Software Architectural Patterns

Lecture 08

BIL428 Software Architectures

Mustafa Sert
Asst. Prof.
msert@baskent.edu.tr

Department of Computer Engineering, Başkent University
Ankara 06810 TURKEY

Agenda..

- Categorization of Software Patterns
- Architecture Design Patterns
  - Layers Pattern
  - Pipes and Filters
  - Repository/Blackboard
  - Broker
  - Proxy
  - Model View Controller (MVC)

Categorization of Software Patterns

- SW Architectural Patterns (Styles)
- SW Design Patterns
  - Provides schemes for refining the architecture
- Others
  - Organizational, Analysis, UI Design, etc.

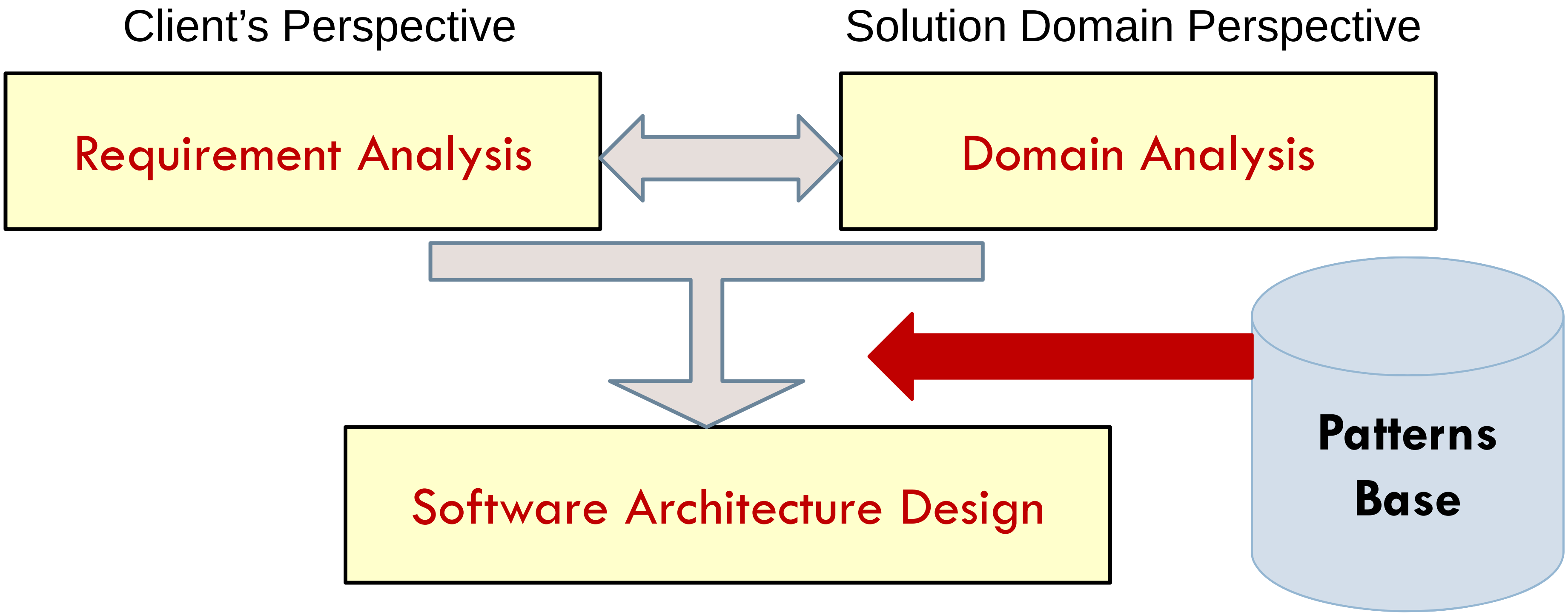
SW Architecture Patterns

- Express a fundamental structural organization schemes for software systems
- Provide a set of predefined subsystems
- Specify their responsibilities
- And includes rules and guidelines for organizing the relationships btw them

SW Design Patterns

- A design pattern provides a scheme for refining the architectural entities of a SW system
- It describes commonly recurring structure of communicating components that solves a general design problem within a particular context
- Design patterns are generally classified into
  - Creational Patterns: Abstract factory, Builder, Singleton, Prototype, …
  - Structural Patterns: Adapter (2), Bridge, Decorator, Façade, Proxy, …
  - Behavioral Patterns: Iterator, Observer, Strategy, Mediator, Command, State, …
- **We have already studied the Adapter & Controller patterns**

Patterns in the Architecture Design Process



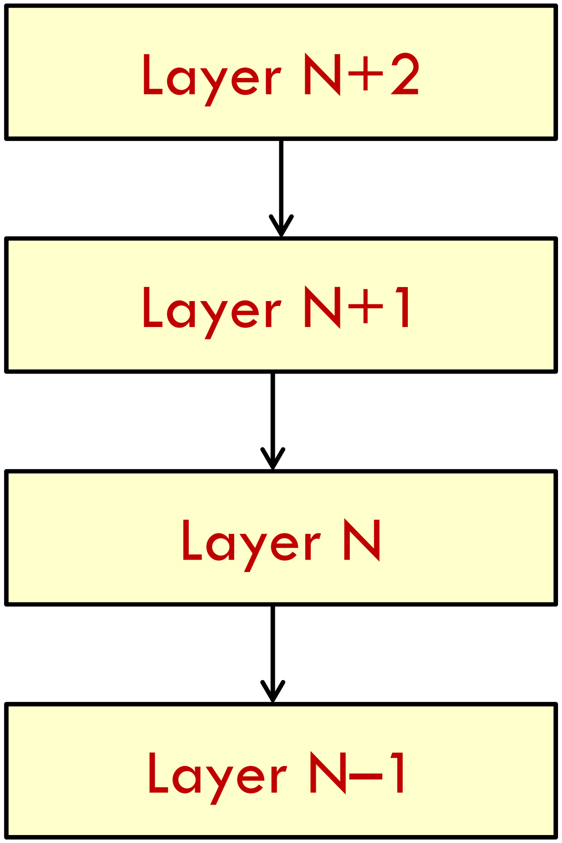
- □ Patterns support architectural qualities such as portability, maintainability, stability, etc.
- □ This might not be directly provided by the domain!

Layers Pattern – The Problems

- A large system, which is characterized with a mix of high- and low-level issues, where high-level (HL) operations rely on low-level (LL) issues
- The mapping of HL issues to LL issues is not straight-forward
- Source code change should be local and not compass through the whole system
- Portability is required
- Part of the system must be exchangeable
- Several external boundaries of the system are specified a priori (e.g., functional interface)
- Interfaces must be stable (may even be prescribed by a standard)

Layers Pattern – The Solution

- Structure the system in a set of layers
- Layer is a subsystem that uses the services of a lower abstraction to provide services to a higher abstraction
  - A layer can only depend on lower layers
  - A layer has no knowledge of higher layers
- **We have already studied the design issues of layers pattern in Layers Viewtype**



Benefits/Drawbacks of Layers Pattern

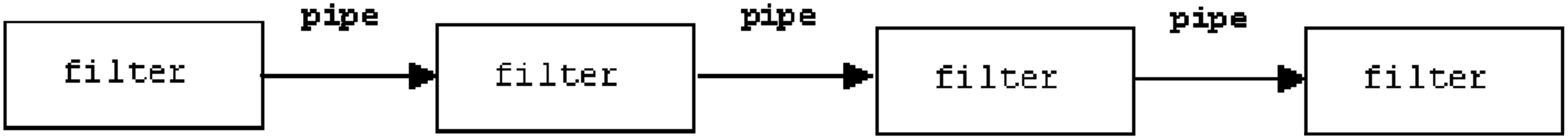
- **Benefits**
  - Reuse of layers
  - Support for standardization
  - Dependencies are kept local
  - Exchangeability
- **Drawbacks**
  - Cascades of changing behavior (e.g., crosscutting concerns)
  - Lower efficiency
    - Layered architecture is less efficient than a monolithic structure
  - Unnecessary work
    - Additional or duplicate work throughout the layers
  - Difficulty of establishing a correct granularity of layers
    - How many layers to select?

