Shared Memory

Shared Memory (can only be used for data flow):

- usually possible within a single process
- if different tasks work on one memory region, synchronization is necessary
- files can be used as a shared memory between different processes
- most operating systems provide functionality to reserve a shared memory that can be used by different processes

Control flow vs. data flow

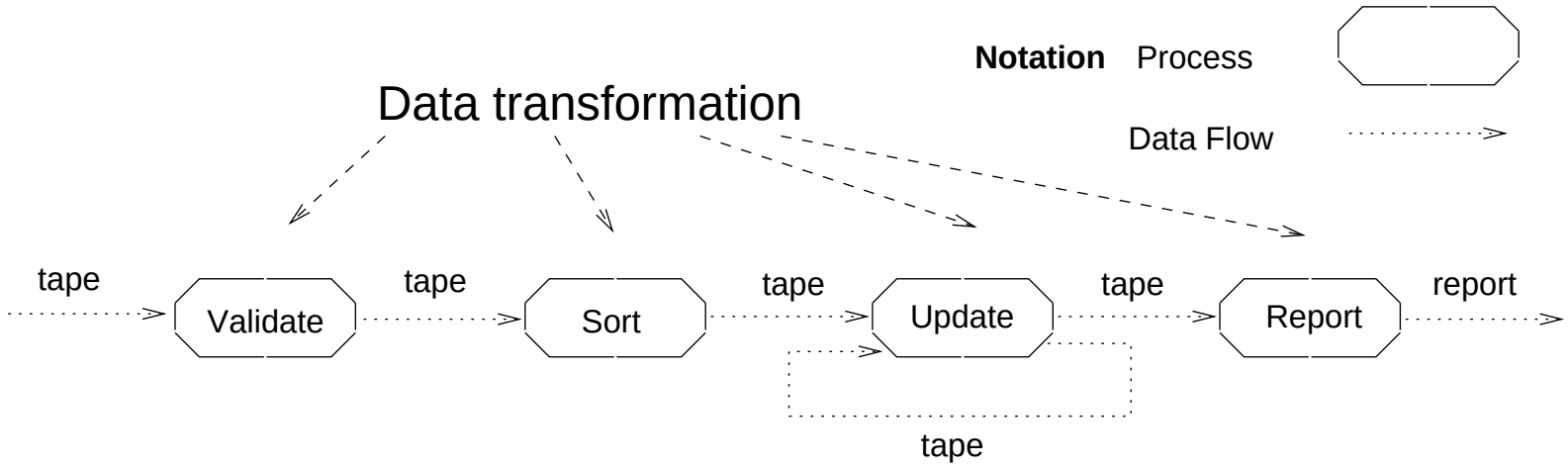
Control flow	Data flow
Decisive question: how does the location of control move through the program?	Decisive question: how does the data move through the program?
Data can go along with control, but is not decisive.	The control is activated where the data is situated.
Important: sequence of computations	Important: availability and transformation of data

Organization of components

- Batch sequential (independent components using file system as shared memory or passive components with call-and-return)
- Pipes & filters (can be implemented with pipes, messages or as call-and-return system)
- Layered architectures (using calls or messages)
- Client-server architecture, using streams (e.g. Sockets) or remote messages (e.g., RPCs)
- Data-centered systems (repositories)
  - Data bases
  - Blackboards
- Event systems (implemented with messages, Observer pattern applied)

Batch sequential

- Processing steps are independent programs that run as different processes
- Each step terminates before the next one begins
- Data are transferred as a whole
- File can be used as a shared memory between the different processes

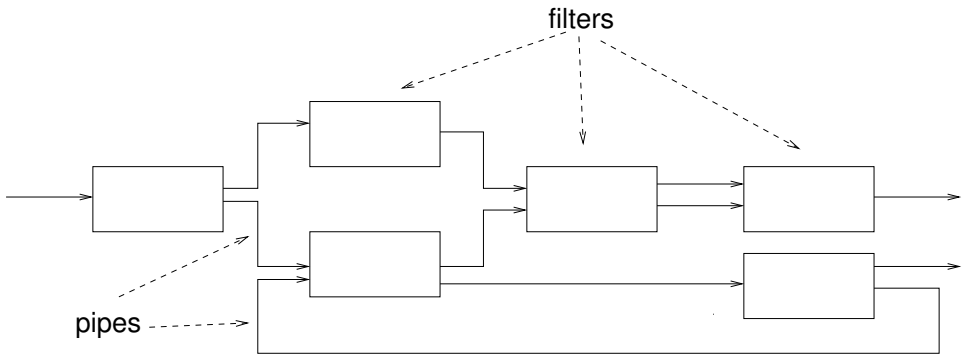


Examples: typical transformational applications such as computing salaries, or the like

