Software Architectures

2 SWS Lecture 1 SWS Lab Classes

Hans-Werner Sehring
Miguel Garcia

Arbeitsbereich Softwaresysteme (STS)
TU Hamburg-Harburg

HW.Sehring@tuhh.de
Miguel.Garcia@tuhh.de
http://www.sts.tu-harburg.de/teaching/ss-05/SWArch/entry.html

Summer term 2005

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
6. Layered Architectures & Persistence Management
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
- Bruce Eckel: Thinking in Java, Prentice Hall, 1998.
  Chapter 10: The Java IO-System

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

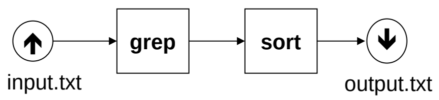
- A pipe delivers an (infinite) stream of data

Interaction:

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

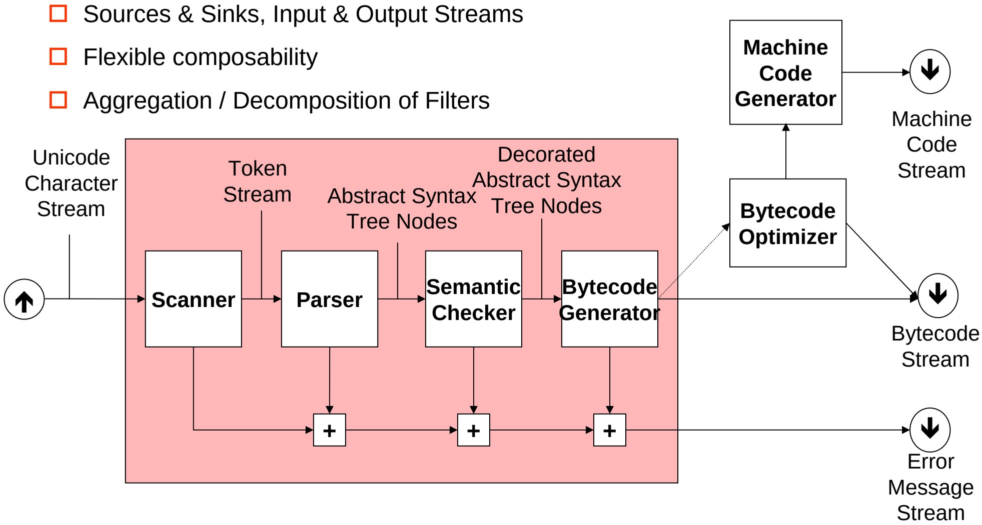
Example:

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



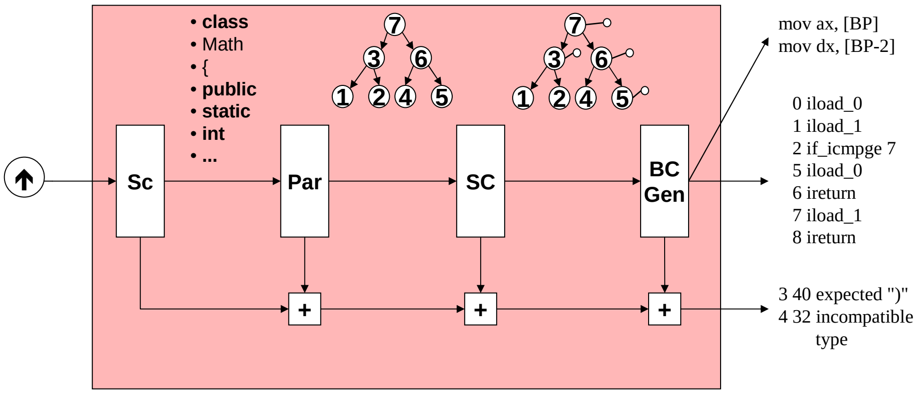
Example: P&F Compiler Architecture (1)

