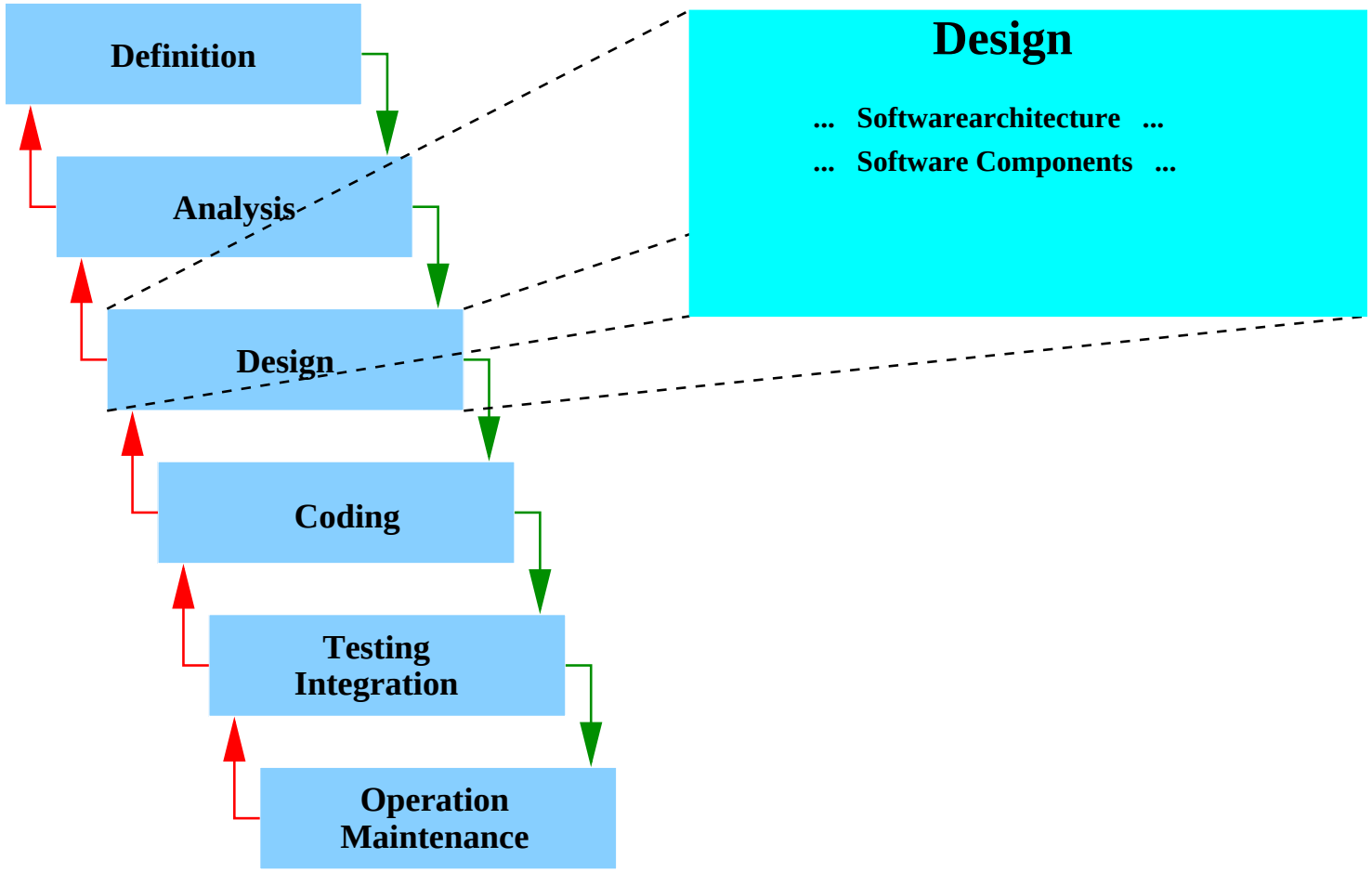
5 Design — an Overview



The Design Phase

- "Programming in the large"
- **GOAL:** transform requirements (requirements specification, product model) into a **software architecture**
- decomposition into components
- SW architecture $\hat{=}$ components and connectors
- component
  - designated computational unit with specified interface
  - Examples: client, server, filter, layer, database
- connector
  - interaction point between components
  - Examples: procedure call, event broadcast, pipe

