Fundamental architectural styles

Fundamental architectural styles

- Outline and bibliography:
  - **Layers**
  - **Pipes and Filters**
  - **Repository**
  - **Event-driven (Implicit Invocation, Publisher-Subscriber)**

Fundamental Architectural Styles

- *"An architectural style defines a **family** of software systems in terms of their **structural organization**. An architectural style expresses **components** and the **relationships** between them, with the **constraints** of their application, and the associated composition and design **rules** for their construction."*

- **Structural organization** in software architecture : software architecture is formed by **multiple structures**, each structure corresponds to an architectural **viewpoint**

Fundamental Architectural Styles

- *The fundamental architectural styles are defined in a certain viewpoint:*
  - ***Module (static) viewpoint:*** *architectural styles which are defined by how the code is organized in parts*
    - *Layers*
  - ***Component-connector (dynamic) viewpoint****: architectural styles which are defined by how the components interact at runtime*
    - *PipesFilters, Repository, EventDriven*

Subsystems: Coupling and Cohesion

- Grouping elements into subsystems(components, modules) must **minimize coupling and maximize cohesion**
- Coupling: Dependencies *between* two subsystems
  - If coupling is high, can't change one without affecting the other
- Cohesion: Dependencies *within* a subsystem
  - High cohesion implies closely-related functionality

Layers

Layers

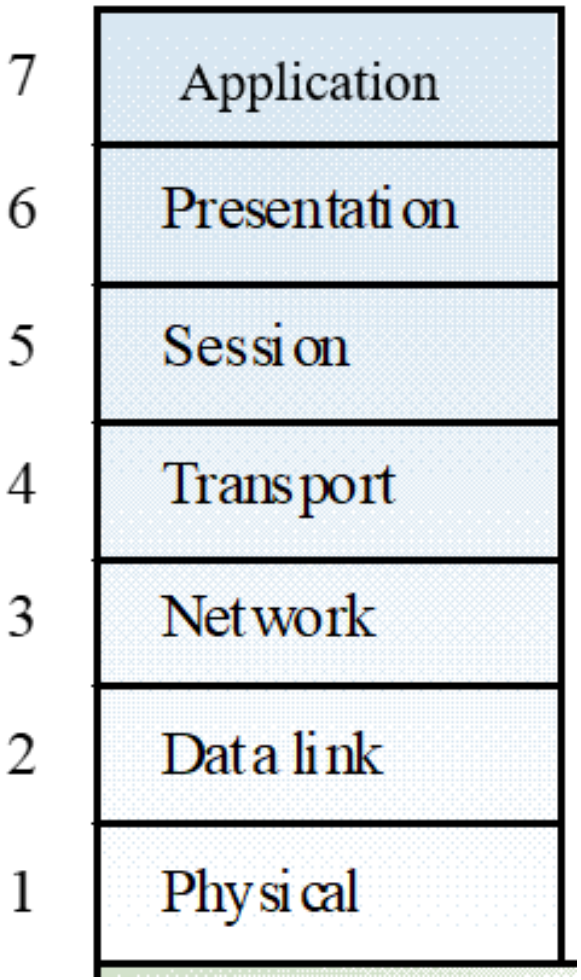
*The **Layers** architectural pattern helps to structure applications in **modules** (subsystems) that are called Layers.*

*Each Layer **provides services** to the Layer on top of it. Each Layer **is allowed to use** only services from Layers below.*

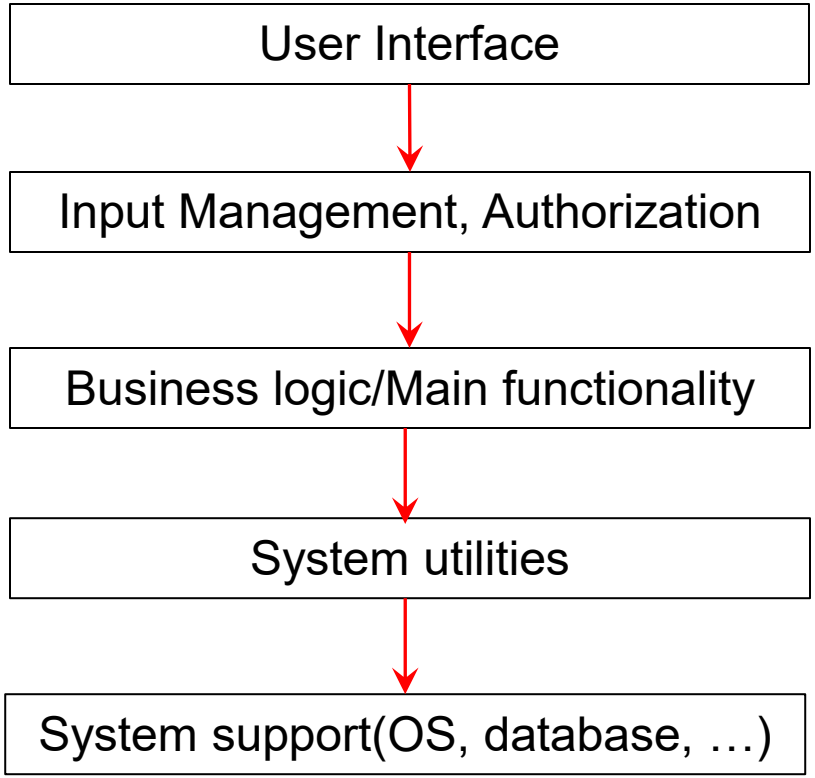
*Each Layer is a group of subtasks which represent a **meaningful abstraction.***