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



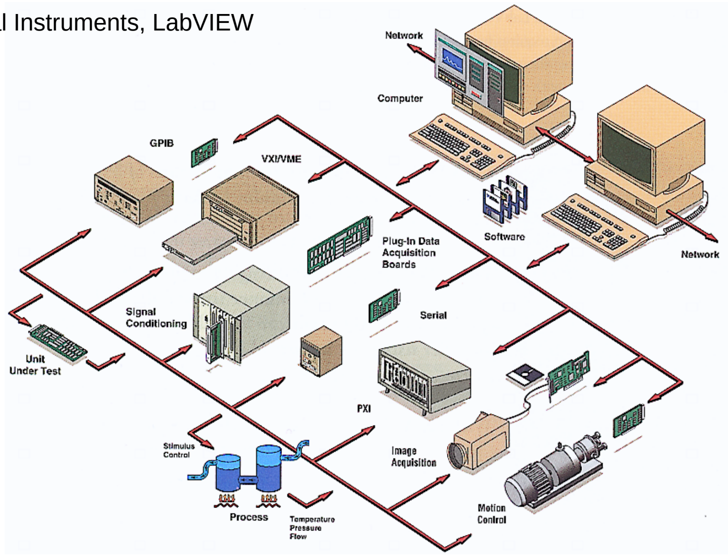
Example: P&F Compiler Architecture (2)

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

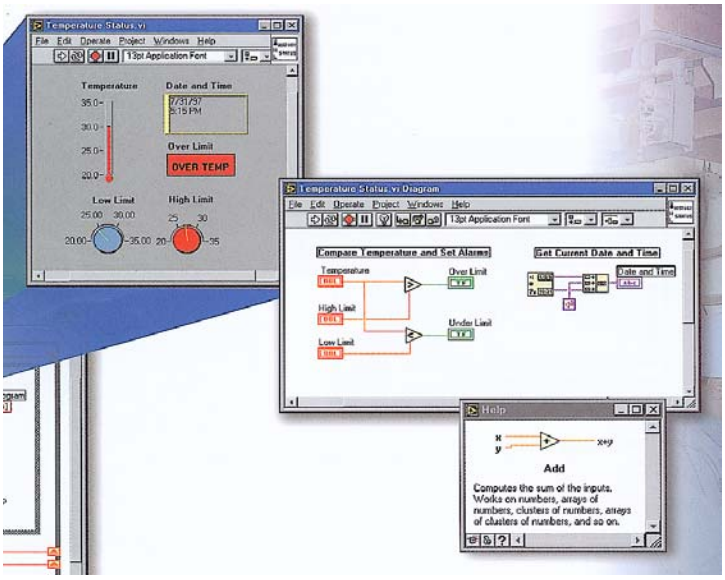


Example: Virtual Instrumentation (1)

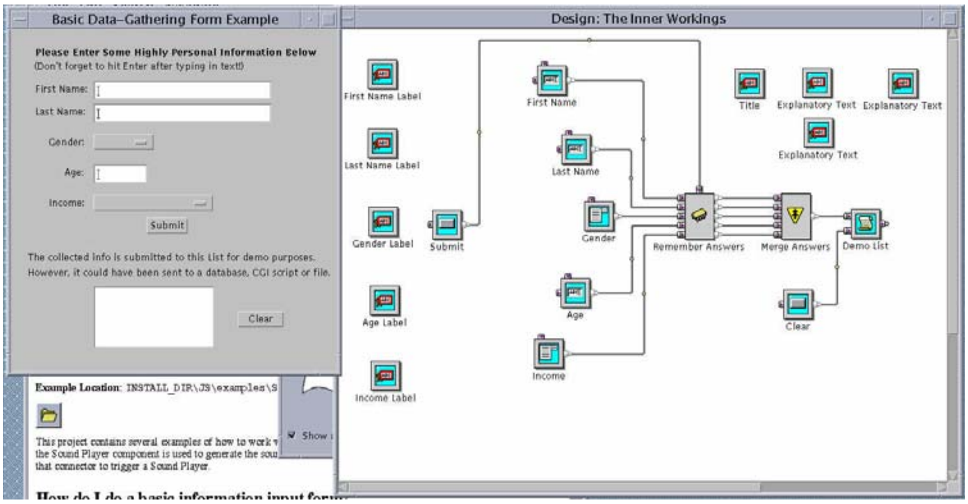
Product: National Instruments, LabVIEW



Example: Virtual Instrumentation (2)



Example: Java Studio for Java Beans



Class Responsibility Cards (CRC)

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

Class: Pipe	**Collaborators:**
Responsibilty • Transfers data. • Buffers data • Synchronizes active neighbors	• Data source • Data sink • Filter

Class: Data Source	**Collaborators:**
Responsibilty • Delivers a data stream	• Pipe

Class: Data Sink	**Collaborators:**
Responsibilty • Consumes a data stream	• Pipe

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:

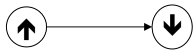
- An **Active Filter** drives the data flow on the pipes
- A **Passive Filter** is driven by the data flow on the (input/output) pipes
- In a P&F-Architecture there has to be at least one active filter
- This active filter can be the environment of the system (e.g. user-input)

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

- files, arrays, dictionaries, trees, ...