Pipes and Filters Pattern – The Problem

- System must process a stream of data
- System consists of a sequence of independent processing steps
- System has to be built by several developers
- Requirements are likely to change; processing steps might be exchanged or reordered
- Feature requirements must be possible by exchanging or reordering processing steps

Pipes and Filters Pattern – The Solution

- Divide the task into several sequential processing steps (filters):
  - Filters are independent entities for processing:
    - No shared state, no knowledge about other filters
  - Pipes connect entities and provide data flow
- Divide the task into several sequential processing steps (filters):

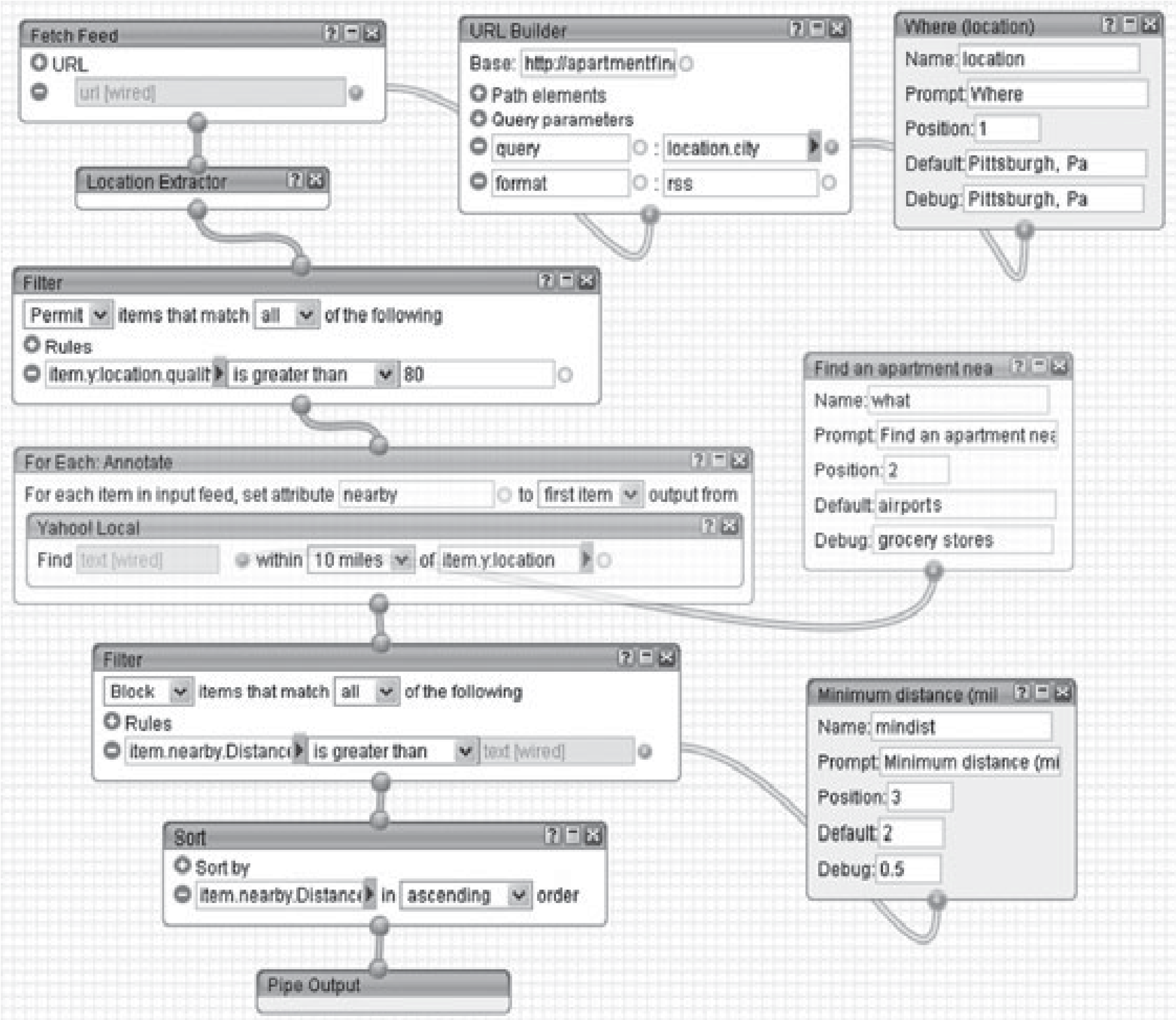


Pipes and Filters Pattern

- A **pipe** is a message queue. A message can be anything.
- A **filter** is a process, thread, or other component that perpetually reads messages from an input pipe, one at a time, processes each message, then writes the result to an output pipe.
  - Thus, it is possible to form **pipelines** of filters connected by pipes
- Pipelines architectures appear in many software contexts. (They appear in hardware contexts, too. For example, many processors use pipeline architectures.)
  - UNIX and DOS command shell users create pipelines by connecting the standard output of one program (i.e., cout) to the standard input of another (i.e., cin):

Pipes and Filters – Example

A Yahoo! Pipes application for finding apartments for rent near a given location (shown using the notation of the Yahoo! Pipes editor).



Fetch Feed
URL
url [wired]

Location Extractor

Filter
Permit items that match all of the following
Rules
item.y:location.qualit is greater than 80

For Each: Annotate
For each item in input feed, set attribute nearby to first item output from
Yahoo! Local
Find text [wired] within 10 miles of item.y:location

Filter
Block items that match all of the following
Rules
item.nearby.Distanc is greater than text [wired]

Sort
Sort by
item.nearby.Distanc in ascending order

Pipe Output

URL Builder
Base: http://apartmentfin
Path elements
Query parameters
query : location.city
format : rss

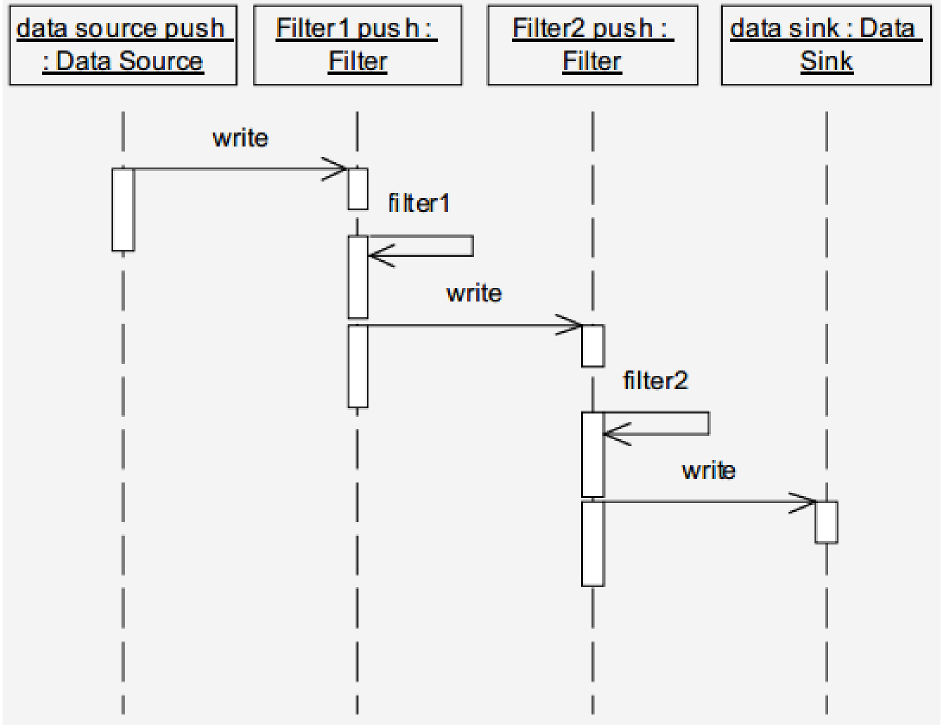
Where (location)
Name: location
Prompt: Where
Position: 1
Default: Pittsburgh, Pa
Debug: Pittsburgh, Pa

