

***Software
Technology
Group***

TU Darmstadt | FB Informatik

Software Engineering Design

7. Architectural Patterns

Prof. Dr. Mira Mezini

Dipl.-Inform. Christoph Bockisch

Dipl.-Ing. Michael Haupt

Patterns Recap

- benefits of patterns
 - document existing expert design knowledge
 - applicable at various scales of scale and abstraction
 - can be used with any programming paradigm and language
- patterns provide a **mental toolbox**
 - achieve both functional and non-functional requirements

Architectural Patterns

- object-oriented design patterns express collaboration of classes and objects
- architectural patterns express **fundamental structural organisation schemas** for **software systems**
 - set of predefined subsystems
 - specify responsibilities
 - rules and guidelines for organising subsystem relationships
- architectural patterns exist for
 - bringing **structure** into a system
 - organising **distributed, interactive, adaptable** systems

Architectural Patterns

- Layers (structuring pattern)
 - already discussed when introducing patterns
- Pipes and Filters (structuring pattern)
- Microkernel (adaptable systems)

- more architectural patterns
 - Buschmann et al.
Pattern-Oriented Software Architecture, Volume 1
A System of Patterns
John Wiley & Sons, 1996 (5th reprint, 2000)
 - Schmidt et al.
Pattern-Oriented Software Architecture, Volume 2
Patterns for Concurrent and Networked Objects
John Wiley & Sons, 2000

Pipes and Filters

- context: processing data streams
- problem:
 - system built by several developers
 - system decomposes naturally into several processing stages
 - processing stages are likely to be added/removed/changed
- forces
 - exchanging/recombining processing steps
 - non-adjacent processing steps do not share information
 - different input sources exist
 - final results stored/presented in various ways
 - potentially concurrent processing of data
- solution: decompose into pipes and filters

Pipes and Filters: Collaboration

Class Filter

- responsibility
 - gets input data
 - performs a function on its input data
 - supplies output data
- collaborators: Pipe

Class Pipe

- responsibility
 - transfers data
 - buffers data
 - synchronises active neighbours
- collaborators
 - DataSource, DataSink
 - Filter