Pipe

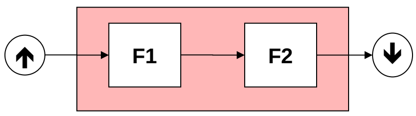
- A pipe is a first-class object
- A pipe transfers data from one data source to one data sink
- A pipe may implement a (bounded / unbounded) buffer



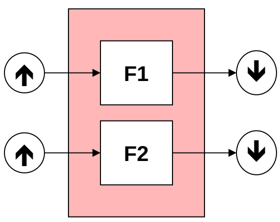
- Pipes between two threads of a **single process** (e.g. Java Streams)
  - stream may contain references to shared language objects
- Pipes between two processes on a **single host** computer (e.g. Unix Named Pipes)
  - stream may contain references to shared operating system objects (files!)
- Pipes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

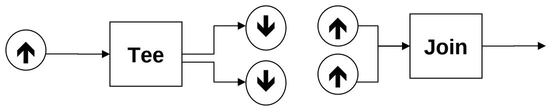
- Sequential Composition
  Unix: F1 | F2



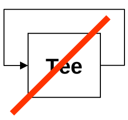
- Parallel Composition
  Unix: F1 & F2



- Tee & Join



- Restriction to **Linear Composition**



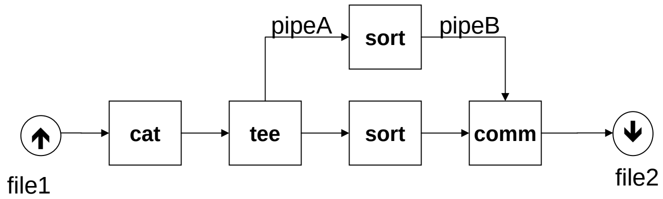
Example: Tee & Join in Unix

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
```



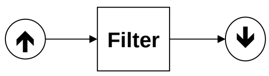
Active Filters

- 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)) ;
}
```

Blocking read

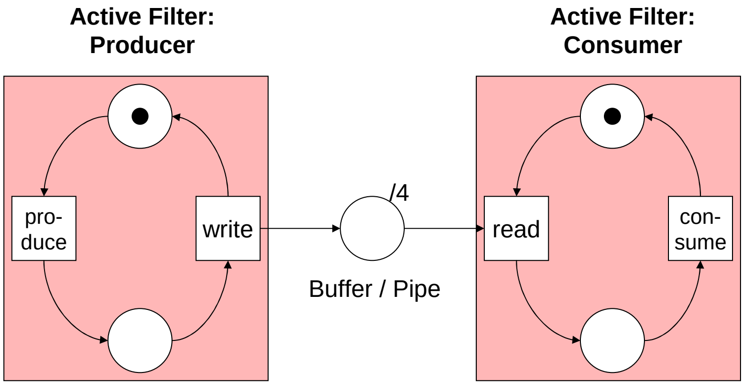
Blocking write



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

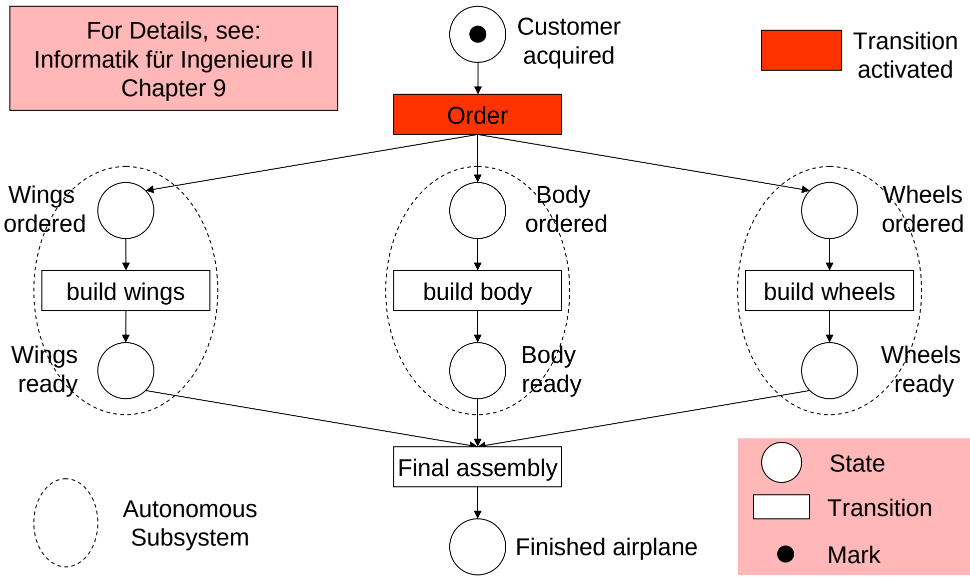
Producer / Consumer Problem

- 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)

