# Architectural Styles and Patterns

# Architectural Patterns & Styles

## Consensus

- Established Solutions
- Common vocabulary
- Document
- Reason over quality

## Controversy

- Philosophy (context-problem-solution vs components-connections)
- Granularity (vs Design patterns)
- Categorization

# Architectural Patterns (Wikipedia)

- Presentation-abstraction-control
- Three-tier
- Pipeline
- Implicit Invocation
- Blackboard
- Peer-to-peer
- Service-oriented architecture
- Naked Objects
- Model-View Controller

# Architectural Patterns (Shaw & Garland)

- Dataflow Systems -- Batch, Pipes and Filters
- Call-and-return systems -- Main program and subroutines, OO systems, Hierarchical Layers
- Independent components -- Communicating processes, Event systems
- Virtual Machines -- Interpreters, Rule-based systems
- Data-Centered Systems -- Databases, Hypertext systems Blackboards

# Elements of Architecture (Shaw & Garland)

## Components

- Clients
- Servers
- Filters
- Layers
- Databases

## Connectors

- Procedure calls
- Events broadcast
- Database protocols
- Pipes


- What are the structural patterns permitted?
- What is the underlying computational model?
- What are the invariants of the style?
- What are examples of its use?
- What are the tradeoffs?
- ...

# Architectural Patterns
## (Baas, Clements & Kazman)

- Dataflow Systems -- Batch, Pipes and Filters
- Data-Centered -- Repository, Blackboard
- Virtual machine -- Interpreter, Rule-based System
- Independent Components
- Call/Return.

# Elements of Architecture

## (Baas, Clements & Kazman)

Element Types

Interaction Mechanisms

Topological Layout

Semantic Constraints

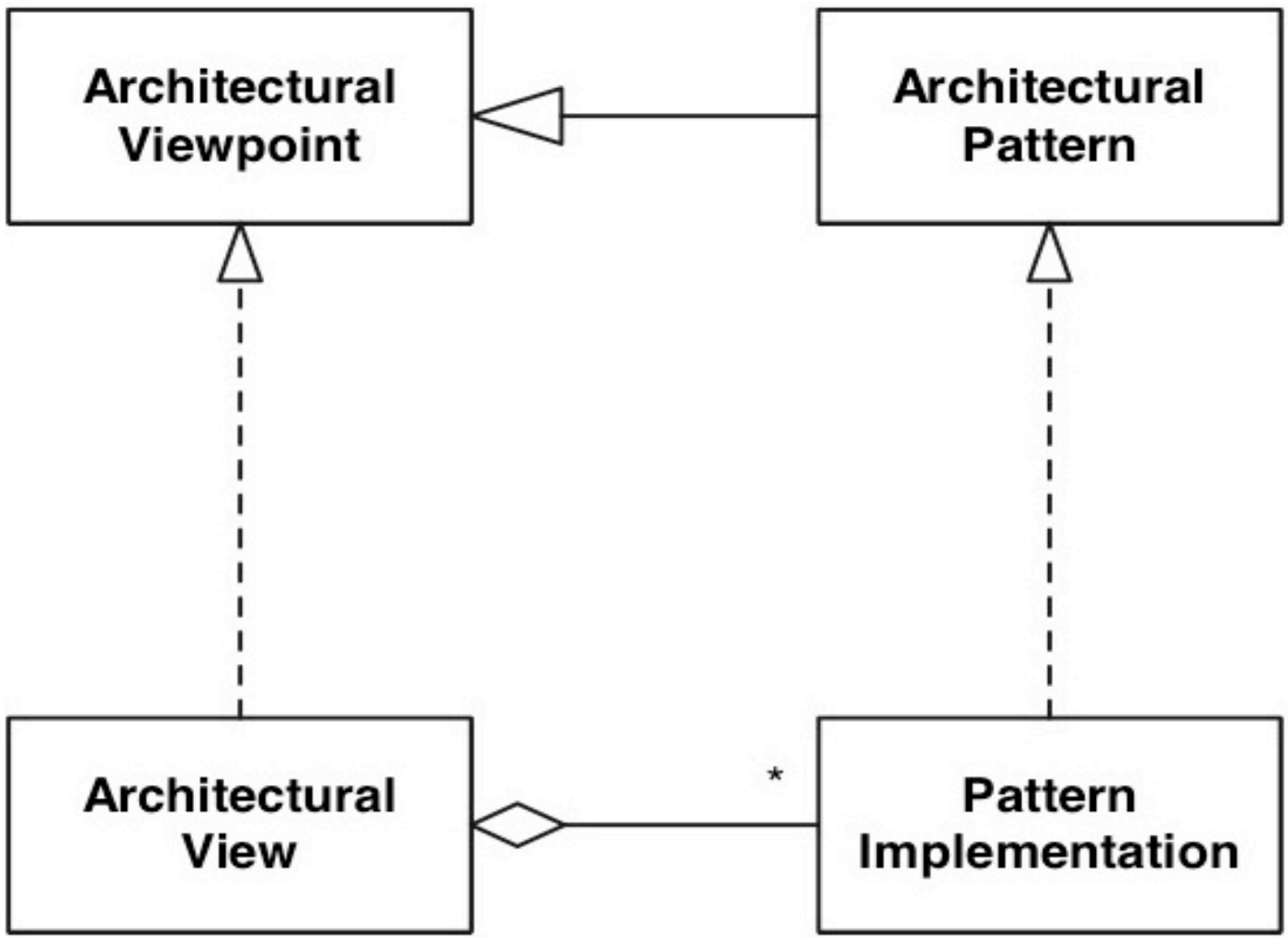
# Architectural views

# Architectural Patterns
## (Avgeriou & Zdun)

Structured around the concept of views

- Layered
- Data-flow
- Data-centered
- Adaptation
- Language extension
- User Interaction
- Component interaction
- Distribution

# Viewpoint, views and patterns

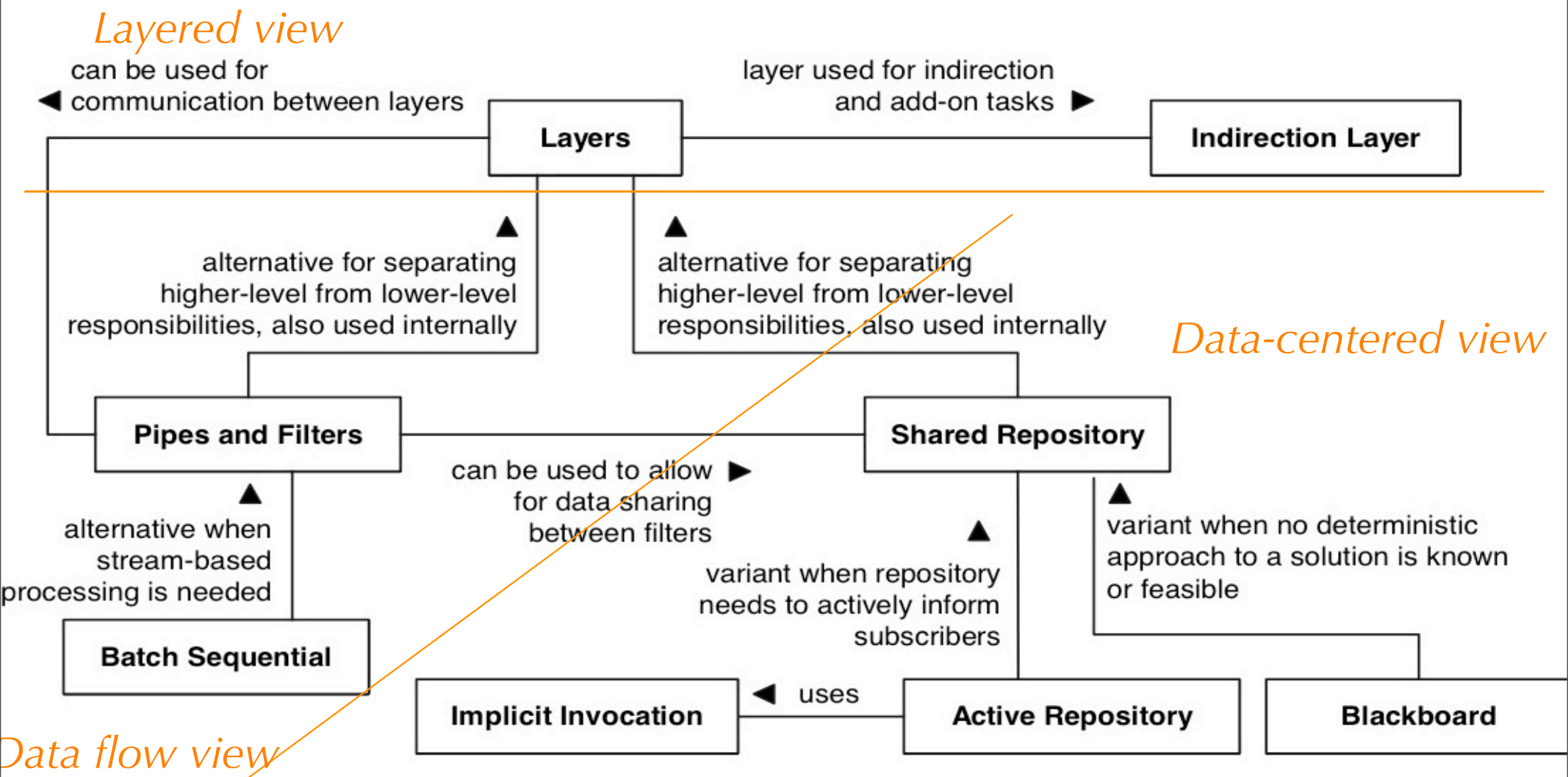


# Patterns and views

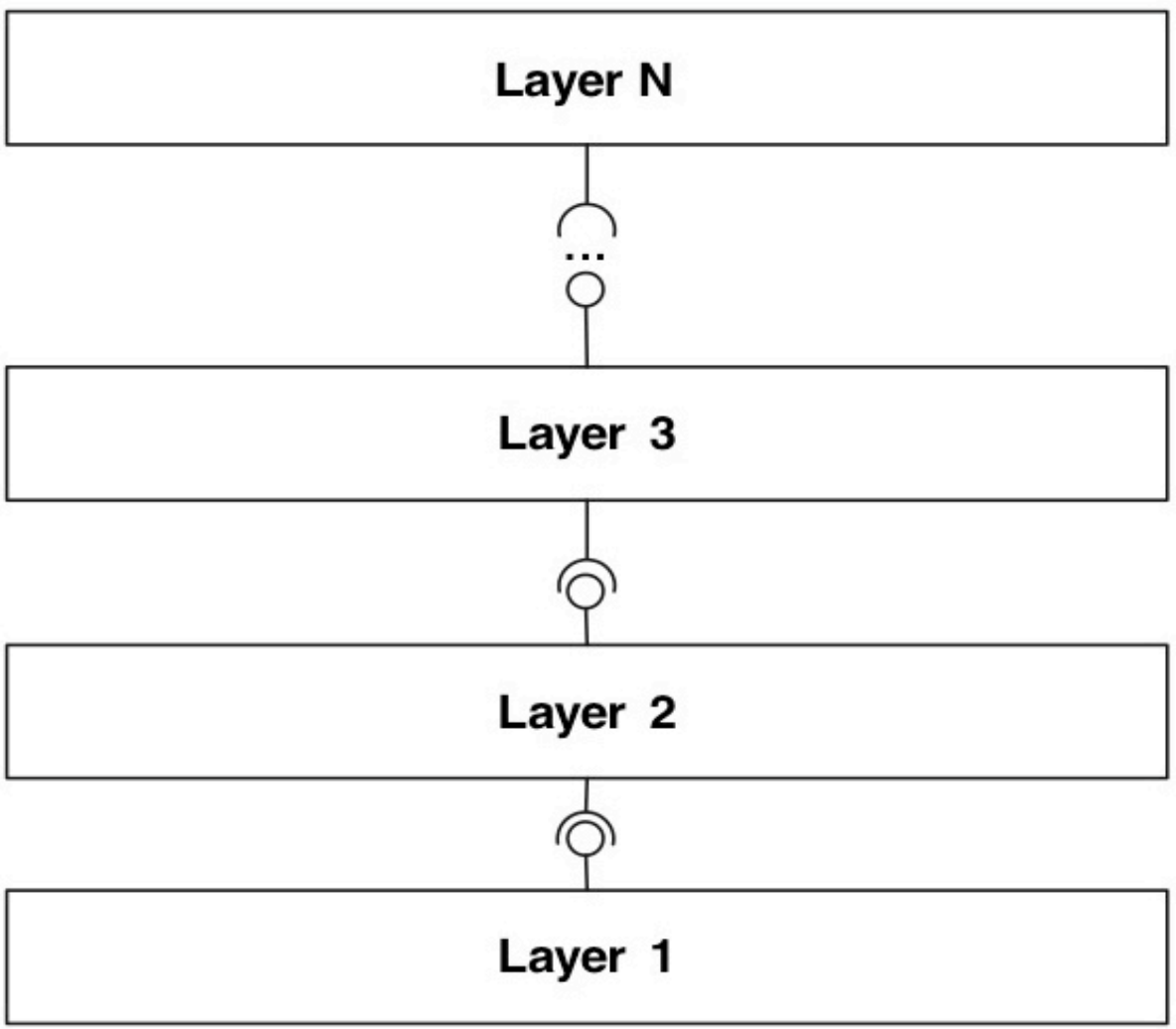
- **Architectural viewpoint**: types of elements and relationships and other meta-information
- **Architectural view**: an instance of a viewpoint for a particular system
- **Architectural pattern**: defines how types of elements and relationships work together to solve a problem

