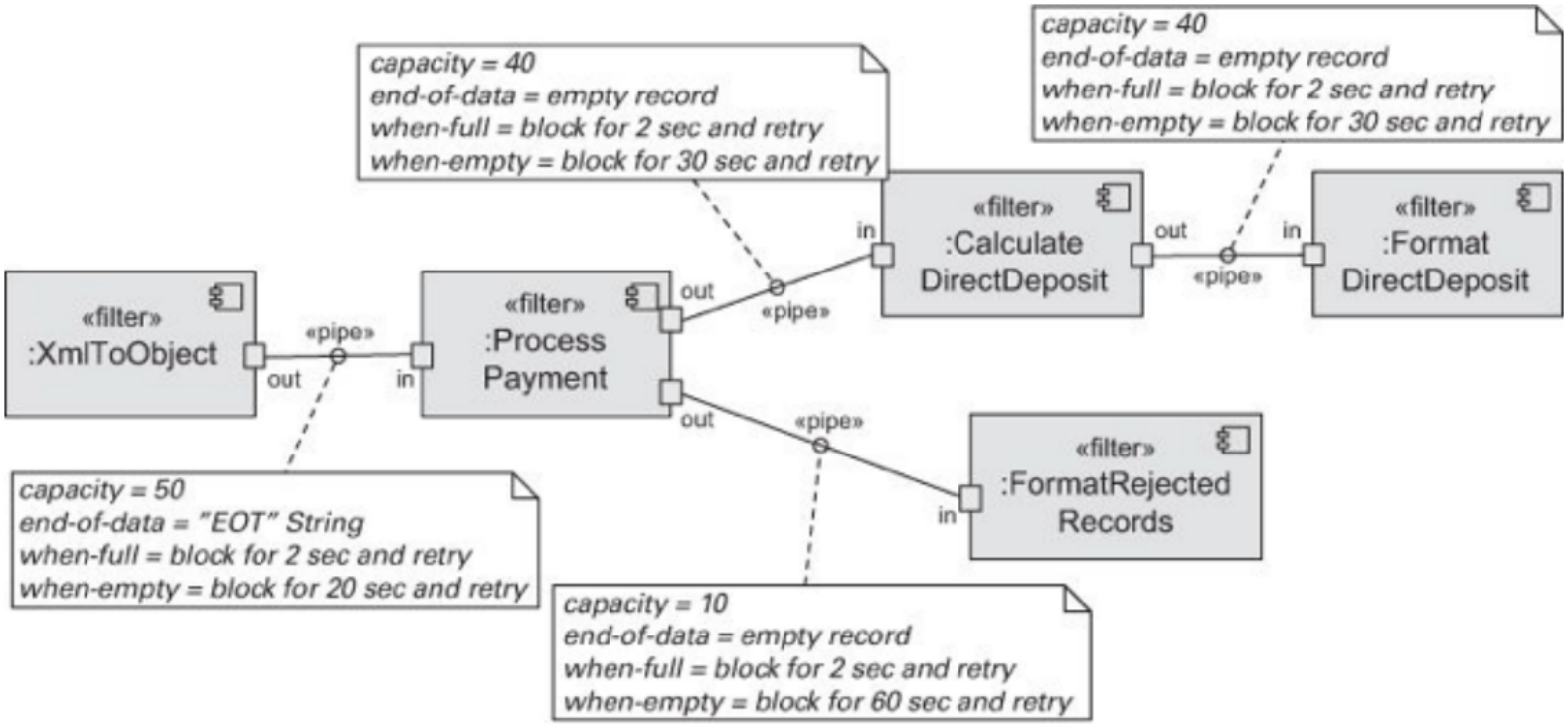


Pipe-and-Filter Pattern

Pipe-and-Filter Pattern Solution

Overview	Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.
Elements	*Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s). Filters can execute concurrently with each other. Filters can incrementally transform data; that is, they can start producing output as soon as they start processing input. Important characteristics include processing rates, input/output data formats, and the transformation executed by the filter. *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through. Important characteristics include buffer size, protocol of interaction, transmission speed, and format of the data that passes through a pipe.
Relations	The *attachment* relation associates the output of filters with the input of pipes and vice versa.
Constraints	Pipes connect filter output ports to filter input ports. Connected filters must agree on the type of data being passed along the connecting pipe. Specializations of the pattern may restrict the association of components to an acyclic graph or a linear sequence, sometimes called a pipeline. Other specializations may prescribe that components have certain named ports, such as the *stdin*, *stdout*, and *stderr* ports of UNIX filters.
Weaknesses	The pipe-and-filter pattern is typically not a good choice for an interactive system. Having large numbers of independent filters can add substantial amounts of computational overhead. Pipe-and-filter systems may not be appropriate for long-running computations.