5.1 Architectural Styles — Overview

Dataflow systems

Batch sequential, Pipes and filters

Call-and-return systems

Main program and subroutine, OO systems, Hierarchical layers

Independent components

Communicating processes, Event systems

Virtual machines

Interpreters, Rule-based systems

Data-centered systems (repositories)

Databases, Hypertext systems, Blackboards

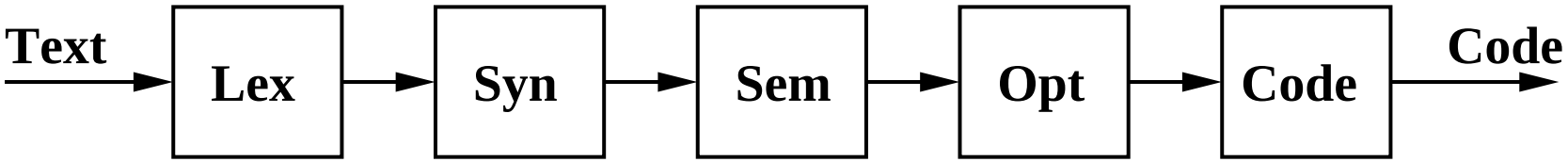
(according to Shaw and Garlan, Software Architecture, Prentice Hall)

Classification of an Architectural Style

- design vocabulary—types of components and connectors
- allowable structural patterns
- underlying computational model (semantic model)
- essential invariants
- common examples of use
- advantages/disadvantages
- common specializations

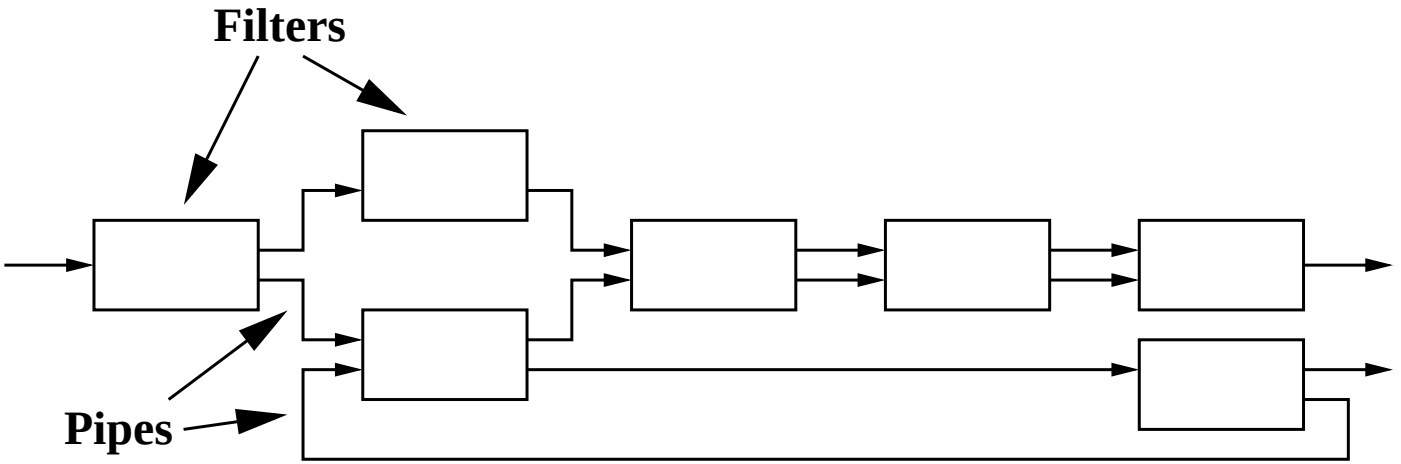
5.1.1 Batch Sequential

- separate passes
- each runs to completion before the next starts
- Example: traditional compiler architecture



5.1.2 Pipes and Filters

- each component (filter) transforms input streams to output streams incrementally
- buffered channels (pipes) connect inputs to outputs
- filters are independent entities
- common specializations: pipeline (linear sequence of filters), bounded pipes, typed pipes

