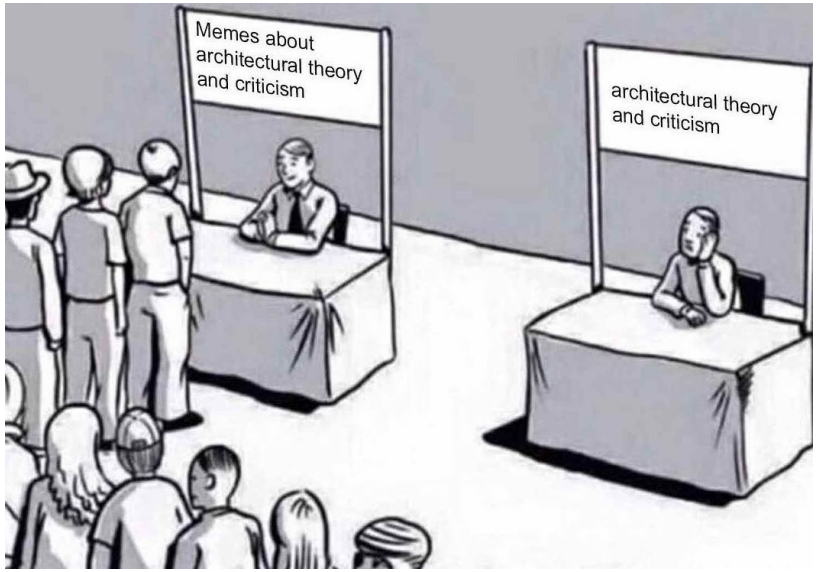
Topic 8

System Architecture and Architectural Patterns

References:
Based on the book Software Architecture in Practice
by Len Bass, Paul Clements, Rick Kazman

Difference between Architecture and Design
(Ref: Tom Graves)



- Arriving at appropriate, useful, maintainable systems requires both architecture and design in appropriate balance. Main difference:
  - **Architecture** faces towards the macro view .. strategy, structure and purpose, what is to be done and where (**languages, frameworks, protocols, platforms, hardware etc.**)
  - **Design** faces towards more detailed implementation and practice, towards the concrete, how systems will actually be implemented at the module/behavioural level (the main focus of this course!)
  - Architecture (**System Architecture**) implies at least process boundaries but more often than not hardware (or VM) **boundaries**

Difference between Architecture and Design (contd.)

- Architecture without design does nothing; since design is required for the actual realisation/implementation of a working system.
- Design without architecture tends toward point-solutions that are optimised solely for a single task and context, often developed in isolation without awareness of deployment realities.
- *Some practitioners do not consider design and architecture to be readily separable. i.e. when you design a class/package/module structure is this not an architecture on some level? (**component** versus **system** architecture)*

What Makes a “Good” Architecture?

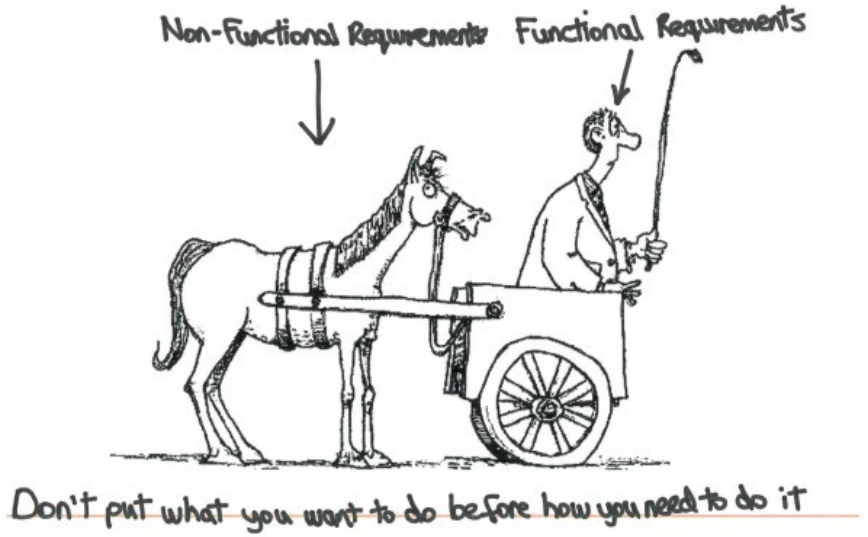
- Is there such a thing as an inherently good or bad architectural choice?
  - e.g. simple script based web system versus multi-tier managed web application
  - better to consider whether an architecture is fit for purpose
- Architectures should be evaluated in the context of specific stated goals and the ability to meet the system’s stated quality requirements.
  - This should occur early in the life cycle and repeated as appropriate (e.g as requirements change, business scales etc.).

System Requirements

- System requirements can be categorized as:
  - **Functional requirements**. These requirements state what the system must do, how it must behave or react to run-time stimuli.
  - **Non functional requirements**. These requirements specify qualify requirements. e.g. performance, reliability, usability, *maintainability*, security etc.
  - **Constraints**. A constraint is a design decision with zero degrees of freedom. That is, it's a design decision that has already been made for you.
    - e.g. the system must adhere to specific government regulations (safety, accessibility etc.)
    - existing skills of required team, technologies already licensed etc.

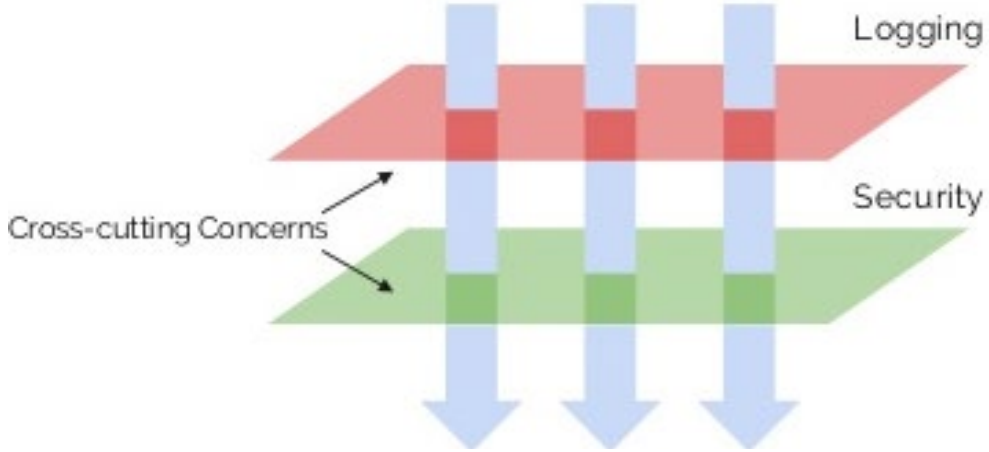
Functionality and System Architecture

- Functional requirements have a limited relationship to architecture:
  - functionality does not determine architecture; given a set of required functionality, there are many architectures you could create to satisfy that functionality
  - therefore functionality and non-functional quality attributes are usually orthogonal (independent)
  - non functional requirements more explicitly influence architecture (they are cross-cutting)



Quality Attribute Considerations

- If a functional requirement is "when the user presses the green button the Options dialog appears”:
  - a performance QA annotation might describe how quickly the dialog will appear;
  - an availability QA annotation might describe how often this function will fail, and how quickly it will be repaired;
  - a usability QA annotation might describe how easy it is to learn and use this function (over time).



Specifying Quality Attribute Requirements

A common form can be used to specify all quality attribute requirements as scenarios.

Source of stimulus. This is some entity (a human, a computer system, or any other actor) that generated the stimulus.

Stimulus. The stimulus is a condition that requires a response when it arrives at a system.

Environment/Context. The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state.

Artifact. Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it.

Response. The response is the activity undertaken as the result of the arrival of the stimulus.

Response measure. When the response occurs, it should be measurable in some fashion so that the requirement can be tested. e.g. MTBF, ops/s etc.

Achieving Quality Attributes Through Tactics

- **Tactics** are a collection of primitive design techniques that an architect can use to achieve a quality attribute response (guidelines).
- Tactics, like design patterns, are techniques that architects have used in practice.
- Rather than being invented, tactics generally capture what architects already do.

Tactics Versus Patterns

- **Architectural Patterns** solve a specific problem but are neutral or have weaknesses with respect to other qualities.
- Consider the *broker* pattern
  - May have performance bottlenecks
  - May have a single point of failure
- Using tactics such as
  - Increase resources will help performance
  - Maintain multiple copies will help availability
- Like patterns, a tactic may have pluses (its reason for being) and minuses i.e. negative side effects.
- Use of tactics can help alleviate the minuses.
- But nothing is free…

Tactics Versus Patterns (contd.)

- In this lecture we **first** consider **tactics** for some quality attributes
  - Performance
  - Availability
  - Modifiability
  - Security
  - Usability
- We **then** look at some specific common architectural patterns