Pipe-and-Filter Based System

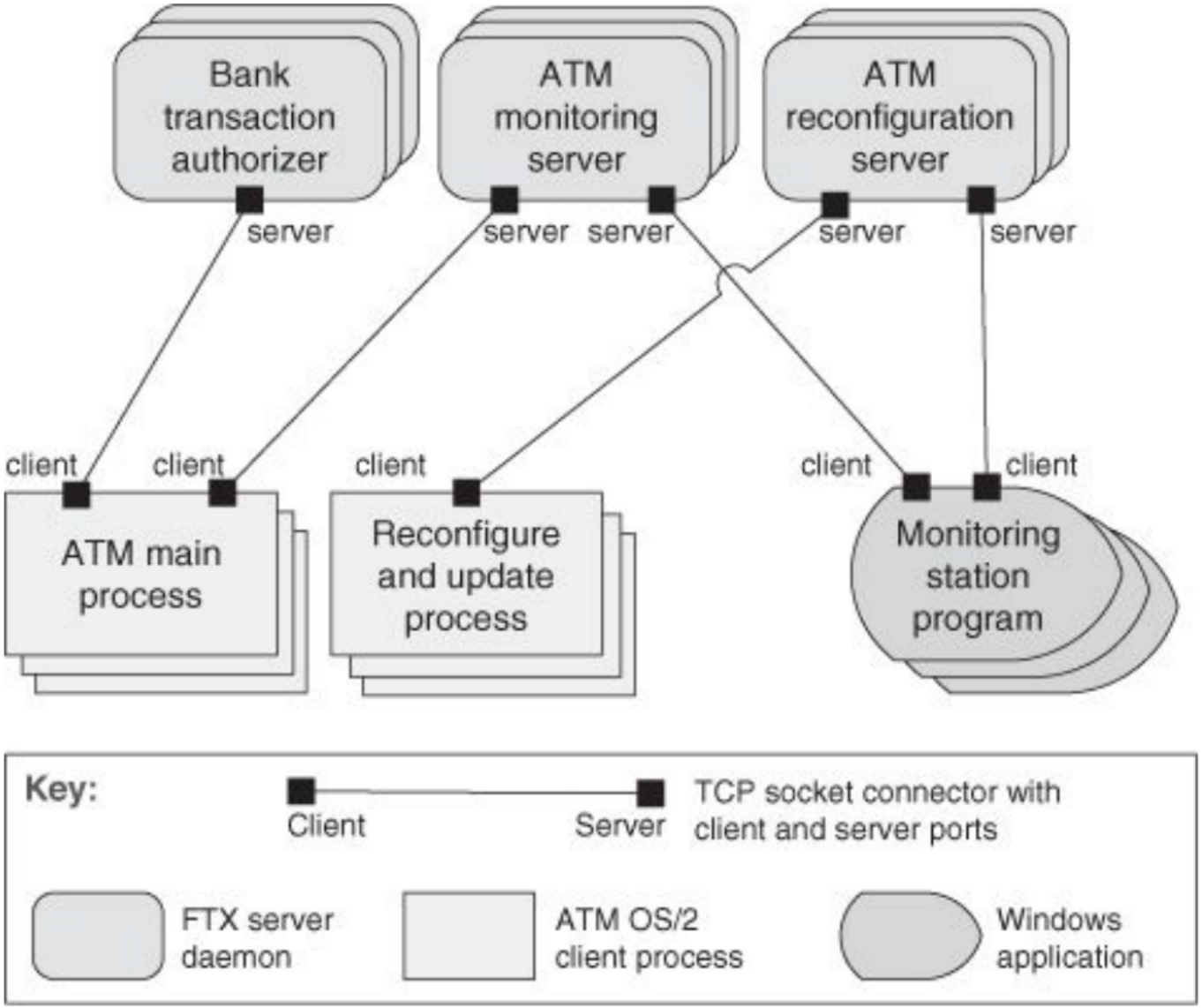


Client-Server Pattern

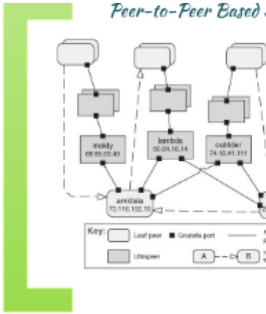
Client-Server Pattern Solution

Overview	Clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests.
Elements	*Client,* a component that invokes services of a server component. Clients have ports that describe the services they require. *Server,* a component that provides services to clients. Servers have ports that describe the services they provide. Important characteristics include information about the nature of the server ports (such as how many clients can connect) and performance characteristics (e.g., maximum rates of service invocation). *Request/reply connector,* a data connector employing a request/reply protocol, used by a client to invoke services on a server. Important characteristics include whether the calls are local or remote, and whether data is encrypted.
Relations	The *attachment* relation associates clients with servers.
Constraints	Clients are connected to servers through request/reply connectors. Server components can be clients to other servers. Specializations may impose restrictions: ▪ Numbers of attachments to a given port ▪ Allowed relations among servers Components may be arranged in tiers, which are logical groupings of related functionality or functionality that will share a host computing environment (covered more later in this chapter).
Weaknesses	Server can be a performance bottleneck. Server can be a single point of failure. Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

Client-Server Based System



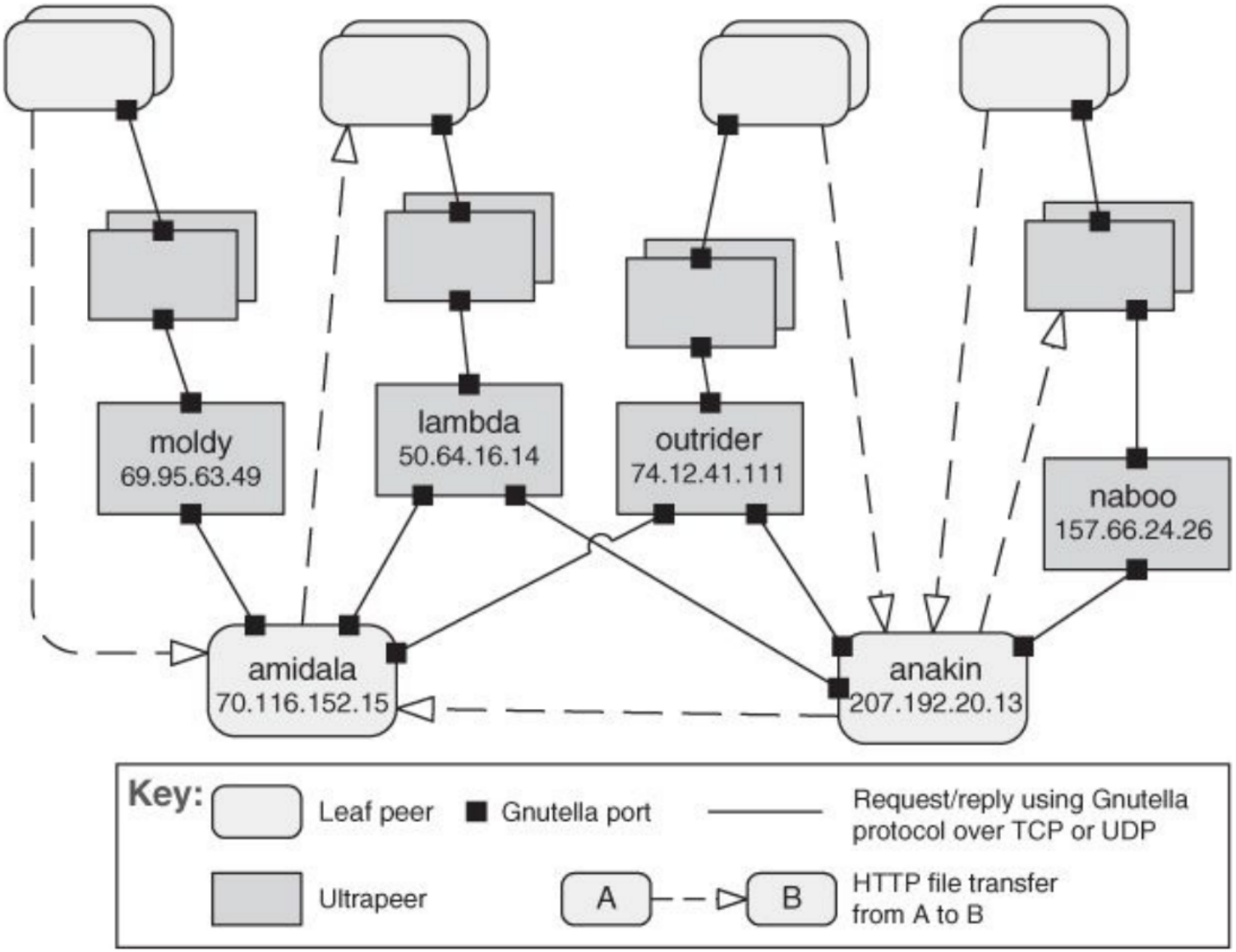
Peer-to-Peer Pattern



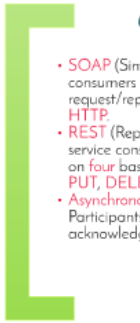
Peer-to-Peer Pattern Solution

Overview	Computation is achieved by cooperating peers that request service from and provide services to one another across a network.
Elements	*Peer,* which is an independent component running on a network node. Special peer components can provide routing, indexing, and peer search capability. *Request/reply connector,* which is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply is done away with.
Relations	The relation associates peers with their connectors. Attachments may change at runtime.
Constraints	Restrictions may be placed on the following: ▪ The number of allowable attachments to any given peer ▪ The number of hops used for searching for a peer ▪ Which peers know about which other peers Some P2P networks are organized with star topologies, in which peers only connect to supernodes.
Weaknesses	Managing security, data consistency, data/service availability, backup, and recovery are all more complex. Small peer-to-peer systems may not be able to consistently achieve quality goals such as performance and availability.

