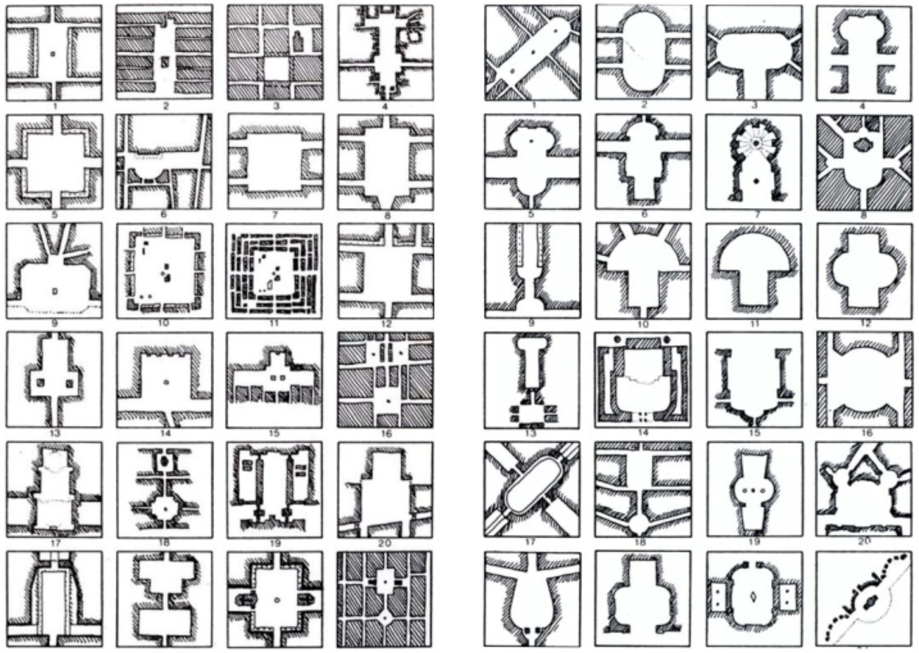
Architectural styles for software systems

Prof. Paolo Ciancarini
Software Architecture
CdL M Informatica
Università di Bologna

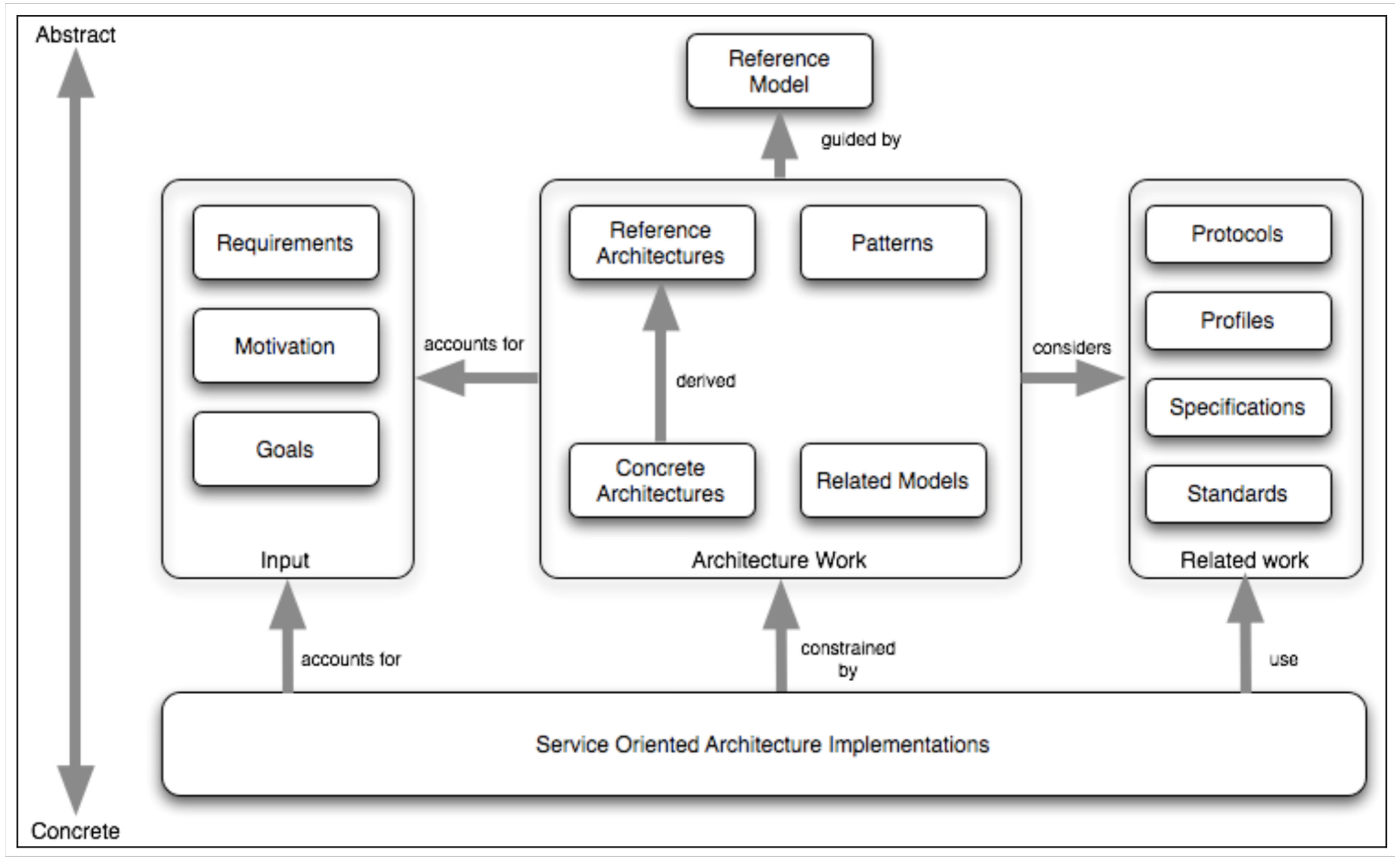
Agenda

- Types of architectural styles
- Basic decomposition techniques
  - layering, tiering
- Architectural styles
  - Pipes and filters
  - Repository
    - Passive repository
    - Active repository
  - Client/Server
    - two-tiers;
    - three-tiers;
    - n-tiers (microservices)
  - Peer-To-Peer
    - blockchain

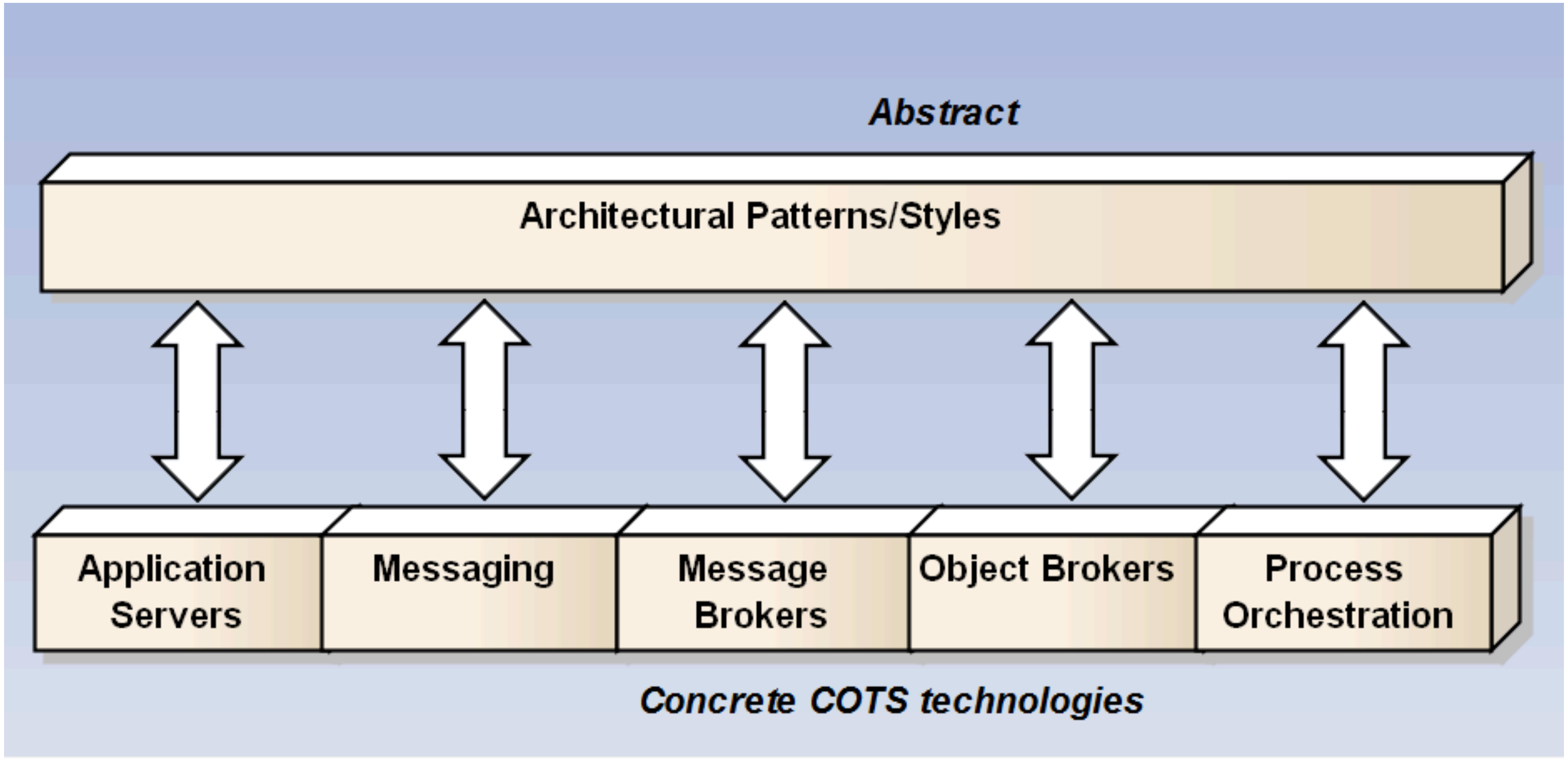


Reference architectures and patterns

A **Reference Architecture** is, in essence, a predefined architectural **pattern**, or set of **patterns**, possibly partially or completely instantiated



From patterns to technologies



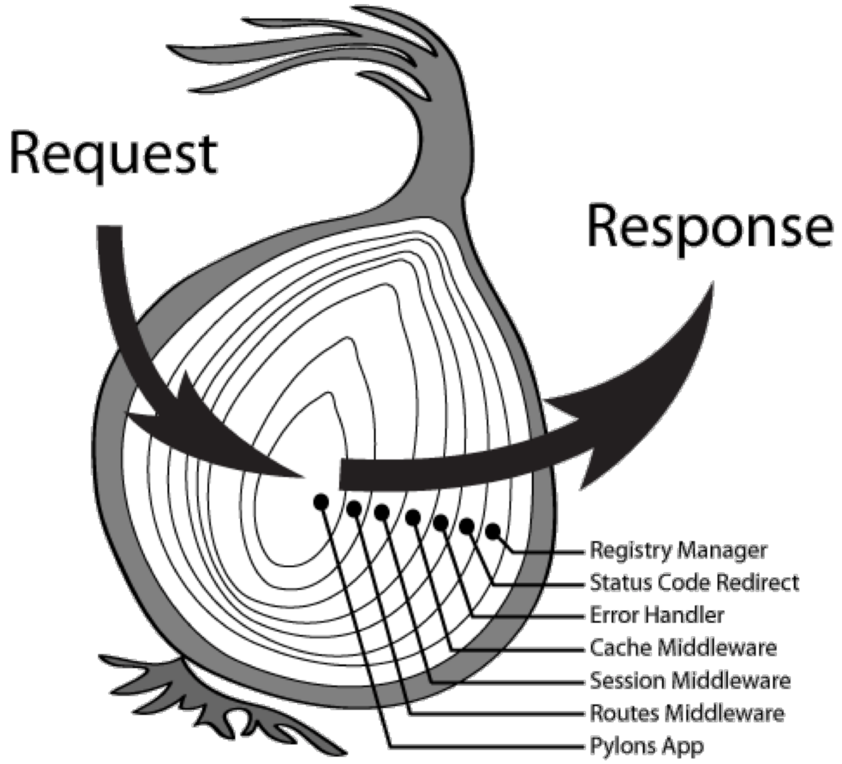
Architectural elements

- Components
  - Processes
  - Objects
  - Agents
  - Services
- Connectors
  - Channels
  - Protocols (middleware)
  - Name systems

What is an architectural style?

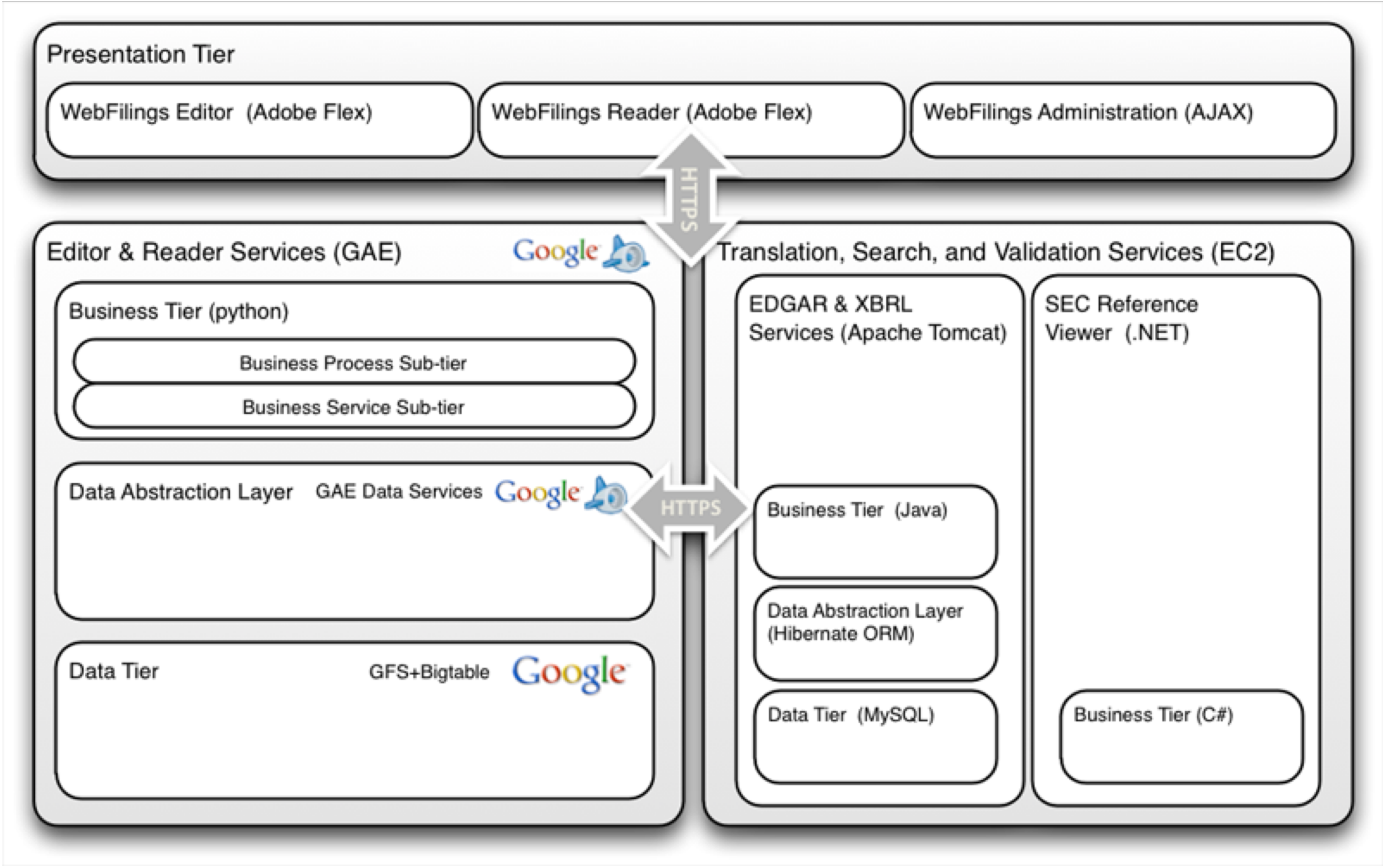
- A family of systems sharing the configuration (structure, behaviors) of their architectural elements
- A vocabulary of components and connectors, with constraints on how they can be combined (Garlan and Shaw)
- A set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done (Shaw & Clements)
- A set of constraints put on system development, namely a collection of design decisions applicable in a given context, specific to a particular system within that context, and choosing some beneficial qualities in each resulting system (Taylor, Medvidović and Dashofy)

Architectural vegetables



Onion: layers

Garlic: partitions


Presentation Tier
WebFilings Editor (Adobe Flex)
WebFilings Reader (Adobe Flex)
WebFilings Administration (AJAX)
HTTPS
Editor & Reader Services (GAE)
Google
Business Tier (python)
Business Process Sub-tier
Business Service Sub-tier
Data Abstraction Layer
GAE Data Services
Google
Data Tier
GFS+Bigtable
Google
HTTPS
Translation, Search, and Validation Services (EC2)
EDGAR & XBRL Services (Apache Tomcat)
Business Tier (Java)
Data Abstraction Layer (Hibernate ORM)
Data Tier (MySQL)
SEC Reference Viewer (.NET)
Business Tier (C#)

Layers and tiers (1)

- The architecture of a system is generally achieved by decomposing it into subsystems, following a ***layered*** and/or a **partition** based approach
- These are orthogonal approaches:
  - A layer is a logical structuring mechanism for the elements that make up a software solution (e.g., a kernel is a layer)
  - A partition (or **tier**) is a physical structuring mechanism for the system infrastructure (e.g., a user interface is a tier)

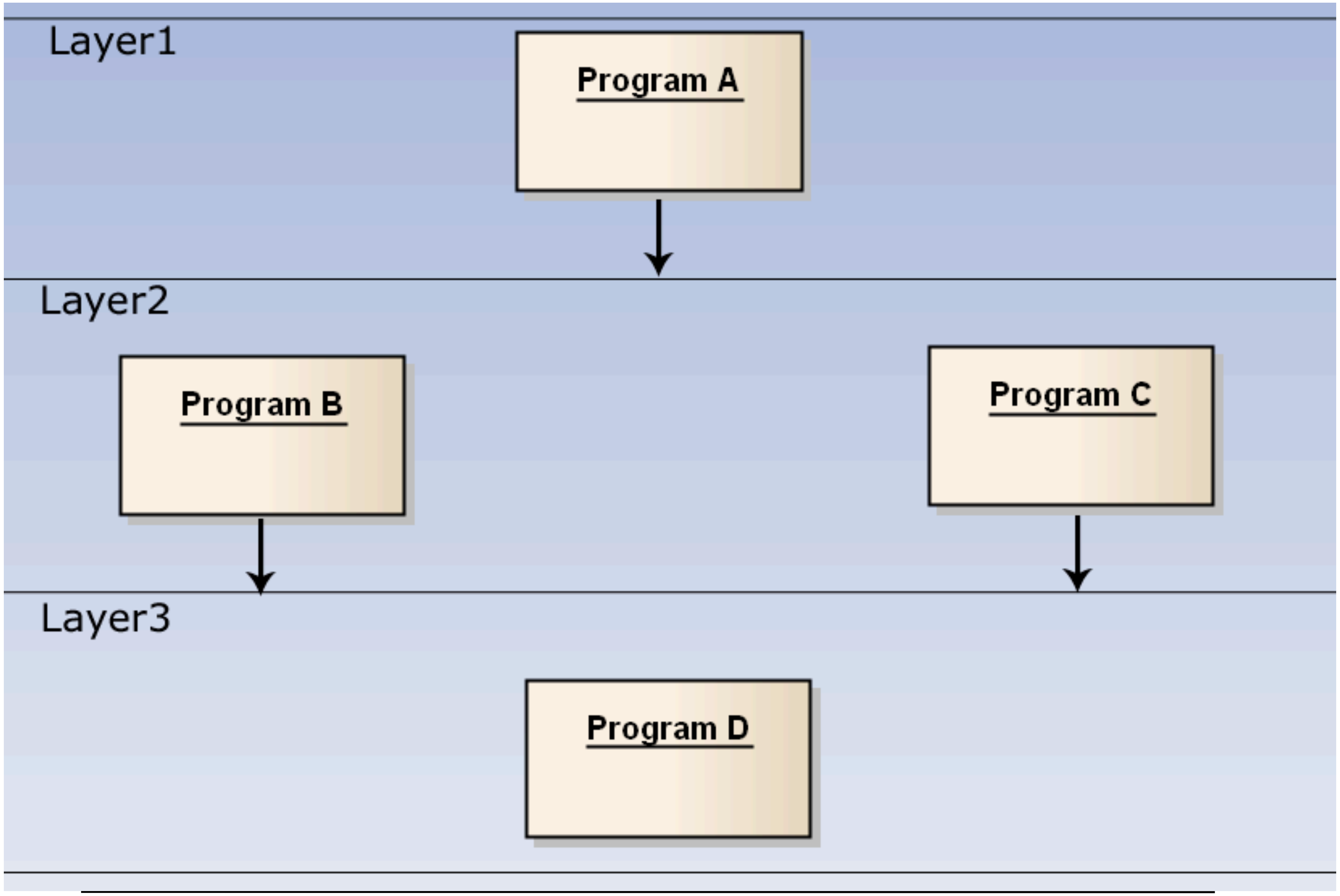
Layers and tiers (2)

- A complete decomposition of a given system comes from both layering and partitioning:
  - First, the system in divided into top level subsystems which are responsible for certain functionalities (*partitioning*);
  - Then, if necessary, each subsystem is organized into several layers, up to the definition of *simple enough* layers

Layered approach

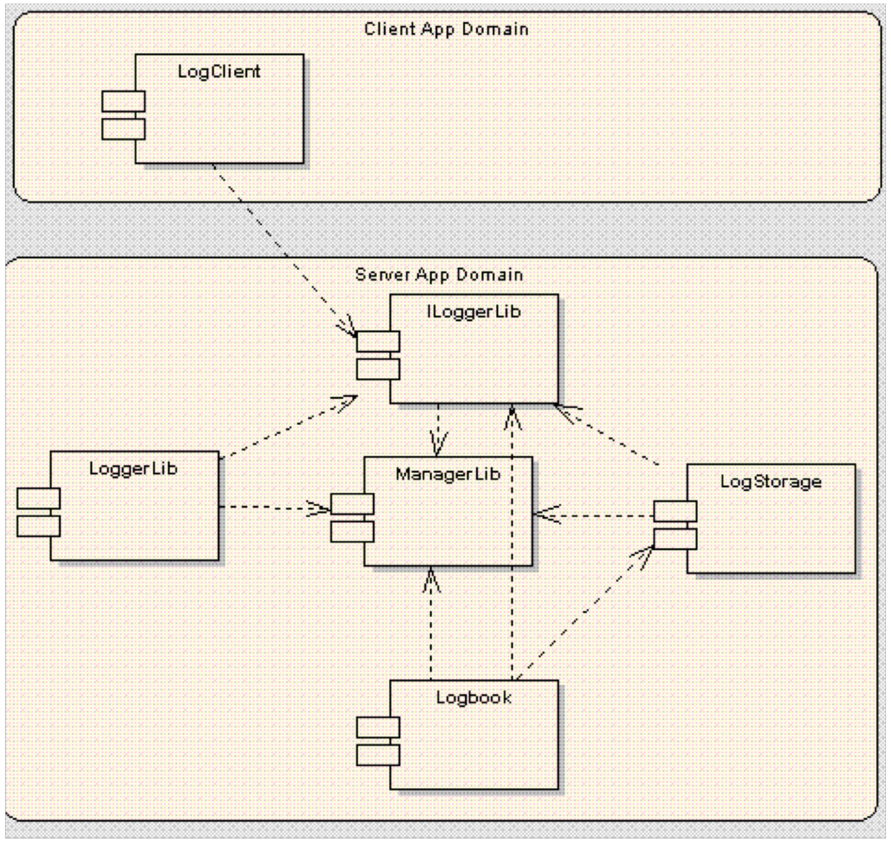
- An architecture which has a hierarchical structure, consisting of an ordered set of layers, is *layered*
- A **layer** is a set of subsystems which are able to provide related services, that can be realized by exploiting services from other layers
- A layer depends only from its lower layers (i.e., layers which are located at a lower level into the architecture)
  - A layer is only aware of lower layers