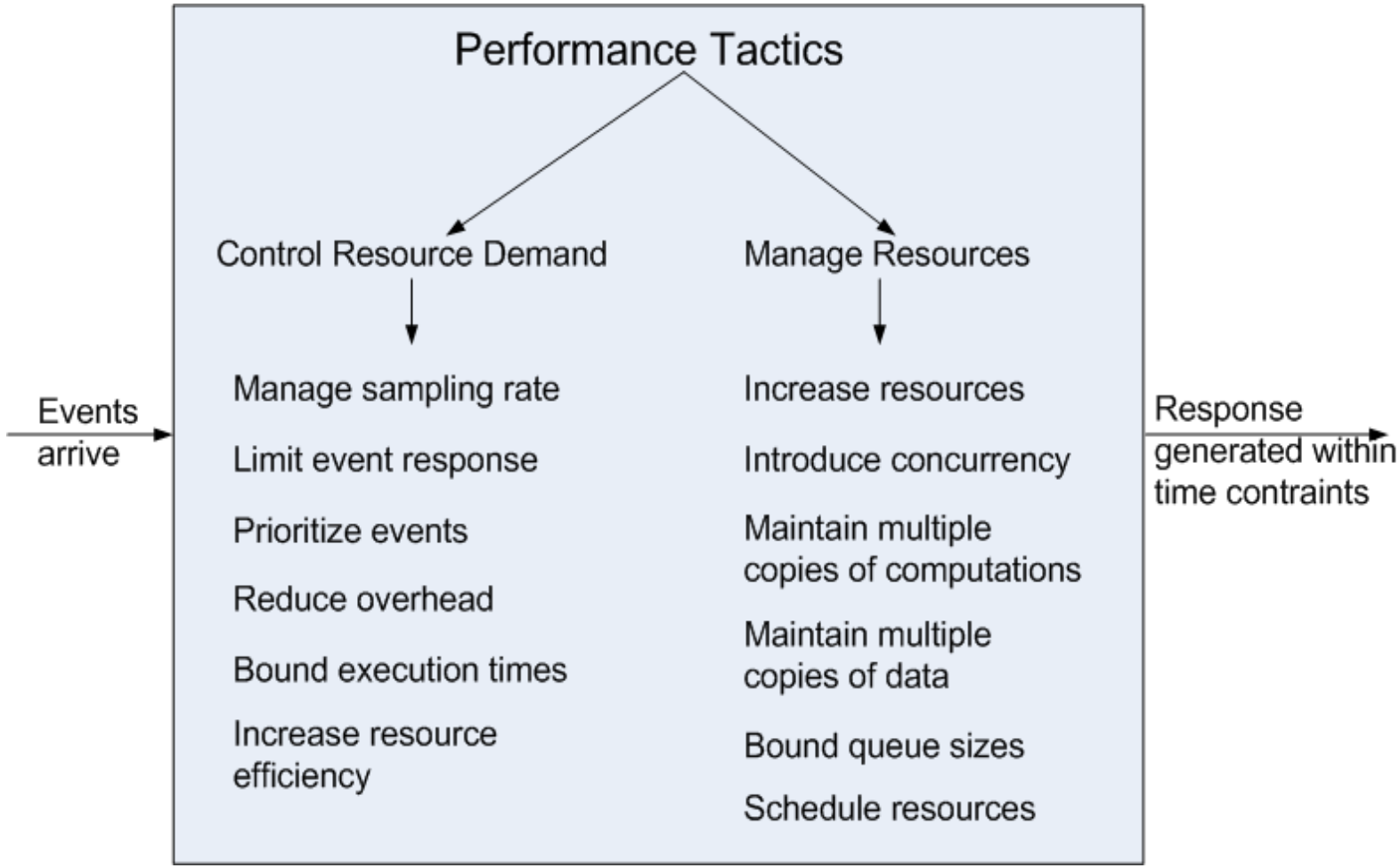
Performance

- Performance is about time and the software system's ability to meet timing requirements.
- When events occur – interrupts, messages, requests from users or other systems, or clock events marking the passage of time – the system, or some element of the system, must respond to them in time.
  - e.g. consider **a real-time versus a non-realtime** system
  - realtime systems have hard **functional** deadlines e.g. automotive, aviation, process control, medical etc.
- Characterizing the events that can occur (and when they can occur) and the system or element's time-based response to those events is the essence is discussing performance.

Performance General Scenario

Portion of Scenario	Possible Values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system.
Environment	Operational mode: normal, emergency, peak load, overload.
Response	Process events, change level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

Performance Tactics



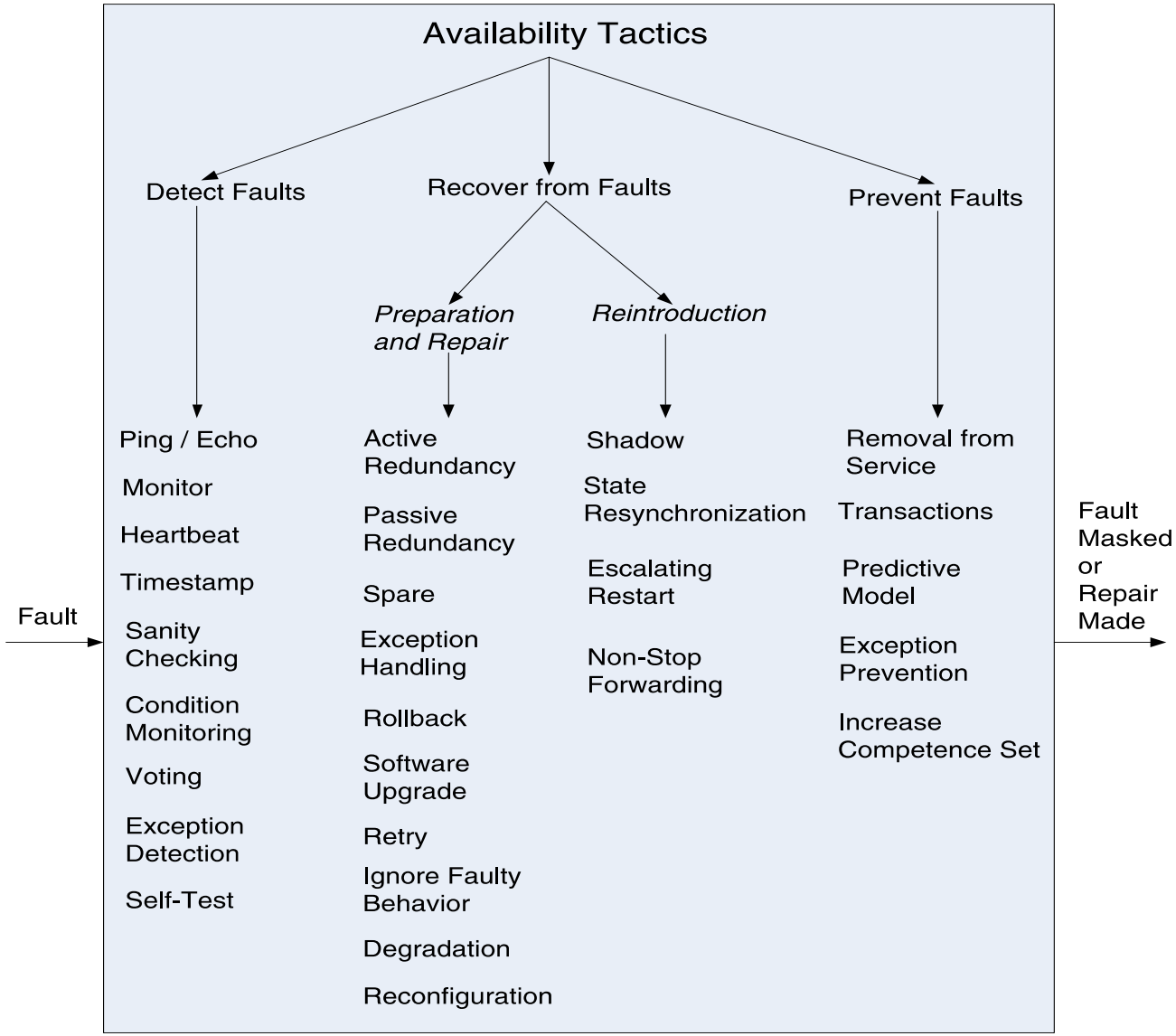
Availability

- Availability refers to a property of software that it **ready to carry out its task** when you need it to be.
- This is a broad perspective and encompasses what is normally called reliability.
- Availability builds on reliability by adding the notion of recovery (repair).
- Fundamentally, availability is about minimizing service outage time by detecting and mitigating faults.

Availability General Scenario

Portion of Scenario	Possible Values
Source	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact	System's processors, communication channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	Prevent the fault from becoming a failure Detect the fault: • log the fault • notify appropriate entities (people or systems) Recover from the fault • disable source of events causing the fault • be temporarily unavailable while repair is being effected • fix or mask the fault/failure or contain the damage it causes • operate in a degraded mode while repair is being effected
Response Measure	Time or time interval when the system must be available Availability percentage (e.g. 99.999%) Time to detect the fault Time to repair the fault Time or time interval in which system can be in degraded mode Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

Availability Tactics



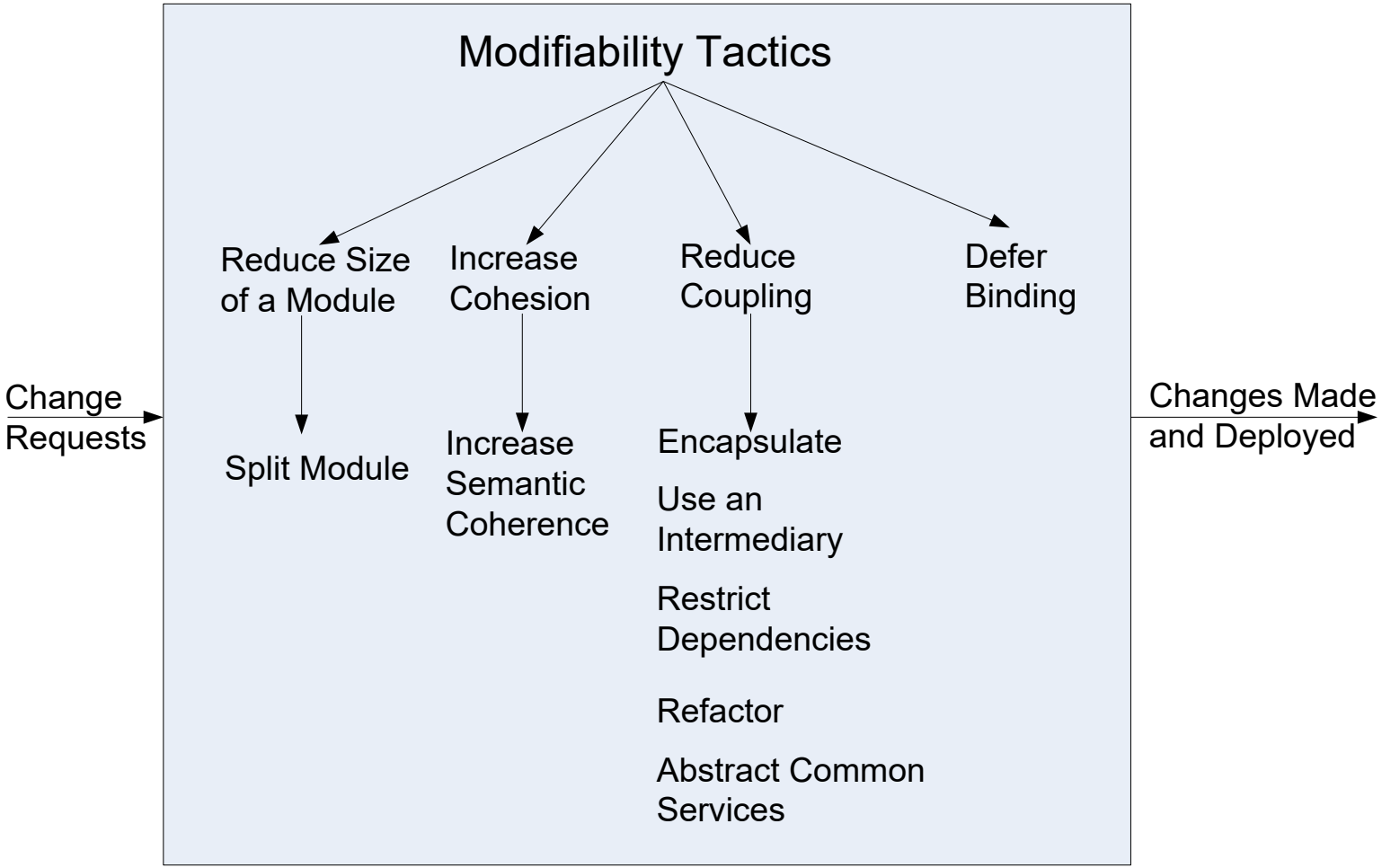
Modifiability

- Modifiability is about change and our interest in it is in the cost and risk of making changes
  - *is a big emphasis in this course in our focus on design!*
- To plan for modifiability, an architect has to consider three questions:
  - What can change?
  - What is the likelihood of the change?
  - When is the change made and who makes it?

