Chapter 13: Architecture Patterns

SAiP Chapter 13

Topics

- What is a Pattern?
- Pattern Catalog
  - Module patterns
  - Component and Connector Patterns
  - Allocation Patterns

Architectural Styles (Patterns)

Patterns – a Review

- Work on software patterns stemmed from work on patterns for building architecture carried out by Christopher Alexander (*A Pattern Language: Towns, Buildings, Construction* (1977))
- All well-structured software systems are full of patterns
  - **Architectural patterns** – system level structural organization
  - **Design patterns** – component level design
  - **Programming idioms** – reoccurring constructs expressed in different languages (programming tasks, algorithms, data structures; e.g., increment counter)

What is a Software Architectural Pattern?

- A pattern is a **solution** to a **problem** in a **context**
  - **Context**. A recurring, common situation in the world that gives rise to a problem.
  - **Problem**. The problem, appropriately generalized, that arises in the given context.
  - A **solution**. A successful architectural resolution to the problem, appropriately abstracted
- An **architectural pattern** expresses a fundamental **structural organization abstraction** for software systems
  - A set of structural elements
  - Their relationships
  - Rules and guidelines for organizing the relationships between them

Software Architecture Patterns

- **Versus software design patterns –** higher level **system wide in scope** ; some overlap
  - Recall the distinction between architecture and design work
- Most software systems **cannot be structured** according to a **single architectural pattern**
  - Example: Design a system for flexibility of component distribution in a heterogeneous computer network and for adaptability of their user interfaces
- How do you think about software design? Essentially the same cognitive process for architecture design …
  - Requirements driven, separation of concerns, top-down, apply patterns

Describing a Pattern Solution

- A pattern solution is determined and described by:
  - A set of **element types** (for example, 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**
  - I.e., **views**
- Organize patterns by **predominant structure** – module, connector, allocation

A Pattern Catalog

Module	Component & Connector	Allocation
Layered*	Broker*	Map-Reduce*
Domain decomposition	Model-View-Controller*	Multi-tier functional mapping*
	Pipe-and-Filter*	Platform
	Client-Server*	Team work allocation
	Peer-to-Peer*	
	Service-Oriented Architecture*	
	Microservices*	
	Publish-Subscribe (Observer)*	
	Shared-Data*	
	Black Board*	
	Event Driven*	

* Detailed below

Class Activity

- Each team will prepare a short ~five minute presentation of an architectural pattern
  - Context, problem, solution, constraints/weaknesses
  - Example view (not from the text)
  - Candidate for your project?

Team 1 – Broker

Team 2 – Event Driven

Team 3 – Publish-Subscribe

Team 4 – Pipe-and-Filter

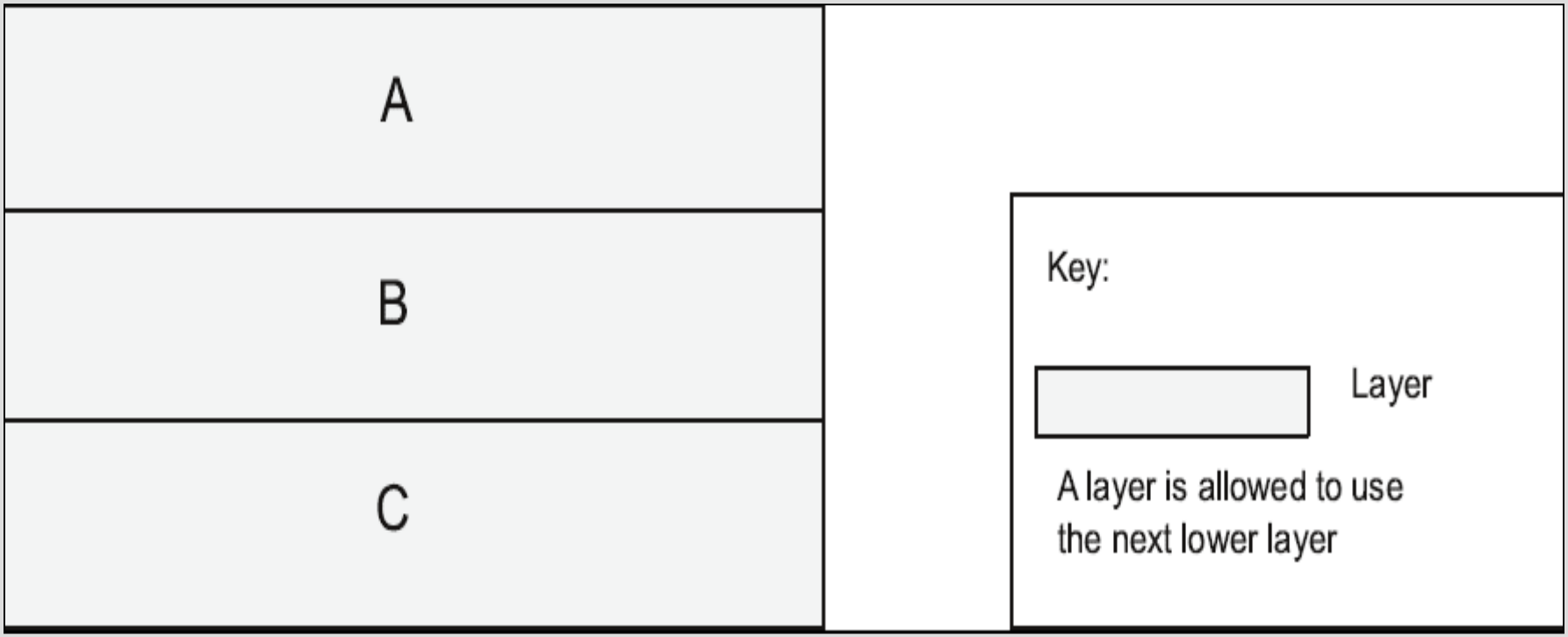
Team 5 – Map-Reduce

Team 6 – Microservices

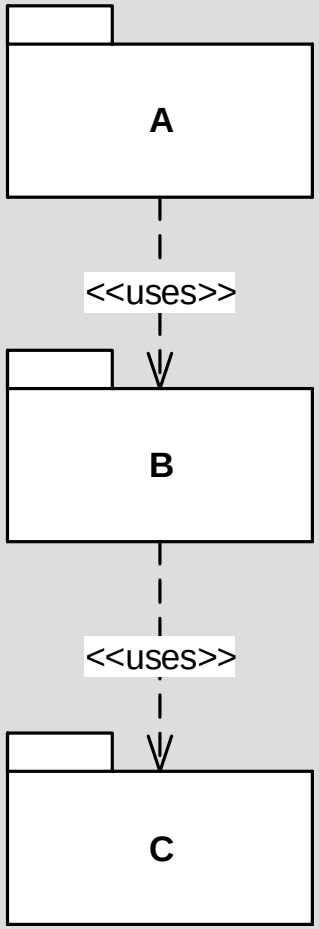
Layer Pattern

- **Context:** Complex systems need to **develop** and **evolve** portions of the system **independently**. Developers need well-documented **separation of concerns**, so system modules may be independently developed and maintained.
- **Problem:** The software needs to be segmented in such a way that the modules can be **developed and evolved separately** with **little interaction** among the parts, supporting **portability, modifiability, and reuse.**
- **Solution:** To achieve this separation of concerns, the layered pattern **divides the software into units called layers**. **Each layer** is a grouping of modules that offers a **cohesive set of services**. The **usage** must be **unidirectional**. Layers completely partition a set of software, and each partition is exposed through a **public interface**.

Layer Pattern Example

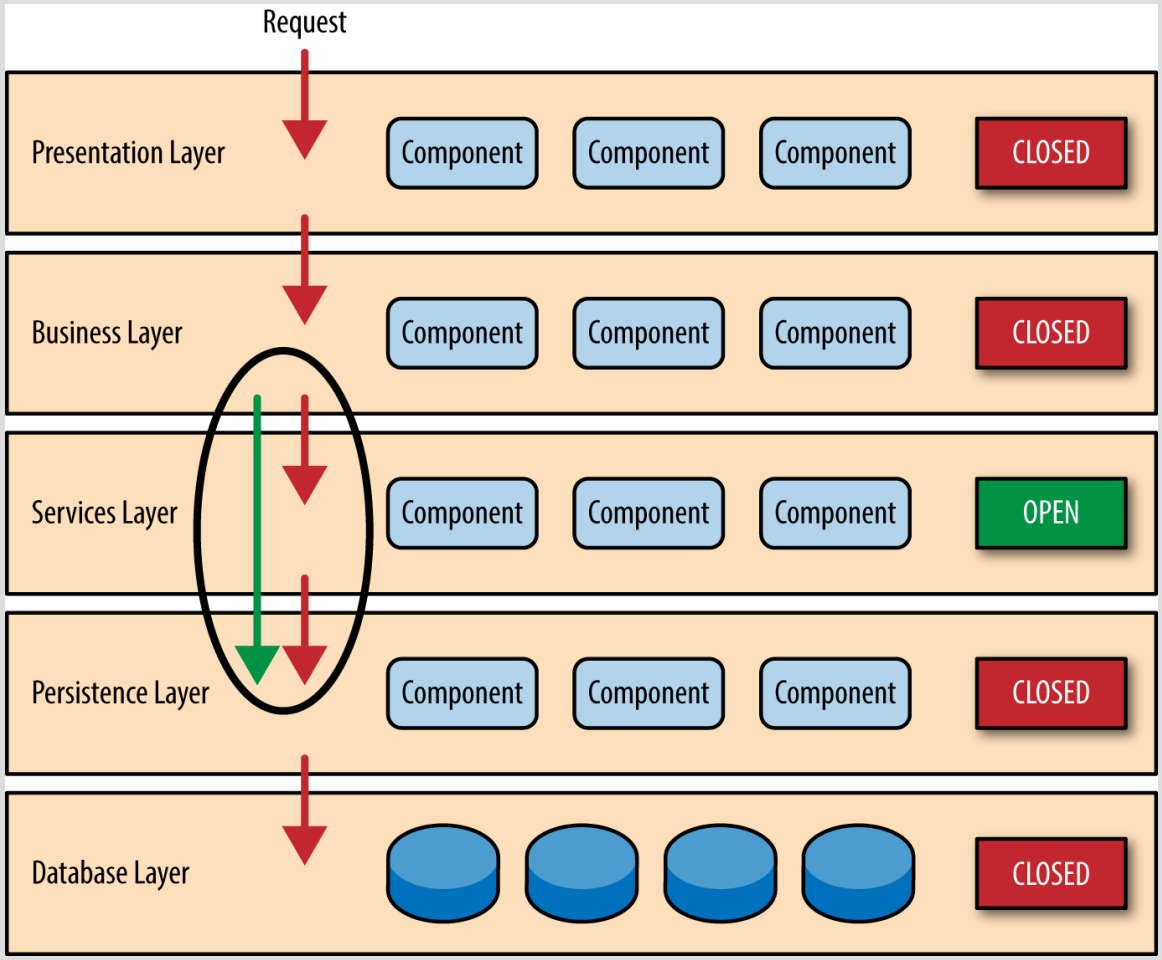


UML Package Notation for Layer Diagrams



Note: The textbook uses various informal architecture notations. That is **OK if you use legends (keys)** to explain the components, connectors, and structures.