Example: layers of Virtual Machines



Layers: component diagram

- Connectors for layered systems are often procedure calls
- Each level implements a different virtual machine with a specific input language

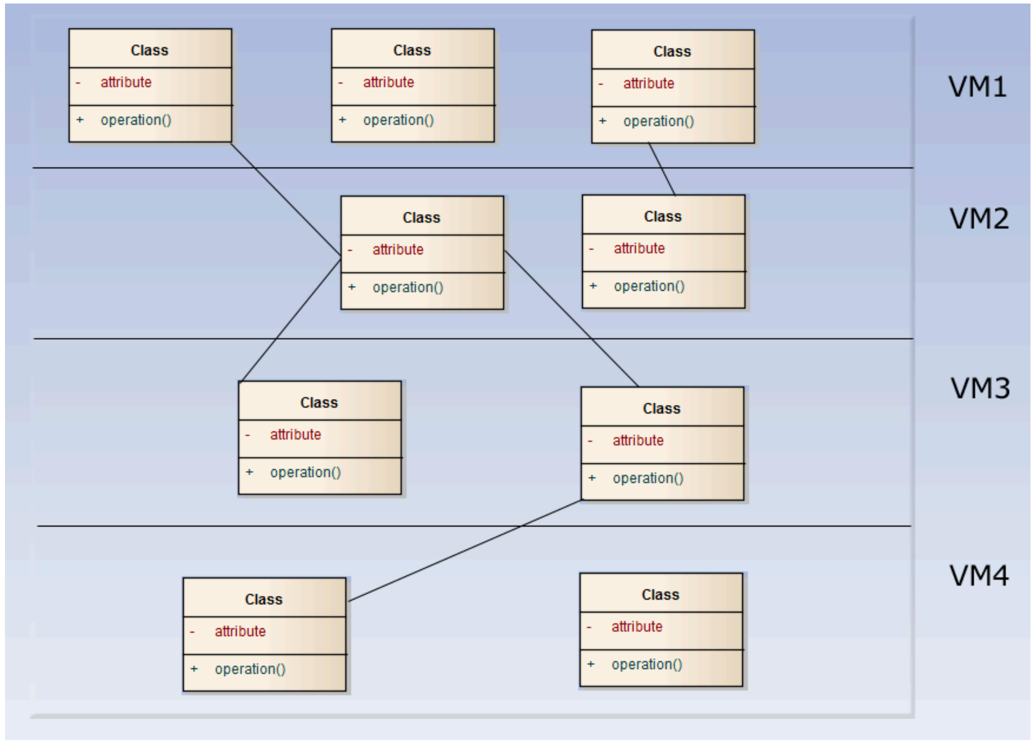


Closed vs. open layers

- **Closed Layers Architecture (CLA)**: the *i-th* layer can only have access to the layer *(i-1)-th*
- **Open Layers Architecture (OLA)**: the *i-th* layer can have access to all the underlying layers (i.e., the layers lower than *i*)

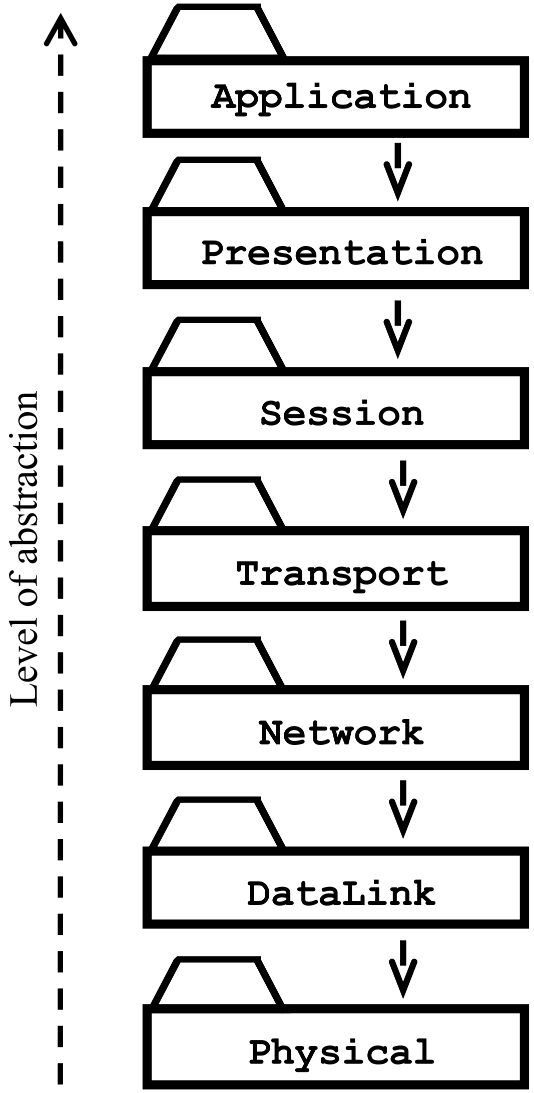
Closed Layers Architecture

- The *i-th* layer can only invoke the services and the operations which are provided by the layer *(i-1)-th*
- The main goals are the system maintainability and high portability (eg. Virtualization: each layer *i* defines a Virtual Machine - VM*i*)



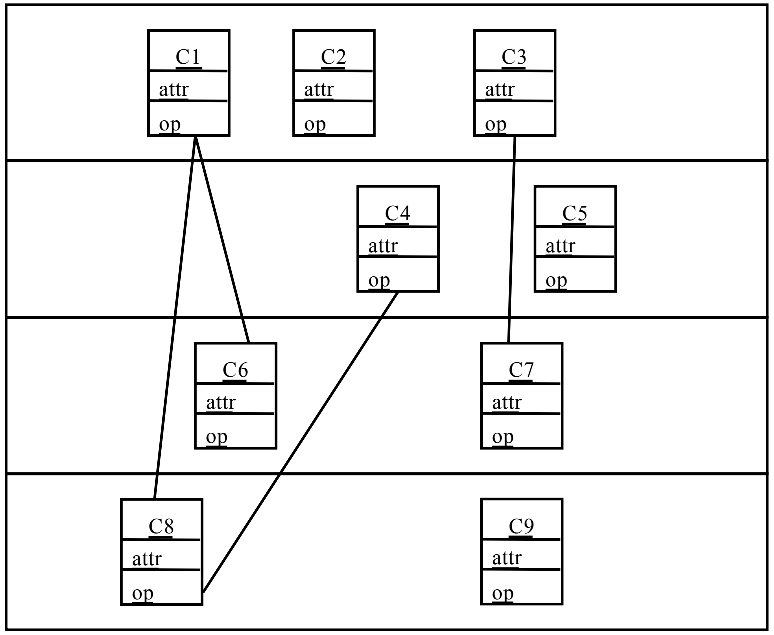
Example of CLA: ISO/OSI stack

- The ISO/OSI reference model defines 7 network layers, characterized by an increasing level of abstraction
- Eg: the Internet Protocol (IP) defines a VM at the network level able to identify net hosts and transmit single packets from host to host

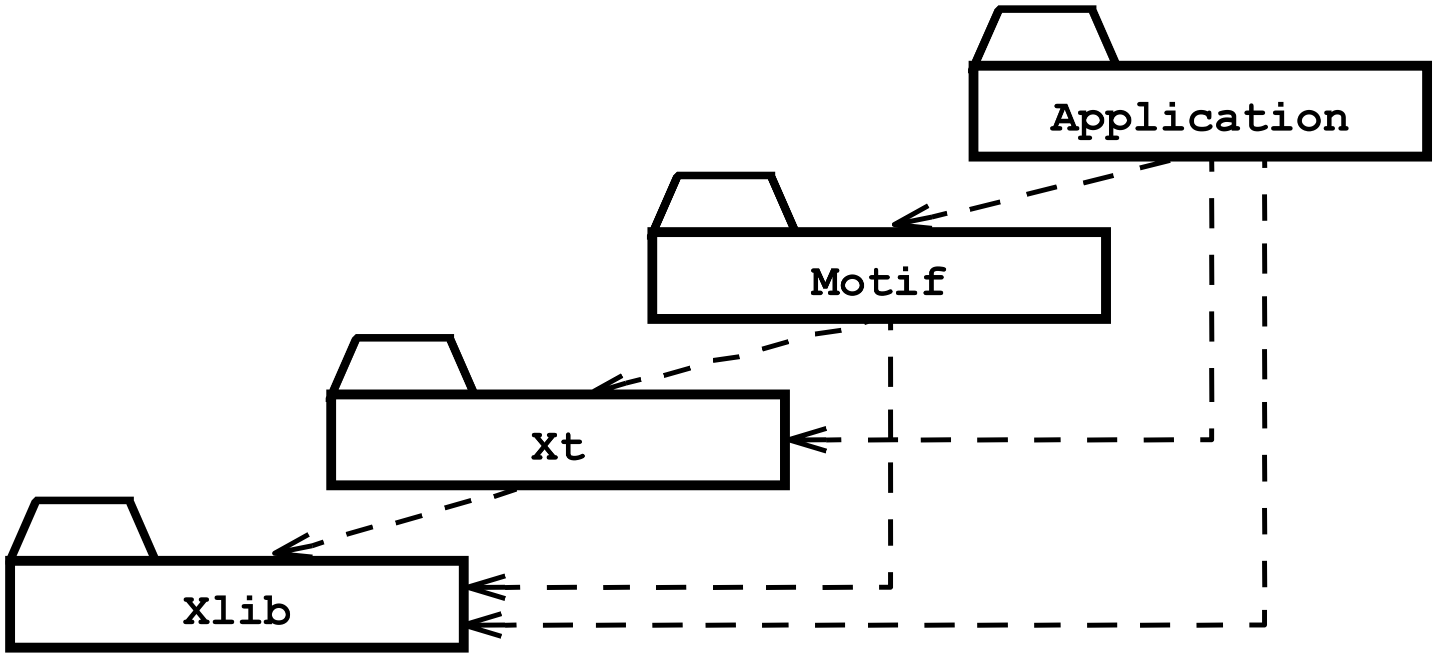


Open Layers Architecture (OLA)

- The *i-th* layer can invoke the services and the operations which are provided by all the lower layers (the layers lower than *i*)
- The main goals are the execution time and efficiency



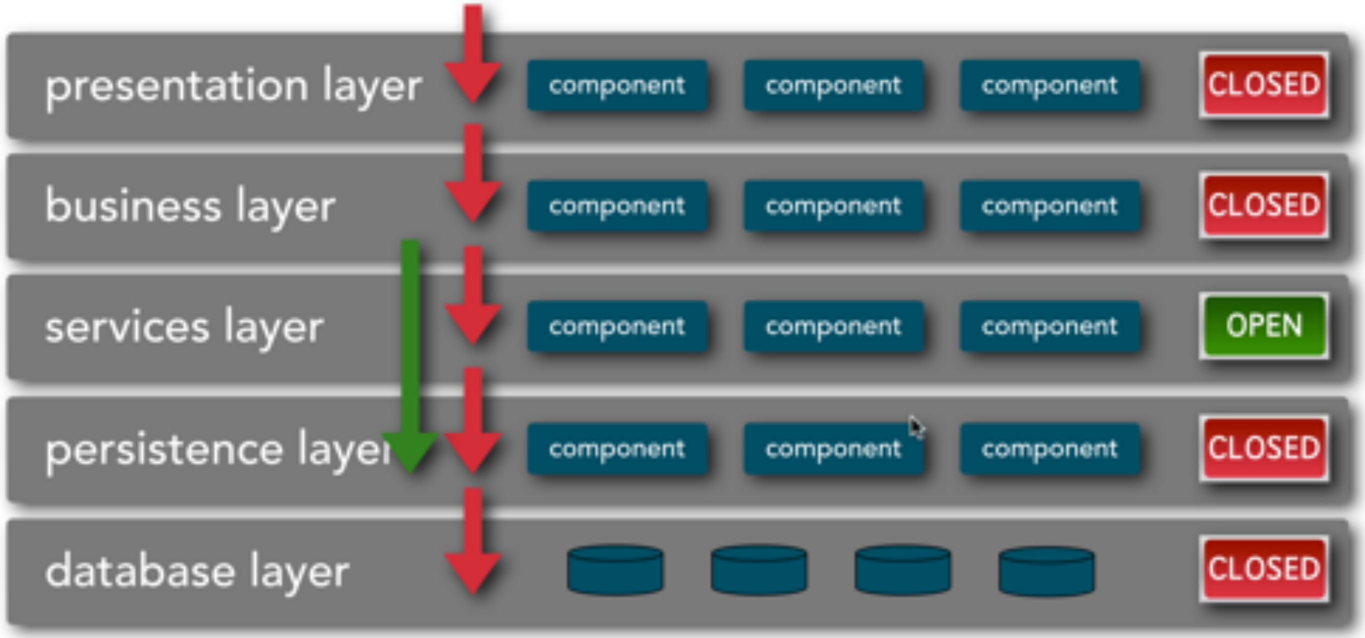
Example of OLA: OSF/Motif



- **`Xlib:`** provides low-level drawing facilities;
- **`Xt:`** provides basic user interface widget management;
- **`Motif:`** provides a large number of sophisticated widgets;
- **`Application:`** can access each layer independently.

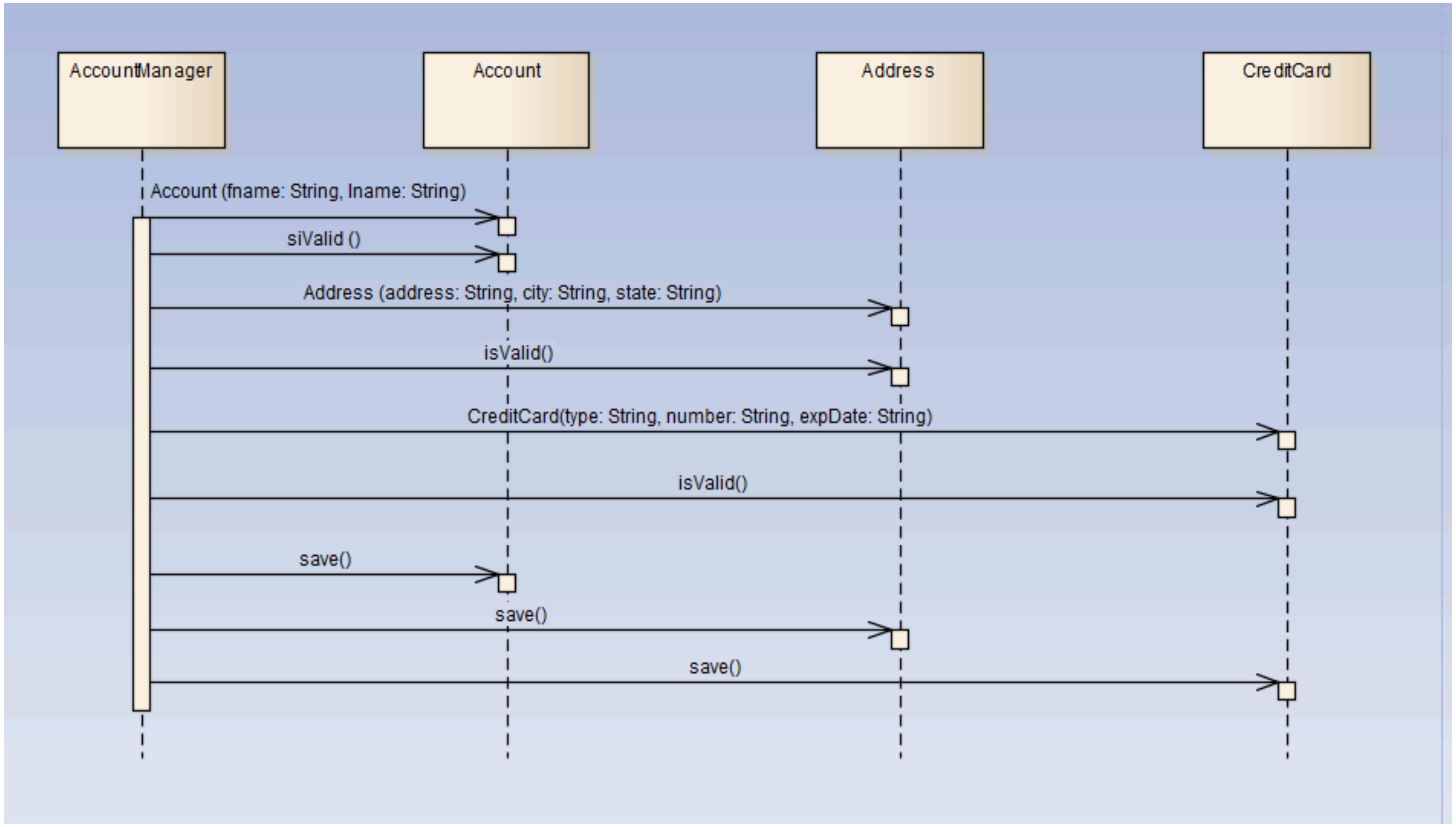
http://motif.ics.com

Mixed open-closed layers



https://www.oreilly.com/ideas/contrasting-architecture-patterns-with-design-patterns

Creating layers: Façade (1)



AccountManager
Account
Address
CreditCard
Account (fname: String, lname: String)
siValid ()
Address (address: String, city: String, state: String)
isValid()
CreditCard(type: String, number: String, expDate: String)
isValid()
save()
save()
save()

Figure 22.5 How a Client Would Normally Interact (Directly) with Subsystem Classes to Validate and Save the Customer Data

Creating layers: Façade (2)

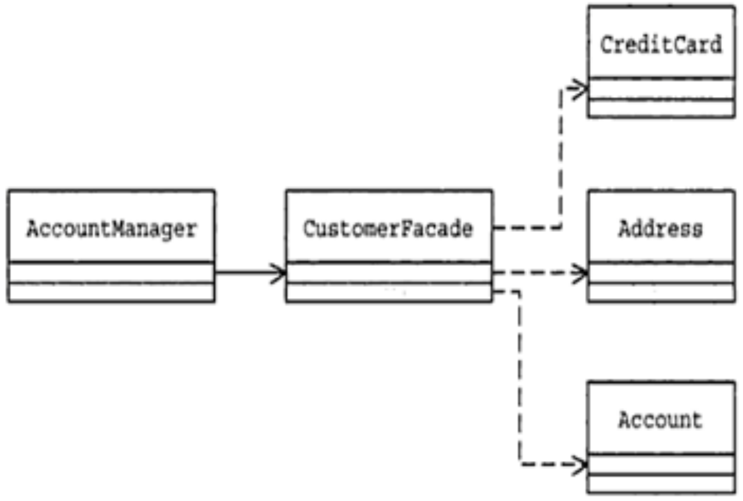
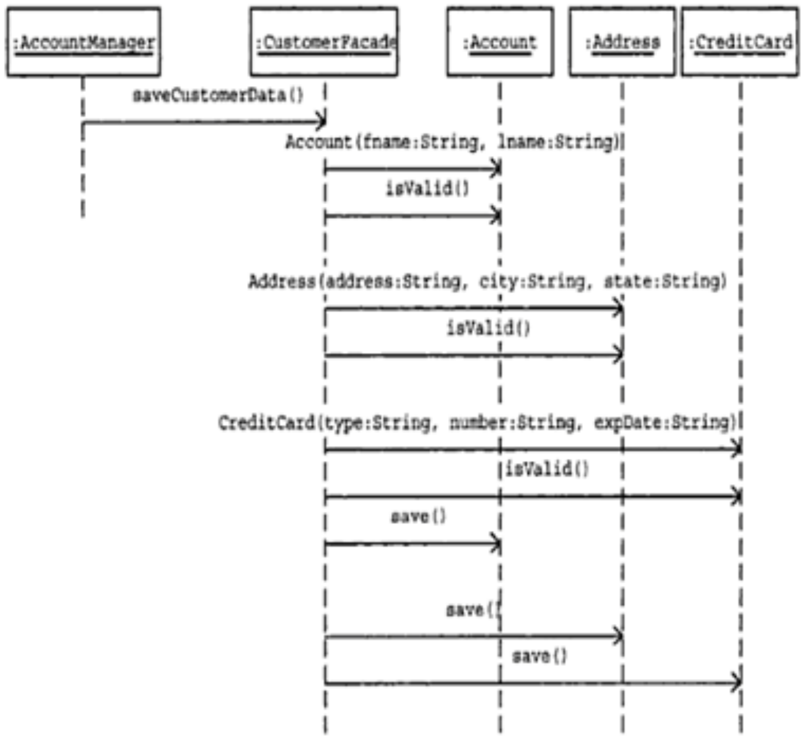


Figure 22.7 Class Association with the Façade Class in Place



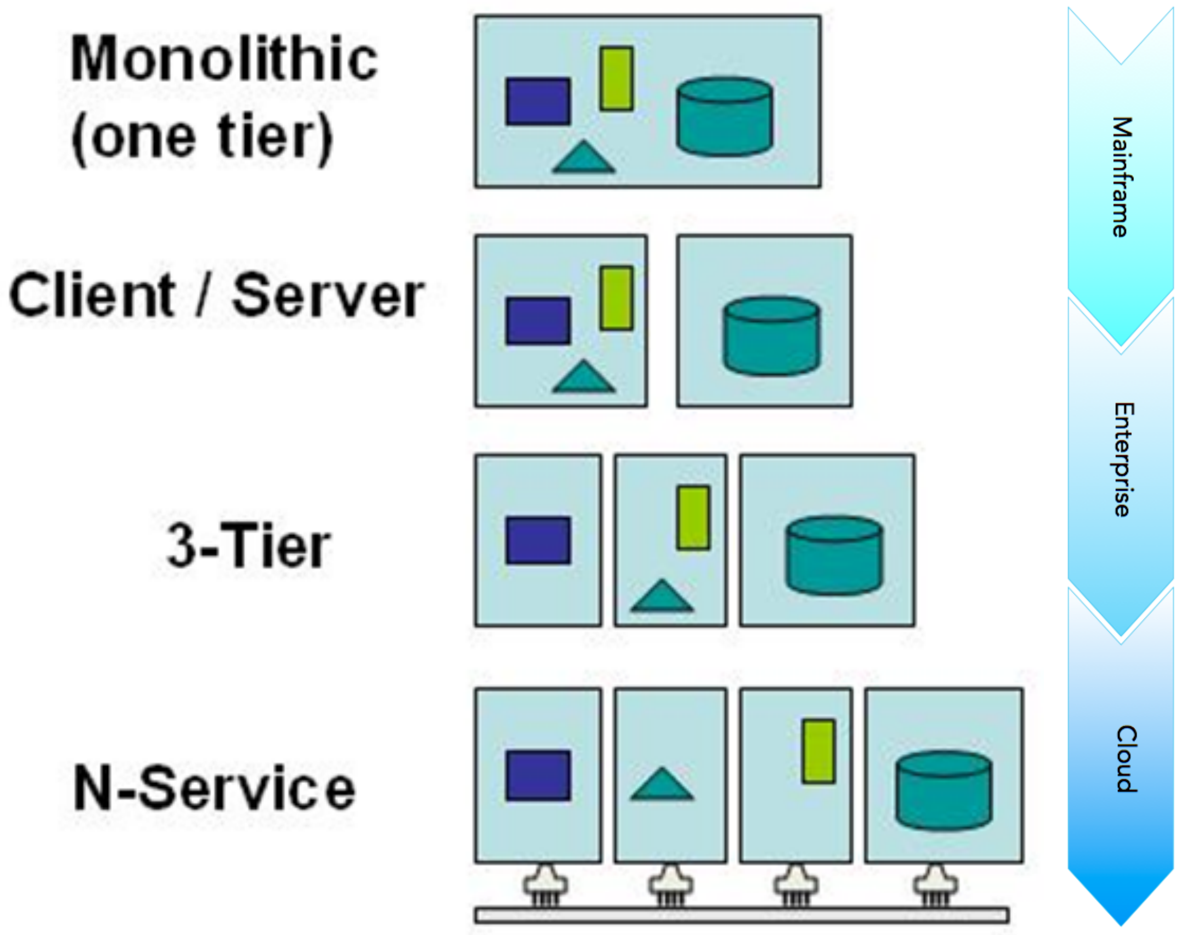
:AccountManager
:CustomerFacade
:Account
:Address
:CreditCard
saveCustomerData()
Account(fname:String, lname:String)
isValid()
Address(address:String, city:String, state:String)
isValid()
CreditCard(type:String, number:String, expDate:String)
isValid()
save()
save()
save()