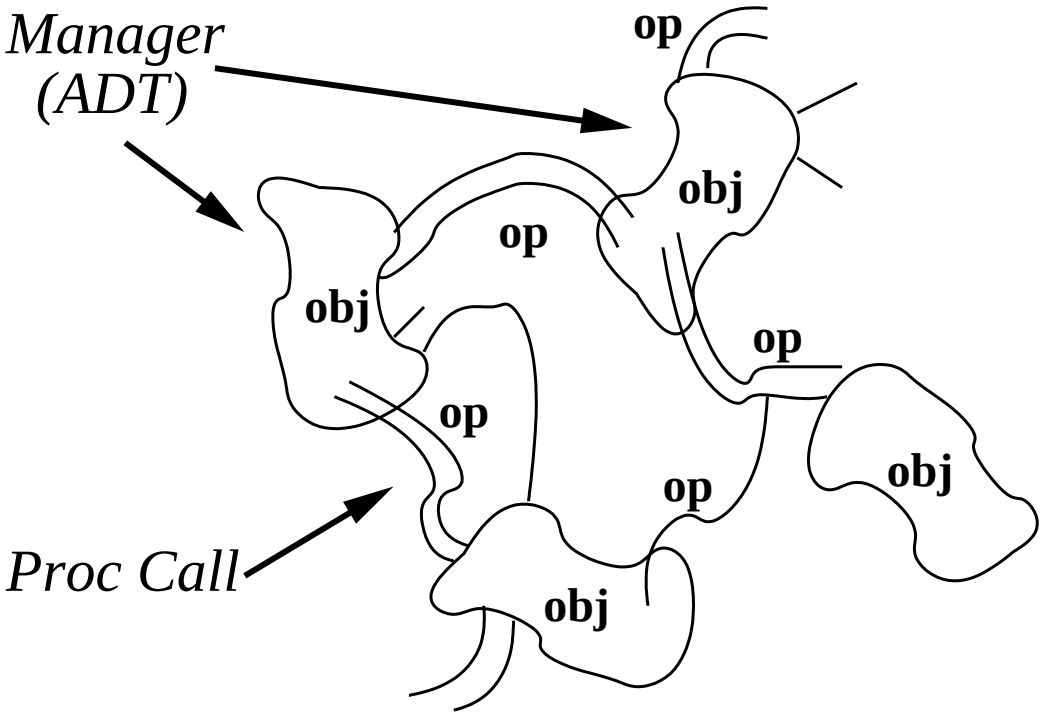


Properties of Pipes and Filters

+ global understanding supported
+ reuse supported
+ easy to maintain and enhance
+ specialized analysis supported
+ potential for concurrent execution
– interactive applications
– correpondences between streams
– common format for data transmission

5.1.3 Object-Oriented Organization

- components: objects, connectors: method invocation
- each object is responsible for its integrity
- each object's representation is hidden



Properties of OO Organization

+ implementation can be changed without affecting clients
+ bundling of operations with data
– objects must know their interaction partners (contrast with filters!)

5.1.4 Event-based, Implicit Invocation

- also called *reactive integration* or *selective broadcast*
- each component may
  - announce events
  - register an interest in certain events, associated with a callback
- when event occurs, the system invokes all registered callbacks

$\Rightarrow$ announcer of event does not know which components are registered

- order of callback invocation cannot be assumed
- applications: integration of tools, maintaining consistency constraints, incremental checking