- Typical example:
  - Network protocol stacks
  - The protocols specify agreements at a variety of abstraction levels
  - Each layer deals with a specific aspect of communication and uses the services of the next lower layer.

Layers: Example: OSI network model

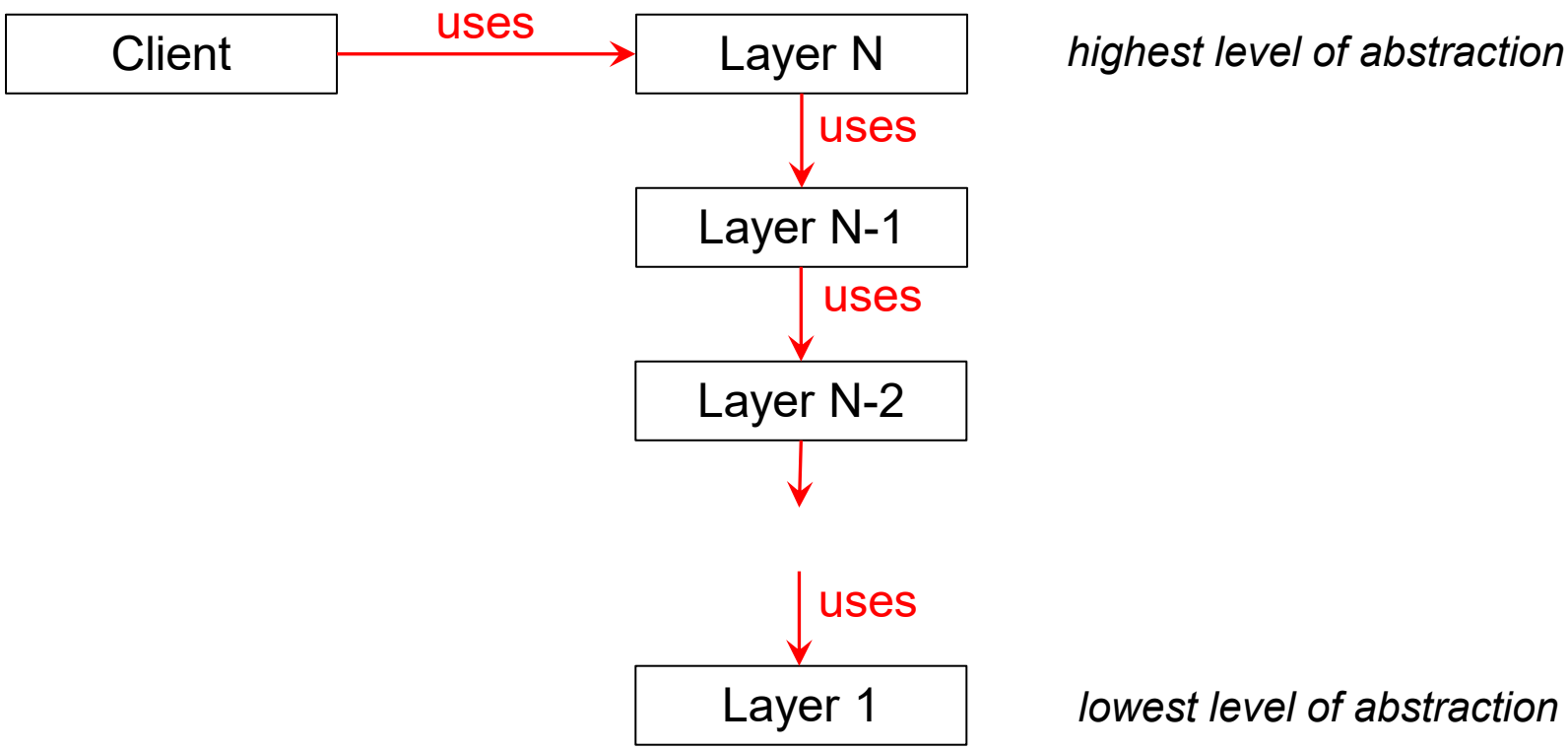


Layers: A generic example



Layers: Structural characteristics

- The system is organized as a o stack of subsystems (called layers)
- The Layers pattern is defined by **restrictions of the *uses relationships*: LayerJ *is allowed to use* LayerJ-1**