Figure 22.8 In the Revised Design, Clients Interact with the Façade Instance to Interface with the Subsystem

Tiered approach

- Another approach to managing complexity consists of *partitioning* a system into sub-systems (*tiers*), which are responsible for a class of services maintained as independent modules
- These modules are separated from each other by physical boundaries:
  - machine boundaries
  - process boundaries
  - corporate boundaries

Tiers and computing evolution

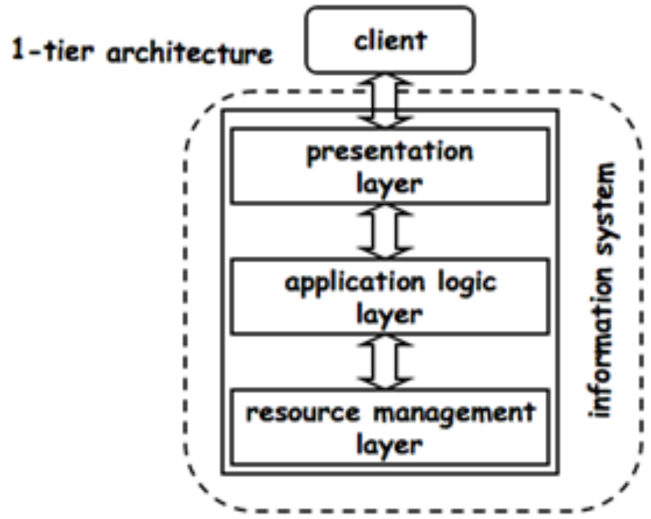


1-tier, 2-tiers, 3-tiers

- A **1-tier model (monolithic)** describes a single-tiered application in which the user interface and data access code are combined into a single program from a single platform
- A **2-tiers model (client/server)** represents a split monolithic model composed by a *client* tier that interacts directly with a *server* tier
- A **3-tiers model (n-tiers)** is a client/server model in which the presentation, the application processing, and the data management are logically (ad often physically) separate processes

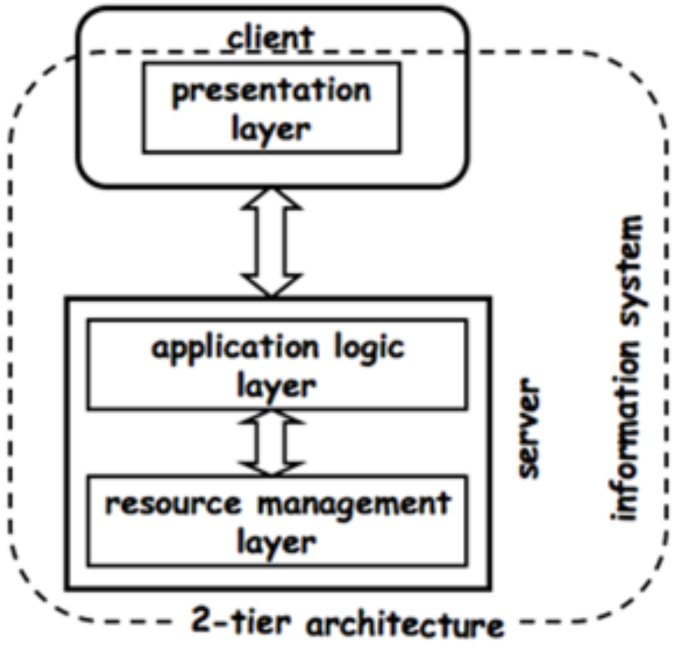
1-tier model: Monolithic

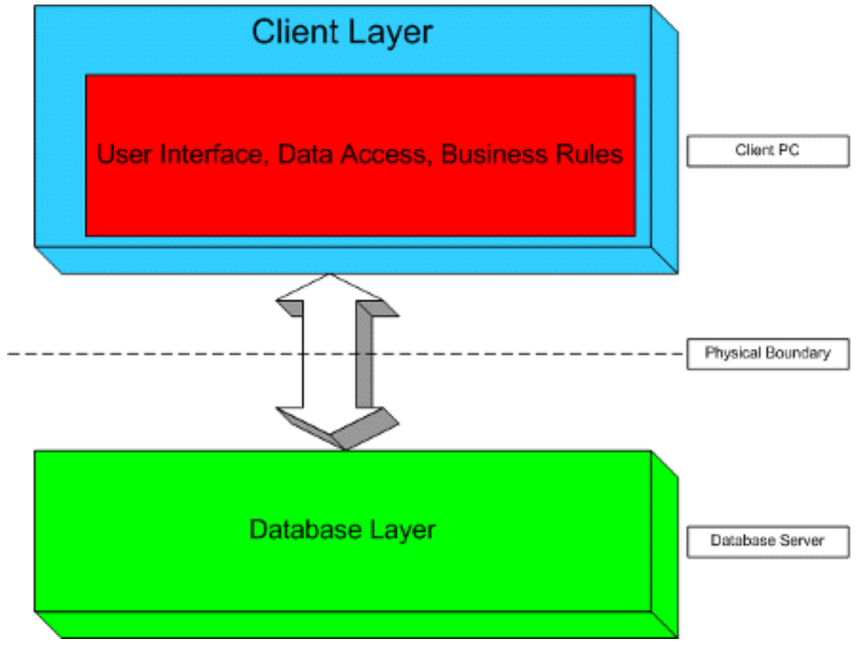
- The presentation, application logic and resource manager are built as a monolithic entity (no modularity)
- Users/programs access the system through "dumb" terminals managed by an *information system*
- Typical model for mainframes



2-tiers model: Client/Server

- This model allows to move the presentation tier to the client (*fat client*)
- It introduces the concept of API, an interface to invoke the system from the outside



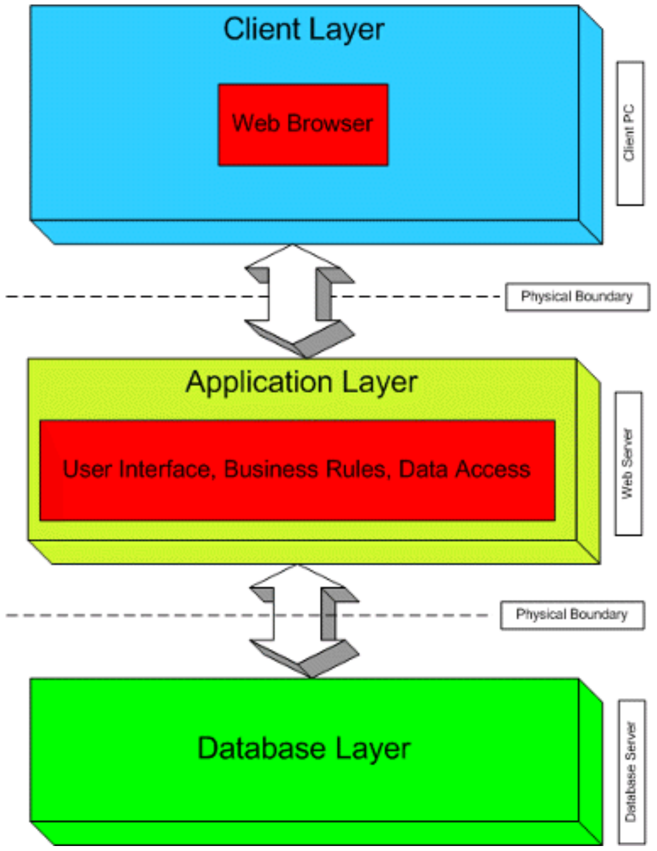


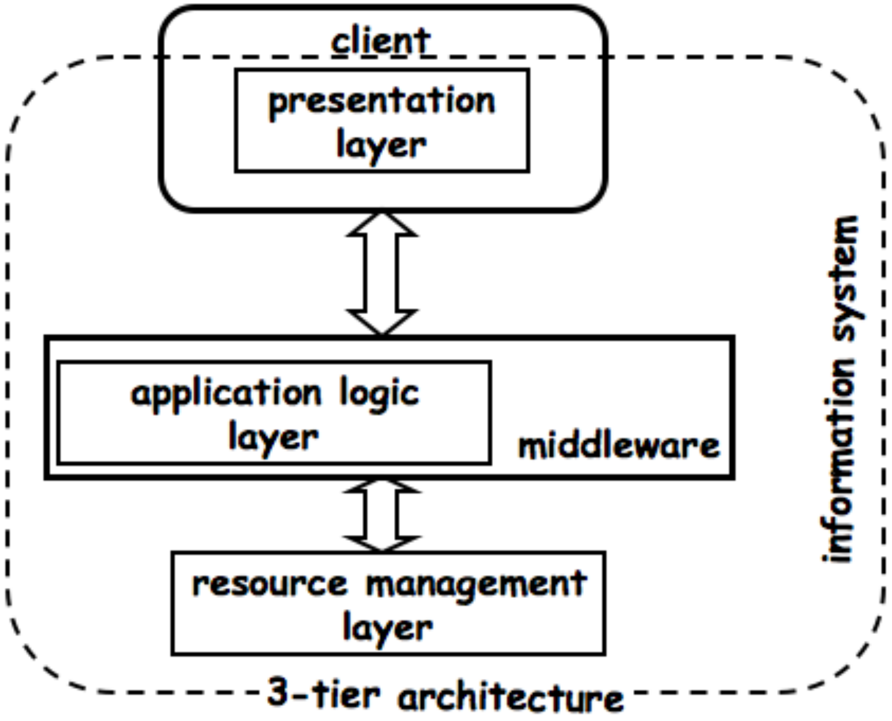
3-tiers model: Multitiered (1)

- In this approach, the user interface runs on a desktop PC or workstation and uses a standard graphical user interface (*thin client*)
- The functional process logic may consist of one or more separate modules running on a workstation or application server
- An RDBMS on a database server or mainframe contains the computer data storage logic
- This is a specialization (the most widespread) of the n-tiers model

3-tiers model: Multitiered (2)

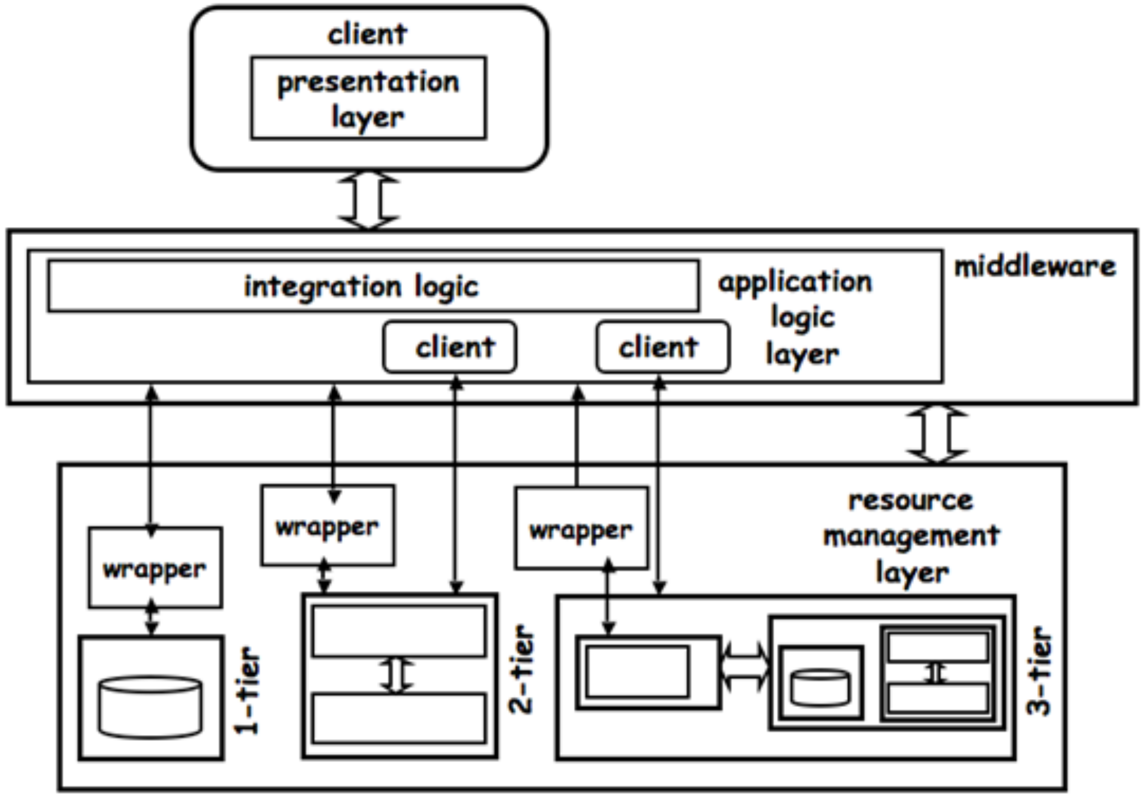
- Concept of *middleware,* that introduces an additional tier of business logic encompassing all the underlying systems



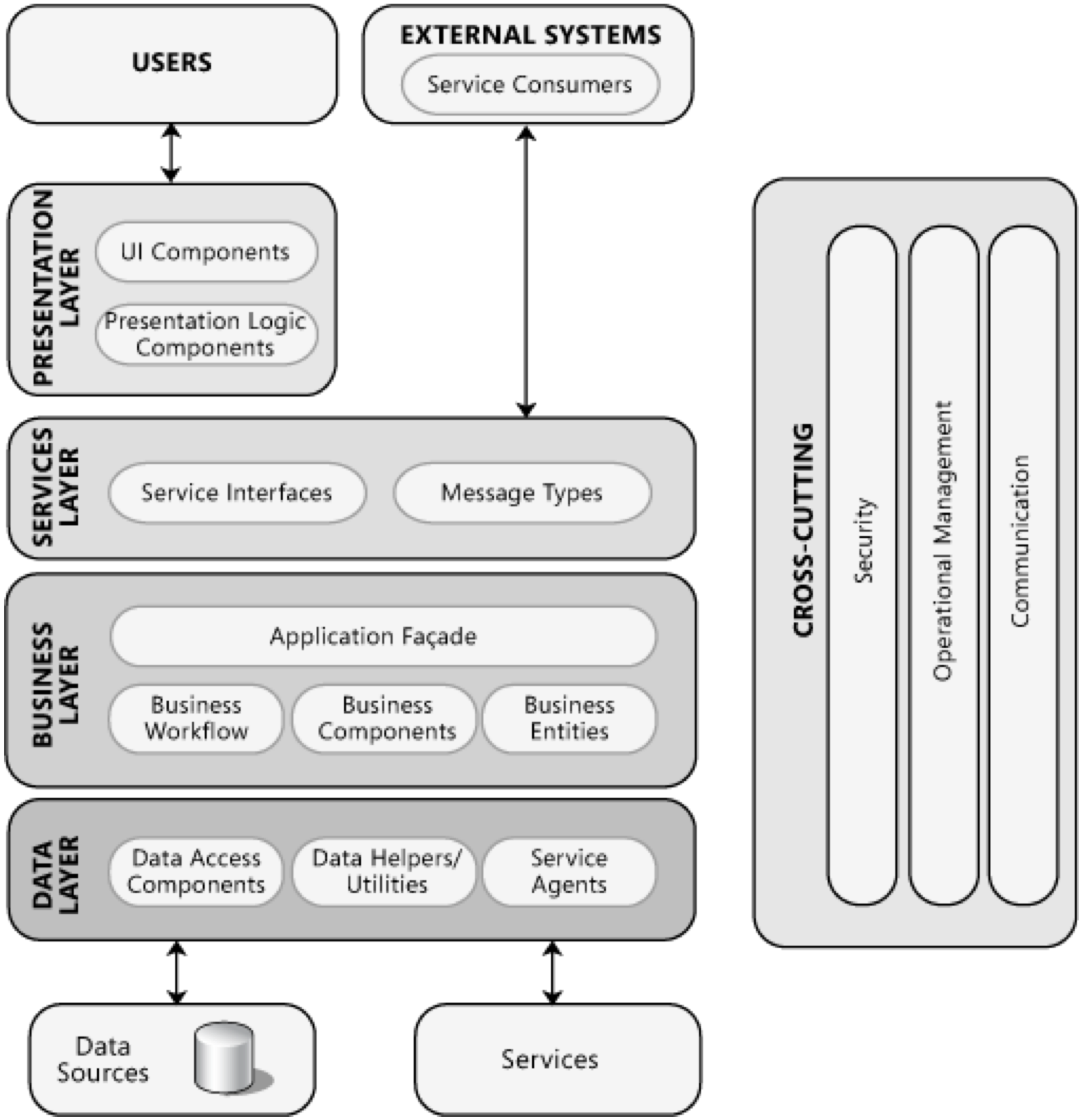


Middleware as a glue

- Middleware systems also enable the integration of systems built using other architecture models



Do not confuse layers and tiers!



msdn.microsoft.com/en-us/library/ee658124.aspx

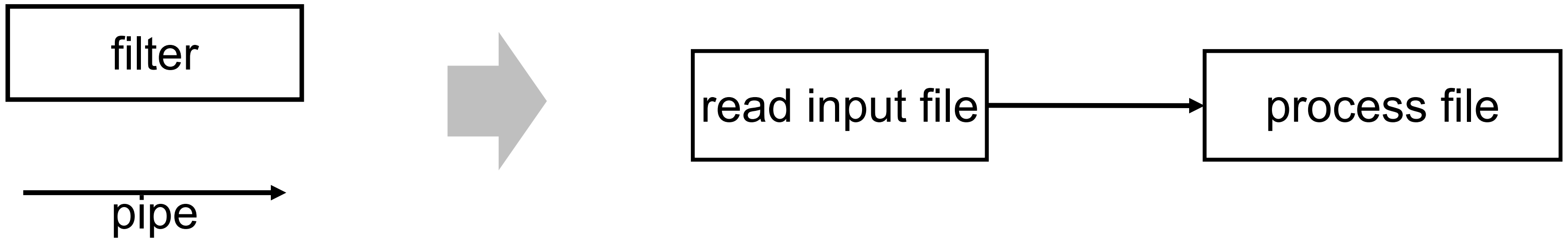
Basic architectural styles

- Several architectural styles have been defined in the literature of software engineering
- They can be used as the basis for configuring software architectures
- The basic styles which follow include:
  - Pipe & Filter
  - Shared-data
  - Client-Server
  - Peer-To-Peer

Pipe & Filter style

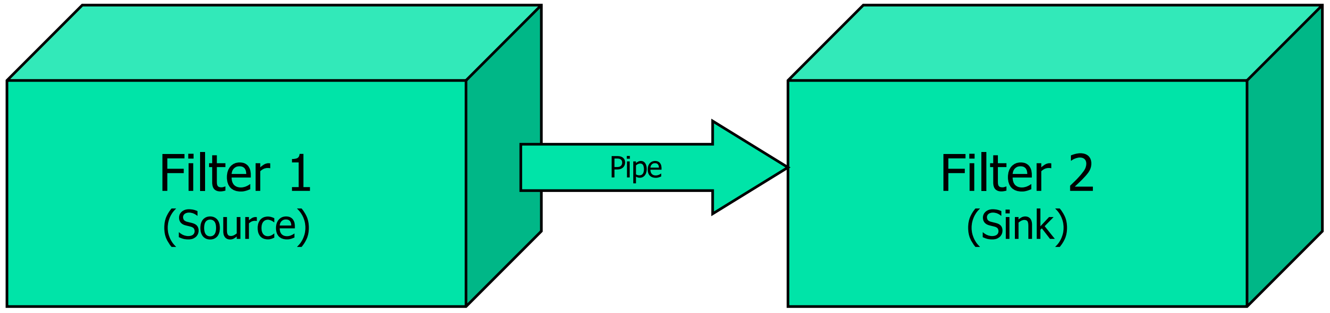
Pipe & Filter: Overview

- A component reads streams of data as input and produces streams of data as output
- Suitable for applications that require a sequence of computations to be performed on data
- This architectural style focuses on the dynamics (interaction) rather than the structure

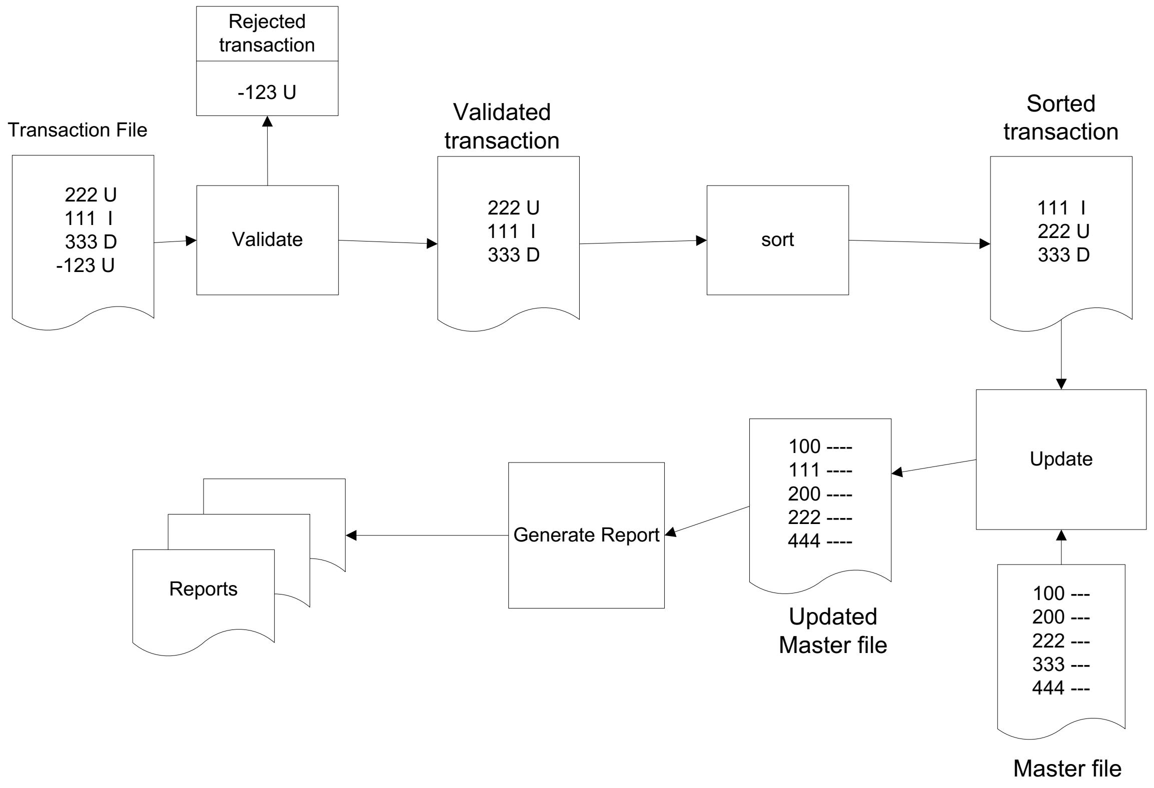


Pipe & Filter: Overview

- Filter 1 is a **Source**:
  - may only send data to Filter 2
  - may not receive data
- Filter 2 is a **Sink**:
  - may only receive data from Filter 1
  - may not send data



Pipe & Filter example: Batch Sequential



- A transformation subsystem or module cannot start its process until the previous module completes its computation