Business IT Example



Software Architecture Patterns, Mark Richards

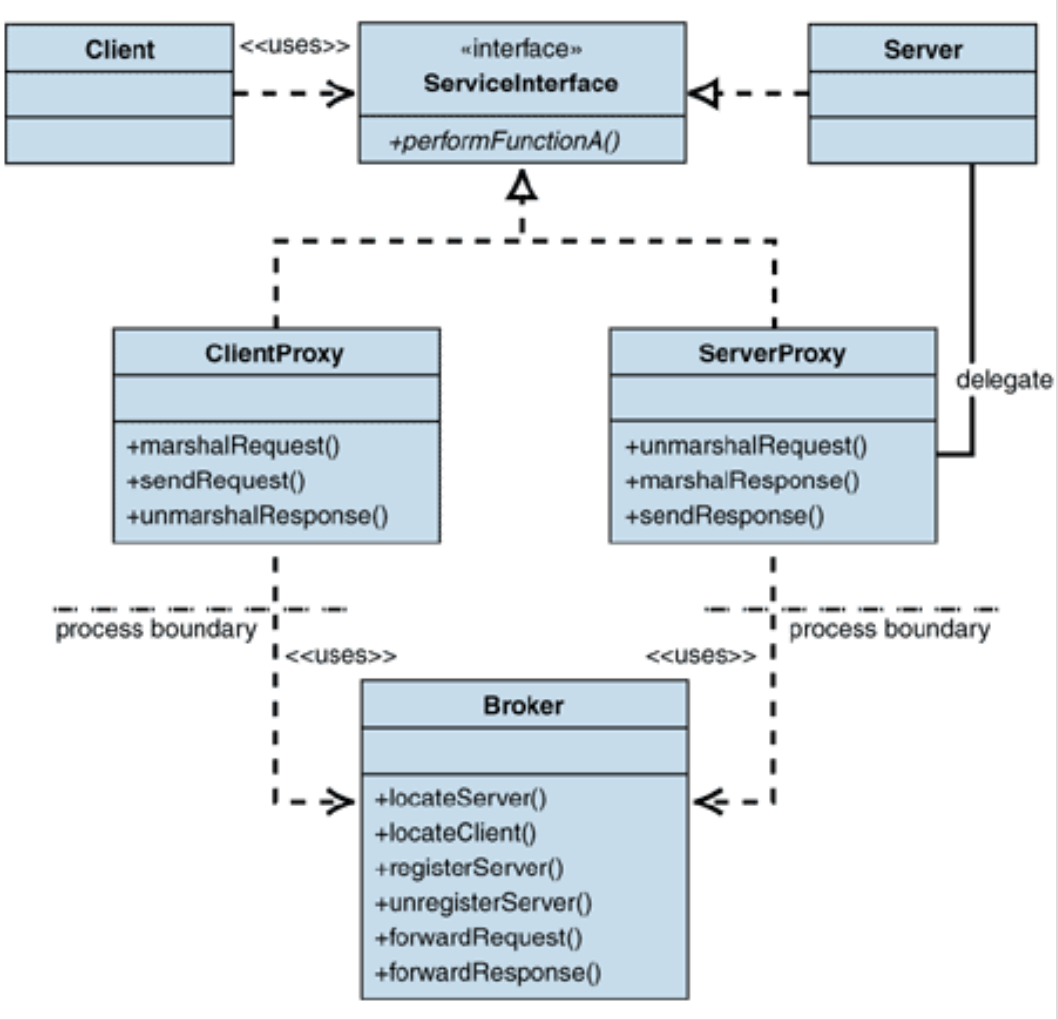
Layer Pattern Solution

- **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.
- **Elements**: ***Layer*, a kind of module**. The description of a layer should define what modules the layer contains.
- **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**.
  - 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**.

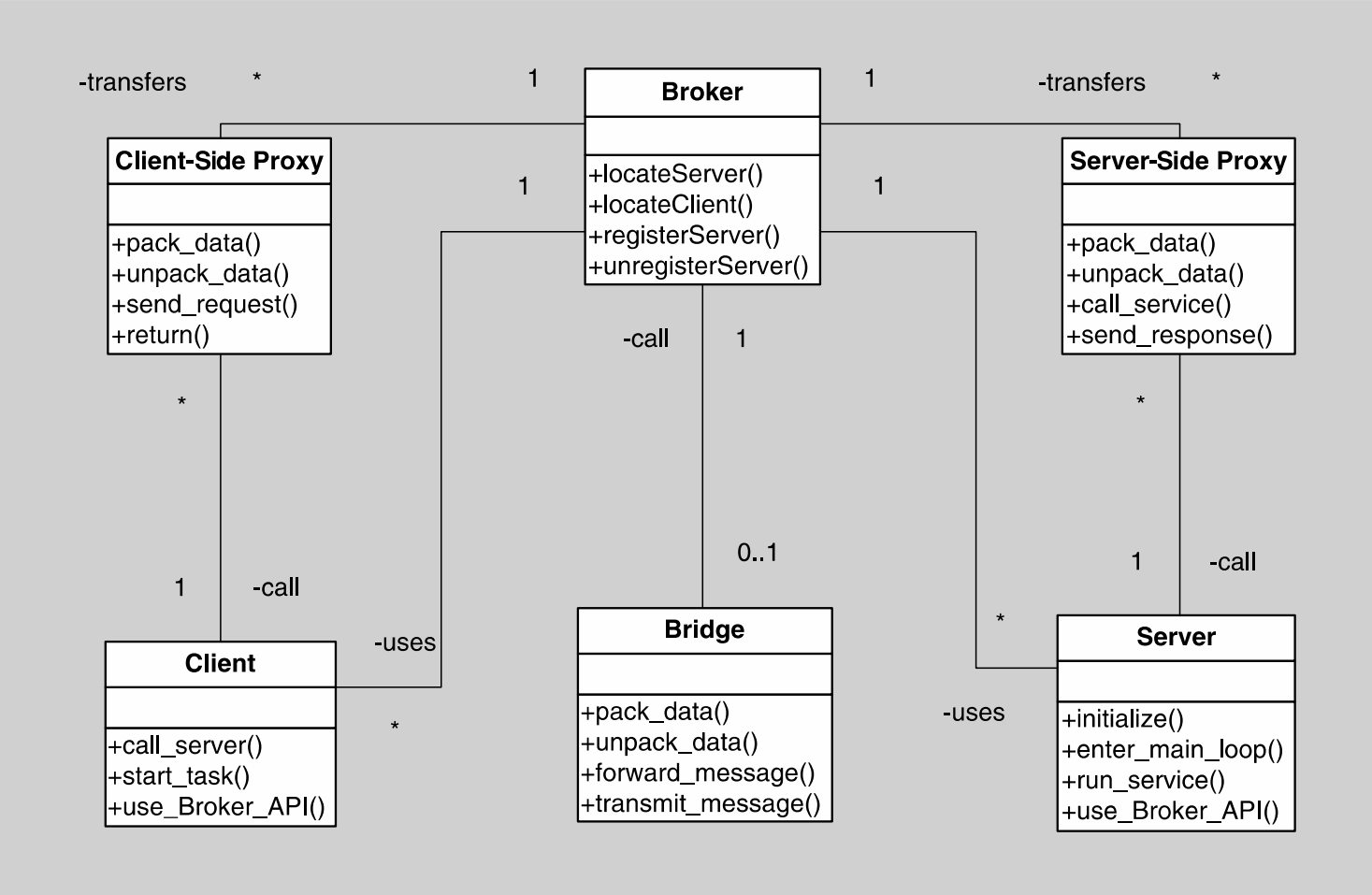
Broker Pattern

- **Context**: Many systems are constructed from a **collection of services distributed across multiple servers**. Implementing these systems 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**: How do we structure distributed software so that **service users do not need to know** the nature and location of **service providers**, making it easy to **dynamically change** the **bindings** between users and providers?
- **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. The broker then **forwards** the client's service **request to a server**, which processes the request.

Broker Example



Broker Example



Broker Solution – 1

- 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

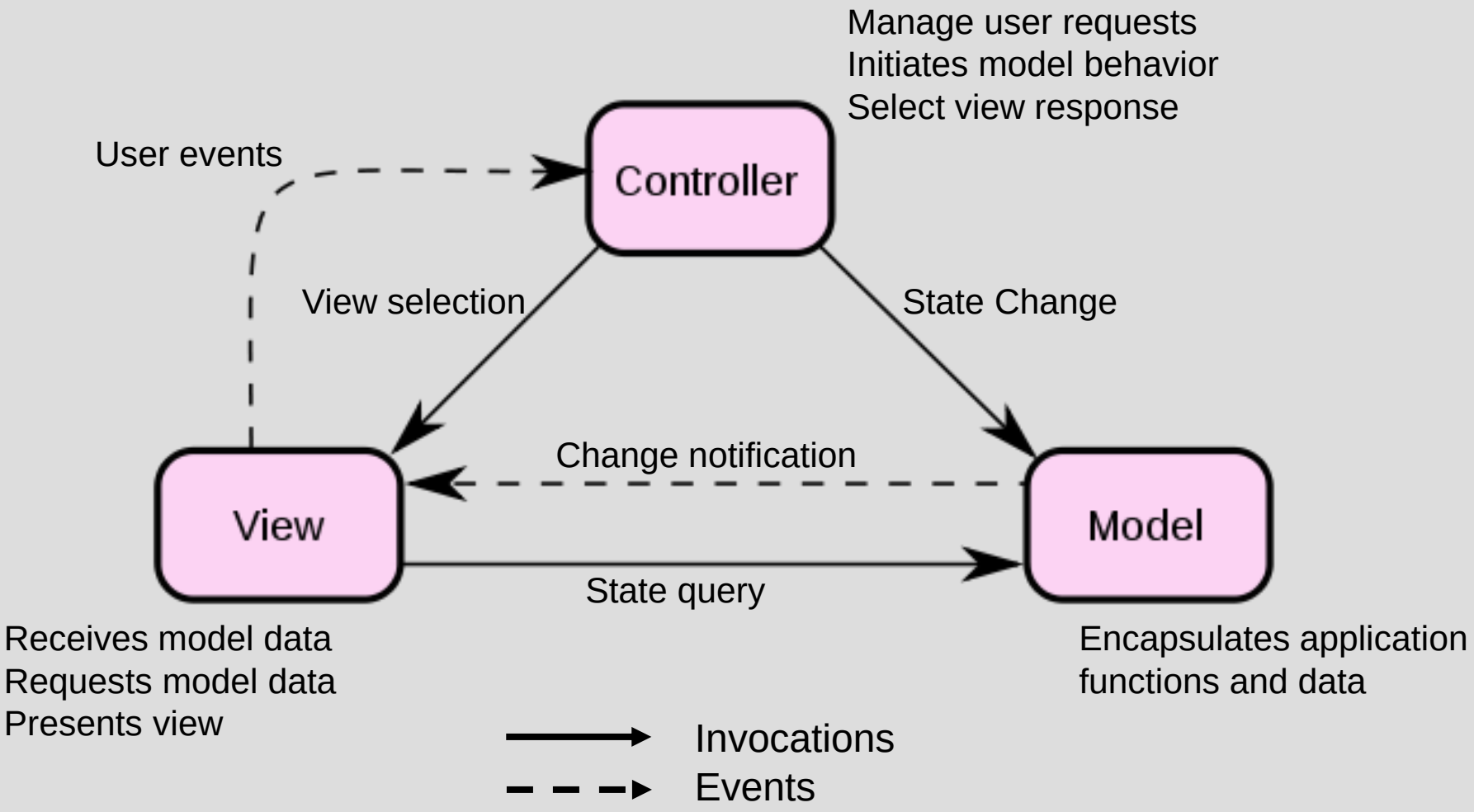
Broker Solution - 2

- 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.