Modifiability General Scenario

Portion of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifacts	Code, data, interfaces, components, resources, configurations, …
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: • make modification • test modification • deploy modification
Response Measure	Cost in terms of: • number, size, complexity of affected artifacts • effort • calendar time • money (direct outlay or opportunity cost) • extent to which this modification affects other functions or quality attributes • new defects introduced

Modifiability Tactics



Security

- Measures the system's ability to protect data from unauthorized access while still providing authorized access to people and systems.
- An action taken against a computer system with the intention of doing harm is called an *attack* and can take a number of forms.
- It may be an unauthorized attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.
- Security has three main characteristics, called **CIA**:
  - **Confidentiality** is the property that data or services are protected from unauthorized access. For example, a hacker cannot access your income tax returns on a government computer.
  - **Integrity** is the property that data or services are not subject to unauthorized manipulation. For example, your grade has not been changed since your instructor assigned it.
  - **Availability** is the property that the system will be available for legitimate use. For example, a denial-of-service attack won't prevent you from ordering a product from an online store.

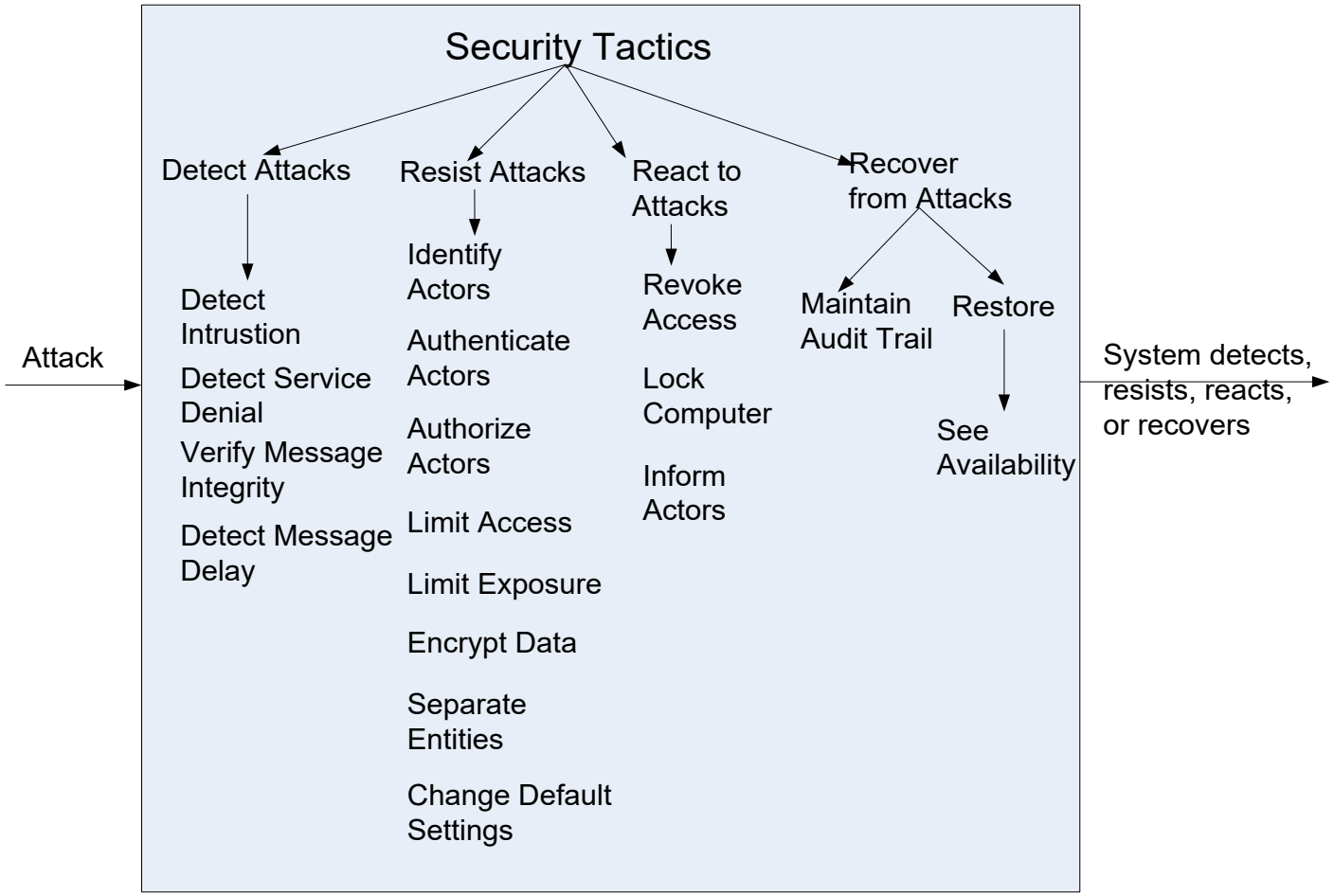
Security (contd.)

- Other characteristics that support **CIA** are
  - **Authentication** verifies the identities of the parties to a transaction and checks if they are truly who they claim to be. For example, when you get an e-mail purporting to come from a bank, authentication guarantees that it actually comes from the bank
    - passwords, digital certificates, two factor authentication, private/public keys etc.
  - **Authorization** grants a user the privileges to perform a task. For example, an online banking system authorizes a legitimate user to access his account.
  - **Nonrepudiation** guarantees that the sender of a message cannot later deny having sent the message and that the recipient cannot deny having received the message. For example, you cannot deny ordering something from the Internet, or the merchant cannot disclaim getting your order.

Security General Scenario

Portion of Scenario	Possible Values
Source	Human or another system which may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
Stimulus	Unauthorized attempt is made to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
Artifact	System services; data within the system; a component or resources of the system; data produced or consumed by the system
Environment	The system is either online or offline, connected to or disconnected from a network, behind a firewall or open to a network, fully operational, partially operational, or not operational
Response	Transactions are carried out in a fashion such that • data or services are protected from unauthorized access; • data or services are not being manipulated without authorization; • parties to a transaction are identified with assurance; • the parties to the transaction cannot repudiate their involvements; • the data, resources, and system services will be available for legitimate use. The system tracks activities within it by • recording access or modification, • recording attempts to access data, resources or services, • notifying appropriate entities (people or systems) when an apparent attack is occurring.
Response Measure	One or more of the following • how much of a system is compromised when a particular component or data value is compromised, • how much time passed before an attack was detected, • how many attacks were resisted, • how long does it take to recover from a successful attack, • how much data is vulnerable to a particular attack

Security Tactics



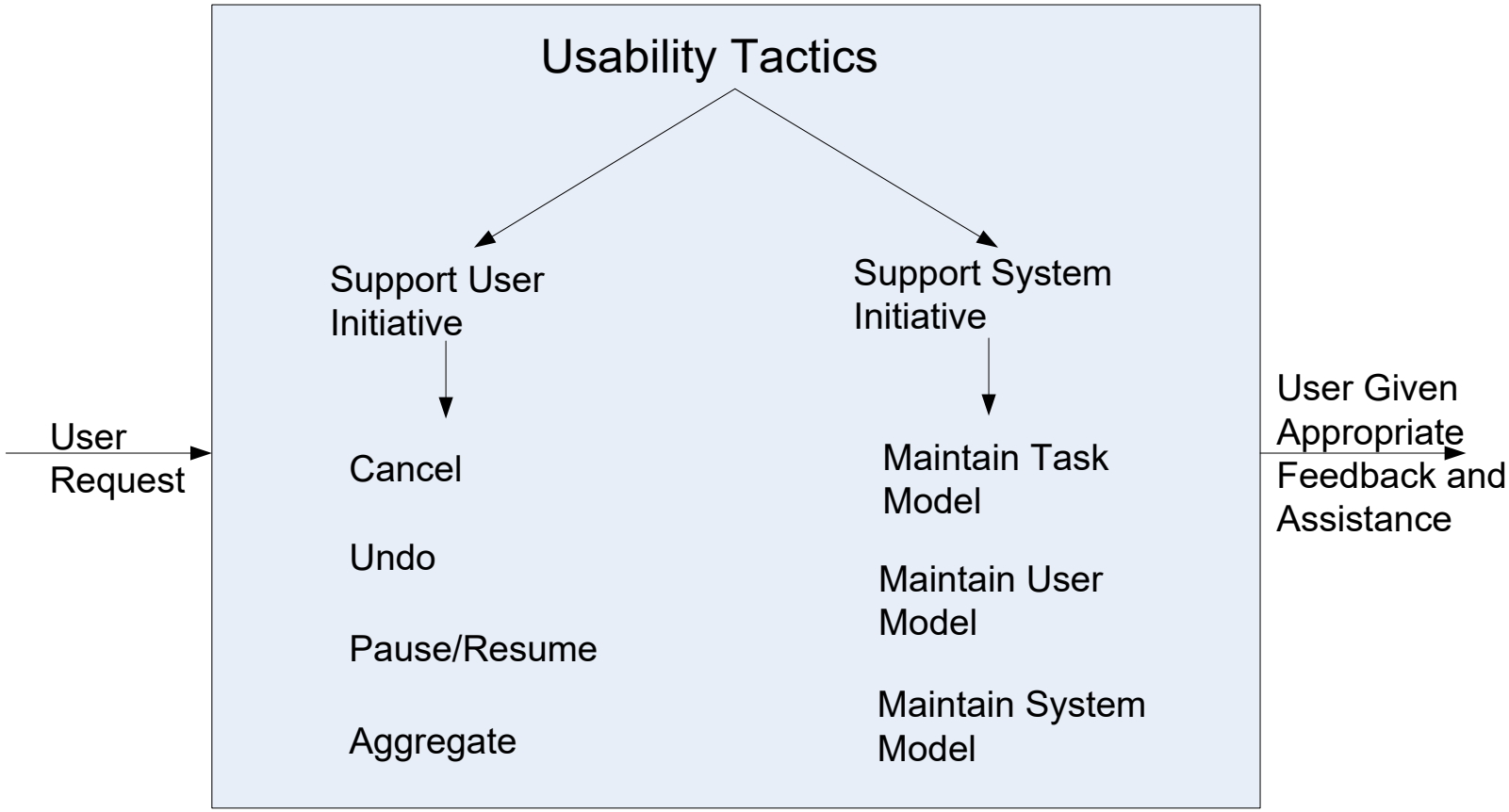
Usability

- Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides.
- Over the years, a focus on usability has shown itself to be one of the cheapest and easiest ways to improve a system's quality (or, more precisely, the user's *perception* of quality).
- Usability comprises the following areas:
  - Learning system features.
  - Using a system efficiently.
  - Minimizing the impact of errors.
  - Adapting the system to user needs.
  - Increasing confidence and satisfaction.