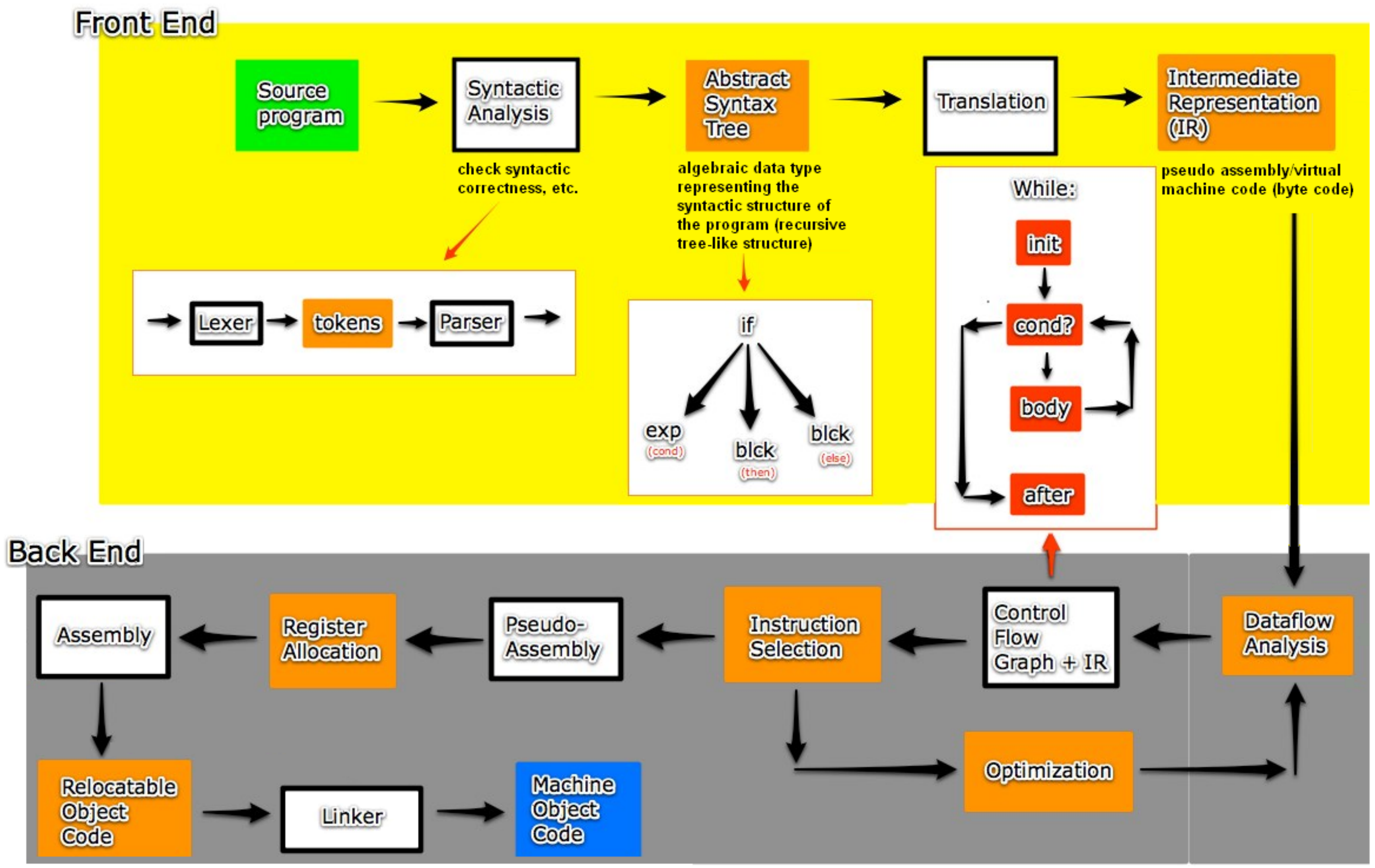
Pipe & Filter example: Unix Shell

- Unix shell command line processing of the pipe symbol "`|`":
  - the output from the command to the left of the symbol "`|`", is to be sent as input to the command to the right of the pipe
  - this mechanism got rid of specifying a file as **`std output`** of a command and then specifying that same file as the **`std input`** to another command, including possibly removing this intermediate file afterwards
- Example: counting occurrences in a mailing list file called "*swarch.txt*":

```
# cat swarch.txt | grep studio | wc
```

- Does this break any assumption of the "pure" Pipe & Filter style?

Pipe & Filter example: Compiler



Pipe: CRC

Class Pipe	**Collaborators**
Responsibilities •Connector: connects a filter with another filter •Flows data from its input end to its output end •It might be passive (data buffer) or active (object)	• Filter

Filter: CRC

Class Filter
Subclasses: Source, Sink

Collaborators

Responsibilities

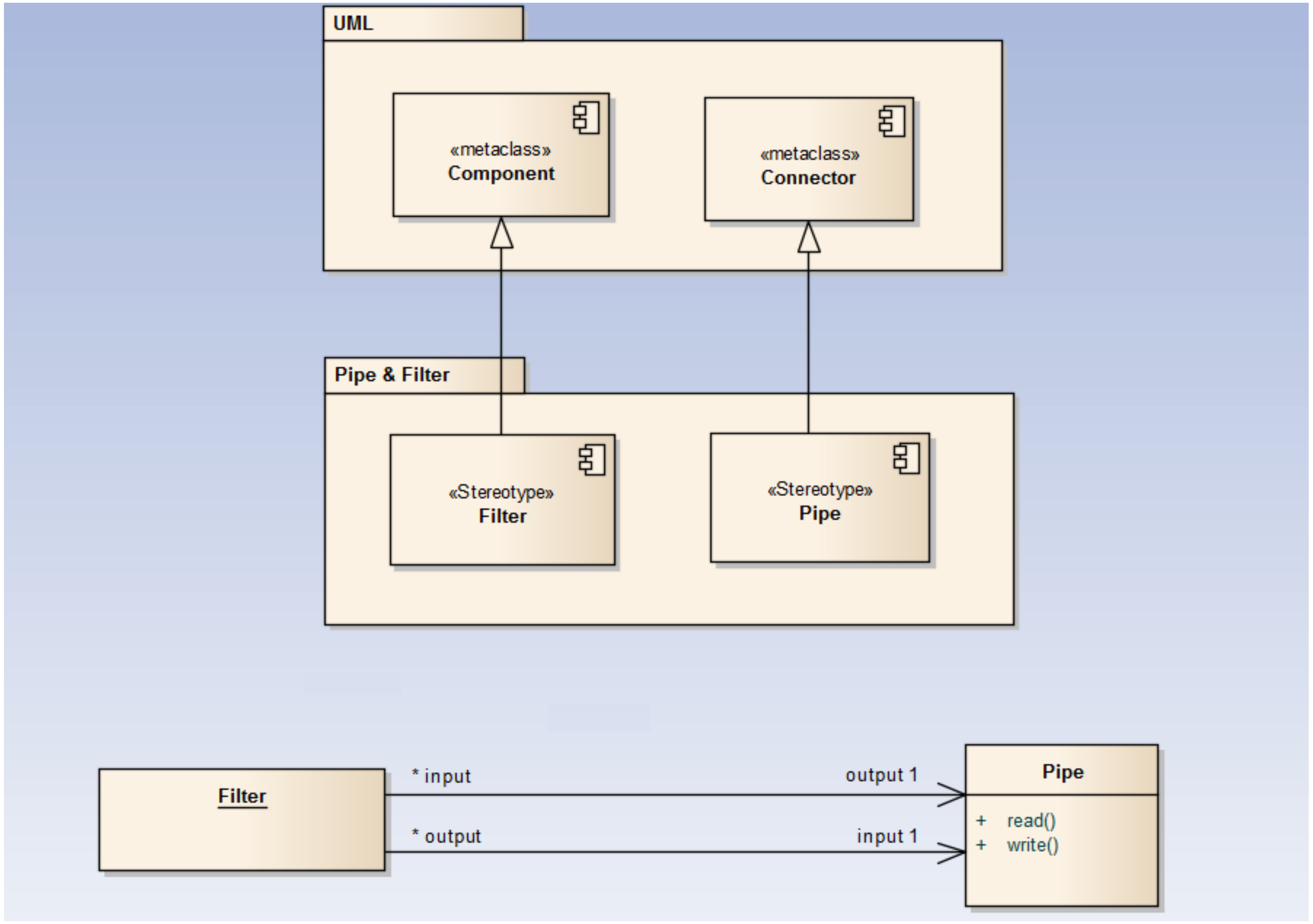
- Component (computes a transformation function)
- Transforms data from its input port to its output port
- *Sources* are special filters without inputs
- *Sinks* are special filters without outputs

- Pipe

Pipe & Filter: Architecture

- Components:
  - **Filter**
    - reads data from its input stream, processes it, and writes it over a pipe for next filter to process
    - only knows its connected pipes, does not know what are at the other end of the pipe
    - Is independent from other filters
    - Its output can begin before knowing all input
- Connectors:
  - **Pipe**
    - Is stateless and used to move streams of data between filters

Pipe & Filter: Metamodel and Model



Pipe & Filter: Data Flow types

- There are three ways to make the data flow:
  - **Push only (Write only)**
    - A data source may pushes data in a downstream
    - A filter may push data in a downstream
  - **Pull only (Read only)**
    - A data sink may pull data from an upstream
    - A filter may pull data from an upstream
  - **Pull/Push (Read/Write)**
    - A filter may pull data from an upstream and push transformed data in a downstream

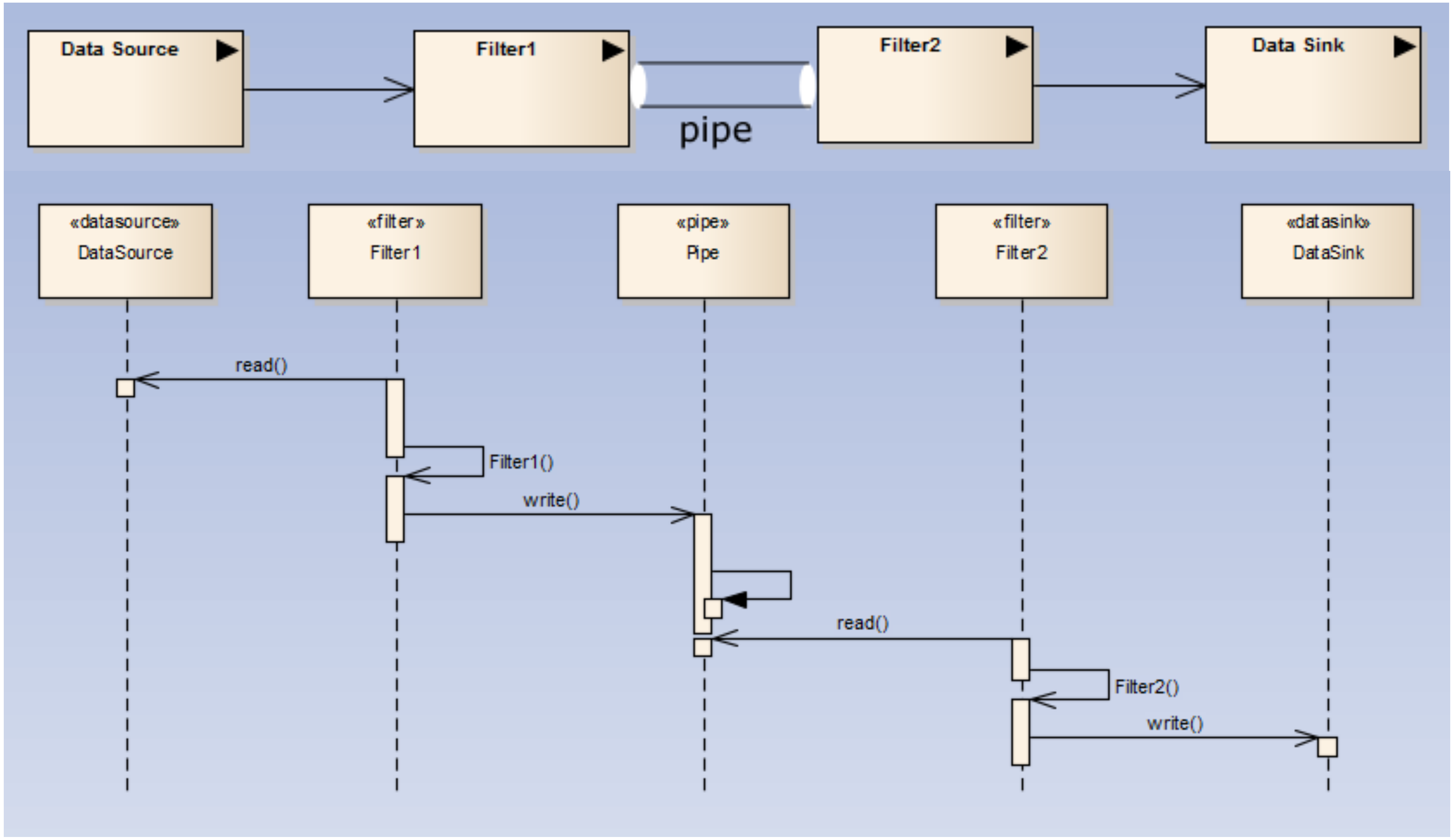
Pipe & Filter: active or passive filters (1)

- **Active filter**:
  - An active filter pulls in data and push out the transformed data
  - It works with a passive pipe which provides read/write mechanisms for pulling and pushing
  - Pull/Push data flow type
- Examples:
  - The UNIX Pipe&Filter mechanism
  - The **`PipedWriter`** and **`PipedReader`** classes in Java are also passive pipes that active filters must use to drive the data stream forward

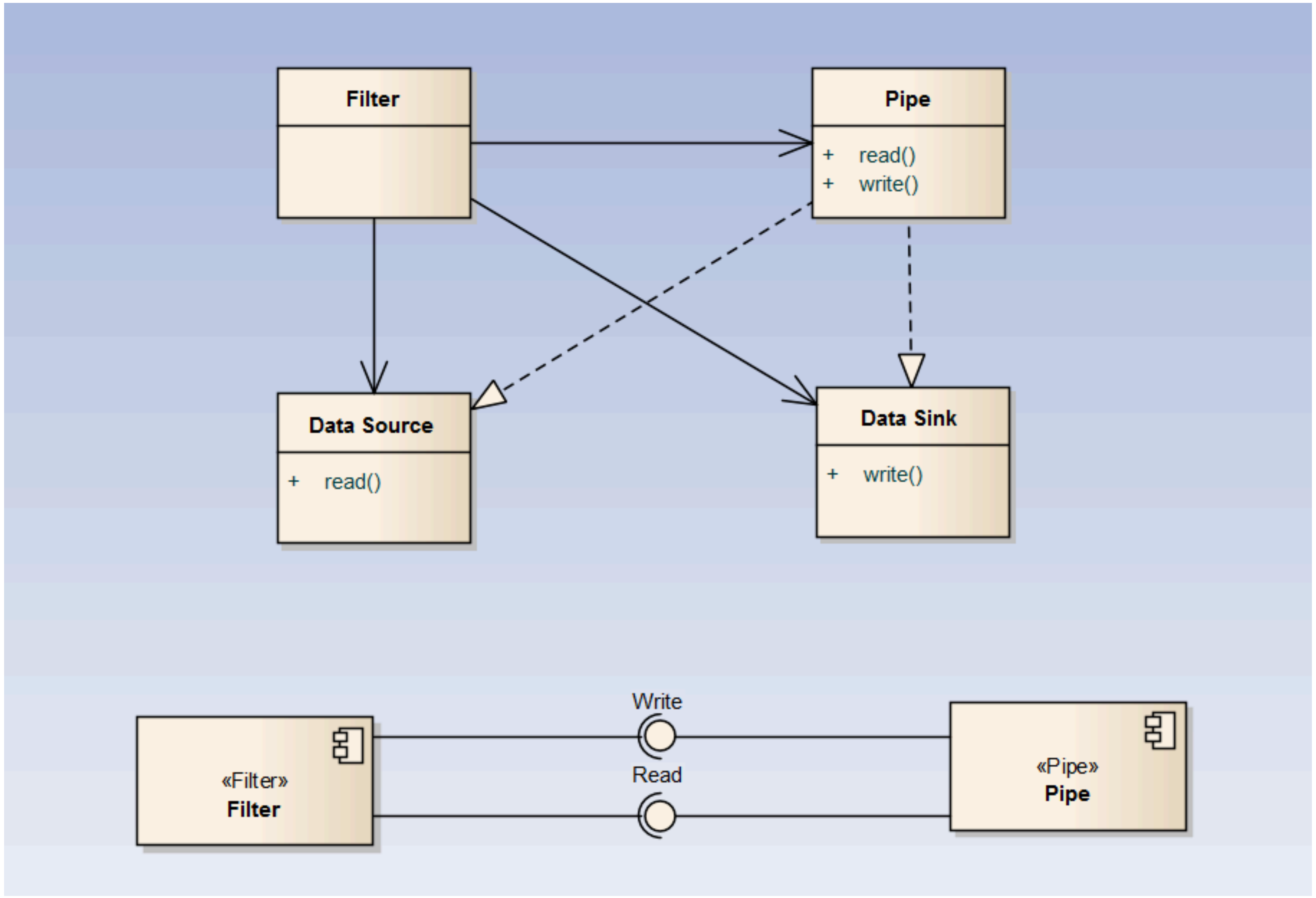
Pipe & Filter: active or passive filters (2)

- **Passive filter**:
  - Lets connected pipes to push data in and pull data out
  - It works with active pipes that pull data out from a filter and push data into the next filter
  - The filter must provide the read/write mechanisms in this case
- Examples:
  - Very similar to the data flow hardware architecture

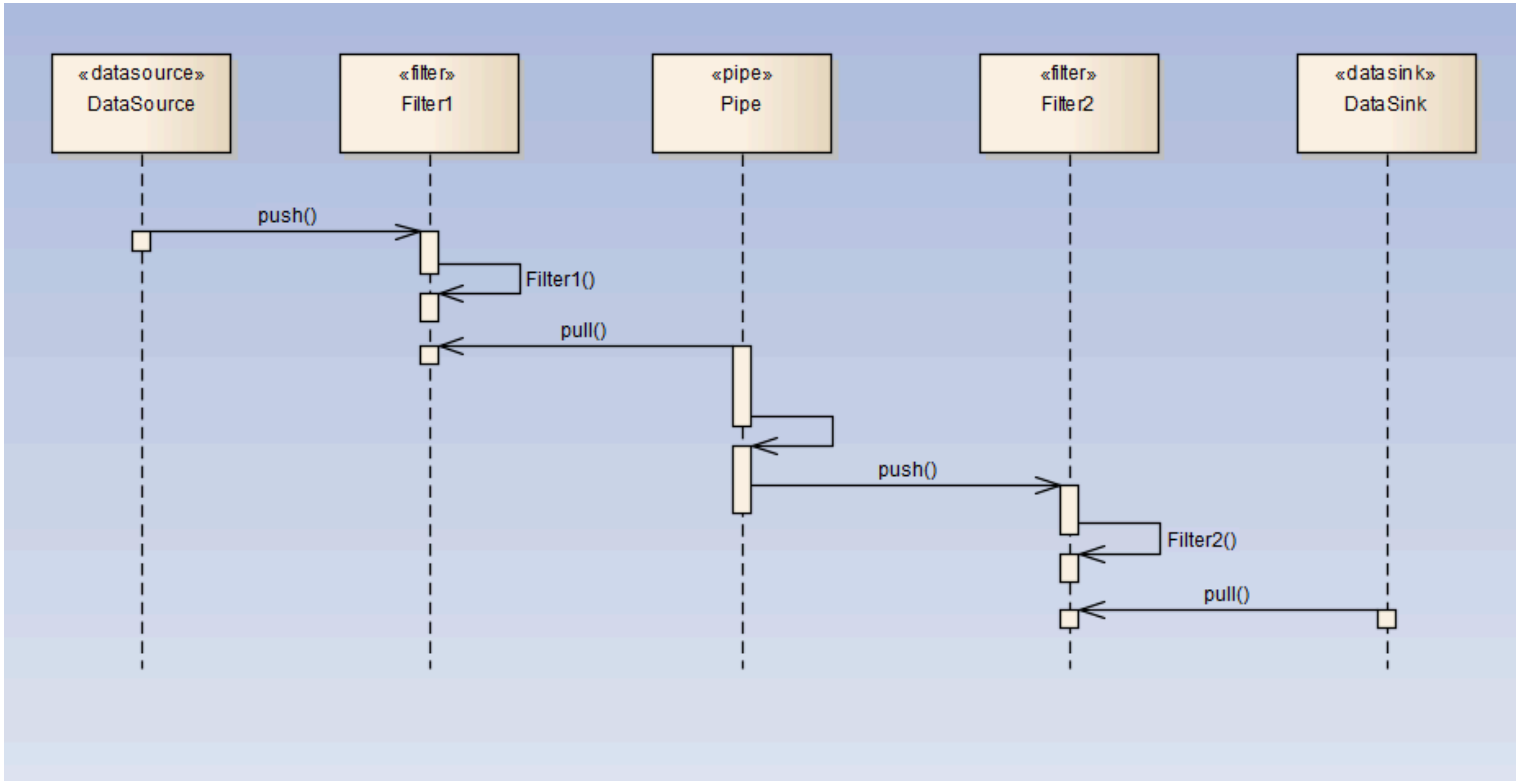
Active Filter: Sequence Diagram



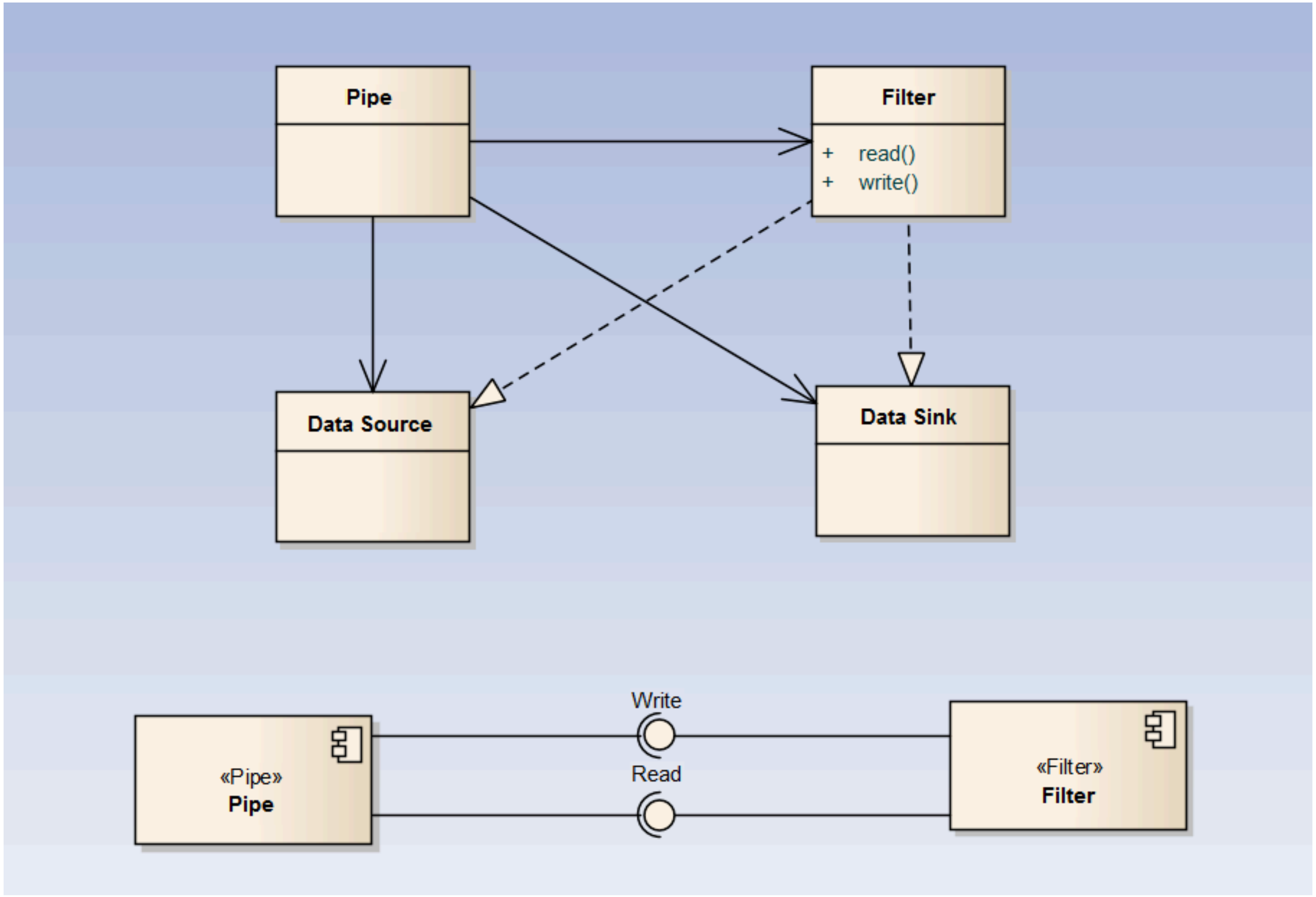
Active Filter: Class and Component Diagram