- A particular system implements one or more architectural patterns
- An architectural pattern is classified in a particular view when the pattern implementation shows up in that view
- Patterns have one primary view, the most suitable - sometimes patterns show up in more that one view

# Layered, data flow and data centered


Layered view
can be used for communication between layers
layer used for indirection and add-on tasks
Layers
Indirection Layer
alternative for separating higher-level from lower-level responsibilities, also used internally
alternative for separating higher-level from lower-level responsibilities, also used internally
Data-centered view
Pipes and Filters
Shared Repository
can be used to allow for data sharing between filters
alternative when stream-based processing is needed
variant when repository needs to actively inform subscribers
variant when no deterministic approach to a solution is known or feasible
Batch Sequential
Implicit Invocation
uses
Active Repository
Blackboard
Data flow view

# Layers



# Layers
# What and Why

- ✓ High-level components depend on low-level components to do their job
- ✓ Decoupling the components is crucial to support modifiability, portability and reusability
- ✓ Components also require horizontal structuring but that is orthogonal to the vertical layering
- ✓ Layer n provides a set of services to layer n+1 and uses the services of layer n-1
- ✓ Between adjacent layers a clearly defined interface is provided that remains stable, implementation details can change
- ✓ Within one layer components work on the same level of abstraction and interact

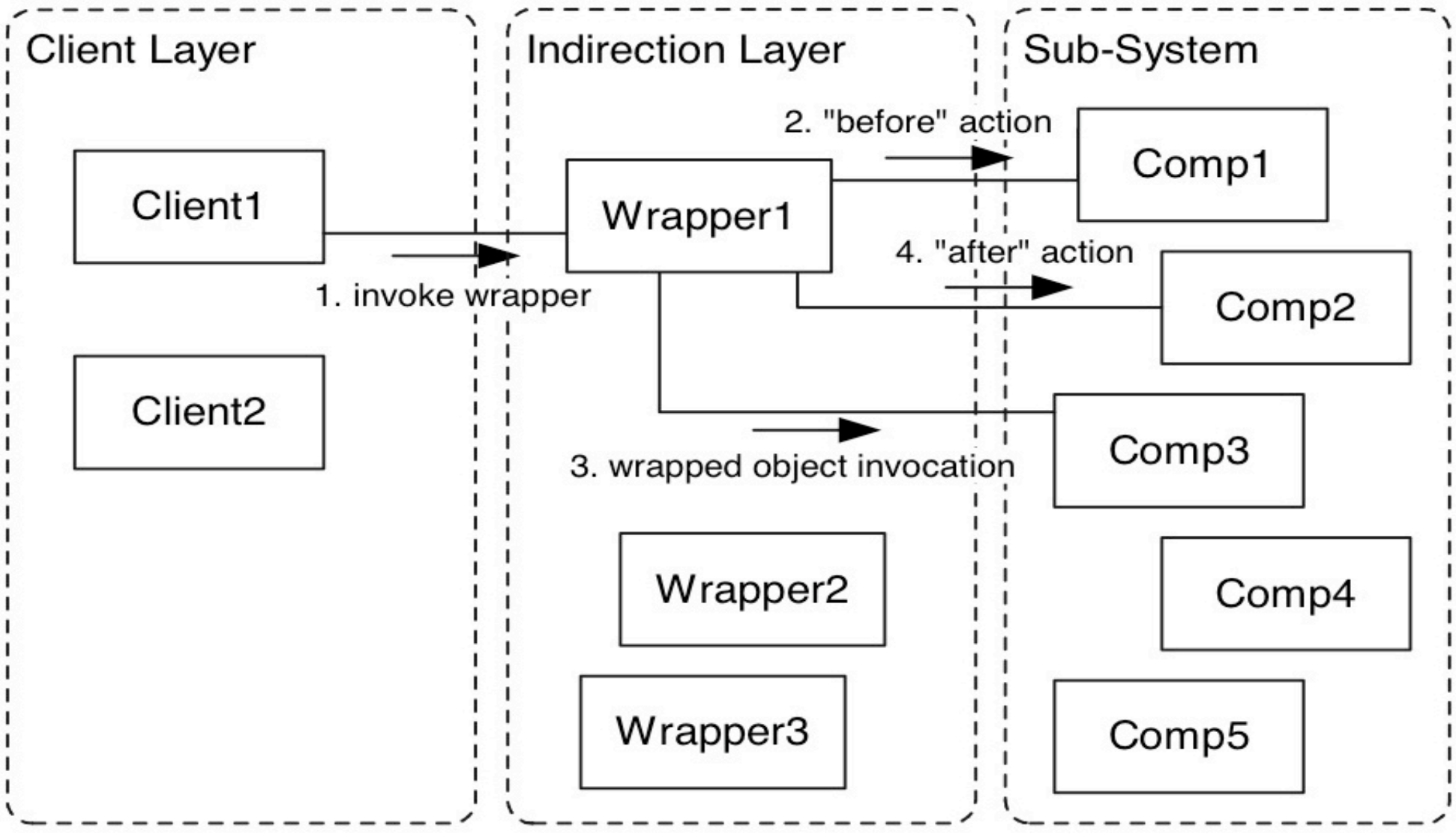
# Layers
# variations and relations

- In the pure form of the pattern, layers should not be by-passed
- Two adjacent layers can be considered as a *client-server* pair
- A common use is the *three tier* architecture with a back-end server (often a data base), a business logic layer, and a presentation layer
- *Microkernel* has also 3 layers: external servers, the microkernel, and internal servers
- *Presentation-abstraction-control* also enforces layers: a top layer with one agent, several intermediate layers with numerous agents and one bottom layer with the "leaf" agents
- In *indirection layer* a special layer hides the details of a subsystem and
provides acces to its services

# Layers
# uses and examples

- Layered communication (OSI, TCP/IP)
- Hardware interface libraries
- Information systems (three-tiers)
- In combination with Client-Server (Distribute layers)

# Indirection Layer



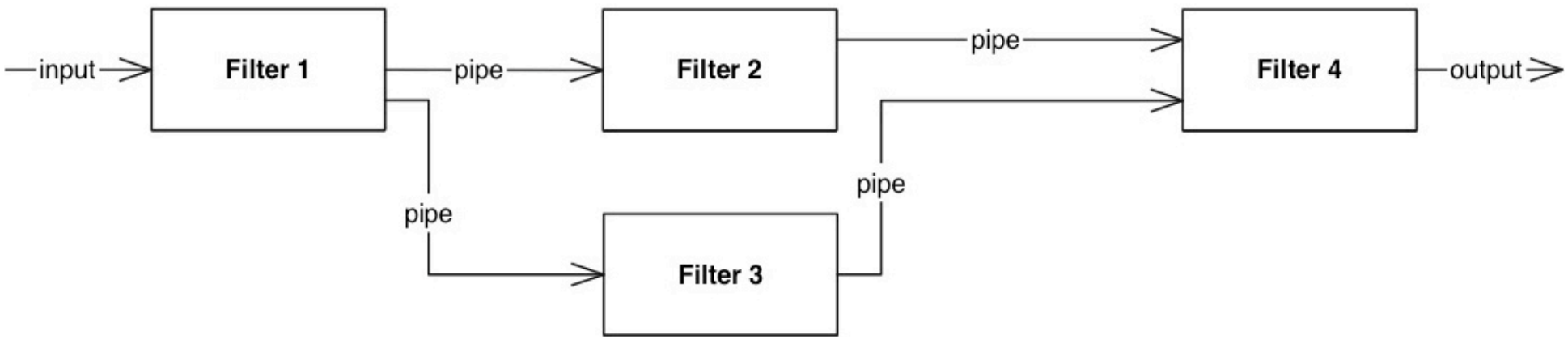
# Indirection layer
# what and why

- ✓ A sub-system should be accessed by one or more components but direct access to the sub-system is problematic
- ✓ Appears at different levels of scale: between 2 components in one environment (for example to avoid hard wiring), between components in different languages, between components in different systems (e.g. when a legacy system is accessed)
- ✓ The indirection layer wraps all accesses to the sub-system and should not be by-passed
- ✓ The indirection layer can perform additional tasks while deviating invocations such as converting or tracing the invocations

# Indirection Layer variations and relations

- ✓ *Indirection layer* can be integrated in the subsystem (as in *virtual machine*) or be an independent entity (as in the *adapter* of *facade* design patterns). The pattern can therefore be through of as either a 2- layer or a 3-layer pattern.
- ✓ *Indirection layer* is a foundation for the architectures of *virtual machine*, *interpreter*, and *rule-based system.* These patterns provide an execution engine for a language defined on some platform. They introduce an indirection layer between the instructions of that language and the instructions of the platform.
- ✓ *Reflection* can also be implemented with indirection layer. Indirection layer can capture all invocations of the components in the system and record the current structure and behaviour of the system for later use in the reflection API

# Pipes and Filters



# Pipes and Filters
# what and why

- ✓ Consider a task that can be sub-divided into a number of smaller tasks which can be defined as a series of independent computations
- ✓ With the pipes and filter pattern each sub-task is realized as an independent component, named *filter*
- ✓ Filters are connected flexibly using pipes, they may specify the type of input they expect and the type of output they produce but they are never aware of the identity of the adjacent filters
- ✓ Each *pipe* realizes a stream of data between two adjacent components
- ✓ Filters consume and produce data incrementally, pipes act as buffers between adjacent filters
- ✓ Forks, joins and feedback loops are allowed, the filters can potentially work in parallel

