Software Architecture

FEUP-M.EIC-ASSO-2024

Ademar Aguiar, Neil Harrison

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

POSA Patterns: the books

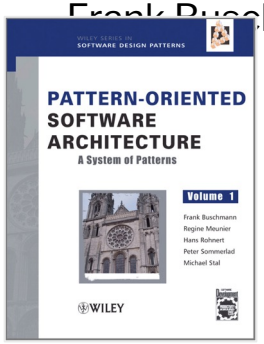
A System of Patterns. 1996. POSA1
Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal

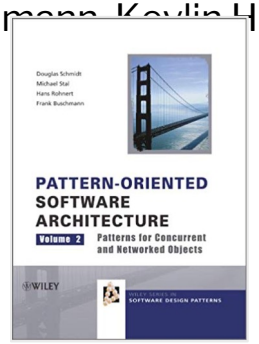
Patterns for Concurrent and Networked Objects. 2000. POSA2
Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann

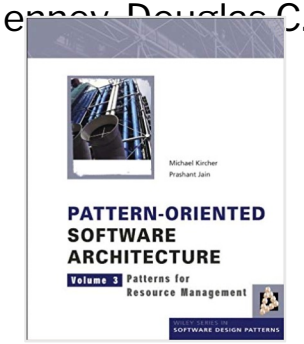
Patterns for Resource Management. 2007. POSA3
Michael Kircher, Prashant Jain

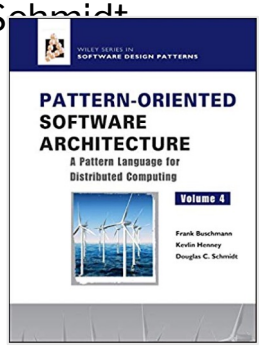
A Pattern Language for Distributed Computing. 2007. POSA4
Frank Buschmann, Kevlin Henney, Douglas C. Schmidt

On Patterns and Pattern Languages. 2007. POSA5
Frank Buschmann, Kevlin Henney, Douglas C. Schmidt

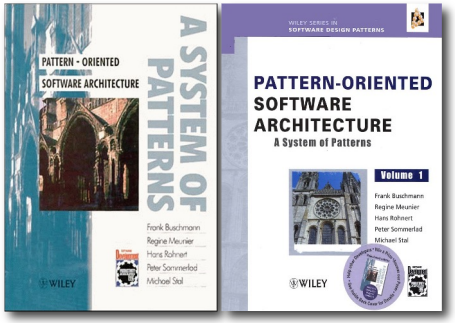








Pattern Oriented Software Architecture: A System of Patterns
(POSA 1)



Frank Buschmann
Regine Meunier
Hans Rohnert
Peter Sornmerlad
Michael Stal

Wiley, 1996

Pattern Oriented Software Architecture
A System of Patterns, Buschmann et al, 1996

A book about patterns for software architecture.

A book to support both novices and experts in software development.

It should support experts In the design of large-scale and complex software systems with defined properties.

It should also enable them to learn from the experience of other experts.

Architectural Patterns

Definition (Buschmann et al)

- An ***architectural pattern*** expresses a fundamental structural organization schema for software systems. It provides a set of predefined ***subsystems***, specifies their ***responsibilities***, and includes rules and guidelines for organizing the ***relationships*** between them.

Architectural templates

- Architectural patterns are templates for concrete software architectures.
- They specify the system-wide structural properties of an application, and have an impact on the architecture of its subsystems.

Fundamental design decisions

- The selection of an architectural pattern is therefore a fundamental design decision when developing a software system.

Four Categories of Architectural Patterns (POSA 1)

From Mud to Structure

- To help avoiding a 'sea' of components or objects by supporting a controlled decomposition of a task into cooperating subtasks.
- Patterns: **Layers** (31), **Pipes and Filters** (53), **Blackboard** (71).

Distributed Systems

- To help on architecting distributed systems.
- Patterns: **Broker** (99); refers to **Microkernel** (171), **Pipes and Filters** (53).

Interactive Systems

- To support the structuring of software systems that feature HCI.
- Patterns: **Model-View-Controller** (125), **Presentation-Abstraction-Control** pattern (145).

Adaptable Systems

- To support extension of applications and their adaptation to evolving technology and changing functional requirements.
- Patterns: **Reflection** (193), **Microkernel** (171).

Design Patterns

Definition (Gamma et al.)