Usability General Scenario

Portion of Scenario	Possible Values
Source	End user, possibly in a specialized role
Stimulus	End user tries to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or configure the system
Environment	Runtime or configuration time
Artifacts	System or the specific portion of the system with which the user is interacting.
Response	The system should either provide the user with the features needed or anticipate the user's needs.
Response Measure	One or more of the following: task time, number of errors, number of tasks accomplished, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time or data lost when an error occurs.

Usability Tactics



Architectural Patterns: Examples

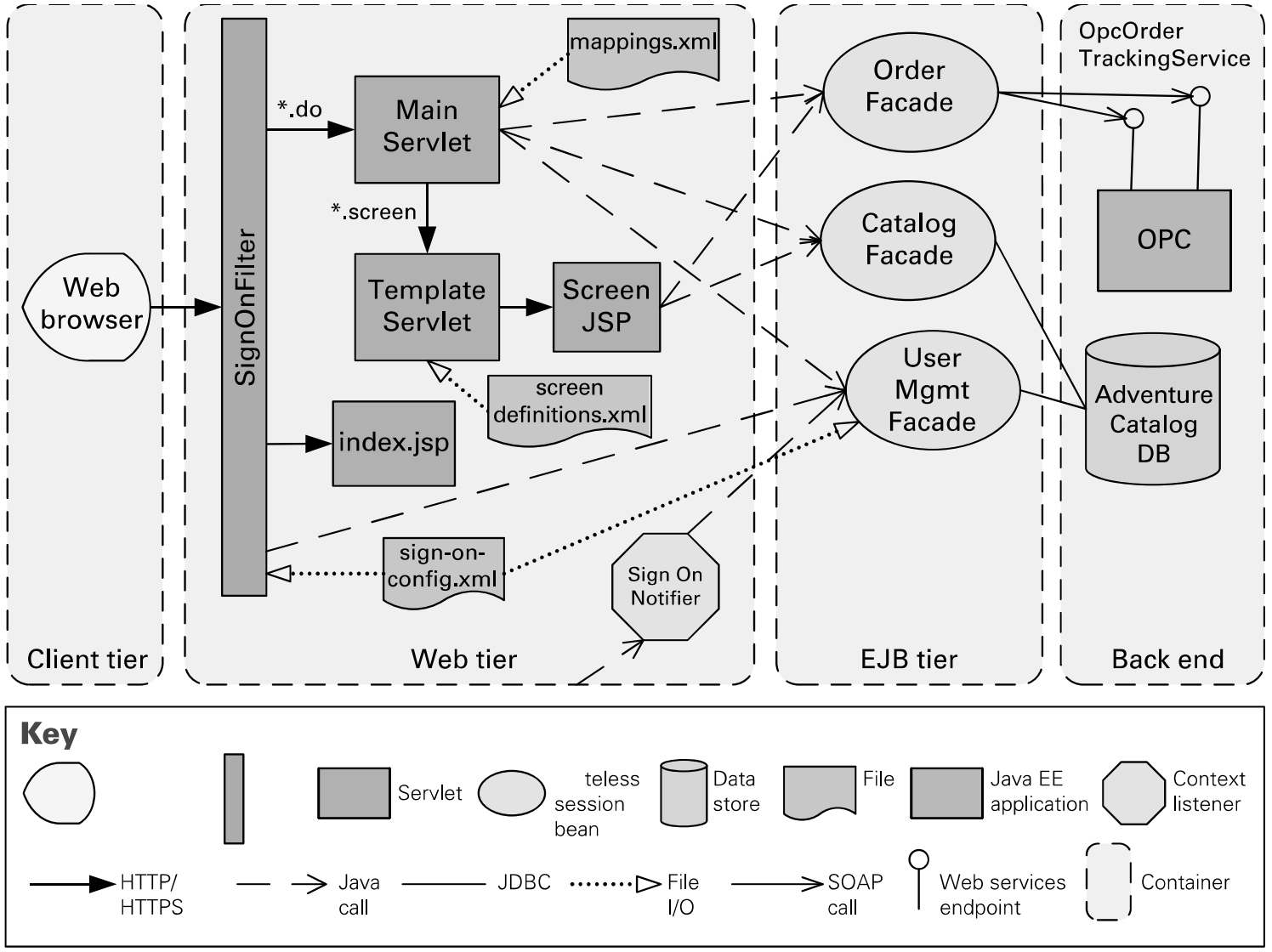
An architectural pattern establishes a relationship between:

- *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 architectural resolution to the problem, appropriately abstracted and is described by:
  - A set of **element types** (data repositories, processes, and objects)
  - A set of **interaction mechanisms** or connectors (for example, method calls, events, or message bus)
  - A **topological layout** of the components
  - A set of **semantic constraints** covering topology, element behavior, and interaction mechanisms

Multi-Tier Pattern

- **Context**: In a distributed deployment, there is often a need to distribute a system's infrastructure into distinct subsets.
- **Problem**: How can we split the system into a number of computationally independent execution structures—groups of software and hardware—connected by some communications media?
- **Solution**: The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a tier.

Multi-Tier Example



Multi-Tier Solution

- Overview: The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a *tier*.
- Elements:
  - *Tier,* which is a logical grouping of software components.
- 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.
- Benefits: Highly scalable independently at each tier
- Weaknesses: Substantial up-front cost and complexity. Requires complex middleware/execution enviroments.

NOTE: Similar to layered pattern but tiers generally imply some sort of managing container and physical distribution

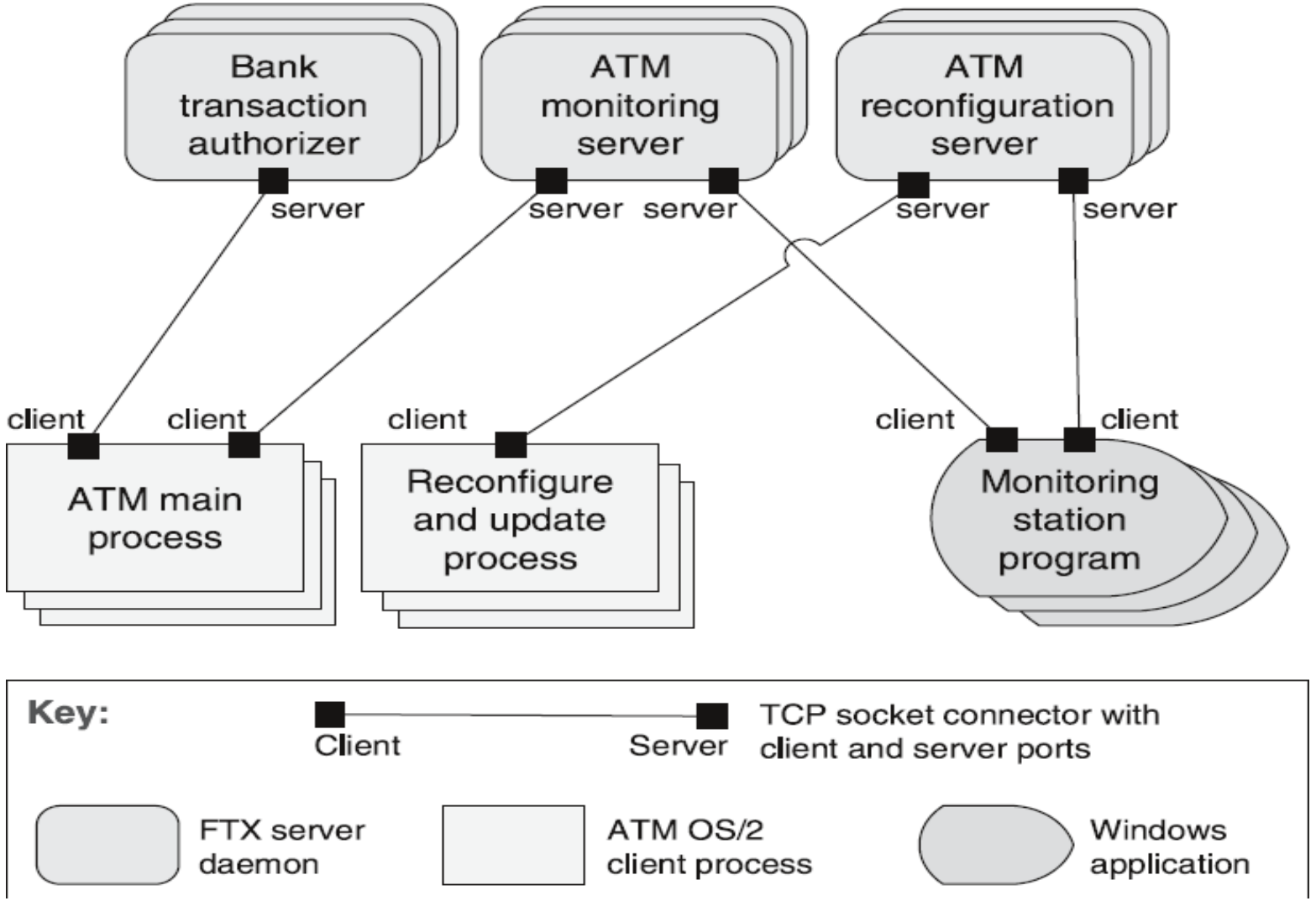
Client-Server Pattern

- **Context:** There are shared resources and services that distributed clients wish to access, and for which we wish to control access or quality of service.