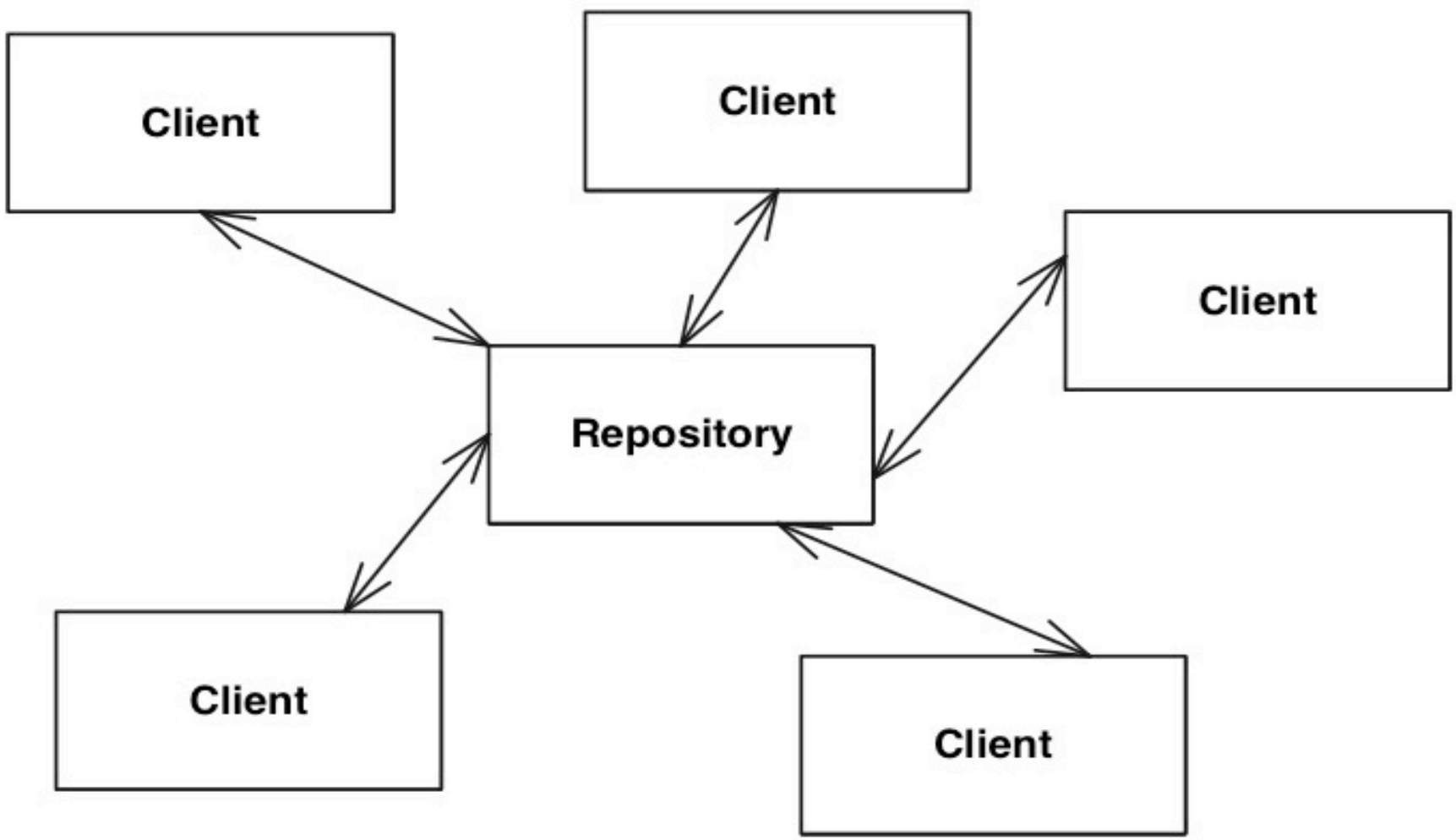
# Pipes and Filters
## variations and relations

- ✓ *Pipelines* restrict topology to a linear sequence of filters
- ✓ *Bounded pipes* restrict the amount of data in a pipe
- ✓ *Batch-sequential* is a degenerated case where each filter processes all of its input as a single entity, during each step a batch of data is processed and sent as a whole to the next step
- ✓ The pure form of *pipes and filters* entails that only adjacent filters can share data through their pipe. More relaxed forms of pipes and filters can be combined with data-centered architectures like *repository* and *blackboard* to allow data sharing between non-adjacent filters
- ✓ *Pipes and filters* can also be used for communication between layers if data flows through layers are needed

# Pipes and Filters
## examples and uses

- THE UNIX WAY
- Stream programming
- Compilers
- Signal and image processors
- Document management systems

# Shared Repository



# Shared Repositories
## what and why

- ✓ When data needs to be shared between components, passing the data along with the invocation might be inefficient for large data sets
- ✓ Long term persistency of data requires a centralized data management
- ✓ In the shared repository pattern one component is used as a central data store and accesses by other independent components
- ✓ This shared repository offers suitable means for accessing the data, I.e. a query language or a query API
- ✓ The shared repository must be scalable and must ensure data consistency. It must handle problems of resource contention for example by locking data.
- ✓ The shared repository might also offer additional services such as security and transaction management

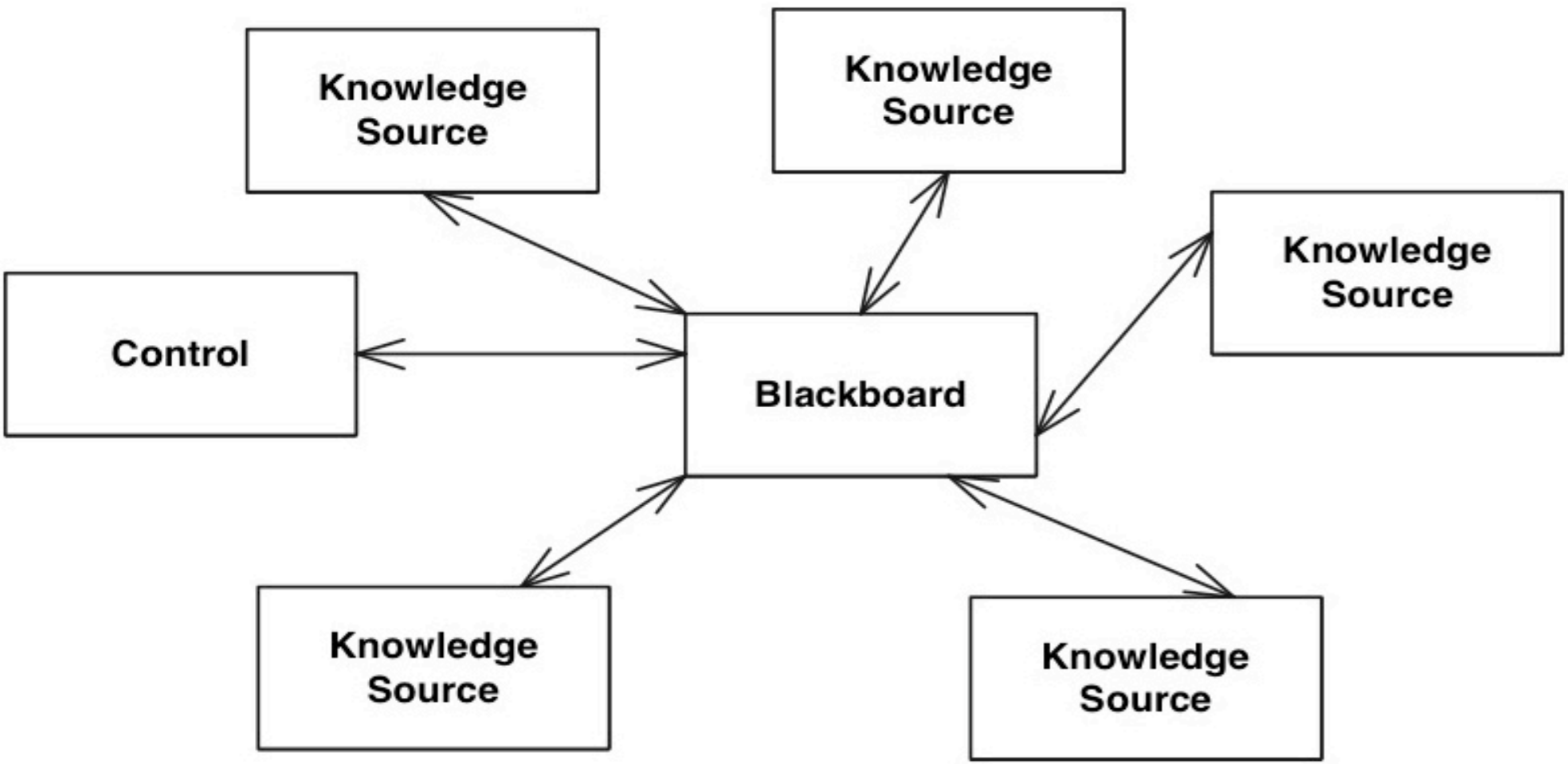
# Shared Repository
## variations and relations

- ✓ *Shared repository* can be used in *pipes and filter* to allow for data sharing between non-adjacent filters
- ✓ A shared repository where all the components are independent can be considered as *client-server* with the data storage playing the server part.
- ✓ Similarly it can be considered a system of 2 layers where higher level clients access the lower level services of the shared repository
- ✓ *Active repository* and *blackboard* are common variants

# Active Repository what and why

- ✓ A system needs a shared repository where clients need to be informed of specific events in the shared repository such as changes of data or access of data
- ✓ Polling the shared repository does not work because it does not deliver information timely or inflicts too much overhead on system performance
- ✓ *Active repository* maintains a registry of clients and informs subscribers of specific events that happen in the shared repository through a notification mechanism
- ✓ The notification can be realised using ordinary explicit invocations but in most cases implicit invocations such as *publish-subscribe* are used

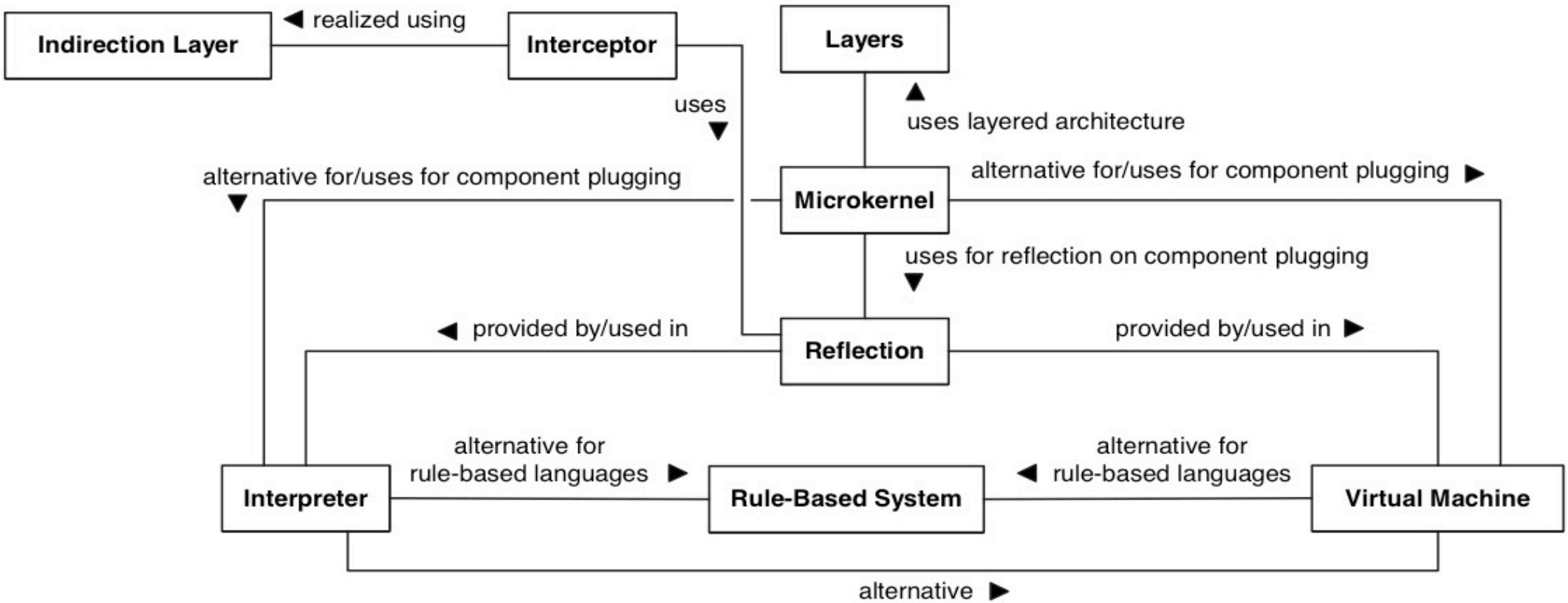
# Blackboard


Knowledge Source
Knowledge Source
Knowledge Source
Control
Blackboard
Knowledge Source
Knowledge Source