Peer-to-Peer Based System



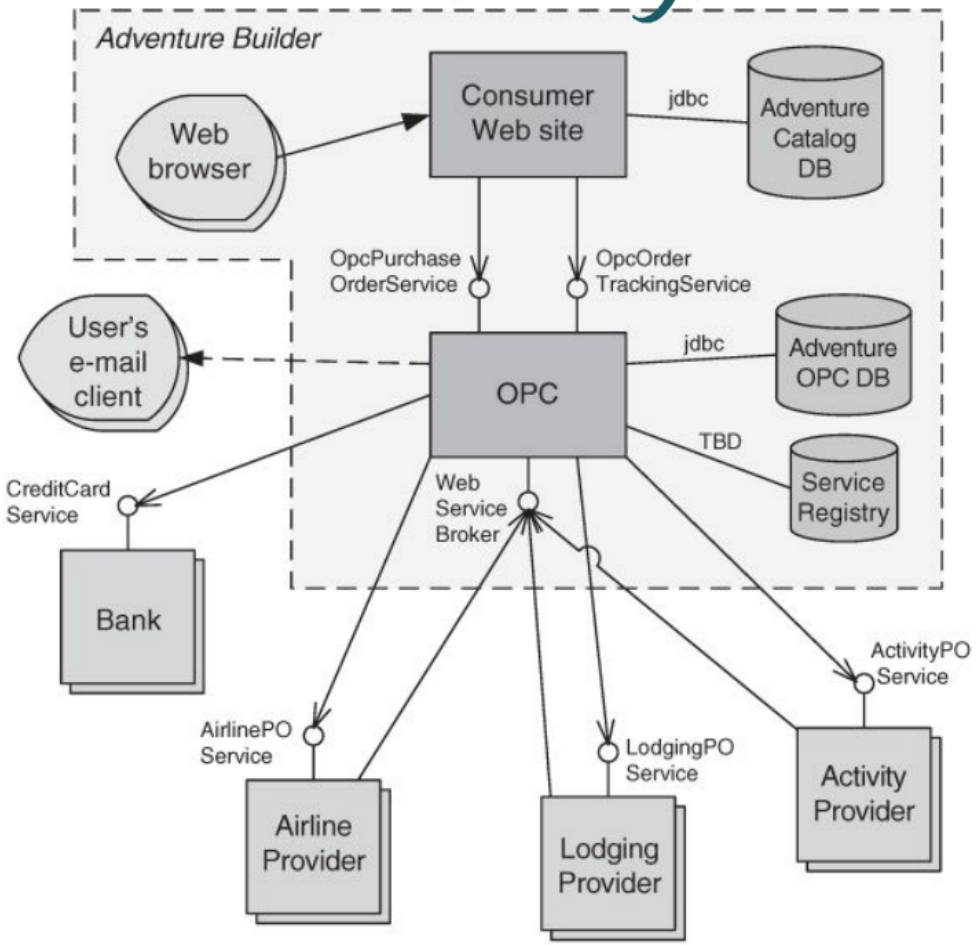
Service-Oriented Pattern

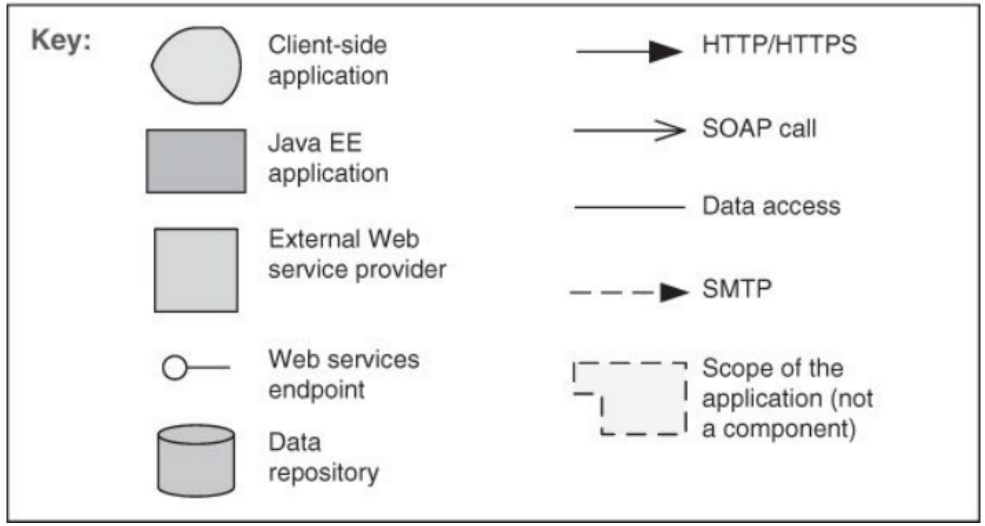


SOA Pattern Solution

Overview	Computation is achieved by a set of cooperating components that provide and/or consume services over a network. The computation is often described using a workflow language.
Elements	Components: ▪ *Service providers,* which provide one or more services through published interfaces. Concerns are often tied to the chosen implementation technology, and include performance, authorization constraints, availability, and cost. In some cases these properties are specified in a service-level agreement. ▪ *Service consumers,* which invoke services directly or through an intermediary. ▪ *Service providers* may also be service consumers. ▪ *ESB,* which is an intermediary element that can route and transform messages between service providers and consumers. ▪ *Registry of services,* which may be used by providers to register their services and by consumers to discover services at runtime. ▪ *Orchestration server,* which coordinates the interactions between service consumers and providers based on languages for business processes and workflows. Connectors: ▪ *SOAP connector,* which uses the SOAP protocol for synchronous communication between web services, typically over HTTP. ▪ *REST connector,* which relies on the basic request/reply operations of the HTTP protocol. ▪ *Asynchronous messaging connector,* which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.
Relations	Attachment of the different kinds of components available to the respective connectors
Constraints	Service consumers are connected to service providers, but intermediary components (e.g., ESB, registry, orchestration server) may be used.
Weaknesses	SOA-based systems are typically complex to build. You don't control the evolution of independent services. There is a performance overhead associated with the middleware, and services may be performance bottlenecks, and typically do not provide performance guarantees.

