Chapter 2 Pipes and Filters

Pipes and Filters is a simple, yet powerful pattern to implement component-based data transformation problems such as a compiler or Computer Graphics Pipeline (the latter is used in the accompanying exercise). It allows to express a sequence of processing steps on a data stream, using components called *Filters*, connected through channels called *Pipes*. The Pipes and Filters architectural style is therefore used to divide a larger processing task into a sequence of smaller, independent processing steps (Filters) that are connected by channels (Pipes). See Figure 2.1 for a conceptual diagram of Pipes and Filters.

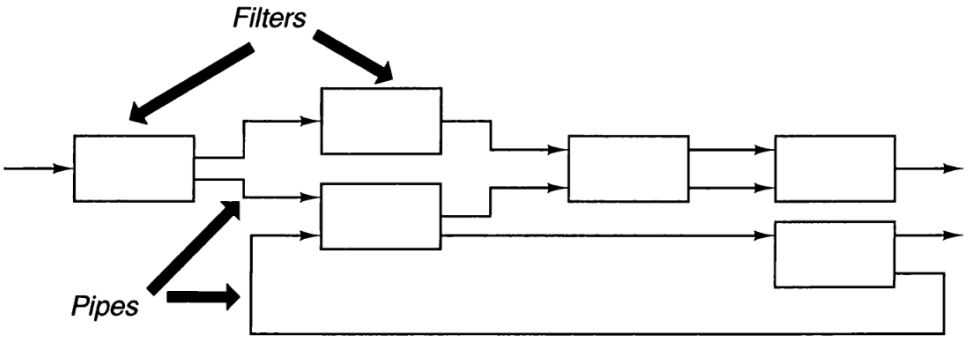

Figure 2.1: Pipes and Filters (Figure taken from [21])

Each filter exposes a very simple interface: it receives data on the inbound pipe, processes the data, and publishes the results to the outbound pipe. The pipe connects one filter to the next, sending output data from one filter to the next. The connection between filter and pipe is sometimes called port. In the basic form, each filter component has one input port and one output port.

See Figure 2.2 for an example of Pipes and Filters in a more realistic example of processing an incoming order. The Pipes and Filters architecture results in three filters, connected by two pipes: the first filter decodes an encrypted order, the second authenticates the order and the third catches duplicated orders (for example if the order was sent twice). One additional pipe is needed to send data to the decryption component and one to send the clear-text order data from the de-duper to the order management system. This makes for a total of four pipes.

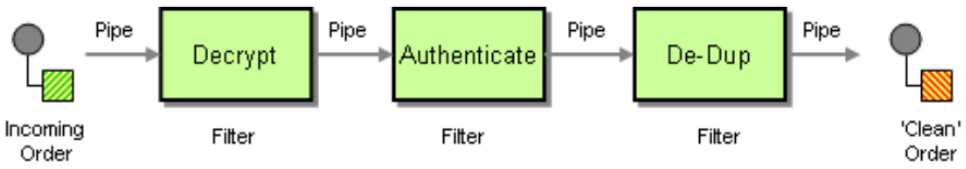

Figure 2.2: A pipes and filters example for processing an incoming order (Figure taken from [20])

The central concept in Pipes and Filters is processing of a data stream. A data stream is understood as a sequence of uniform data entities (bytes, characters of a character set, digitized audio signal,…), which thus are not modelled in a particular structure, but instead are simply given by a sequence. Sender and receiver agree on the semantics of the sequence (image, table, audio signal, …). As a consequence, it is not clear, for example, when the stream has reached its end - i.e. there isn't any structural information on the end of a data stream. See Java Streams for a more technical discussion of the problem.

2.1 Components

In the following sections we discuss the components of the Pipes and Filters pattern in more detail.

2.1.1 Filters

Filter components are the processing units of the pipeline. A filter enriches, refines or transforms its input data. It enriches data by computing and adding information, refines data by concentrating or extracting information, and transforms data by delivering the data in some other representation. A concrete filter implementation may combine all three basic principles. The activity of a filter can be triggered by several events:

- The subsequent pipeline element pulls output data from the filter.
- The previous pipeline element pushes new input data to the filter.
- The filter is active in a loop, pulling its input from and pushing its output down the pipeline.

The first two cases denote *passive* filters, whereas the last case is an *active* filter. An active filter starts processing on its own as a separate program, process or thread. A passive filter component is activated by being called either as a function (*pull*) or as a procedure (*push*). See Control Flow for a more technical discussion of active and passive filters.

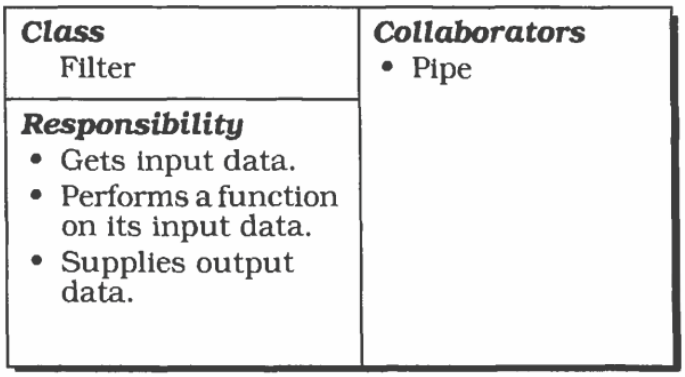

Figure 2.3: Filter responsibilities and collaborators (Figure taken from [5]).

Among the important invariants of the style is the condition that filters must be independent entities and should not share state with other filters. Another important invariant is that filters do not know the identity of their upstream and downstream filters. Their specification might restrict what appears on their upstream and downstream filter. Their specification might restrict what appears on the input pipes or make guarantees about what appears on the output pipes, but they may not identify the components at the ends of those pipes. Furthermore, the correctness of the output of a Pipes and Filters network should not depend on the order in which the filters perform their incremental processing (given fair scheduling).

Summarising Filters:

- Filters must be independent with no shared state.
- Filters don't know and must not make assumptions about upstream or downstream filter identity.
- Correctness of output from network must not depend on execution order in which individual filters provided their incremental processing (fair scheduling).
- Emphasise the port concept as they have input ports (for incoming data) and output ports (for outgoing data).
- Ports are connection points of a filter
- The port data are typed, and may follow a protocol: start and end of a (sub)stream.
- Read streams of data on input producing streams of data on output.
- Ideally: local incremental transformation to input stream, that means, output usually begins before input is consumed.
- The computation is done incrementally and locally. Incrementally: the portion of data that is available at the input ports is transformed. Locally: no outside influence except through ports.
- Enrich input data: e.g. add product details to product ID.
- Refine input data: e.g. filter out uninteresting or redundant data.
- Compose input data: e.g. streams of words to sentences.
- Transform input data: e.g. transform units of values.
- Analyze data : e.g. counting, statistics, identification, classification.
- Active filters: drive the data flow on the pipes (a pump): pulls autonomously its input from upstream and pushes it downstream.
- Passive filters: is driven by the data flow on the pipes: gets pushed or pulled.

2.1.2 Pipes

Pipes denote the connections between filters, between the data source and the first filter, and between the last filter and the data sink. If two active components are joined, the pipe synchronises them with a FIFO buffer. If activity is controlled by one of the adjacent filters, the pipe can be implemented by a direct call from the active to the passive component - direct calls make filter recombination harder, however.

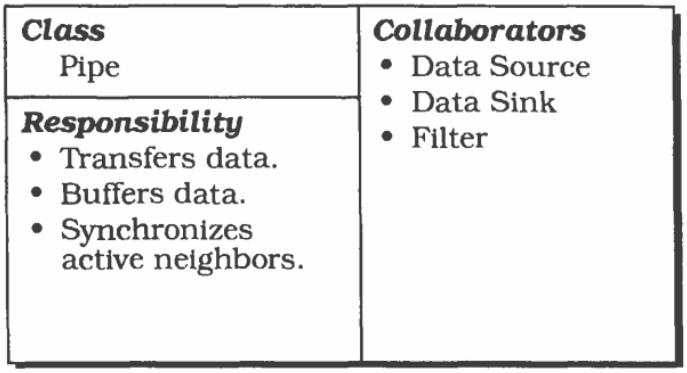

Figure 2.4: Pipe responsibilities and collaborators (Figure taken from [5]).

Summarising Pipes:

- Conduits for streams, e.g. first-in-first-out buffer.
- Transmit outputs of one filter to the input of another.
- May synchronize data flow between two filters.
- A pipe usually is a first class object (can be represented and manipulated in its own right).
- A pipe transfers data from one filter to the next.
- A pipe may implement a (bounded or unbounded) buffer.
- Restrictions on their ends may be possible (for example in *typed pipes*), but: no data transformation!
- A pipe may sit in between
    - Two filter objects/functions in a single process/thread.
    - Two threads in a single process: e.g. a Java Pipe, stream may contain simultaneous references to shared objects.
    - Two processes on a single host: e.g. a Unix Named Pipe.
    - Two processes in a distributed system: e.g. a Socket.