Pipes & Filters

Name of that style originates from programs written in the Unix programming environment

- Filters: transform streams of input data into streams of output data in an incremental way
- Pipes: move data from a filter output to a filter input
- General scheme of computation:
  let pipes and filters operate in a non-deterministic manner until no further computations are possible



Specialization: pipelines, i.e., linear sequences of filters

Advantages of pipes & filters architectures

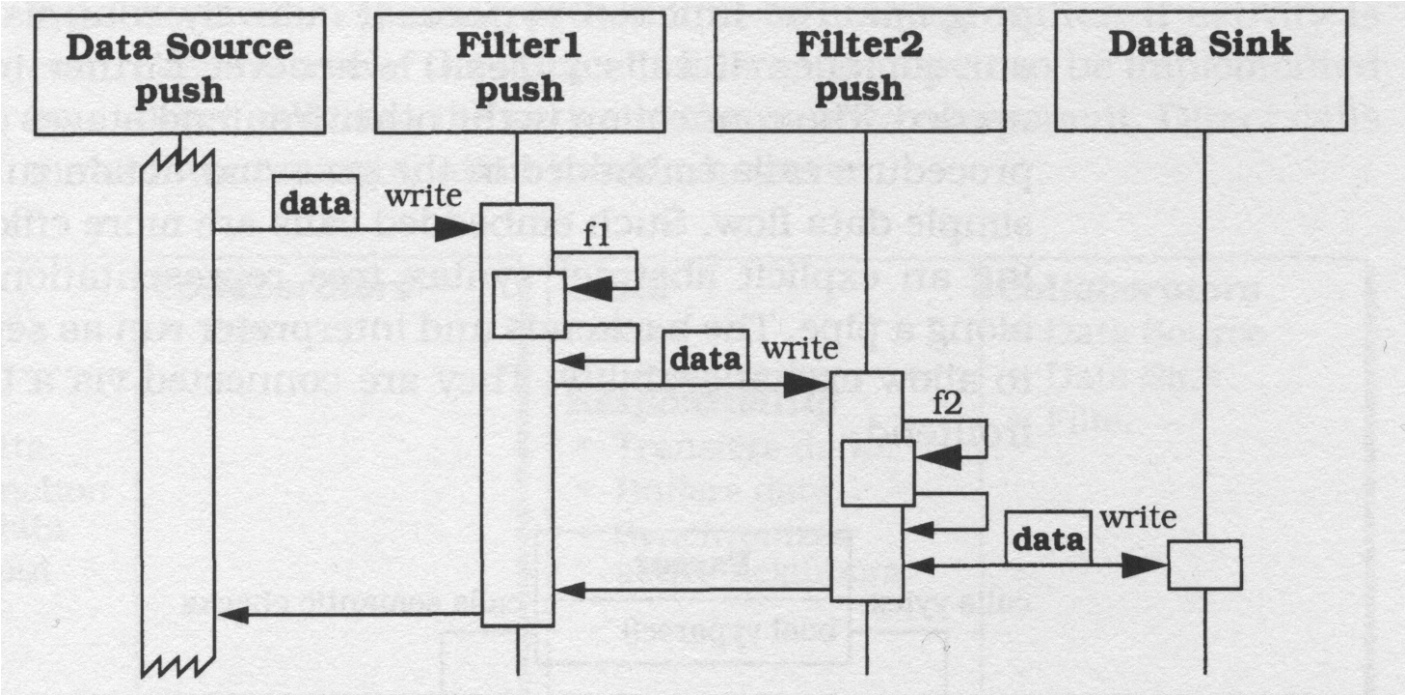
- The overall input-output behavior is determined by a simple composition of the behavior of the individual filters.
- Re-use of filters is possible.
- Easy to maintain and to improve by adding or replacing filters.
- Concurrency is supported in a natural way, because filters can operate independently of each other.
- Can be analyzed well, for example concerning throughput of deadlocks.