- *A design pattern provides a scheme for **refining the subsystems or components** of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.*

Design patterns are medium-scale patterns

- **Smaller in scale** than architectural patterns
- Tend to be **independent of a particular programming language or programming paradigm**.
- The application of a design pattern has **no effect on the fundamental structure** of a software system, but may have a strong influence on the architecture of a subsystem.
- Many design patterns provide **structures for decomposing** more complex services or components.
- Others address the effective **cooperation** between them.

Categories of Design Patterns (POSA1)

Structural Decomposition

- To support a suitable decomposition of subsystems and complex components into cooperating parts.
- Patterns: **Whole-Part** (225)

Organization of Work

- To define how components collaborate together to solve a complex problem.
- Patterns: **Master-Slave** (245)

Access Control

- To guard and control access to services or components.
- Patterns: **Proxy** (263).

Management

- To handle homogenous collections of objects, services and components in their entirety.
- Patterns: **Command Processor** (277), **View Handler** (291).

Communication

- To organize communication between components.
- Patterns: **Forwarder-Receiver** (307), **Client Dispatcher-Server** (323), **Publisher-Subscriber** (339).

Pattern form

Name The name and a short summary of the pattern.

Also Known As Other names for the pattern, if any are known.

Example A real-world example demonstrating the problem and the pattern's need.

Context The situations in which the pattern may apply

Problem The problem the pattern addresses, including a discussion of its forces.

Solution The fundamental solution principle underlying the pattern.

Structure A detailed specification of the structural aspects of the pattern.

Dynamics Typical scenarios describing the run-time behavior of the pattern.

Implementation Guidelines or suggestions for implementing the pattern.

Example Resolved Discussion of any important aspects for resolving the example.

Variants A brief description of variants or specializations of a pattern.

Known Uses Examples of the use of the pattern, taken from existing systems.

Consequences The benefits the pattern provides, and any potential liabilities.

See Also References to patterns that solve similar problems, or that help refine it.

Basic and Simple Patterns first

Simple patterns are easy to understand and appear in many well-structured software systems.

Architectural patterns

- **Layers (31)**
- **Pipes and Filters (53)**

Design patterns

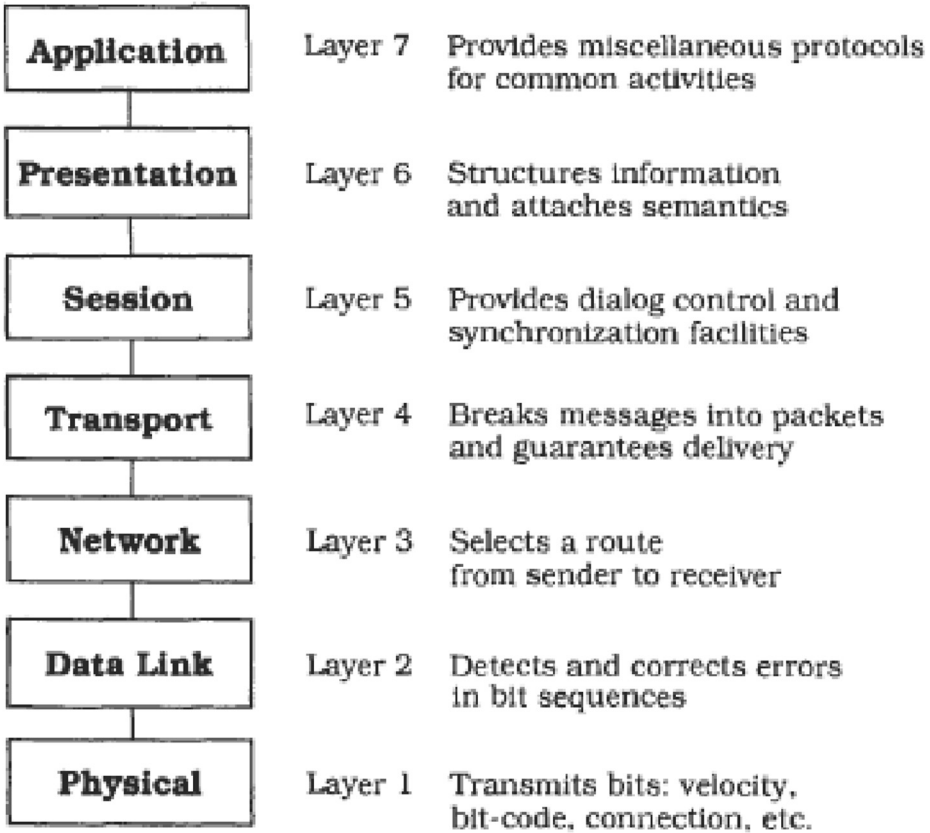
- **Proxy (263)**
- **Forwarder-Receiver (307)**

Layers (31)

*“The LAYERS pattern 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, granularity, hardware-distance, or other partitioning criteria"*

Layers (31) – Example

Example: networking protocols, OSI 7-layer model



Layers (31) – Context, Problem

Context

- A large system that requires decomposition

Problem

- Imagine that you are designing a system whose dominant characteristic is a **mix of low- and high-level issues**, where **high-level operations rely on the lower-level ones**.