2.1.3 Data Source

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 line of text, or a polygon model. The data source of a pipeline can either actively push the data values to the first processing stage, or passively provide data when the first filter pulls.

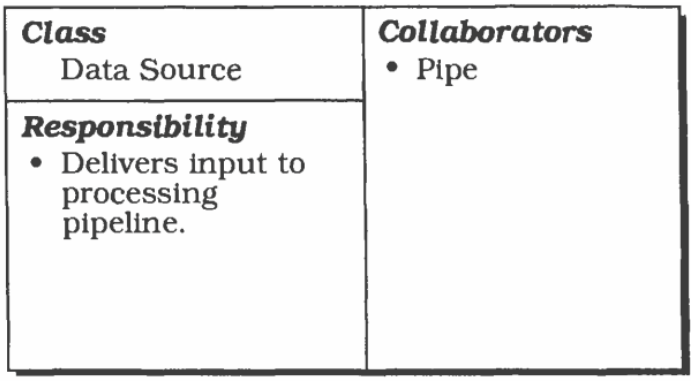

Figure 2.5: Data source responsibilities and collaborators (Figure taken from [5]).

2.1.4 Data Sink

The *data sink* collects the results from the end of the pipeline. Two variants of the data sink are possible. An active data sink pulls results of the preceding processing stage, while a passive one allows the preceding filter to push or write the results into it.

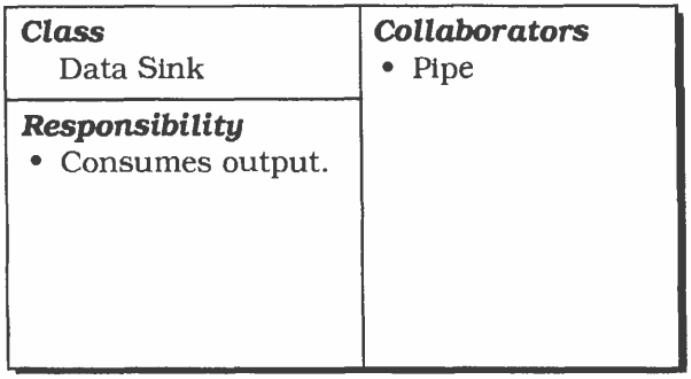

Figure 2.6: Data sink responsibilities and collaborators (Figure taken from [5]).

2.2 Control Flow

The following scenarios show different options for implementing the control flow between adjacent filters.

Scenario	Filters	Pipes	Data Source	Data Sink
I	passive	can be optional	passive	active
II	passive	can be optional	active	passive
III	some active	obsolete	passive	passive
IV	active	first class objects: synchronise	passive	passive

2.2.1 Pull Strategy with passive Filters

In this strategy the filters are passive objects that are driven by the subsequnt pipeline element *pulling output data* from the filter. The only active element is the data sink, which pulls data from the passive filters, eventually arriving at the data source. Filter activity is triggered by reading from the passive filters. See Figure 2.11 for such a pull pipeline with passive filters.

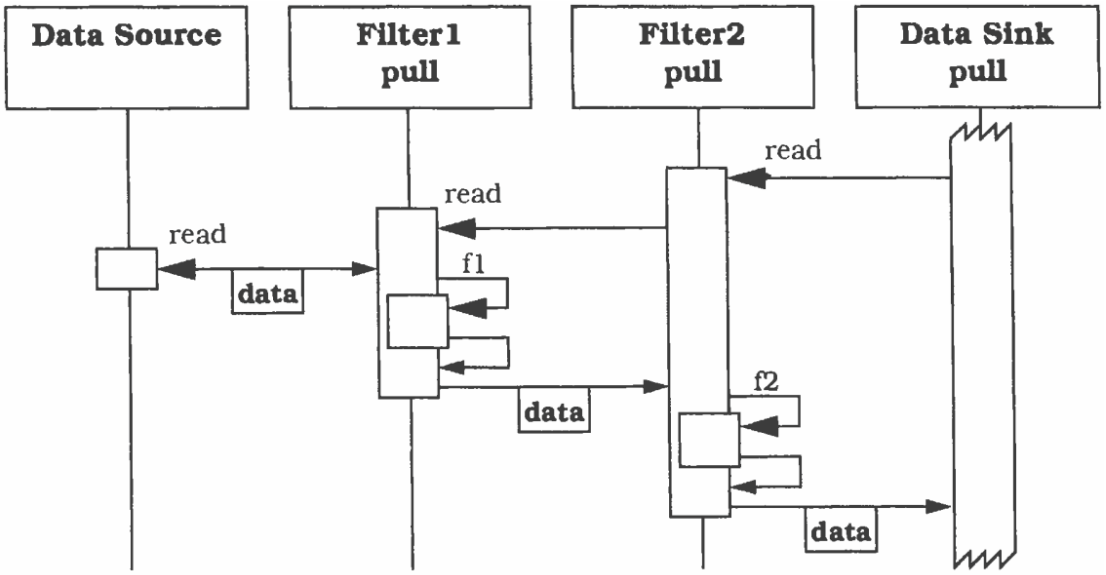

Figure 2.7: A pull pipeline with passive filters and an active data sink. Pipes are not first class objects (Figure taken from [5]).

In this implementation there are no explicit pipes, which means that they are not a first class object. Such a direct call (*read*) is easier to implement but not having pipes as first class objects is a serious disadvantage in the Pipes and Filters architecture and should be avoided. The reason for it is that it makes recombination harder as a filter may presume certain previous/subsequent functions whereas a pipe presumes only data.

An example for such a very simple implementation of a passive pull strategy without pipes as first class objects is the `InputStream` of Java, see Figure 2.8.

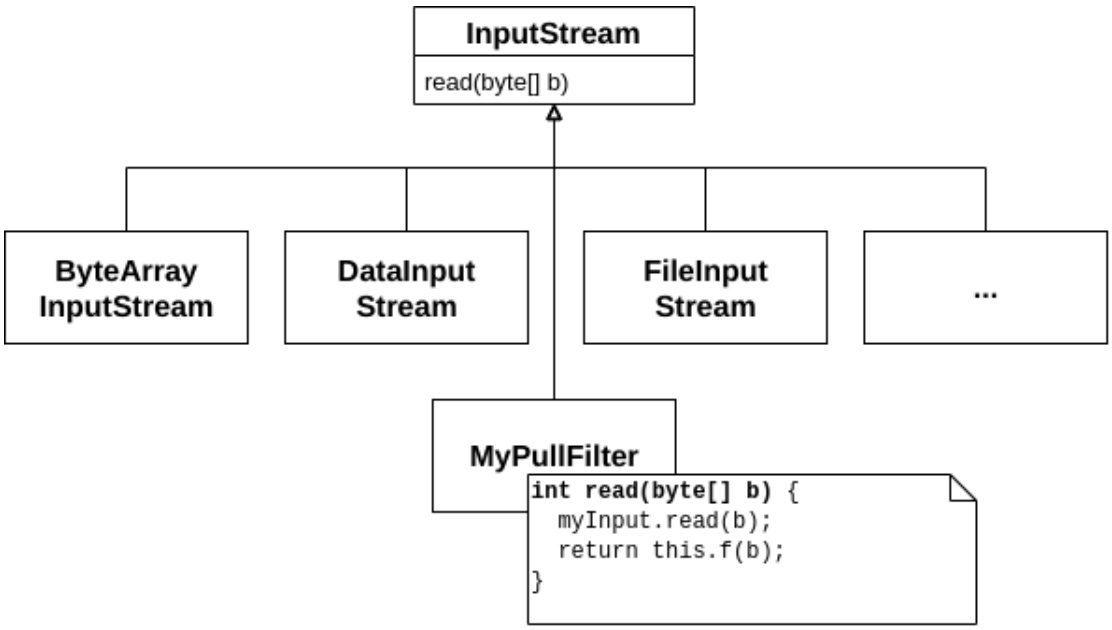

Figure 2.8: InputStream of Java as a simple implementation of a passive pull strategy without pipes as first class objects.

As noted above, not having pipes as first class objects in the Pipes and Filters architecture should be generally avoided as it leads to the loss of many of the architectures benefits. Fortunately, introducting pipes as first class objects into a pull strategy with passive filters is straightforward, see Figure 2.9.