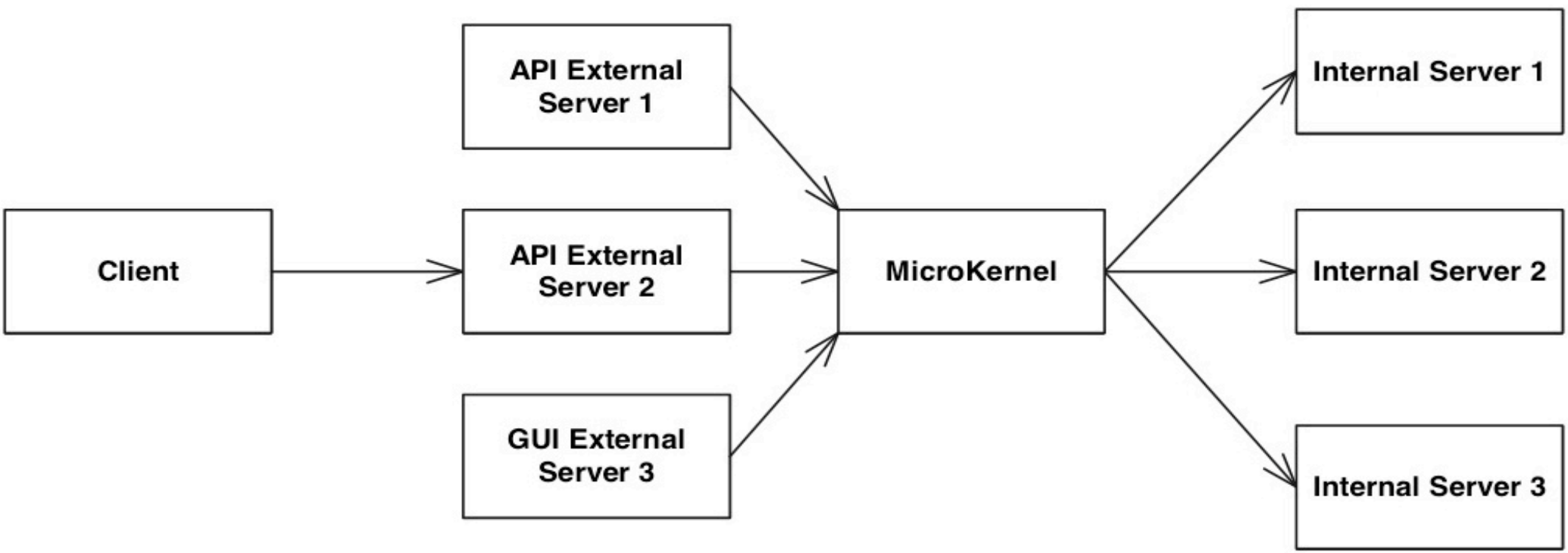
# Blackboard
## what and why

- ✓ When a shared repository is used in an immature domain in which no deterministic approach to a solution is known or feasible (image recognition, speech recognition)
- ✓ In a *blackboard* architecture the complex task is divided in smaller sub- tasks for which a solution is known and the blackboard is a shared repository that uses results of its clients for heuristic computation
- ✓ Each client can access the blackboard to see if new inputs are presented for further processing and to deliver new partial results after processing
- ✓ A control component monitors the blackboard and coordinates the clients according to the state of the blackboard (opportunistic reasoning

# Adaptation and language extension


Indirection Layer
realized using
Interceptor
Layers
uses
uses layered architecture
alternative for/uses for component plugging
Microkernel
alternative for/uses for component plugging
uses for reflection on component plugging
provided by/used in
Reflection
provided by/used in
alternative for
rule-based languages
alternative for
rule-based languages
Interpreter
Rule-Based System
Virtual Machine
alternative

# Microkernel



# Microkernel
## what and why

✓ Consider a system family where different versions of a system must be supported but the whole family should be realized with a common architecture to ease maintenance and foster reuse

✓ A *microkernel* realizes services that all systems needs AND provides a plug-and-play infrastructure for the system specific services

✓ *Internal servers* are used to realize version specific services and they are accessed through the microkernel

✓ *External servers* offer API's and other user interfaces to clients and they are used by clients to access the functionality of the system

✓ The microkernel pattern promotes flexible architectures which allow systems to adapt successfully

# Microkernel

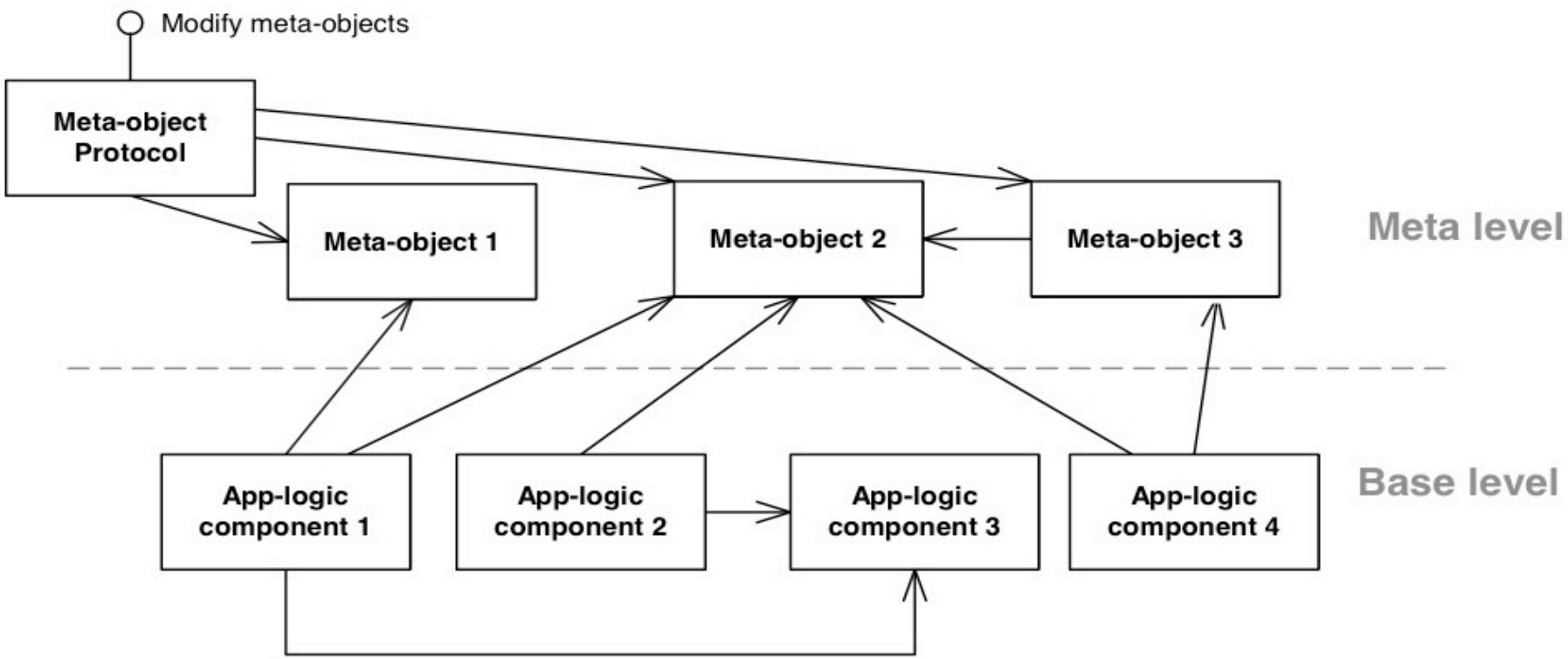
## variations and relations

- ✓ *Microkernels* are often structured in layers: the bottom layer implements an abstraction of the system platform and is in fact an indirection layer
- ✓ A microkernel introduces an indirection that can be used in client-server configurations for reasons of security or modifiability because all communication between clients and servers is mediated through the microkernel
- ✓ To develop distributed *microkernel* architectures a combination with the broker pattern can help to hide communication details between clients that request services and servers that implement them
- ✓ Reflection can be useful to support the plug-and-play of components, i.e. find out which components are currently composed in which way

# Microkernels
# uses and examples

- Operating systems (*)
- Distributed operating systems
- IDEs
- Server architectures
- Software product lines
- Games

# Reflection


Modify meta-objects
Meta-object Protocol
Meta-object 1
Meta-object 2
Meta-object 3
Meta level
App-logic component 1
App-logic component 2
App-logic component 3
App-logic component 4
Base level

# Reflection
# why and what

✓ Software systems constantly evolve over time and coping with unanticipated change is also required

✓ In a *reflection* architecture all structural and behavioral aspects of a system are stored in meta- objects and separated from the application objects

✓ Application objects can query the meta-objects to execute their functionality and the meta-objects can change at any point in time

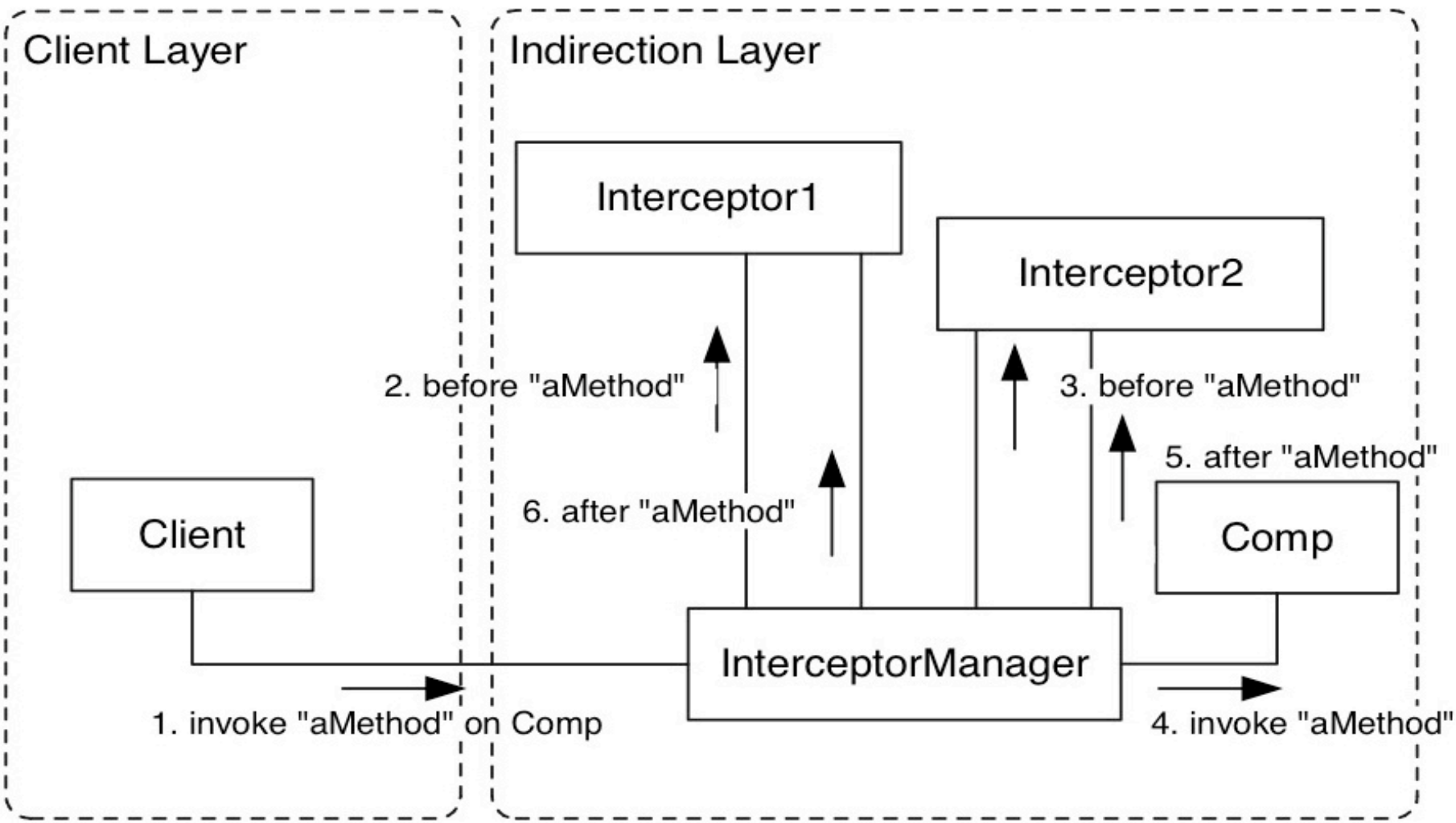
# Reflection
## variants and relations

- A *reflection* pattern is typically organized in 2 *layers*: a meta level and a base level
- *Reflection* is used in the context of aspect-oriented composition frameworks, i.e. the *introspection option*
- Indirection layer is a more general pattern than reflection. It can be used to build a reflection infrastructure
- In cases where an adaptable framework is needed to accommodate future services, the *interceptor* pattern is appropriate