Class DataSource

- responsibility
 - delivers input to processing pipeline
- collaborators: Pipe

Class DataSink

- responsibility
 - consumes output
- collaborators: Pipe

Pipes and Filters: Filter Types

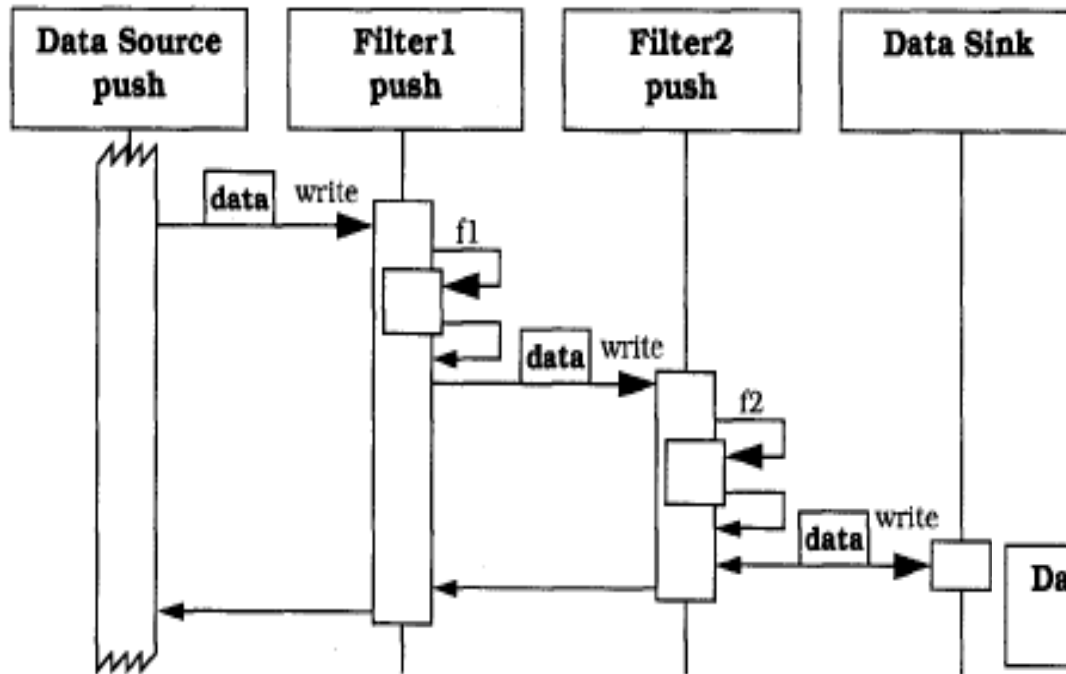
Active Filters

- start processing on their own as separate programs or threads
- filters are active in a loop
 - pull input from and push output to the pipeline

Passive Filters

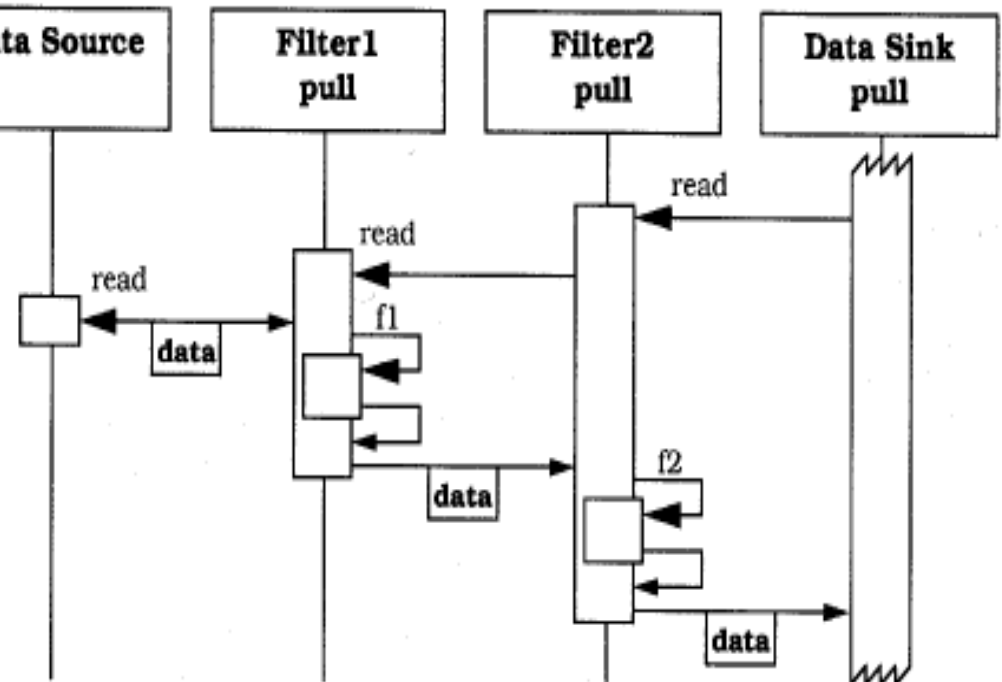
- activated by being called as a function or a procedure
- function (*pull* filters)
 - subsequent pipeline element pulls output from the filter
 - filter then obtains input from the previous pipeline element
- procedure (*push* filters)
 - previous pipeline element pushes input to the filter
 - filter then writes output to the subsequent pipeline element

Passive Filters: Push/Pull Pipelines



Push Pipeline

Activity is started with the data source pushing data to the pipeline.