Model-View-Controller Pattern

- **Context**: **User interface software** is typically the **most frequently modified** portion of an interactive application. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.
- **Problem**: How can **user interface functionality** be kept **separate** from **application functionality** and yet still be responsive to user input, or to changes in the underlying **application's data**? And how can **multiple views** of the user interface be created, maintained, and coordinated when the underlying **application data changes**?
- **Solution**: The model-view-controller (MVC) pattern separates application functionality into three kinds of components:
  - A **model**, which contains the **application's data**
  - A **view**, which **displays** some portion of the underlying **data and interacts with the user**
  - A **controller**, which **mediates between the model and the view** and manages the notifications of state changes

MVC Example



MVC Solution - 1

- 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.

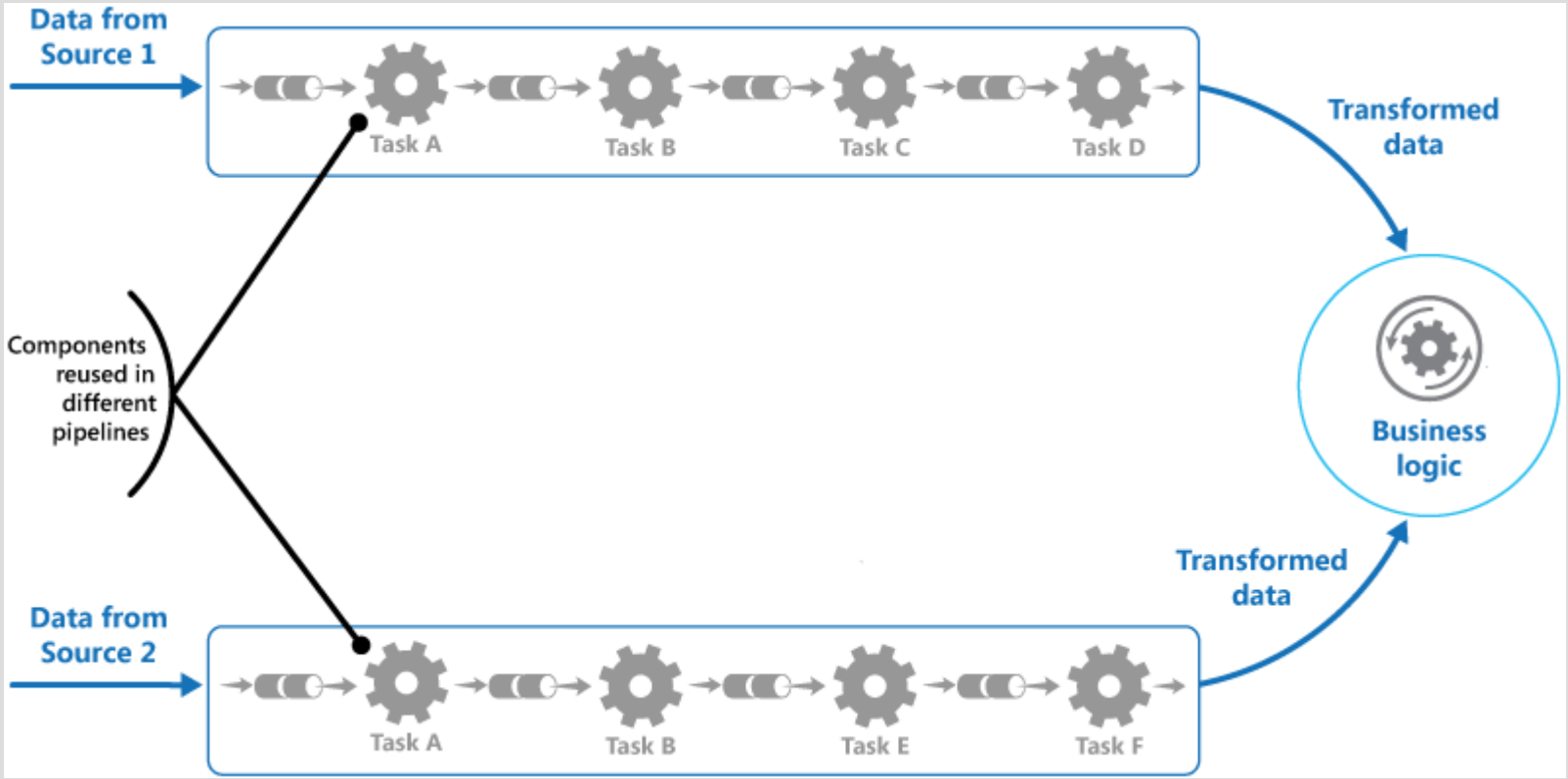
MVC Solution - 2

- 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.

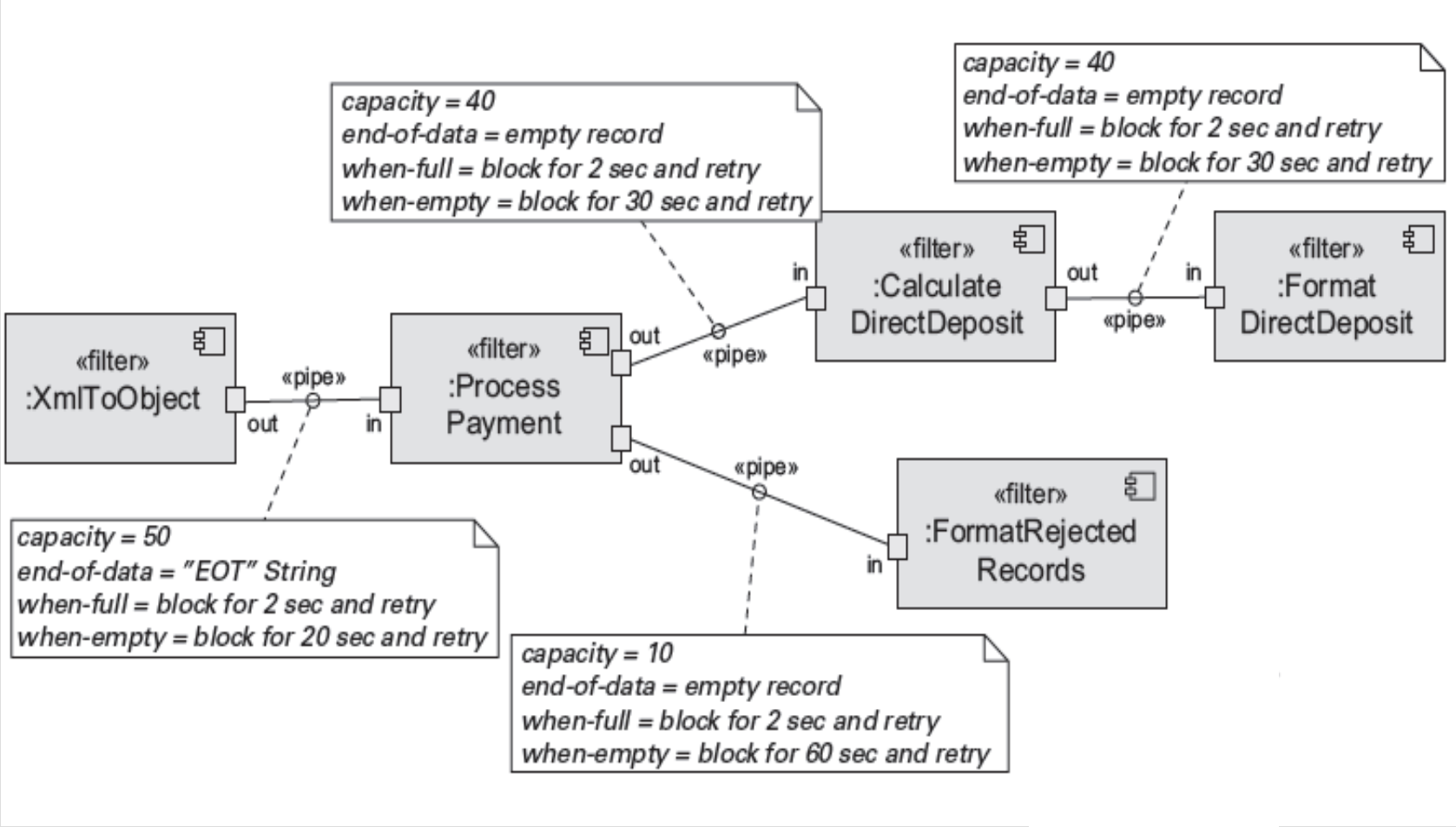
Pipe and Filter Pattern

- **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 Example



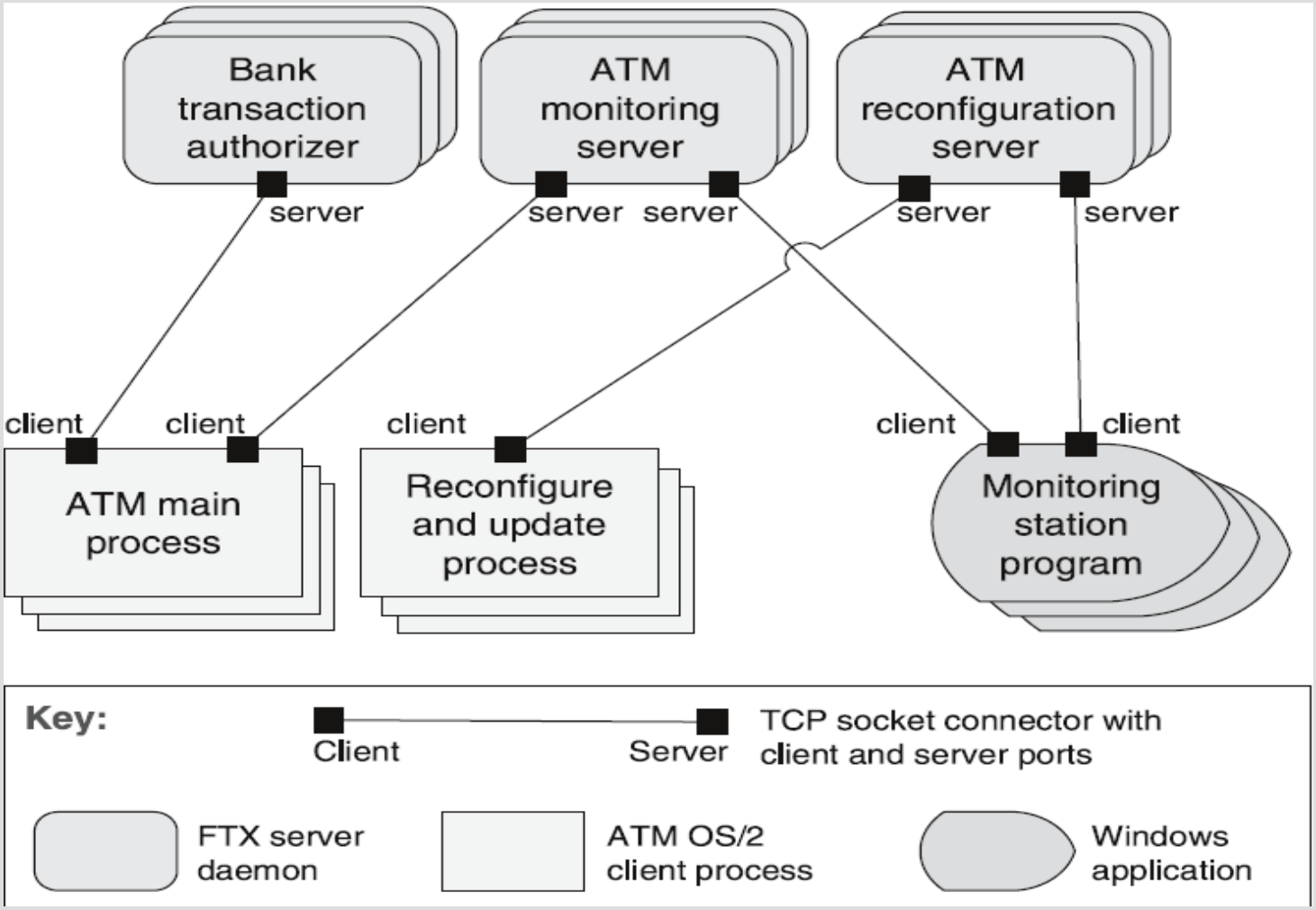
Pipe and Filter Solution

- 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.

