4. Pipes & Filters Architectures

1. Motivation and Fundamental Concepts
2. Revisiting Object-Oriented Analysis, Design, and Implementation
3. Design Patterns
4. Pipes & Filters Architectures
5. Event-based Architectures
7. Framework Architectures
8. Component Architectures
Learning Objectives of Chapter 4

Students should be able

- to describe **pipes & filters architectures** and their variations,
- to explain the (dis-)advantages of pipes & filters architectures compared with object-oriented systems architectures,
- to relate pipes & filters to the design of the Java IO libraries and
- to apply the **decorator pattern** in their own system designs.

**Recommended Reading**

- [ShGa96] Section 2 & Section 4
- [GHJV95] Iterator (p. 257)
- [GHJV95] Decorator (p. 175)
- [BMRSS96] Section 2.1 + 2.2

The Pipes and Filters Architectural Pattern

**System Components**: Filters process streams of data
- A filter encapsulates a processing step (algorithm)

**Topology**: A Pipe connects a source and a sink component
- A pipe delivers an (infinite) stream of data

**Interaction:**
- Data (message) exchange
- Filters can be recombined freely to build families of related systems.
- Purely data-driven interaction.

**Example:**
- Unix shell: `cat input.txt | grep "text" | sort > output.txt`

![Diagram](https://via.placeholder.com/150)
Example: P&F Compiler Architecture (1)

- Sources & Sinks, Input & Output Streams
- Flexible composability
- Aggregation / Decomposition of Filters

Diagram:

- Scanner
- Token Stream
- Abstract Syntax Tree Nodes
- Bytecode Optimizer
- Machine Code Stream
- Error Message Stream

Example: P&F Compiler Architecture (2)

```java
class Math {
    public static int min(int a, int b) {
        return a < b ? a : b;
    }
    // min
    /// class Math
}
```

Diagram:

- Class Math
- Method min
- Bytecode Stream
- Machine Code Stream
- Error Message Stream

Example:

```
mov ax, [BP]
mov dx, [BP-2]
0 iload_0
1 iload_1
2 if_icmpge 7
5 iload_0
6 ireturn
7 iload_1
8 ireturn
9 30 expected "j"
4 32 incompatible type
```
Example: Virtual Instrumentation (1)

Product: National Instruments, LabVIEW

Example: Virtual Instrumentation (2)
### Example: Java Studio for Java Beans

![Java Studio Diagram](image)

#### Class Responsibility Cards (CRC)

<table>
<thead>
<tr>
<th>Class</th>
<th>Responsibility</th>
<th>Collaborators</th>
<th>Class</th>
<th>Responsibility</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Filter</strong></td>
<td>• Gets input data</td>
<td>• Pipe</td>
<td><strong>Pipe</strong></td>
<td>• Transfers data.</td>
<td>• Data source</td>
</tr>
<tr>
<td></td>
<td>• Performs a function on its input data</td>
<td></td>
<td></td>
<td>• Buffers data</td>
<td>• Data sink</td>
</tr>
<tr>
<td></td>
<td>• Supplies output data</td>
<td></td>
<td></td>
<td>• Synchronizes active neighbors</td>
<td>• Filter</td>
</tr>
<tr>
<td><strong>Data Source</strong></td>
<td>• Delivers a data stream</td>
<td>• Pipe</td>
<td><strong>Data Sink</strong></td>
<td>• Consumes a data stream</td>
<td>• Pipe</td>
</tr>
</tbody>
</table>
Driving Forces leading to P&F Architectures

- Future system enhancements should be possible by exchanging processing steps or by a recombination of steps, *even by users of the systems*
- Small processing steps are easier to reuse in different contexts than large components (e.g., pretty-printer in compiler)
- Non adjacent processing steps do not share information
- Different sources of input data exist (network, terminal, file, ...)
- It should be possible to present or store final results in various ways
- Explicit storage of intermediate results for further processing may be introduced.
- Synchronization of processing steps is not essential
  - sequential execution
  - parallel execution (pipelining)
- There is no need for a closed “feedback loop”

Filter

Basic activities of filters (often combined in a single filter)

- enrich input data (e.g., by data from a data store or computed values)
- refine input data (e.g., filter out “uninteresting” input, sort input)
- transform input data (e.g., from streams of words to streams of sentences)

There are two strategies to construct a filter:

- An **Active Filter** drives the data flow on the pipes
- A **Passive Filter** is driven by the data flow on the (input/output) pipes
- In a P&F-Architecture there has to be at least one active filter
- This active filter can be the environment of the system (e.g., user-input)

(Persistent) collections can be used to buffer the data passed through pipes:

- files, arrays, dictionaries, trees, ...
**Pipe**

- A pipe is a first-class object
- A pipe transfers data from one data source to one data sink
- A pipe may implement a (bounded / unbounded) buffer

Pipes between two threads of a **single process** (e.g. Java Streams)
- stream may contain references to shared language objects

Pipes between two processes on a **single host** computer (e.g. Unix Named Pipes)
- stream may contain references to shared operating system objects (files!)

