Patterns in Software Engineering

Lecturer: Raman Ramsin

Lecture 7

GoV Patterns – Architectual

Part 1

GoV Patterns for Software Architecture

- According to Buschmann et al.:
  - A *pattern for software architecture* describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution.
  - The solution scheme is specified by describing
    - the constituent components
    - The responsibilities and relationships of the components
    - the ways in which the components collaborate.

GoV Patterns: Pattern Schema

- **Context:** Design situation giving rise to a design problem
- **Problem:** Set of forces (requirements, constraints, and desirable properties) repeatedly arising in the context that need to be addressed
- **Solution:** Configuration to balance the forces
  - *Structure* with components and relationships
  - Run-time *behavior*

GoV Patterns: Categories

- **Architectural**
  - Expresses a fundamental structural organization schema for software systems.
    - Provides a set of predefined subsystems (or components), specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- **Design**
  - Provides a scheme for refining the subsystems or components of a software system, or the relationships between them.
    - Describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.
- **Idiom**
  - Low-level pattern specific to a programming language.
    - Describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Architectural Patterns: Categories

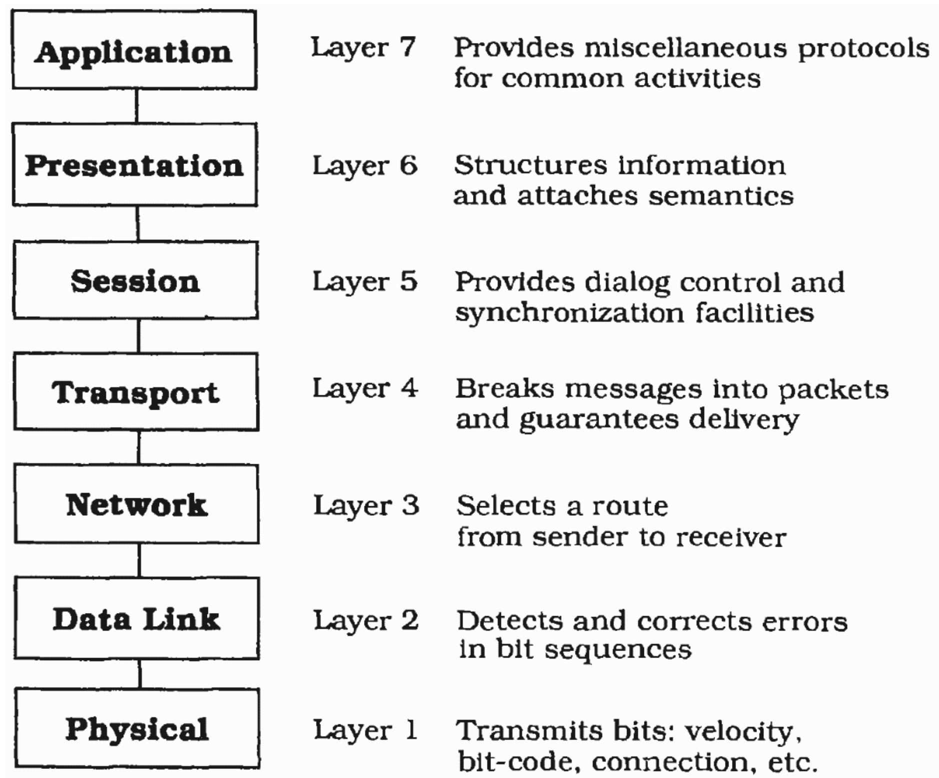
- **From Mud to Structure**
  - Support a controlled decomposition of a system task into cooperating subtasks.
  - *Layers*, *Pipes and Filters*, and *Blackboard*
- **Distributed Systems**
  - Deal with the infrastructure of distributed applications.
  - *Broker*, also *Microkernel* and *Pipes and Filters*, which only consider distribution as a secondary concern.
- **Interactive Systems**
  - Support the structuring of systems that feature human-computer interaction.
  - *Model-View-Controller* and *Presentation-Abstraction-Control*
- **Adaptable Systems**
  - Support extension of applications and their adaptation to evolving technology and changing functional requirements.
  - *Reflection* and *Microkernel*

Architectural: From Mud to Structure

- **Layers:** Helps to structure applications that can be decomposed into groups of subtasks.
  - Each group of subtasks is at a particular level of abstraction.
- **Pipes and Filters:** Provides a structure for systems that process a stream of data.
  - Each processing step is encapsulated in a filter component.
  - Data is passed through pipes between adjacent filters.
  - Recombining filters allows you to build families of related systems.
- **Blackboard:** Useful for problems for which no deterministic solution strategies are known.
  - Several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

From Mud to Structure: Layers

- Helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.