Disadvantages of pipes & filters architectures

- Often lead to batch-processing, i.e. concurrency is not utilized
- Not appropriate for interactive applications
- Efficiency may be problematic
- All components have to parse the input

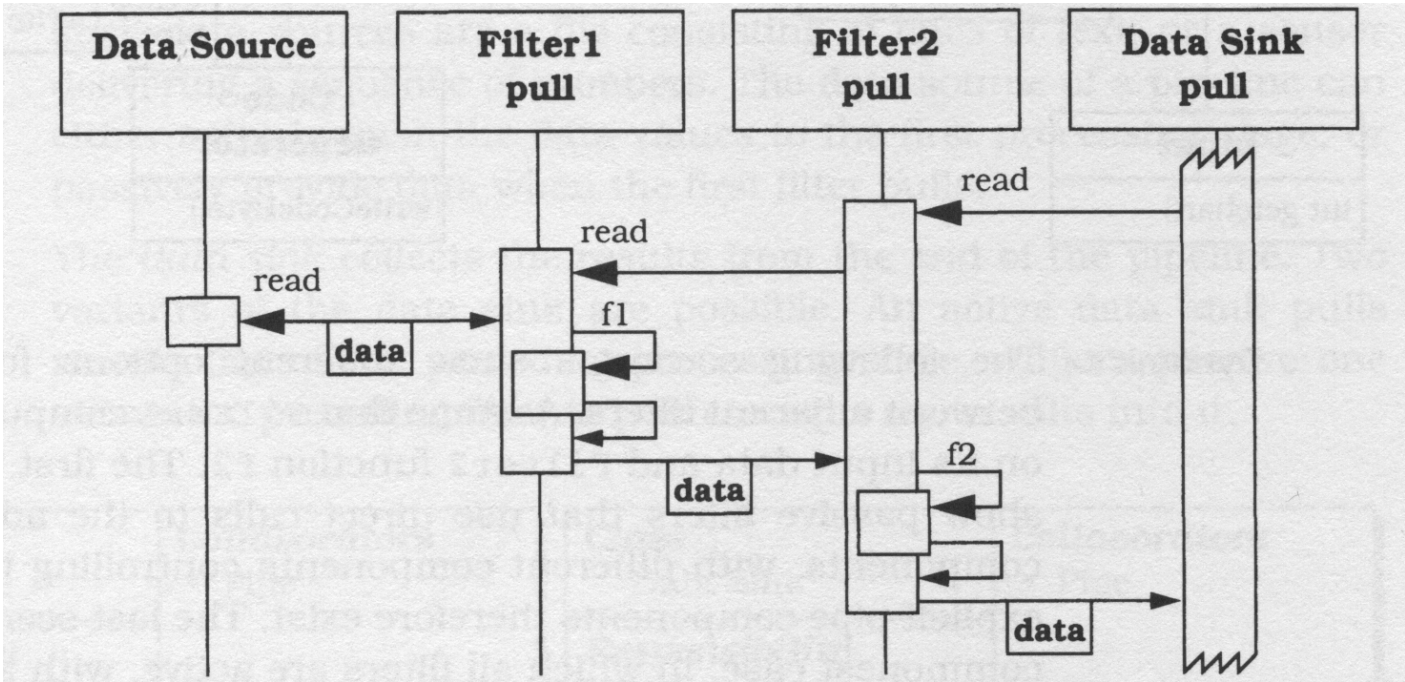
Pipes & filters' dynamic behavior / implementation alternatives (Buschmann et al. (1996)) I

Alternative 1: push pipeline with passive filter components and synchronous calls. Activity starts with the data source.