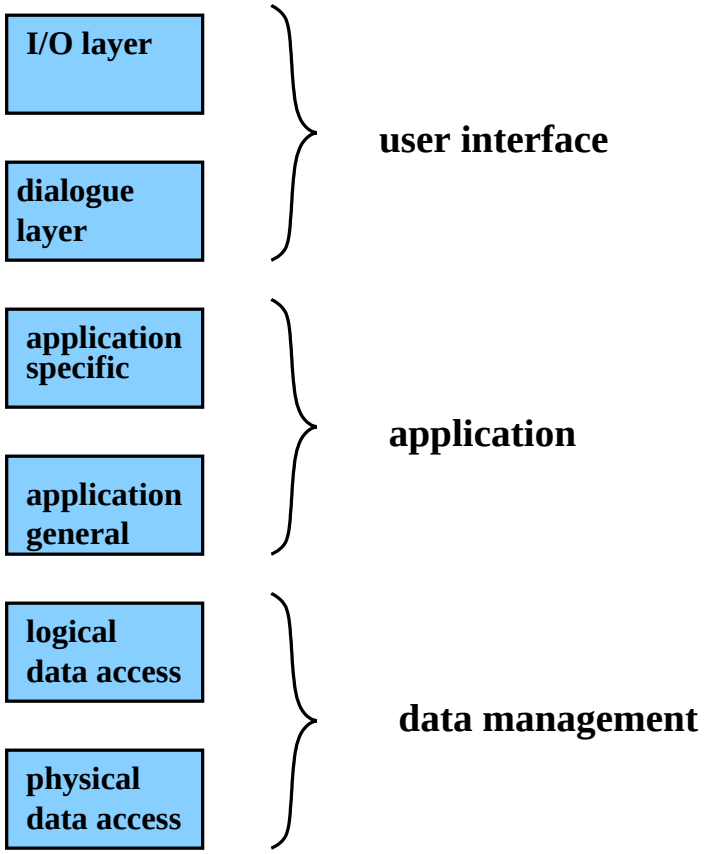
Properties of Implicit Invocation

+ strongly supports reuse
+ eases system evolution
– lack of control
– data passed through shared repository
– correctness?

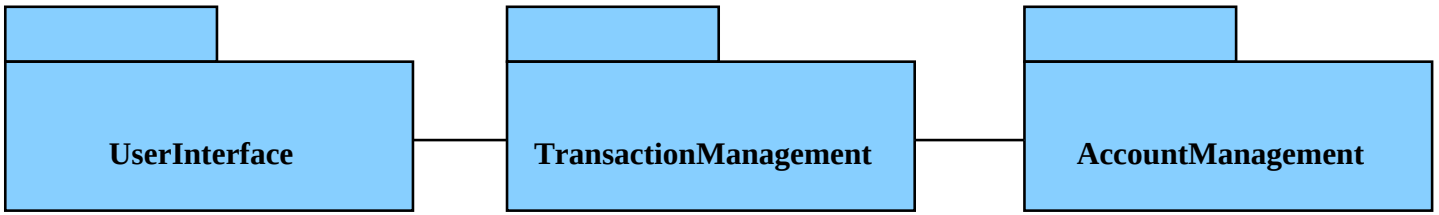
5.1.5 Layered Systems

- hierarchy of system components, grouped in layers
- inside of layer: arbitrary access between components
- between layers
  - access restricted to lower layers: linear, strict, treeshaped
  - small interfaces
- **advantages:** clarity, reusability, maintainability, testability
- **disadvantages:** not always appropriate, loss of efficiency, no restrictions inside layers
- examples: communication protocols (OSI), database systems, operating systems