- **Problem:** By managing a set of shared resources and services, we can promote modifiability and reuse, by factoring out common services and maintaning these in a single location, or a small number of locations. We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.

- **Solution:** Clients interact by requesting services of servers, which provide a set of services. Some components may act as both clients and servers. There may be one central server or multiple distributed ones.

Client Server Example



Client Server Solution

- Overview: Clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests.
- **Can be considered a reduced multi-tier with only two tiers!**
- 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.
- *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.
- Java RMI is a client/server protocol

Client Server Solution (contd.)

- 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.
- 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 resources may be wasted/left unused**
  - Network may introduce unacceptable latency

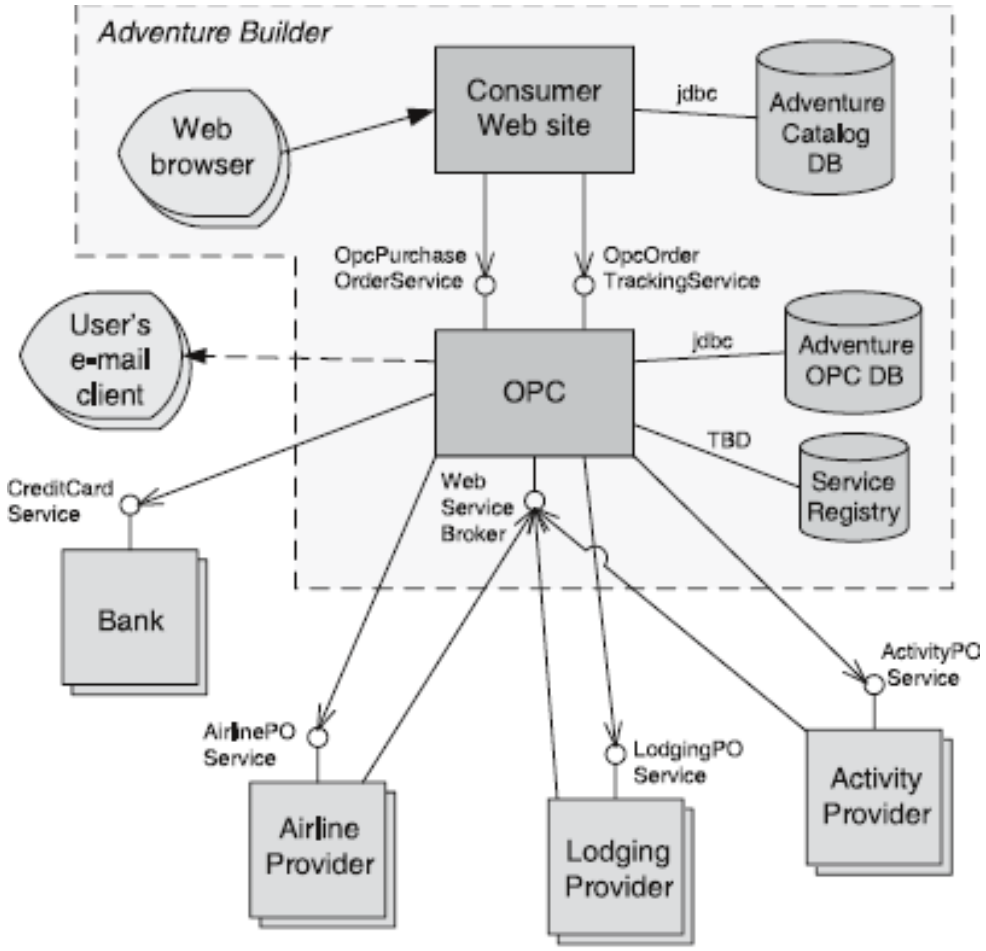
Service Oriented Architecture Pattern

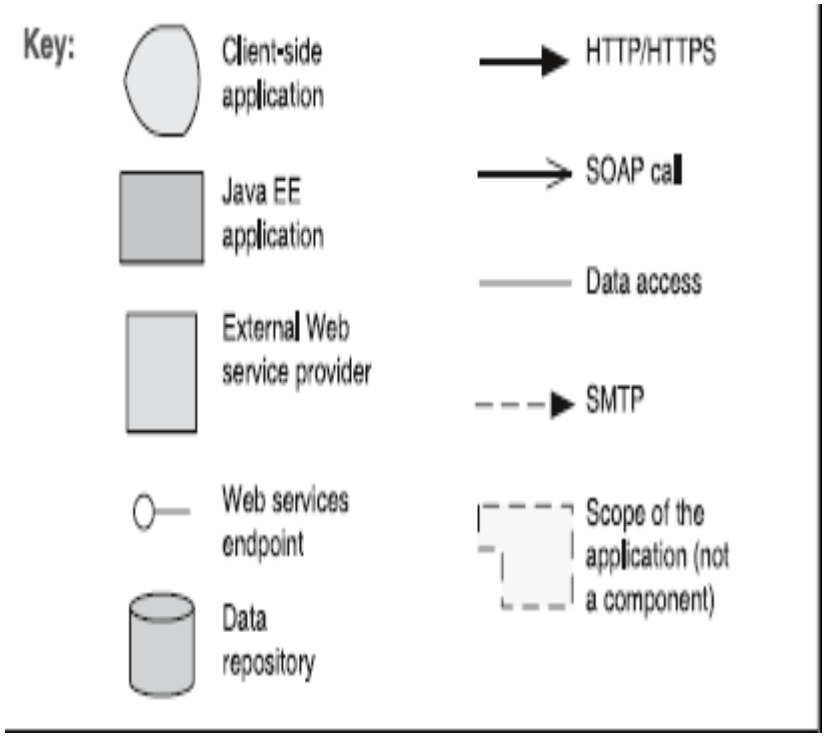
- **Context**: A number of services are offered (and described) by service providers and consumed by service consumers. Service consumers need to be able to understand and use these services without any detailed knowledge of their implementation.

- **Problem**: How can we support interoperability of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet?

- **Solution**: The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services.

Service Oriented Architecture Example





OPC = Order Processing Centre (the orchestration engine)

Service Oriented Architecture Solution

- Overview: Computation is achieved by a set of cooperating components that provide and/or consume services over a network.
- Elements:
  - *Service providers,* which provide one or more services through published interfaces.
  - *Service consumers,* which invoke services directly or through an intermediary.
  - *NOTE: Service providers may also be service consumers.*

Service Oriented Architecture Solution (contd.)

- *Enterprise Service Bus (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 (see **Broker** pattern).
- *Orchestration server/engine,* which coordinates the interactions between service consumers and providers based on languages for business processes and workflows.
  - e.g. using executable language such as Business Process Execution Language (BPEL or WS-BPEL)

Service Oriented Architecture Solution (contd.)

- 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.

Service Oriented Architecture Solution (contd.)

- 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 may not provide performance guarantees.

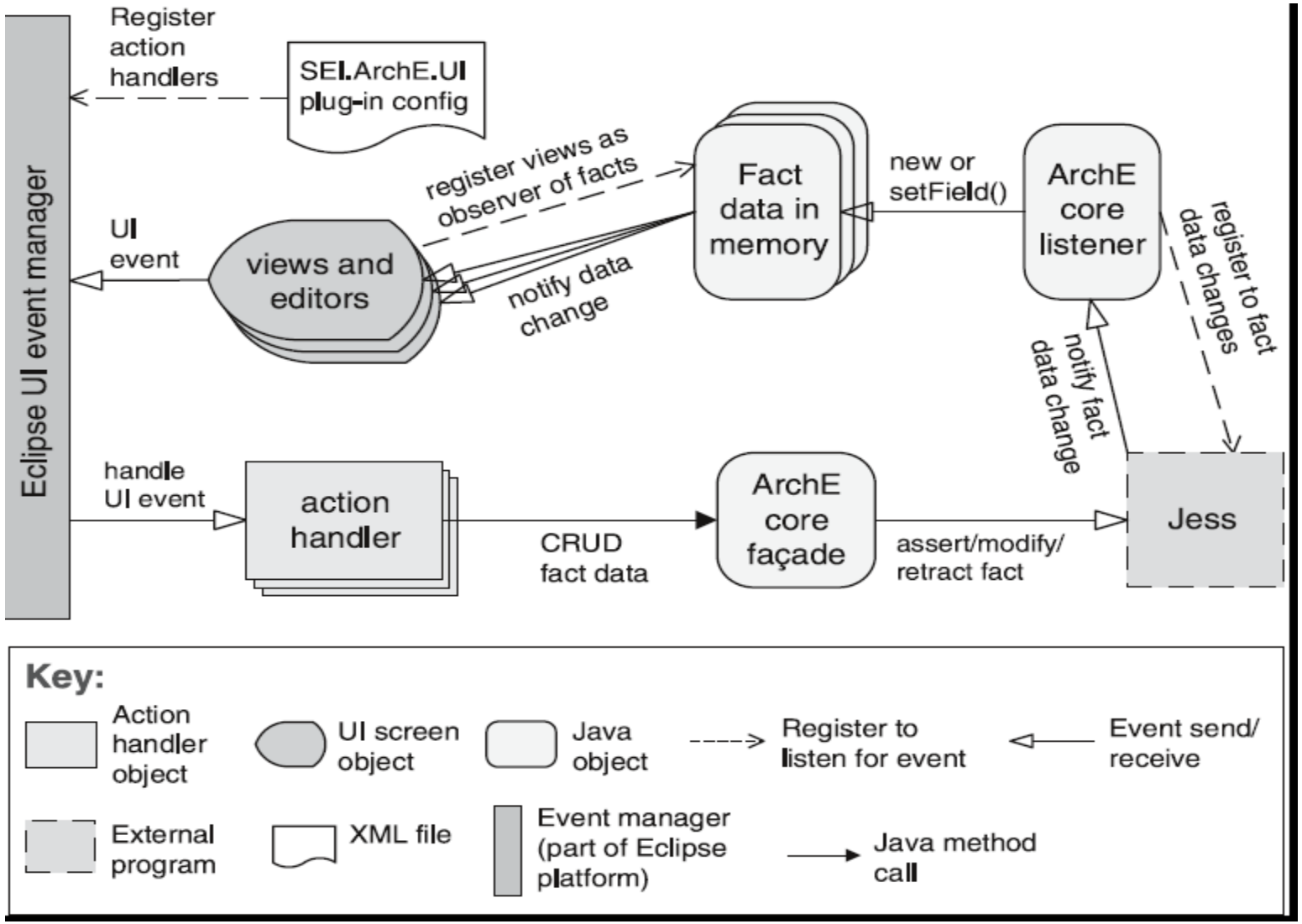
Micro Services Architecture

- A variation on SOA that is becoming increasingly popular
- As the name suggests the granularity of services is smaller
- Generally self-contained and manage their own persistence
- Create service boundaries even when no organizational/structural imperative
- Provides low coupling
- Enables rapid system evolution
- Usually have a simple REST protocol to simplify integration and testing
- NOTE: in practice the services may not be as ‘micro’ as the name suggests!