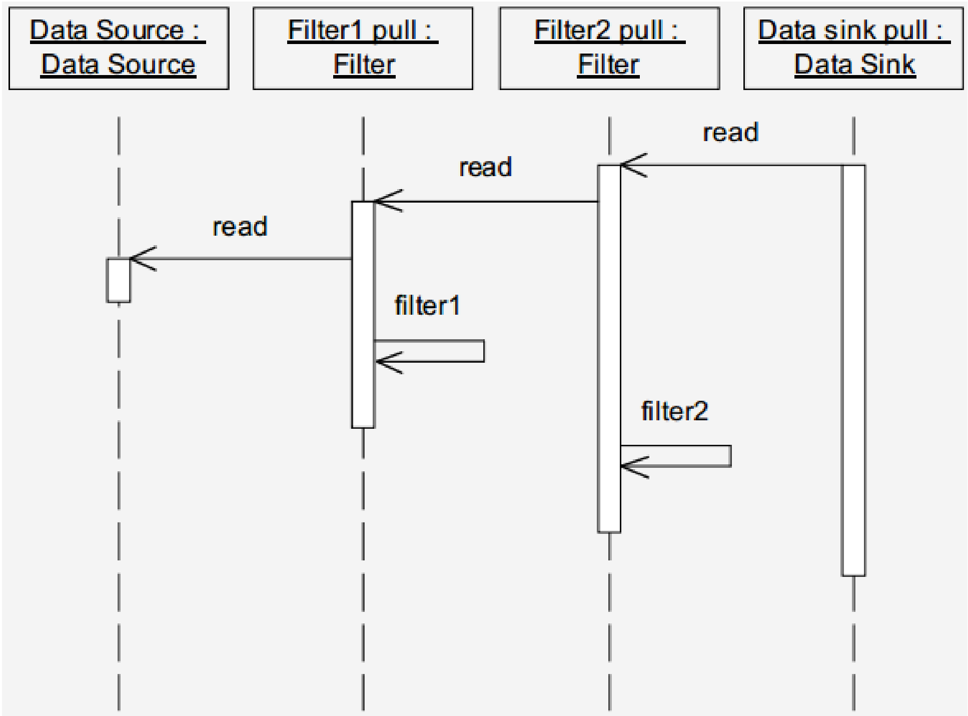
Find an apartment nea
Name: what
Prompt: Find an apartment nea
Position: 2
Default: airports
Debug: grocery stores

Minimum distance (mi
Name: mindist
Prompt: Minimum distance (mi
Position: 3
Default: 2
Debug: 0.5

Pipes and Filters – Dynamics

- **Push Pipeline**
  - In which activity starts with the data source
  - Filter activity is triggered by writing data to the passive filters
- **Pull Pipeline**
  - Control flow is started by the data sink calling for data
- **Parallel Pipeline** (see the next slide)

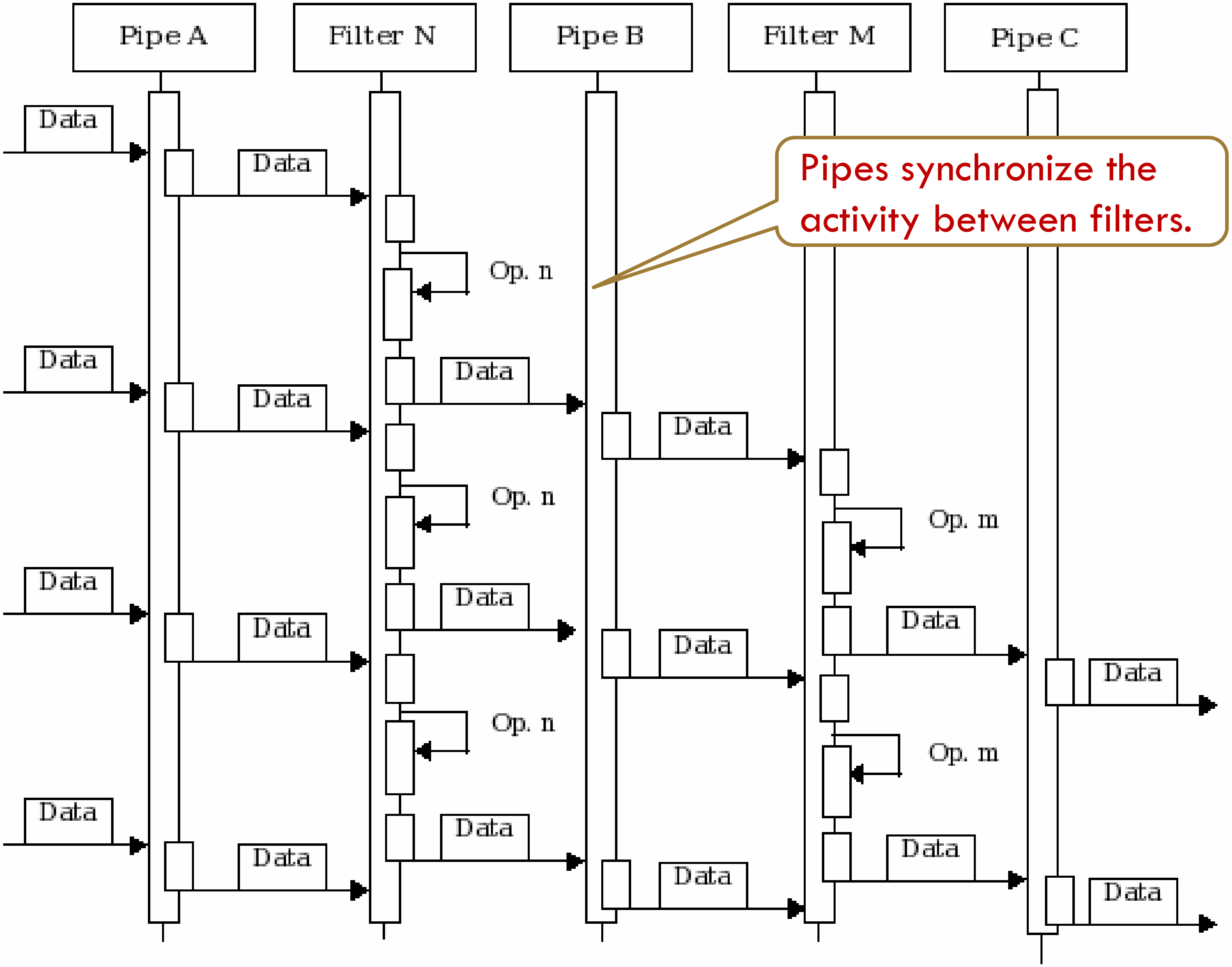




Pipes and Filters – Dynamics

- Due to the parallel execution of the components of the pattern, the typical scenario given on the right hand side describes basic run-time behavior

Interaction Diagram of the Pipes and Filters pattern



Pipes synchronize the activity between filters.