SOA Based System





Connectors in SOA

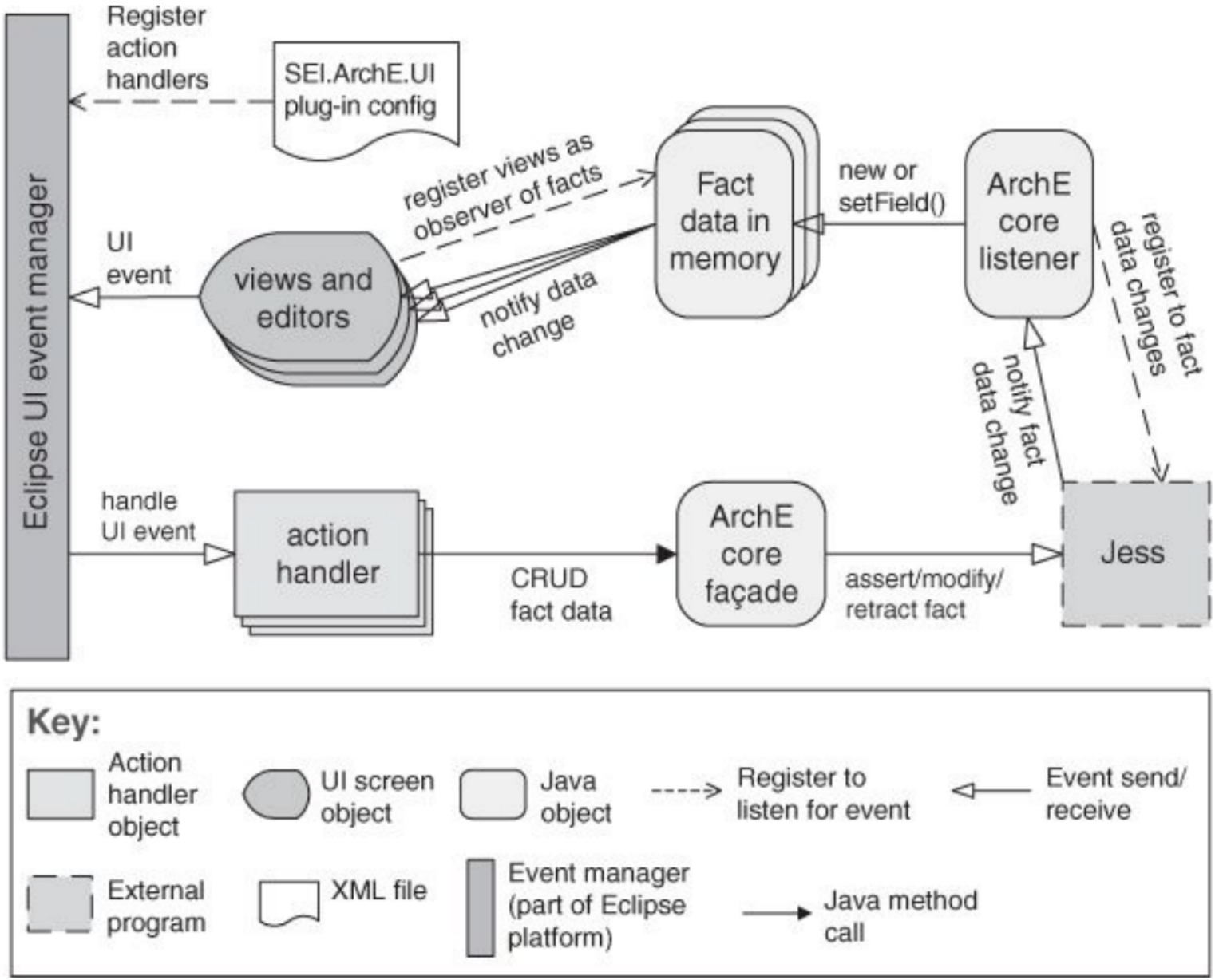
- SOAP (Simple Object Access Protocol): Service consumers and providers interact by exchanging request/reply XML messages typically on top of HTTP.
- REST (Representational State Transfer) : A service consumer sends HTTP requests that rely on four basic HTTP commands (POST, GET, PUT, DELETE)
- Asynchronous messaging ("fire-and-forget"): Participants do not have to wait for an acknowledgment of receipt.

Publish-Subscribe Pattern

Publish-Subscribe Pattern Solution

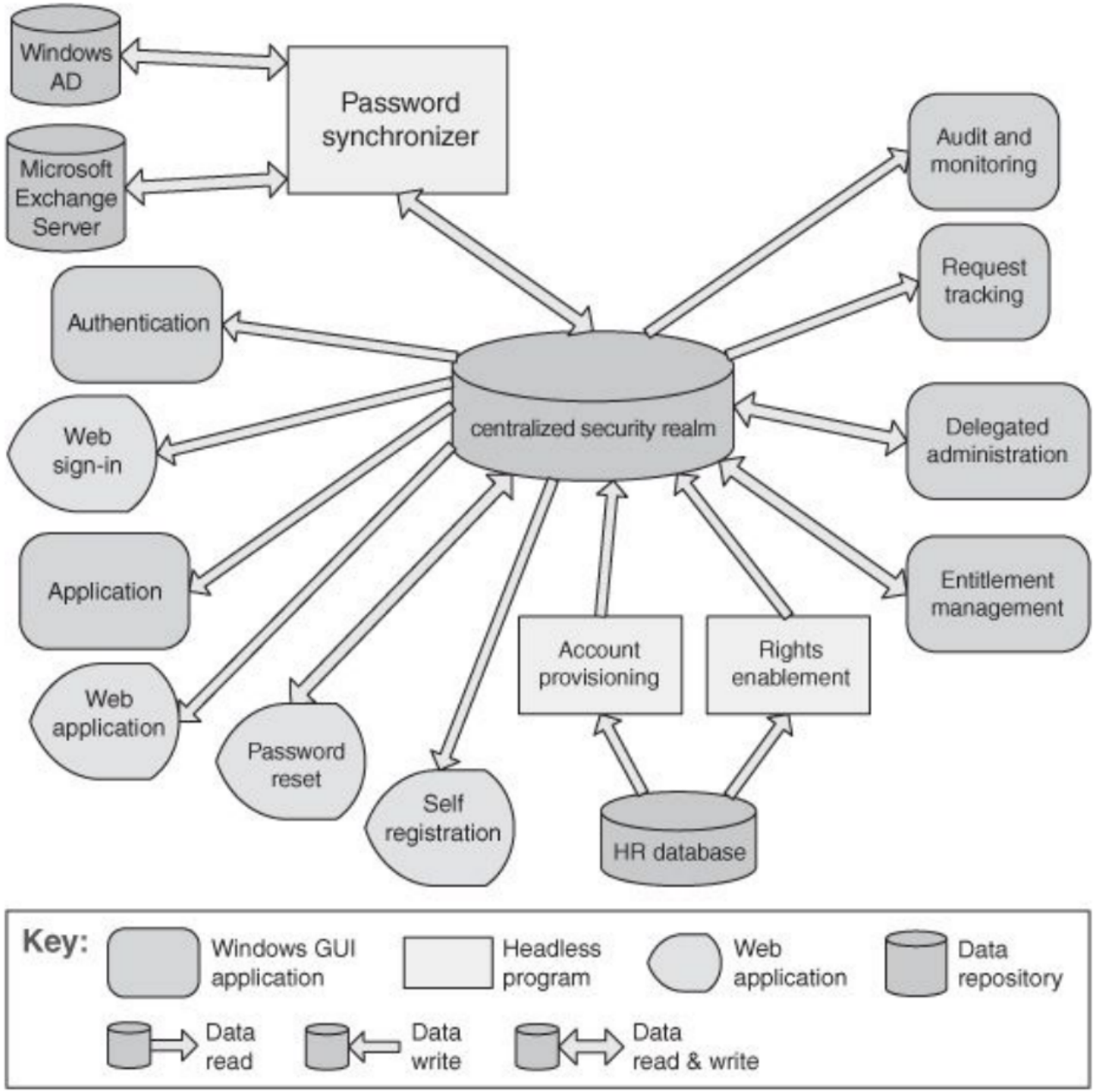
Overview	Components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers.
Elements	*Any C&C component* with at least one publish or subscribe port. Concerns include which events are published and subscribed to, and the granularity of events. *The publish-subscribe connector*, which will have *announce* and *listen* roles for components that wish to publish and subscribe to events.
Relations	The *attachment* relation associates components with the publish-subscribe connector by prescribing which components announce events and which components are registered to receive events.
Constraints	All components are connected to an event distributor that may be viewed as either a bus—connector—or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles. Constraints may restrict which components can listen to which events, whether a component can listen to its own events, and how many publish-subscribe connectors can exist within a system. A component may be both a publisher and a subscriber, by having ports of both types.
Weaknesses	Typically increases latency and has a negative effect on scalability and predictability of message delivery time. Less control over ordering of messages, and delivery of messages is not guaranteed.