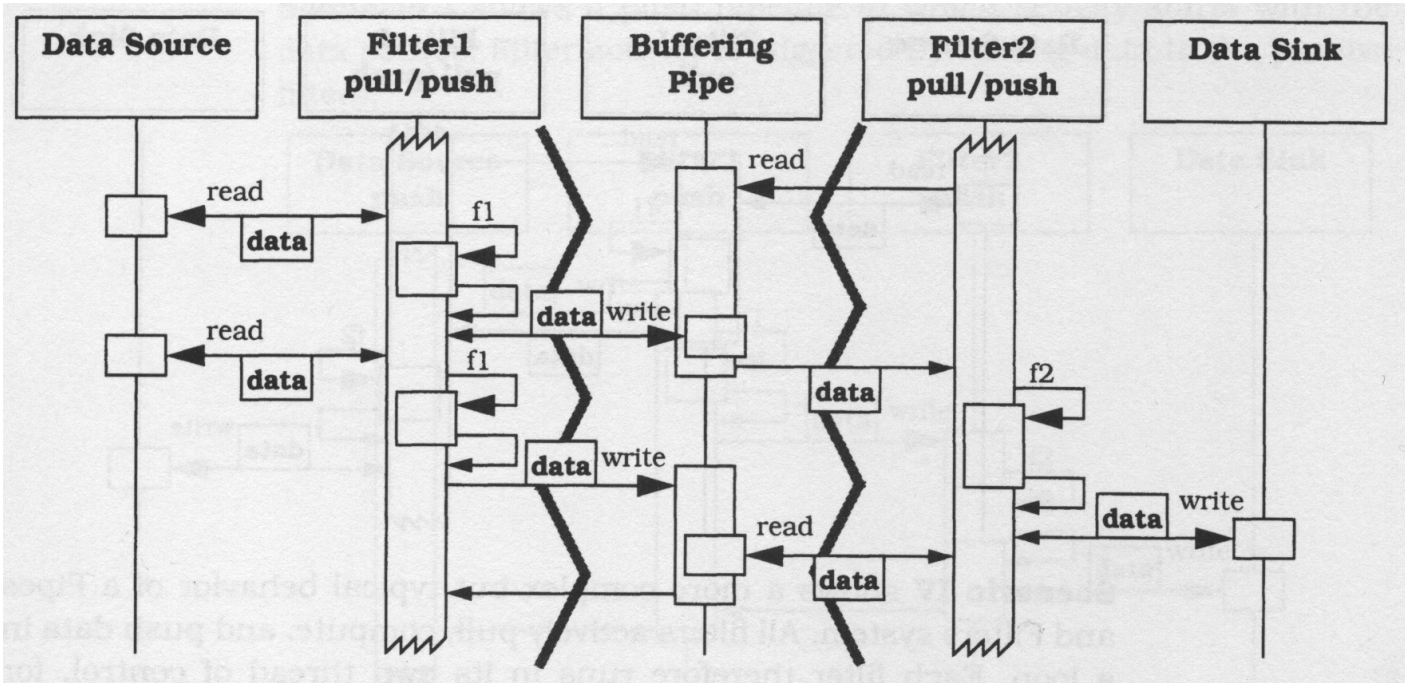
Pipes & filters' dynamic behavior / implementation alternatives (Buschmann et al. (1996)) II

Alternative 2: pull pipeline with passive filter components and synchronous calls. Data sink starts the activity by calling for data.



Pipes & filters' dynamic behavior / implementation alternatives (Buschmann et al. (1996)) III

Alternative 3: pipeline with active filter components that pull, process and than push data. Each filter runs in its own thread of control. Buffering pipes are used for communication and synchronize the flow of data.



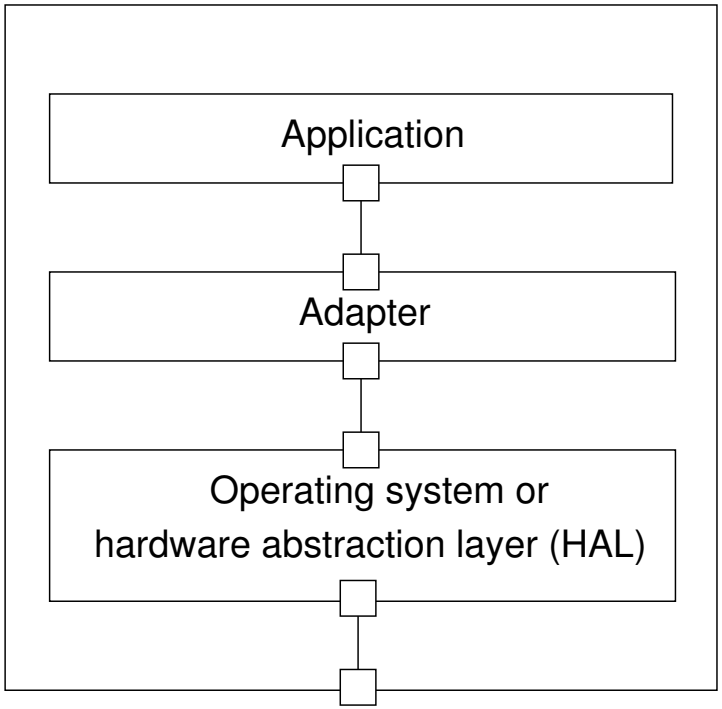
Comparison of batch sequential and pipes & filters

- Both decompose software systems into fixed sequences of computations
- In both cases components interact only through data flow

batch sequential	pipes & filters
coarse-grained, total	fine-grained, incremental
no concurrency	concurrency possible
not interactive	often interactive, but inelegant

Layered architectures

Hierarchical Organization. Each layer offers services for the layers above.