Pipes and Filters – Design

- □ Divide system’s task into a sequence of processing steps
- □ Define data format to be passed along each pipe
- □ Decide how to implement each pipe connection
- □ Design and implement the filters
- □ Setup the processing pipeline

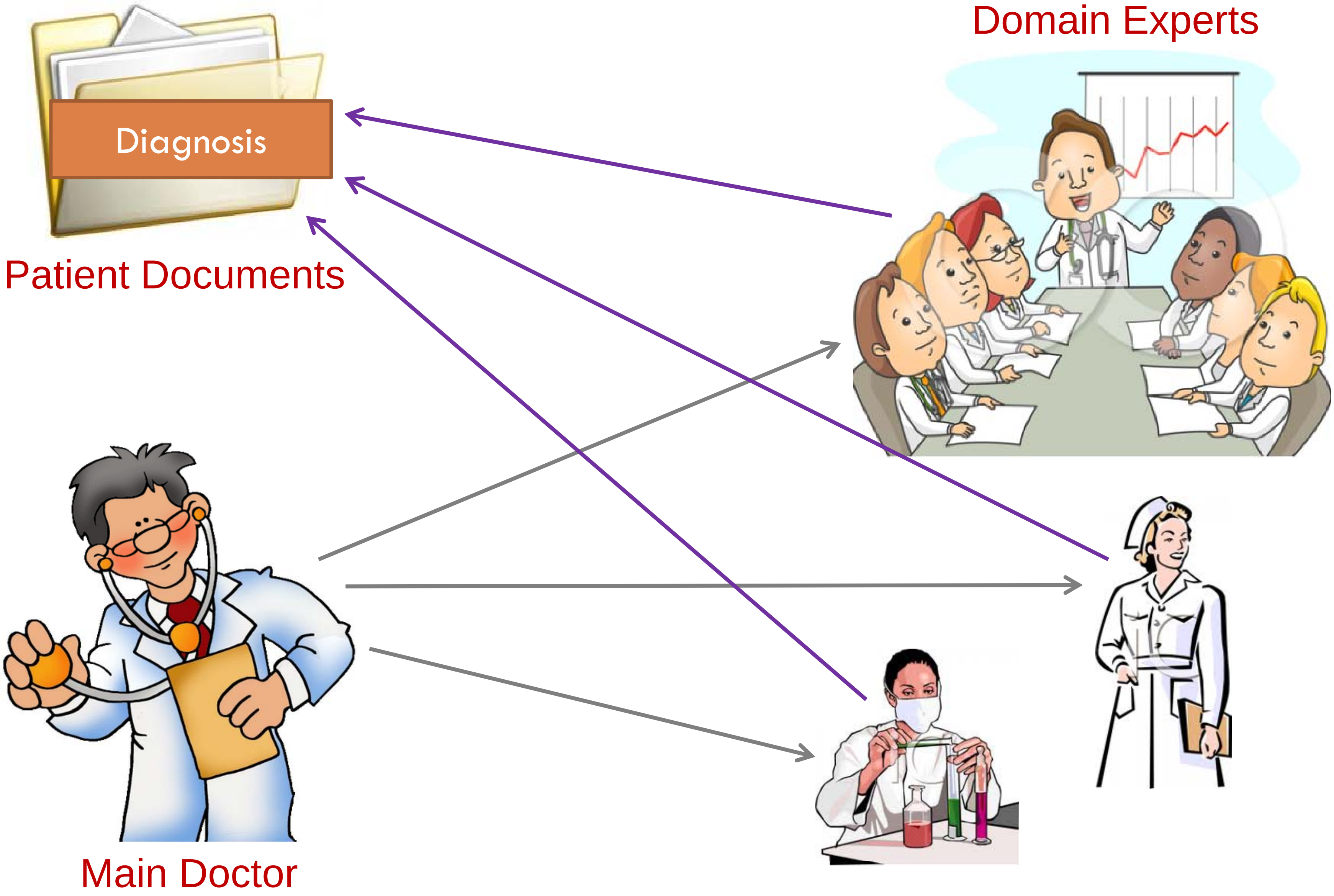
Benefits/Drawbacks of Pipes and Filters

- **Benefits**
  - **Understandability:** Overall input/output behavior is clear; no complex component interactions
  - **Reusability, maintainability, and extensibility**: Filters are black boxes; easy to reuse, replace, and to add..
- **Drawbacks**
  - Batch mentality, which is NOT good for interactive systems
  - Filter ordering can be difficult
  - Filters cannot interact cooperatively to solve a problem
- **Well known examples**
  - Unix shell, batch systems, compilers, language processors..

Blackboard Pattern – The Problem

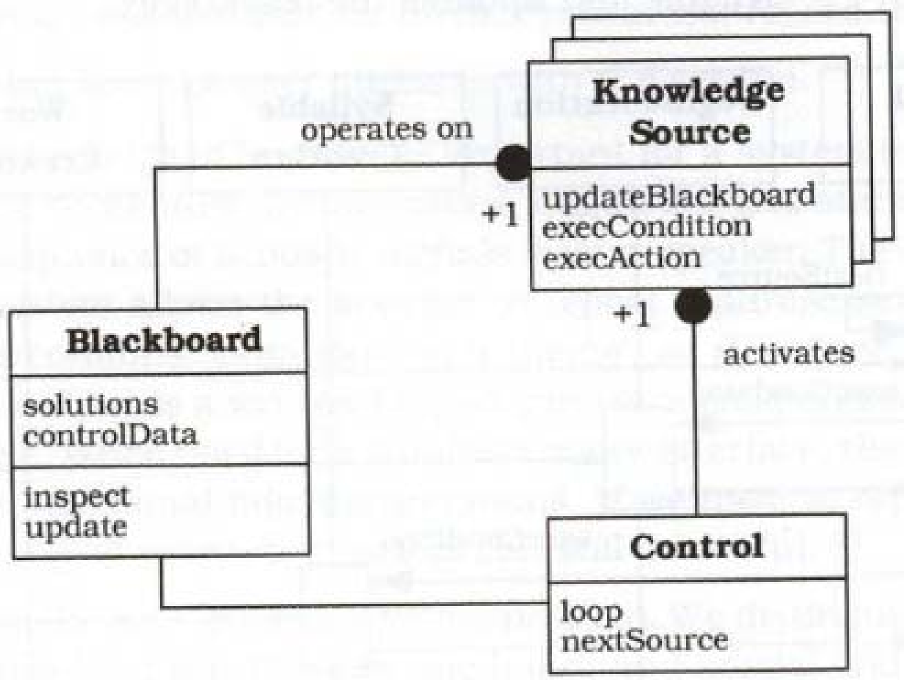
- Problem solving in immature domains
- Problem decomposition results in sub-problems that span several fields of expertise
- Solution to partial problems require different representations and paradigms
- No deterministic strategy how partial problem solvers should combine their knowledge
- Might require experimentation
- No feasible deterministic solution for transformation of data

Blackboard – An example in medical diagnosis



Blackboard Pattern – The Solution

- Divide system into Blackboard, a collection of Knowledge Sources, and a control component
  - **Blackboard** – stores and manages central data
  - **Knowledge sources** – separate independent components that contain domain specific knowledge; solve specific aspects of the overall problem; updates blackboard
  - **Control** – monitors changes on blackboard, schedules knowledge source activations