Passive Filter: Sequence Diagram

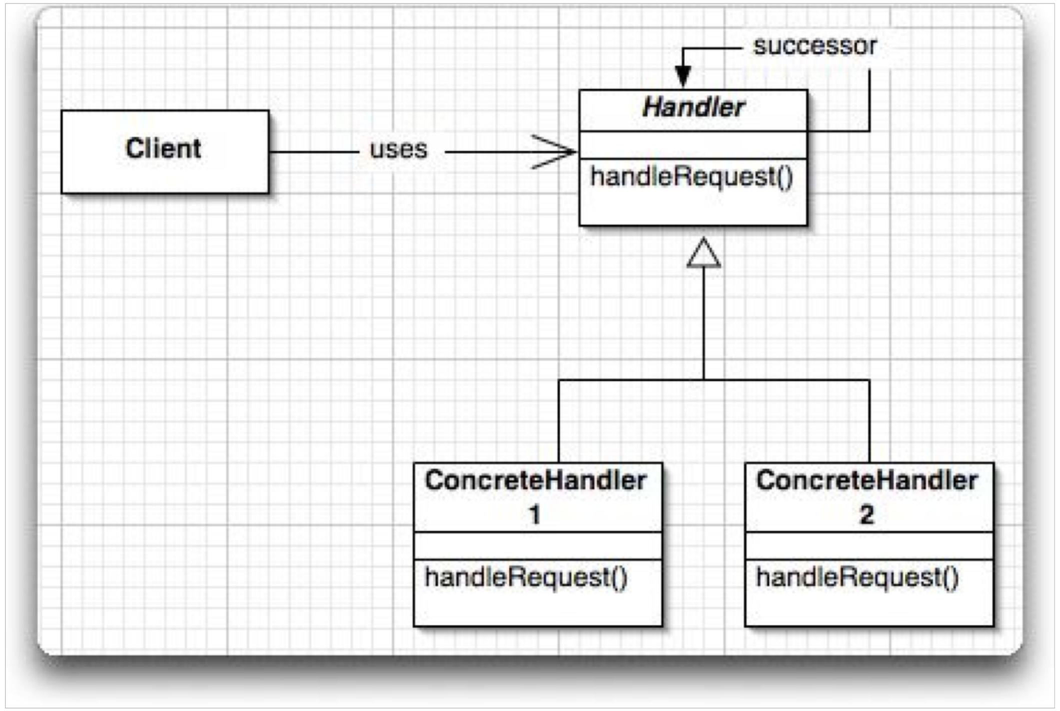


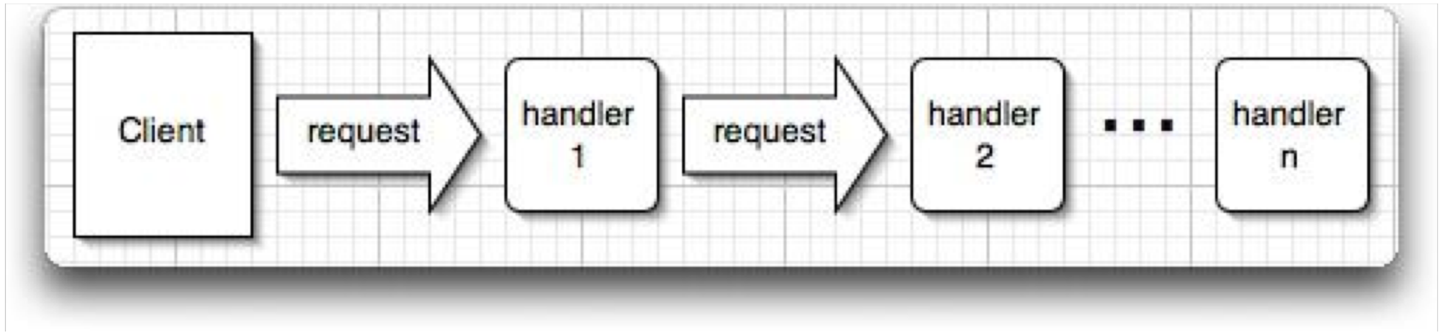
Passive Filter:



Pipe&filter and Chain of Responsibility

- The style Pipe&filter is inspired by the GoF design pattern Chain of Responsibility





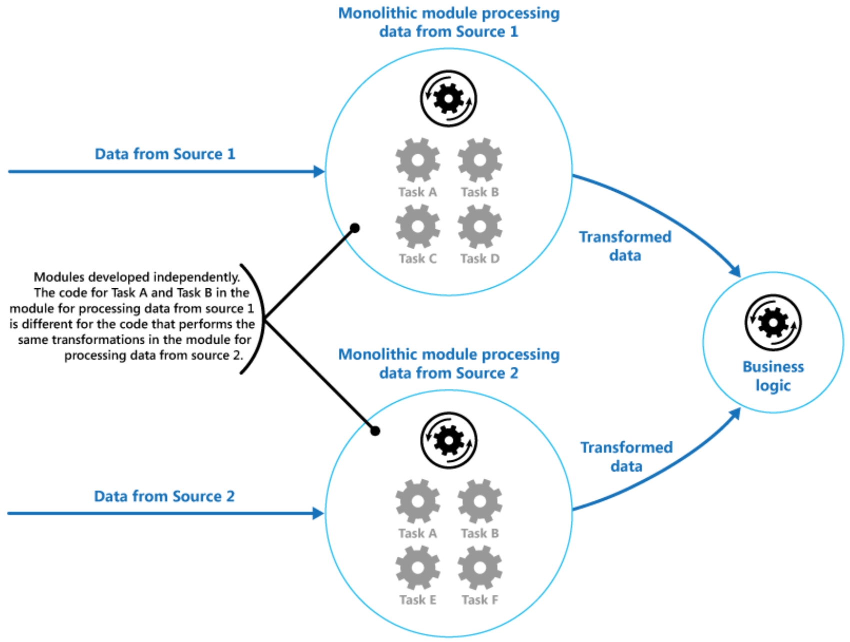
Pipe & Filter: Pro & Cons (1)

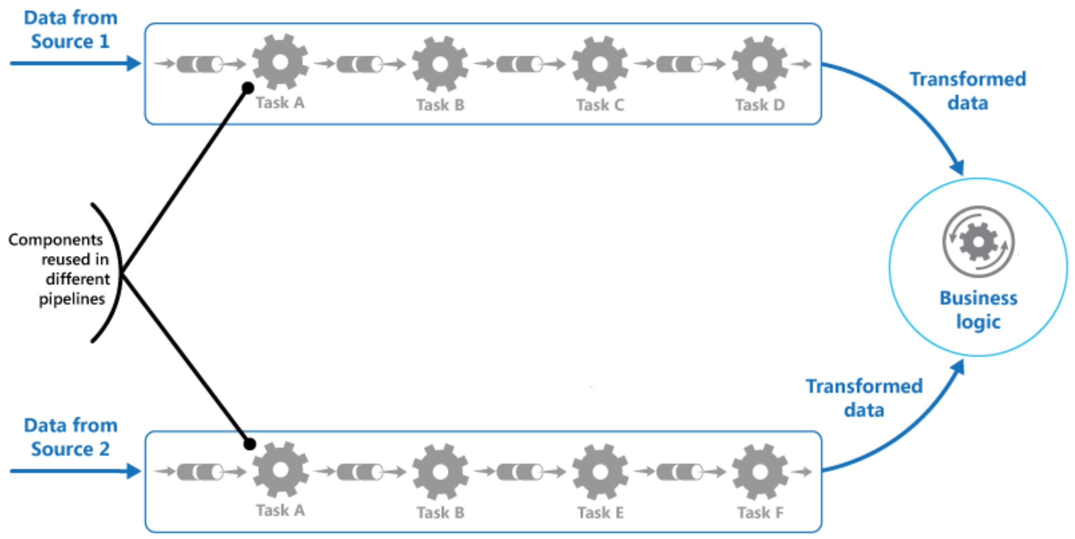
- Benefits:
  - Filters are self containing processing services that perform a specific function thus the style is cohesive
  - Filters communicate (pass data most of the time) through pipes only, thus the style results in low coupling
  - They naturally support concurrent execution: each filter can be implemented as a separate task and potentially executed in parallel with other filters

Pipe & Filter: Pro & Cons (2)

- Pitfalls:
  - The architecture is static (no dynamic reconfiguration)
  - Filter processes which send streams of data over pipes is a solution that fits well with heavy batch processing, but may not do well with any kind of user-interaction
  - Anything that requires quick and short error processing is still restricted to sending data through the pipes, possibly making it difficult to interactively react to error events

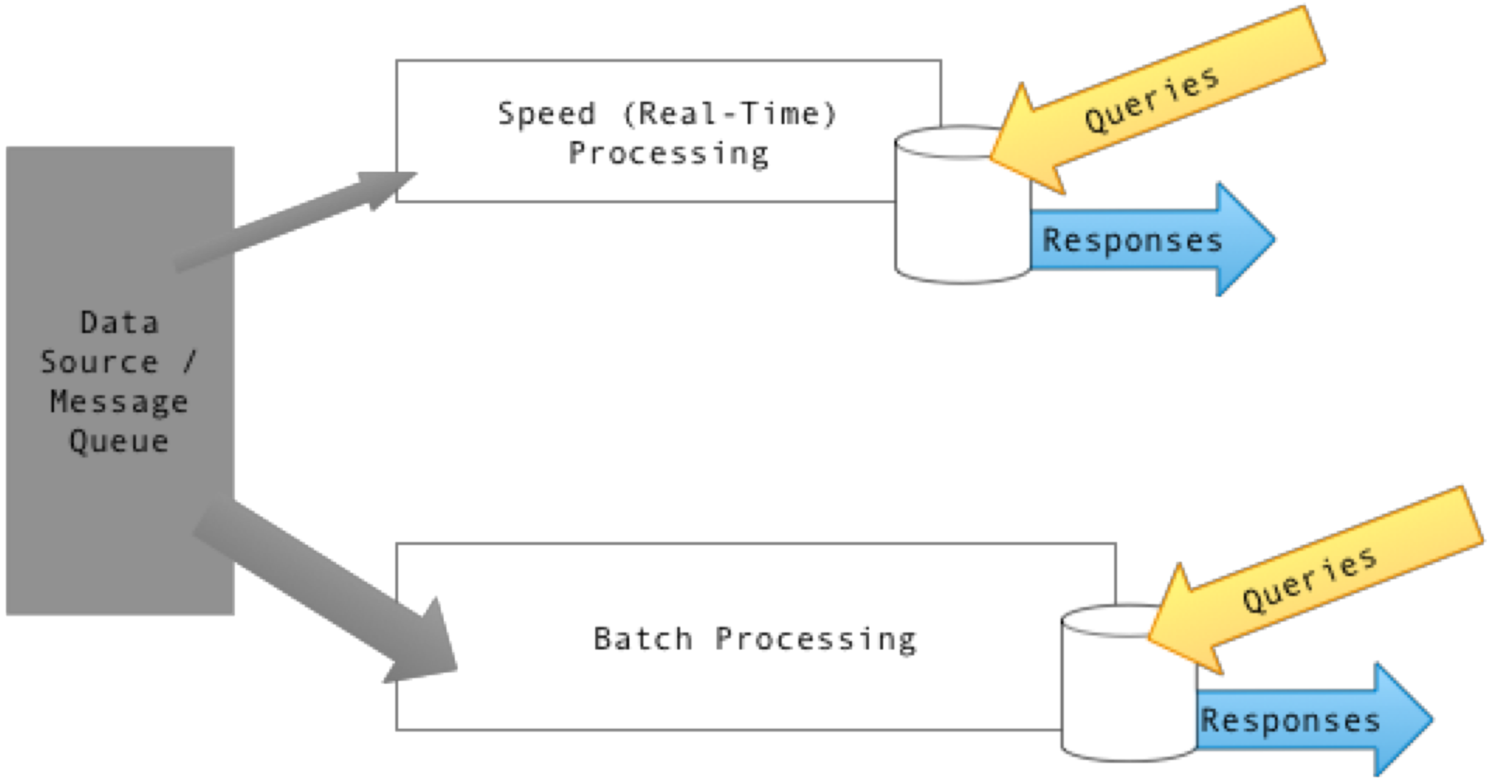
Pipe & filter


Monolithic module processing data from Source 1
Data from Source 1
Task A
Task B
Task C
Task D
Transformed data
Modules developed independently. The code for Task A and Task B in the module for processing data from source 1 is different for the code that performs the same transformations in the module for processing data from source 2.
Monolithic module processing data from Source 2
Data from Source 2
Task A
Task B
Task E
Task F
Transformed data
Business logic



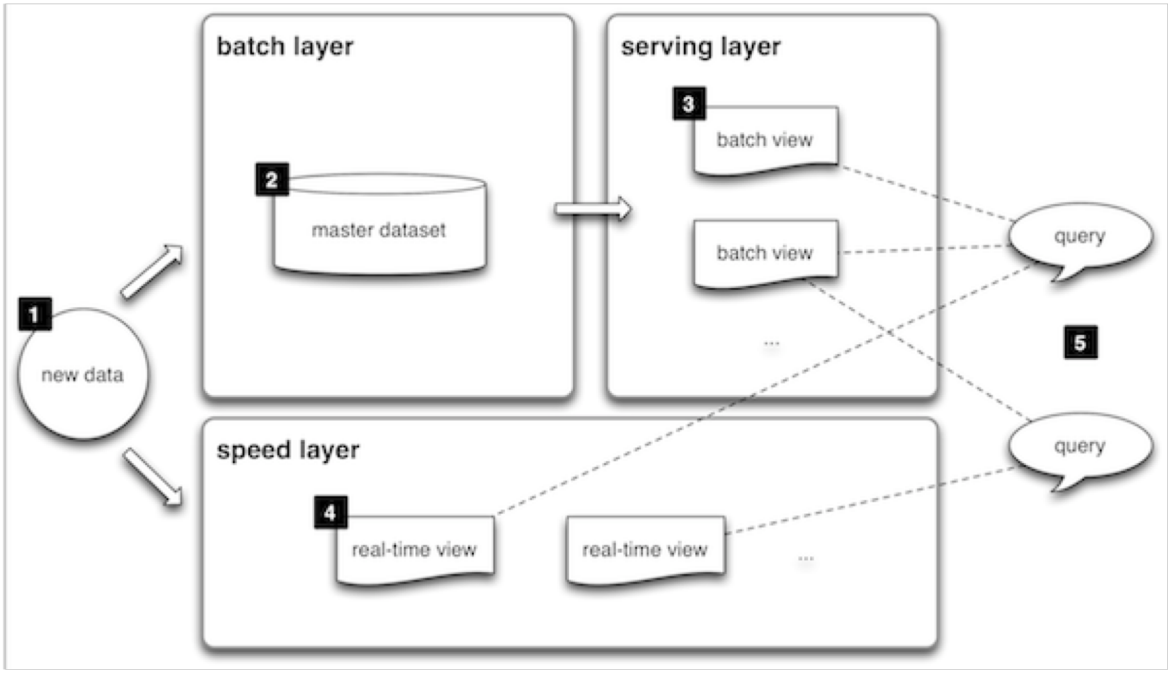
https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters

Lambda architecture



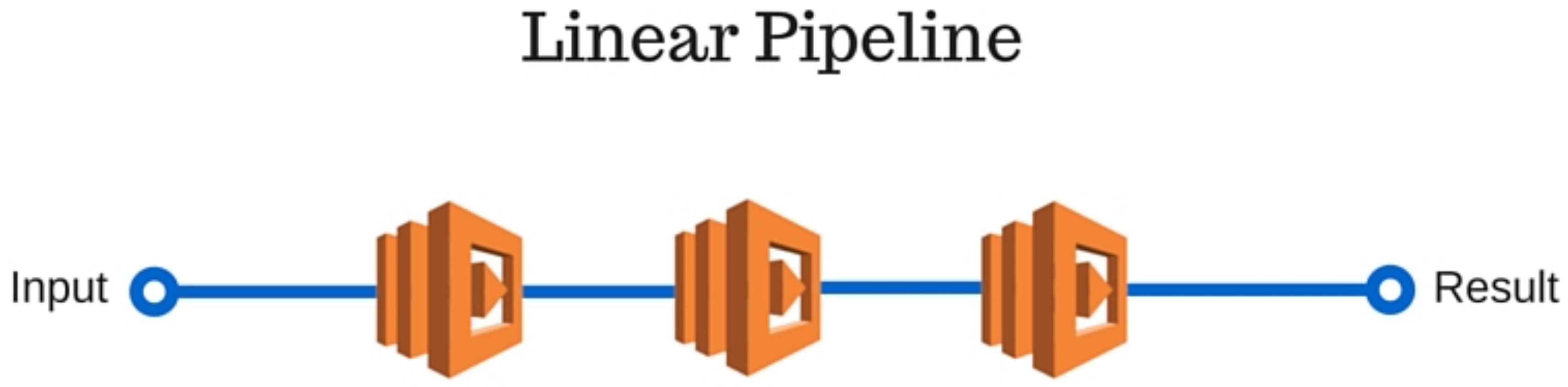
Lambda architecture

All **data** entering the system is dispatched to both the batch layer and the speed layer for processing.
The **batch layer** has two functions: (i) managing the master dataset (an immutable, append-only set of raw data), and (ii) to pre-compute the batch views.
The **serving layer** indexes the batch views so that they can be queried in low-latency, ad-hoc way.
The **speed layer** compensates for the high latency of updates to the serving layer and deals with recent data only.
Any incoming **query** can be answered by merging results from batch views and real-time views



http://lambda-architecture.net

Lambda piper



http://arjunkomath.github.io/lambda-piper/

Shared-data style

Shared-data: Overview

- A number of agents (data consumers/producers) communicate via a centralized data store
- The primary goal of this style is total decoupling among agents, that can be added or detached at runtime without knowing each other
- Architectures based on data sharing have the goal of achieving data integration
- The agents are relatively independent of each other (interact only through the data store)
- The data store independent from the agents

Shared-data: control flow

- The control flow within a system based on a shared data store can be managed either by the data store itself or by the agents
- These two kinds of control flow distinguish the two substyles:
  - ***Repository (passive)***: an agent sends a request to the data store to perform some action (e.g. insert data)
  - ***Blackboard (active)***: the data store notifies and sends data to subscribing agents when something changes

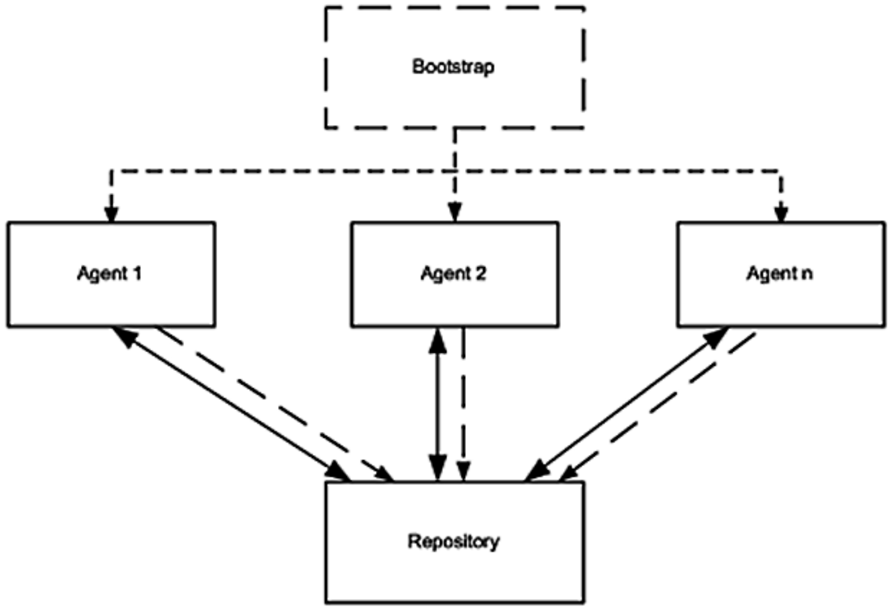
Data store: CRC

Class DataStore	**Collaborators**
Responsibilities •Connector: connects agents •Stores data using some schema •Allows writing or modifying or deleting data •Allows reading data	• Agent

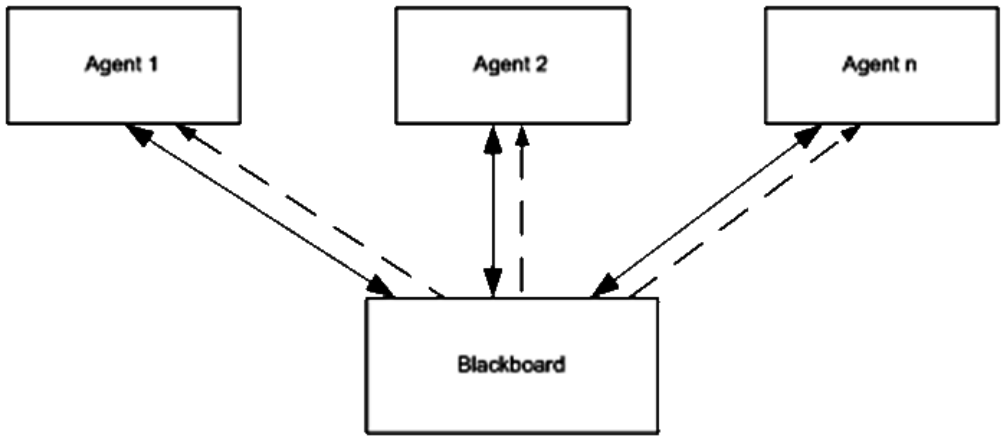
Agent: CRC

Class Agent	**Collaborators**
Responsibilities •Component (computes a transformation function) •Reads data from DataStore •Writes or modifies or deletes data from DataStore	• DataStore

Shared-data: control flows



1. Repository Control Flow

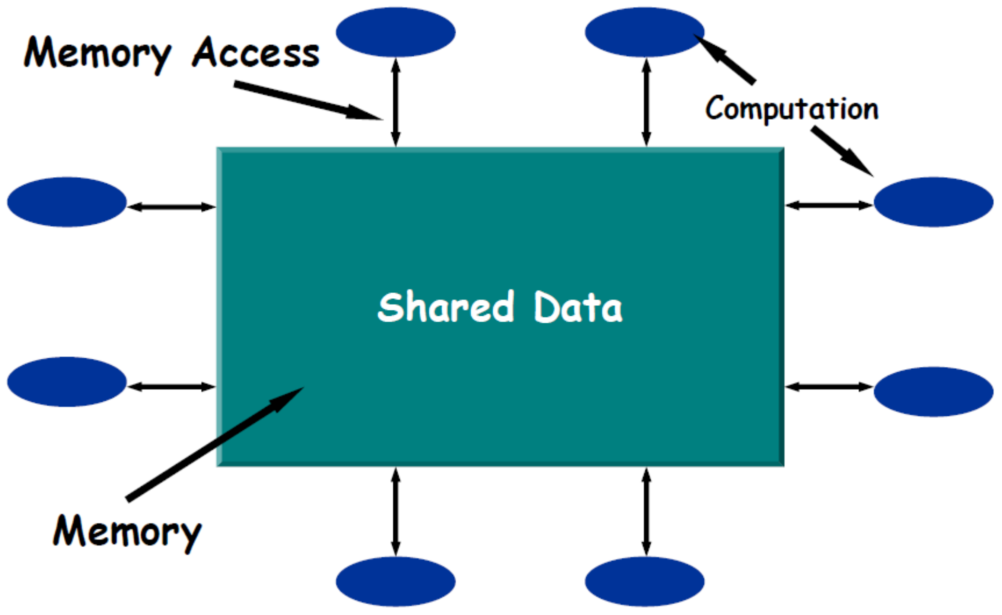


2. Blackboard Control Flow

Shared-data: Architecture

The substyles share two kinds of components:

- a shared data store representing the current state
- a collection of independent agents operating on the data store



Copyright © Emad Shihab, Architecture Styles

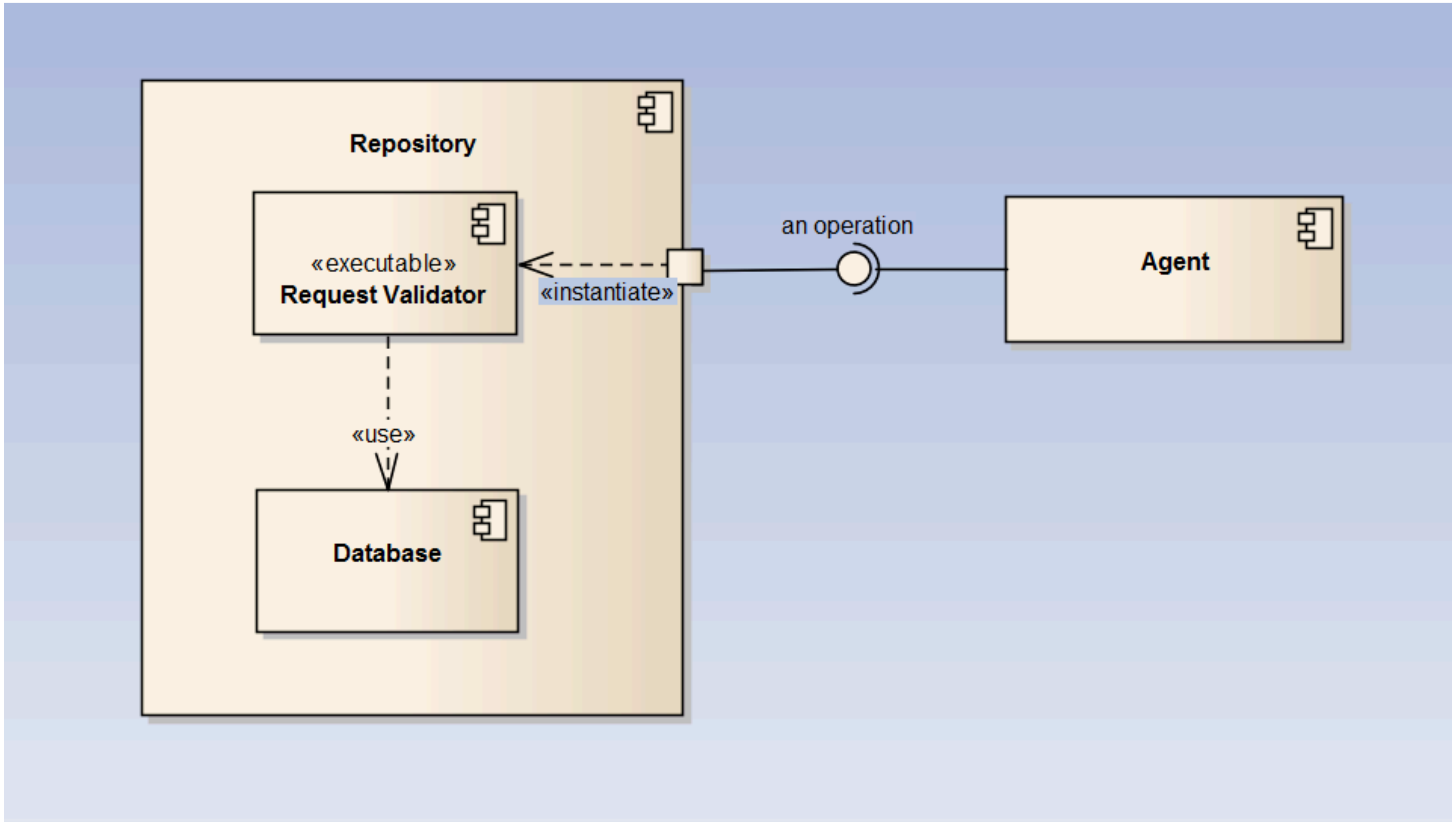
Style Repository: Overview

- Data store is passive
- Agents read and write in the data store
- The style supports interactive data processing
- Agents control the computation flow

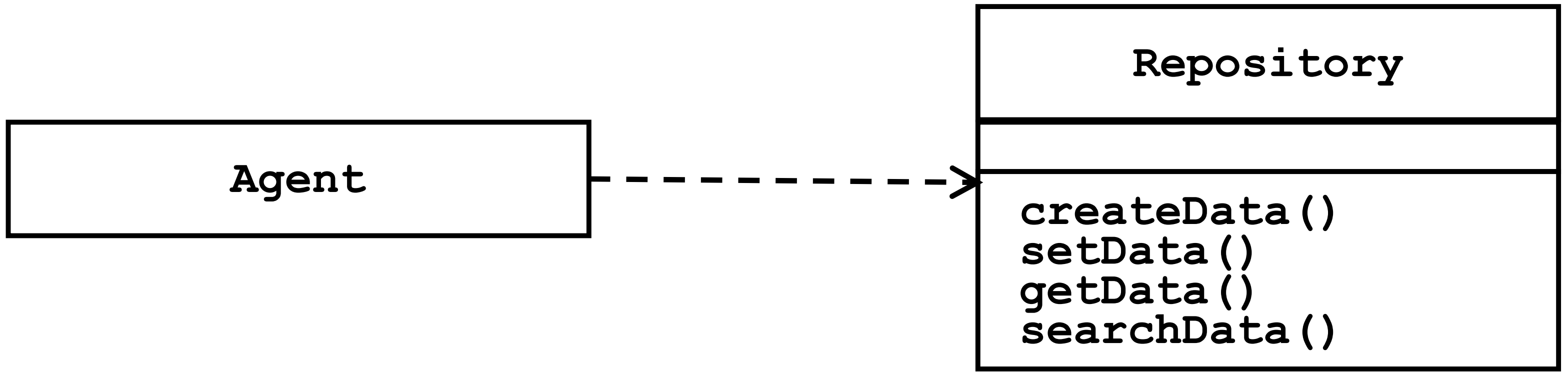
Repository: Architecture

- Components:
  - **Agents**
    - Independent modules that collectively contain the knowledge needed to solve a problem
    - They all have a direct connection with the Repository
  - **Repository**
    - Central store containing the shared data
    - Communication among agents mediated by the Repository
    - Communication may be initiated by any agent
- Connectors:
  - **Procedure calls or direct memory accesses**
    - C.R.U.D. (Create – Read – Update – Delete) Data

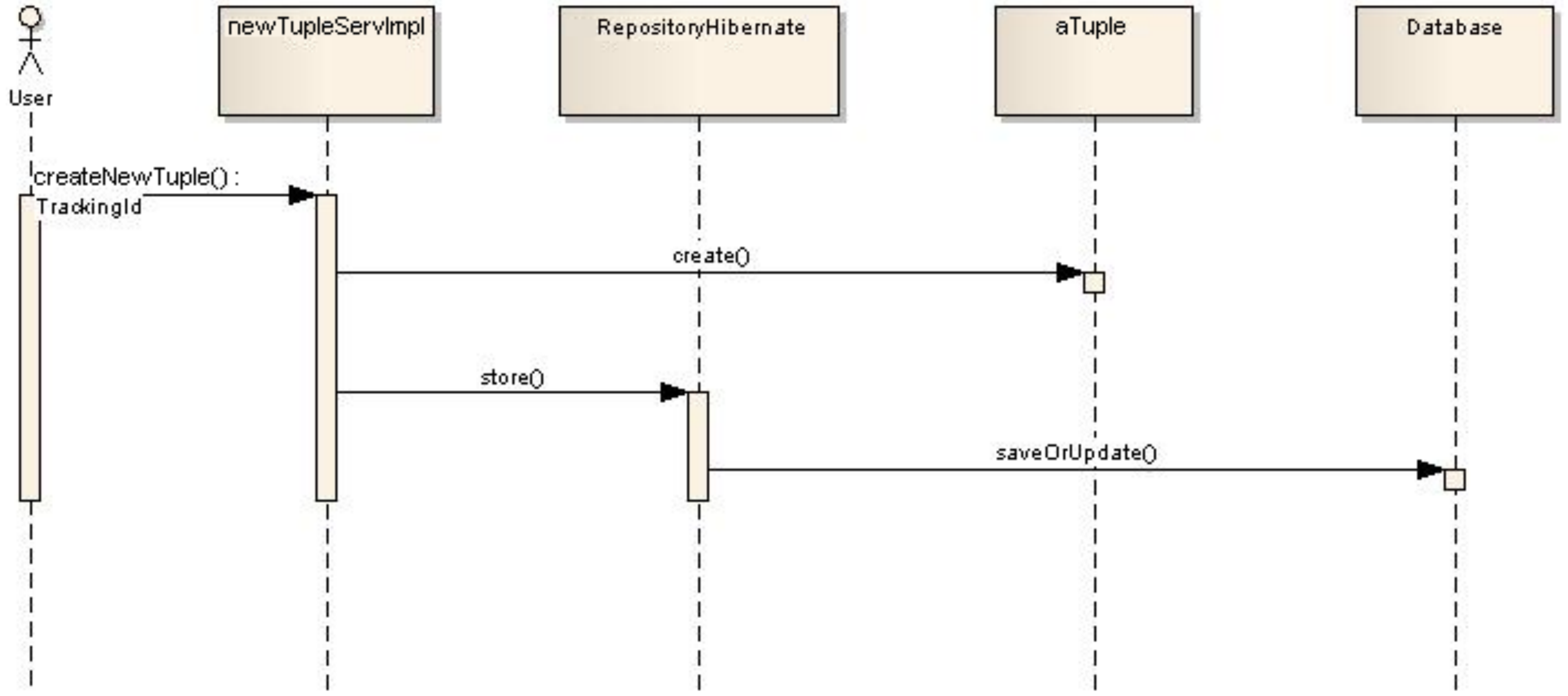
Repository: Component Diagram



Repository: Class Diagram

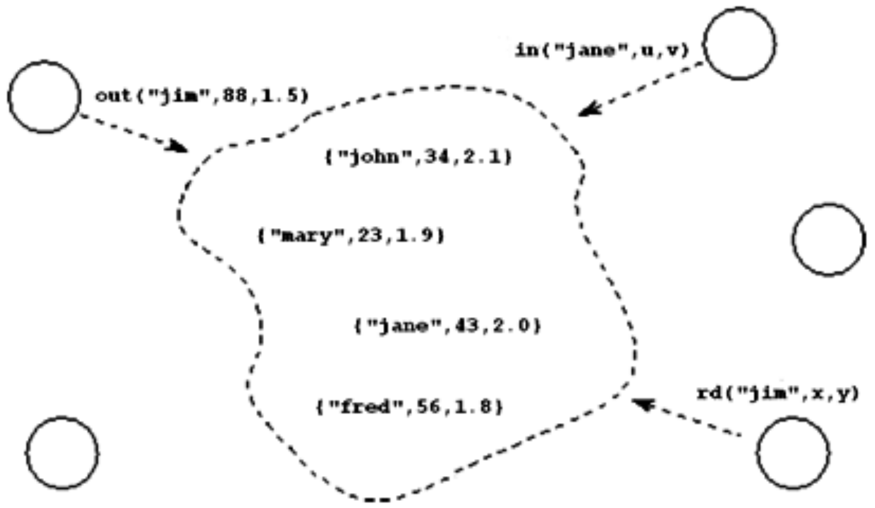


Repository: Sequence Diagram



Repository example: Tuple Space

- A Tuple Space is a repository where agents (*workers*) read or write tuples by pattern matching
- Main primitives:
  - **`out(tuple)`** non blocking, creates new tuple
  - **`rd(tuple pattern)`** blocking, does not delete matching tuple
  - **`in(tuple pattern)`** blocking, deletes matching tuple
  - **`eval(tuple)`** non blocking, creates new worker



Blackboard: Overview

- A number of *Knowledge Sources* (KS) have access to a shared data store called “blackboard”
- The *Blackboard* provides an interface to inspect and update its content
- The *Control* module/object activates the KSs following some strategy
- Upon activation, a KS inspects that blackboard to see if it can contribute to solving the problem
- If it can, the Control object can allow the KS to put its partial (or final) solution on the board
- Particularly suited to solve nondeterministic problems (decision support, signal processing, speech recognition…)

Blackboard: components

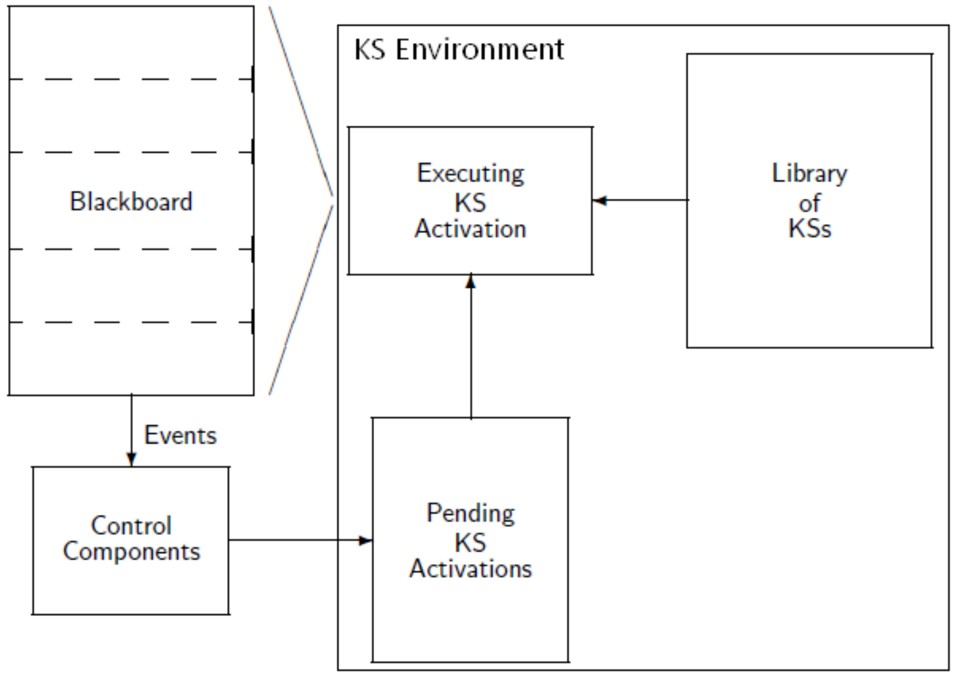
Components:

- **Knowledge Sources**
  - Independent modules that contain the knowledge needed to solve the problem
- **Control**
  - Makes runtime decisions about the course of problem solving and the expenditure of problem-solving resources
  - Provide a mechanism (needed by Knowledge Sources) to organize their use in the most effective and coherent fashion
- **Blackboard**
  - Global database containing input data, partial solutions, and other data that are in various problem-solving states

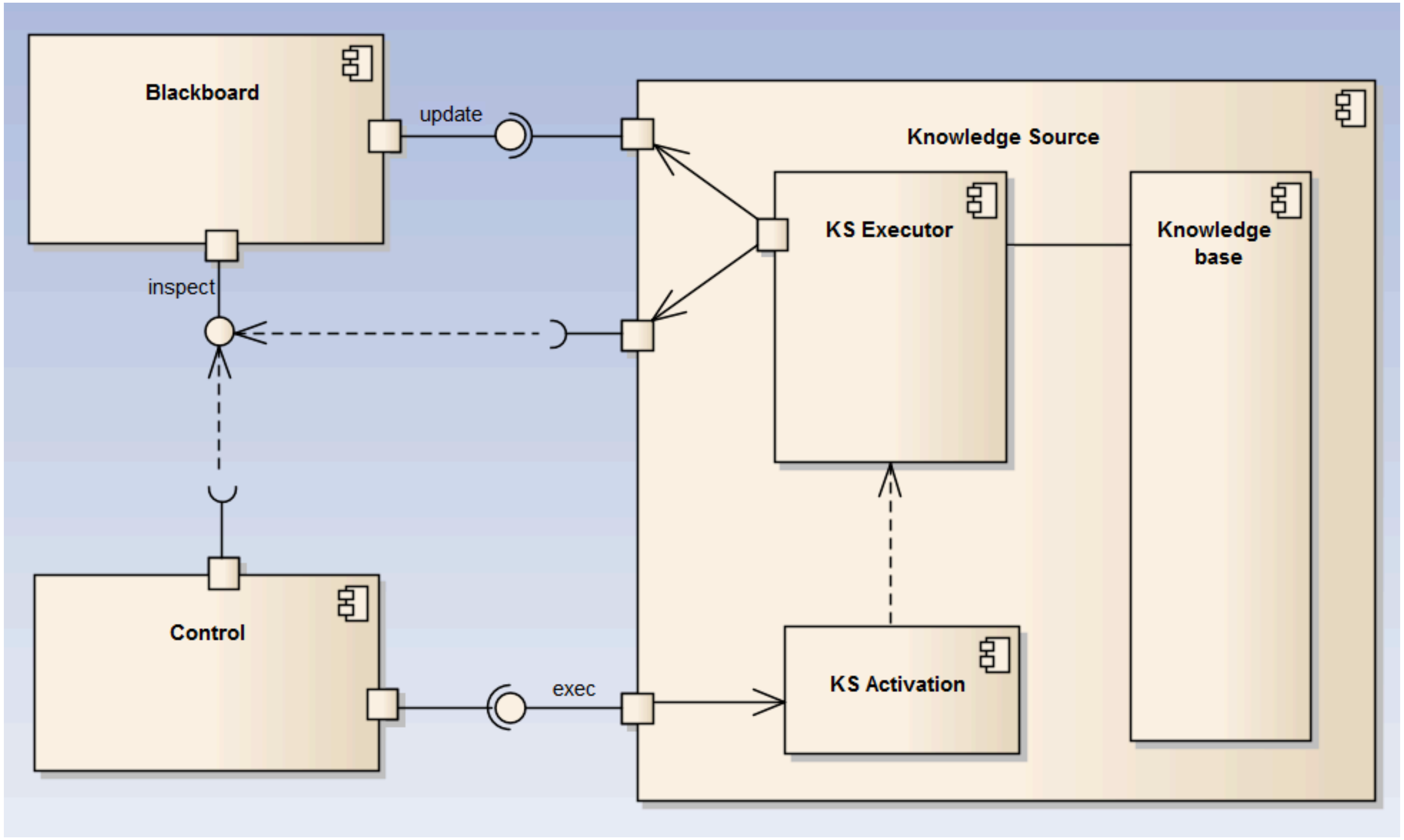
Blackboard: connectors

Connectors:

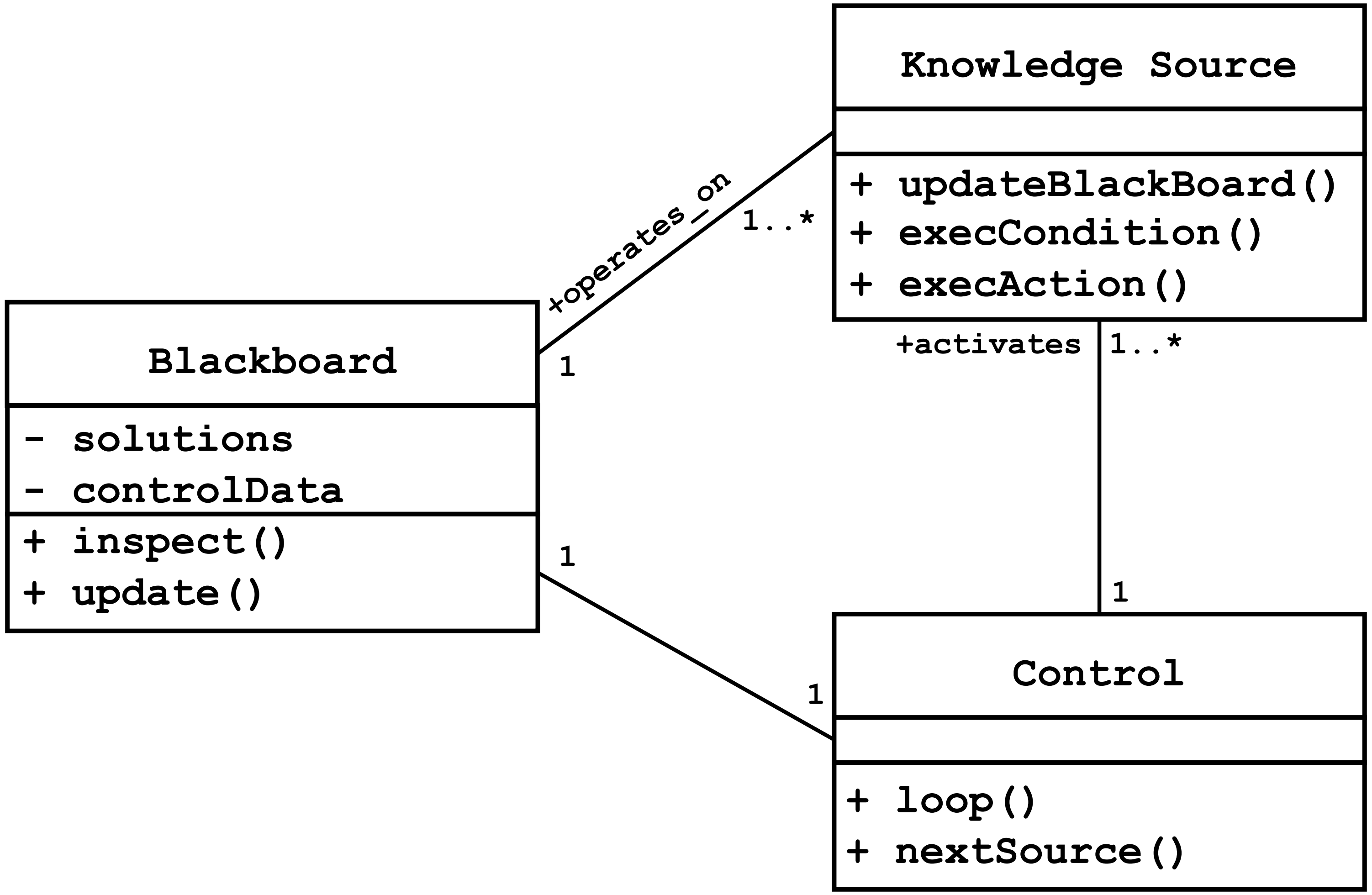
- **Procedure calls**
  - Triggered by the current state of the blackboard, rather than by external inputs