  (...)

- A typical pattern of communication flow consists of **requests moving from high to low level**, and **answers to requests**, incoming data or notification about events **traveling in the opposite direction**.

  (...)

- Such systems often also require some **horizontal structuring** that is orthogonal to their vertical subdivision. This is the case where several operations are on the same level of abstraction but are largely independent of each other.

Layers (31) – Forces

Late source code changes should not ripple through the system. They should be confined to one component and not affect others.

Interfaces should be stable, and may even be prescribed by a standards body.

Parts of the system should be exchangeable without affecting the rest of the system. An extreme form of exchangeability might be a client component dynamically switching to a different implementation of a service that may not have been avallable at start-up.

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**, for example when a substantial amount of data must be transferred over several boundaries, or where there are many boundaries to cross.

The system will be built by a **team of programmers**, and work has to be subdivided along clear boundarles.

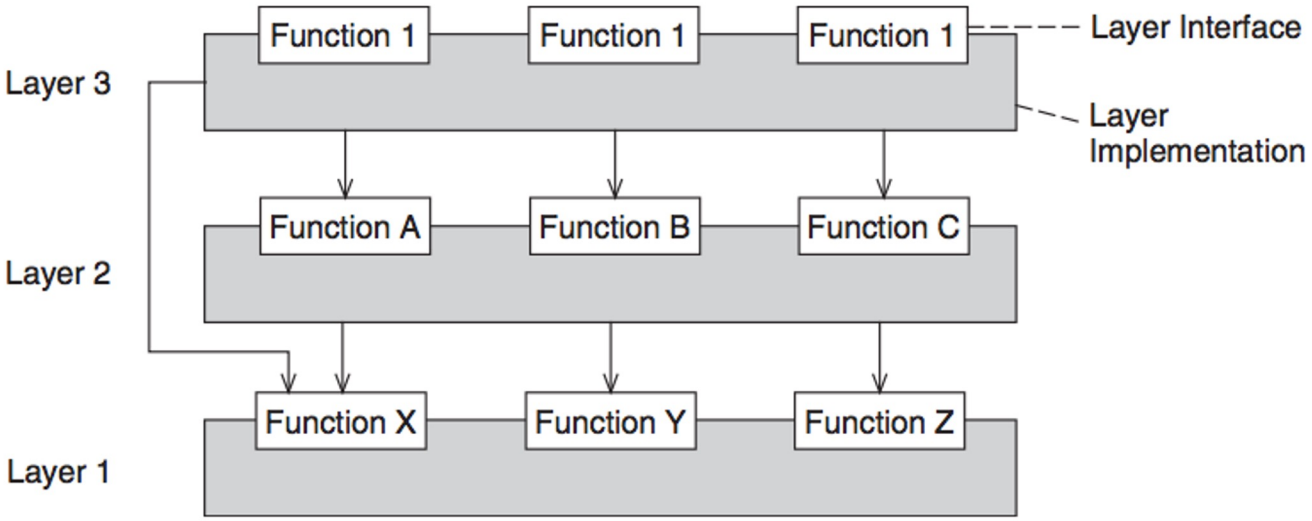
Layers (31) – Solution

Structure your system into **an appropriate number of layers** and place them on top of each other.

Start at the lowest level of abstraction-call it Layer 1. This is the base of your system.

Work your way up the abstraction ladder by putting Layer J on top of Layer J - 1 **until you reach the top level** of functionality-call it Layer N.

Layers – Structure



Layers (31) - Dynamics

Scenario I: “best-known”

- A client Issues a request to Layer N. Since Layer N cannot carry out the request on its own, it calls the next Layer N - 1 for supporting subtasks

Scenario II: “bottom-up communication”

- A chain of actions starts at Layer 1, and reports it to Layer 2.
- Top-down information and control flow are often described as **requests**.
- Bottom-up calls can be termed as **notifications.**

Scenario III: “subset of layers”

- Situations where requests only travel through a subset of the layers.
- Examples: caching mechanisms.

Scenario IV: “events stopped prematurely”

- An event is detected in Layer 1, but stops at Layer **3** instead of traveling all the way up to Layer N.