# Reflection
## uses and examples

- ✓ JAVA reflection API
- ✓ CLOS meta-object protocol
- ✓ Smalltalk meta-objects
- ✓ Integrated Development Environments
- ✓ Adaptive Object-Model architectures
- ✓ Self-healing systems

# Interceptor



# Interceptor
# why and what

- ✓ A framework offers reusable services to applications that extend it. These services need to be updated as the application domain matures. The framework developer cannot anticipate all future updates. The application developer can not make changes to the framework
- ✓ An interceptor is a mechanism for transparently updating the services offered by a framework
- ✓ Application developers can register with the framework any number of interceptors that implement new services
- ✓ The framework provides the application with a means to introspect on the frameworks behavior

# Interceptor variants and relations

- ✓ The *interceptor* pattern can be realized with an *indirection layer* or one of its variants. Incoming events are then re-routed through the indirection layers that consists of several interceptors before they are dispatched to the intended receiver
- ✓ *Interceptor* can use *reflection* to query the framework and provide the necessary information to the interceptors
- ✓ A variant in the context of aspect-oriented composition is a *message interceptor*
- ✓ A variant in the context of middleware architectures is a *invocation interceptor*

# Interpreter
# why and what

- ✓ Interpret at run-time a program or script offered in some language syntax and grammar
- ✓ An *interpreter* provides both parsing facilities and a run-time environment
- ✓ The programs or scripts are portable to different realizations of the interpreter
- ✓ Some interpreters use optimizations such as on-the-fly byte- code compilers thus realizing elements of a virtual machine

# Virtual Machine
# why and what

- ✓ An efficient execution environment for some programming language is needed
- ✓ Virtual machine defines a simple machine architecture on which not machine code but an intermediate form called byte-code can be executed
- ✓ A program is compiled into that byte-code and the executed on the virtual machine
- ✓ The virtual machine can be realized on different platforms so that the byte-code becomes portable between such platforms
- ✓ The virtual machine redirects invocations from a byte-code layer into an implementation layer for the commands of the byte-code

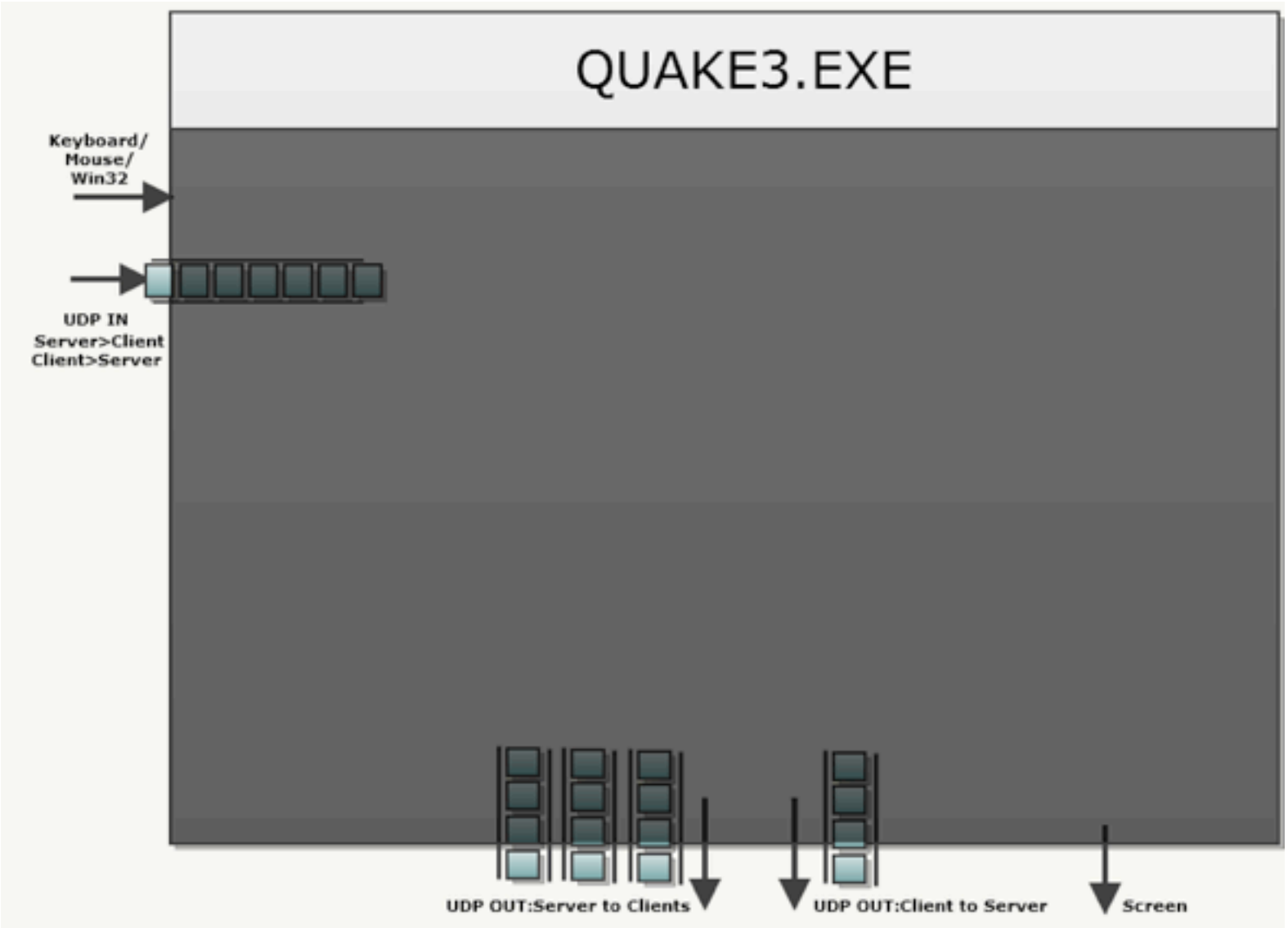

# Intermezzo

# The architecture of Quake 3

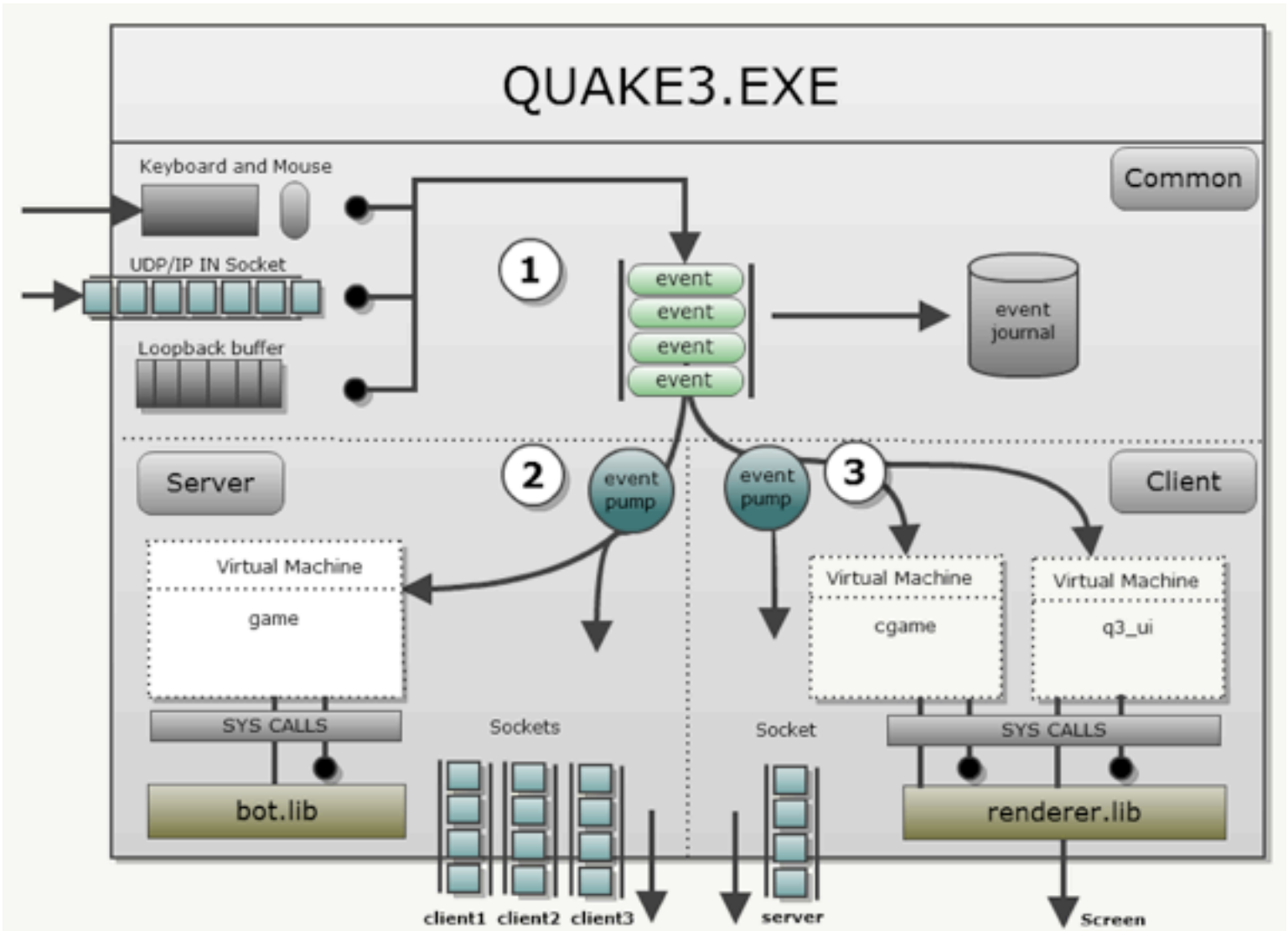
- Source code released under GPL2 in 2005 (Git https://github.com/id-Software/Quake-III-Arena.git )
- Circa 350KLoCs
- 70%-30% split of code between Engine (quake3.exe) and associated tooling (Editor, Preprocessors, LCC)