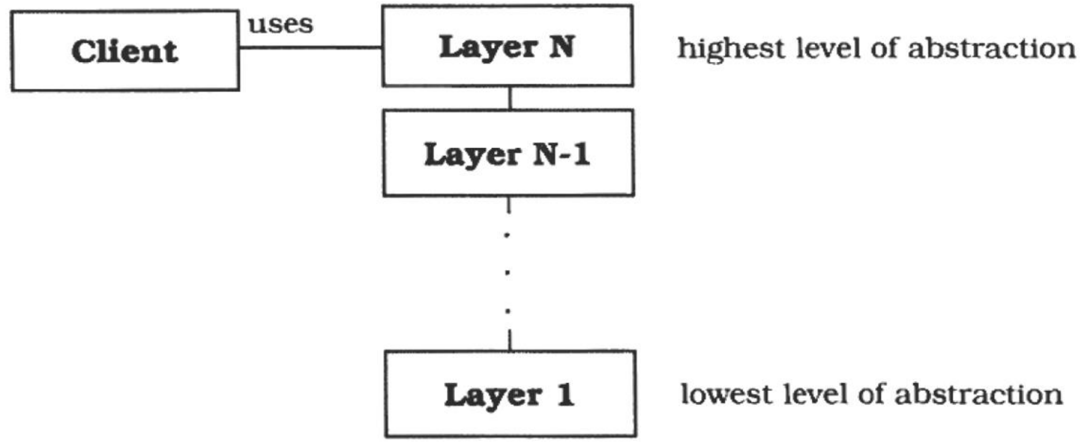
From Mud to Structure: Layers

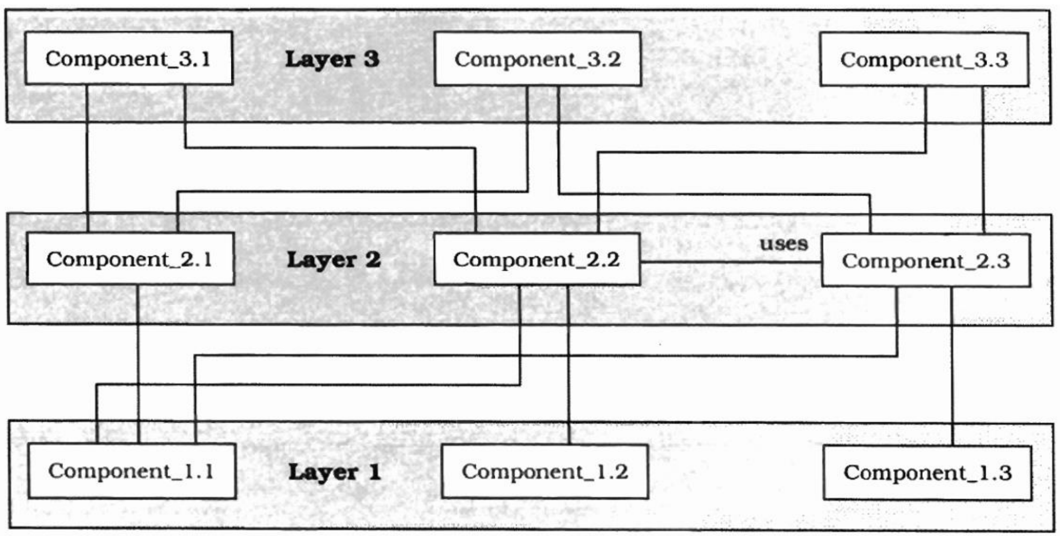
- **Context -** A large system that requires decomposition.

- **Problem -** Forces are as follows:
  - Late source code changes should not ripple through the system.
  - Interfaces should be stable.
  - Parts of the system should be exchangeable.
  - It may be necessary to build other systems at a later date with the same low-level issues as the system you are currently designing.
  - Similar responsibilities should be grouped to help understandability and maintainability.
  - There is no 'standard' component granularity.
  - Complex components need further decomposition.
  - Crossing component boundaries may impede performance.
  - The system will be built by a team of programmers, and work has to be subdivided along clear boundaries.

Layers: Structure

Class Layer J	**Collaborator**
Responsibility • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1.	• Layer J-1





Layers: Dynamics

- **Scenario I:** Top-down communication initiated by a client.
- **Scenario II:** Bottom-up communication, when a chain of actions starts at Layer 1; for example when a device driver detects input.
- **Scenarios III and IV:** Requests only travel through a subset of the layers.
  - III: A top-level request may only go to level N-1 If this level can satisfy the request; e.g. when level N-1 acts as a cache.
  - IV: A bottom-level request may only travel through the next few upper-level layers.
- **Scenario V:** involves two stacks of N layers communicating with each other.

Layers: Consequences

- ✓ Reuse of layers
- ✓ Support for standardization
- ✓ Dependencies are kept local
- ✓ Exchangeability

- ✗ Cascades of changing behavior
- ✗ Lower efficiency
- ✗ Potential for unnecessary work
- ✗ Difficulty of establishing the correct granularity of layers

From Mud to Structure: Pipes and Filters

- Provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters.
- **Context -** Processing data streams.
- **Problem -** Forces are as follows:
  - Future system enhancements should be possible by exchanging processing steps or by recombination of steps, even by users.
  - Small processing steps are easier to reuse in different contexts than large ones.
  - Non-adjacent processing steps do not share information.
  - Different sources of input data exist.
  - It should be possible to present or store final results in various ways.
  - Explicit storage of intermediate results for further processing in files clutters directories and is error-prone, if done by users.
  - You may not want to rule out multi-processing the steps, for example running them in parallel.

Pipes and Filters: Structure – Filters and Pipes

- **Filter:** processing unit of the pipeline.
  - *Enriches*, *refines* or *transforms* its input data.
  - Its activity can be triggered by several events:
    - subsequent pipeline element pulls output data from the filter (*passive*).
    - The previous pipeline element pushes new input data to the filter (*passive*).
    - Most commonly, the filter is *active* in a loop.
- **Pipe:** connection between filters.
  - If two active components are joined, the pipe synchronizes them. This synchronization is done with a first-in- first-out buffer.