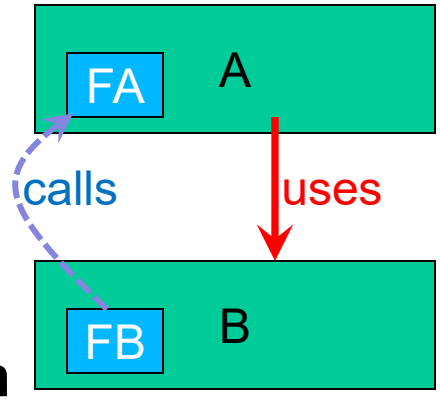


Layers: Dynamic behavior

- Two scenarios of dynamic behavior:
  - Top-down communication:
    - A client issues a request to Layer N. Layer N cannot calls the Layer N – 1, which calls Layer N-2, etc until reaching Layer 1.
  - Bottom-up communication:
    - A chain of actions starts at Layer 1 (for example when a device driver detects input). Layer1 notifies Layer 2 about it, and so on. In this way data moves up through the layers until it arrives at the highest layer.
    - This scenario must be realized by ***callbacks*** ! This mechanism allows having an ascending flow of data and control, but without having ascending Uses relationships

A Side-Note about The Callback Mechanism

- Callback can solve the situation described by following relationships:
  - *FB calls FA* but *A uses B* and *B does not use A*
- The Callback mechanism is realized by transmitting the name of the function to be called (FA in the example) by the bottom layer **in some form of data**.
- This mechanism allows an ascending flow of data and control, but without ascending Uses relationships. In this way it is not contradicting the restrictions imposed by the Layered style



- Thinking question: techniques for implementing callbacks?

Layers: Properties of the style

- Benefits:
  - Reusability: each layer can be individually reused, ***if*** it represents a coherent abstraction of some services and has a **well defined *interface***
  - Exchangeability: implementations of layers can be changed without affecting the rest, as long as the ***interface*** is preserved
  - Portability: low-level (hardware specific) details are isolated in bottom layer
- Liabilities:
  - Cascades of changing behavior: it is still possible that by changing one layer it will impact on others as well
  - Lower efficiency: less efficient that a monolithical implementation
  - Unnecessary work: if lower levels implement services that are not requested by upper levels
  - leaky abstractions: clear separation is difficult

Layers: Concluding remarks

- The Layers architectural style is defined by restricting the direction of allowed dependency relationships (uses relationships)
- *Restricting the allowed dependency relationships (especially for avoiding cyclic dependencies !) is a general design principle, not only for systems labeled as belonging to the layered style!*
- There are tools for the analysis of dependency relationships in code

Pipes and Filters

Pipes and Filters

The ***Pipes and Filters*** architectural style 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. Filters are **independent** of each other. **Recombining filters** allows you to build **families** of related systems.

- Most well-known example: operating system command pipelines: `ls | grep old | more`

P&F: When used

- Context: Processing data flows
- Problem: an application that can be naturally decomposed in processing stages;
  - Flexibility will be needed, to reorder (recombine) the components
  - It is possible to build a family of similar systems by reusing different subsets of components
  - The application is not an interactive application
  - Non-adjacent components do not share information
  - It is possible that the stages are executed in parallel

P&F: Example

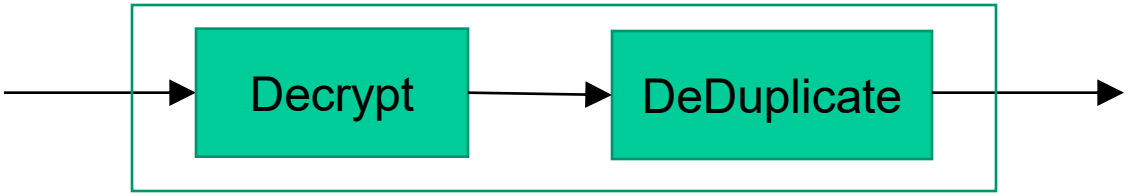
A server receives messages containing orders from clients.

The server frontend processor:

Incoming data: a stream of possibly duplicated, encrypted messages containing extra authentication data

Outgoing data: a stream of unique, simple plain-text order messages

Another server frontend processor for another service does not require authentication. It will be possible to reuse the independent Filters in order to build it