Pipes between two processes in a **distributed system** (e.g. Internet Sockets)
- stream contents limited to “raw bytes”
- protocols implement higher-level abstractions (e.g. pass pipes as references, pass CORBA object references)

---

**Composition Rules**

- **Sequential Composition**
  Unix:  \( F_1 \| F_2 \)

- **Parallel Composition**
  Unix:  \( F_1 \& F_2 \)

- **Tee & Join**

- **Restriction to Linear Composition**
Example: Tee & Join in Unix

**Task:** Print a sorted list of words that occur more than once

```
mknod pipeA p
mknod pipeB p
sort pipeA > pipeB &
cat file1 | tee pipeA | sort -u | comm -13 - pipeB > file2
```

---

Active Filters

- The filter is an active process or thread that performs a loop, pulling its input from and pushing its output down the pipeline

```
while (true) {
    Element x = inputPipe.read (...);
    outputPipe.write (f (x));
}
```

- Many command-line-oriented operating systems provide an input stream and an output stream as a parameter to each program. (standard input / standard output)
### Producer / Consumer Problem

- Concurrent (autonomous) activity of producer and consumer
- Data-driven synchronization has to be supported by the operating system (process scheduler) or the programming-language runtime system (thread scheduler)

![Petri-Net model](image)

Petri-Net models possible concurrent execution paths.

---

### Example: Production process (1)

For Details, see: Informatik für Ingenieure II Chapter 9

![Production process diagram](image)
Example: Production process (2)

Order → Customer acquired → Parallelism created!

Wings ordered → build wings

Body ordered → build body

Wheels ordered → build wheels

Wheels ready → Body ready → Final assembly

Wings ordered → Body ordered → Wheels ordered

Wings ordered → build wings

Body ordered → build body

Wheels ordered → build wheels

Wheels ready → Body ready → Final assembly

Customer acquired

Final assembly → Finished airplane

Example: Production process (3)

Order → Customer acquired → Synchronisation through transition "Final assembly"!

Wings ordered → build wings

Body ordered → build body

Wheels ordered → build wheels

Wings ordered → build wings

Body ordered → build body

Wheels ordered → build wheels

Wings ready → Body ready → Final assembly

Wings ordered → Body ordered → Wheels ordered

Wheels ready → Body ready → Final assembly

Final assembly → Finished airplane
Example: Production process (4)

- Customer acquired
- Order
- Wings ordered
  - build wings
- Wings ready
- Body ordered
  - build body
- Body ready
- Wheels ordered
  - build wheels
- Wheels ready
- Final assembly
- Finished airplane

Synchronisation through transition "Final assembly!"

Example: Production process (5)

- Customer acquired
- Order
- Wings ordered
  - build wings
- Wings ready
- Body ordered
  - build body
- Body ready
- Wheels ordered
  - build wheels
- Wheels ready
- Final assembly
- Finished airplane

Synchronisation through transition "Final assembly!"
Example: Production process (6)

```
Customer acquired

Order

Wings ordered
build wings
Wings ready

Body ordered
build body
Body ready

Wheels ordered
build wheels
Wheels ready

Final assembly
Finished airplane
```

Alternative Step in Production process

The selection of the transition that triggers next is non-deterministic!
Passive Filters: Pull Strategy

- The filter is a passive object that is driven by the subsequent pipeline element that pulls output data from the filter.

![Diagram of Passive Filters]

InputStream
read () : int

javax.io.InputStream in the Java Class Library

ByteArray
InputStream
StringBuffer
InputStream
File
InputStream
Piped
InputStream

MyPullFilter
int read () {
    int x = myInput.read () ;
    return f (x) ;
}

Scenario: Two Passive Pull Filters

![Diagram of Two Passive Pull Filters]

Data Source (pull)
Filter 1 (pull)
Filter 2 (pull)
Data Sink (pull)

read
read
read

data
f1

data
f2

data
Passive Filters: Push Strategy

- The filter is a passive object that is driven by the previous pipeline element that pushes input data into the filter.

```java
OutputStream
write (b : int)
```

```java
java.io.OutputStream in the Java Class Library
```