Class Filter	**Collaborators**
Responsibility • Gets input data. • Performs a function on its input data. • Supplies output data.	• Pipe

Class Pipe	**Collaborators**
Responsibility • Transfers data. • Buffers data. • Synchronizes active neighbors.	• Data Source • Data Sink • Filter

Pipes and Filters: Structure – Sources and Sinks

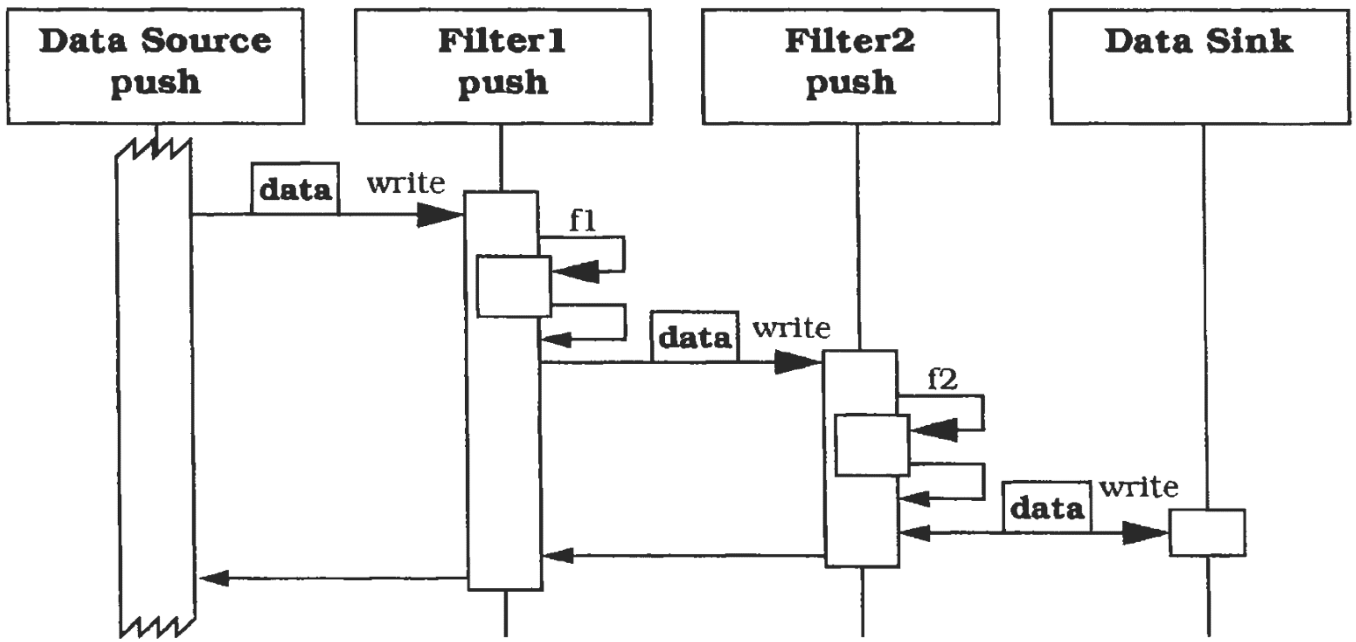
- The *data source* represents the input to the system, and provides a sequence of data values of the same structure or type.
  - Examples of such data sources are a file consisting of lines of text, or a sensor delivering a sequence of numbers.
- The *data sink* collects the results from the end of the pipeline.

Class Data Source	**Collaborators**
Responsibility • Delivers input to processing pipeline.	• Pipe

Class Data Sink	**Collaborators**
Responsibility • Consumes output.	• Pipe

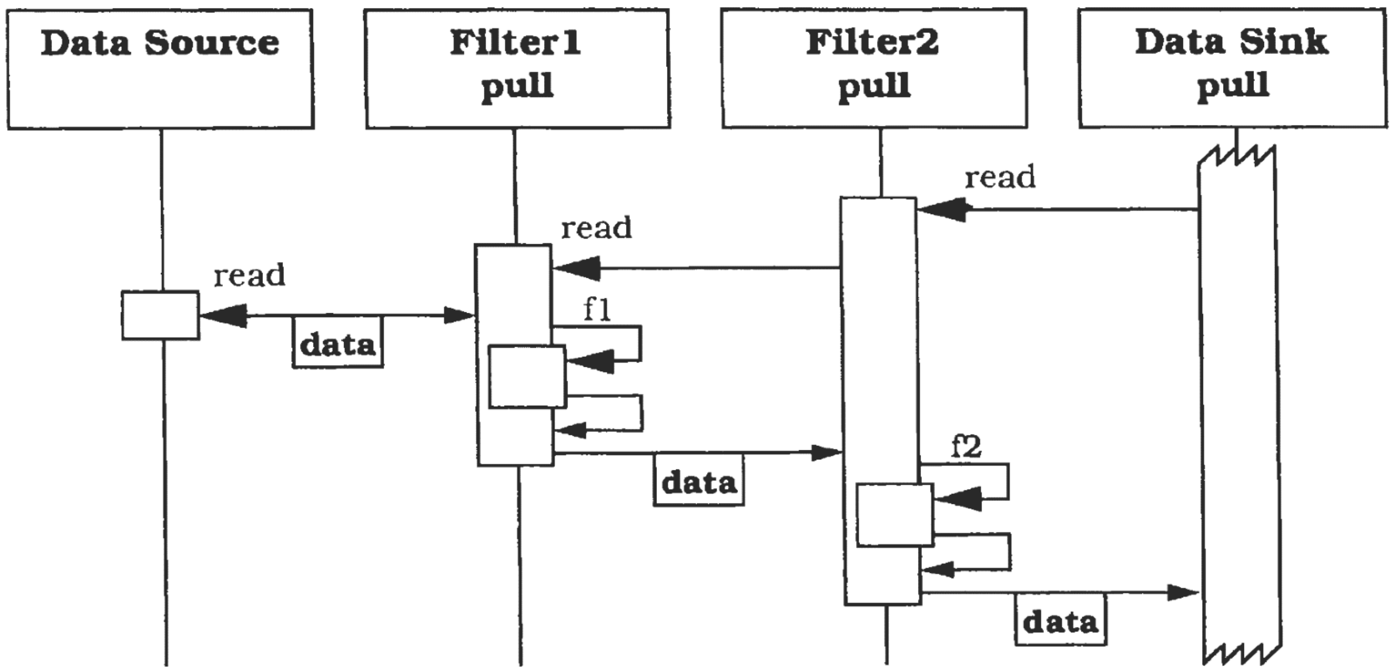
Pipes and Filters: Dynamics – Scenario I