http://fabiensanglard.net/

# The architecture of Quake 3



# The architecture of Quake 3

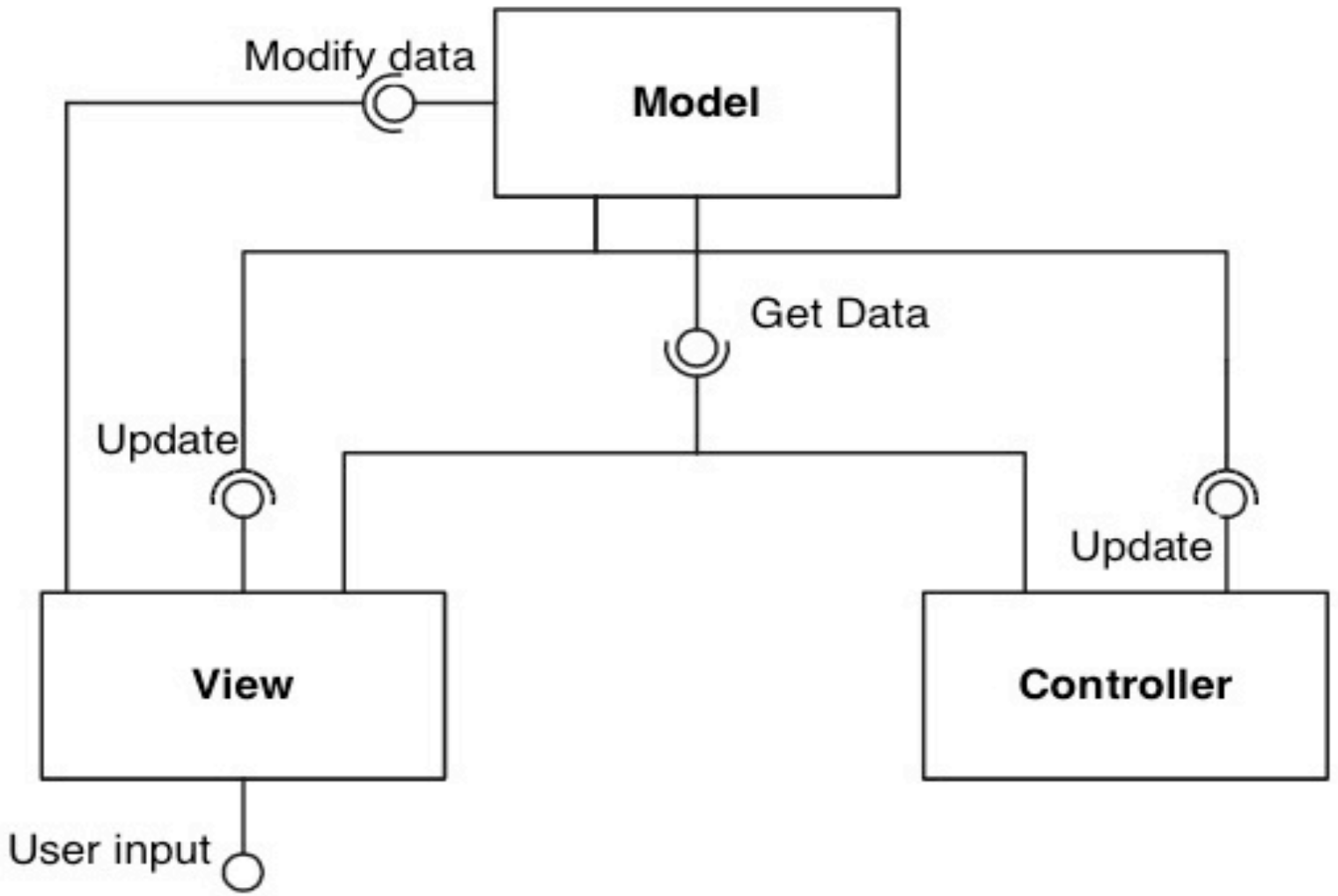


# Rule-based System why and what

- ✓ Solve problems that are better formulated as a set of declarative if-then rules or constraints than in an imperative programming style
- ✓ A rule-based systems consists of 3 things: a **rule-base** (a set of condition-action pairs for a generic problem in a given application domain), a **working memory** (a set of data about the problem at hand), and an **engine** that acts on them
- ✓ The engine matches facts to rules, does conflict resolution when more than one rule has its conditions fulfilled, executes the selected rule and by doing so asserts new facts or changes existing facts in the working memory which in turn will trigger other rules

# User interaction

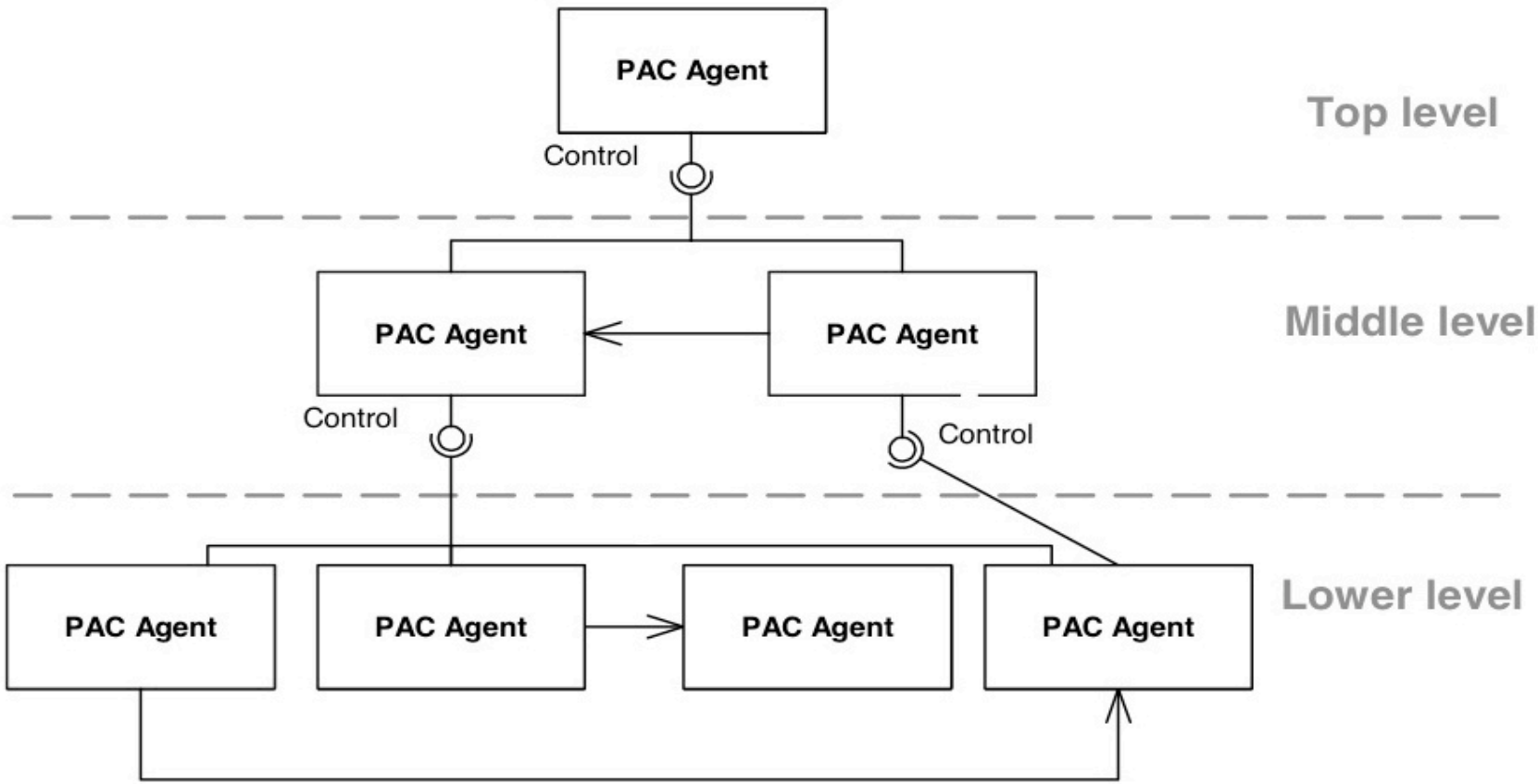
# Model View Control



# Model View Control
# why and what

- ✓ A system may offer multiple user-interfaces, each user-interface depicts all or part of some application data, changes to the data should be automatically reflected to all the user-interfaces and the user-interfaces must be easily modified without affection the application logic
- ✓ The system is divide in three different parts: the model encapsulates application data and the logic that manipulates the data, one or multiple views that display a specific portion of the data to the user, a controller associated with each view that receives user input and translates it to a request for the model
- ✓ Views and controllers constitute the user interface, users interact strictly through the views and their controllers

# Presentation Abstraction Control



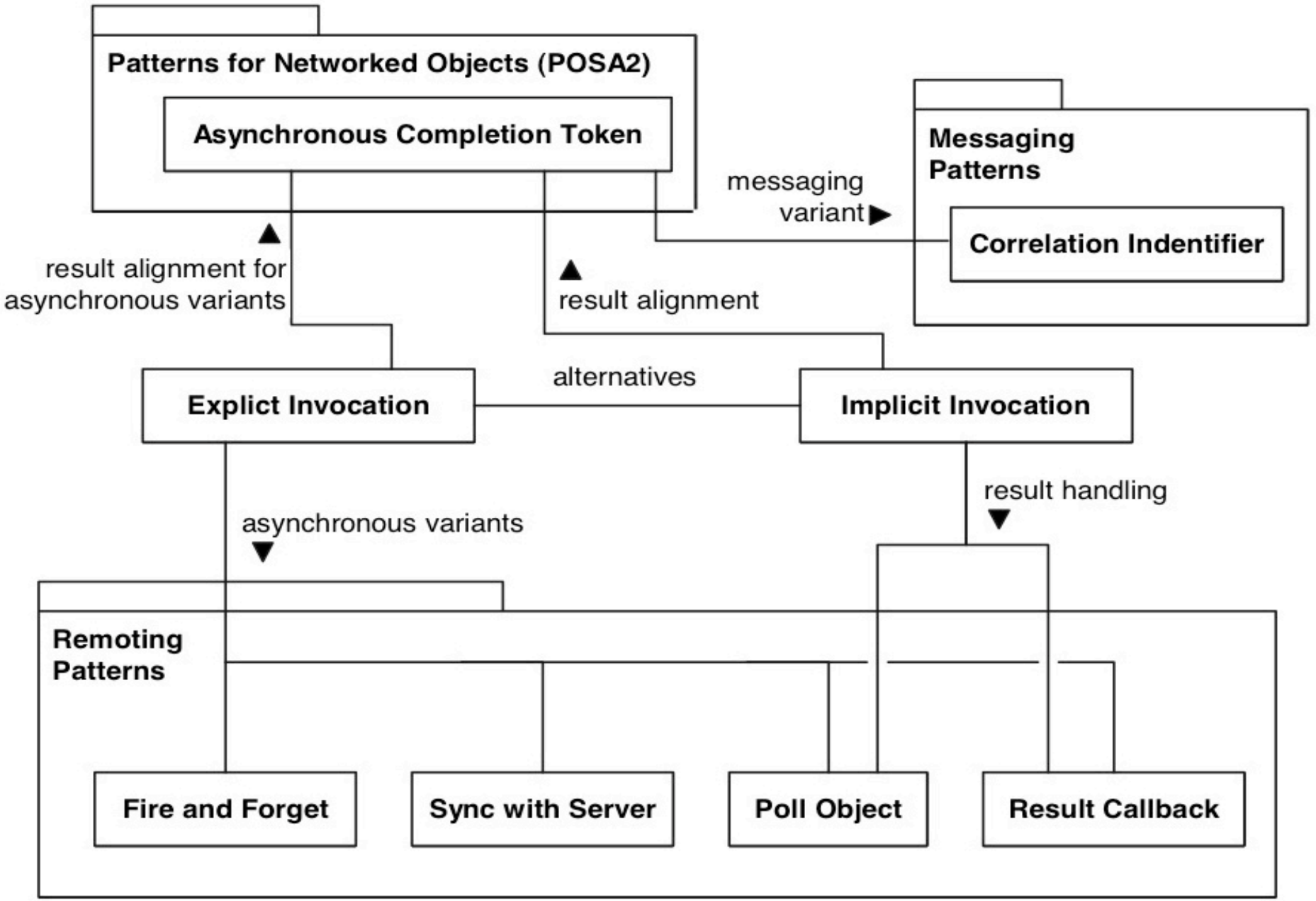
# Presentation Abstraction Control
## why and what

✓ An interactive system may offer diverse functionalities that need to be presented to the user through a coherent and consistent user interface. The various functionalities require a custom user-interface and also need to interact with other functionalities to achieve a greater goal