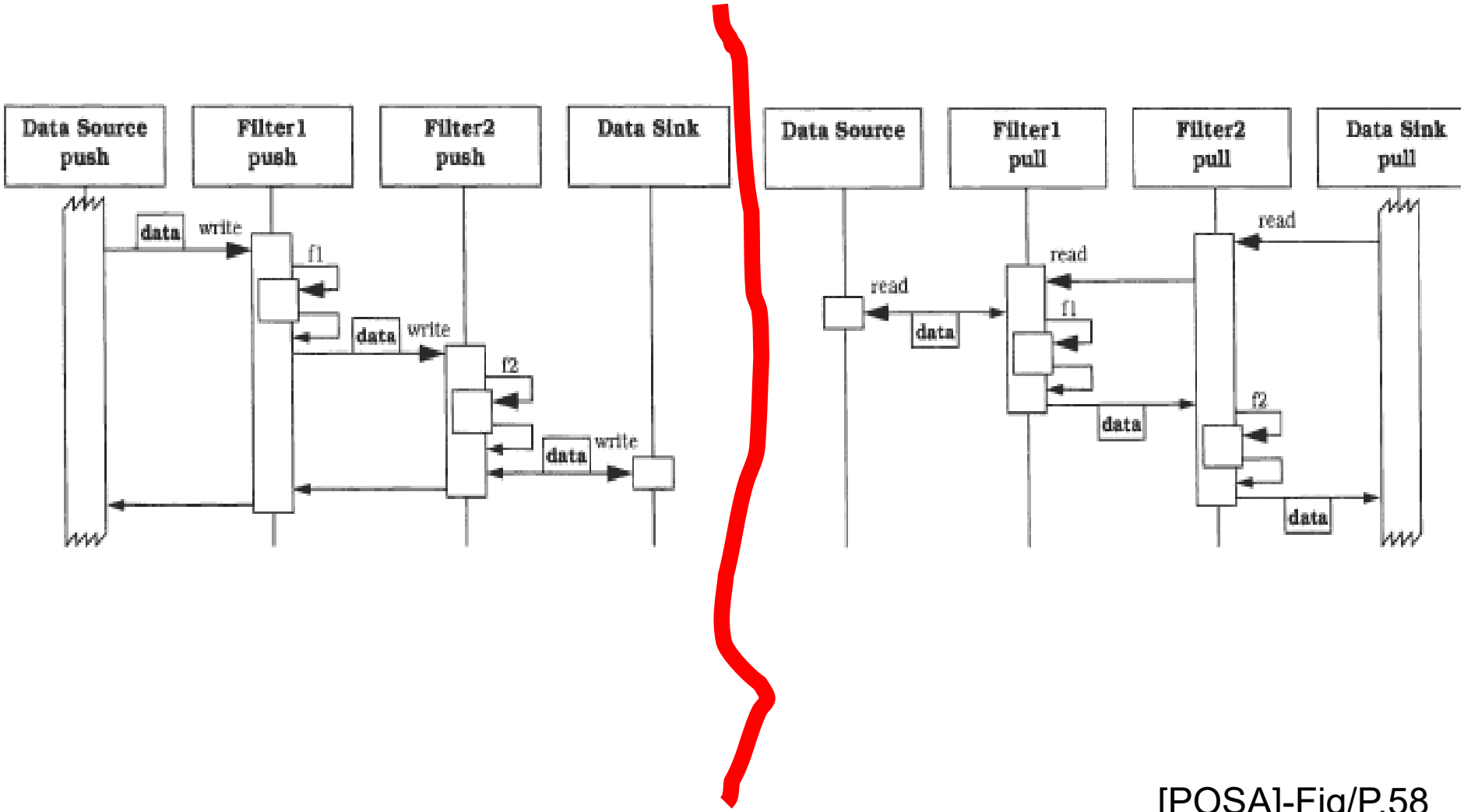
P&F - structure

- The system can be naturally decomposed in processing stages
- Each stage is implemented by a component called a Filter
- A Filter has following characteristics:
  - Reads data from input streams
  - Transforms data
  - Writes data on output streams
- The succession of Filters is given by the transformations of the data flow
- Pipes (connectors) :
  - Transport dataflows between consecutive filters
  - Can be implemented by different mechanisms: function calls, pipes, messages over a network, etc