Publish-Subscribe Pattern

- **Context:** There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers are not predetermined or fixed, nor is the data that they share.
- **Problem:** How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers so they are unaware of each other's identity, or potentially even their existence?
- **Solution:** In the publish-subscribe pattern, components interact via announced messages, or events. Components may subscribe to a set of events. Publisher components place events on the bus by announcing them; the connector then delivers those events to the subscriber components that have registered an interest in those events.

Publish Subscribe Example



Publish-Subscribe 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 component* with at least one publish or subscribe port.
  - *The publish-subscribe connector*, which will have *announce* and *listen for* 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.
- Like SOA is generally based on an Enterprise Service Bus (ESB) to provide reliable messaging

Publish-Subscribe Solution (contd.)

- 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.
- Benefits;
  - guaranteed delivery (via ESB)
  - less resource intensive (since can be scheduled during down time)
- Weaknesses:
  - Typically increases latency and has a negative effect on scalability and predictability of message delivery time.
  - Less control over ordering of messages which must be taken into account in terms of processing rules

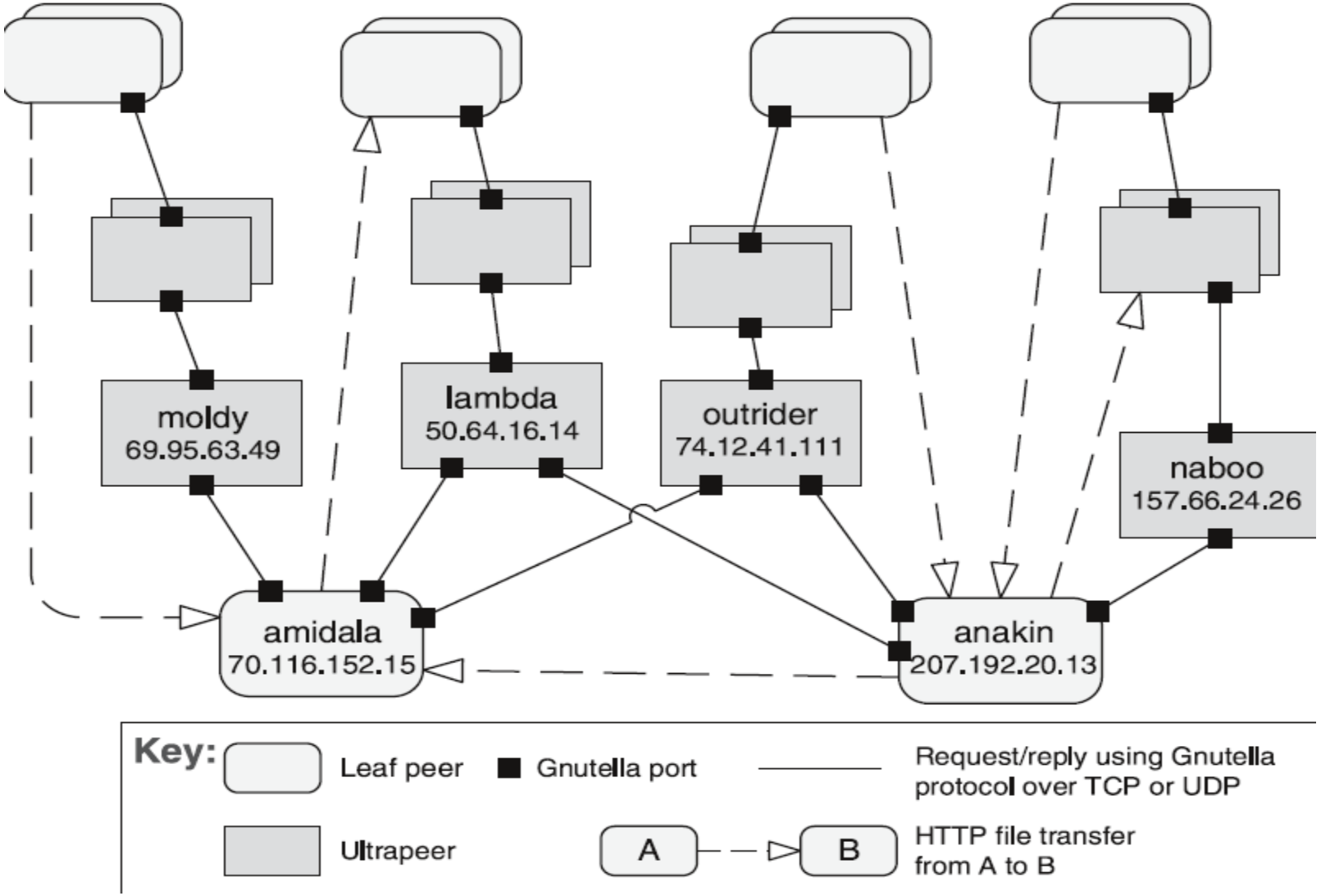
Peer-to-Peer Pattern

- **Context:** Distributed computational entities—each of which is considered equally important in terms of initiating an interaction and each of which provides its own resources—need to cooperate and collaborate to provide a service to a distributed community of users.

- **Problem:** How can a set of “equal” distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?

- **Solution:** In the peer-to-peer (P2P) pattern, components directly interact as peers. All peers are “equal” and no peer or group of peers can be critical for the health of the system. Peer-to-peer communication is typically a request/reply interaction without the asymmetry found in the client-server pattern.

Peer-to-Peer Example



Peer-to-Peer Solution

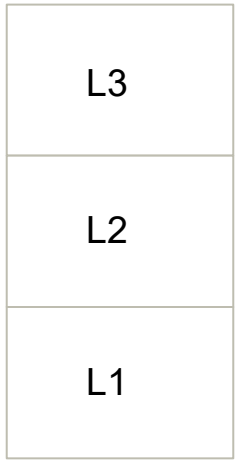
- Overview: Computation is achieved by cooperating peers that request services from and provide services to one another across a network.
- Elements:
  - *Peer,* which is an independent component running on a network node. Special peer components can provide routing, indexing, and peer search capability.
  - *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.

Peer-to-Peer Solution (contd.)

- 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.

Layer Pattern

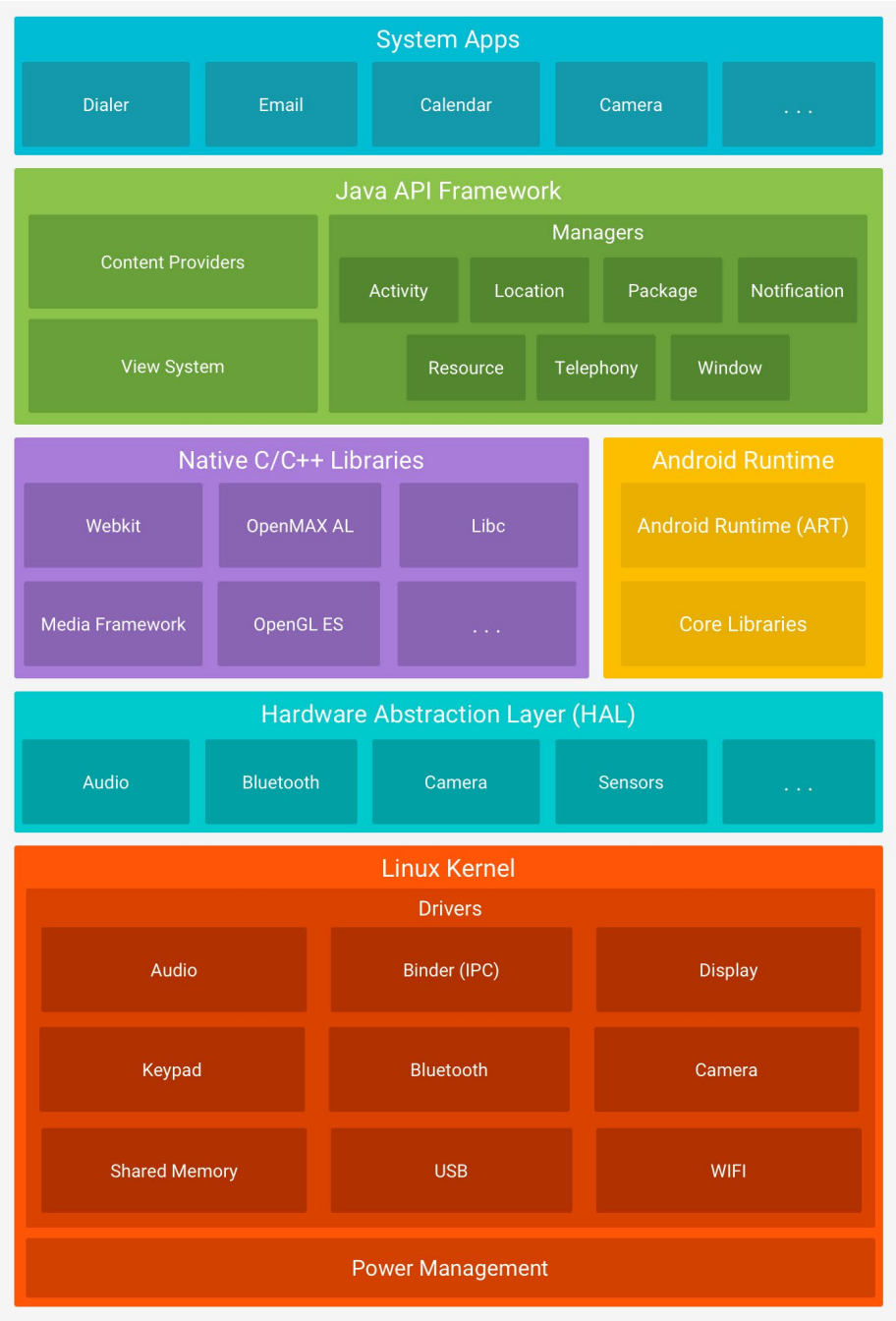
- **Context:** Complex systems experience the need to develop and evolve portions of the system independently. Thus developers need a well-documented separation of concerns, allowing modules to be independently developed and maintained.
- **Problem:** Segment software into modules that can be developed and evolved separately with little interaction, supporting portability, modifiability, and reuse.
- **Solution:** To this end, the layered pattern divides the software into layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional through a public interface and is **usually hierarchical**.
- **Related to Facade Design Pattern!**



Layer 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 (usually hierarchical).
- Elements: *Layer*, a kind of module.
- Relations: *Allowed to use.* The design should define what the layer usage rules are and any allowable exceptions.
- Constraints:
  - Every piece of software is allocated to exactly one layer.
  - At least two layers (but usually there are three or more).
  - The no circular *allowed-to-use* relations.
- Weaknesses:
  - The addition of layers adds up-front cost and complexity.
  - Layers contribute a performance penalty.