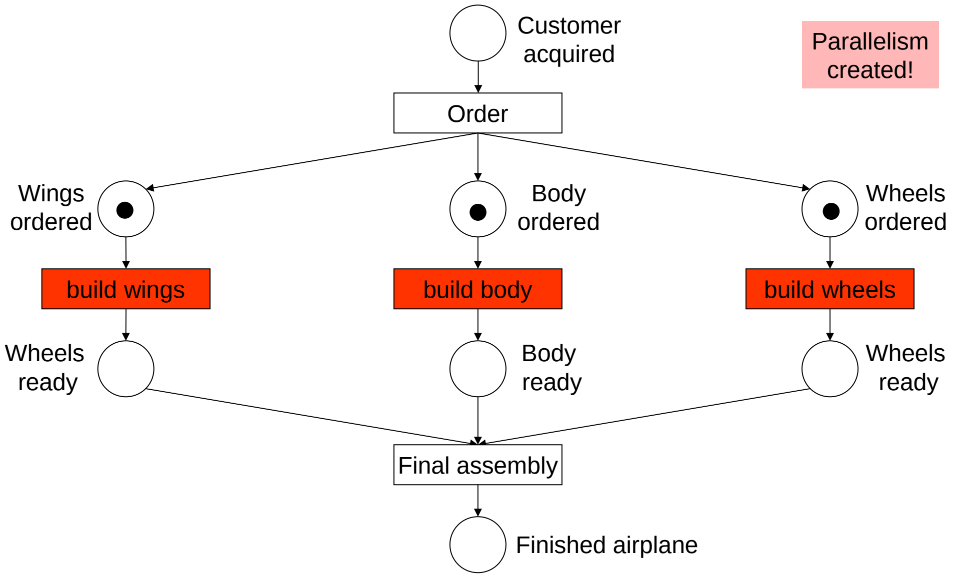


Petri-Net models possible concurrent execution paths.

Example: Production process (1)

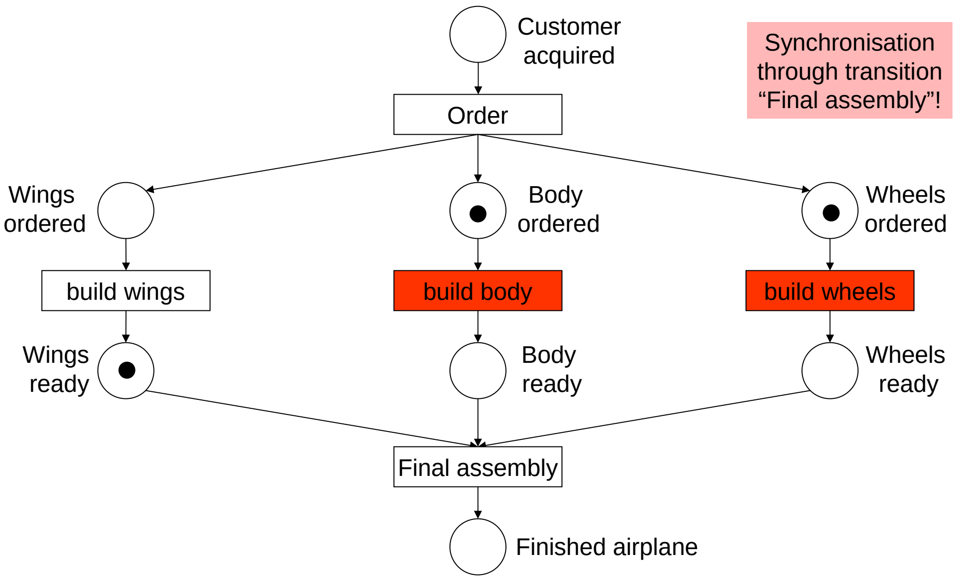


Example: Production process (2)