P&F - behavior

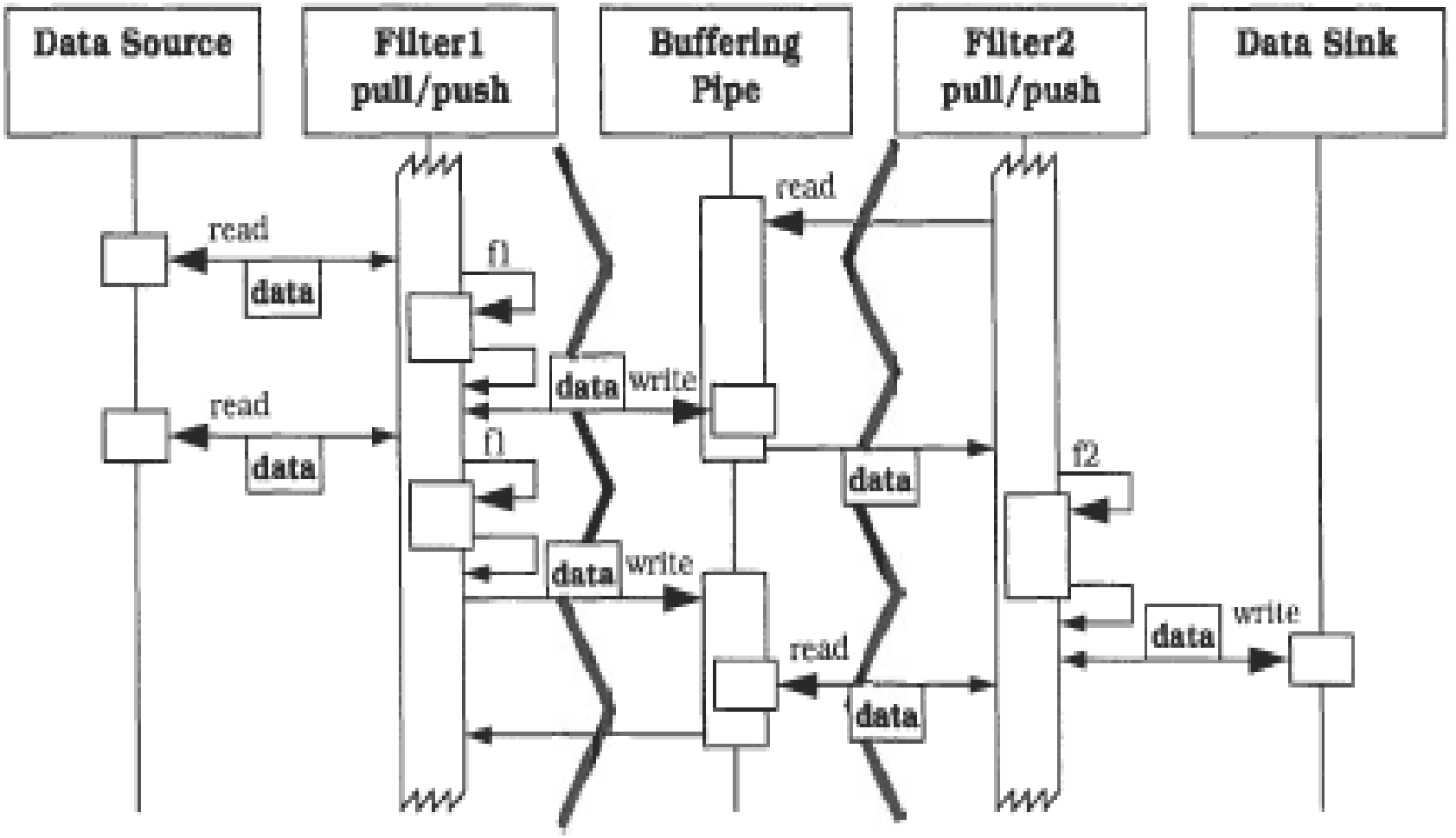
- Types of Filters:
  - Passive: they are explicitly activated by the preceding filter (push) or by the next filter (pull)
  - Active: the body of the filter consists in a continuous loop that reads-transforms-writes data (a separate thread or process)
    - With Active Filters, the *Pipes* must ensure *buffering* and *synchronization* !
    - In order to reduce the latency and to permit a concurrent processing, the Filters must function *incrementally* (read in one step *small* items of data, process them, write output and repeat)

Passive Filters: Push pipeline vs. Pull pipeline



[POSA]-Fig/P.58

Active Filter: Push/pull pipeline



[POSA]-Fig/P.60

P&F as Integration pattern

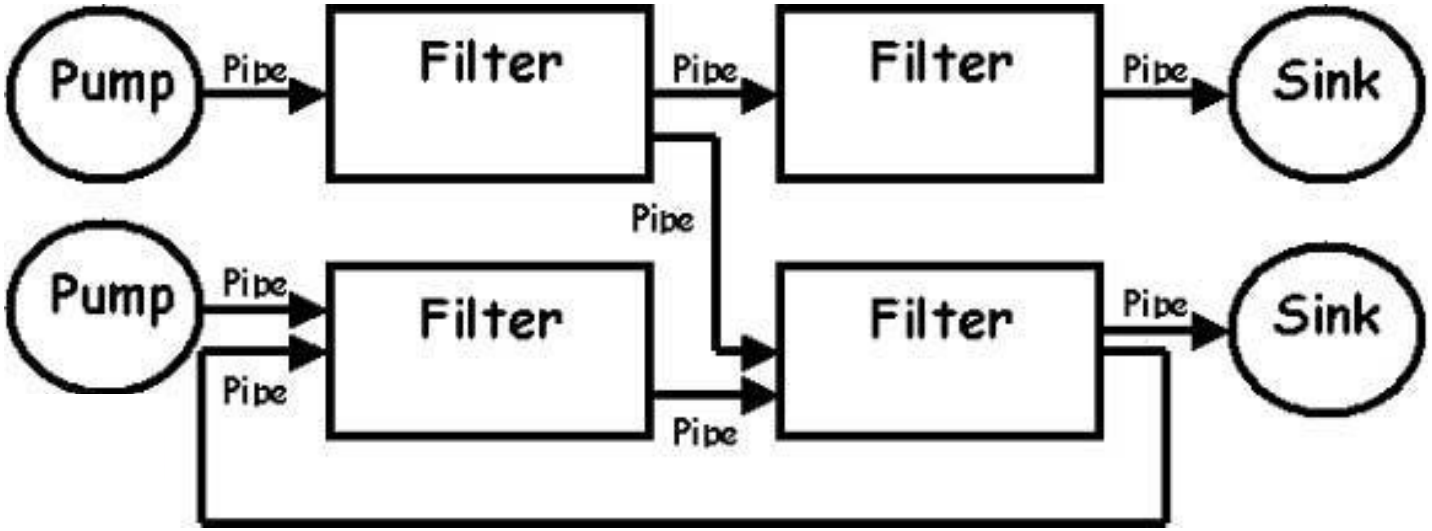
Integration = connecting different applications, systems, services, and data sources so they can work together as a unified system. Enables **data flow, coordination, and interoperability** across distributed systems.