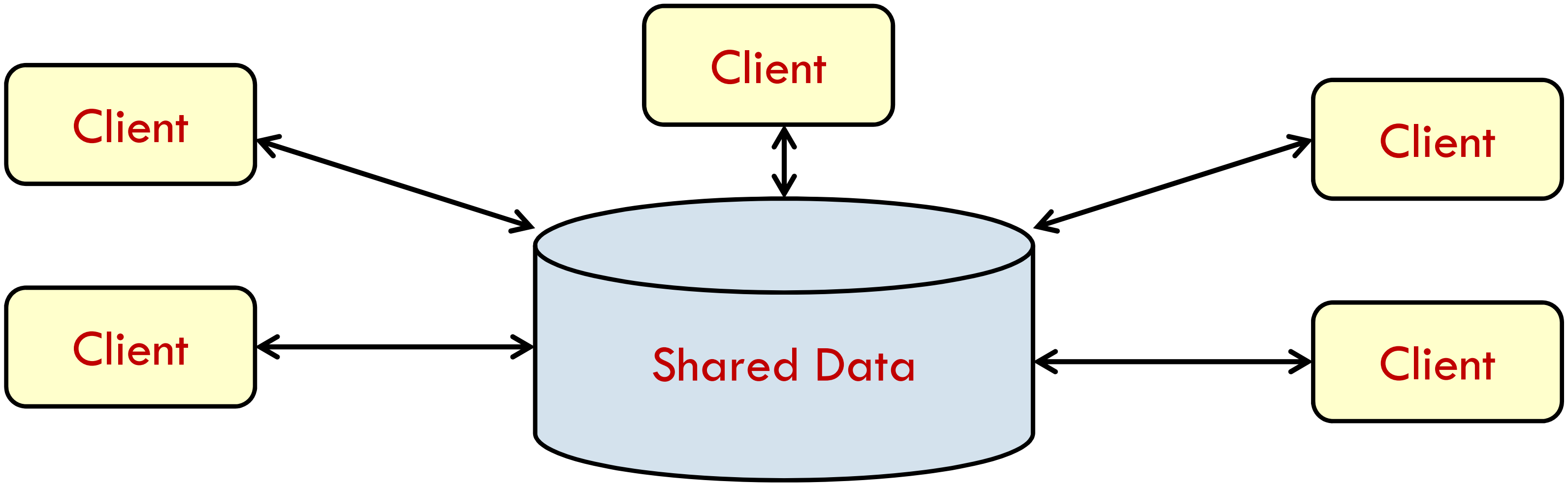


Blackboard Pattern – Design

- Define the problem (domain and general fields of knowledge necessary to find a solution)
- Divide the solution process into steps
- Divide the knowledge into specialized knowledge sources with certain subtasks
- Define the vocabulary of the blackboard
- Specify control of the system
- Implement the knowledge sources

Blackboard variant – Repository

- Divide system into repository and knowledge sources
  - Repository – Manages central data
  - Knowledge sources – evaluates its own applicability, computes a result, updates repository
  - No control – Passive data (traditional databases)



Benefits/Drawbacks of Blackboard

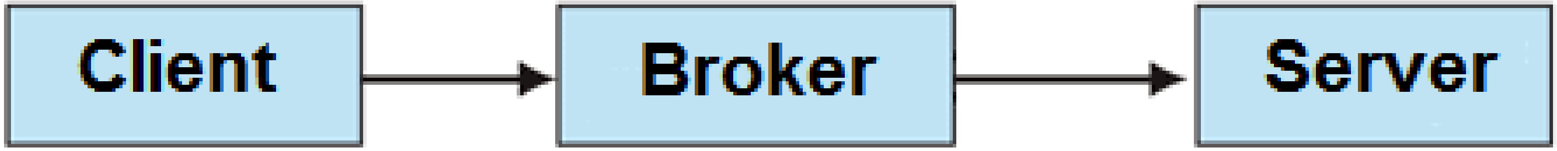
- **Benefits**
  - Reusable knowledge sources
  - Changeability and Maintainability – Blackboard, Control and Knowledge Sources are strictly separated
- **Drawbacks**
  - Difficulty of testing
  - No good solution is guaranteed
  - Difficult to establish a good control strategy (might need experimental approach)

Broker Pattern – The Problem

- **Context**
  - Distributed, heterogeneous system with independent cooperating components
- **Problems**
  - Clients should be able to access services provided by servers through remote, location-transparent service invocations
  - You need to exchange, add, or remove servers at runtime

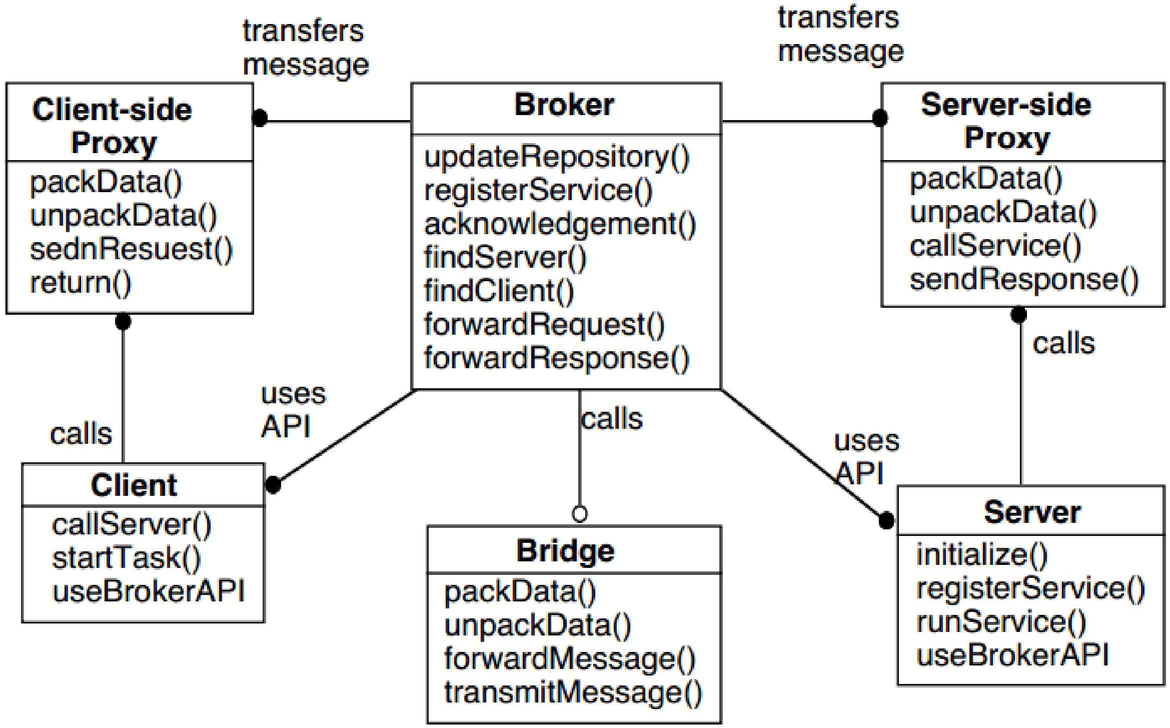
Broker Pattern – The Solution

- Introduce a Broker component to achieve better decoupling of clients and servers
- Servers register themselves with the Broker
- Client access functionality of servers by sending requests via the Broker
- The Broker locates the appropriate server, forwards the request to the server and transmits results back to clients