Blackboard: Component Diagram

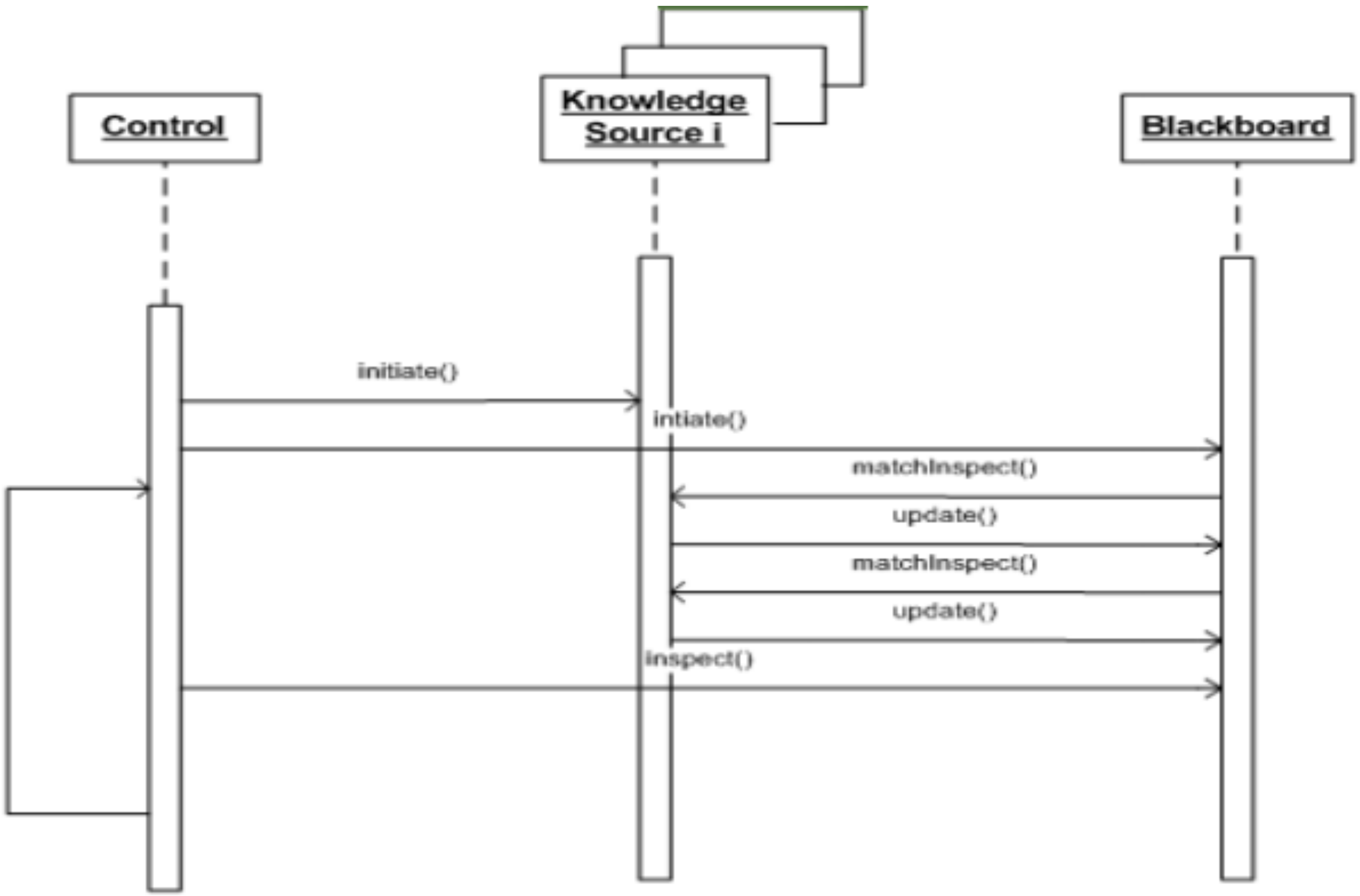


Blackboard: Class Diagram

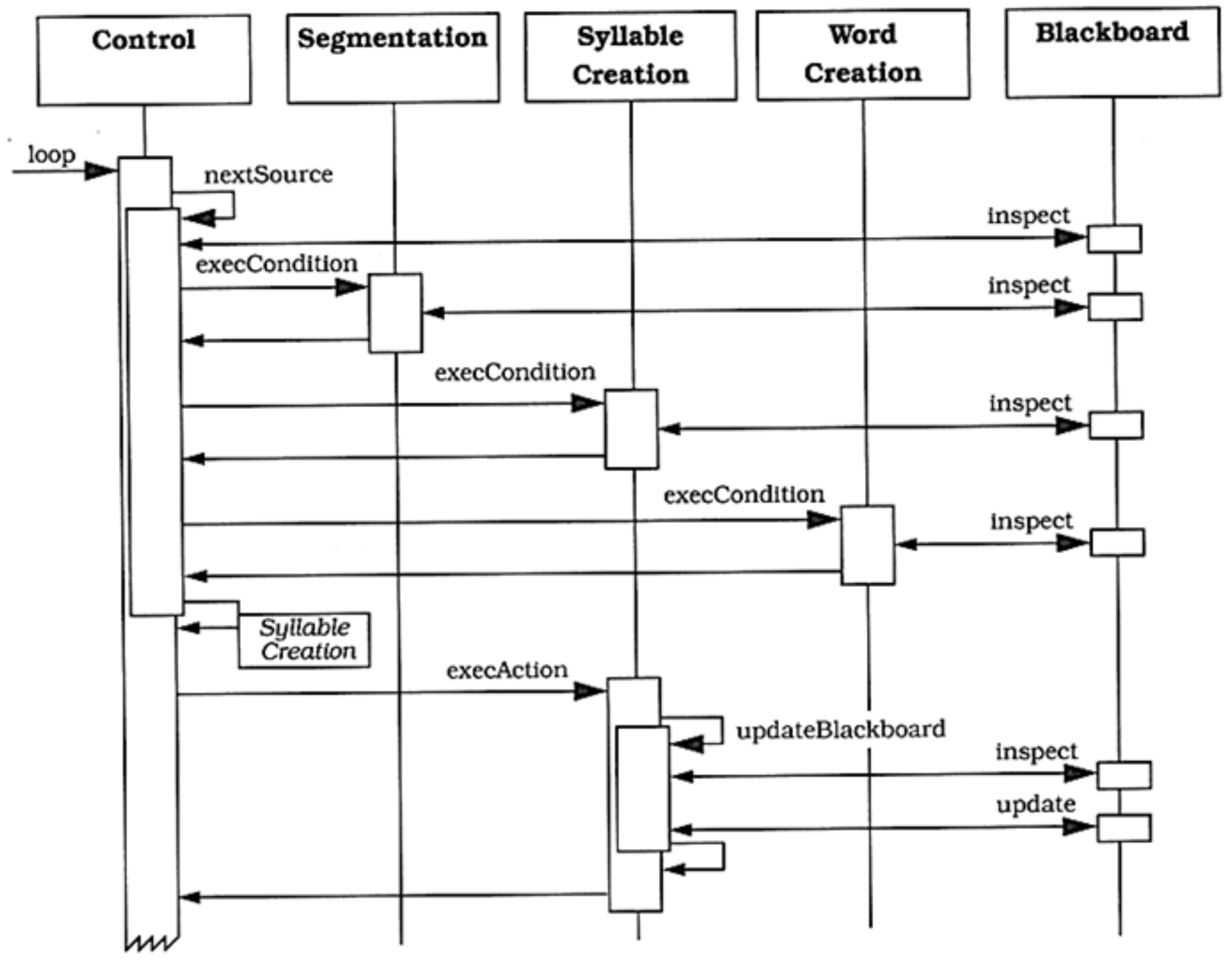


Copyright © Nguyễn Ngân, Blackboard Architecture

Blackboard: Sequence Diagram (1)



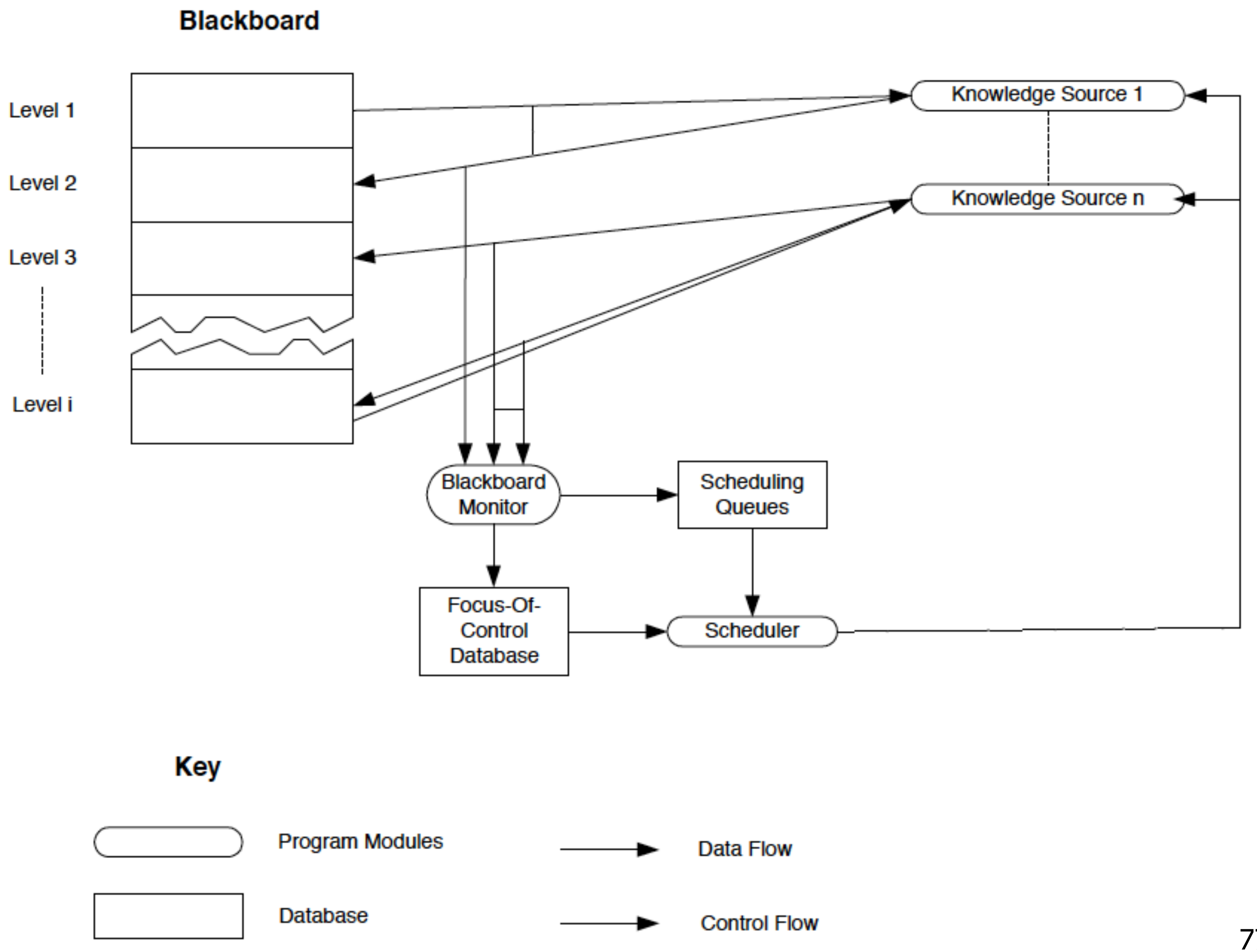
Blackboard: Sequence Diagram (2)



Blackboard: Sequence Diagram (3)

- The main loop of Control started
- Control calls **`nextSource()`** to select the next knowledge source
- **`nextSource()`** looks at the blackboard and determines which knowledge sources to call
    - For example, **`nextSource()`** determine that Segmentation, Syllable Creation and Word Creation are candidate
        - **`nextsource()`** invokes the condition-part of each candidate knowledge source
        - The condition-parts of candidate knowledge source inspect the blackboard to determine if and how they can contribute to the current state of the solution
    - The Control chooses a knowledge source to invoke and a set of hypotheses to be worked on (according to the result of the condition parts and/or control data)
        - Apply the action-part of the knowledge source to the hypothesis
        - New contents are updated in the blackboard

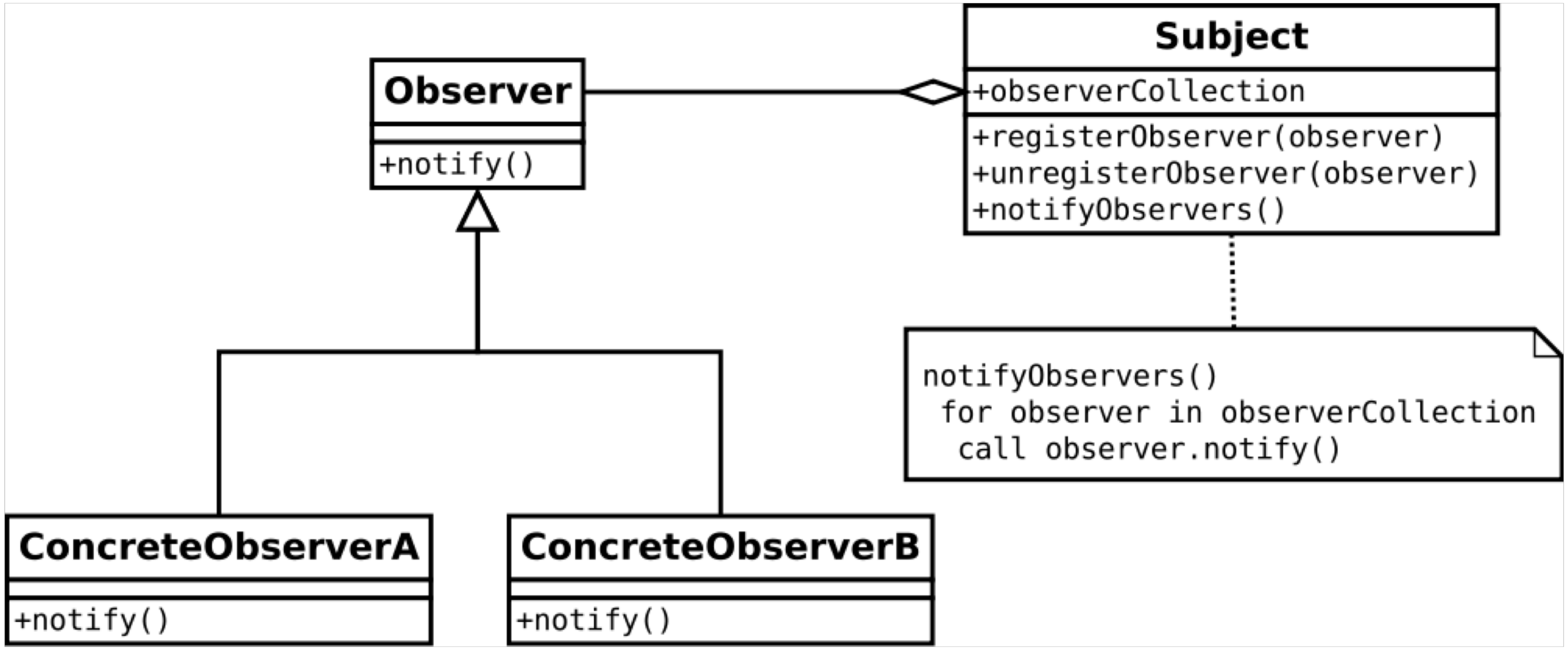
Blackboard example: Hearsay II



Shared data: Publish/Subscribe

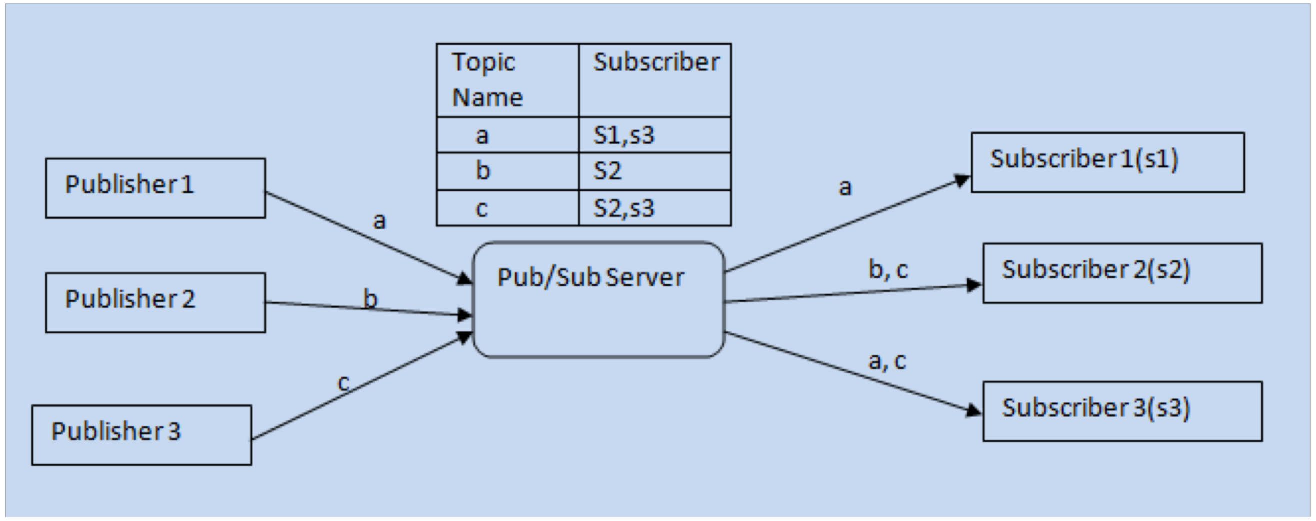
- The original blackboard system Hersay was sequential
- In a distributed system agents are concurrent and interact by broadcast or multicast messages
- Each agent notifies a state change via a message (it “raises an event”)
- All agents interested to the event (“observers”) receive a copy of the message
- Event generators do not know the observers (decoupling)
- This variant makes use of the *Observer Pattern*