Examples: Spring Integration Framework and Microsoft Azure

They support integration of services in a P&F style, where Filters = Services and Pipes = message queues

Pipes-and-filters: Variants

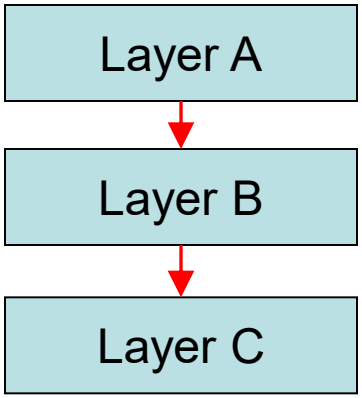
- Tee and join pipeline systems
  - The single-input single-output filter specification of the Pipes and Filters pattern can be varied to allow filters with more than one input and/or more than one output.
  - Processing can then be set up as a directed graph that can even contain feedback loops.



P&F: Properties of the style

- Benefits:
  - Flexibility by filter exchange or recombination
  - Reuse of filter components
  - Rapid prototyping of pipelines
  - Potential for efficient concurrent processing
- Liabilities:
  - Sharing state information is expensive or inflexible.
  - Data transformation overhead
  - Error handling difficulties

Discussion: Pipes-and-Filters vs. Layers



Represents direction of dataflow
(dynamic, runtime viewtype)

Represents *uses* relationships
(dependencies)
(static, module viewtype)

Is this P&F?

- Java Stream and ParallelStream filter:

```
List<Integer> result = numbers.stream()
                            .filter(n -> n % 2 == 0)
                            .filter(n -> n > 5)
                            .collect(Collectors.toList());
```

From the functional point of view: a processing pipeline where each filter() is a stage, data flows through stages and stages are composable

From the point of view of concurrency pattern: ParallelStream is **NOT** the typical stage parallelism. It implements Data parallelism: the data source is split into chunks, each chunk is assigned to a worker thread which fully processes all stages on that chunk.

Repository

Repository

Sub-systems must exchange large amounts of data. The shared data is held in a central database or repository and may be accessed by all sub-systems (components). The components do not interact directly and are independent of each other.

Example: an IDE (Integrated Development Editor)

Repository: Properties of the style

- When used:
  - When large volume of information is generated that has to be stored for a long time.
  - Also in data-driven systems where the inclusion of data in the repository triggers an action or tool.
- Advantages:
  - Components can be independent - they do not need to know of the existence of other components.
  - All data can be managed consistently (e.g., backups done at the same time) as it is all in one place
- Liabilities:
  - The repository is a *single point of failure*
  - May be *inefficient* in organizing all communication through the repository (a point of *bottleneck*)