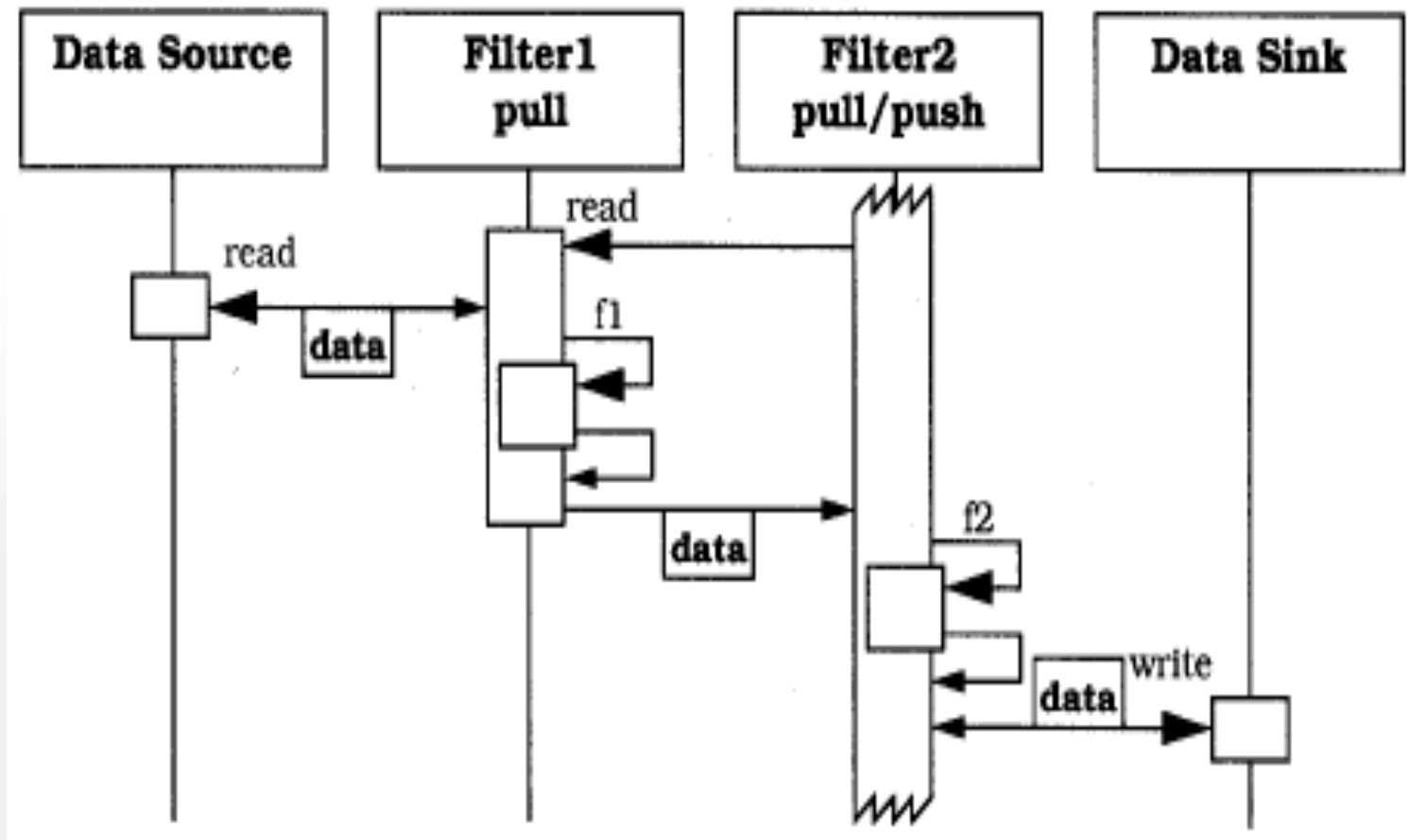
Pull Pipeline

Activity is started with the data sink pulling data from the pipeline.