MyPushFilter

```java
void write (int b) {
    myOutput.write (f (b));
}
```

Treat an active filter like a passive one

Scenario: Two Passive Push Filters

Data Source
(push)

Filter 1
(push)

Filter 2
(push)

Data Sink
(push)
## A First Comparison of Architectures

<table>
<thead>
<tr>
<th><strong>OO System Architecture</strong></th>
<th><strong>Pipes &amp; Filter Architecture</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects passed as arguments of messages by reference</td>
<td>Data values passed as copies between filters</td>
</tr>
<tr>
<td>“Shared everything” (data, code, threads)</td>
<td>“Shared nothing”</td>
</tr>
<tr>
<td>Very large number of object links</td>
<td>Very small number of pipes</td>
</tr>
<tr>
<td>Object creation defined by other objects</td>
<td>Filters and topology defined “outside” of the filters</td>
</tr>
<tr>
<td>Frequent bidirectional object exchange between objects</td>
<td>Unidirectional data flow</td>
</tr>
<tr>
<td>Focus on control flow (mostly sequential)</td>
<td>Focus on data flow (highly concurrent)</td>
</tr>
<tr>
<td>“Everything is an object”</td>
<td>Filters have a complex internal structure that cannot be described by pipes and filters alone</td>
</tr>
<tr>
<td>small-grain system structuring</td>
<td>large-grain system structuring</td>
</tr>
<tr>
<td>dynamic object links</td>
<td>mostly static pipe topology</td>
</tr>
</tbody>
</table>

### Implementation Issues

- Identify the processing steps (re-using existing filters as far as possible)
- Define the data format to be passed along each pipe
- Define end-of-stream symbol
- Decide how to implement each pipe connection (active / passive)
- Design and implement the filters
- Design error handling
Stream Data Formats

Tradeoff
- compatibility & reusability vs. type safety
  - “everything is a stream”
  - “stream of Persons, stream of Texts”

Popular Stream Data Formats
- raw byte stream
- stream of ASCII text lines with line separator
- record stream (record attributes are strings, separated by tabulator or comma)
- nested record stream (record attribute is in turn a sequence)
- stream representing a tree traversal (inner nodes / leaf nodes enumerated in preorder, postorder, inorder)
- typed stream with a header containing its type information (e.g. column headings)
- event streams (event name and event arguments)
  (internal streams in a programming language: stream of object references)

Benefits of P&F Architectures
- No intermediate data structures necessary (but possible)
  (Pipeline processing subsumes batch processing)
- Flexibility through filter exchange
- Flexibility by recombination
- Reuse of filter components
- Rapid prototyping
- Parallel processing in a multiprocessor environment
Limitations of P&F Architectures

- Sharing state information is expensive or inflexible
- Efficiency loss in a single processor environment
  - cost of transferring data
  - data dependencies between stream elements (e.g. sorting, tree traversal)
  - cost of context switching (in particular for non-buffered pipes)
- Data transformation overhead
  - data on the stream
  - objects in memory
- Difficulty of coordinated error handling

P&F in Java: The Decorator Design Pattern

Intent:
- Attach additional responsibilities to an object dynamically. Combine multiple responsibilities without subclassing.

Motivation:
Possible responsibilities of a Pipe / Stream
- Buffering the data
- Formatting / parsing of integers, floating point numbers, ...
- Keeping track of the current line number (for error reporting)
- Provide a single character "lookahead" without actually consuming the character

```
myStream
  ↕ aLineNumberInputStream
  ↗ aBufferedStream
  ↘ aStringBufferInputStream
    ↗ myStringBuffer
```
Remember: The Decorator Pattern

```
Component
  operation()

ConcreteComponent
  operation()

Decorator
  operation()

ConcreteDecoratorA
  operation()
  addedState

ConcreteDecoratorB
  operation()
  addedBehavior()
  super.operation();
```

Decorating Pattern for Input / Output Streams

Classes from the Java class library

Component = InputStream / OutputStream  
Concrete Component = FileInputStream, ... / FileOutputStream, ...
Decorator = FilterInputStream / FilterOutputStream  
ConcreteDecorator = BufferedInputStream / PushbackInputStream /  
BufferedOutputStream, CipherInputStream / CipherOutputStream,  
DataInputStream / DataOutputStream, LineNumberInputStream, ...

The classes FilterInputStream and FilterOutputStream define the common  
interface (and default implementations).

Programmers have to compose these streams dynamically:

```java
myStringBuffer = new StringBuffer ("This is a sample string to be read")
filterInputStream myStream = new LineNumberInputStream (new BufferedInputStream (myStringBuffer));
myStream.read ()
myStream.getLine ()
```