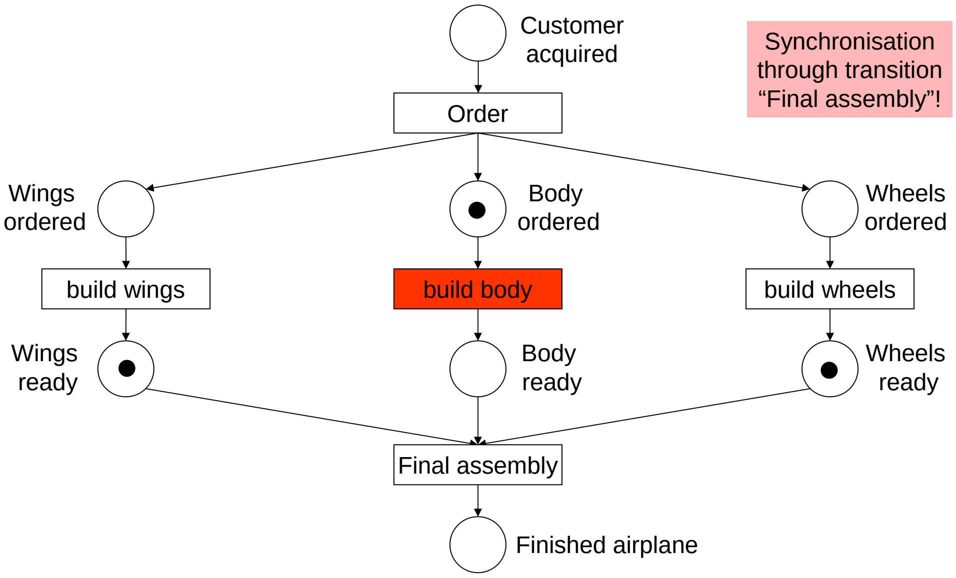
Software Architectures: Pipes & Filters Architectures 3.19

Example: Production process (3)