- Push pipeline in which activity starts with the data source. Filter activity is triggered by writing data to the passive filters.



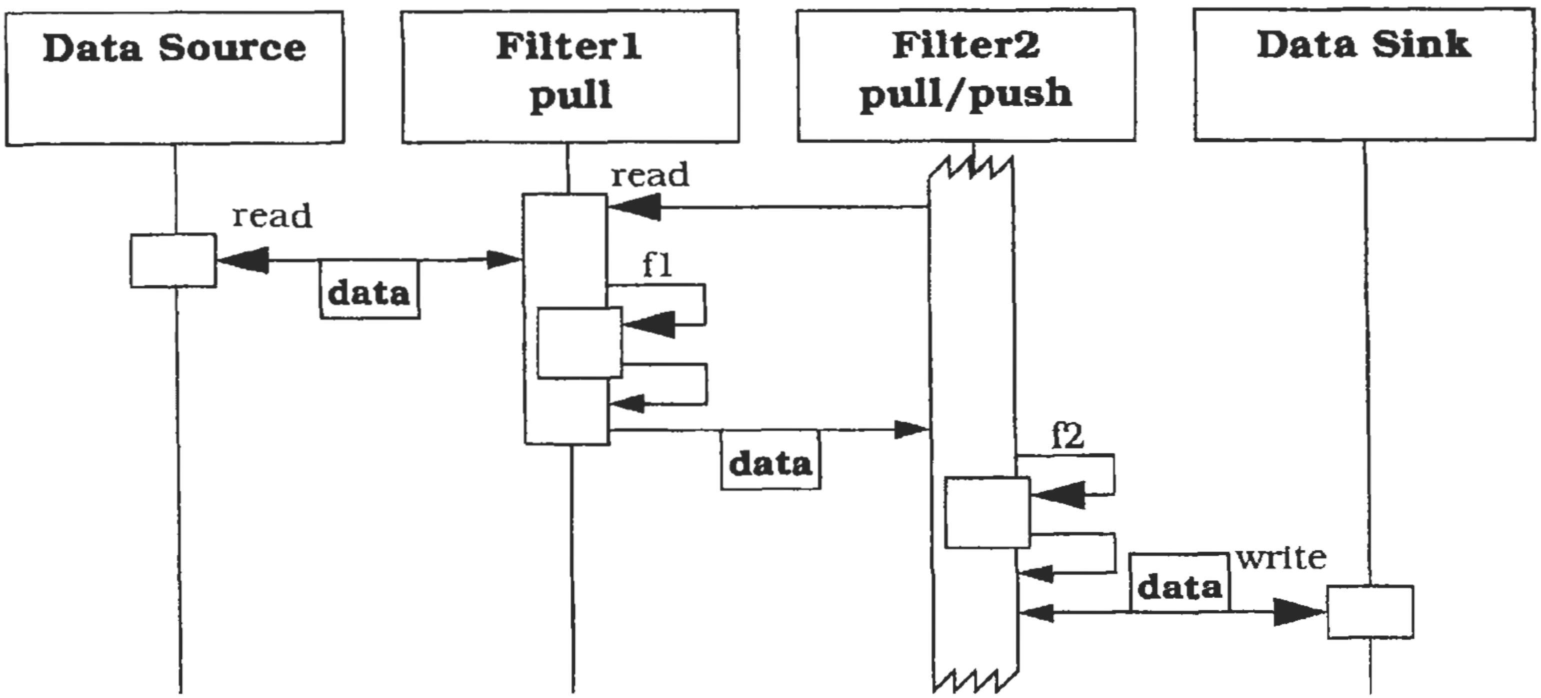
Pipes and Filters: Dynamics – Scenario II

- Pull pipeline in which the control flow is started by the data sink calling for data.