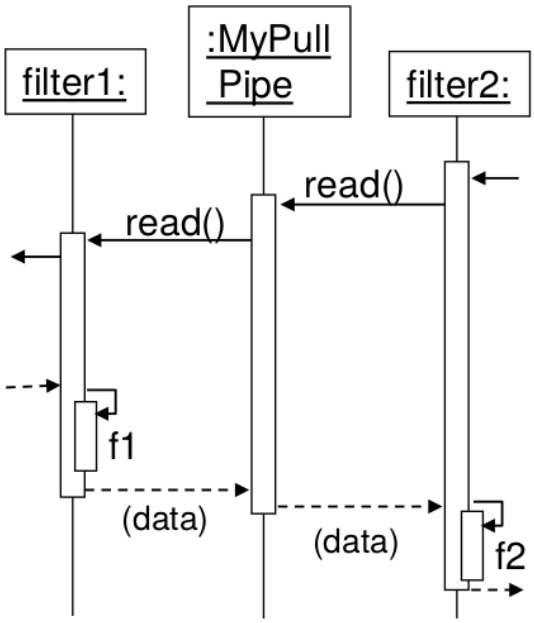

Figure 2.9: Pipes as first class objects in a passive pull pipeline

In between each filter sits then a pipe, see Figure 2.10.

Figure 2.10: Flow in a passive pull pipeline with pipes as first class objects. The green data sink is the only active element.

2.2.2 Push Strategy with passive Filters

The filter is again a passive object that is driven by the previous pipeline element that *pushes input data* into the filter. The only active element is the data source, which pushes data to the passive filters, eventually arriving at the data sink Filter activity is triggered by writing to the passive filters. See Figure 2.11 for such a pull pipeline with passive filters.

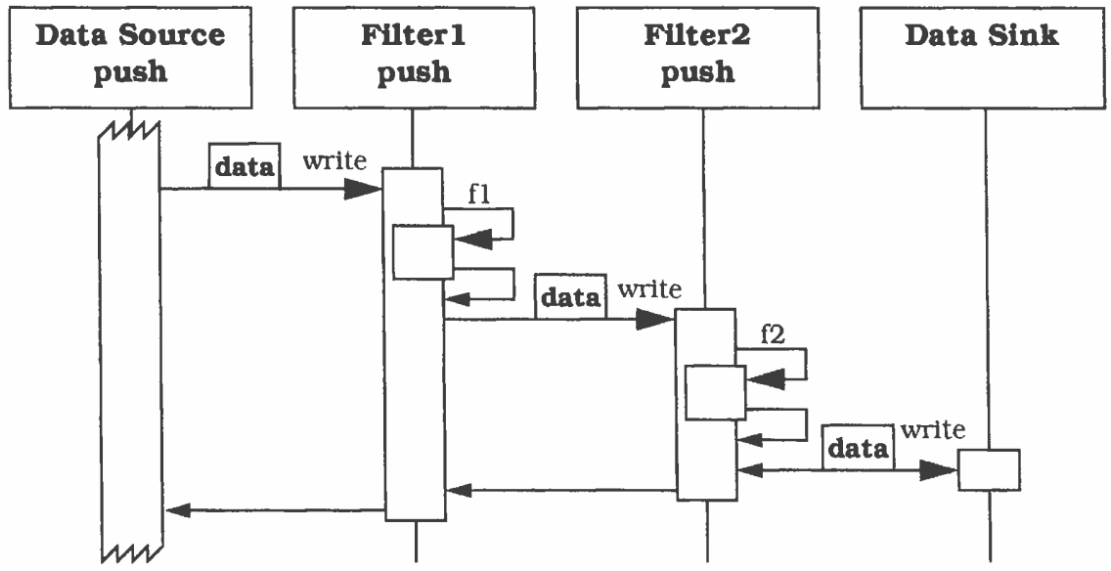

Figure 2.11: A push pipeline with passive filters and an active data source. Pipes are not first class objects (Figure taken from [5]).

Again, this implementation is not using pipes as first class objects, which makes implementation easier (a simple read call) however it makes filter recombination harder. Avoid this!

An example for such a very simple implementation of a passive push strategy without pipes as first class objects is the `OutputStream` of Java, see Figure 2.12.

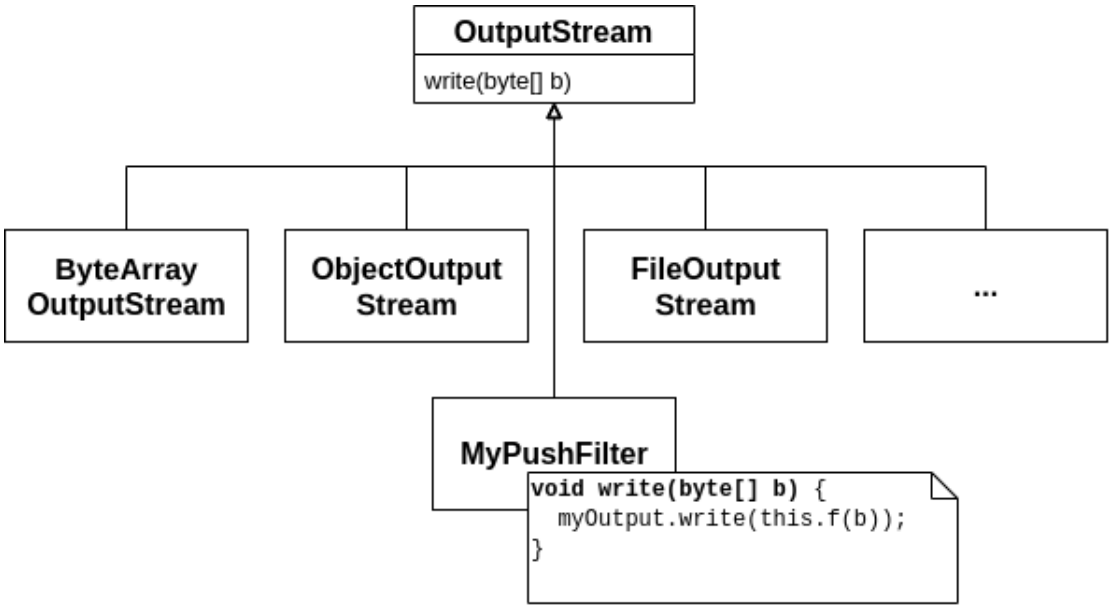

Figure 2.12: OutputStream of Java as a simple implementation of a passive push strategy without pipes as first class objects.

As noted above, not having pipes as first class objects in the Pipes and Filters architecture should be generally avoided as it leads to the loss of many of the architectures benefits. Fortunately, introducting pipes as first class objects into a push strategy with passive filters is straightforward, see Figure 2.13.

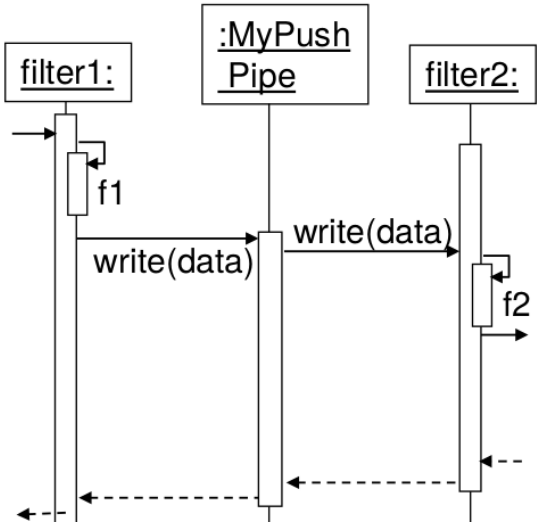

Figure 2.13: Pipes as first class objects in a passive push pipeline

In between each filter sits then a pipe, see Figure 2.10.

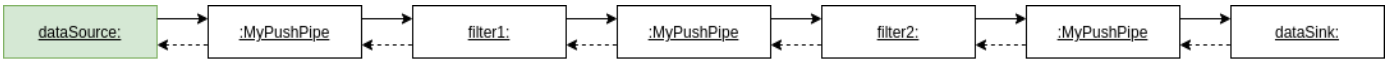

Figure 2.14: Flow in a passive push pipeline with pipes as first class objects. The green data source is the only active element.

2.2.3 Mixed Push-Pull Strategy

In a mixed push-pull strategy one of the filters becomes the active element and *both* the data source and data sink remain passive. See Figure 2.15 for such a mixed push-pull pipeline with a single active filter (Filter2).