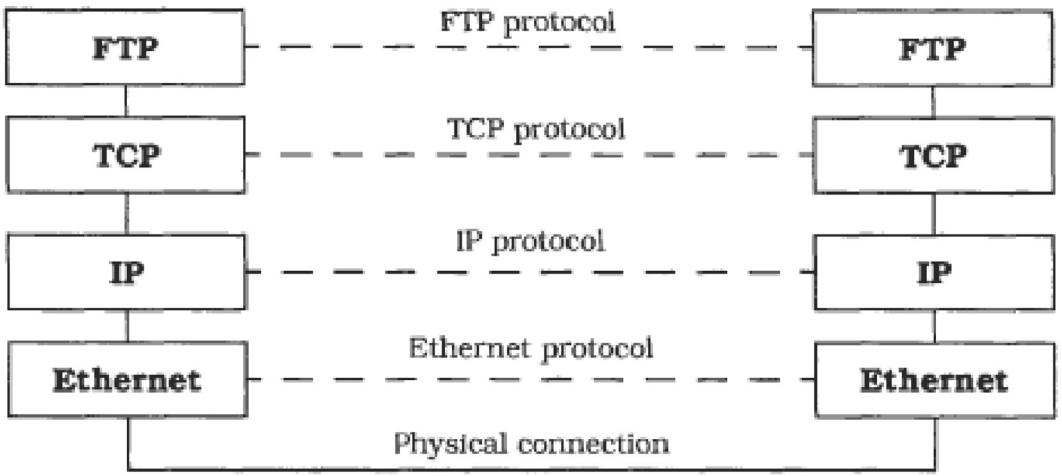
Scenario V: “two stacks of N layers communicating with each other”

- Well-known scenario from communication protocols, where the stacks are known as protocol stacks.

Layers (31) - Implementation

1. Define the abstraction criterion for grouping tasks into layers.
2. Determine the number of abstraction levels according to your abstraction criterion.
3. Name the layers and assign tasks to each of them.
4. Specify the services.
5. Refine the layering.
6. Specify an interface for each layer.
7. Structure individual layers.
8. Specify the communication between adjacent layers.
9. Decouple adjacent layers.
10. Design an error-handling strategy.

Layers (31) – Example Resolved



Layers (31) - Variants

Relaxed Layered System

- A variant of the Layers pattern less restrictive about the relationship between layers.
- In a Relaxed Layered System each layer **may use the services of all layers below it**, not only of the next lower layer.
- A layer may also be partially opaque meaning that **some of its services are only visible to the next higher layer**, while others are visible to all higher layers.
- The **gain of flexibility and performance** in a Relaxed Layered System is paid for by a **loss of maintainability** !!!

Layering Through Inheritance

- In this variant lower layers are implemented as base classes.
- An advantage of this scheme is that higher layers can modify lower-layer services according to their needs.
- A drawback is that such an inheritance relationship closely ties the higher layer to the lower layer.

Layers (31) – Known Uses

Virtual Machines

- Platform specific code (JVM)
- Bytecode instructions (compiled programs)
- Java programs (source code)

APIs

- Operating system functionalities
- High-level functionalities
- Utility functionalities

Information Systems

- Presentation
- Application logic
- Domain layer
- Database

Layers (31) – Consequences

Benefits

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

Liabilities

- Cascades of changing behavior
- Lower efficiency
- Unnecessary work
- Difficulty of establishing the correct granularity of layers