Client-Server Pattern

- **Context:** There are **shared resources and services** that **large numbers of 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 having to modify 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 **request**ing **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
ATM Banking System



Client-Server Solution - 1

- 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.
- *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.

Client-Server Solution- 2

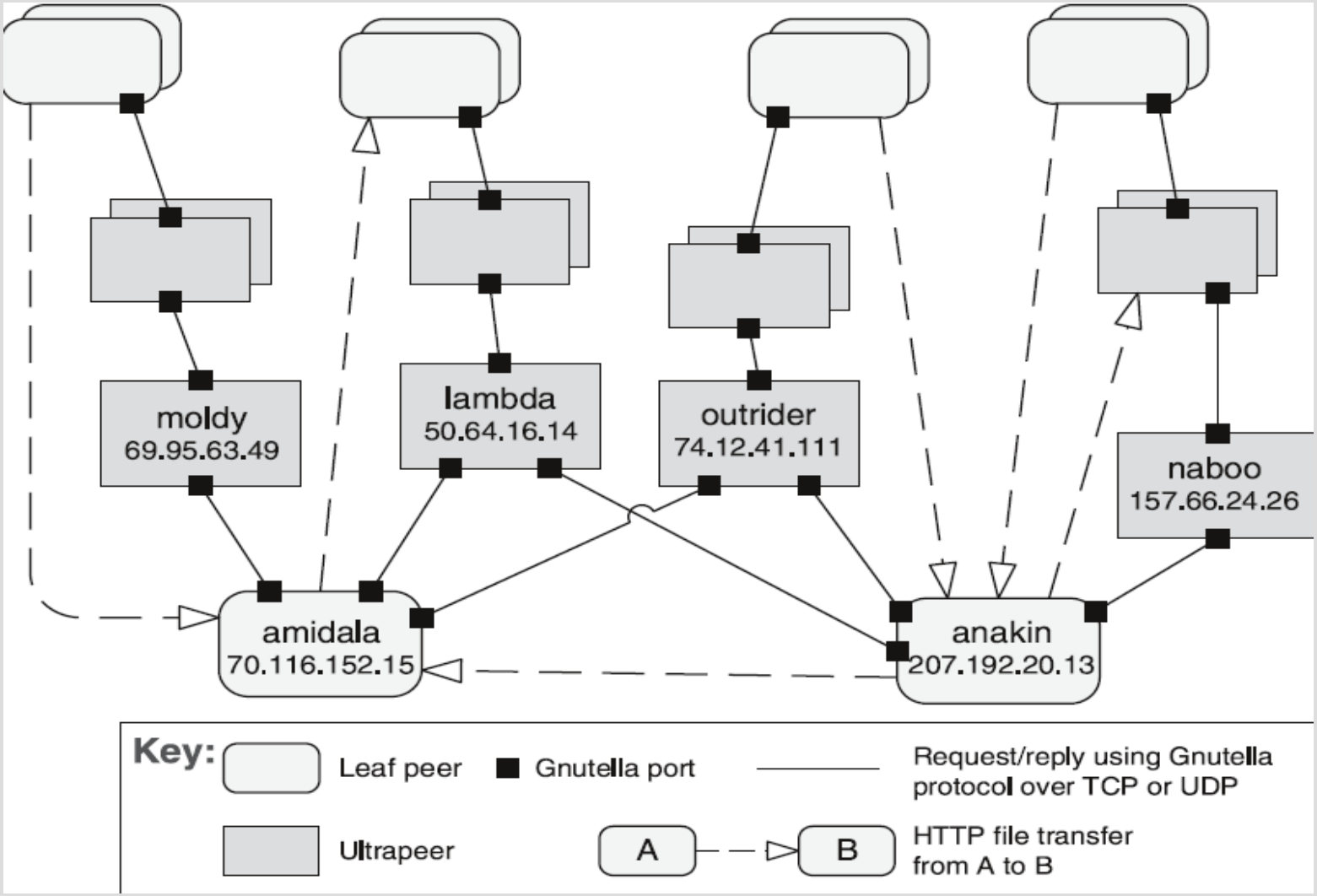
- Relations: The attachment relation associates clients with servers.
- Constraints:
  - Clients are connected to servers through request/reply connectors.
  - Server components can be clients to other servers.
- Weaknesses:
  - Server can be a performance bottleneck.
  - Server can be a single point 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.

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 **connect**ed 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

Gnutella Network



Peer-to-Peer Solution - 1

- 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.

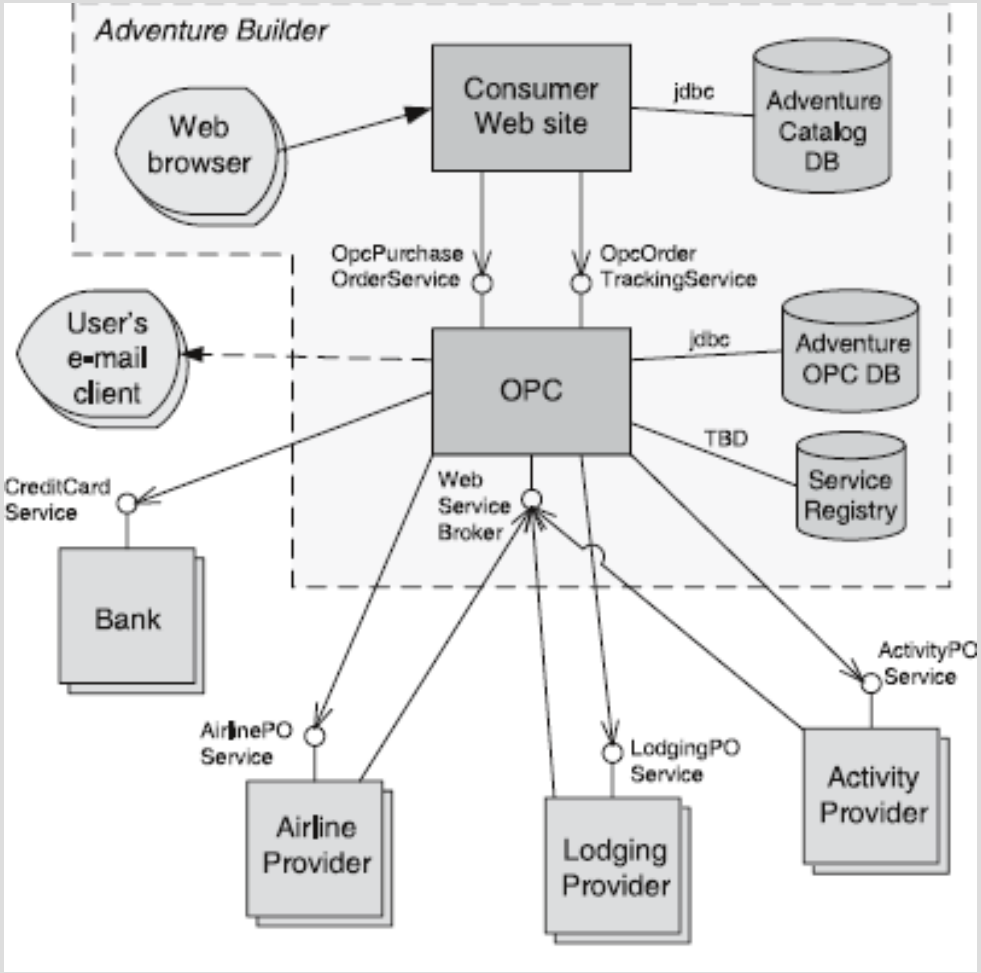
Peer-to-Peer Solution - 2

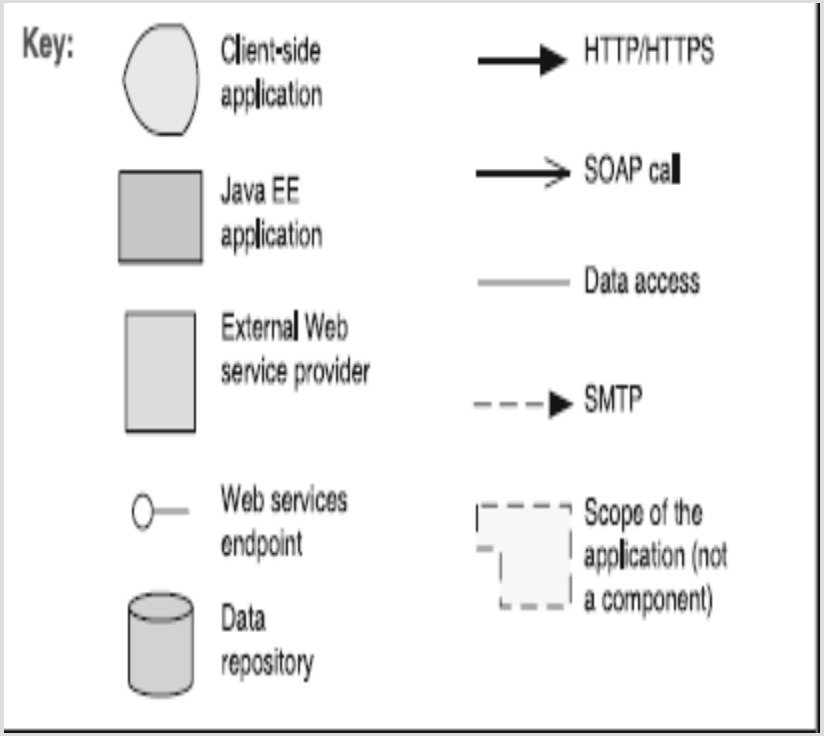
- Constraints: Restrictions may be placed on the following:
  - The number of allowable attachments to any given peer
  - The number of hops used for searching for a peer
  - Which peers know about which other peers
  - Some P2P networks are organized with star topologies, in which peers only connect 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.

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
"Adventure Builder"





OPC – Order Processing Center

Service Oriented Architecture Solution - 1

- Overview: Computation is achieved by a set of cooperating components that provide and/or consume services over a network.
- Elements:
  - Components:
    - *Service providers,* which provide one or more services through published interfaces.
    - *Service consumers,* which invoke services directly or through an intermediary.
    - *Service providers* may also be service consumers.
  - 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.
  - *Orchestration server,* which coordinates the interactions between service consumers and providers based on languages for business processes and workflows.