Mixed Push/Pull-Pipeline

Mixed Pipeline

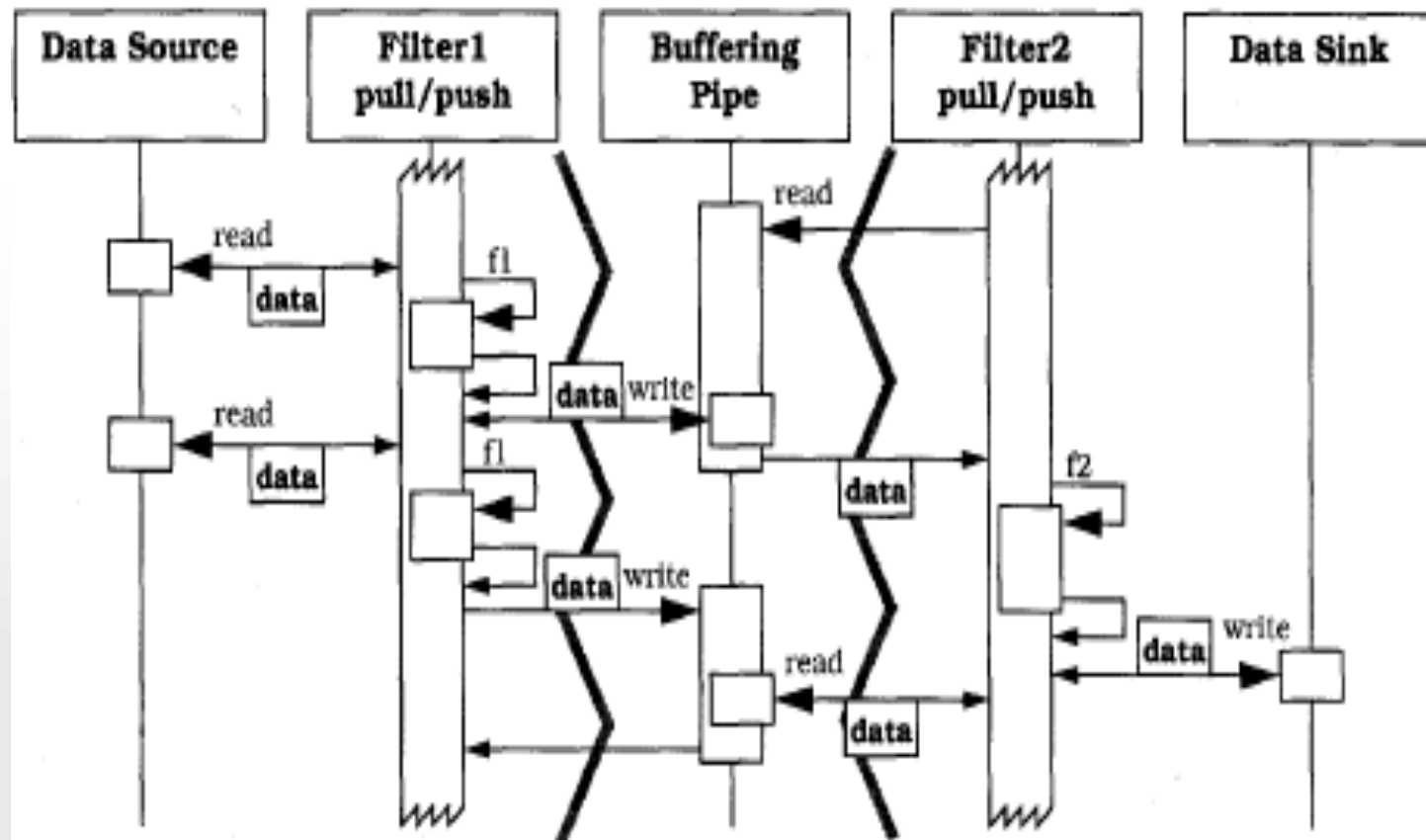
Activity is started and controlled by Filter2.



Active Filters with Pipes

Active Components

- all filters actively pull and push data in a loop
- filters run in separate threads
- synchronisation via buffering pipe