Layers (31) – See also

Composite message pattern

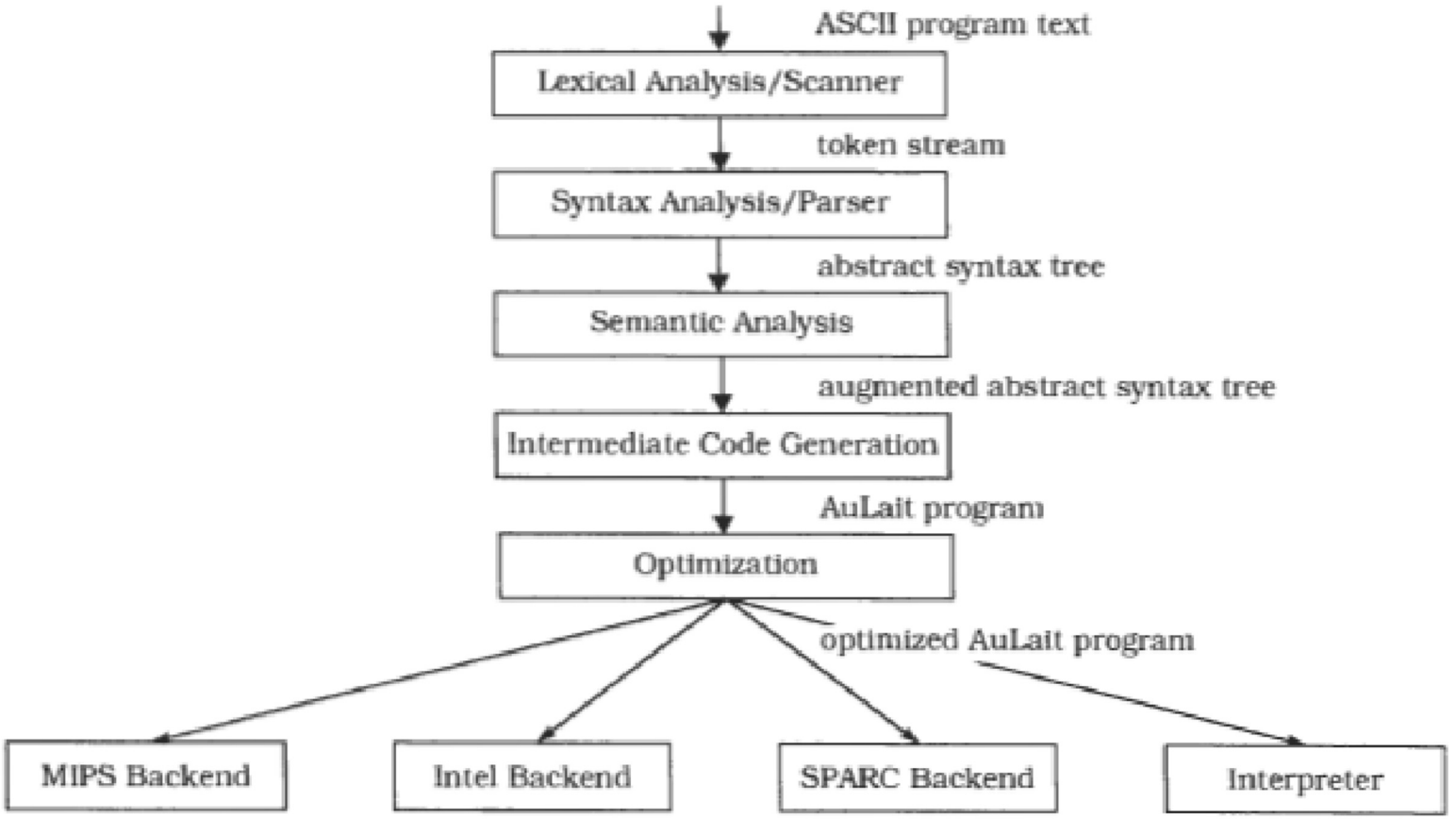
A Microkernel architecture (171)

The Presentation-Abstraction-Control architectural pattern (145)

Pipes and Filters (53)

"The Pipes and Filters architectural pattern 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"*

Pipes and Filters (53) - Example



Pipes and Filters (53) – Context, Problem

Context

- Processing data streams.

Problem

- Imagine you are building a system that must process or transform a stream of input data.
- Implementing such a system as a single component may not be feasible for several reasons:
  - the system has to be built by several developers,
  - the global system task decomposes naturally into several processing stages,
  - and the requirements are likely to change.
- You therefore plan for future flexibility by exchanging or reordering the processing steps.

Pipes and Filters (53) - Forces

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 components.

Non-adjacent processing steps do not share information.

Different sources of input data exist.

It should be possible **to present or store 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 or quasi-parallel.

Pipes and Filters (53) - Solution

The Pipes and Filters architectural pattern divides the task of a system into several **sequential processing steps**.

These steps are connected by the data flow through the system-the output data of a step is the input to the subsequent step.

Each processing step is implemented by a **filter** component.

A filter consumes and delivers data incrementally - in contrast to consuming all its input before producing any output-to achieve low latency and enable real parallel processing.

The input to the system is provided by a data source such as a text file.

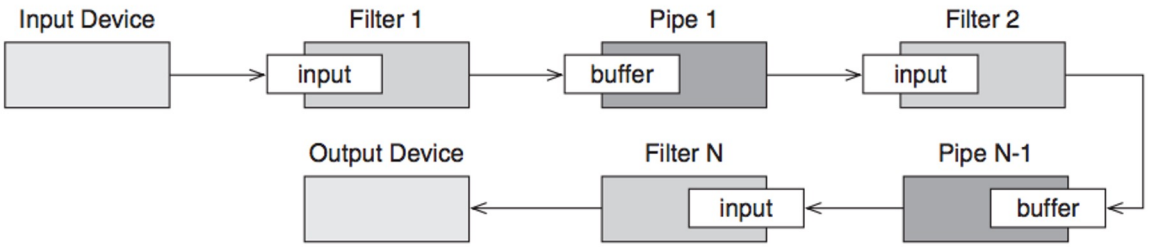
The output flows into a data sink such as a file, terminal, animation program and so on.