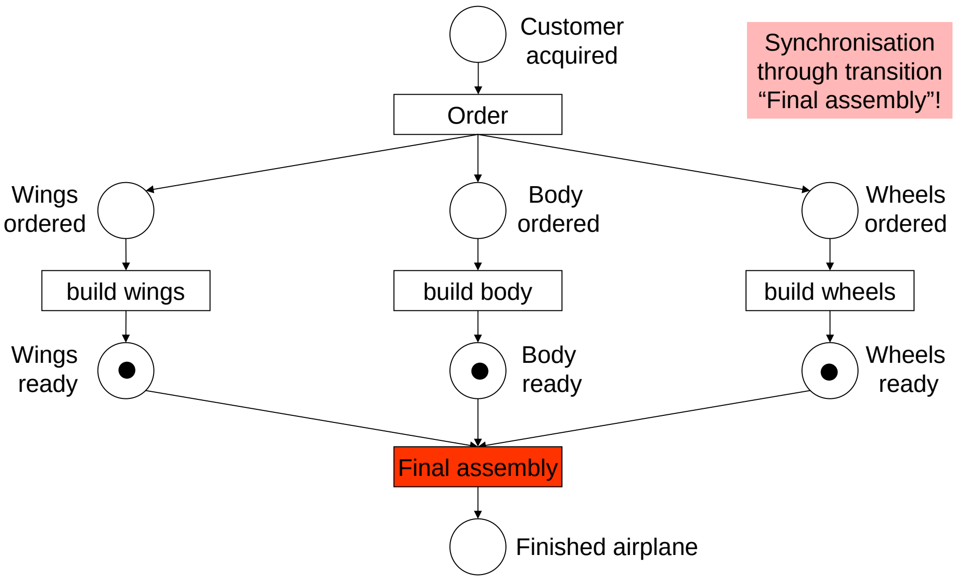


Software Architectures: Pipes & Filters Architectures 3.20

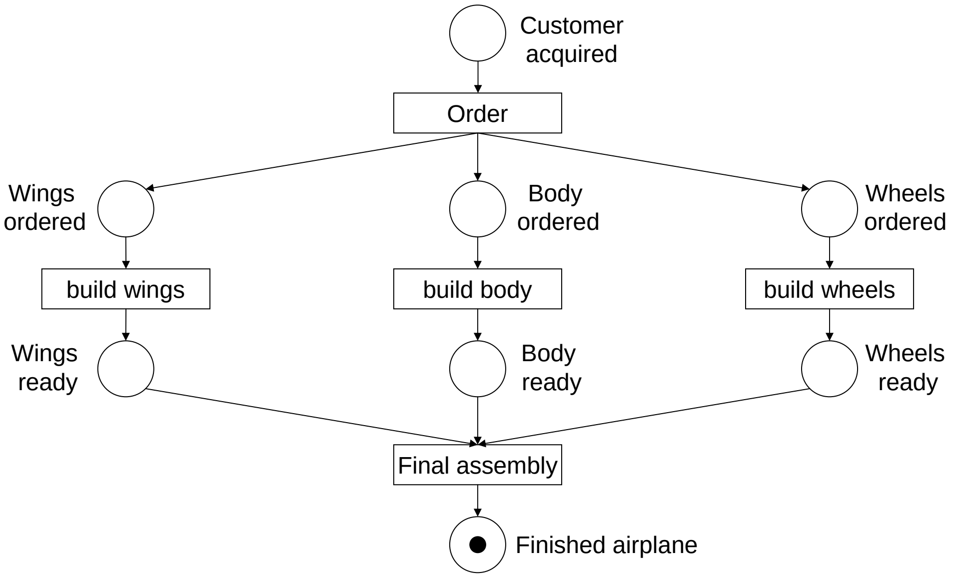
Example: Production process (4)



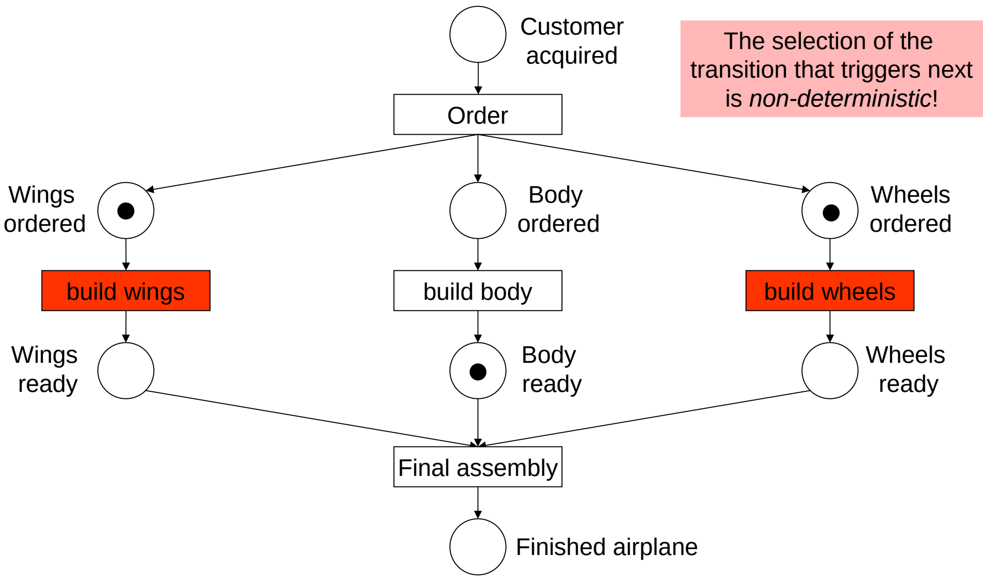
Example: Production process (5)



Example: Production process (6)