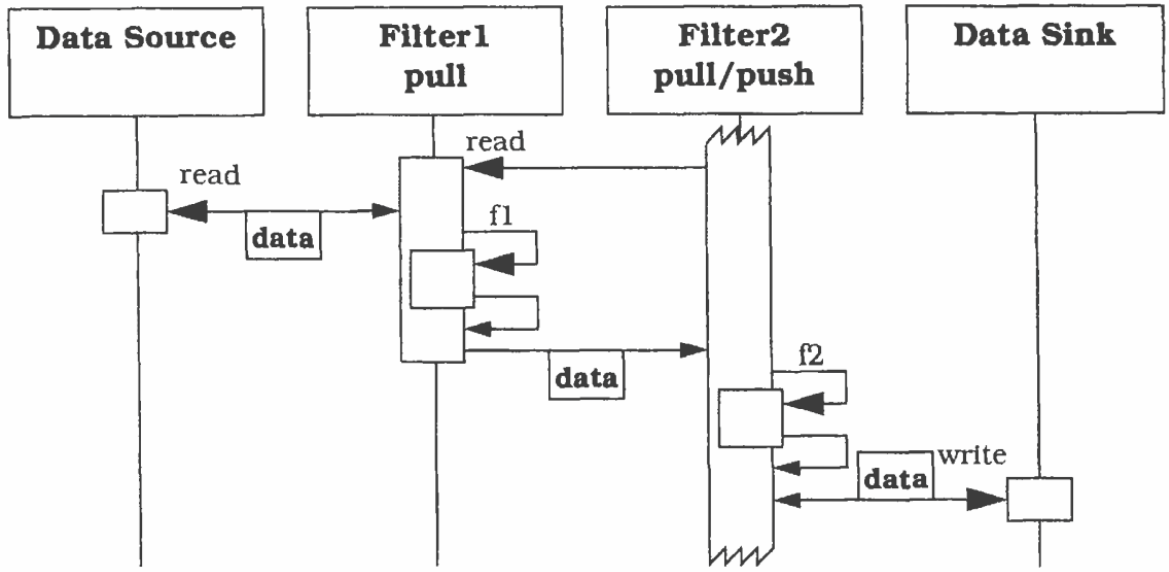

Figure 2.15: A mixed push-pull pipeline with one active and one passive filter, passive data sink and data source. Pipes not implemented as first class objects. (Figure taken from [5]).

Again, this implementation is not using pipes as first class objects, which makes implementation easier (a simple read call) however it makes filter recombination harder. Avoid this as not having pipes as first class objects in the Pipes and Filters architecture leads to the loss of many of the architectures benefits. Fortunately, introducing pipes as first class objects into a push strategy with passive filters is straightforward, see Figure 2.16.

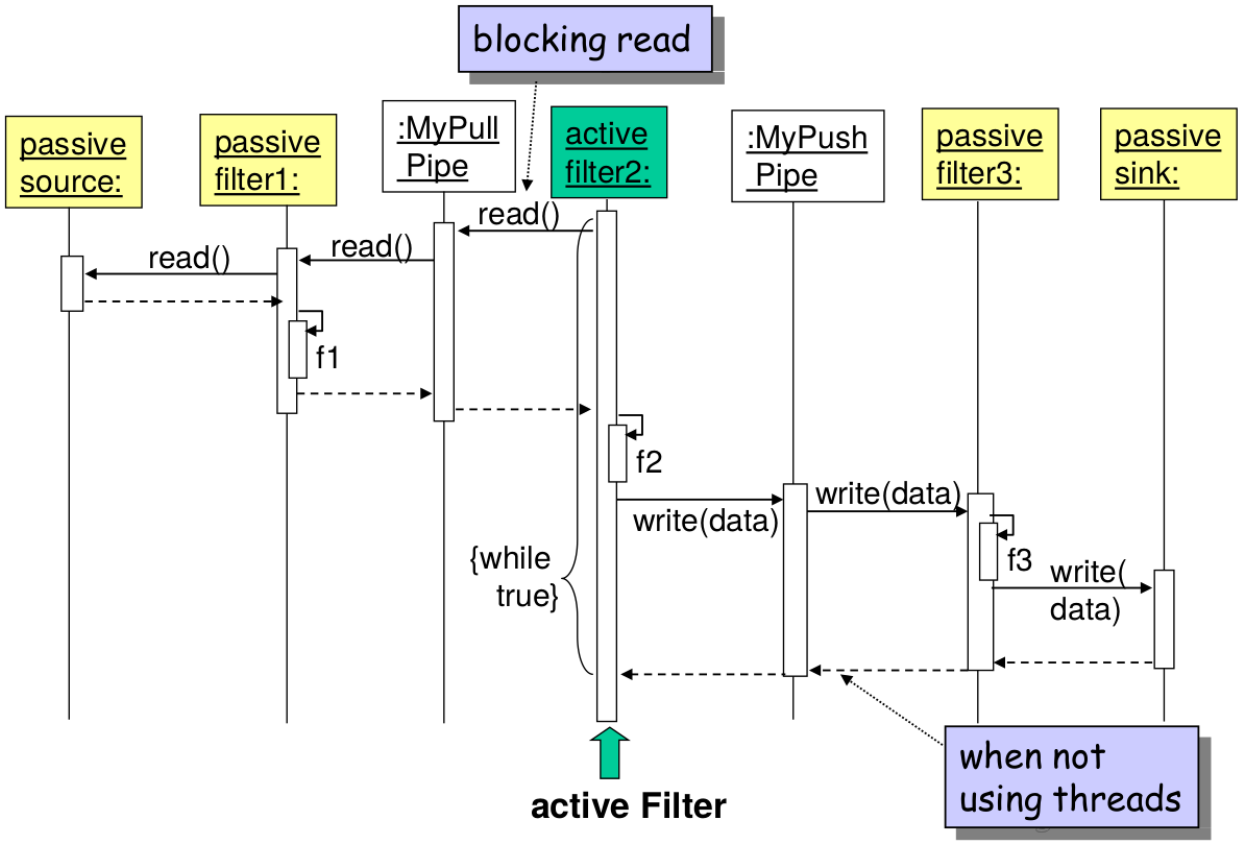

Figure 2.16: A mixed push-pull pipeline with pipes as first class objects.

2.2.4 Active Push-Pull Strategy

In the active push-pull strategy the data source and data sink remain passive, as in the mixed push-pull strategy, however filters now actively pull, compute and push data in a loop. Such a filter therefore runs in its own thread of control and are synchronised by an *active*, buffering pipe between them, see Figure 2.17.

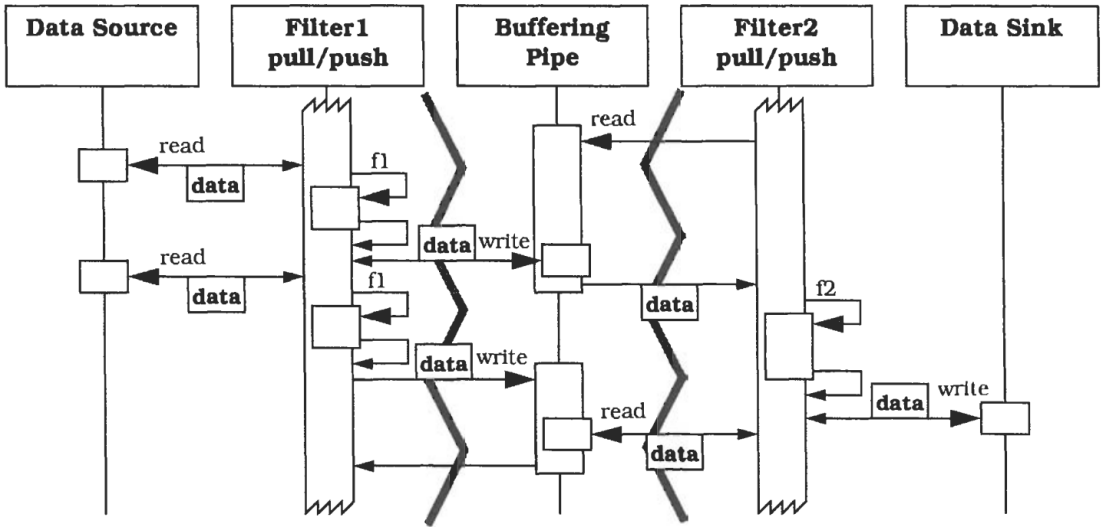

Figure 2.17: Active push-pull pipeline with an active buffering pipe between two filters running in separate threads (Figure taken from [5]).

Figure 2.18 depicts this strategy in more detail.