Service Oriented Architecture Solution - 2

- 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 - 3

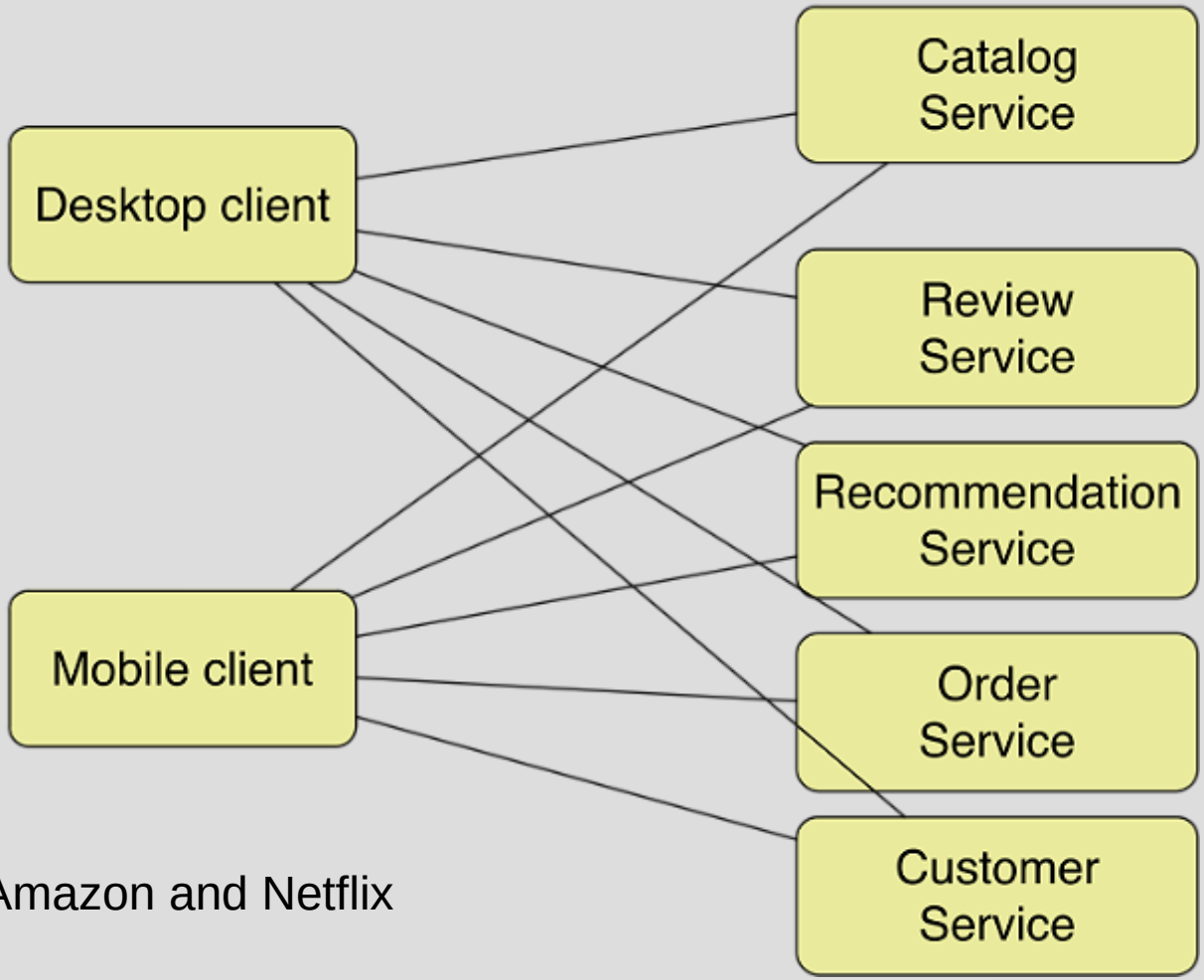
- Relations: Attachment of the different kinds of components available to the respective connectors
- Constraints: Service consumers are connected to service providers, but intermediary components (e.g., ESB, registry, orchestration server) may be used.
- Weaknesses:
  - SOA-based systems are typically 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.

Microservices Architecture Pattern

- **Context** - deploy server based enterprise applications that support a variety of browser and native mobile clients. The application handles client requests by executing business logic, accessing a database, exchanging messages with other systems, and returning responses. The application might expose a 3rd party API.
- **Problem** – Monolithic applications can become too large and complex for efficient support, and deployment for optimal distributed resource utilization such as in cloud environments
- **Solution** - build applications as suites of services. Each service is independently deployable and scalable, and has its own API boundary. Different services can be written in different programming languages, manage their own database, and developed by different teams

http://martinfowler.com/articles/microservices.html

Microservices Architecture Pattern Example



Used by Amazon and Netflix

Microservices Architecture Pattern Solution

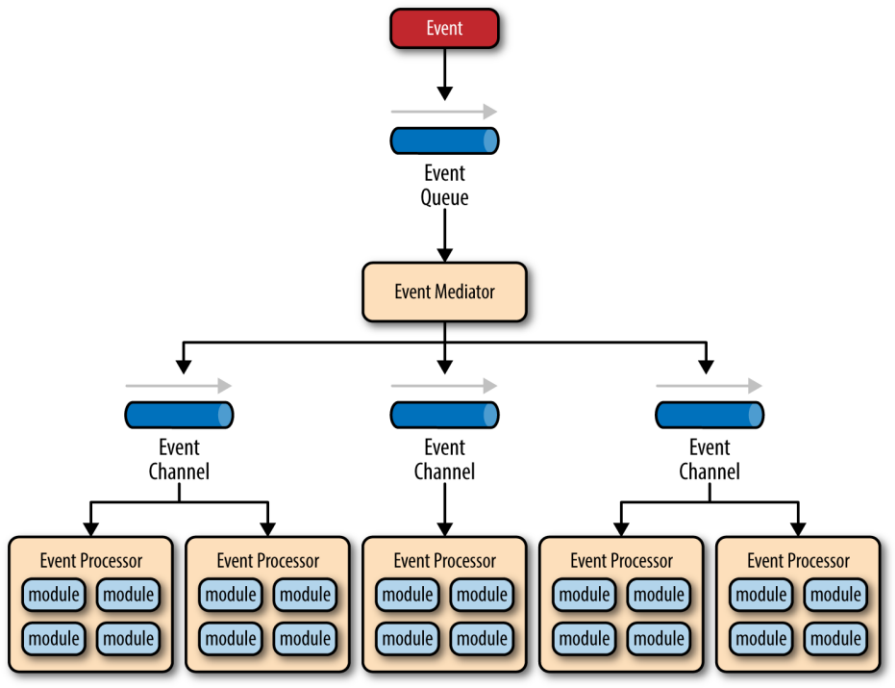
- Overview – decompose business logic into a series of **independently developed and deployable services**
- Elements – services are packaged as out-of-process components with well defined service boundaries (APIs). They can be implemented in any programming language. They may manage their own database as part of the service boundary.
- Relationships – clients invoke services via simple remote procedure calls/web services using for example HTTP RESTful interfaces, or using a lightweight message bus.
- Constraints – complexity of distributed systems. Decentralized data management is harder to manage such as support for cross service transactions. Development team experience to make good service decomposition decisions, testing and deployment know how
- Weaknesses – systems must be designed to tolerate service failures which requires more system monitoring. Service choreography and event collaboration overhead. More memory

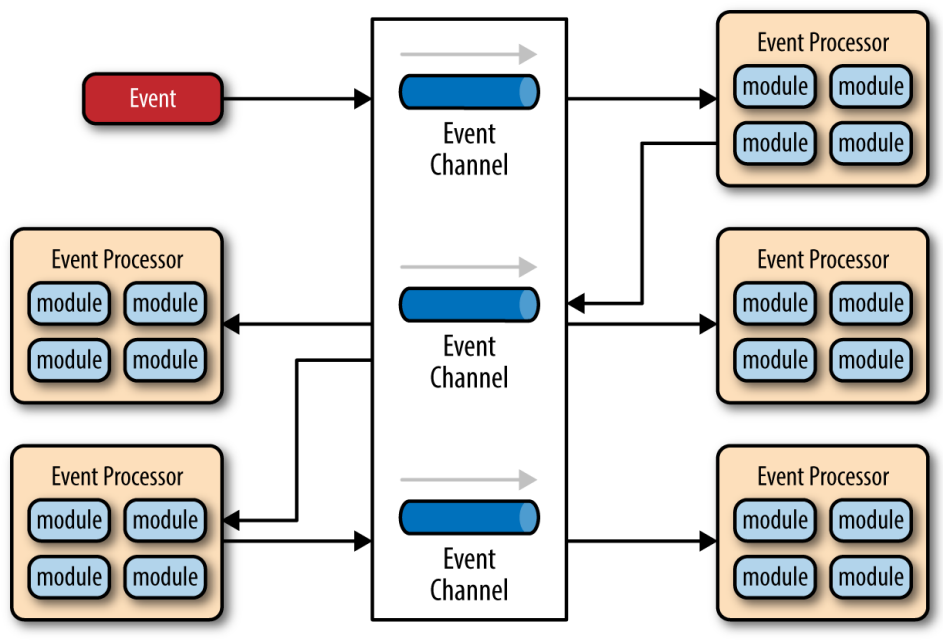
Event Driven Architecture

- **Context –** computational and information **resources** need to be provided to **handle incoming independent asynchronous application generated events** in a manner that can **scale** up as demand increases.
- **Problem –** construct **distributed** systems that can **service asynchronous arriving messages associate with an event**, and that can **scale from small and simple to large and complex**.
- **Solution –** deploy **independent event processes/processors** for event handling. Arriving events are **queued**. A **scheduler pulls events** from the queue and **distributes** them to the appropriate **event handler** based on a scheduling **policy**.

Software Architecture Patterns, Mark Richards

Event Driven Architecture Example





Mediator topology – **orchestrate multiple steps** to handle an event according to some application policy

Broker topology – distribute messages in a **simple chained** message **flow**

Software Architecture Patterns, Mark Richards

Event Driven Architecture Pattern Solution

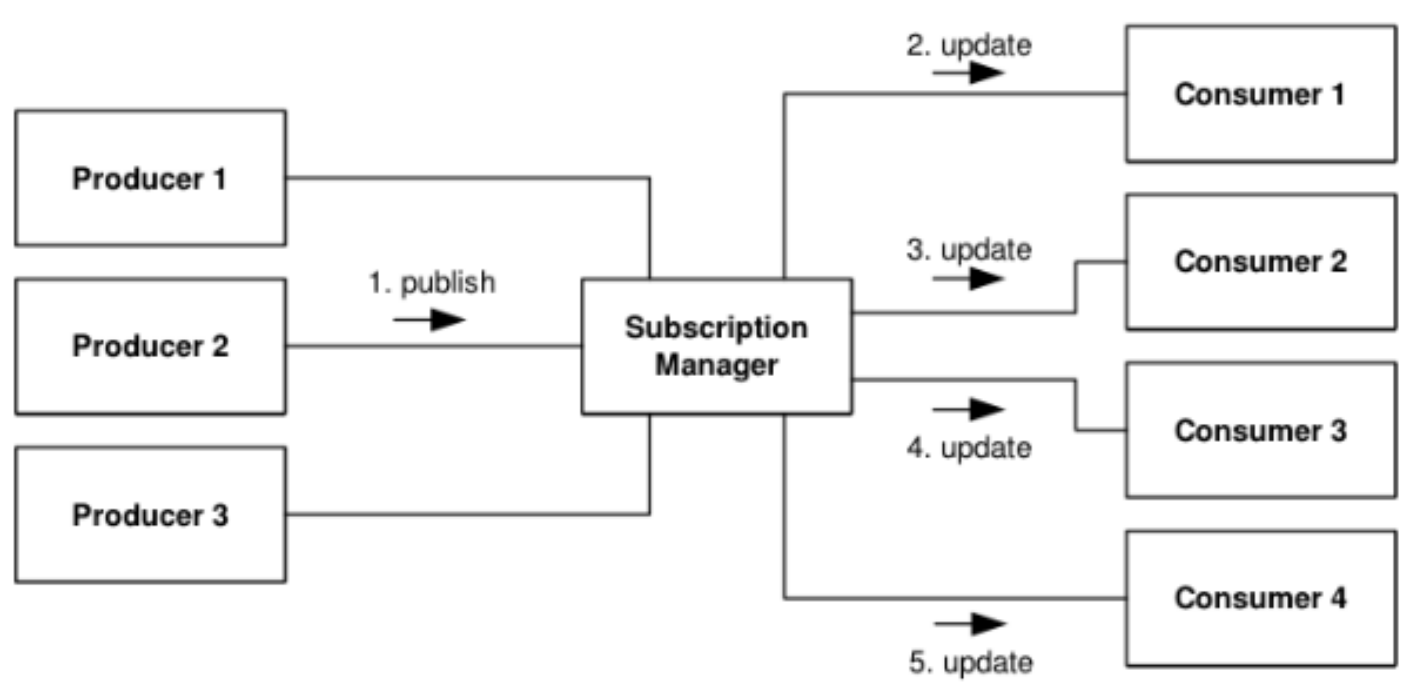
- Overview – arriving client generated requests for service arrive as events. They are queued, and then directed to an appropriate event handler according to some application policy. This pattern can be used in a wide range of applications.
- Elements – the event queue, the scheduler, and the collection of pertinent event handlers. Event handlers may be deployed as independent processes and/or processors.
- Relationships – clients and event handlers are usually distributed on a network but do not have to be. Message communication protocols and message formats should be consistent and standards based
- Constraints – there is no built-in transactional support. The inherent complexity of deploying distributed systems.
- Weaknesses – performance and error recovery may be issues. On the plus side, with low coupling the resulting systems are highly scalable.