Typical Setup



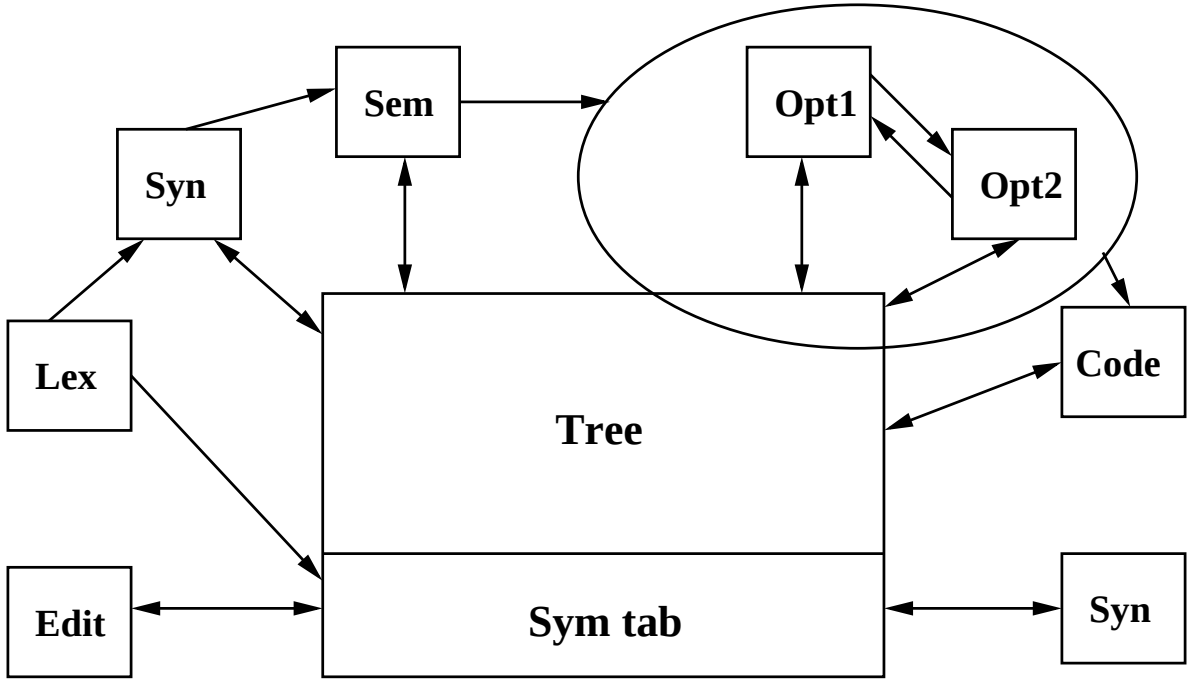
Example: Three-Tier Architecture



- Three kinds of subsystems
  - user interface
  - control
  - database
- Enables consistent look-and-feel
- Useful with single data repository
- Web architecture: Browser, Webserver, Applicationserver

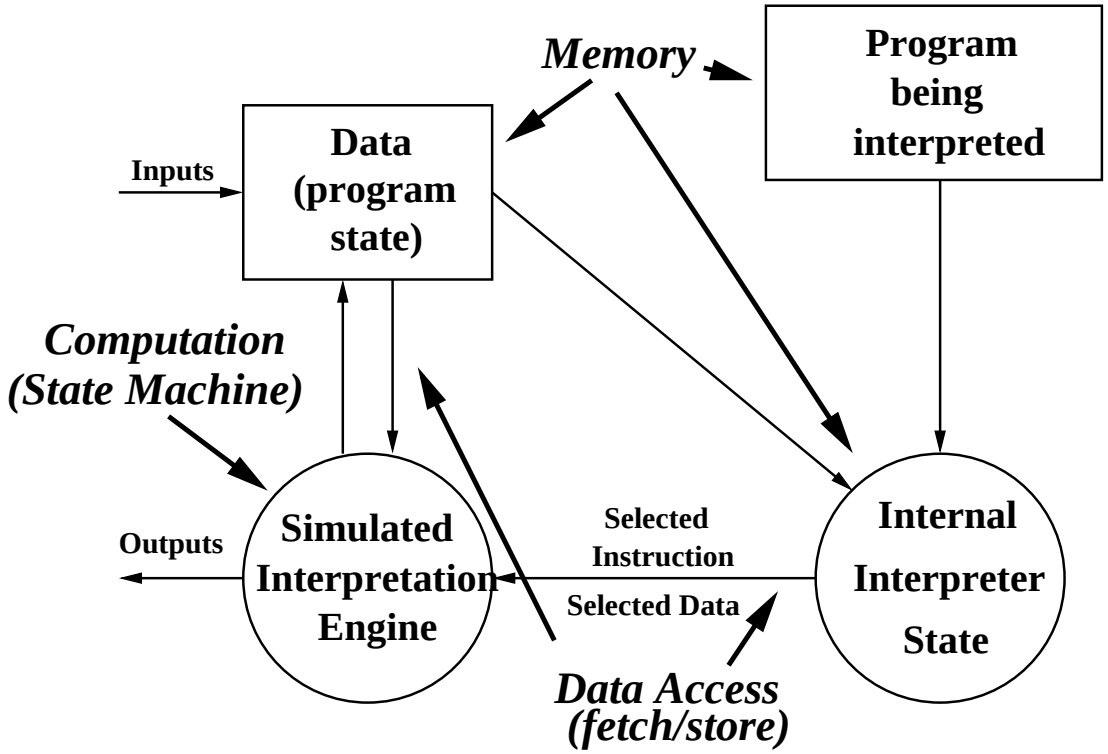
5.1.6 Repositories

- central data structure (current state)
- independent components acting on it
- Example: architecture of modern compiler