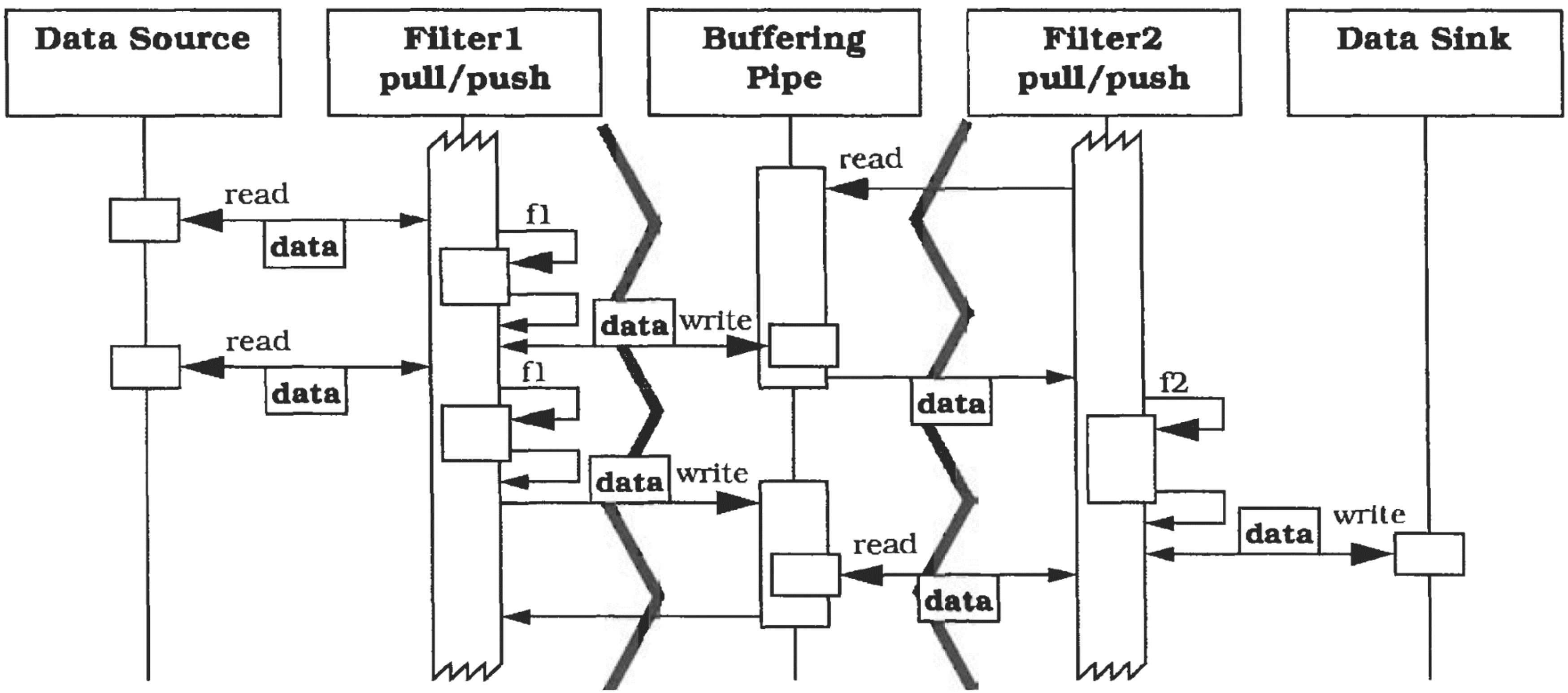
Pipes and Filters: Dynamics – Scenario III

- mixed push-pull pipeline with passive data source and sink.



Pipes and Filters: Dynamics – Scenario IV

- All filters actively pull, compute. and push data in a loop.



Pipes and Filters: Consequences

- ✓ No intermediate files necessary, but possible
- ✓ Flexibility by filter exchange and recombination
- ✓ Reuse of filter components
- ✓ Rapid prototyping of pipelines
- ✓ Efficiency by parallel processing

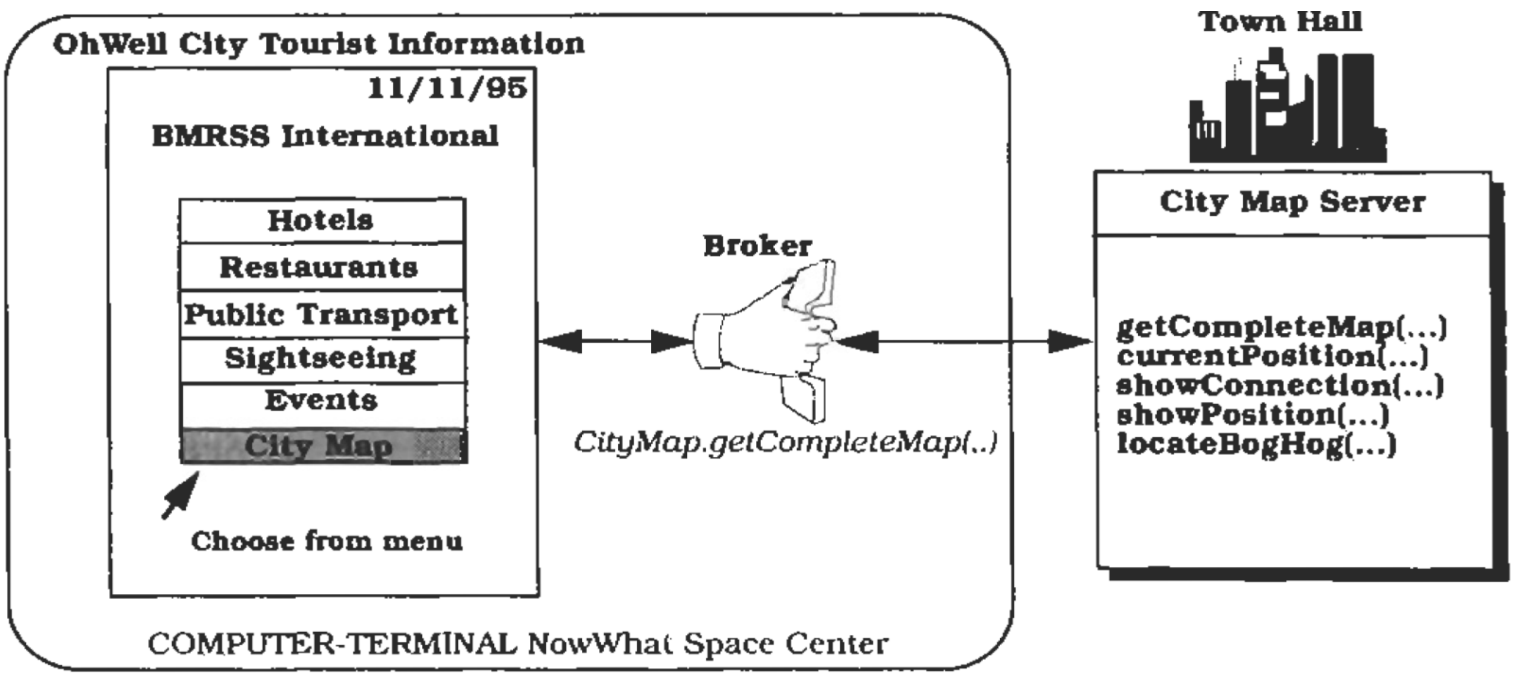
- ✗ Sharing state information is expensive or inflexible
- ✗ Efficiency gain by parallel processing is often an illusion
- ✗ Data transformation overhead