Pipes and Filters: Implementation

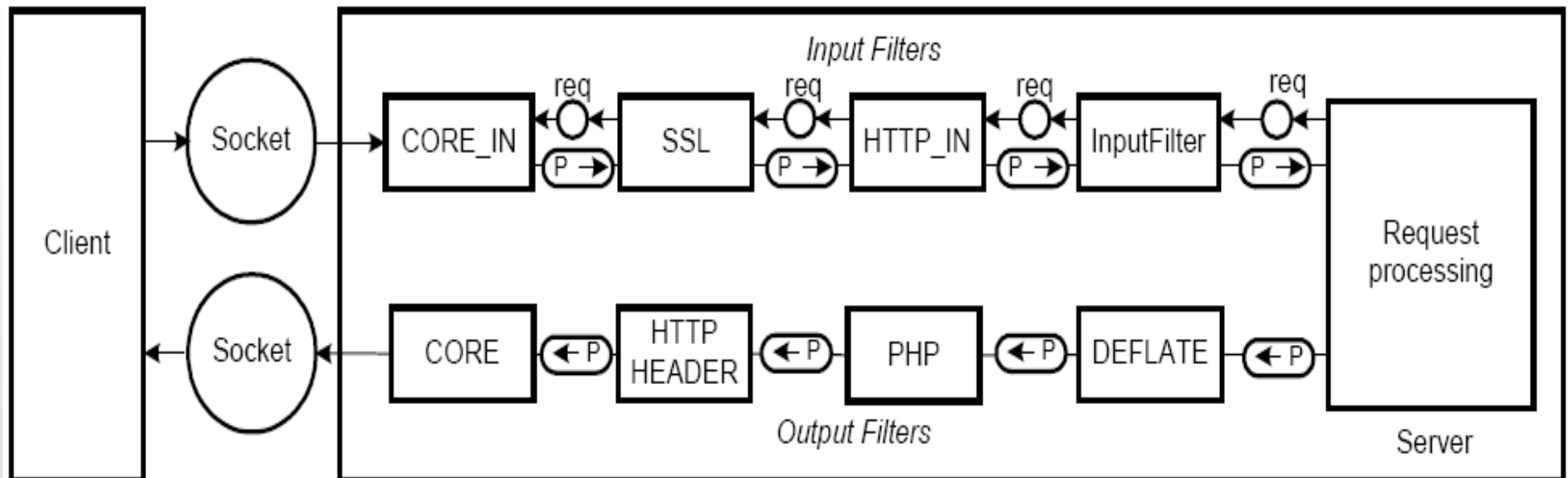
- divide the system's task into a sequence of **processing stages**
- define **data format** to be passed along each pipe
 - uniform or not
- decide how to implement **pipe connections**
 - direct calls between filters, or
 - separate pipe mechanism
- design and implement **filters**
 - pipe buffer size: bad performance when frequent context switches occur
- design **error handling**
 - hard because pipeline components should be decoupled

Known Uses

- Unix shell: look for a specific process and kill it
 - Unix shell filters are *active*

```
ps -ef  
| grep dial  
| kill `awk '{print $2}'`
```

- Apache web server



Pipes and Filters: Consequences

- **benefits**
 - flexibility by filter exchange
 - flexibility by recombination
 - reuse of filter components
 - efficiency by parallel processing
- **liabilities**
 - sharing state information expensive or inflexible
 - efficiency gain by parallel processing often an illusion
 - data transformation overhead if uniform data format
 - error handling very difficult

Microkernel Pattern

- separates minimal functional core from extended functionality and customer-specific parts
- serves as socket for plugging in extensions
- context
 - develop different applications in the same application domain that build on the same core functionality
- problem
 - application platform must cope with continuous hardware and software evolution
 - platform should be portable, extensible, adaptable to allow easy integration of emerging technologies
 - applications need to support different but similar application platforms
- solution: encapsulate fundamental services in a microkernel component

Microkernel Pattern: Collaboration

Class Microkernel

- responsibility
 - provides core mechanisms
 - offers communication facilities
 - encapsulates system dependencies
 - manages and controls resources
- collaborators: InternalServer (a.k.a. *subsystem*)
- main component of the pattern
- implements *atomic* services
 - "mechanisms" on which more complex functionalities, "policies" are built
- clients only see particular views of underlying application domain

Microkernel Pattern: Collaboration

Class InternalServer

- responsibility
 - implements additional services
 - encapsulates some system specifics
- collaborators: Microkernel
- also known as subsystem
- extends microkernel functionality
- microkernel invokes services
 - e.g., device drivers for particular video cards

Class ExternalServer

- responsibility
 - provides programming interfaces for its clients
- collaborators: Microkernel
- uses the microkernel to implement its own view of the underlying application domain
- receives service requests from clients