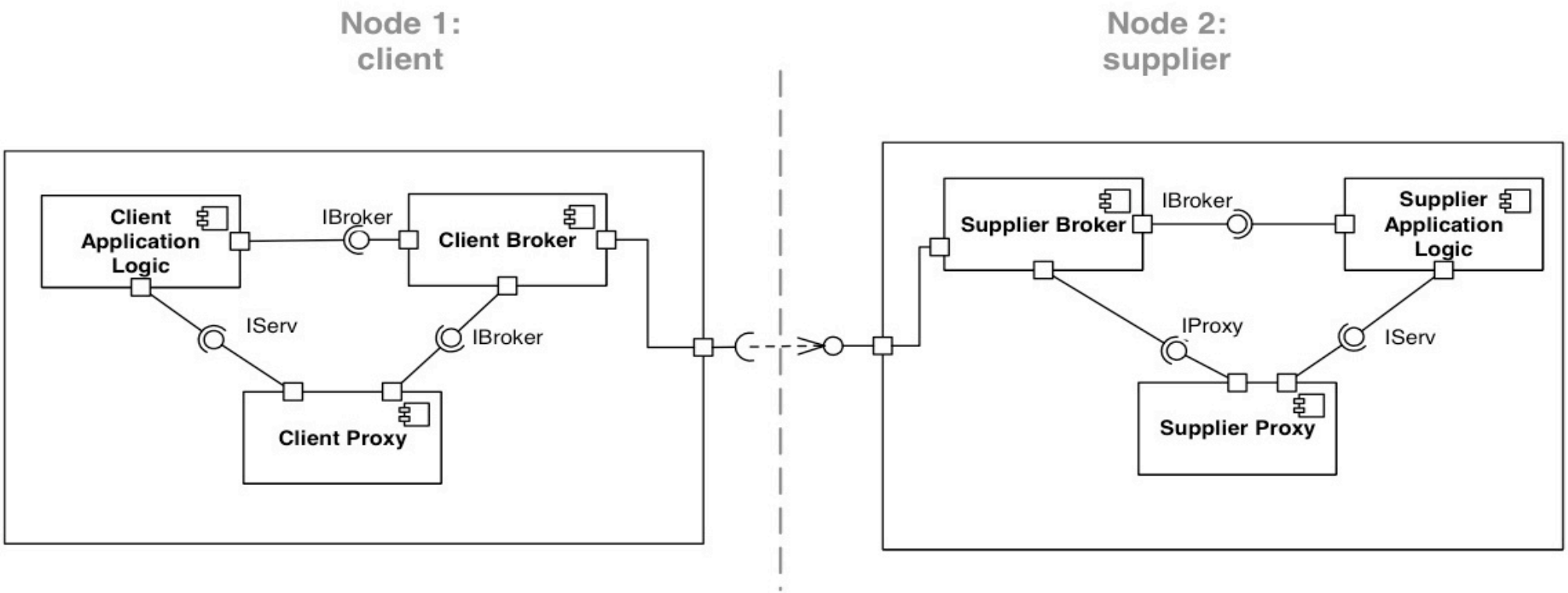
✓ The system is divided into a tree-like hierarchy of agents. The leave agents are responsible for specific functionalities and offer a specific user-interface, middle-layer agents combine the functionalities of lower-level agents and at the top of the tree there is one agent that orchestrates the middle-layer agents to offer the collective functionality.

✓ Each agent is comprised of three parts: a *Presentation* takes care of the user- interface, an *Abstraction* maintains the application data and the logic that modifies it, a *Control* mediates between Presentation and Abstraction and handles all communication with the Controls of other agents

# Basic component Interaction


Patterns for Networked Objects (POSA2)
Asynchronous Completion Token
Messaging Patterns
Correlation Indentifier
messaging variant
result alignment for asynchronous variants
result alignment
alternatives
Explict Invocation
Implicit Invocation
result handling
asynchronous variants
Remoting Patterns
Fire and Forget
Sync with Server
Poll Object
Result Callback

# Explicit Invocation



# Explicit Invocation why and what

- ✓ Client needs to invoke a service defined in another component or supplier
- ✓ An explicit invocation allows the client to invoke services on a supplier when the client knows the exact location, name and parameters of the service when initiating the invocation
- ✓ The explicit invocation mechanism performs the invocation and delivers the result to the client
- ✓ Two main variant are synchronous explicit invocation (the client blocks and waits for the result) and asynchronous explicit invocation (the client continues with its work and the result is delivered when it is computed

# Explicit Invocation variants and relation

✓ Four patterns describe different variants for asynchronous invocation in distributed systems :

- **Fire and Forget:** best effort semantics, does not convey results or acknowledgements
- **Syn with Server**: server sends an acknowledgement back to the client as soon as the operation invocation arrives on the server side, does not convey results
- **Poll Object:** clients poll or query for the results
- **Result Callback**: server notifies the client that the result is ready

✓ These patterns can be hard-coded (optimization) or a broker that provides a reusable implementation of one such pattern can be used

# Implicit Invocation why and what

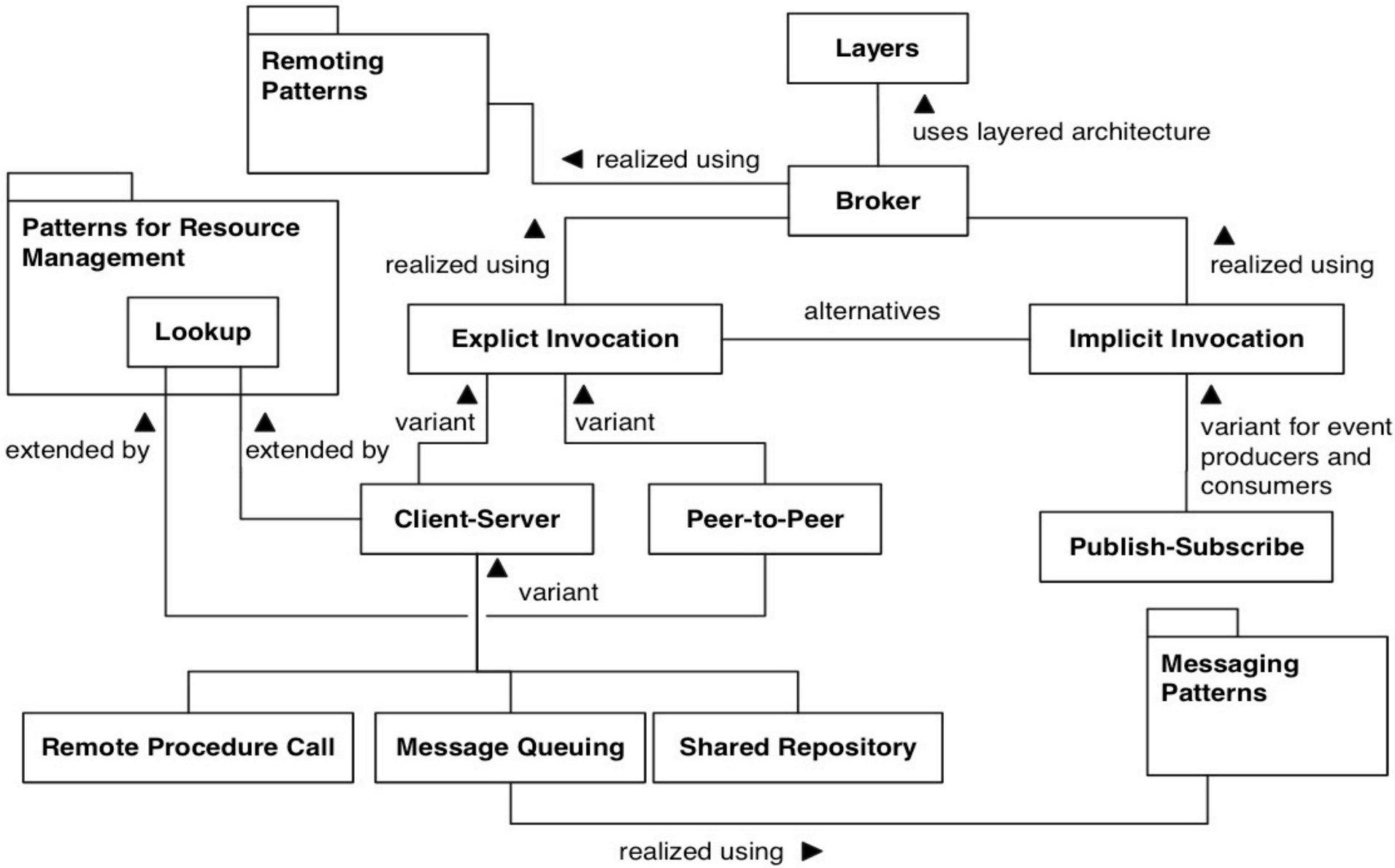
✓ A client needs to invoke a service defined in another component and the client must be decoupled from the supplier (client does not know the supplier that will serve the invocation, client does not need the result right away, client does not initiate the invocation, supplier is not ready to reply until some condition is met, clients may be added and removed dynamically, client does not know whether supplier is up and running or down, client and supplier are part of dissimilar systems and thus the invocation must be transformed, queued, or otherwise manipulated during delivery)

✓ In the implicit invocation pattern the invocation is indirect through a special mechanism such as Publish-Subscribe, Broadcast or Message Queueing that decouples clients from suppliers

# Implicit Invocation variants and relation

- ✓ The synchronization between Model, View and Control in the MVC pattern is an example of implicit invocation
- ✓ Implicit invocation can be synchronous or asynchronous but are most often asynchronous - the patterns for result handling apply as well
- ✓ In explicit invocation the invocation is deterministic from client to supplier, in implicit invocation the trigger may happens randomly (e.g. through an event)
- ✓ Same as in explicit invocation as broker can be introduced to hide the network details and allow the components to contain application logic only
- ✓ A broadcast mechanism broadcasts an invocation (over a network or a software bus) because the location of the invocation receiver is not known. Used for example in peer-to-peer systems for looking up initial references.
- ✓ The Message Queuing pattern queues invocations and results to increase delivery reliability and to handle temporal outages of the supplier

# Component Interaction and Distribution


Remoting Patterns
Layers
uses layered architecture
realized using
Broker
Patterns for Resource Management
Lookup
realized using
realized using
alternatives
Explict Invocation
Implicit Invocation
variant
variant
variant for event producers and consumers
extended by
extended by
Client-Server
Peer-to-Peer
Publish-Subscribe
variant
Messaging Patterns
Remote Procedure Call
Message Queuing
Shared Repository
realized using

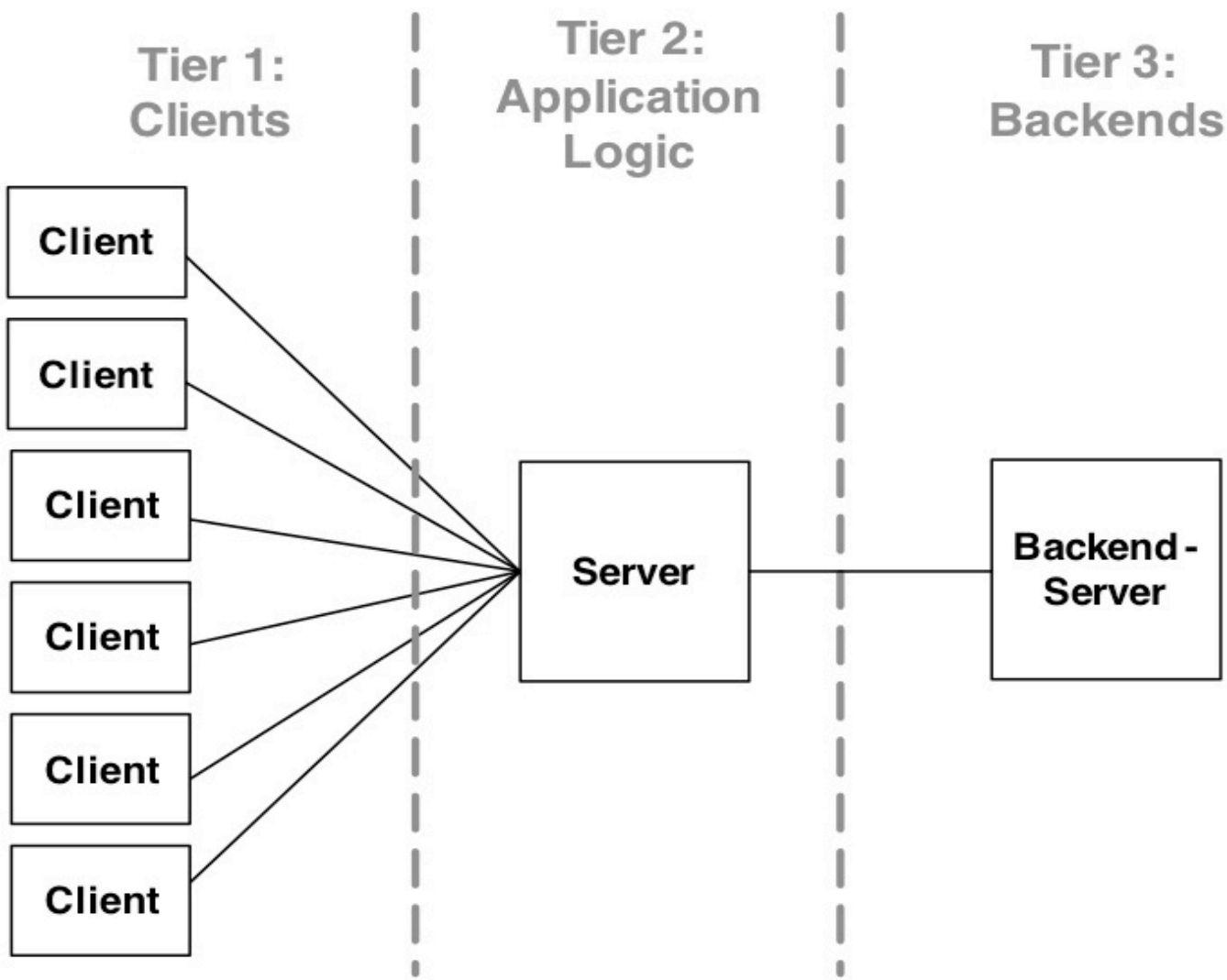
# Client-Server why and what

- ✓ Two components need to communicate, they are independent, running in different process spaces or distributed in different machines
- ✓ The two components are not equal peers, one is initiating the communication asking for a service that the other provides
- ✓ Multiple components may ask the same service from the second component; this component must be able to cope with numerous request
- ✓ The Client-Server pattern distinguishes two types of components:
  - The client request information or services from a server; the client needs to know how to access the server and the server';s interface
  - The server responds to the requests of the clients, it does not know the client before the interaction takes place