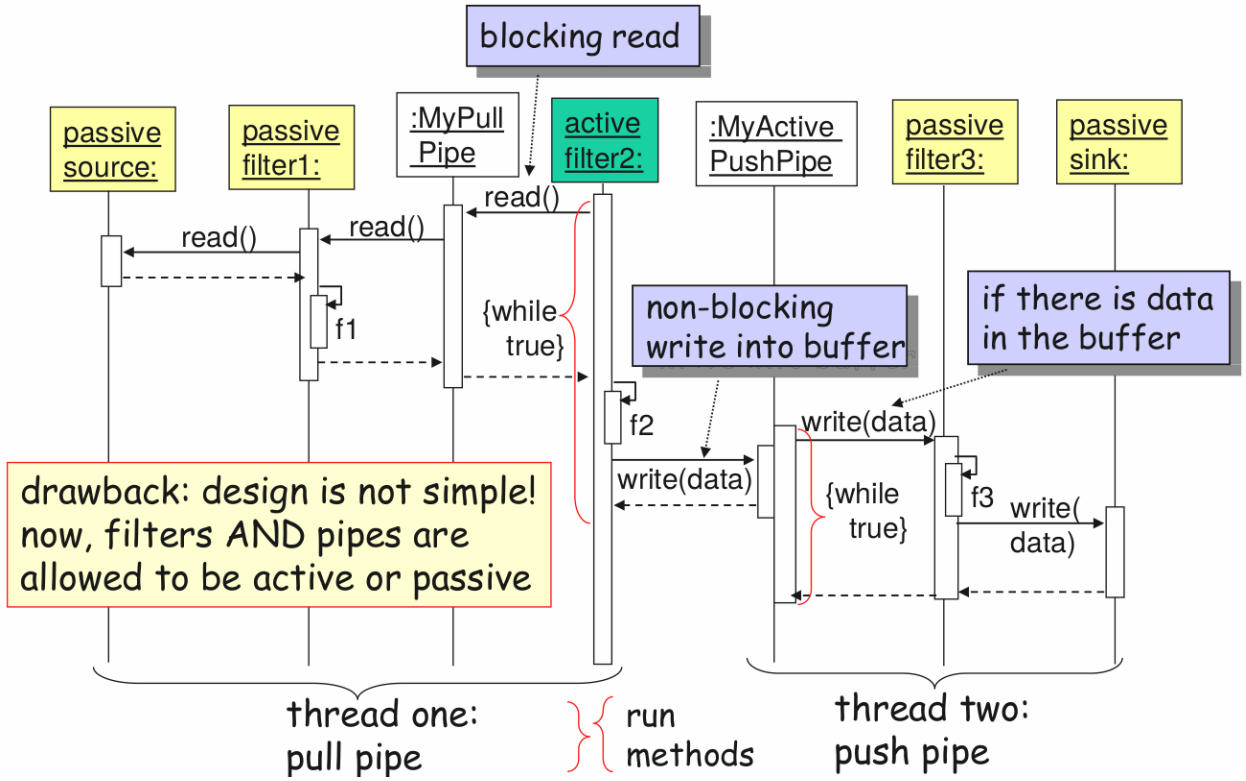

Figure 2.18: Version 1 of a mixed push-pull pipeline with threads using active pipes.

A drawback of this design is that it is not simple: now filters *and* pipes are allowed to be active or passive. The alternative is to use two active filters and a synchronising pipe between them. This is a case of the producer-consumer problem, see Figure 2.19.

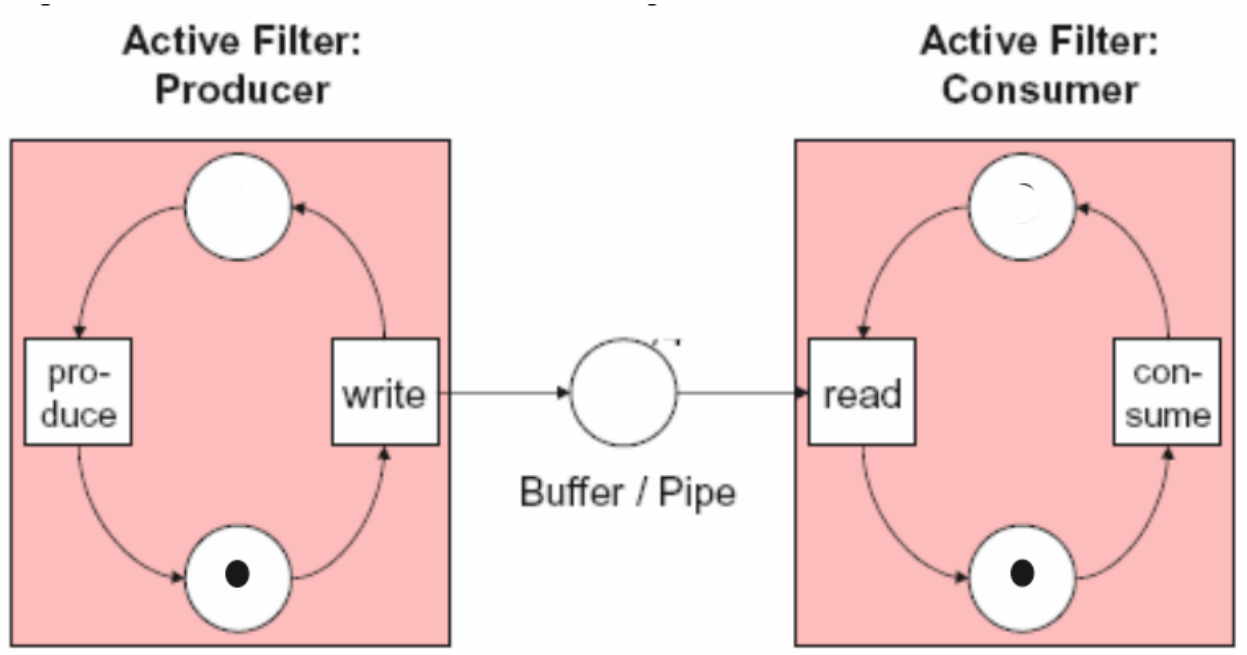

Figure 2.19: Producer-Consumer problem with two active filters and a synchronising passive pipe in between.

Changing the implementation of Figure 2.18 to use only passive pipes is shown in Figure 2.20.

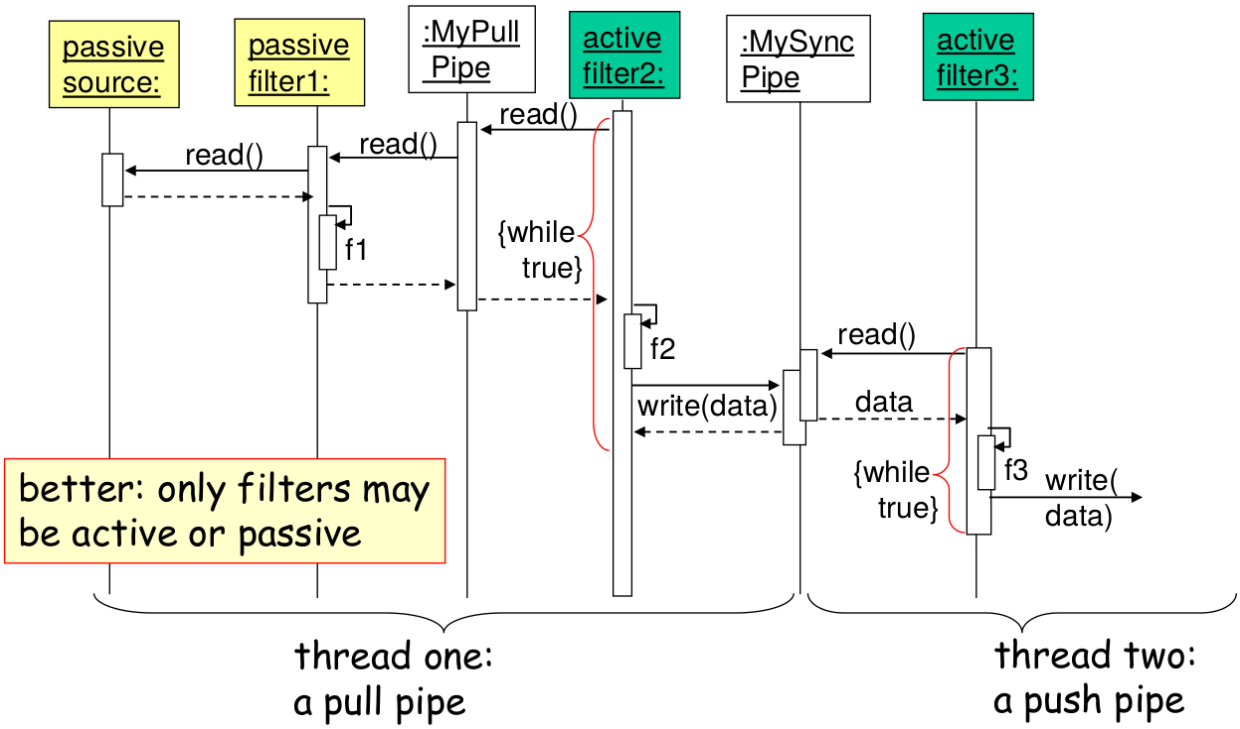

Figure 2.20: Version 2 of a mixed push-pull pipeline with threads using only passive pipes.

There also exists a scenario for this strategy, where the sink can be active, that is we have an active filter running in its own thread and an active sink running in its own thread. This is useful for GUI programming, where we need to decouple long running operations from the GUI thread, to avoid blocking and subsequent freezing of the GUI until the operation has finished. See Figure 2.21 for an example.