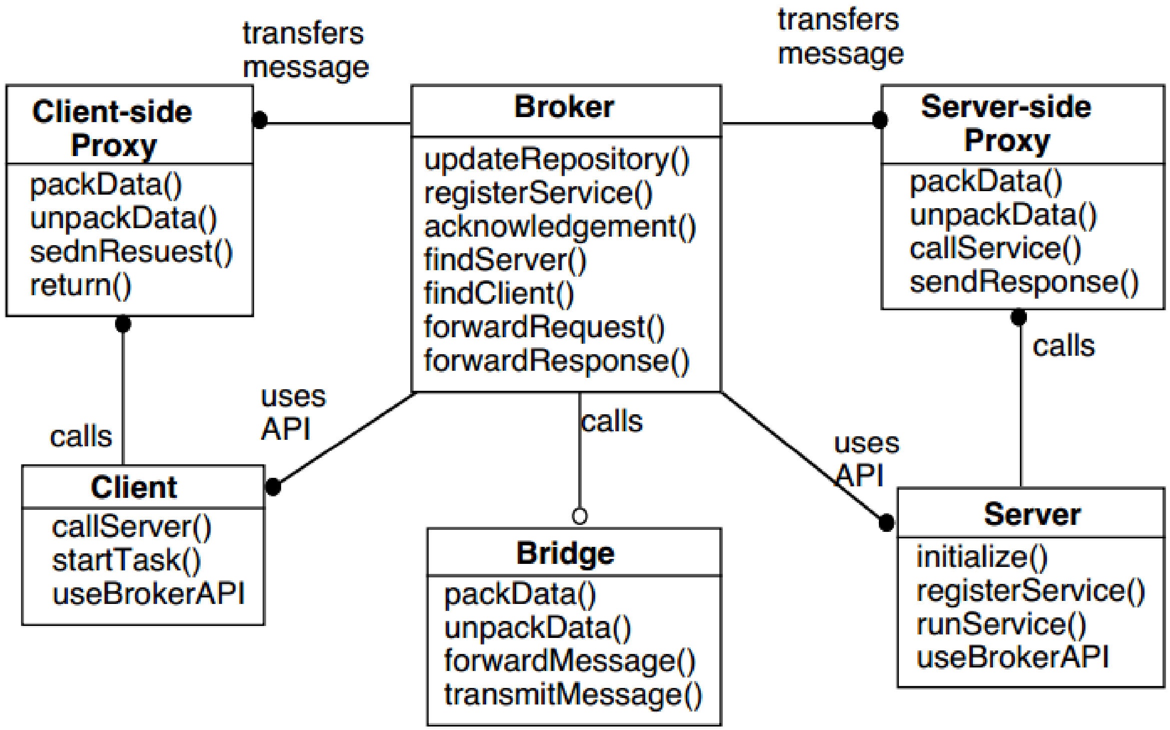
Broker Pattern – Clients and Servers

- **Server**
  - Implements (objects that expose their) functionality through interfaces
- **Client**
  - Applications that access the services of server(s)
  - To call remote services, clients forward requests to the Broker
  - After an operation has executed, clients receive responses or exceptions (☺) from the Broker



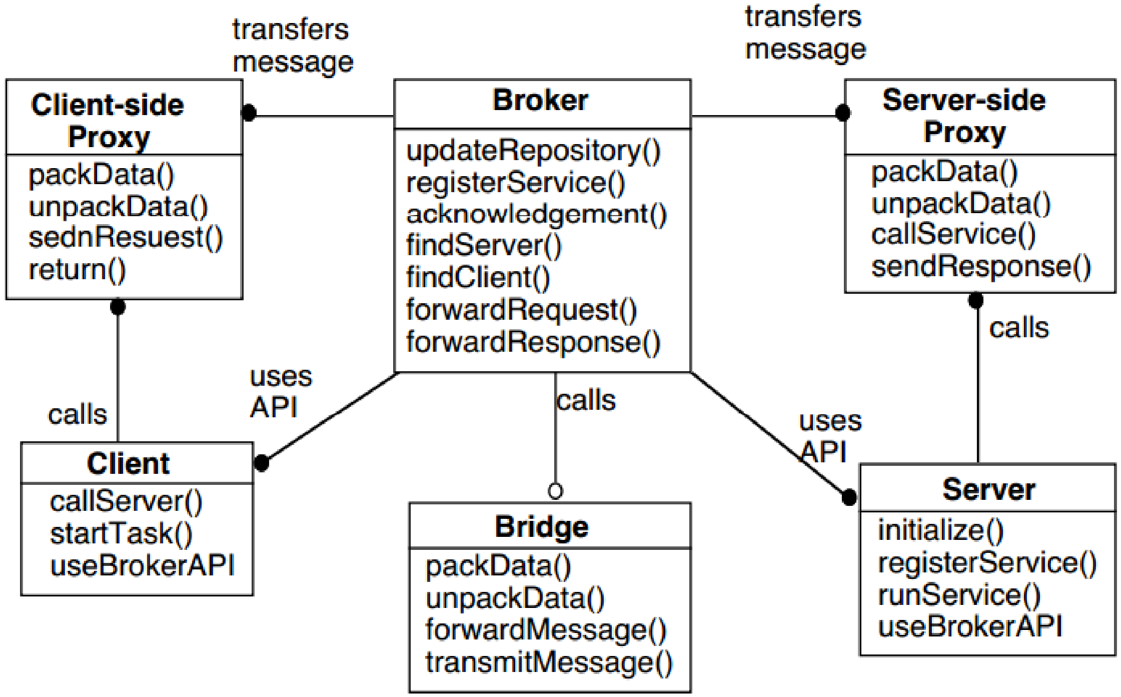
Broker Pattern – Broker

- **Broker**
  - Responsible for the transmission of requests from clients to servers
  - As well as the transmission of responses and exceptions back to the client
  - Provides API to clients and servers that include operations for registering servers and for invoking server methods



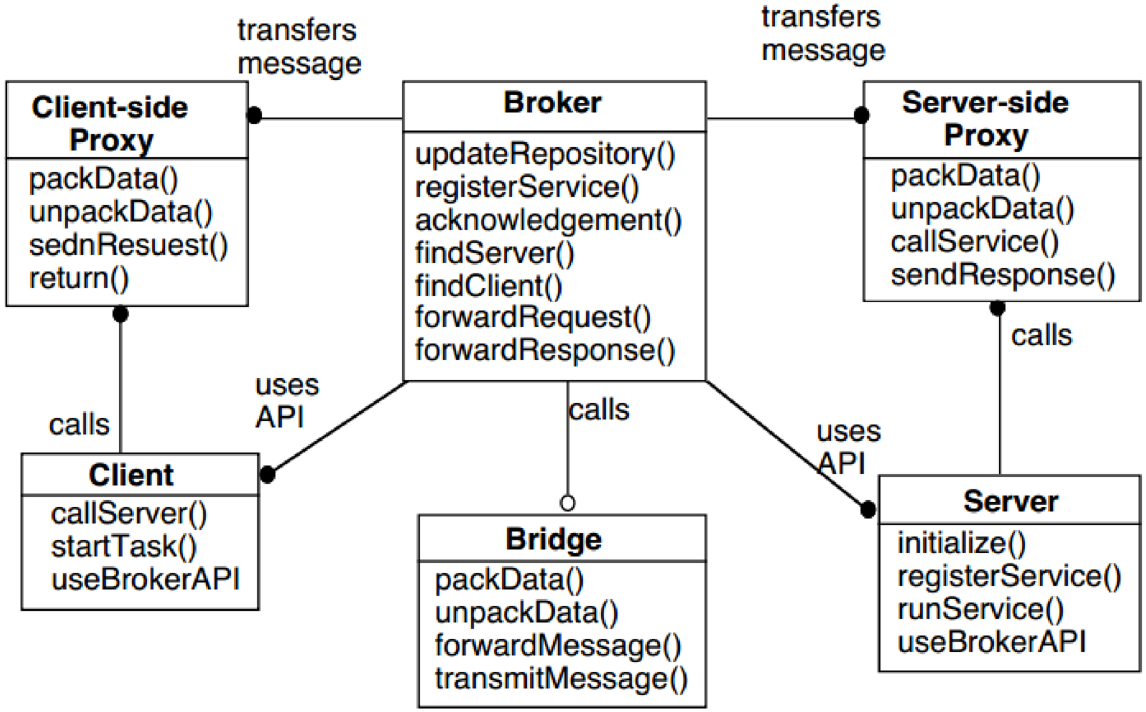
Broker Pattern – Proxies

- **Client-Side Proxies**
  - Represent a layer btw clients and the broker
  - Provides tranparency, in that a remote object appears to the client as a local one
  - Hides implementation details from the clients