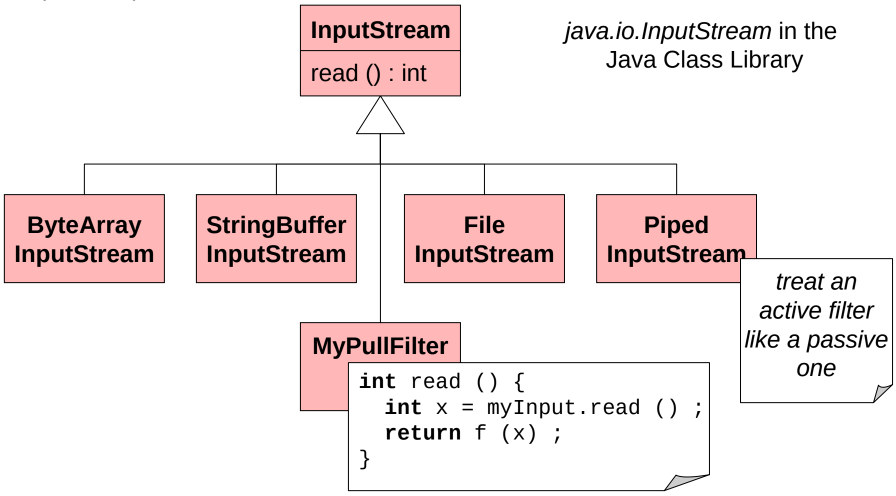


Alternative Step in Production process

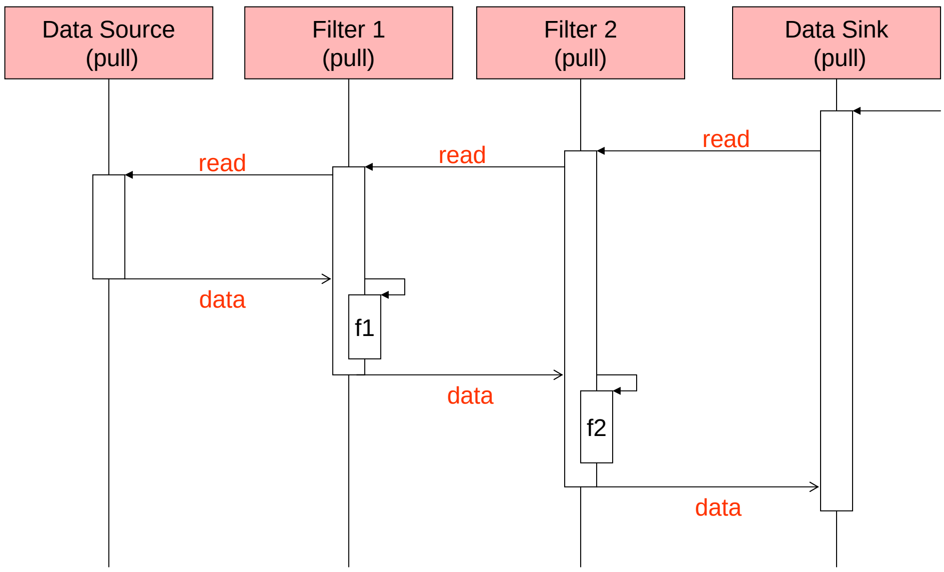


Passive Filters: Pull Strategy

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

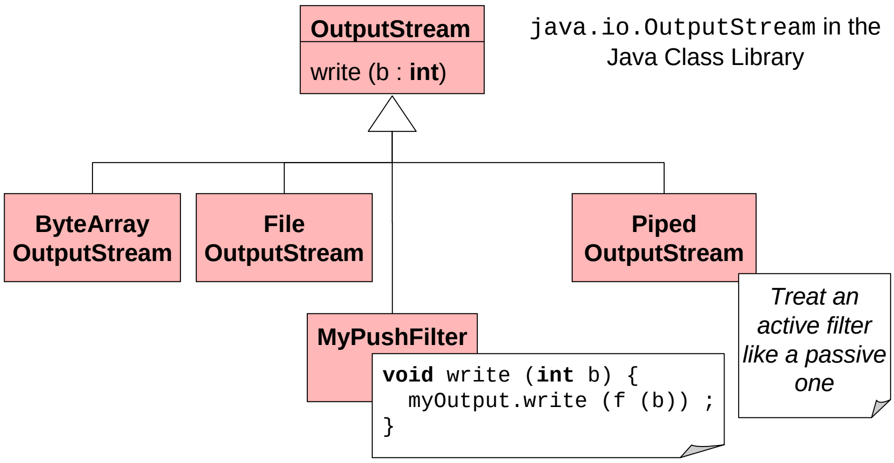


Scenario: Two Passive Pull Filters

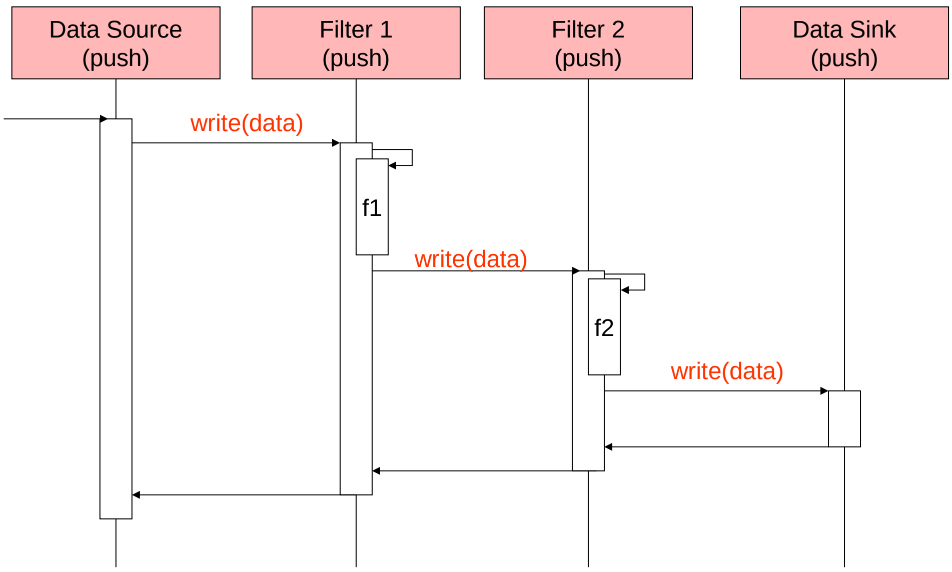


Passive Filters: Push Strategy

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



Scenario: Two Passive Push Filters



A First Comparison of Architectures

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

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 — "everything is a stream"
- vs. type safety — "stream of Persons, stream of Texts"

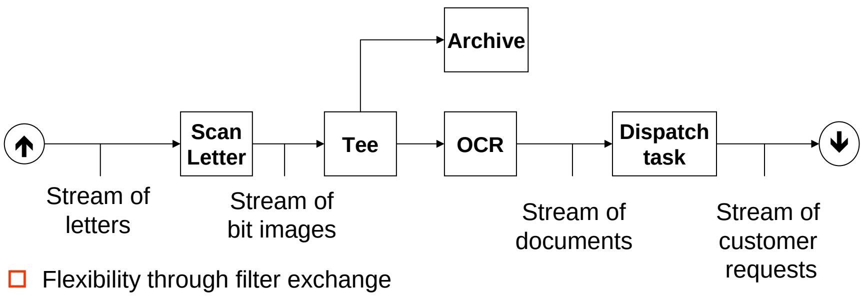