Architectural: Distributed Systems

- **Broker:** Used to structure distributed software systems with decoupled components that interact by remote service invocations.
  - A *broker* component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.
- ***Microkernel:*** Separates a minimal functional core *(microkernel)* from extended functionality and customer-specific parts.
  - Applies to software systems that must be able to adapt to changing system requirements.
  - Microkernel systems employ a Client-Server architecture in which clients and servers run on top of the microkernel component.
- ***Pipes and Filters:*** Provides a structure (possibly distributed) for systems that process a stream of data.

Distributed Systems: Broker

- Used to structure distributed software systems with decoupled components that interact by remote service invocations.
  - A *broker* component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.



Distributed Systems: Broker

- **Context -** Your environment is a distributed and possibly heterogeneous system with independent cooperating components.

- **Problem -** Forces are as follows:
  - Components should be able to access services provided by others through remote, location-transparent service invocations.
  - You need to exchange, add, or remove components at run-time.
  - The architecture should hide system- and implementation-specific details from the users of components and services.

Broker: Structure – Brokers and Bridges

- *Brokers* are messengers that are responsible for the transmission of
  - requests from clients to servers, and
  - responses and exceptions back to the client.
- *Bridges* are optional components used for hiding implementation details when two brokers interoperate.

Class Broker	**Collaborators**
Responsibility • (Un-)registers servers. • Offers APIs. • Transfers messages. • Error recovery. • Interoperates with other brokers through bridges. • Locates servers.	• Client • Server • Client-side Proxy • Server-side Proxy • Bridge

Class Bridge	**Collaborators**
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	• Broker • Bridge

Broker: Structure – Clients and Servers

- *Servers* implement objects that expose their functionality through interfaces that consist of operations and attributes.
- *Clients* are applications that access the services of at least one server.

Class Client	**Collaborators**
Responsibility • Implements user functionality. • Sends requests to servers through a client-side proxy.	• Client-side Proxy • Broker

Class Server	**Collaborators**
Responsibility • Implements services. • Registers itself with the local broker. • Sends responses and exceptions back to the client through a server-side proxy.	• Server-side Proxy • Broker