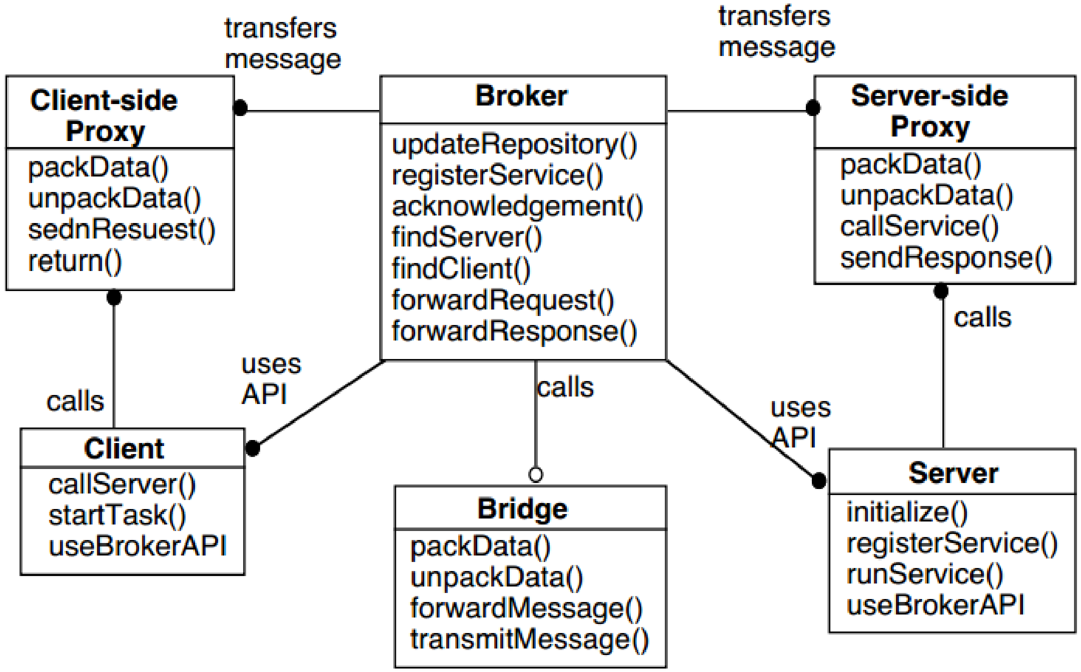
Broker Pattern – Proxies

- **Server-Side Proxies**
  - Responsible for receiving requests, unpacking incoming messages and calling the appropriate service
  - They are used in addition for marshalling results and exceptions before sending them to the client
  - The Client-Side Proxy receives the incoming messages from the broker, unmarshalls the data and forward it to the client



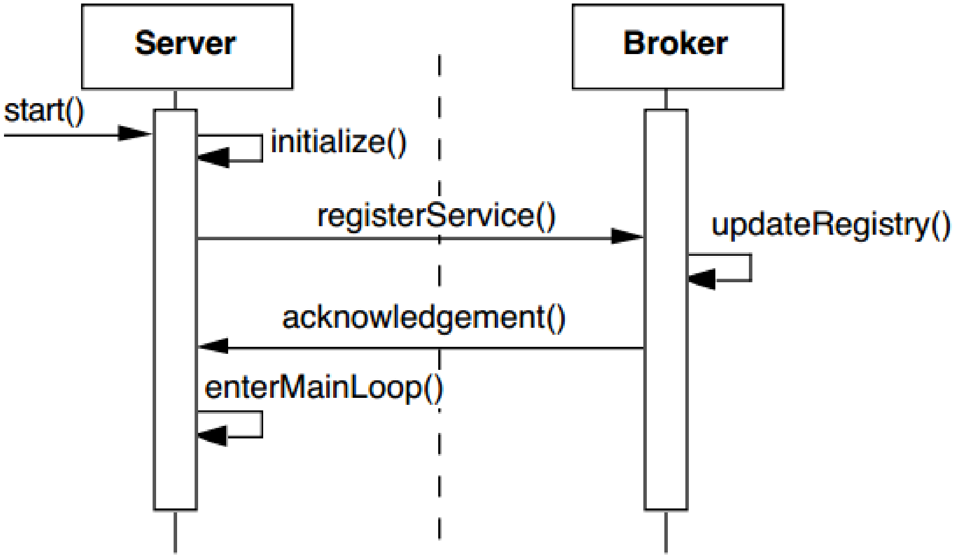
Broker Pattern – Bridge

- **Bridge**
  - Different brokers might need to communicate independently of the different network and OS in use
  - Bridge encapsulates all these system-specific details
  - When a request arrive for a server that is maintained by a local broker, the broker passes the request directly to the server
  - If the specified server is hosted by another broker, the local broker finds a route to the remote broker and forwards the request using this route