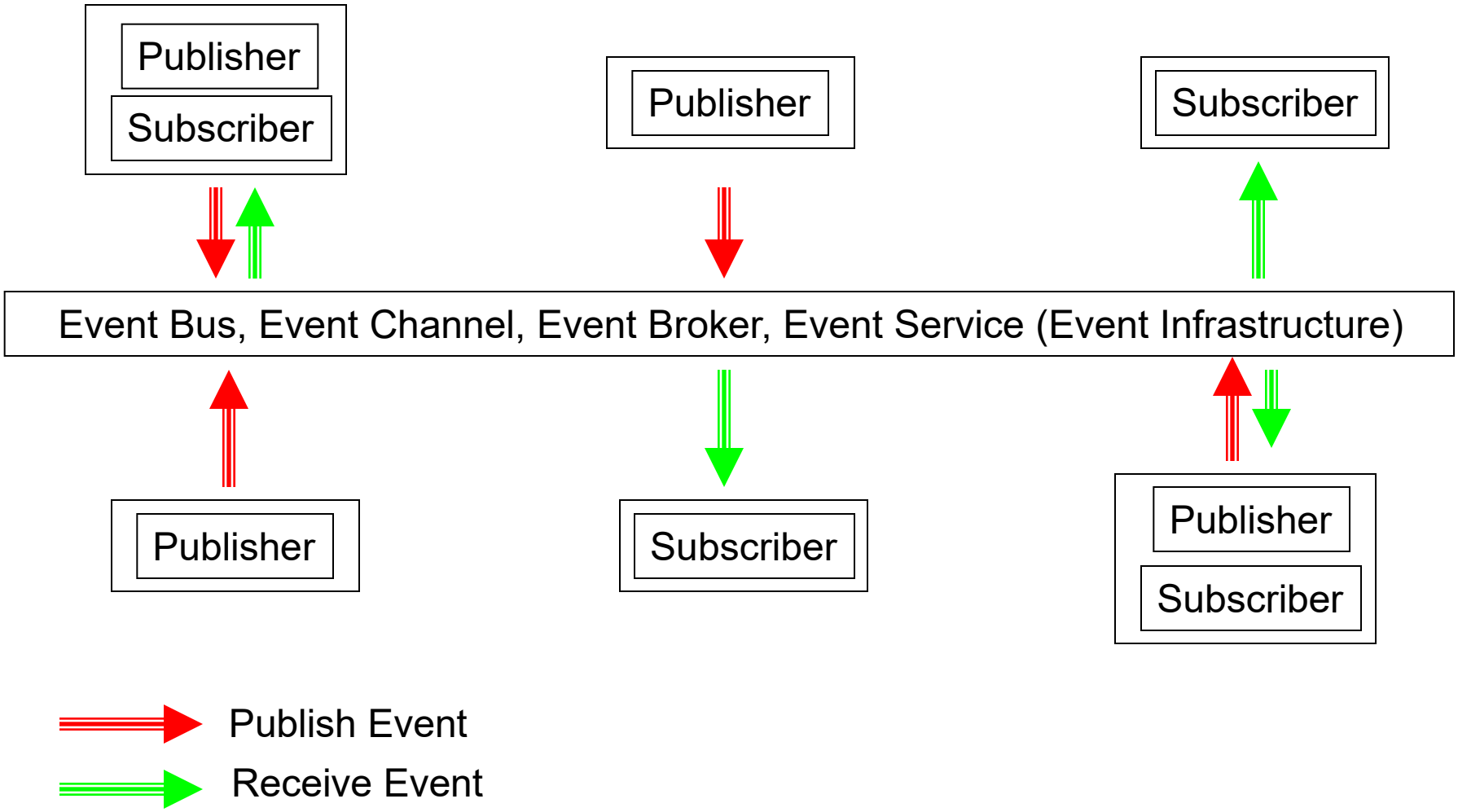
Event-Driven Architectures. Publish-Subscribe

Event-driven arhitecture

Event-driven architecture (EDA) is an architectural pattern where different components of a system communicate by **producing** and **reacting** to **events**.

Event-driven pattern is also called **implicit invocation**: instead of one component directly calling another (explicit invocation), one component emits an event which will have as consequence that another component is called (implicit invocation). There is some **framework/runtime/event infrastructure** which is doing the calling - not the component that emitted the event.

Event Driven Architecture



Event Bus - Description

The central element = types of **Events**

Publishers: components that produce events, but *they do not know the identity of the components that will "consume" these events*.

Subscribers: components that receive certain types of events, but do not know the identity of the components who publish these events

Event Infrastructure (EventBus, EventChannel, EventBroker): distributes events to subscribers

It is possible that different Publishers produce the same types of events.

It is possible that a Subscriber is interested in several different types of events

It is possible that a component is in the same time Publisher and Subscriber (for different event types)

The participants (Publishers or Subscribers) can be dynamically added or removed in the system

Components (Publishers and Subscribers) are loosely coupled, they could be located in different processes

Publisher-Subscriber Models of communication

Usually, when an event is received by (one or more) Subscribers, a data transfer will follow, from the Publisher who generated the event to the receiving Subscribers.

There are different models of communication, according to the type of data transfer:

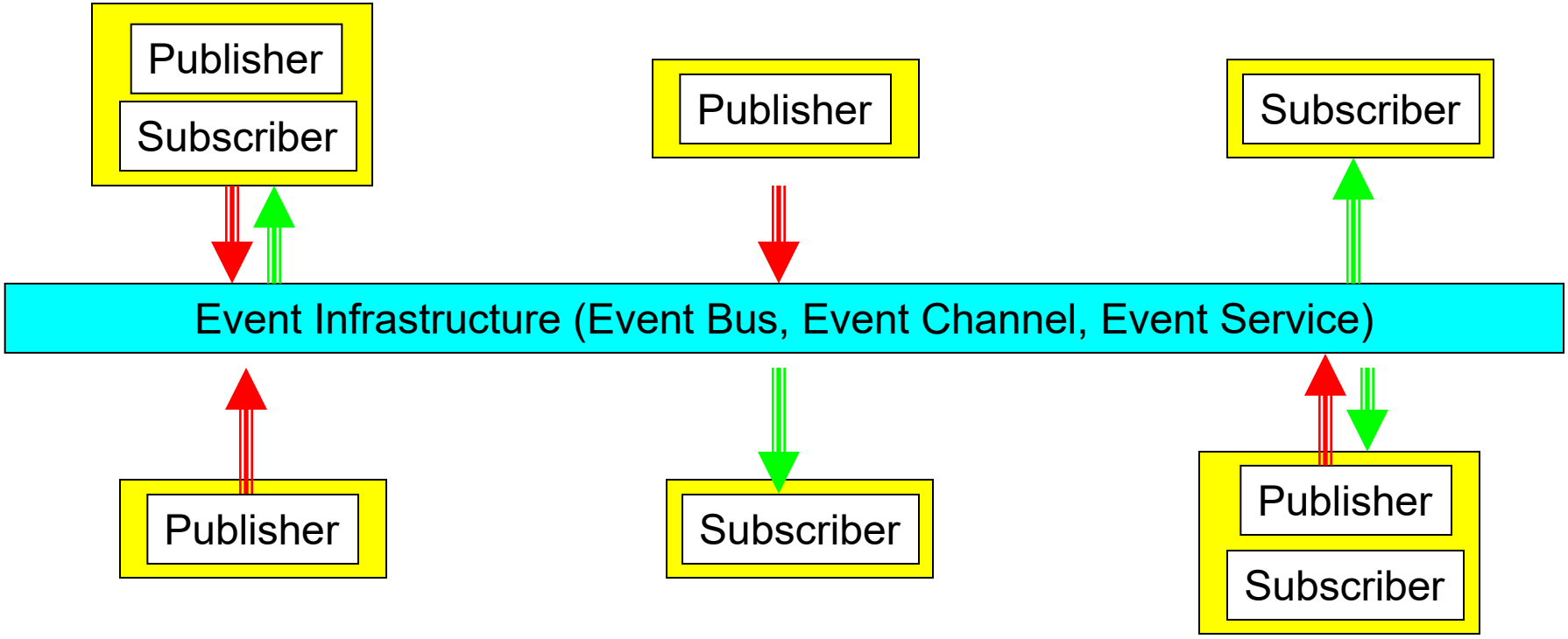
- ***Push***: Publisher is the active source that initiates the transfer, while Subscriber is a passive destination.
  - Subscriber receives notification and all event related data together with the notification
- ***Pull***: Subscriber is active and initiates (requests) the data transfer from a passive source (Publisher) .
  - Subscriber receives notification and if the subscriber wants to receive all the event data it must explicitly request them !

The structure of an event-driven application

Components (Elements): Publishers, Subscribers

Connectors (Relations): publish event, subscribe to event type

Infrastructure: Event Channel (Event Bus, Event Service). Usually it is not part of the application, can be a general-purpose messaging infrastructure !



The Generic Event Infrastructure API

- A generic event infrastructure API must include following features/operations:
  - For Publishers: a method for publishing events
    - Publish(event)
  - For Subscribers: A method for subscribing to certain event types and registering handler function to be called when an event occurs
    - Subscribe(eventType, handler)

Event Service Infrastructures

- **Infrastructures** that provide a subscribe-publish-notify API to be used by applications; Examples:
- Event Bus, in-process:
  - Akka event bus
  - Greenrobot – Android eventbus
- Distributed Publish-Subscribe: part of middleware for distributed computing
  - Message Broker vs. EventBus
  - Java Message Service (JMS)
  - RabbitMQ, ActiveMQ, Kafka

Fundamental architectural styles: Preliminary Conclusions (1)

- They describes ways of structuring a system from different viewpoints:
  - Module viewpoint (static structure): Layers
  - Component & connector viewpoint (dynamic, runtime structure): Pipes-Filters, Repository, Event-driven
- They describe elementary structures
- In real systems, they may appear “pure” or “hybrids”

Fundamental architectural styles: Preliminary Conclusions (2)

- The fundamental archit styles are general structuring solutions, that can be applied independent on a certain programming paradigm or technology
- The components may have different granularities, from objects, components, and up to whole application that are integrated according to these patterns
- Enterprise Integration Patterns: http://www.eaipatterns.com/
    - Pipes and Filters: http://www.eaipatterns.com/PipesAndFilters.html
    - Shared Database (Repository): http://www.eaipatterns.com/SharedDataBaseIntegration.html
    - Messaging (Event-Driven) http://www.eaipatterns.com/Messaging.html
- Cloud integration pattern:
    - Pipes and Filters pattern - Azure Architecture Center | Microsoft Learn
    - Publisher-Subscriber pattern - Azure Architecture Center | Microsoft Learn