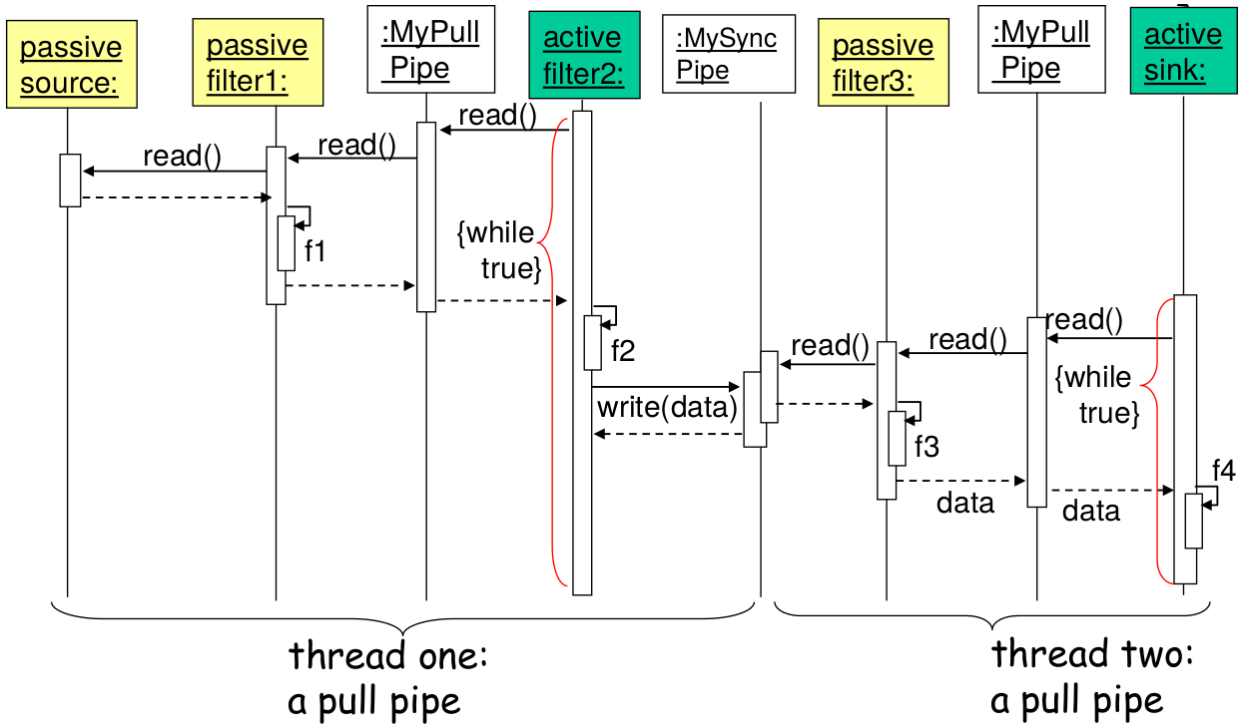

Figure 2.21: Active push-pull strategy with threads and active sink which for example can be a GUI.

2.3 Pipeline Processing

Connecting components with asynchronous pipes allows each filter in the chain to operate in its own thread or its own process. When a filter has completed processing data it can forward it to the output channel and immediately start processing another data.

It does not have to wait for the subsequent components to read and process the data. This allows multiple data to be processed concurrently as they pass through the individual stages. Such a configuration is known as a *processing pipeline* and can significantly increase system throughput when compared to strictly sequential processing. See Figure 2.22 for an example of pipeline processing with Pipes and Filters.

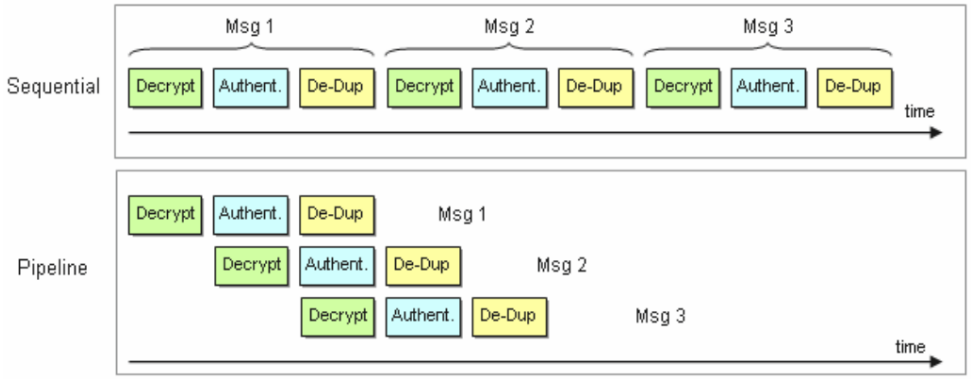

Figure 2.22: Pipeline processing with Pipes and Filters (Figure taken from [20])

2.4 Parallel Processing

The overall system throughput is limited by the slowest process in the chain. To improve throughput it is possible to deploy multiple parallel instances of that process to improve throughput. This allows to speed up the most time-intensive process and improve overall throughput. However, this configuration can cause data to be processed out of order. Parallelizing filters works best if each filter is stateless, that is it returns to the previous state after a message has been processed.

See Figure 2.23 for an example of parallel processing with Pipes and Filters, where it is assumed that decrypting data is much slower than authenticating it, therefore providing three instances of the decrypt filter.

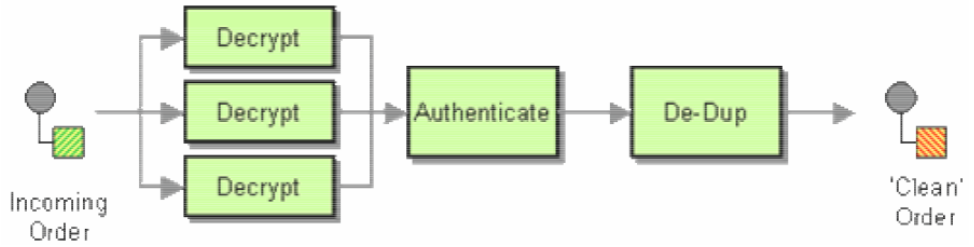

Figure 2.23: Parallel processing with Pipes and Filters (Figure taken from [20])

2.5 Implementing Pipes and Filters

1. Divide the system's task into a sequence of processing stages. Each stage must depend only on the output of its predecessor.

2. Define the data format to be passed along each pipe. Defining a uniform format results in the highest flexibility because it makes recombination of filters easy. If both flexibility and different data representations are needed, transforming filter components can be created to change the data between semantically equivalent representations. Also define how the end of input is marked.

3. Decide how to implement each pipe connection. This decicion directly determines whether the filters are implemented as active or passive components. Adjacent pipes further define whether a passive filter is triggered by push or pull of data. The simplest case of a pipe connection is a non-existing one: a direct call between adjacent filters to push or pull a data value as shown in Control Flow. However, avoid this as it makes it harder to combine such filters with others. Using a separate pipe mechanism that synchronizes adjacent active filters provides a more flexible solution. If all pipes use the

same mechanism, arbitrary recombination of filters becomes possible. A pipe supplies a first-in-first-out buffer to connect adjacent filters that produce and consume unequal amounts of data per computation.

4. Design and implement the filters. The design of a filter component is based both on the task it must perform and on the adjacent pipes. You can implement passive filters as a function, for pull activation, or as a procedure for push activation. Active filters can be implemented either as processes or as threads in the pipeline program.

5. Set up the processing pipeline by combining the pipes and filters implemented previously.

2.6 When to use it

1. The system task can be structured in a *sequence of working steps on conceptually the same object*. An example is image processing where in each step a filter manipulates an image as a whole before sending it to the next step, see Figure 2.24. In this type, repeated *one-shot* tasks operate on conceptually the same object. For such one-shot tasks the Pipes and Filters architecture offers a good modularisation as just one data type flows through the pipe also allowing for task variations. Also it follows a simple process structure by sequential coupling and therefore provides interrupt points for saving intermediate results and where to restart later. Repeated tasks also offer the basis for parallel computing.

Figure 2.24: Image processing with Pipes and Filters: working on the same object.

2. The result is an *incremental* data structure: for example, a page is a sequence of lines which are a sequence of words (and punctuation marks) which are a sequence of characters before the whole text of a page as a sequence of chars has been input, we can already recognize single words/signs, see Figure 2.25. In this type, Pipes and Filters offers a basis for parallel computing through pipelineing, since a complex data structure of the final product is decomposable (which might not be true for type 1, for example an image).

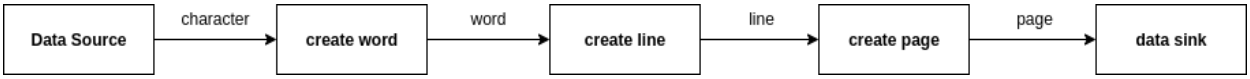

Figure 2.25: Document processing with Pipes and Filters: working on an incremental data structure.