The data source, the filters and the data sink are connected sequentially by **pipes**.

Each pipe implements the data flow between adjacent processing steps.

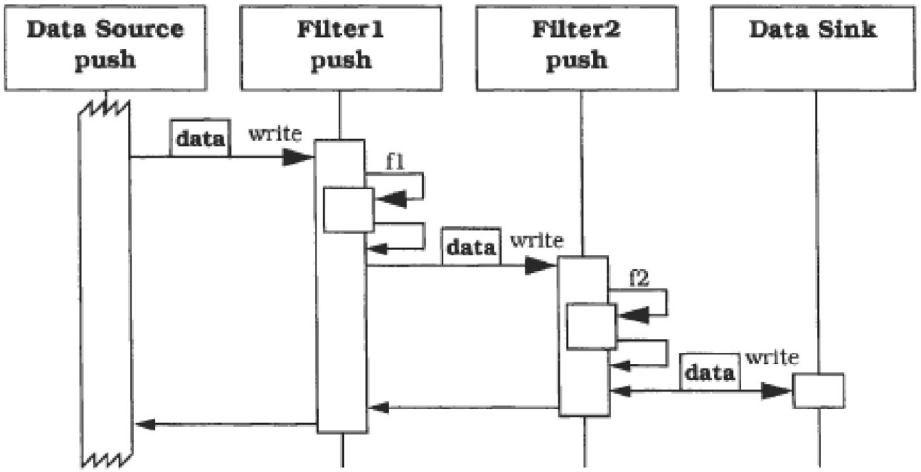
The sequence of filters combined by pipes is called a **processing pipeline**.

Pipes and Filters – Structure

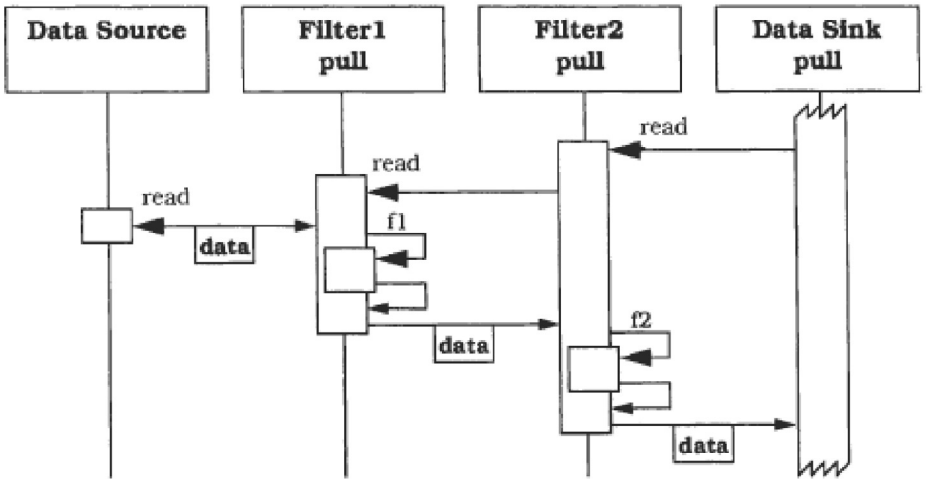


Pipes and Filters (53) - Dynamics

Scenario I: “push pipeline”

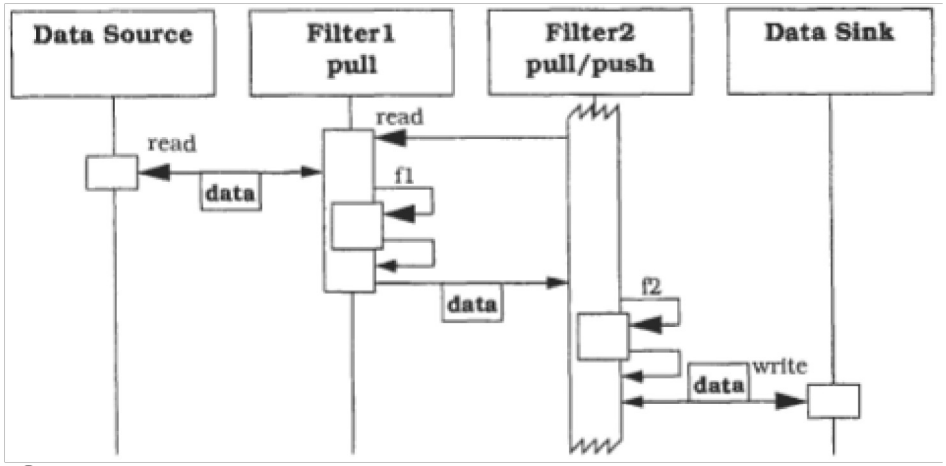


Scenario II: “pull pipeline”