Broker: Structure – Proxies

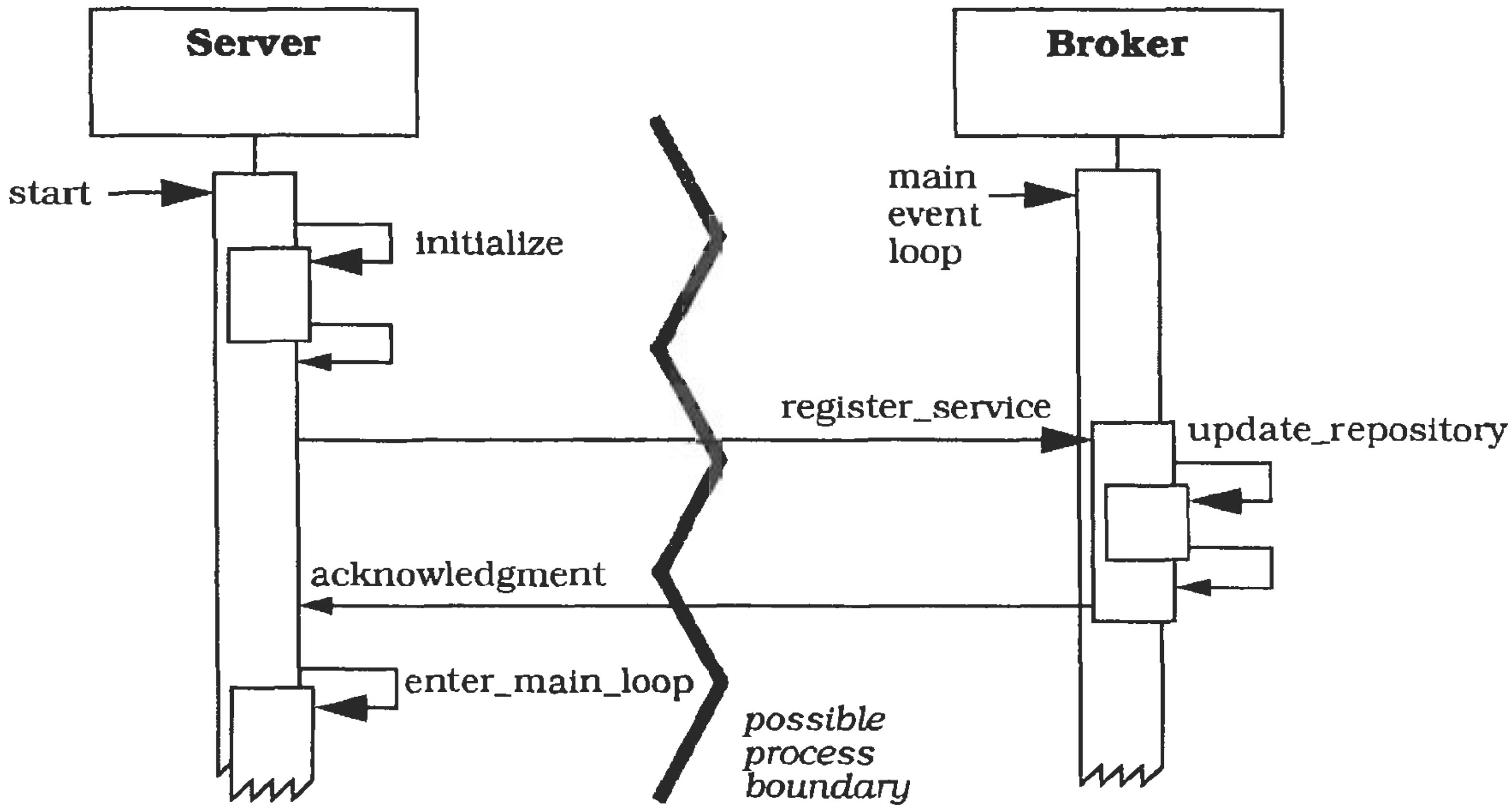
- *Proxies* represent a layer between clients/servers and the broker.
  - This additional layer provides transparency, in that a remote object appears to the client/server as a local one.

Class Client-side Proxy	***Collaborators***
Responsibility • Encapsulates system-specific functionality. • Mediates between the client and the broker.	• Client • Broker

Class Server-side Proxy	***Collaborators***
Responsibility • Calls services within the server. • Encapsulates system-specific functionality. • Mediates between the server and the broker.	• Server • Broker