Also, Pipes and Filters is not necessarily limited to programming, but can also be applied in terms of systems, where subsystems are filters which communicate by messages - the messaging and/or message brokers between them are the pipes. Likewise, different processing steps may be implemented using different programming languages or technologies that prevent them from running inside the same process or even on the same computer.

2.7 Benefits

The Pipes and Filters architectural style comes with a number of benefits:

- No intermediate files necessary (but possible).
- Flexibility by filter exchange.
- Flexibility by recombination. Allows to create new processing pipelines by rearranging filters or by adding new ones.
- Reuse of filter components.
- Rapid prototyping of pipelines from existing filters.
- Efficiency by parallel processing.
- Allow the implementer to understand the overall input/output behaviour of a system as a simple composition of the behaviours of the individual filters.
- Encourage and support re-use. Any two filters can be connected, provided they agree on the data that are being transmitted.
- Easy to maintain and enhanced as new filters can be added to existing systems and old filters can be replaced by improved ones.
- Allow for certain kinds of specialised analysis, such as throughput and deadlock analysis.
- Improves testability due to the ability to test each component in isolation through their clearly defined (often very narrow) interface.
- Overall behaviour is a simple composition of behavior of individual filters.
- Reuse - any two filters can be connected if they agree on the data type of the data transmitted.

- Ease of maintenance - filters can be added or replaced: maintenance focusing on components, not on control
- Prototyping: easy construction of complex applications
- Potential for parallelism - independent filters implemented as separate tasks, consuming and producing data incrementally. (this is particularly the case in Computer graphics pipelines)
- Easy to understand (once the components are understood)
- Filters stand alone and can be treated as black boxes. This isolation of functionality helps to ensure quality attributes such as information hiding, low coupling, modifiability, and reuse.
- Pipes and filters can be hierarchically composed. Higher order filters can be created from any combination of lower order pipes and filters.
- The construction of the pipe and filter sequence can often be delayed until run time (late binding). This permits a controller component to tailor a process based on the current state of the application.

2.8 Drawbacks

- Sharing state information is expensive, inflexible or even prohibitively dangerous. If the processing stages need to share a large amount of global data, applying the Pipes and Filters pattern is either inefficient or does not provide the full benefits of the pattern.
- Efficiency gain by parallel processing is often an illusion for the following reasons:
    - Cost for transferring data between filters may be relatively high compared to the cost of the computation carried out by a single filter. This is especially true for small filter components or pipelines using network connections.
    - Some filters consume all their input before producing any output, either because the task, such as sorting, requires it or because the filter is badly coded, for example by not using incremental processing when the application allows it.
    - Synchronisation overhead might result in even reduced performance.
    - The slowest filter determines the speed of the whole pipeline.
- Data transformation can cause a lot of overhead.
- Error handling is difficult. A common approach is to restart the pipeline in case of an error or exception.

- Not very well suited for handling interactive applications due to their transformational character.
- Depending on the implementation, they may force a lowest common denominator on data transmission, resulting in added work for each filter to parse and unparse its data, which can lead both to loss of performance and increased complexity writing the filters themselves.
- Can degenerate to ‘batch processing’: the filter processes all of its data before passing on (rather than incrementally).
- Sharing global data is expensive or limiting.
- Error handling is Achilles heel, for ex.: a pipeline has consumed three quarters of its input and produced half of its output, and some intermediate filter crashes! Generally restart pipeline.
- Implementation may force lowest common denominator on data transmission. For ex. in Unix pipes: everything is ASCII char.
- Transformational style does not support user interaction.
- Synchronization may be complicated (filters with more than one input).
- If a filter can not produce any output until it has received all of its input, the filter will require a buffer of unlimited size. If fixed size buffers are used, the system could deadlock. A sort filter has this problem.

2.9 Known Usages

Pipes and Filters architectures are by no means a new concept. The simple elegance of this architecture combined with the flexibility and high throughput makes it easy to understand the popularity of Pipes and Filters architectures. In this section we briefly discuss a few well known examples, which make use of the Pipes and Filters pattern / architecture.

2.9.1 Compilers

Pipes and filters are well known to have found a successful application in compilers, where the data structure is *incremental*, see Figure 2.26.

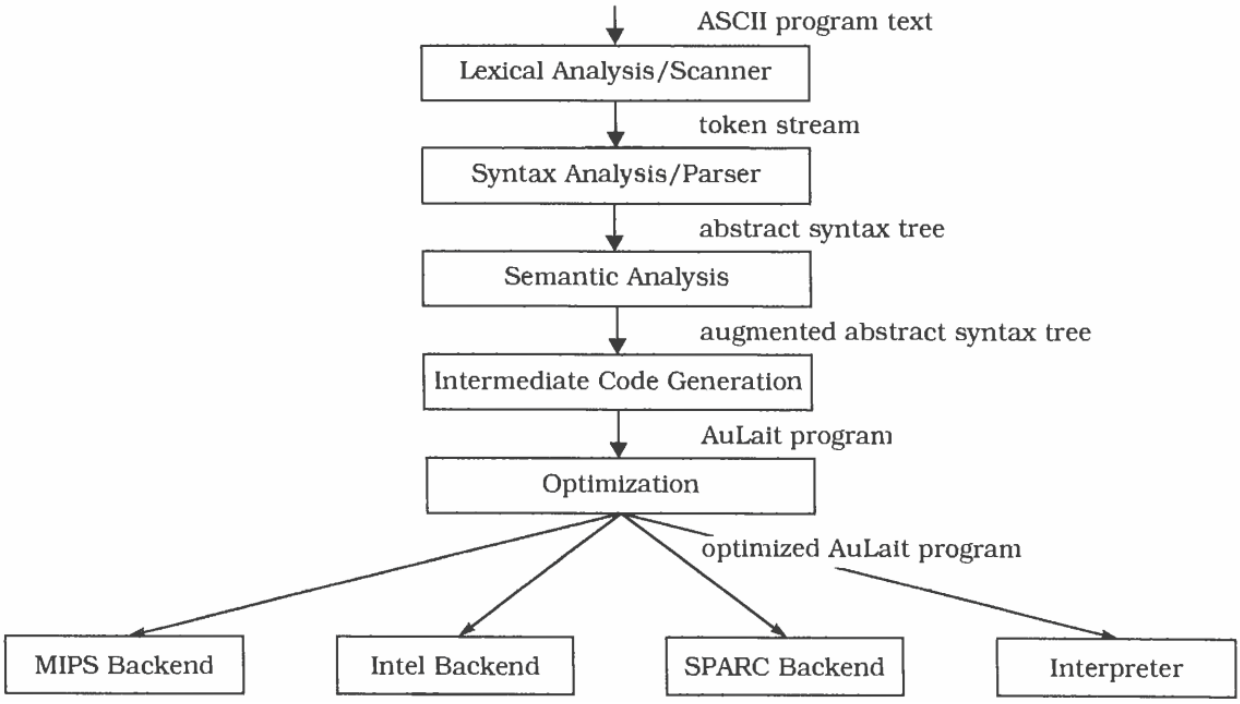

Figure 2.26: Pipes and filters in a compiler architecture (Figure taken from [5]).

2.9.2 Java IO Streams with Decorator Pattern

The decorator pattern is essentially a special pipe with a basic operation and subsequent steps which *decorate* the result, which is happening transparent to the client.

```
import java.io.*;
...
FileInputStream fin = new FileInputStream("myData.dat");
BufferedInputStream bin = new BufferedInputStream(fin);
DataInputStream din = new DataInputStream(bin);
double s = din.readDouble();
```

Object `din` is therefore a composed object, sending a *read* message to object *bin* (delegation) and adds then its decoration: parsing primitive Java types. This is obviously a pull pipeline, as in general Decorators are always a pull pipeline **if** they are a pipeline, see Figure 2.27.