Software Architecture Patterns, Mark Richards

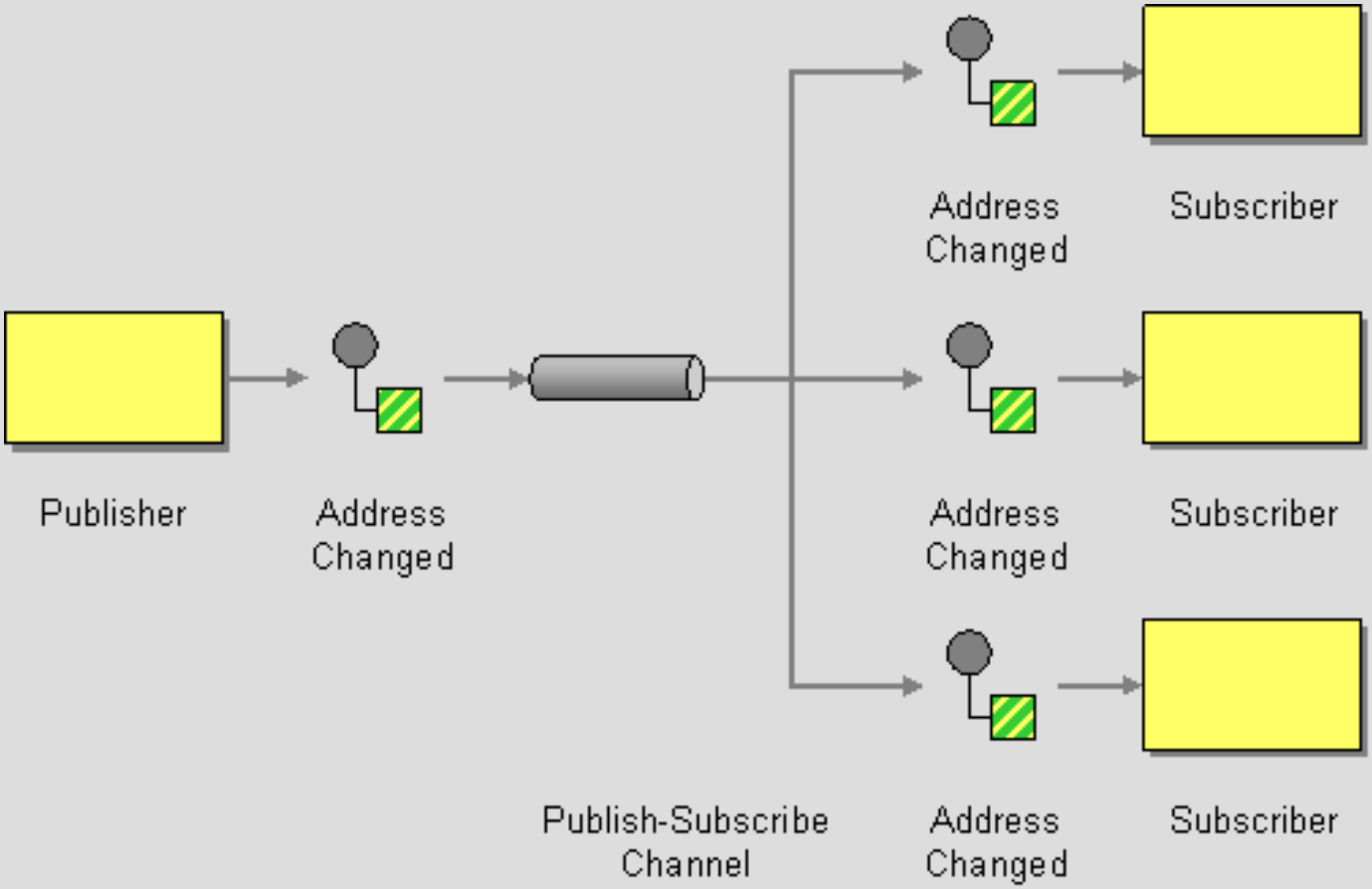
Publish-Subscribe Pattern

- **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 Example



http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html

Publish-Subscribe Solution – 1

- 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.
  - *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.

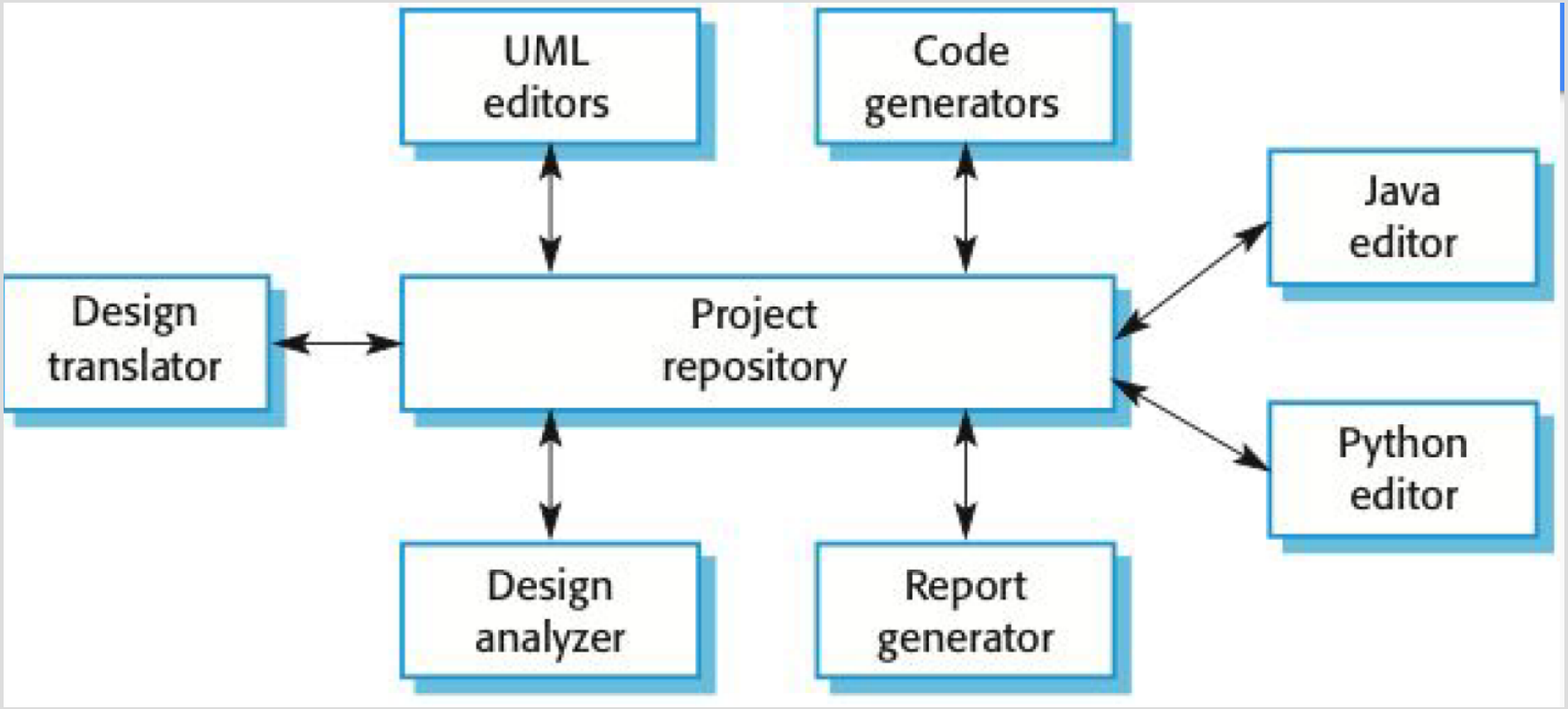
Publish-Subscribe Solution - 2

- 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.
- 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.

Shared-Data Pattern

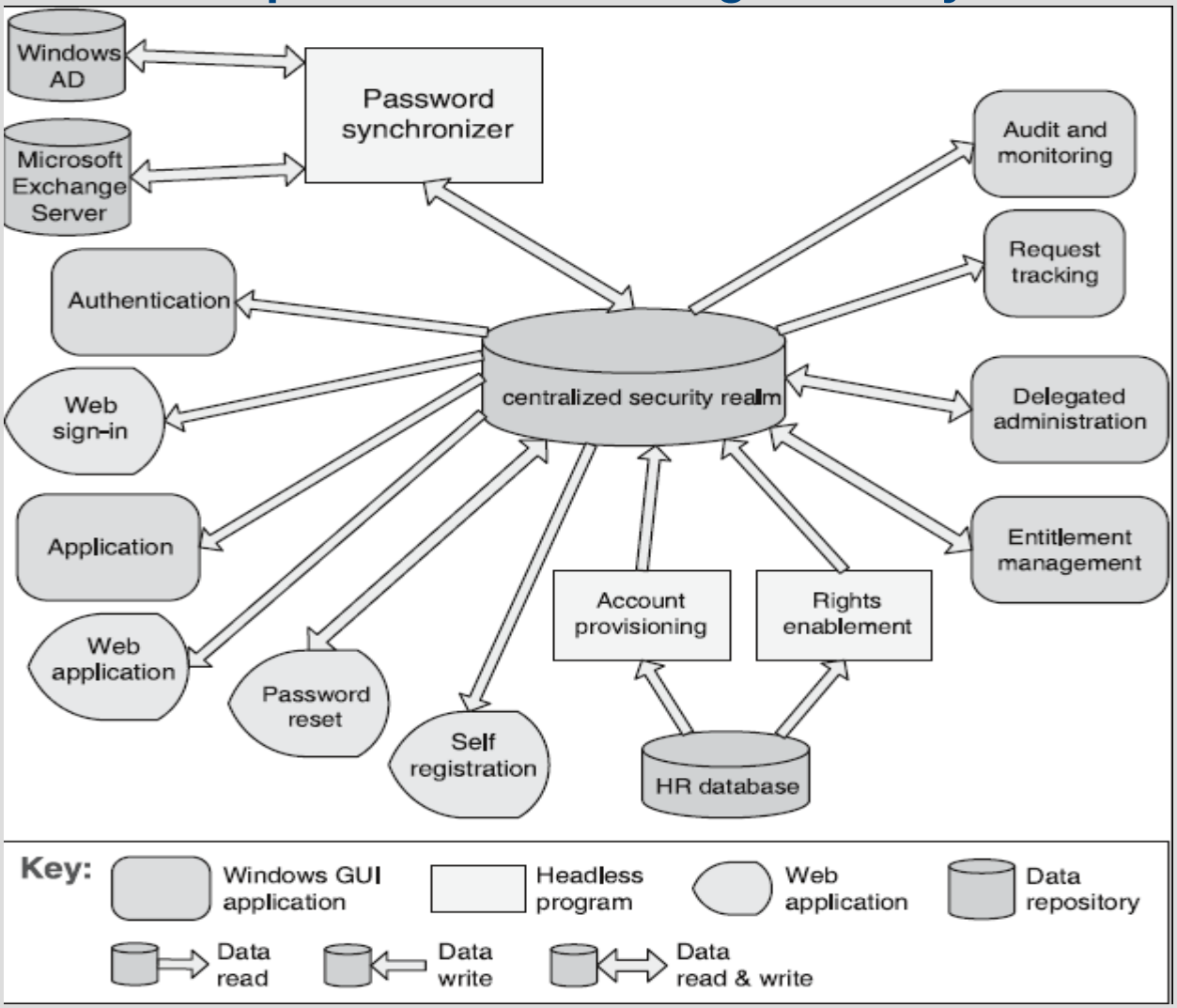
- **Context:** Various **computational components** need to **share** and **manipulate large amounts of data**. This data does not belong solely to any one of those components.
- **Problem:** How can systems **store and manipulate persistent data** that is **accessed** by **multiple independent components**?
- **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 Example