Well-known Example: ISO/OSI-Reference model for communication protocols.

Advantages / disadvantages of layered architectures I

Advantages:

- Design is performed on successive lower abstraction layers, i.e. services are defined at first in an abstract way and then in an increasingly concrete way.
- Can be changed easily, since changes in one layer (should) only effect the adjacent layers.
- Portability is supported.
- Can be implemented as a call-and-return system or composed from independent components.

Advantages / disadvantages of layered architectures II

Disadvantages:

- It is often difficult to identify and clearly separate different abstraction layers.
- The previous reason and reasons of efficiency often lead to *layer bridging* in practice, i.e. not only adjacent layers communicate directly with each other.

Advantages / disadvantages of layered architectures III

Reasons for a layered architecture:

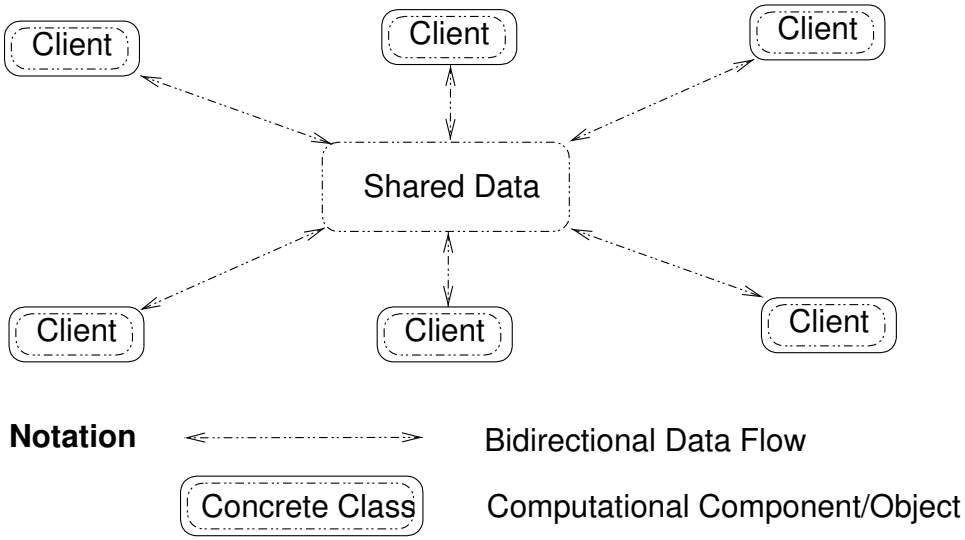
- If the software tasks can be divided into classes, of which one is application-specific and the other is usable for several applications, but platform specific, consider using a layered architecture.
- Also consider using a layered architecture, if the software should be portable or an already developed infrastructure can be used.

Client-server architectures

A server serves several clients, which are usually distributed over a net. Service requests are always initiated by the client, and can be served in a synchronous or asynchronous way.

- Example: web-server and browser (client)
- The repository architecture style is a special client-server architecture
- *Client-dispatcher-server* design pattern is often applied

Data-Centered Systems (Repositories) I



Characteristics:

- The integration of data is an important goal.
- The software can be described by describing how the repository can be used and changed by the different parties.
- Components that access the repository are relatively independent from each other, and the repository is independent of them.

Data-Centered Systems (Repositories) II

- New components can be easily added and are not effected by changes of other components

If components act independently from each other, then such a repository architecture is a client-server-architecture at the same time $\Longrightarrow$ Architectural styles are not disjoint!

Databases

The data storage is passive, the sequence of the operations is defined through the input streams.

Blackboards

A blackboard is an active repository: it sends messages to interested components, when certain data has changed. Overlap with event/action-style.