Publish-Subscribe Based System



Shared-Data Pattern

Shared-Data Based System



Shared-Data Pattern Solution

Overview	Communication between data accessors is mediated by a shared-data store. Control may be initiated by the data accessors or the data store. Data is made persistent by the data store.
Elements	*Shared-data store.* Concerns include types of data stored, data performance-oriented properties, data distribution, and number of accessors permitted. *Data accessor component.* *Data reading and writing connector.* An important choice here is whether the connector is transactional or not, as well as the read/write language, protocols, and semantics.
Relations	*Attachment* relation determines which data accessors are connected to which data stores.
Constraints	Data accessors interact with the data store(s).
Weaknesses	The shared-data store may be a performance bottleneck. The shared-data store may be a single point of failure. Producers and consumers of data may be tightly coupled.

Architectural Patterns

Architecture pattern
- is a package of design decisions that is found repeatedly in practice,
- has known properties that permit reuse, and
- describes a class of architectures.

Architecture pattern establishes a relationship between
- A context: A recurring, common situation in the world that gives rise to a problem.
- A problem: The problem, appropriately generalized, that arises in the given context.
- A solution: A successful architecture resolution to the problem, appropriately abstracted.

Patterns are found in practice
- One does not invent patterns, one discovers them.
- There will never be a complete list of patterns.

Component-Connector Patterns

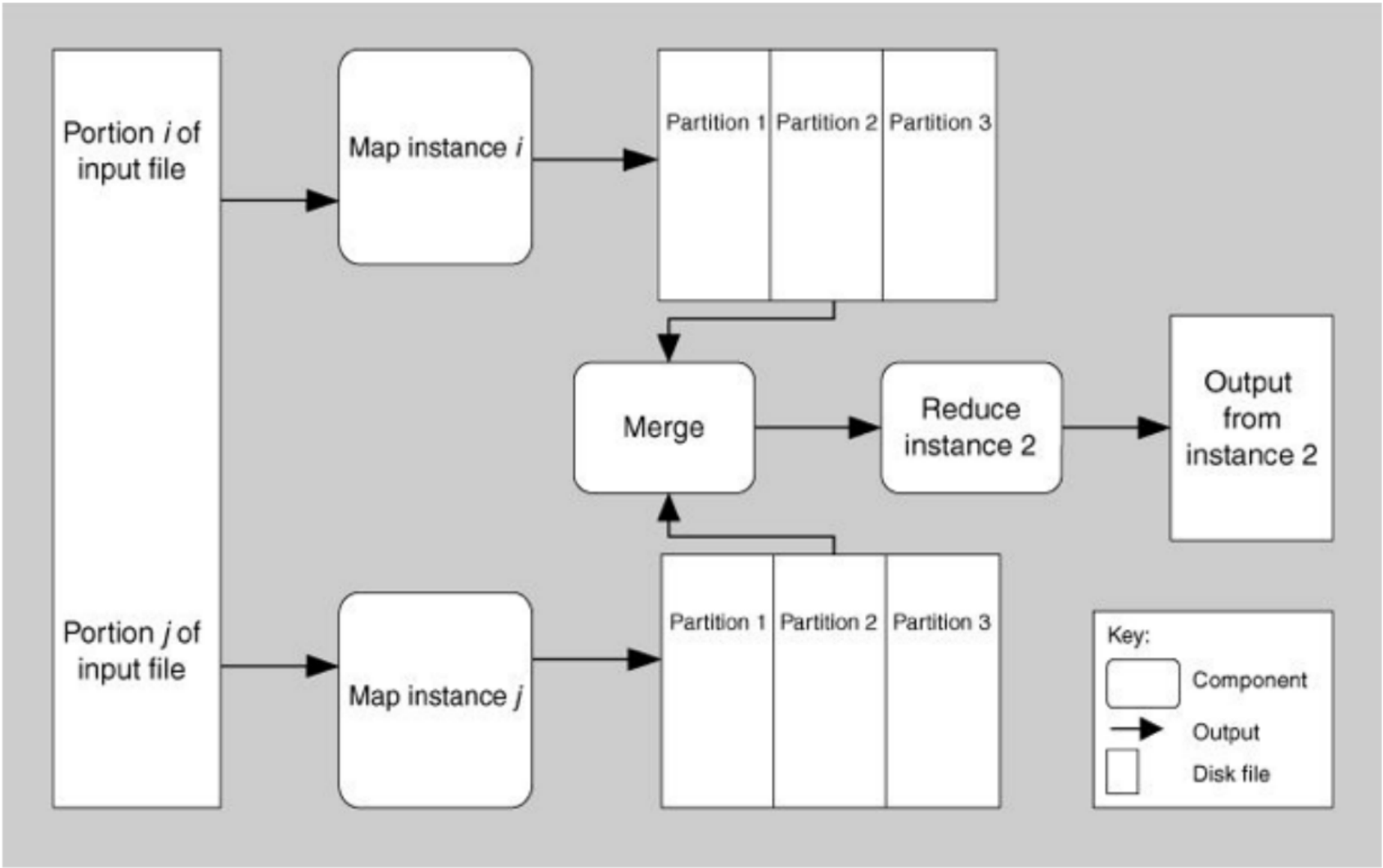
- Broker Pattern
- Shared-Data Pattern
- Model-View-Controller Pattern
- Publish-Subscribe Pattern
- Client-Server Pattern
- Peer-to-Peer Pattern
- Pipe-and-Filter Pattern
- Service-Oriented Pattern