5.1.7 Interpreters

- virtual machine in software
- pseudoprogram + interpretation engine
- Example: a programming language



5.1.8 Process Control

- systems that control physical processes
- based on process control loops
- terminology

  Process variable. Measurable property of a process

  Controlled variable. PV to be controlled by the system

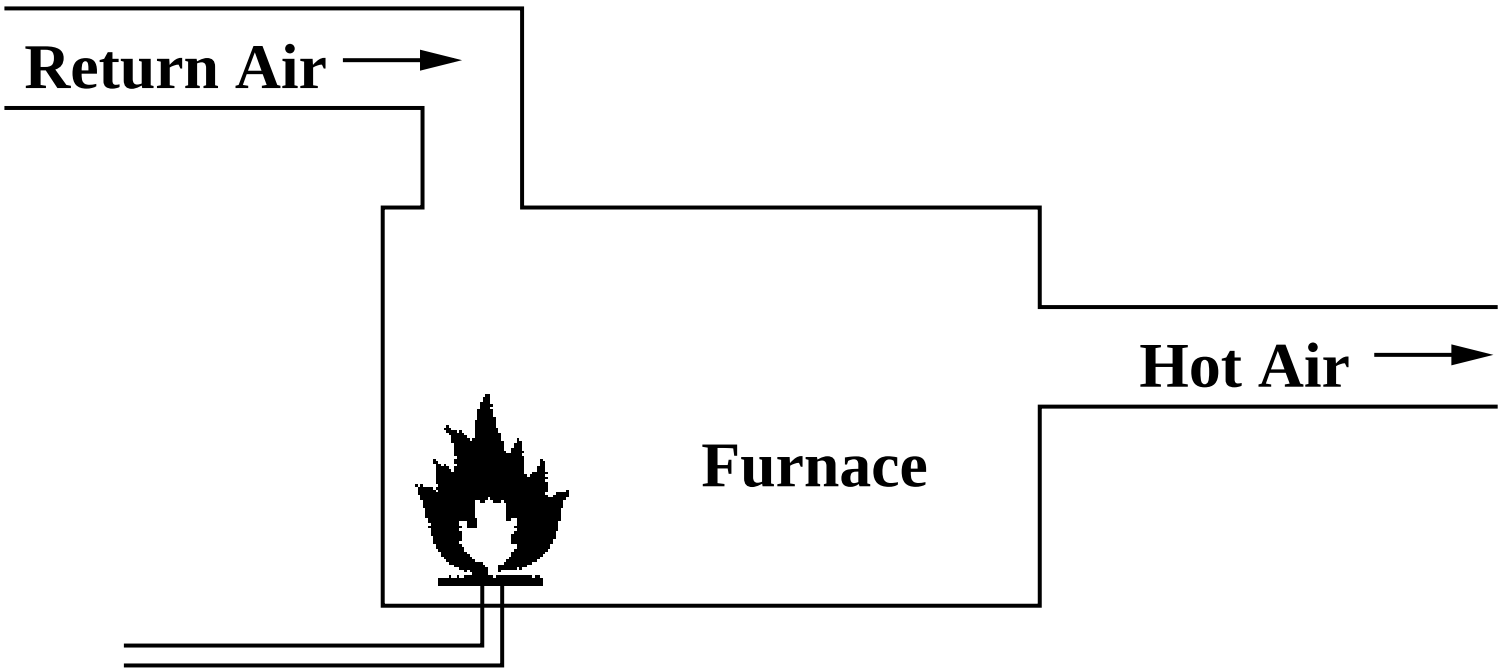
  Input variable. PV measuring input to process