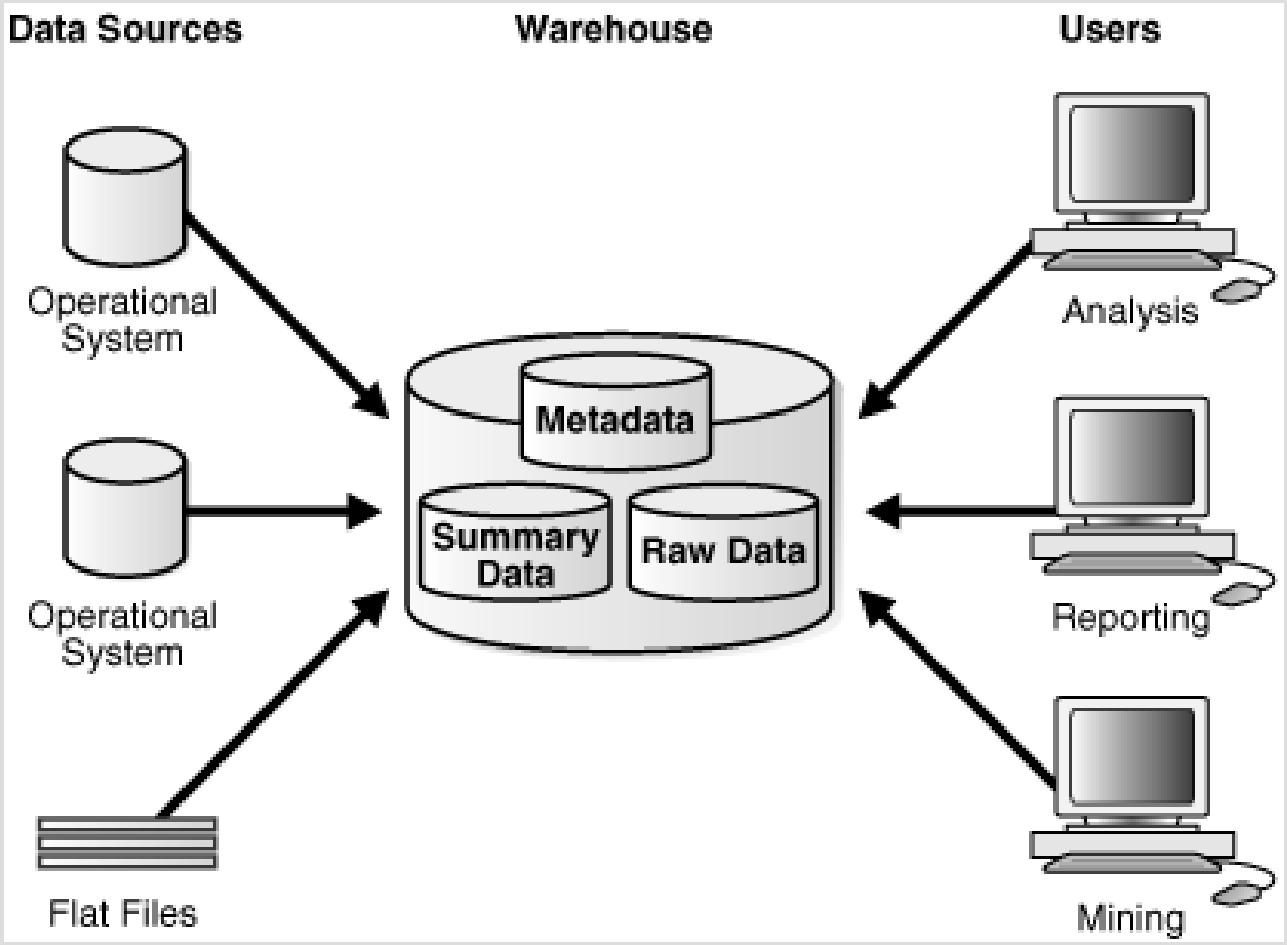


Shared Data Example

Enterprise Access Management System



Data Warehouse Example



Oracle – Introduction to Data Warehousing Concepts

Shared Data Solution - 1

- 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.

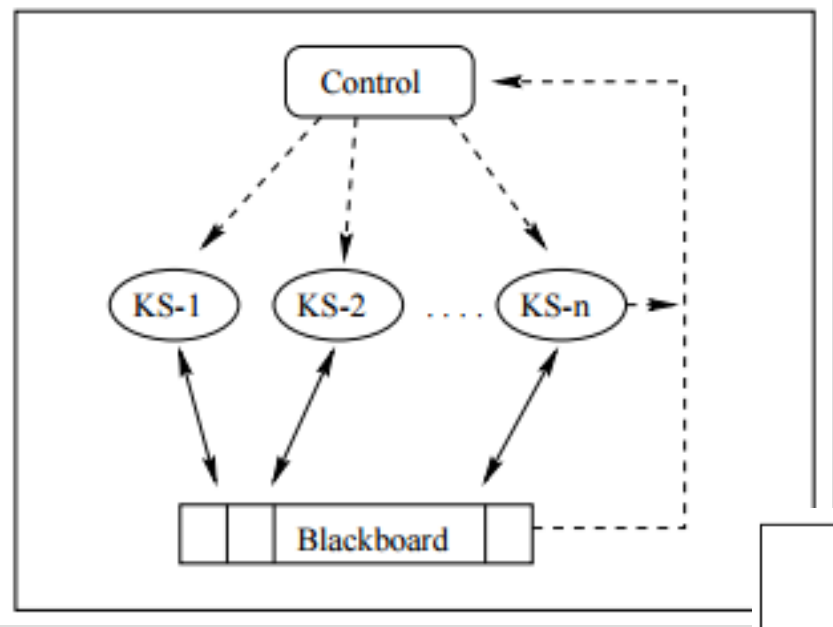
Shared Data Solution - 2

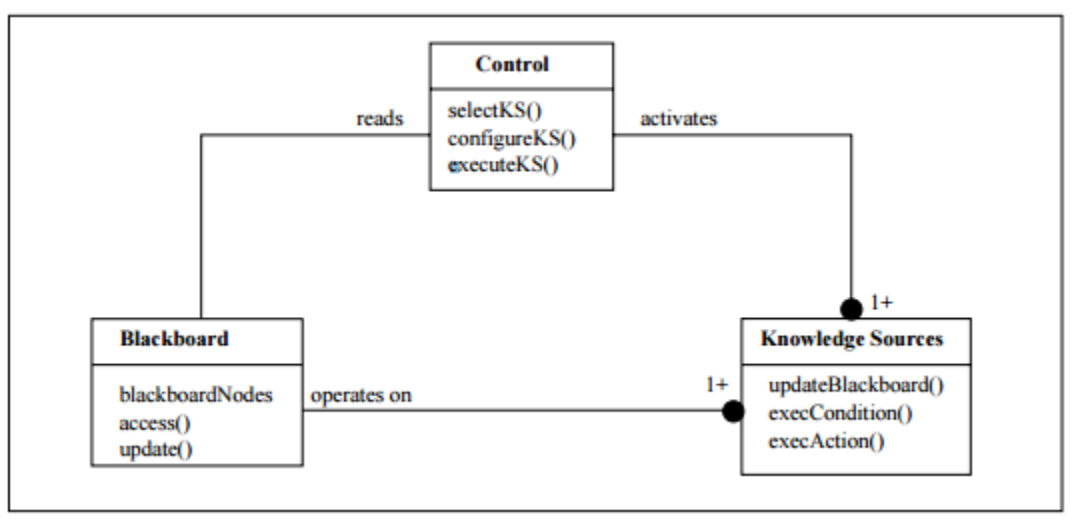
- Relations: Attachment relation determines which data accessors are connected to which data stores.
- Constraints: Data accessors interact only with the data store(s).
- Weaknesses:
  - The shared-data store may be a performance bottleneck.
  - The shared-data store may be a single point of failure.
  - Producers and consumers of data may be tightly coupled.

Blackboard Pattern

- Context - An immature or **poorly specified domain** with **no deterministic or optimal solution** known for the problem. Hence, software systems that need to **integrate large and diverse specialized modules, and implement complex, nondeterministic control strategies**. E.g., speech recognition, other AI “expert” applications. Metaphor of humans collaborating with a blackboard.
- Problem –The problem **spans several fields of expertise** that require a **sequence** of independent algorithmic **transformations.** Intermediate **solutions** require **different representations**. Algorithm experimentation may be required. The **control strategy is complex** and cannot be determined statically.
- Solution - a **collection of independent specialized programs work cooperatively** utilizing a **common data structure**. A **central controller** coordinates the knowledge sources based on the state of the solution.