Map-Reduce
Pattern

Map-Reduce Pattern Solution

Overview	The map-reduce pattern provides a framework for analyzing a large distributed set of data that will execute in parallel, on a set of processors. This parallelization allows for low latency and high availability. The map performs the *extract* and *transform* portions of the analysis and the reduce performs the *loading* of the results. (*Extract-transform-load* is sometimes used to describe the functions of the map and reduce.)
Elements	*Map* is a function with multiple instances deployed across multiple processors that performs the extract and transformation portions of the analysis. *Reduce* is a function that may be deployed as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load. The *infrastructure* is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure.
Relations	*Deploy on* is the relation between an instance of a map or reduce function and the processor onto which it is installed. *Instantiate, monitor, and control* is the relation between the infrastructure and the instances of map and reduce.
Constraints	The data to be analyzed must exist as a set of files. The map functions are stateless and do not communicate with each other. The only communication between the map instances and the reduce instances is the data emitted from the map instances as <key, value> pairs.
Weaknesses	If you do not have large data sets, the overhead of map-reduce is not justified. If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost. Operations that require multiple reduces are complex to orchestrate.

Map-Reduce Based System

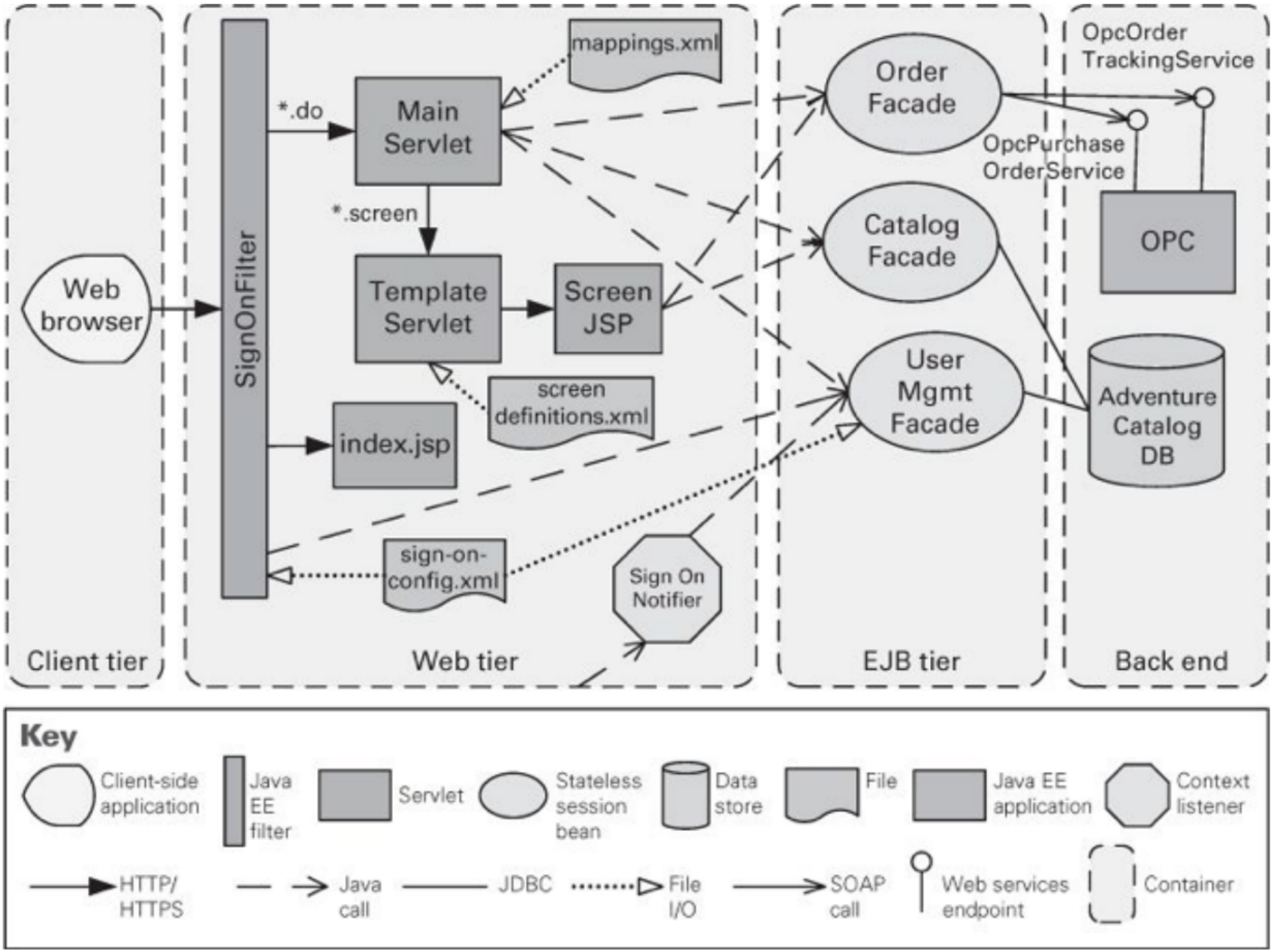


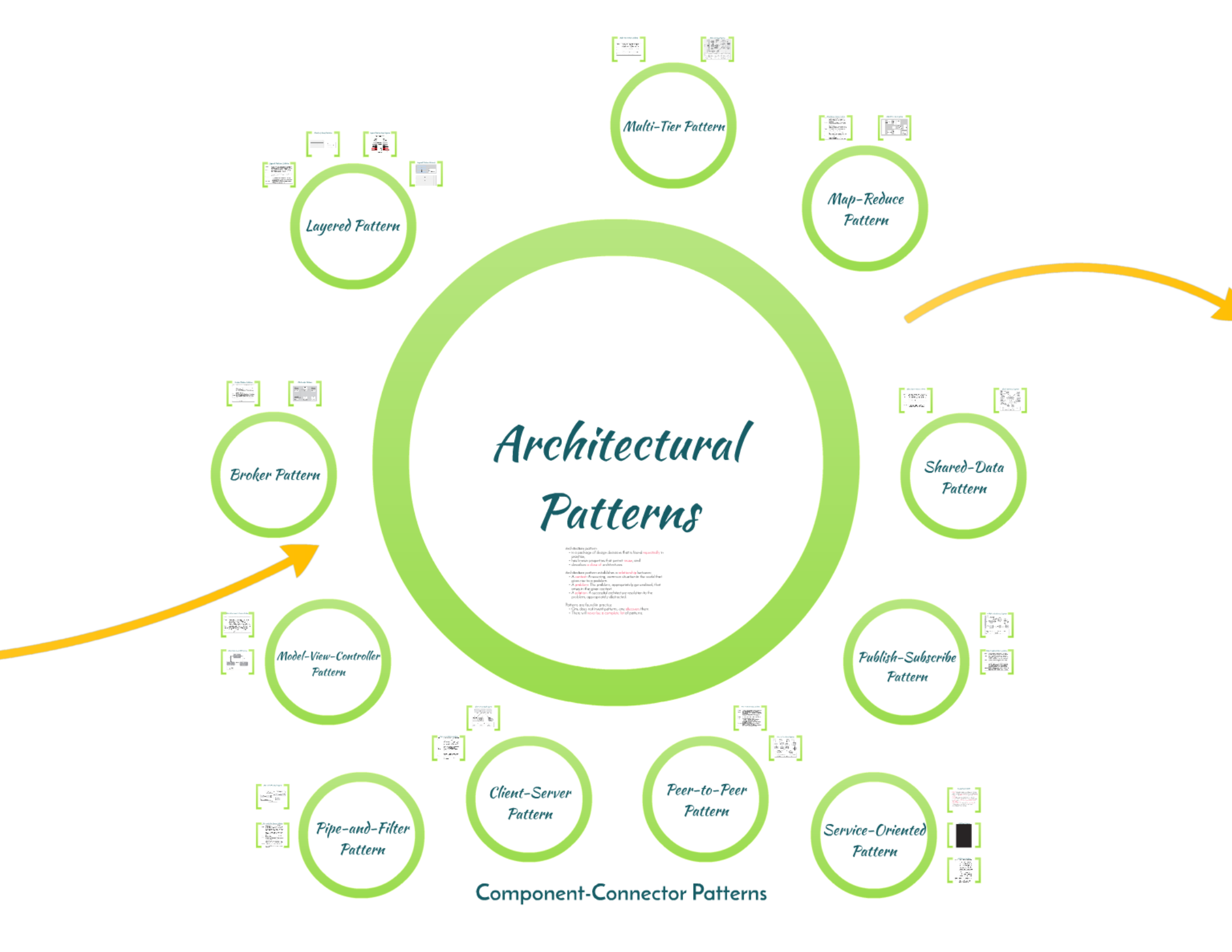

Multi-Tier Pattern

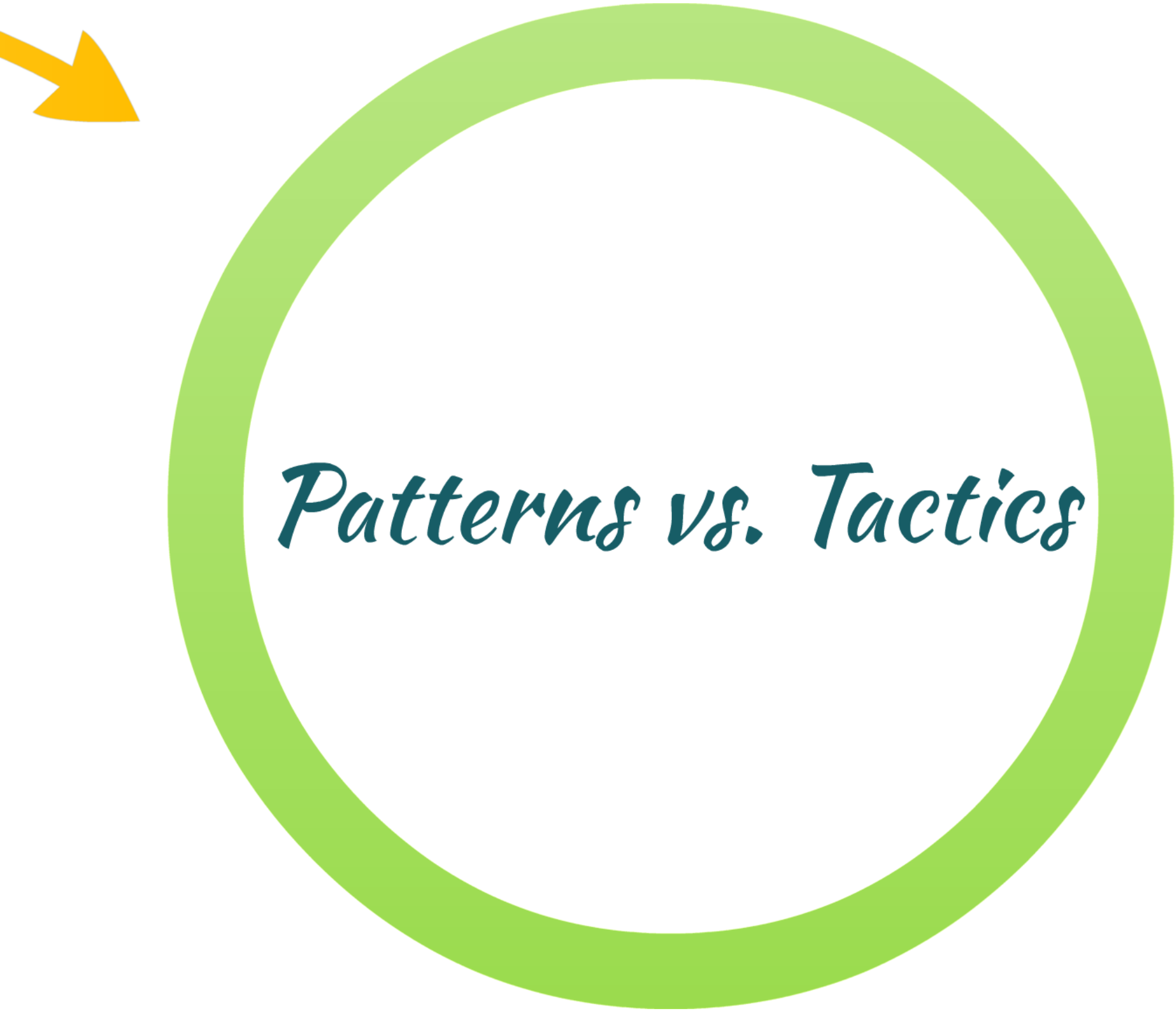
Multi-Tier Pattern Solution

Overview	The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a *tier*. The grouping of components into tiers may be based on a variety of criteria, such as the type of component, sharing the same execution environment, or having the same runtime purpose.
Elements	*Tier*, which is a logical grouping of software components. Tiers may be formed on the basis of common computing platforms, in which case those platforms are also elements of the pattern.
Relations	*Is part of*, to group components into tiers. *Communicates with*, to show how tiers and the components they contain interact with each other. *Allocated to*, in the case that tiers map to computing platforms.
Constraints	A software component belongs to exactly one tier.
Weaknesses	Substantial up-front cost and complexity.