Popular Stream Data Formats

- 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

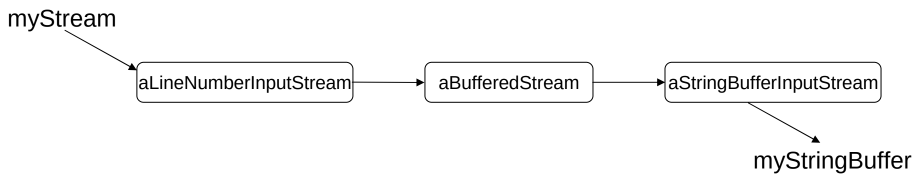
Intent:

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

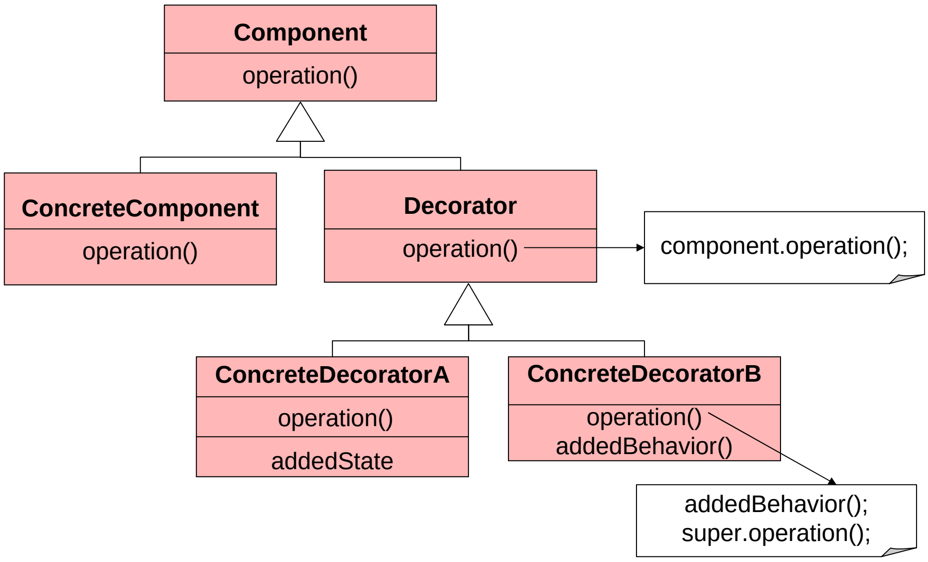
Motivation:

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



Remember: The Decorator Pattern



Decorator 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:

```
myStringBuffer = new StringBuffer ("This is a sample string to be read") ;
FilterInputStream myStream = new LineNumberInputStream (
                               new BufferedInputStream (
                                 new StringBufferInputStream (
                                   myStringBuffer))) ;

myStream.read () ;
myStream.line () ;
```