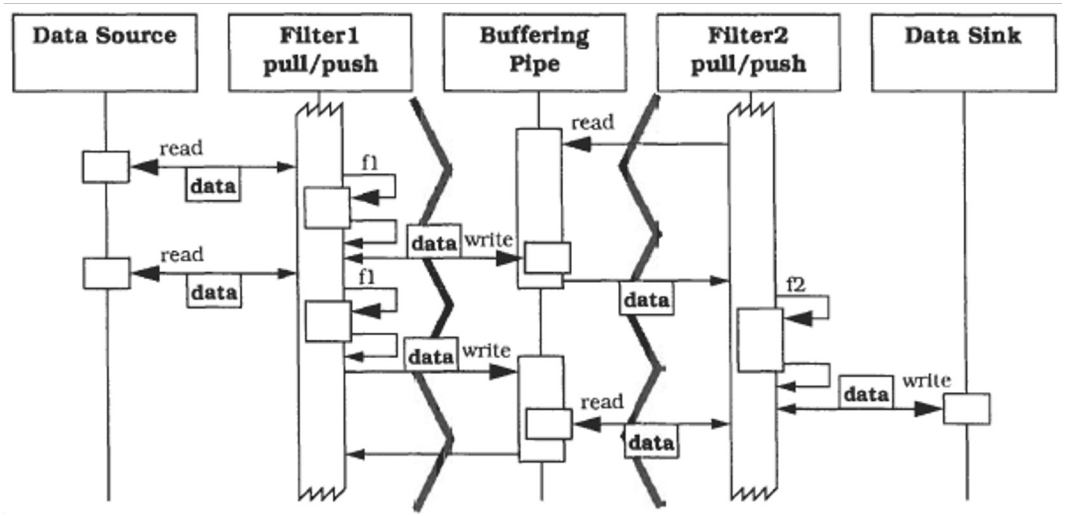


Pipes and Filters (53) - Dynamics

Scenario III: “push-pull pipeline”



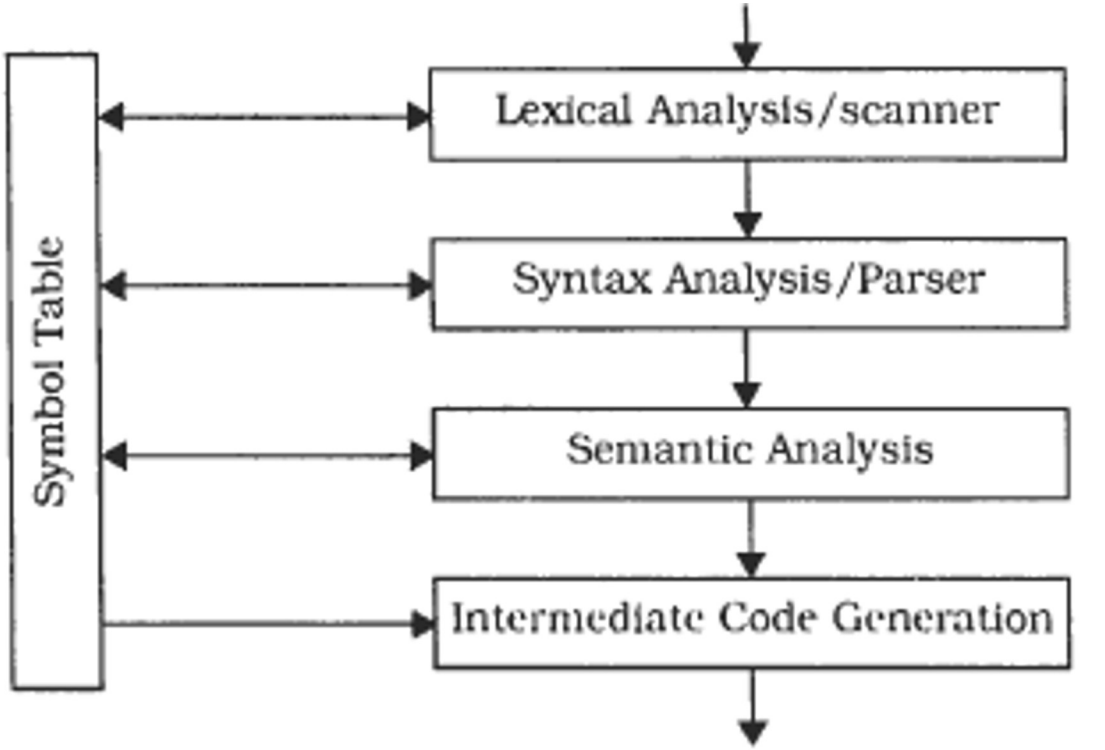
Scenario IV: “loop push-pull pipeline”