Data-Centered Systems (Repositories) III

Heuristics for Repository Architectures

- Central problem is the storage, representation, administration as well as access to a large number of connected, persistent data.
- Choose a database architecture, if the execution order of the components is determined through a stream of queries and transactions, and if the data are highly structured or in case a commercial database system is available, which can then be used for the desired purpose.
- Choose a blackboard architecture, if consumer and producer of data should be easily exchangeable.
- If it is probable that the representation of data will change, prefer an object-oriented architecture.

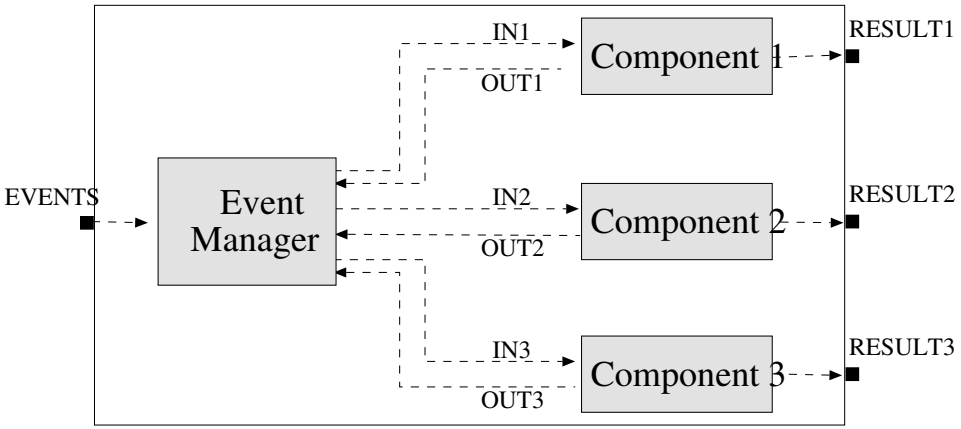
Event systems I

- Also called *Event/Action*-Style
- Independent components do not need to know each other
- Components publish that they offer certain data or services
- Other components announce interest in particular events or data
  $\Longrightarrow$ publish/subscribe-principle
- Often an event- or message manager is responsible for distributing the messages

Choose an event-system, if

- producers and consumers of events should be decoupled.
- scalability is important. Here, new processes can be added, that react to already defined events.

Event systems II



- Each component defines incoming procedure calls and outgoing events in its interface
- The communication among components takes place by publishing events that trigger procedure calls
- Sequence of the called procedures is not deterministic
- Decoupling of implementation and use of components
- Implemented using the *Observer* design pattern

Relation between architectural patterns and design patterns I

Event Systems:

- Implemented using *Observer* pattern

Components:

- Structured using *Facade* pattern
- A component is often implemented using the *Singleton* pattern

User Interface Components:

- Implemented using *MVC* pattern (with *Composite*, *Observer*, *Strategy*, and *Factory Method*)

Relation between architectural patterns and design patterns II

Streams:
If a stream interface is given but messages should be exchanged efficiently, apply

- *Forwarder-Receiver* pattern

Remote Procedure Call (RPC):
In RPC implementations the following design patterns are applied:

- *Client-Dispatcher-Server* pattern to locate the service
- *Proxy* pattern for the operation stubs of the client