Blackboard Example





Blackboard Solution - 1

- Overview - Problem solvers work independently (and opportunistically) on parts of the problem. They share a common data structure (the blackboard). A central controller manages access to the blackboard. The blackboard may be structured (e.g. into levels of abstraction) so problem solvers may work at different levels. Blackboard contains original input and/or partial solutions
- Elements
  - **Knowledge source** - separate, independent subsystems that solve specific aspects of the overall problem
  - **Blackboard** - the central data store for solution space and control data It provides an interface to enable knowledge source access
  - **Control** – uses data in the blackboard to coordinate the sequence interaction between knowledge sources

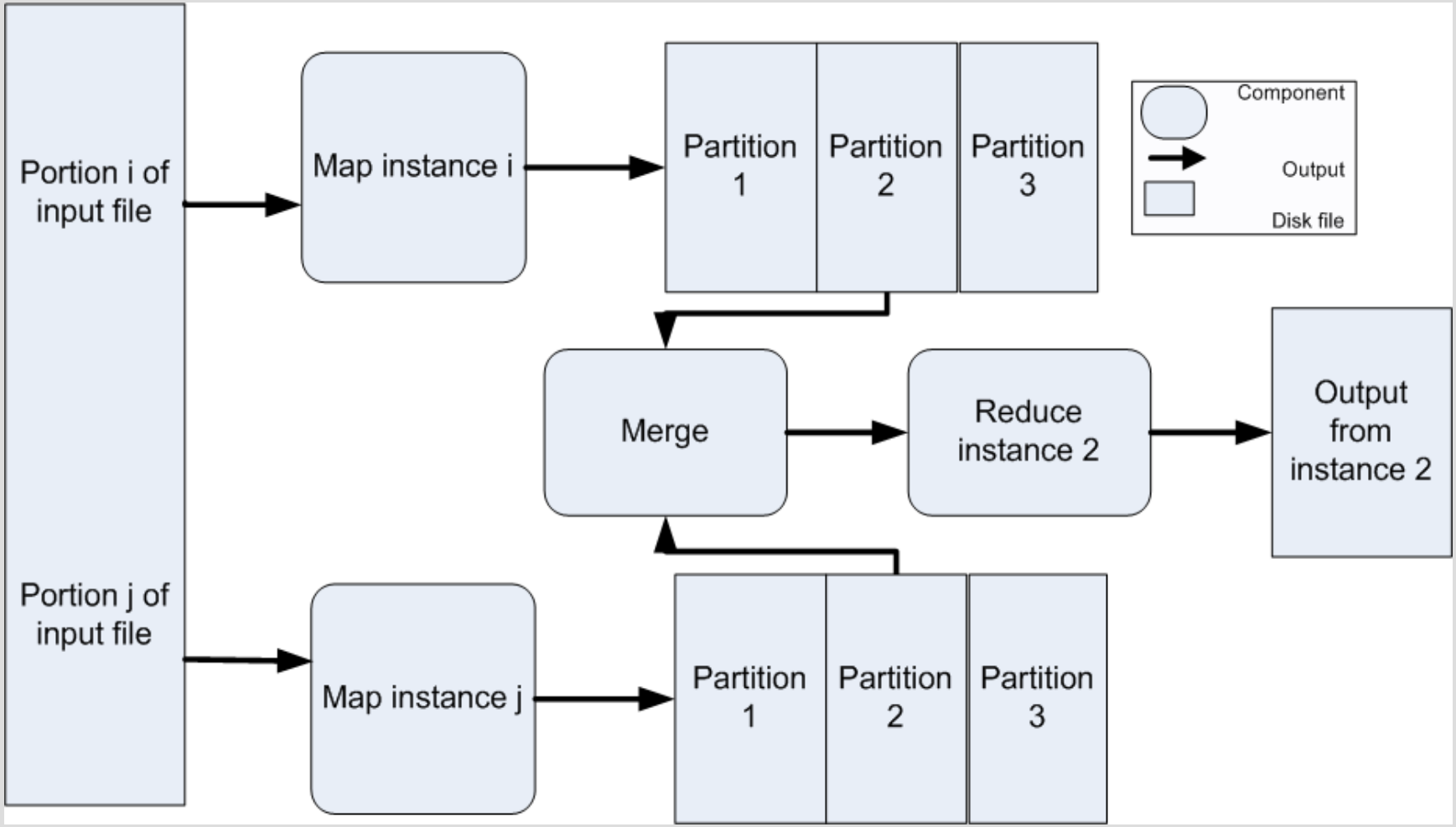
Blackboard Solution – 2

- Relations – knowledge sources are completely independent. They are coordinated by the state of data in the blackboard and control directed actuation.
- Constraints – computationally expensive, low support for parallelism, no good solution is guaranteed, control solution may be heuristic
- Weaknesses - Difficulty of testing, may be hard to develop

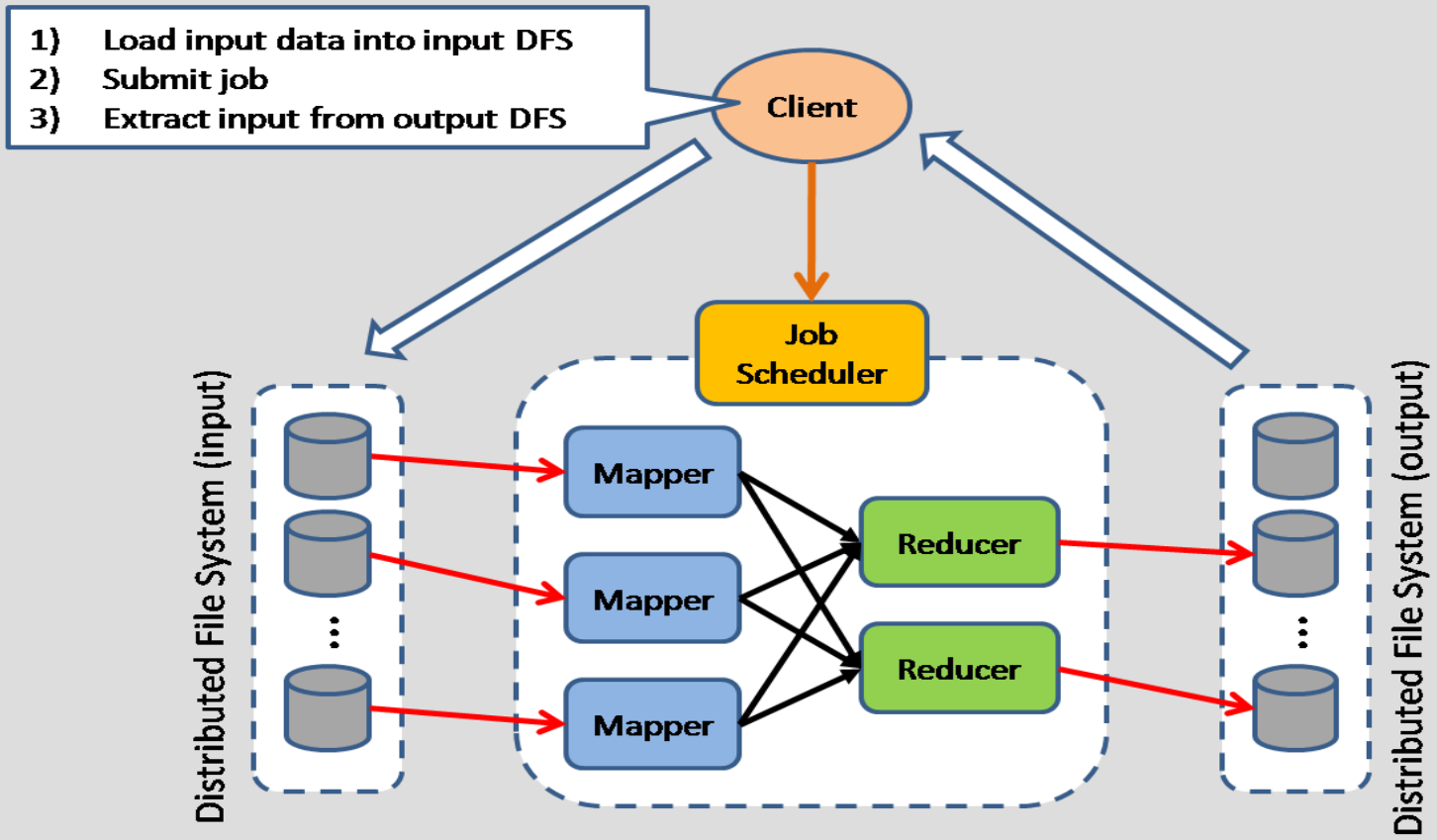
Map-Reduce Pattern

- **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 software** to the **hardware** nodes in a **massively parallel computing environment** and handles **sorting** the **data** as needed.
  - A programmer specified **component called the map** which **filter**s the **data** to retrieve those items to be combined.
  - A programmer specified **component called reduce** which **combines the results of the map**

Map-Reduce Example



Another Map-Reduce View



Map-Reduce Solution - 1

- 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 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.

Map-Reduce Solution - 2

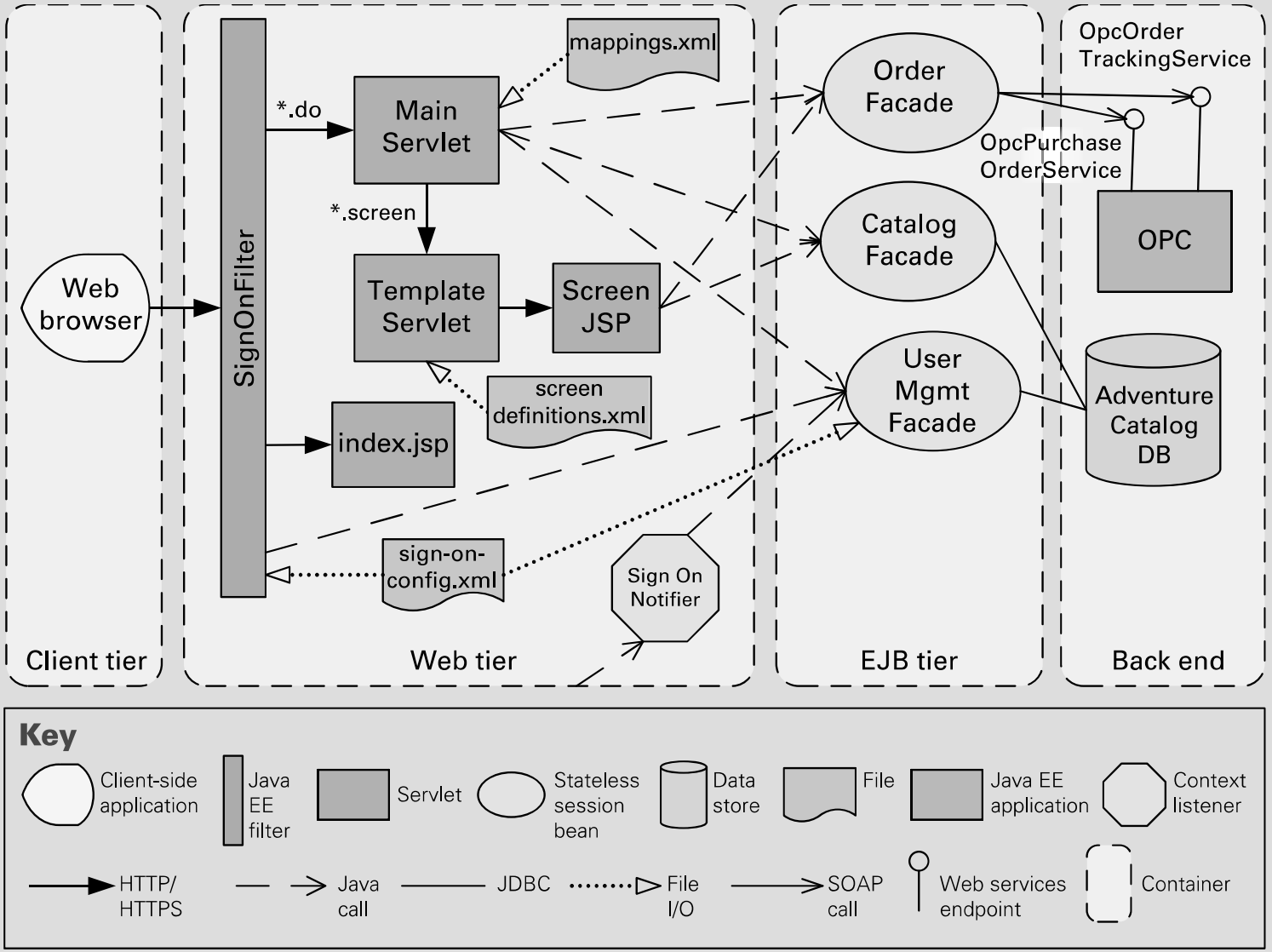
- Relations:
  - Deploy on is the relation between an instance of a map or reduce function and the processor onto which it is installed.
  - Instantiate, monitor, and control is the relation between the infrastructure and the instances of map and reduce.
- Constraints:
  - The data to be analyzed must exist as a set of files.
  - 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.

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

Consumer Web Site Java EE



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.
- Weaknesses: Substantial up-front cost and complexity.