Pipes and Filters (53) – Implementation

1. Divide the system's task into a sequence of processing stages.
2. Define the data format to be passed along each pipe.
3. Decide how to implement each pipe connection.
4. Design and implement the filters.
5. Design the error handling.
6. Set up the processing pipeline.

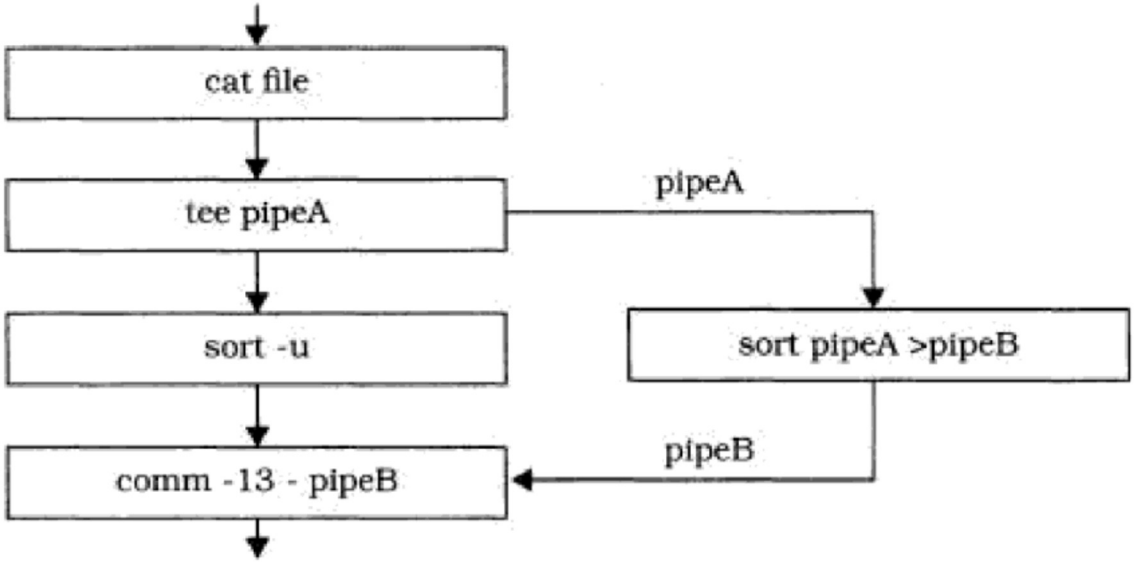
Pipes and Filters (53) – Example Resolved



Pipes and Filters (53) – Variants

Tee and Join pipeline

```
# first create two auxiliary named pipes to be used
mknod pipeA p
mknod pipeB p
# now do the processing using available UNIX filters
# start side fork of processing in background:
sort pipeA > pipeB &
# the main pipeline
cat file | tee pipeA | sort -u | comm -13 - pipeB
```



Pipes and Filters (53) – Known Uses

UNIX

CMS pipelines

LASSPTools

Pipes and Filters (53) – Consequences

Benefits

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

Liabilities

- Sharing state information is expensive or inflexible
- Efficiency gain by parallel processing is often an illusion (data transfer costs, filters consuming all the input at once, context-switching costs, filter synchronization difficulties)
- Data transformation overhead
- Error handling (the Achilles' heel...)

Pipes and Filters (53) – See also

Layers (31)

- The **Layers** pattern (31) is better suited to systems that require reliable operation, because it is easier to implement error handling than with Pipes and Filters.
- However, Layers lacks support for the easy recombination and reuse of components that is the key feature of the Pipes and Filter pattern.

Team work

Review your architecture of the Library System

- Revisit the several ***subsystems*** and how they are ***connected***
- What are the main ***architectural and design challenges***?
- Which ***patterns*** can you use to address those challenges?

Reference

Pattern Oriented Software Architecture, A System of Patterns, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sornmerlad, Michael Stal, Wiley, 1996.

Thank you!