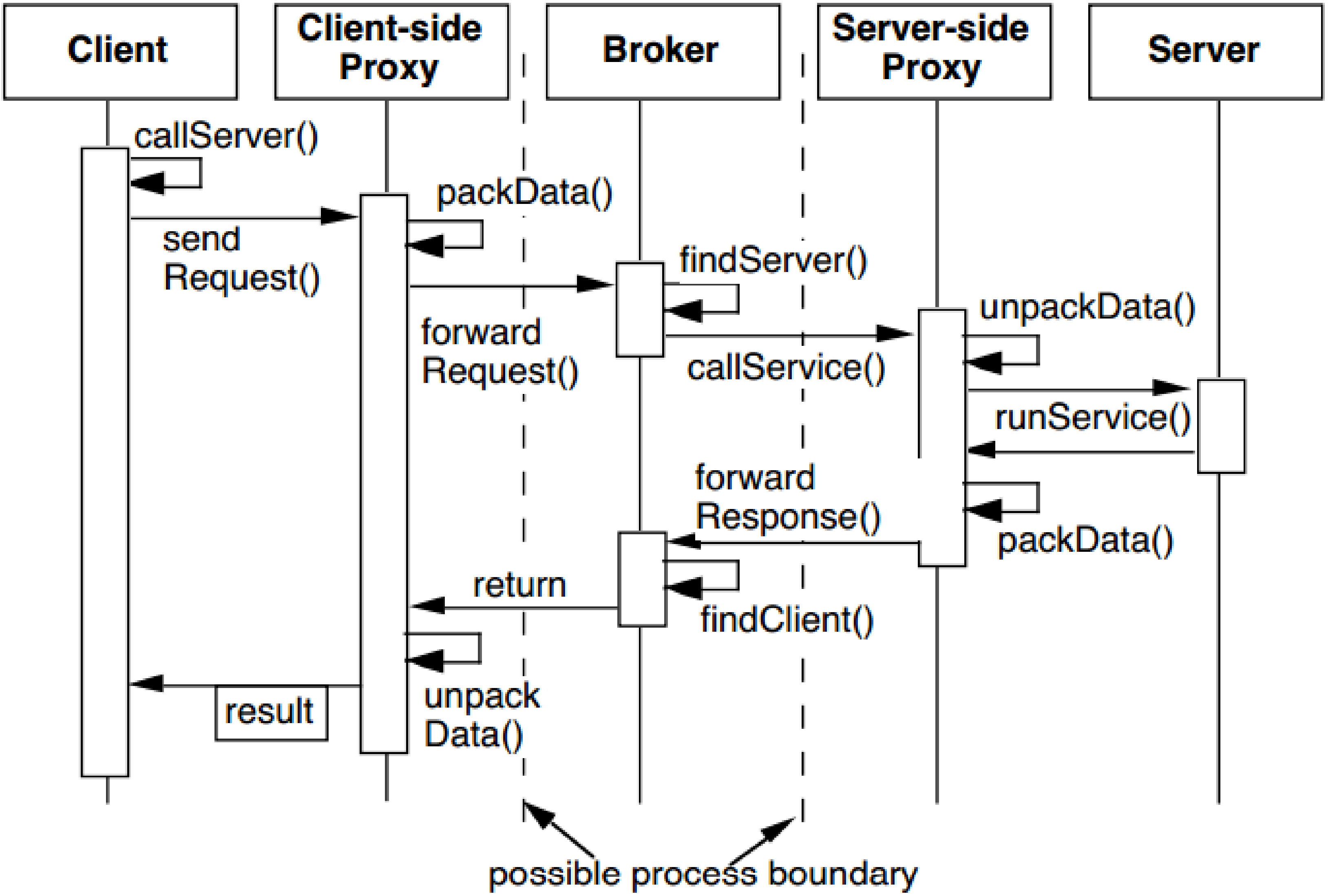


Broker Pattern – Dynamic Behavior

- **Server registers itself with Local Broker**
  - The broker is started at the initialization phase of the system. The broker enters its event loop and waits for incoming messages
  - The server is started, which then executes its initialization code
  - After initialization is complete, the server registers itself with the broker
  - The broker extracts all the necessary information from the registration request and stores it into one or more repositories. These repositories are used to locate and activate servers. An acknowledgment is sent back
  - After receiving acknowledgment from the broker, the server enters its main loop waiting for incoming client requests



Broker Pattern – Dynamic Behavior



Benefits of Broker

- **Location Transparency** – Broker is responsible for locating servers, so clients need NOT know where servers are located, similarly servers do not care where clients are located
- **Changeability & Extensibility of Components** – If server implementations change without affecting interfaces, clients should not affected
- **Portability enhancements** – The Broker hides OS and network system details from clients and servers by using interaction layers such as APIs, proxies and bridges
  - Porting to different platforms/systems is often achieved by porting the broker and its APIs
- **Reusability of services**
- **Interoperability** w/other brokers – by understanding their protocols

Drawbacks of Broker

- **Restricted efficiency**
  - Applications using a Broker implementation are usually slower than applications whose component distribution is static and known
- **Lower fault tolerance**
  - If a server or a broker fails during program execution, all the applications that depend on the server or broker are unable to continue successfully
- **Testing & Debugging may be harder**
  - Since it is distributed..

Broker Examples

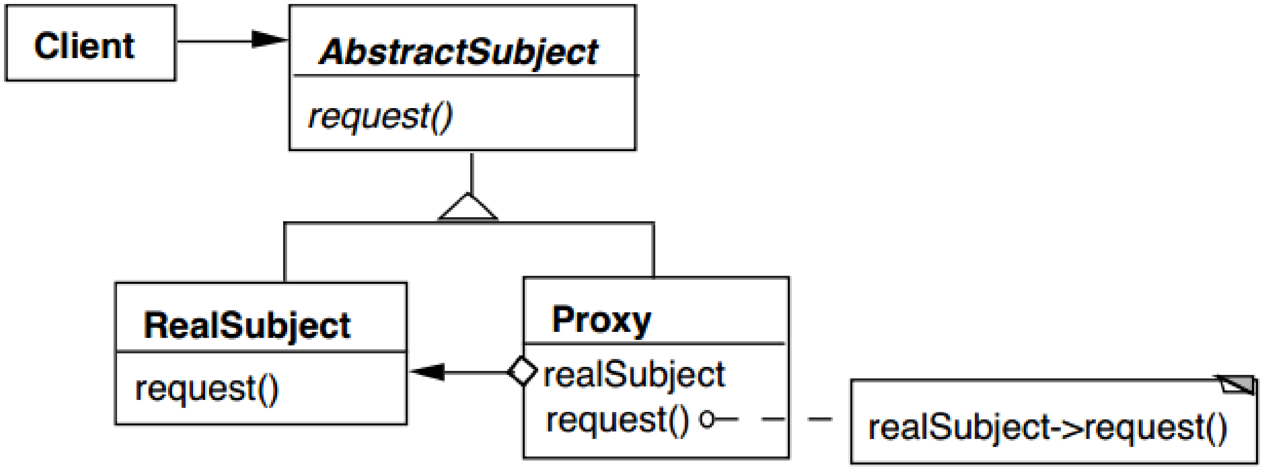
- CORBA
- Microsoft's Object Linking and Embedding (OLE)
- WWW – Hypertext browsers such as Netscape act as brokers and WWW servers play the role of service providers

Proxy Pattern

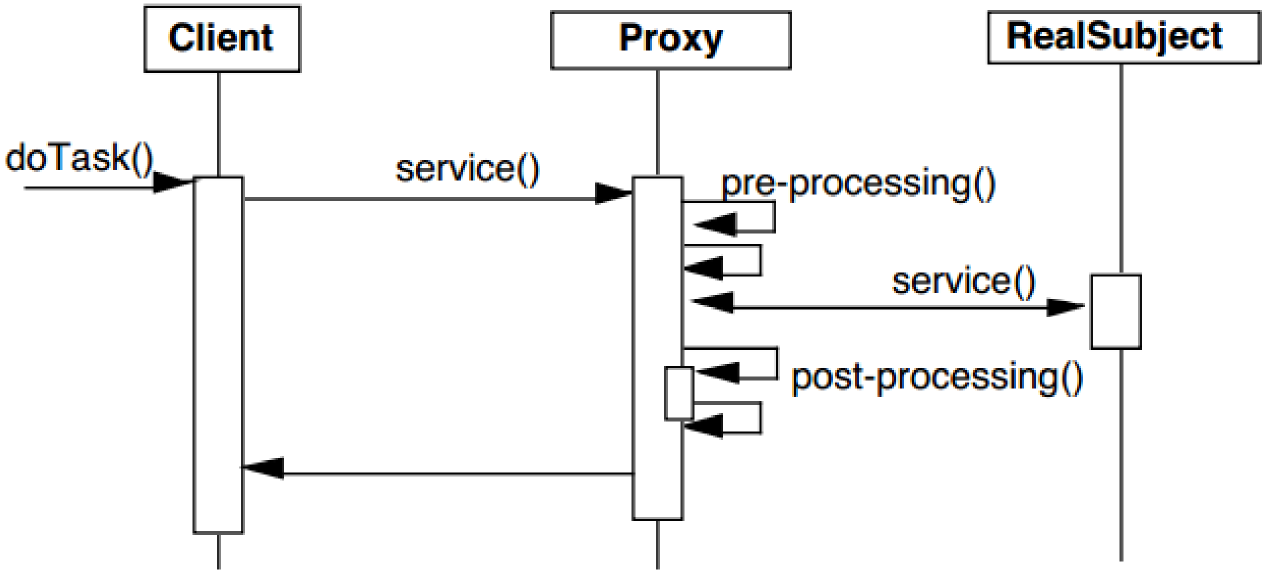
- **Definition:**
  - The agency for a person who acts as a substitute for another person, authority to act for another
- The proxy has the same interface as the original object
- Use common interface (or abstract class) for both the proxy and original object
- Proxy contains a reference to original object, so proxy can forward requests to the original object

Proxy Pattern – Structure & Dynamics

- □ **Structure**



- □ **Dynamics**



Proxy Pattern – Reasons for proxies

- The actual object is on a remote machine (remote address space)
- Hide real details of accessing the object
- Used in CORBA, Java Remote Method Invocation (RMI)
- You may wish to delay creating an expensive object until it is really accessed
- It may be too expensive to keep entire state of the object in memory at one time
- Provides different objects different level of access to original object
- Multiple local clients can share results from expensive operations: remote accesses or long computations
- Protect local clients from outside world

The MVC Pattern

□ This is your job! ;)