# Client-Server variants and relations

- ✓ Both client and server must implement tasks such as security and transaction management adding complexity to simple explicit invocation
- ✓ Sophisticated, distributed Client-Server architectures use the Broker pattern to make the complexity of distributed communication manageable
- ✓ Arbitrary complex architectures can be built by introducing multiple client-server relationships: n-tier architectures
- ✓ 3-tier architecture has a
  - Client tier, responsible for presenting data, receiving user input, and controlling the user interface
  - Application logic tier responsible for implementing the application logic
  - Back-end tier responsible for back-end services such as data-storage or access to a legacy system
- ✓ Client-Server and Layer combine well: the tiers in an n-tier system can be perceived as layers but also both the client and the server can be individually layered (OSI)
- ✓ Shared Repository and Blackboard can be perceived as Client-Server

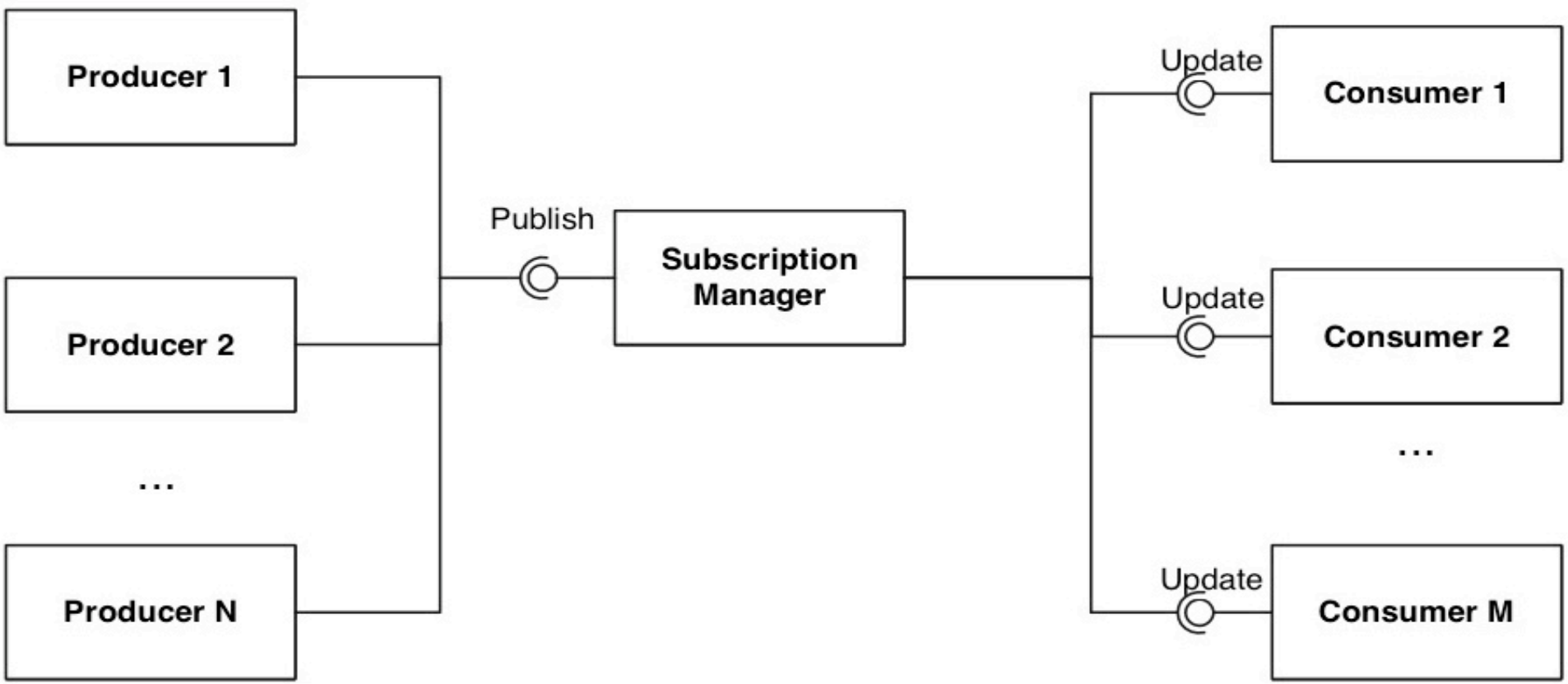
# 3-tier Client-Server



# Peer-to-Peer
# why and what

- ✓ Two components need to communicate, they are independent, running in different process spaces or distributed in different machines
- ✓ The two components are equal peers, they both can provide and consume services from each other
- ✓ In the Peer-to-Peer pattern each component has equal responsibilities, it may act both as a client and as a server
- ✓ Peer-to-Peer networks consist of a dynamic number of components, components know how to access the network and in order to join they must get an initial reference to this network (through a bootstrapping mechanism such a providing public lists of dedicated peers or broadcast messages in the network announcing peers

# Publish-Subscribe



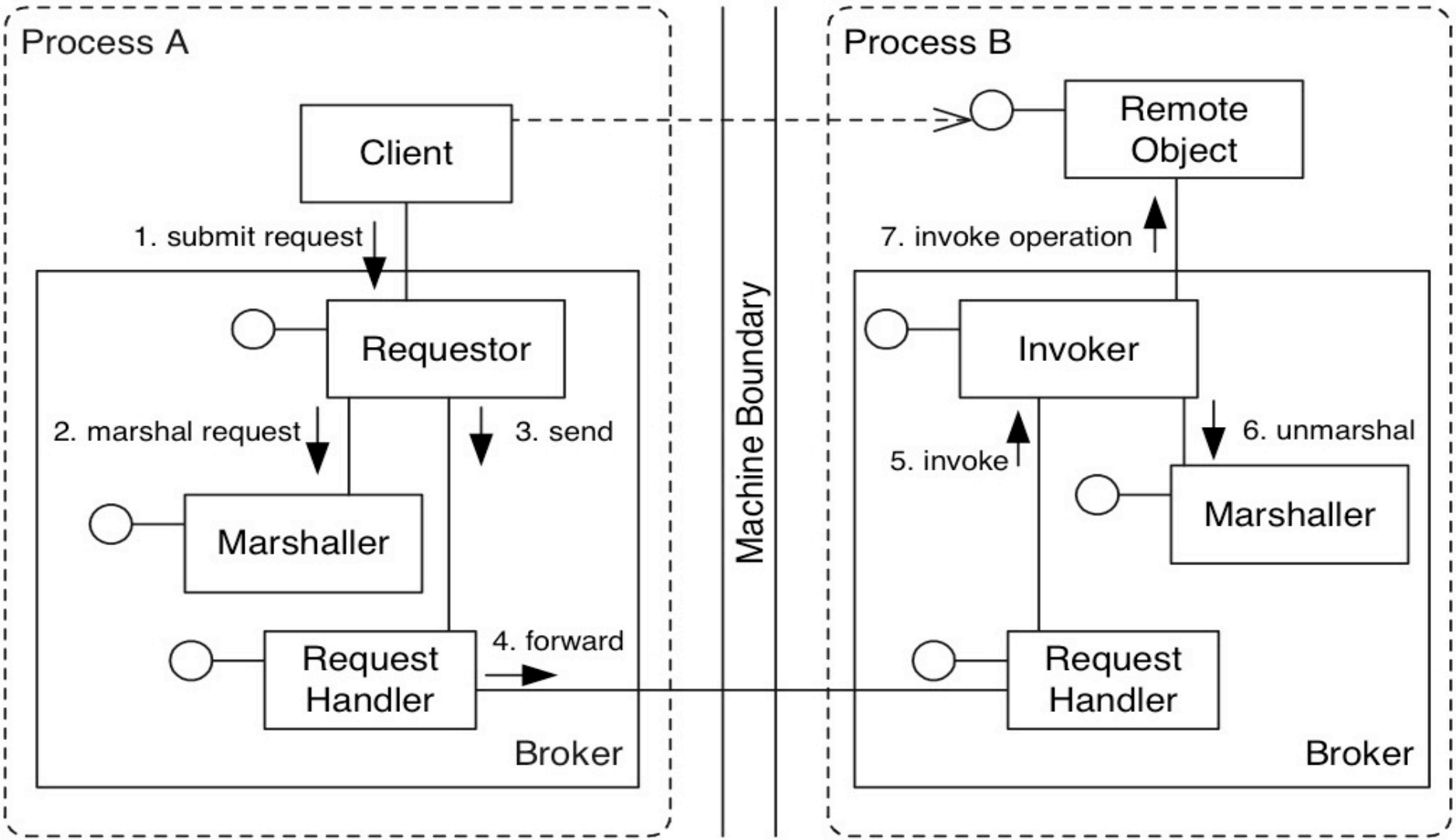
# Publish-Subscribe why and what

- ✓ A component should be informed of a specific run-time event; sometimes a number of components should be actively informed (announcement or broadcast), in other cases only a specific component is interested in the event
- ✓ Publish-Subscribe allows event-consumers (subscribers) to register to specific events and event-producers to publish or raise specific events
- ✓ The Publish-Subscribe mechanism is triggered by the event-producers an automatically executes a callback operation to the event-consumers
- ✓ Publish-Subscribe can be implemented as an independent subscription manager

# Publish-Subscribe variations and relations

- ✓ In a local context the Observer pattern implements Publish-Subscribe as part of the 'subject'; in a remote context it is a patterns on its own or it is used in Message Queuing
- ✓ Most GUI frameworks are based on Publish-Subscribe
- ✓ Publish-Subscribe can be used in the context of an Active Repository
- ✓ Publish-Subscribe can be used to realize distributed Client-Server and Peer-to-Peer architectures because it allows to bridge the asynchronous network events and the synchronous processing model of the server

# Broker


Process A
Client
1. submit request
Requestor
2. marshal request
3. send
Marshaller
Request Handler
4. forward
Broker
Machine Boundary
Process B
Remote Object
7. invoke operation
Invoker
5. invoke
6. unmarshal
Marshaller
Request Handler
Broker

# Broker
# why and what

- ✓ Distributes software systems face challenges: communication across unreliable networks, integration of heterogeneous components, efficient use of network resources, etc.
- ✓ A Broker separates the communication facilities in a distributed system from the application functionality, it hides and mediates all communication between the components of a system
- ✓ A Broker consist of: a client-side Requestor to construct and forward invocation, a server-side Invoker that invokes the operations of the target remote object, and a Marshaller on each side to handle the transformation of the request and replies form programming language native data-types into a byte array that can be transported over the