Is a good solution for adding to systems with a long lifetime or wide usage scenarios (e.g. updating a legacy system, cross platform development)



Example 1: Android Architecture

FROM: https://developer.android.com/guide/platform

Example 2: Open Systems Interconnection (OSI) Model

OSI model				
Layer			Protocol data unit (PDU)	Function[19]
Host layers	7	Application	Data	High-level APIs, including resource sharing, remote file access
	6	Presentation		Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption
	5	Session		Managing communication sessions, i.e., continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes
	4	Transport	Segment, Datagram	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing
Media layers	3	Network	Packet	Structuring and managing a multi-node network, including addressing, routing and traffic control
	2	Data link	Frame	Reliable transmission of data frames between two nodes connected by a physical layer
	1	Physical	Symbol	Transmission and reception of raw bit streams over a physical medium

FROM: https://en.wikipedia.org/wiki/OSI_model

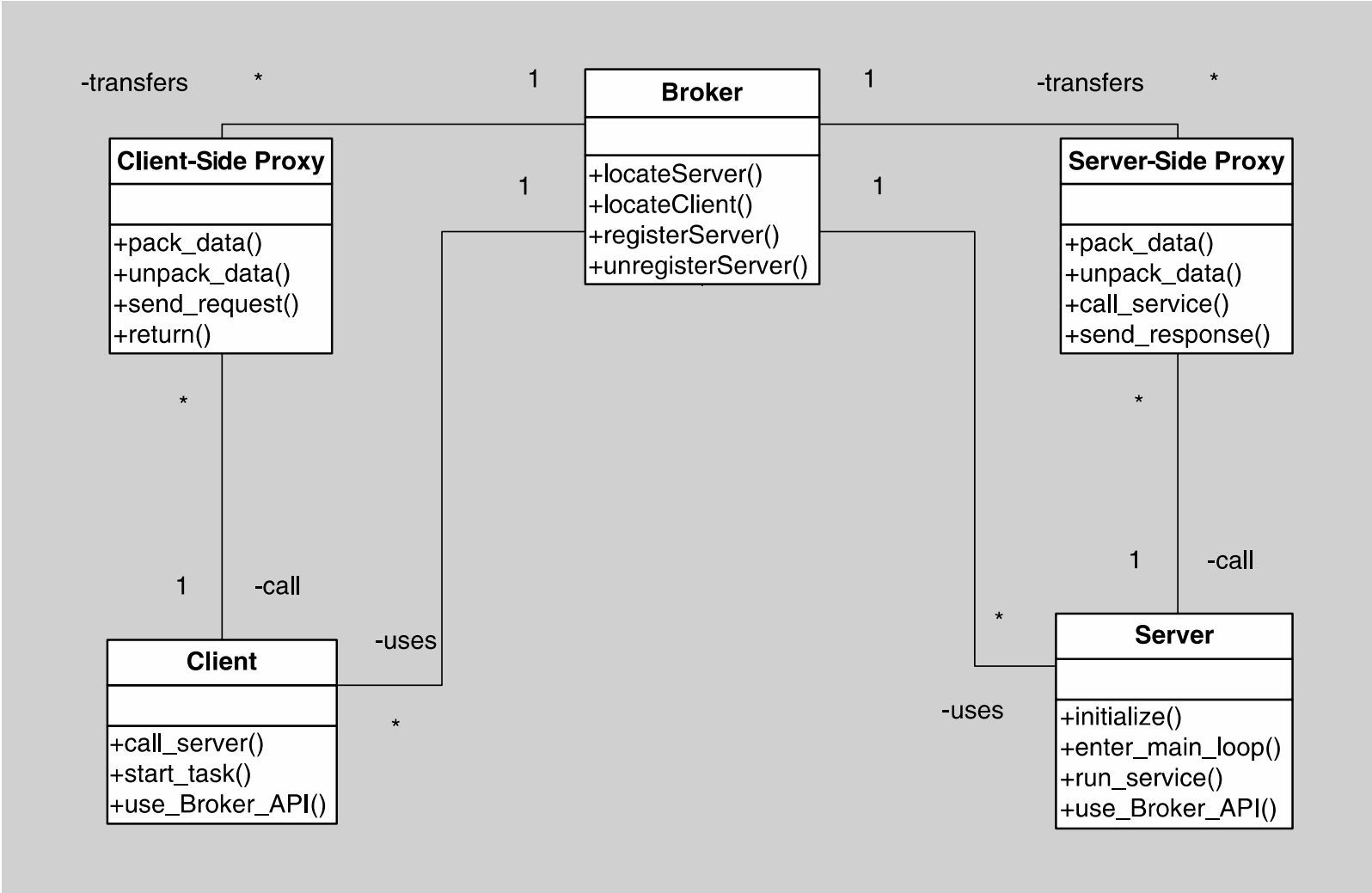
Broker Pattern

- **Context**: Many systems are constructed from a collection of services/components distributed across multiple servers. Combining these is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.
- **Problem**: Structuring distributed software so that users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between them?
- **Solution**: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a **broker**. When a client needs a service, it queries a broker via a service interface, which then forwards the client's service request to a server processing the request.

Broker Pattern (contd.)

- Overview: The broker pattern defines a runtime component (server), called a broker, that mediates the communication between clients and servers.
- Elements:
  - *Client,* a requester of services
  - *Server,* a provider of services
  - *Broker,* an intermediary that locates an appropriate server to meet client's request, forwards the request, and returns the results
  - *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

Broker Pattern Solution



Examples

- Java RMI uses a simple form of this pattern
  - The *rmiregistry* is the broker
  - RMI objects are registered with the registry
  - Clients lookup and use the registered RMI Objects
- Service-Oriented Architecture uses a more complex version of the broker pattern
  - SOAP Web services provide a service description in the form of WSDL (Web Services Discovery Language)
  - The WSDL is registered with the broker
  - Clients look up these services via a service broker and retrieve the WSDL in order to use the services

Broker Solution (contd.)

- Relations: The *attachment* relation associates clients (or proxies) and servers (or 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).
- Advantages: Decouples component/service discovery
- 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 make deployment and testing more difficult.