Microkernel Pattern: Collaboration

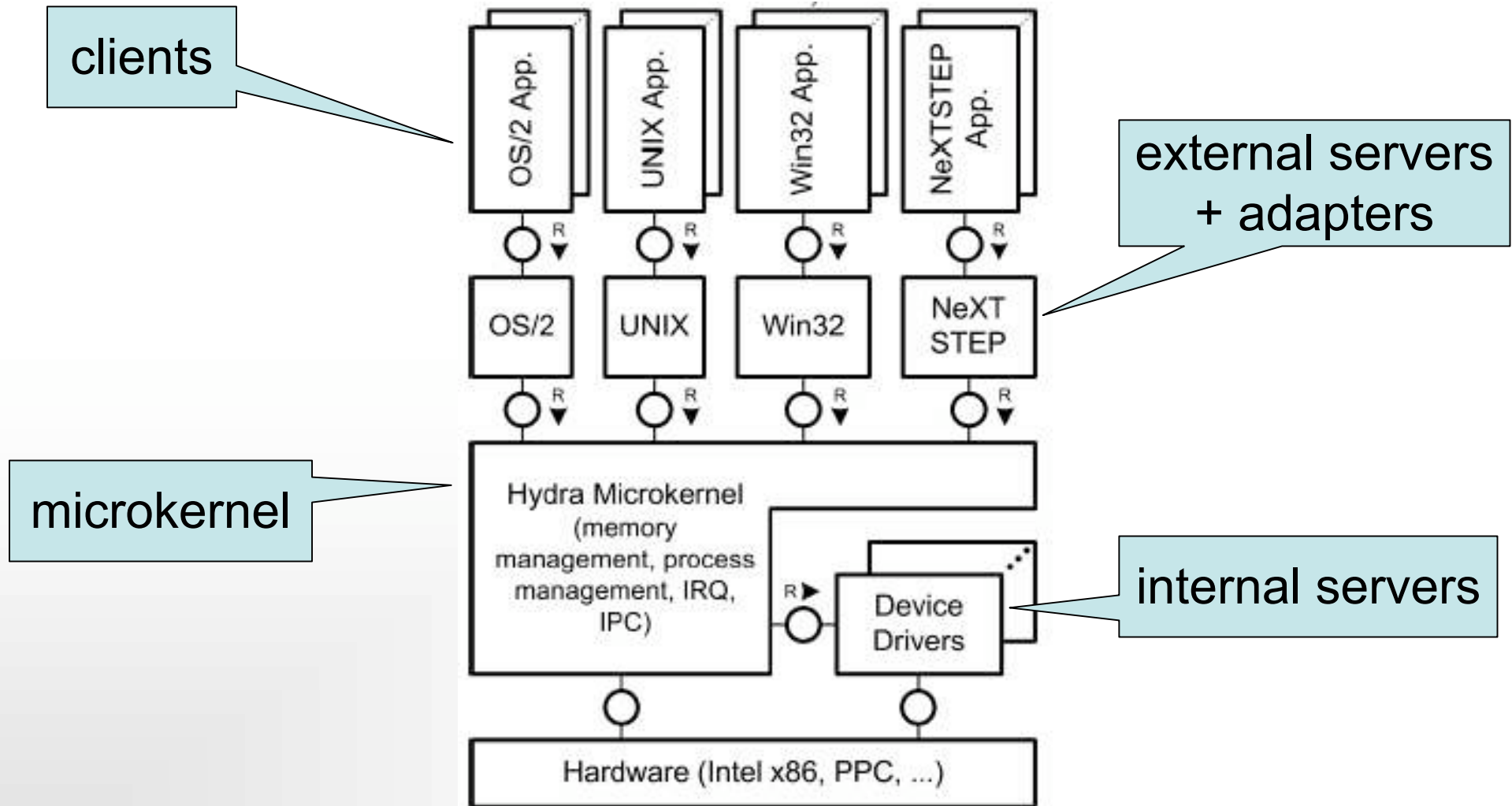
Class Client

- responsibility
 - represents an application
- collaborators: Adapter
(, ExternalServer)
- associated with exactly one external server via an adapter

Class Adapter

- responsibility
 - hides system dependencies such as communication facilities from the client
 - invokes methods of external servers on behalf of clients
- collaborators:
ExternalServer, Microkernel

Example: Hydra Operating Systems



Microkernel: Consequences

- portability
 - no need to port external servers or client applications if microkernel system is ported to new hardware/software environment
 - migrating the microkernel to new hardware environment only requires modifications to hardware-dependent parts
- flexibility and extensibility
 - adding/extending external servers
 - adding/extending internal servers

Microkernel: Consequences

- separation of policy and mechanisms
 - microkernel responsible for atomic mechanisms only
 - external servers can implement their own policy
 - increases maintainability, changeability
- performance
 - in general, monolithic systems are faster
 - need to optimise microkernel
- complexity of design and implementation
 - what is the core functionality?
 - requires in-depth domain knowledge