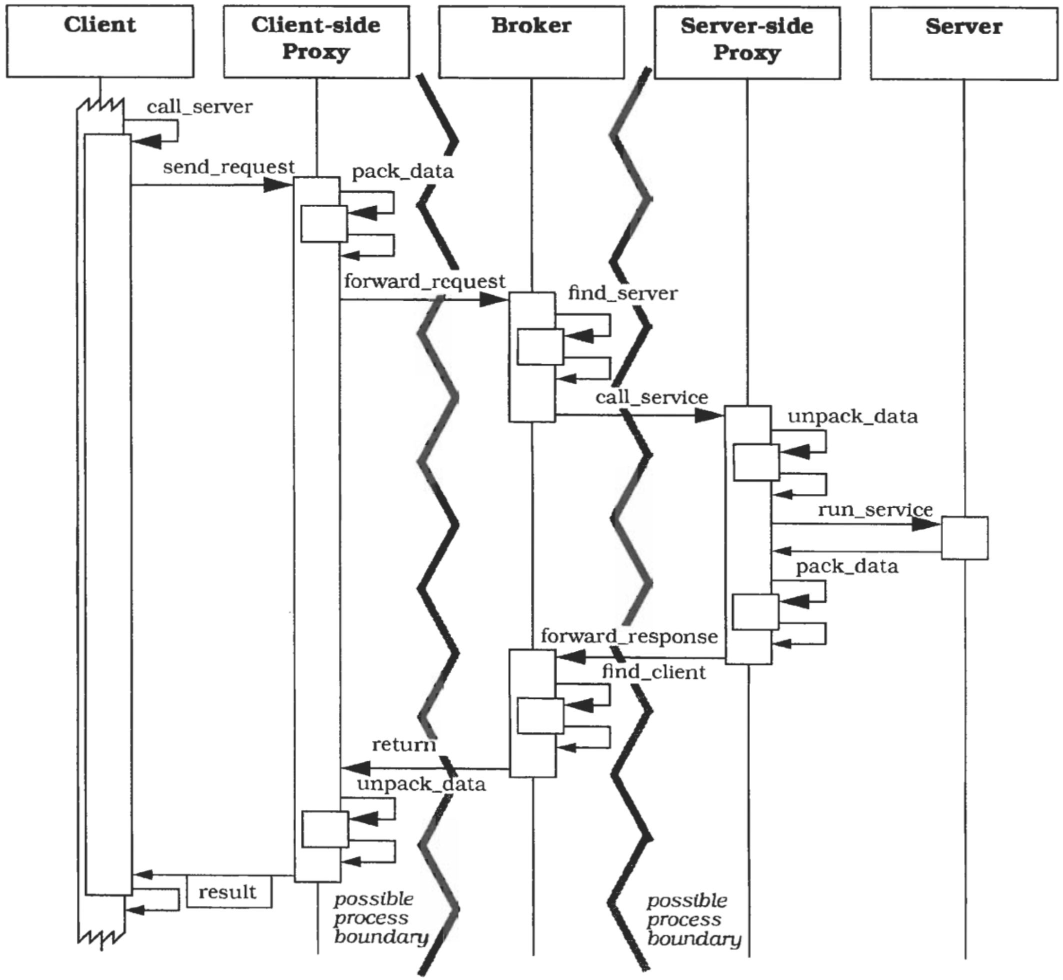
Broker: Dynamics – Scenario I

- A server registers itself with the local broker component.



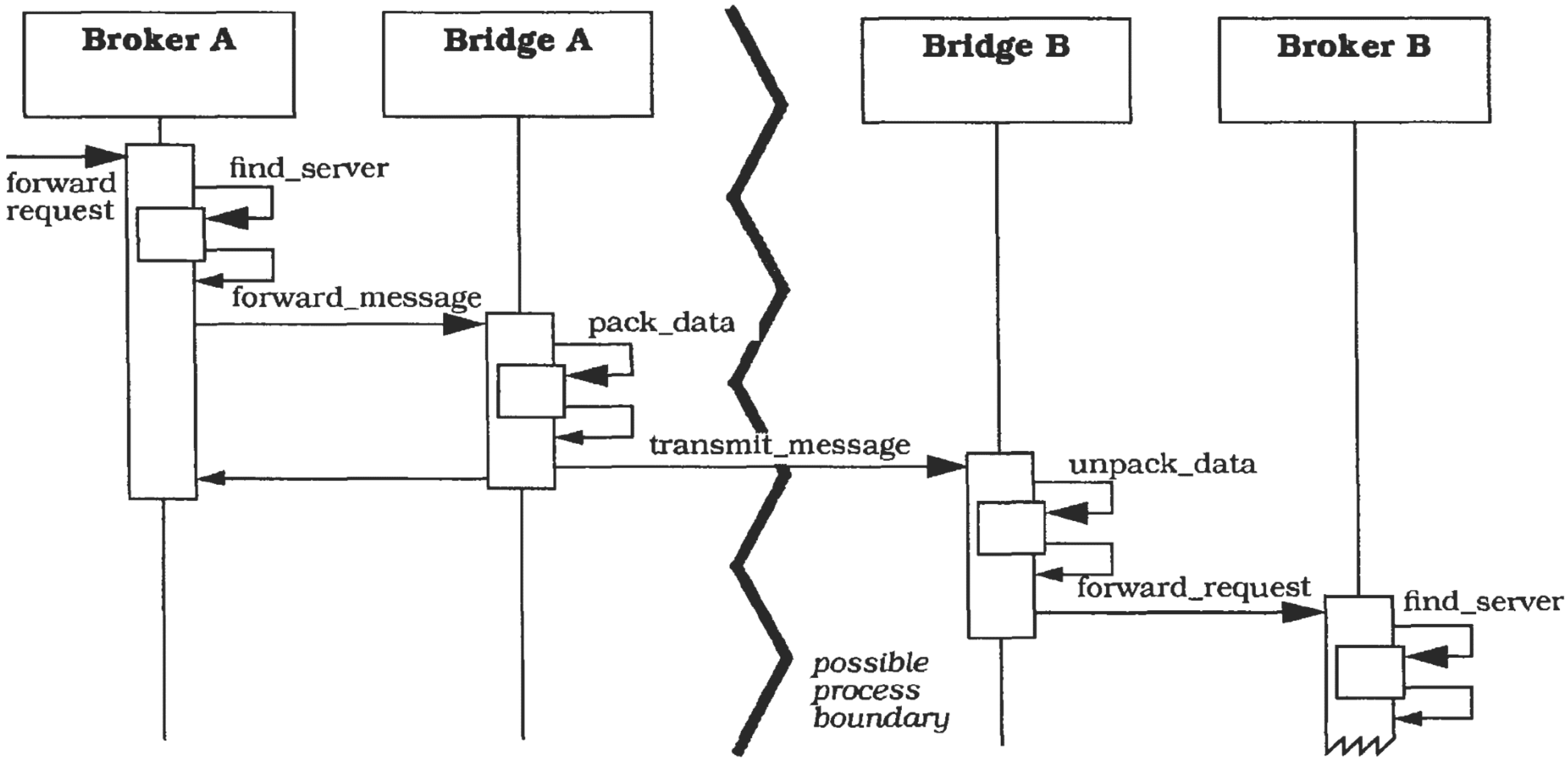
Broker: Dynamics – Scenario II

- A client sends a request to a local server.



Broker: Dynamics – Scenario III

- interaction of different brokers via bridge components.



Broker: Consequences

- ✓ Location Transparency
- ✓ Changeability and extensibility of components
- ✓ Portability of a Broker system
- ✓ Interoperability between different Broker systems
- ✓ Reusability

- ✗ Restricted efficiency
- ✗ Lower fault tolerance
- ✗ Testing and Debugging

Reference

- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., *Pattern-Oriented Software Architecture: A System of Patterns,* Vol. 1. Wiley, 1996.