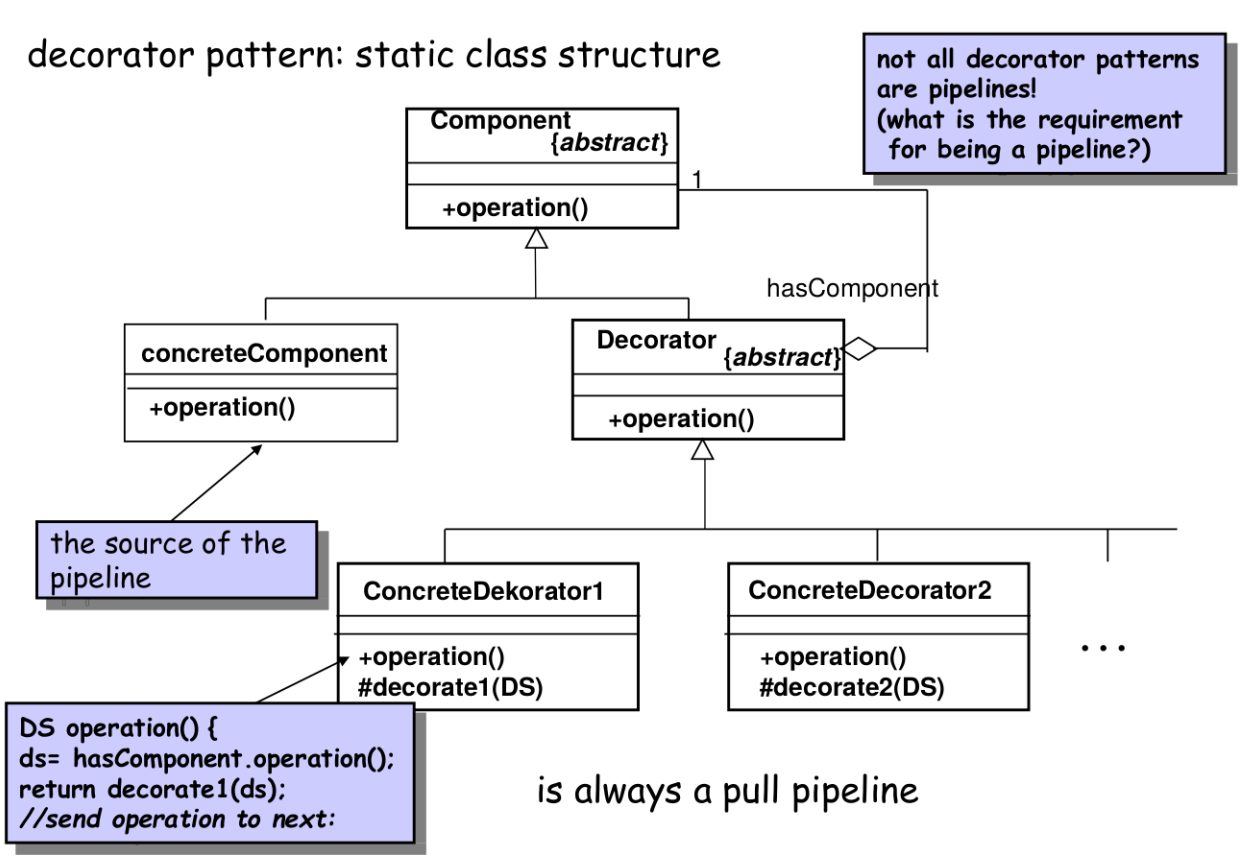

Figure 2.27: The decorator patterns as a pull pipeline.

What is the requirement for being a pipeline? We need access to the data which is transformed! If the decoration happens through implicit side effects in the decorated object it is difficult or even impossible to build a pipeline out of it.

2.9.3 Java Streams

Java Streams were introduced with Java 8 (March 2014) to facilitate functional programming in Java by adding functional interfaces, method references, lambdas *and* streams (not the IO Streams!). Although it is beyond the scope of this course to give an in-depth introduction to Java Streams, we will show the basic concepts, so that it is clear how they related to the Pipes and Filters pattern.

Java streams allow to transform a sequence of elements from a potentially infinite source through computations by acting element-wise in a pipeline fashing resulting in an aggregate or sequence, see Figure 2.28.

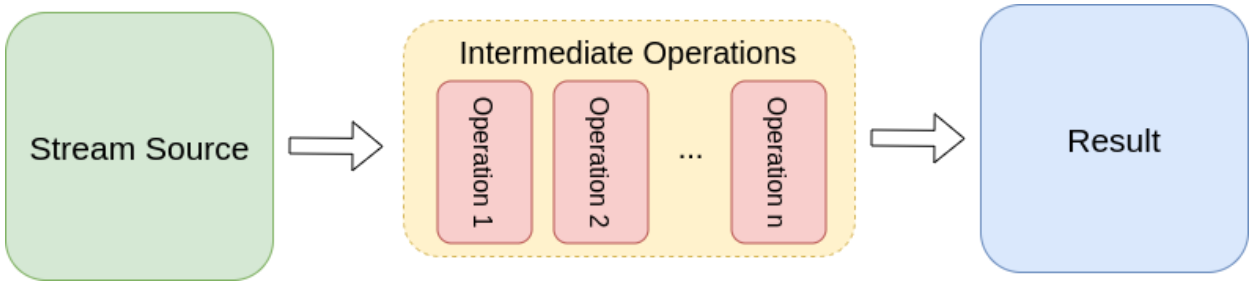

Figure 2.28: Java Streams conceptually.

The stream source can be arrays, collections, IO or generators. The intermediate operations can be one of the many operations provided such as `map, filter, takeWhile, limit, flatMap, distinct` . The result is either an aggregate coming from `sum, count, findFirst, reduce` or a sequence coming from `collect` .

In the following simple example we show how to apply Java Streams to a program which squares values in an int array and filters out the odd elements. This program could have looked like this in pre Java Stream times:

```
int[] vs = {1,2,3,4,5,6,7,8,9,10};
List<Integer> ret = new ArrayList<>();
for (int i : vs) {
  int sq = i * i;
  if (sq % 2 == 0) {
    continue;
  }
  ret.add(sq);
}

> [1, 9, 25, 49, 81]
```

Using Java Streams results in a much more concise and expressive pipeline:

```
int[] vs = {1,2,3,4,5,6,7,8,9,10};
IntPredicate even = x -> x % 2 == 0;
List<Integer> vsList = Arrays.stream(vs)
  .map(x -> x * x)
  .filter(even.negate())
  .mapToObj(Integer::valueOf)
  .collect(Collectors.toList());

> [1, 9, 25, 49, 81]
```

Java Streams also allows to work with potentially infinte streams, for example summing the first 100 prime numbers:

```
BigInteger FIRST_PRIME = BigInteger.valueOf(2);

BigInteger sum = Stream
  .iterate(FIRST_PRIME, BigInteger::nextProbablePrime)
  .limit(100)
  .reduce(BigInteger.valueOf(0), BigInteger::add);

> 24133
```

Java Streams also comes with *parallel* execution functionality, simply by adding `parallel()` :

```
BigInteger FIRST_PRIME = BigInteger.valueOf(2);
Stream
  .iterate(FIRST_PRIME, BigInteger::nextProbablePrime)
  .parallel()
  .limit(664_579) // primes between 1 - 10 million
  .reduce(BigInteger.valueOf(0), BigInteger::add);
```

However, when comparing this with benchmarking tools the parallel version is actually slightly slower than the sequential one. The reason is that `iterate` is inherently not suitable for parallelism because it is stateful. Rather use a generator, such as `LongStream` , which allows to generate streams over closed ranges:

```
LongStream
  .rangeClosed(2, 10_000_000)
  .parallel()
  .mapToObj(BigInteger::valueOf)
  .filter(i -> i.isProbablePrime(50))
  .reduce(BigInteger.valueOf(0), BigInteger::add);
```

This parallel version runs (depending on the hardware) up to twice as fast than its sequential counterpart. Also beware of effectful computations in parallel pipelines, which can lead to nondeterministic behaviour:

```
class Acc {
  long sum = 0;
  void add(long x) { this.sum += x; }
}

Acc acc = new Acc();
LongStream
  .rangeClosed(1, 100)
  .parallel()
  .forEach(acc::add);
System.out.println(acc.sum);

Run 0: sum = 5038
Run 1: sum = 4435
Run 2: sum = 4582
Run 3: sum = 4700
Run 4: sum = 4666
Run 5: sum = 4988
Run 6: sum = 4484
Run 7: sum = 5040
Run 8: sum = 4907
Run 9: sum = 4951
```

2.9.4 Unix Pipes

Unix popularised the Pipes and Filters paradigm, where this concept and pattern is used to transform data produced by a sequence processes (filters) by connecting these processes through pipes (using |). For example to list all files in a directory, having 'Test' in it and sorting them can be achieved in Unix through pipes and filters: `ls –l | grep "Test" | sort`

2.9.5 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. A MapReduce program is composed of a map procedure, which performs filtering and sorting (such as

sorting students by first name into queues, one queue for each name), and a reduce method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies)

2.9.6 Computer Graphics

The Pipes and Filters pattern is used in real-time computer graphics rendering in a pipelined fashion. The data structure is an incremental data stream, starting with vertices, to edges, to fragments, to pixels on screen, going through a number of transformations from 3D space to 2D on screen. See Computer Graphics Pipeline for more details, which is used in the [Lab Exercises].

Sources

This chapter is based on material from [21], [5] and [20].

References

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 2008. *Pattern-oriented software architecture: A system of patterns*. John Wiley & Sons.

[20] Gregor Hohpe and Bobby Woolf. 2004. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

[21] Shaw Mary and Garlan David. 1996. Software architecture: Perspectives on an emerging discipline. *Prentice-Hall* (1996).