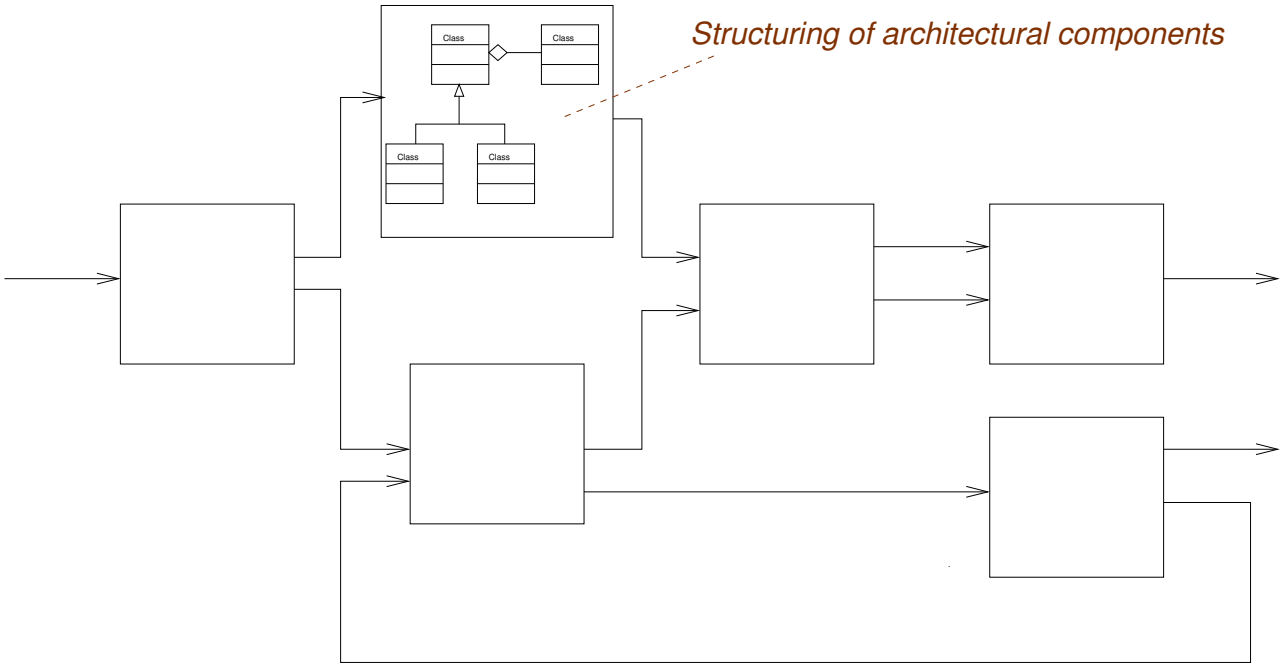
What have we learned on architectural patterns?

- The the software system needs to be structured.
- That structure is called architecture of the software system. It consists of components and connectors.
- Software architectures describe the structure of the solution of a problem.
- Software architectures can be classified. These classes are called architectural styles.
- Usually, several architectures can be used to structure a software. These differ in non-functional characteristics (**quality attributes**).

Design Patterns

Design patterns (Gamma et al. (1995)): characterized by

- Usage for **detailed design**
- Object-oriented paradigm
- "Description of a family of solutions for a software design problem" (Tichy)



Types of design patterns (Gamma et al. (1995))

- **creational**
  concern the process of object creation
- **structural**
  deal with the composition of classes or objects
- **behavioral**
  characterize the ways in which classes or objects interact and distribute responsibility

Second criterion: **scope**
specifies whether the pattern applies primarily to classes or to objects.

Types of design patterns (Tichy) I

- Coupling/decoupling patterns
  System is divided into units that can be changed independently from each other
  e.g. Iterator, Facade, Proxy
- Unification patterns
  Similarities are extracted and only described at one place.
  e.g. Composite, Abstract Factory
- Data-structure patterns
  Process states of objects independently of their responsibilities
  e.g. Memento, Singleton

Types of design patterns (Tichy) II

- Control flow patterns
  Influence the control flow; provide for the right method to be called at the right time
  e.g. Strategy, Visitor
- Virtual machines
  Receive programs and data as input, execute programs according to data
  e.g. Interpreter
  (Remark: no clear boundary to architectural styles)

Advantages of design patterns (Tichy)

- Improvement of team communication
  Design pattern as “short formula” in discussions
- Compilation of essential concepts, expressed in a concrete form
- Documentation of the “state of the art”
  Help for less experienced designers, not constantly reinventing the wheel
- Improvement of the code quality
  Given structure, code examples

Description of design patterns (Gamma et al. (1995)) I

Name and Classification A good name is important, because it will become part of the design vocabulary.

Intent What does the pattern do? Which problems does it solve?

Also Known As Other familiar names.

Motivation Scenario which illustrates the design problem and how the pattern solves the problem.

Applicability What are the situations in which the design pattern can be applied? How can one recognize these situations?

Structure Class and interaction diagrams.

Participants Classes and objects, which are part of the pattern, as well as their responsibilities.

Description of design patterns (Gamma et al. (1995)) II

Collaborations How do the participants collaborate to carry out their responsibilities?

Consequences What are the trade-offs and results of using the pattern? What aspect of system structure does it let one vary independently?

Implementation What pitfalls, hints, or techniques should one be aware of when implementing the pattern? Are there any language-specific issues?

Sample Code Code fragments in C^{++} or Smalltalk.

Known Uses At least two examples of applications taken from existing systems of different fields.

Related Patterns Similar patterns and patterns that are often used in combination with the described pattern.