Observer pattern

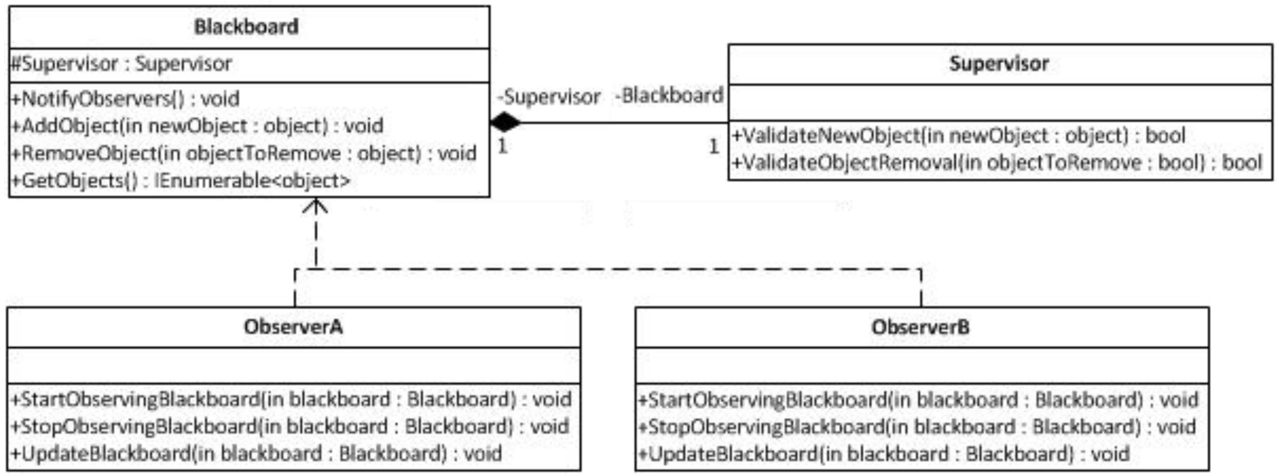


```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

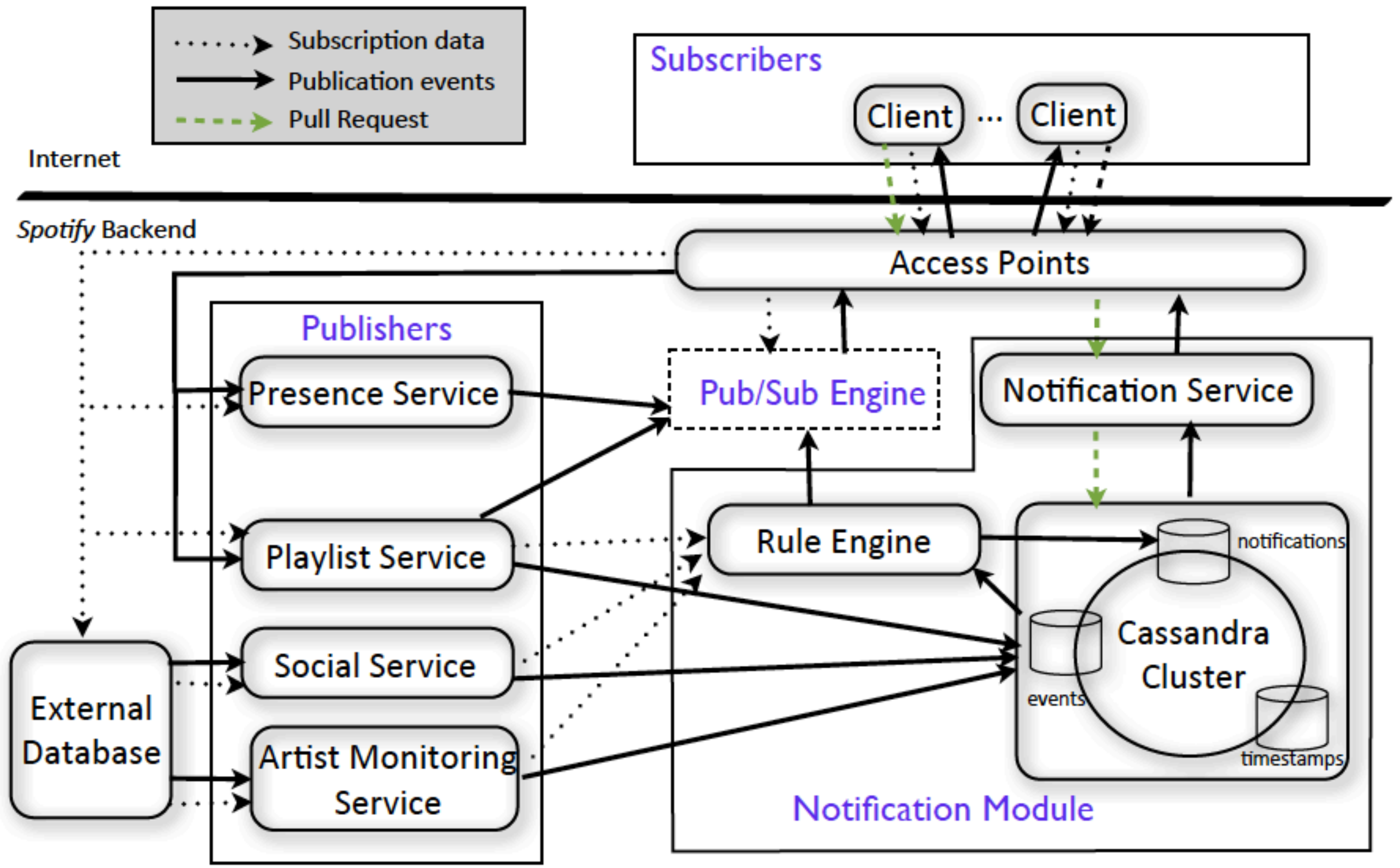
Publish/Subscribe as observer



Topic Name	Subscriber
a	S1,s3
b	S2
c	S2,s3


Blackboard
#Supervisor : Supervisor
+NotifyObservers() : void
+AddObject(in newObject : object) : void
+RemoveObject(in objectToRemove : object) : void
+GetObjects() : IEnumerable<object>
-Supervisor
-Blackboard
1
1
Supervisor
+ValidateNewObject(in newObject : object) : bool
+ValidateObjectRemoval(in objectToRemove : bool) : bool
ObserverA
+StartObservingBlackboard(in blackboard : Blackboard) : void
+StopObservingBlackboard(in blackboard : Blackboard) : void
+UpdateBlackboard(in blackboard : Blackboard) : void
ObserverB
+StartObservingBlackboard(in blackboard : Blackboard) : void
+StopObservingBlackboard(in blackboard : Blackboard) : void
+UpdateBlackboard(in blackboard : Blackboard) : void

Pub/Sub example: Spotify



Setty et al., The Hidden Pub/Sub of Spotify, Proc. ACM DEBS 2013

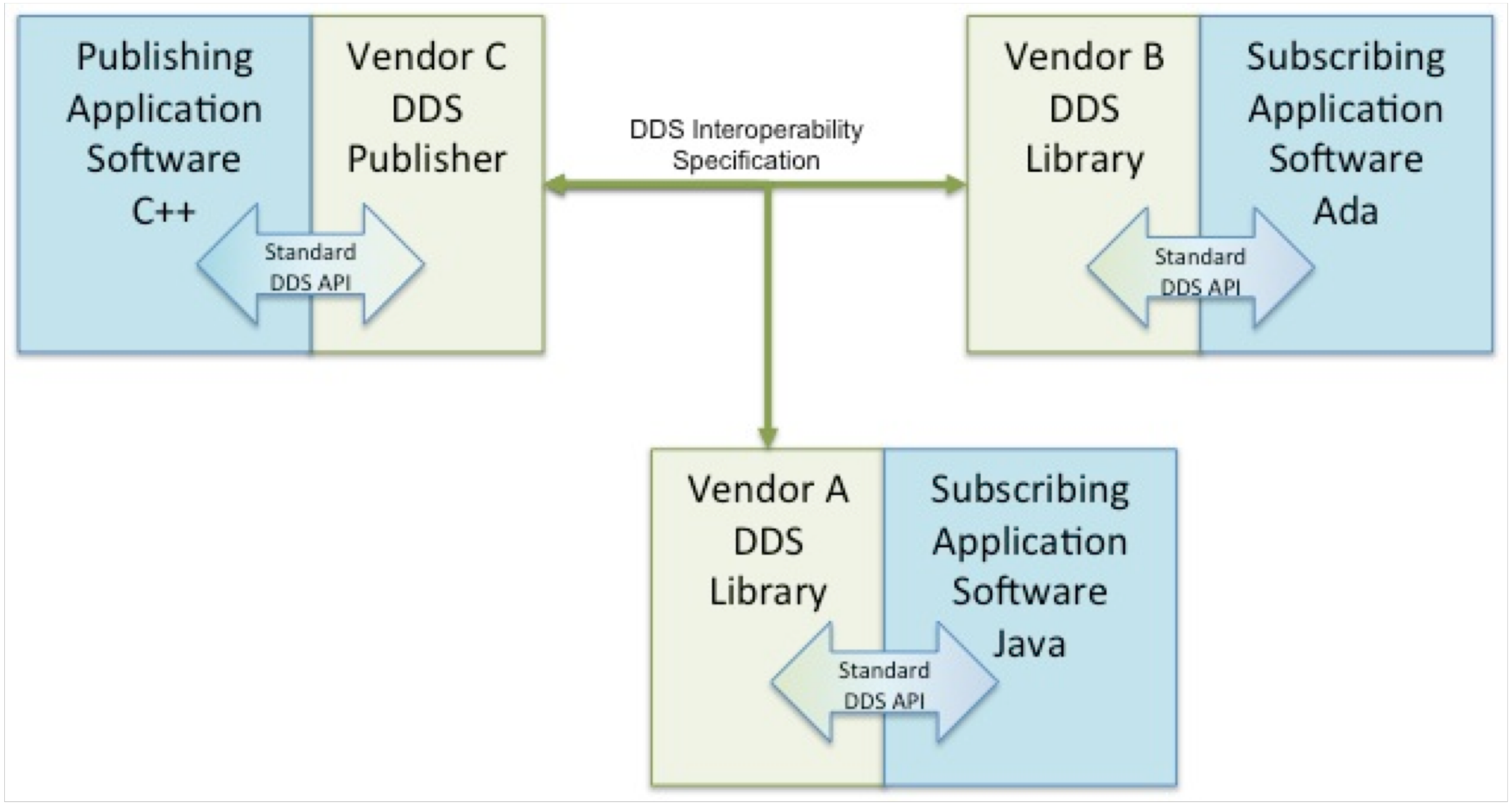
Pub/Sub example: Data Distribution Service (DDS)

- DDS for Real Time systems is a OMG specification of a publish/subscribe middleware architecture which standardizes a data centric model for programming distributed systems
- The standard is used in applications such as smartphone operating systems, transportation systems and vehicles, software defined radio, healthcare, Internet of Things

DDS

- The DDS publish-subscribe model avoids complex network programming for distributed applications
- DDS supports mechanisms that go beyond the basic publish-subscribe model. The key benefit is that applications that use DDS for their communications are entirely decoupled. The applications never need information about the other participating applications, including their existence or locations.
- DDS automatically handles all aspects of message delivery, without requiring any intervention from the user applications, including:
  - determining who should receive the messages
  - where recipients are located
  - what happens if messages cannot be delivered

DDS: example


Publishing Application Software C++
Vendor C DDS Publisher
Standard DDS API
DDS Interoperability Specification
Vendor B DDS Library
Subscribing Application Software Ada
Standard DDS API
Vendor A DDS Library
Subscribing Application Software Java
Standard DDS API

Data distribution services (OMG)

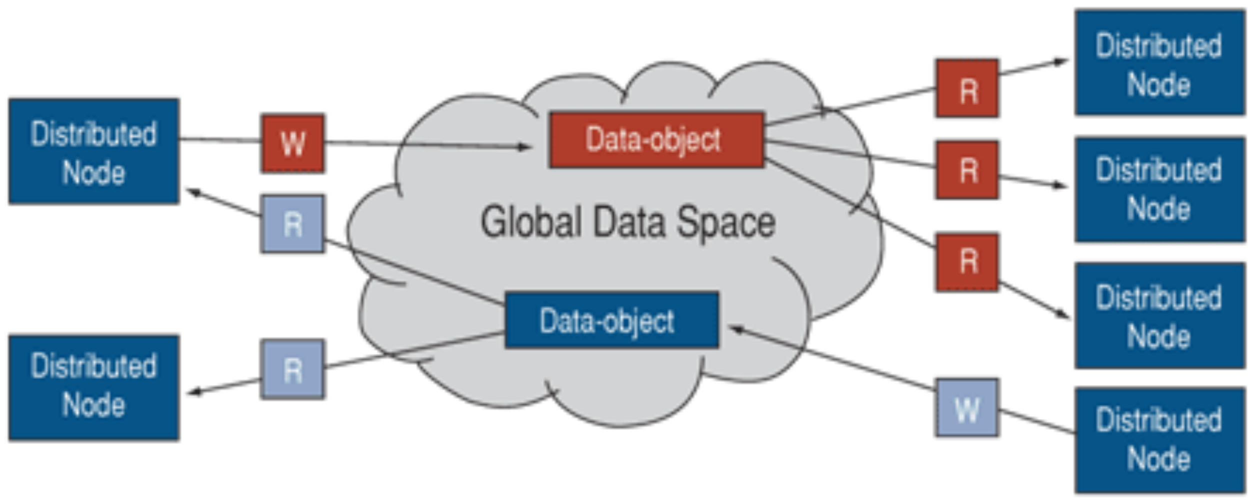
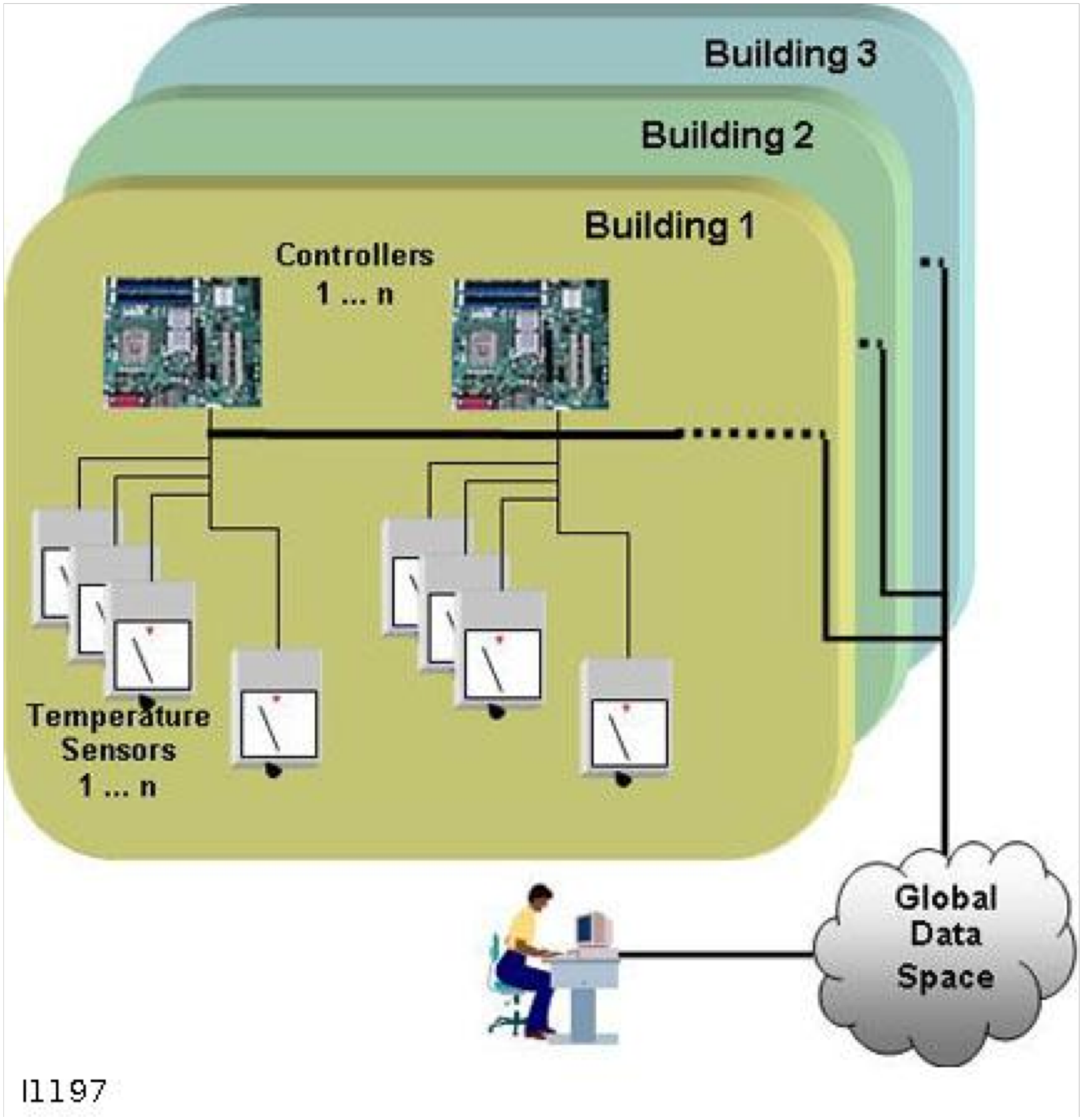


Figure 1 Overall DCPS model. Distributed nodes are able to read (R) and write (W) data objects that live in a shared virtual global data space.


Building 3
Building 2
Building 1
Controllers
1 ... n
Temperature
Sensors
1 ... n
Global
Data
Space
I1197

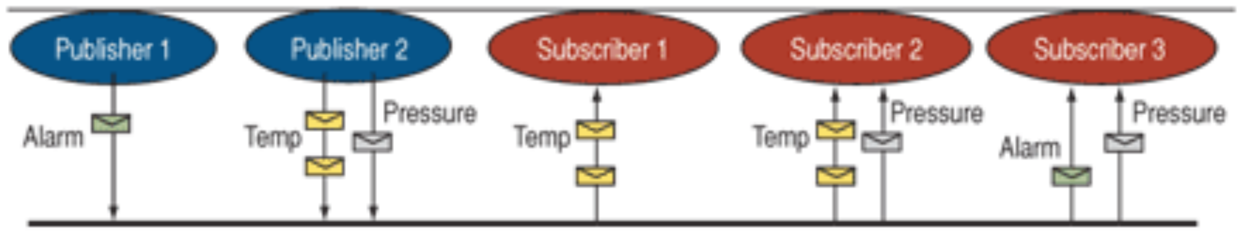


Figure 2 Topic-based publish/subscribe. Topic is an application-selected label—such as "pressure" or "temp"—used to identify publications and subscriptions.

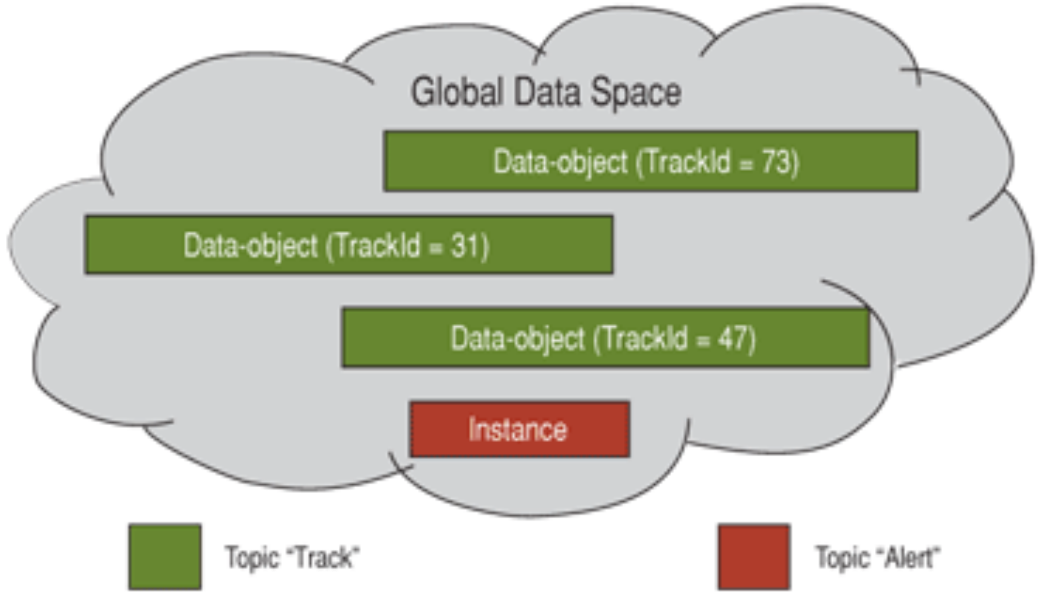
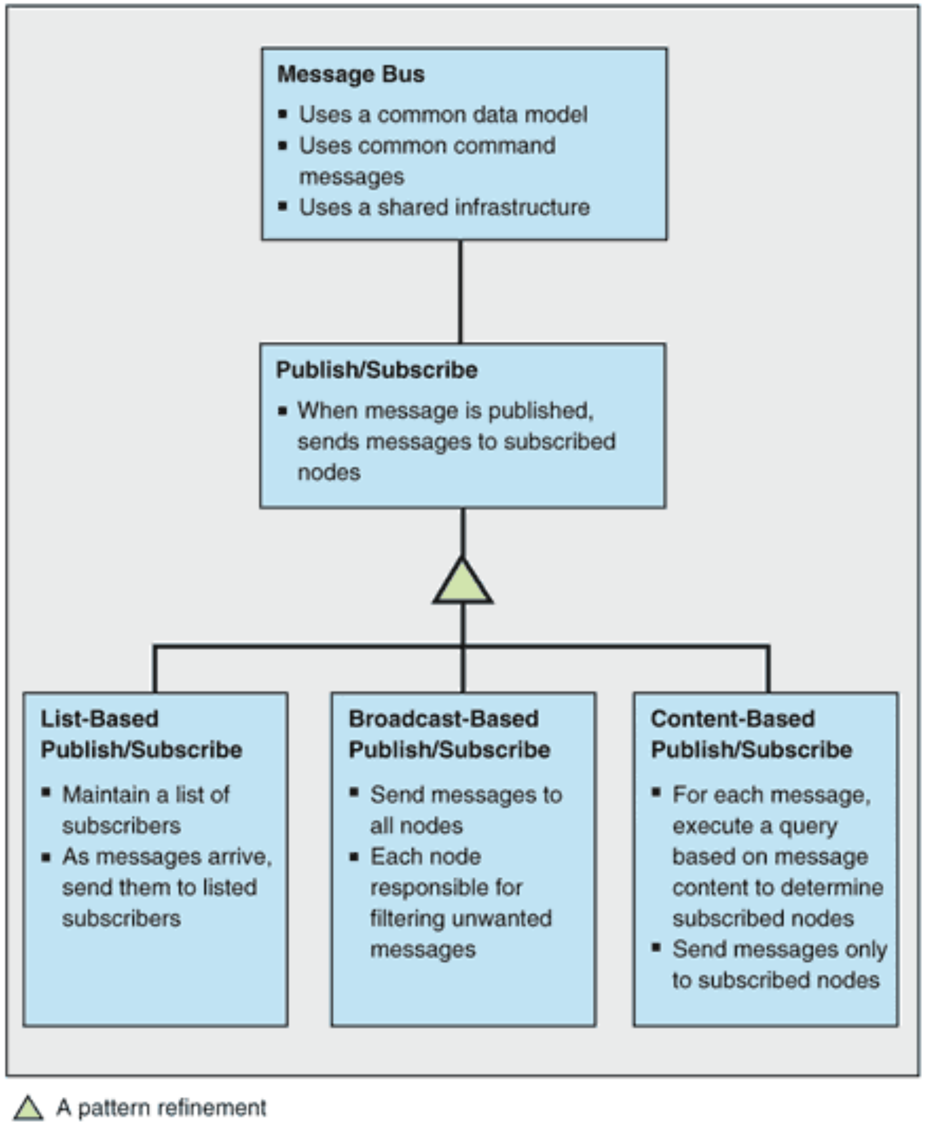

Global Data Space
Data-object (TrackId = 73)
Data-object (TrackId = 31)
Data-object (TrackId = 47)
Instance
Topic "Track"
Topic "Alert"

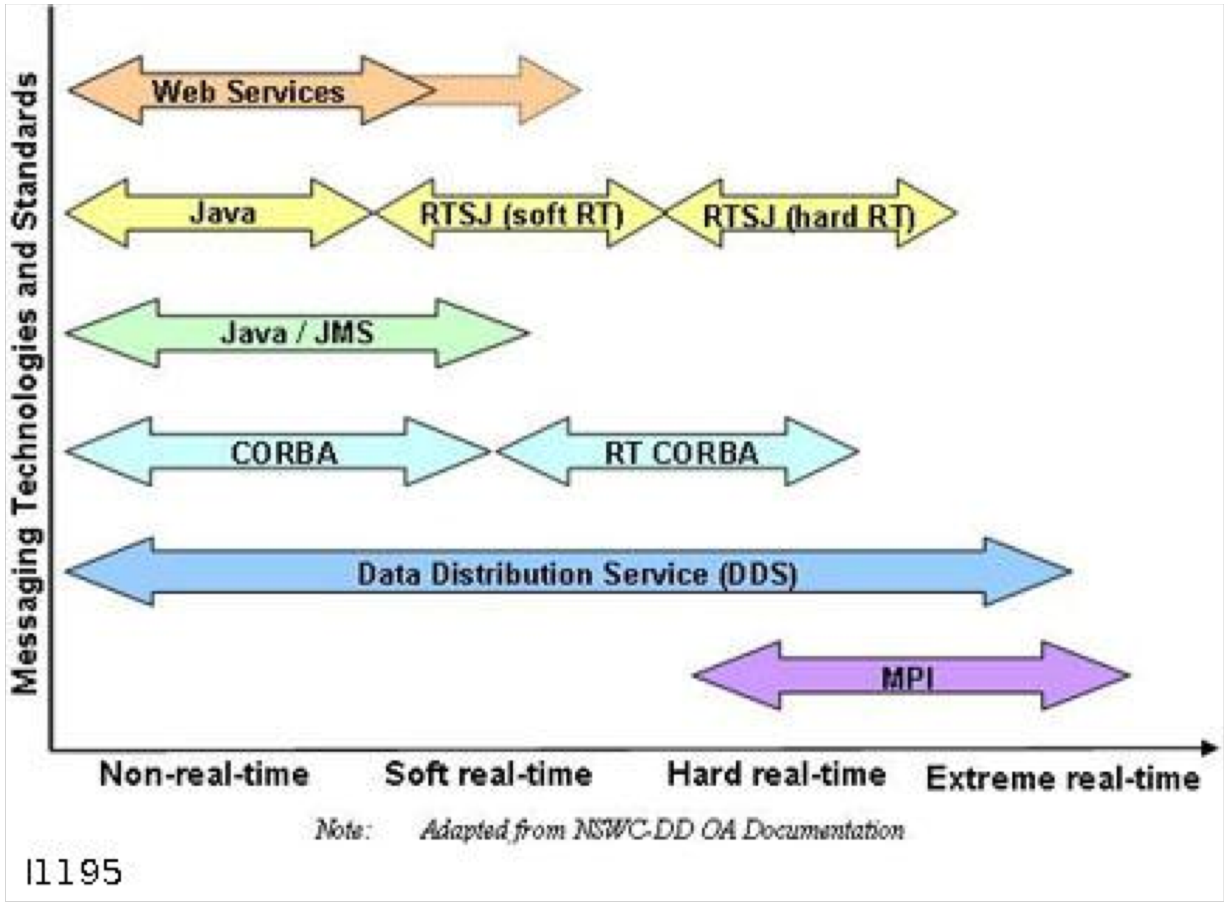
Figure 3 Topic and key identify data objects. A data object is identified by the combination of a topic (such as "Track") and a topic-specific key (such as a TrackId). Some topics (e.g., "Alert" above) may have no key indicating the reads/writes apply to a single data stream.

Publish/Subscribe: variants



Message Bus
▪ Uses a common data model
▪ Uses common command messages
▪ Uses a shared infrastructure
Publish/Subscribe
▪ When message is published, sends messages to subscribed nodes
List-Based Publish/Subscribe
▪ Maintain a list of subscribers
▪ As messages arrive, send them to listed subscribers
Broadcast-Based Publish/Subscribe
▪ Send messages to all nodes
▪ Each node responsible for filtering unwanted messages
Content-Based Publish/Subscribe
▪ For each message, execute a query based on message content to determine subscribed nodes
▪ Send messages only to subscribed nodes
A pattern refinement

Variety or real time constraints



Shared-data: Pro & Cons (1)

- Benefits:
  - It is an effective way to share huge amounts of data: write once for all to read
  - Each subsystem has not to take care of how data are produced/consumed by other subsystems
  - It allows a centralized management of system backups, as well as of security and recovery procedures
  - The data sharing model is available as the *shared-data schema,* hence it is easy to plug new subsystems

Shared-data: Pro & Cons (2)

- Pitfalls:
  - Subsystems have to agree on a data model, thus impacting on performance
  - Data evolution: it is “expensive” to adopt a new data model since:
    - it has to be applied on the entire shared data
    - all the subsystems have to be updated
  - Not for all the subsystems’ requirements in terms of backup and security are always supported by the shared data
  - It is tricky to deploy the shared data on several machines, preserving the logical vision of a centralized entity, due to redundancy and data consistency matters

Architectural styles and design patterns

- A design pattern solves a design problem
- An archititetctural style is not intended to solve a problem: it is a way of organizing the structure and behavior of a system
- Most architectural styles include some design patterns

Summary

- The basic architectural styles are few and their properties should be studied and exploited
- Their descriptions and properties are better understood using a modeling notation
- The architectural models and descriptions can be exploited by a technology like UML via automatic transformations: see the Model Driven Architecture

Self test

- What is the relationship between style and architecture?
- What is an active pipe?
- Which are the main differences between the repository and blackboard styles?

References

- **Clemens et al.,** *Documenting Software Architectures*, Addison Wesley, 2010
- **Qian** et al., *Software Architecture and Design Illuminated*, Jones and Bartlett, 2009

Useful sites

- www.softwarearchitectureportal.org
- www.dossier-andreas.net/software_architecture/
- www.bredemeyer.com/links.htm
- www.opengroup.org/architecture/
- www.iasahome.org
- alistair.cockburn.us

Questions?