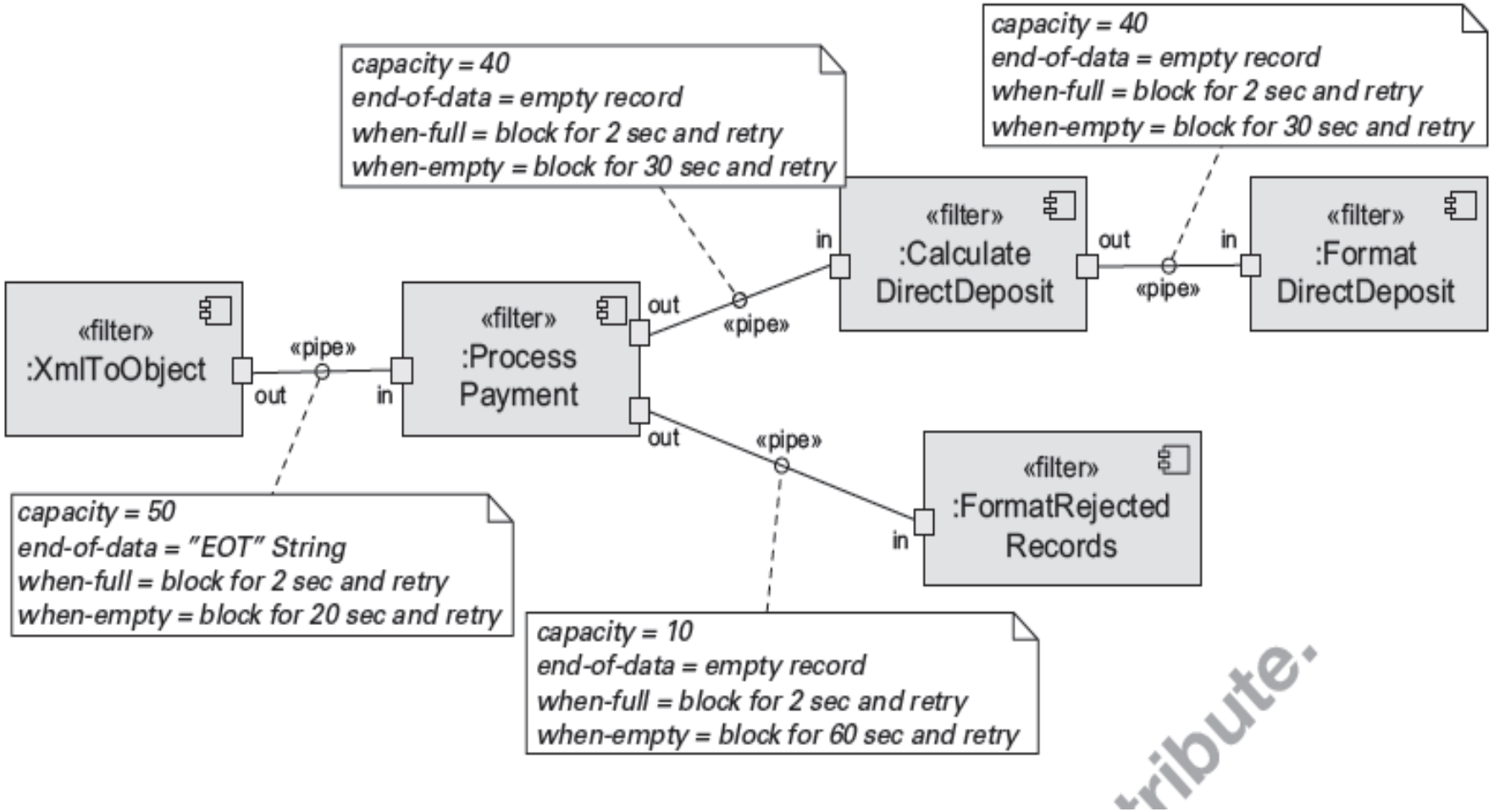
Pipe and Filter Pattern

- **Context:** Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

- **Problem:** Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

- **Solution:** The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

Pipe and Filter Example



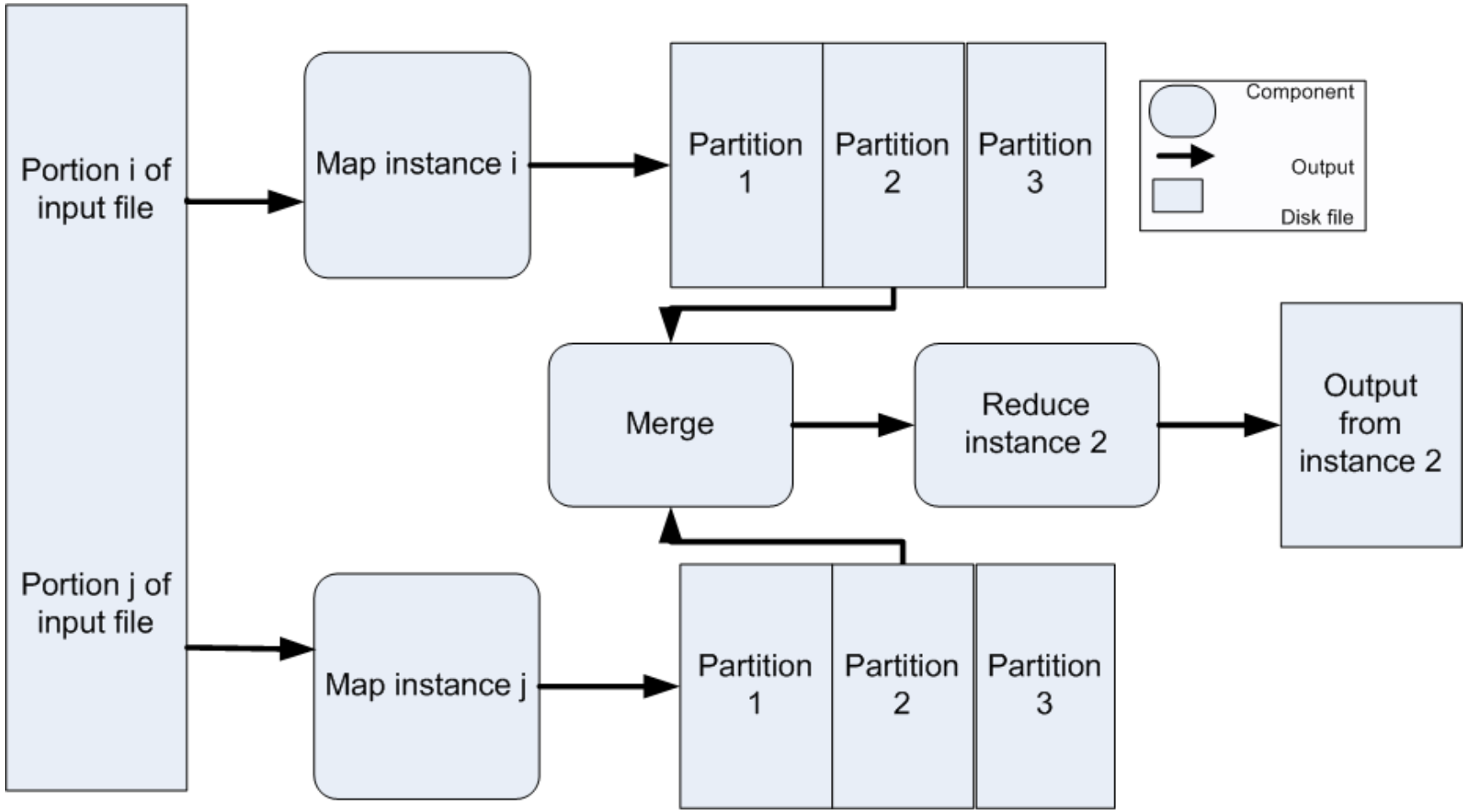
Pipe and Filter Solution

- 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).
  - *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.
- 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.
- Useful for realtime "Big Data" applications
- See `java.util.stream.Stream` API

Map-Reduce Pattern

- **Context:** Businesses have a pressing need to quickly analyze enormous volumes of data they generate or access, at petabyte scale.
- **Problem:** For many applications with ultra-large data sets, sorting the data and then analyzing the grouped data is sufficient. The problem the map-reduce pattern solves is to efficiently perform a distributed and parallel sort of a large data set and provide a simple means for the programmer to specify the analysis to be done.
- **Solution:** The map-reduce pattern requires three parts:
  - A specialized infrastructure takes care of allocating processes to the hardware nodes in a massively parallel computing environment.
  - A programmer specified component called the **map** which sorts/filters/groups the data into subsets for processing.
  - A programmer specified component called **reduce** which aggregates the independent results produced by the map stage

Map-Reduce Example



Map-Reduce 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.
- 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 result calculation portion on the mapped processes.
  - The infrastructure is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure.

Map-Reduce Solution (contd.)

- 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 data.
  - Map functions are stateless and do not communicate with each other.
  - The only communication between map 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.
- See `java.util.stream.Stream` API

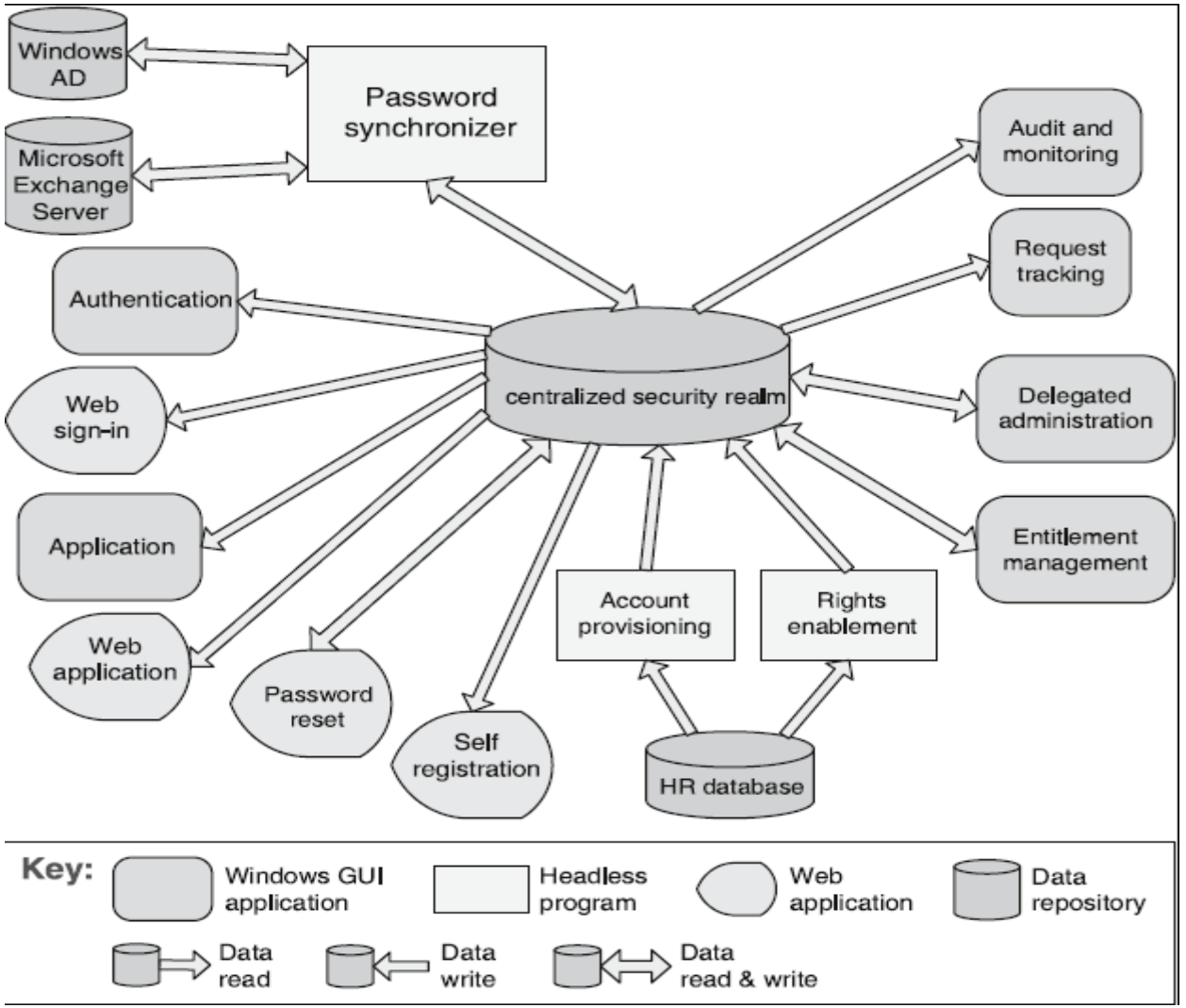
Shared-Data Pattern

- **Context:** Various computational components need to share and manipulate large amounts of data. This data does not belong solely to any one of those components.
- **Problem:** How can systems store and manipulate persistent data that is accessed by multiple independent components?
- **Solution:** In the shared-data pattern, interaction is dominated by the exchange of persistent data between multiple *data accessors* and at least one *shared-data store*. Exchange may be initiated by the accessors or the data store. The connector type is *data reading and writing*.

Shared Data 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, type of access permitted etc.
  - *Data accessor component*.
  - *Data reading and writing connector*.
- **Example**: Single sign-on (shared authentication datastore) .. see next slide (e.g. Radius Server https://freeradius.org/)

Shared Data Example



Shared Data Solution (contd.)

- Relations: *Attachment* relation determines which data accessors are connected to which data stores.
- Constraints: Data accessors interact only with the data store(s) (no client to client communication).
- 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.
- **NOTE: Is a specific type of client-server architecture**
- May also use **publish/subscribe** mechanism to facilitate data updates/sharing

Summary: Why is Architecture important?

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that form the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training a new team member.
14. Tactics support architecture