Multi-Tier Based System




Multi-Tier Pattern
Layered Pattern
Map-Reduce Pattern
Architectural Patterns
Broker Pattern
Shared-Data Pattern
Model-View-Controller Pattern
Publish-Subscribe Pattern
Client-Server Pattern
Peer-to-Peer Pattern
Pipe-and-Filter Pattern
Service-Oriented Pattern
Component-Connector Patterns

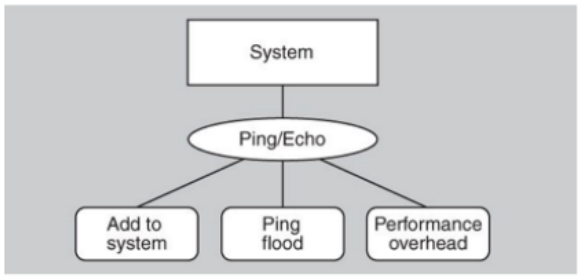

Patterns vs. Tactics

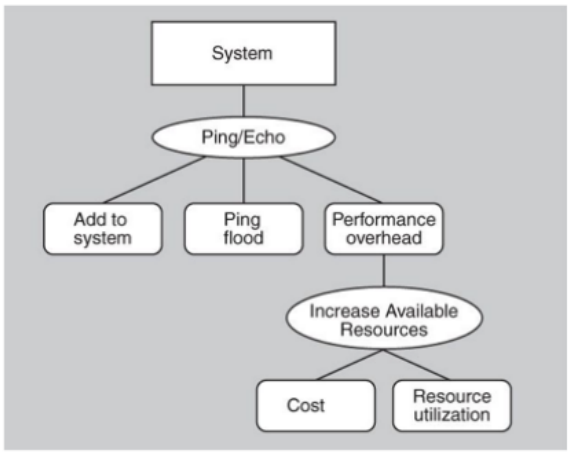
Architecture Patterns and Tactics

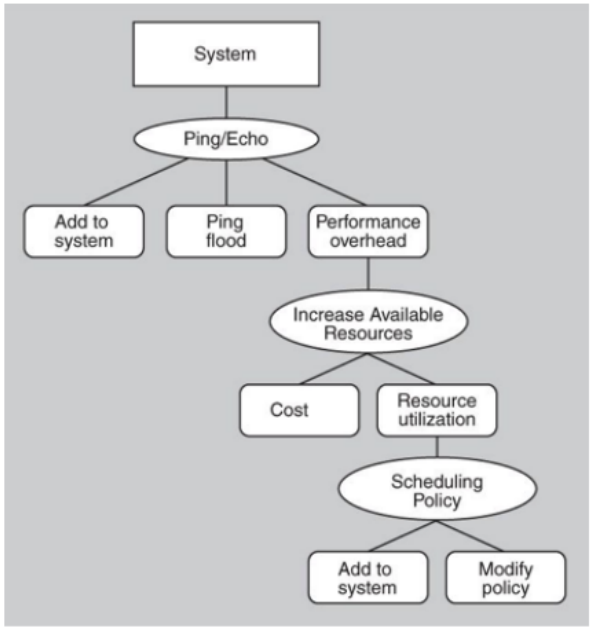
- Tactics are simpler than patterns; they use a single structure or mechanism to address a single architectural force.
- Patterns typically combine multiple design decisions into a package.
- Patterns and tactics together constitute the software architect's primary tools.
- Tactics are "building blocks" of design from which architectural patterns are created.
- Most patterns consist of several different tactics that may:
  - all serve a common purpose,
  - be often chosen to promise different quality attributes.
- Example: layered pattern
  - Increase semantic coherence
  - Restrict dependencies

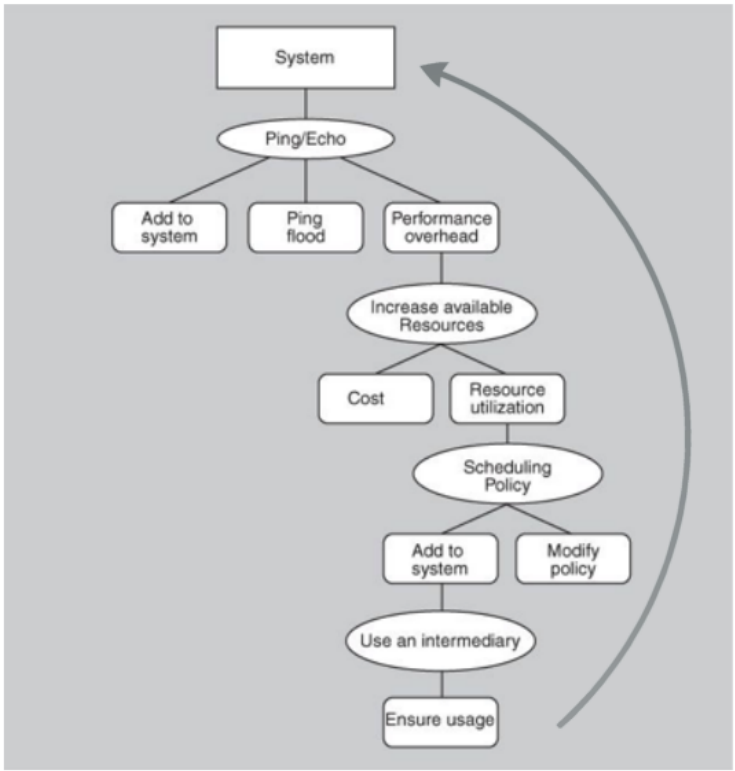
Architecture Patterns and Tactics

	Modifiability										
	Increase Cohesion		Reduce Coupling					Defer Binding Time			
Pattern	Increase Semantic Coherence	Abstract Common Services	Encapsulate	Use a Wrapper	Restrict Comm. Paths	Use an Intermediary	Raise the Abstraction Level	Use Runtime Registration	Use Startup-Time Binding	Use Runtime Binding
Layered	X	X	X		X	X	X			
Pipes and Filters	X		X		X	X			X	
Blackboard	X	X			X	X	X	X		X
Broker	X	X	X		X	X	X	X		
Model View Controller	X		X			X				X
Presentation Abstraction Control	X		X			X	X			
Microkernel	X	X	X		X	X				
Reflection	X		X							


System
Ping/Echo
Add to system
Ping flood
Performance overhead


System
Ping/Echo
Add to system
Ping flood
Performance overhead
Increase Available Resources
Cost
Resource utilization


System
Ping/Echo
Add to system
Ping flood
Performance overhead
Increase Available Resources
Cost
Resource utilization
Scheduling Policy
Add to system
Modify policy


System
Ping/Echo
Add to system
Ping flood
Performance overhead
Increase available Resources
Cost
Resource utilization
Scheduling Policy
Add to system
Modify policy
Use an intermediary
Ensure usage

Reading Materials

- L. Bass, P. Clements and R. Kazman, Software Architecture in Practice, 3rd Edition (2012): Chapter 13, Addison-Wesley.
- F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, Pattern-Oriented Software Architecture (Volume 1): A System of Patterns (1996): Chapter 2, Wiley.
- I. Gordon, Enssential Software Architecture, 1st Edition (2006): Chapter 5; or 2nd Edition (2011): Chapter 7, Springer.