  Manipulated variable. PV that can be changed by controller

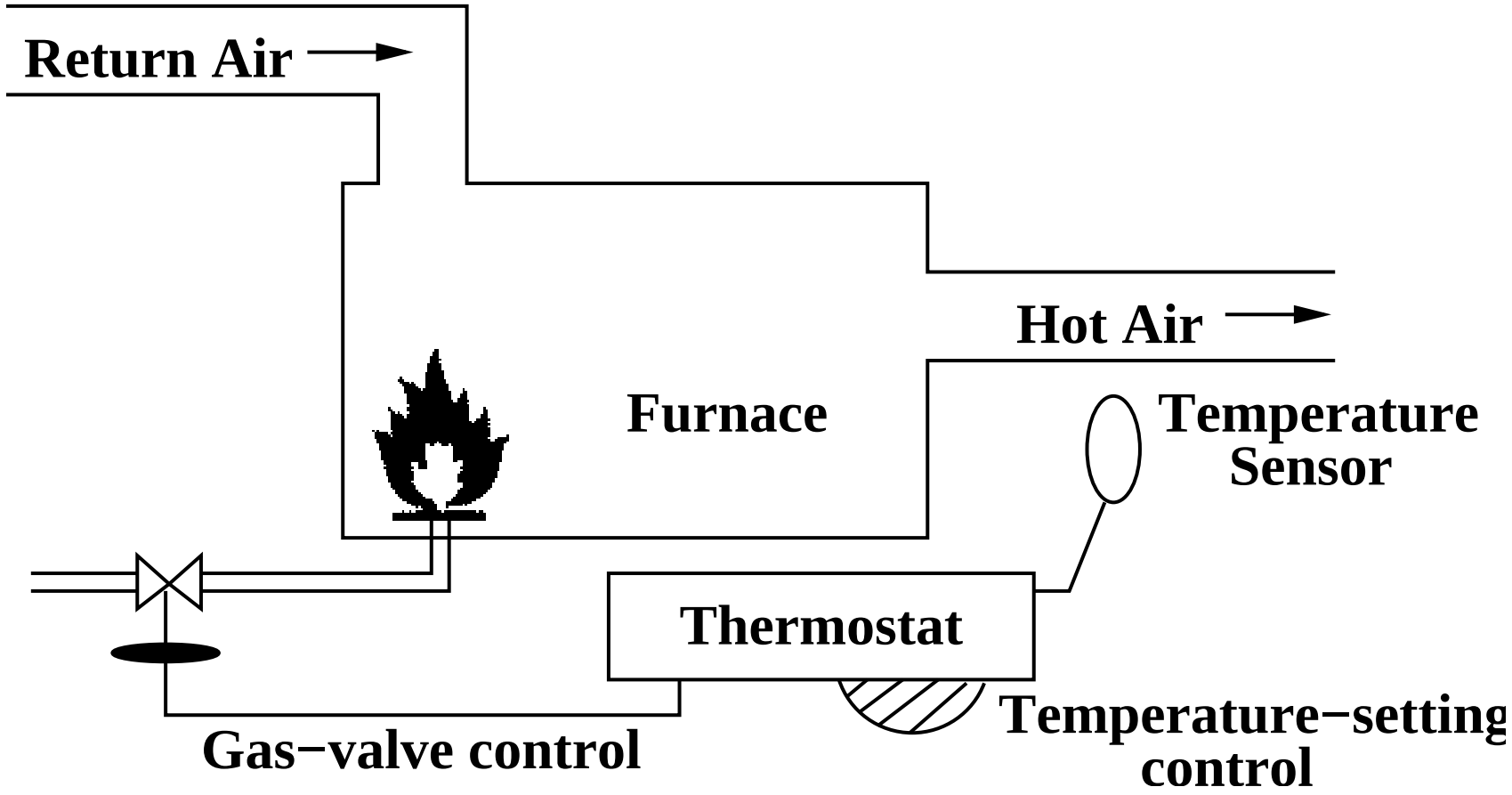
  Set point. Desired value for controlled variable

- basic design choices:
  - open-loop vs closed-loop
  - feedback control vs feedforward control

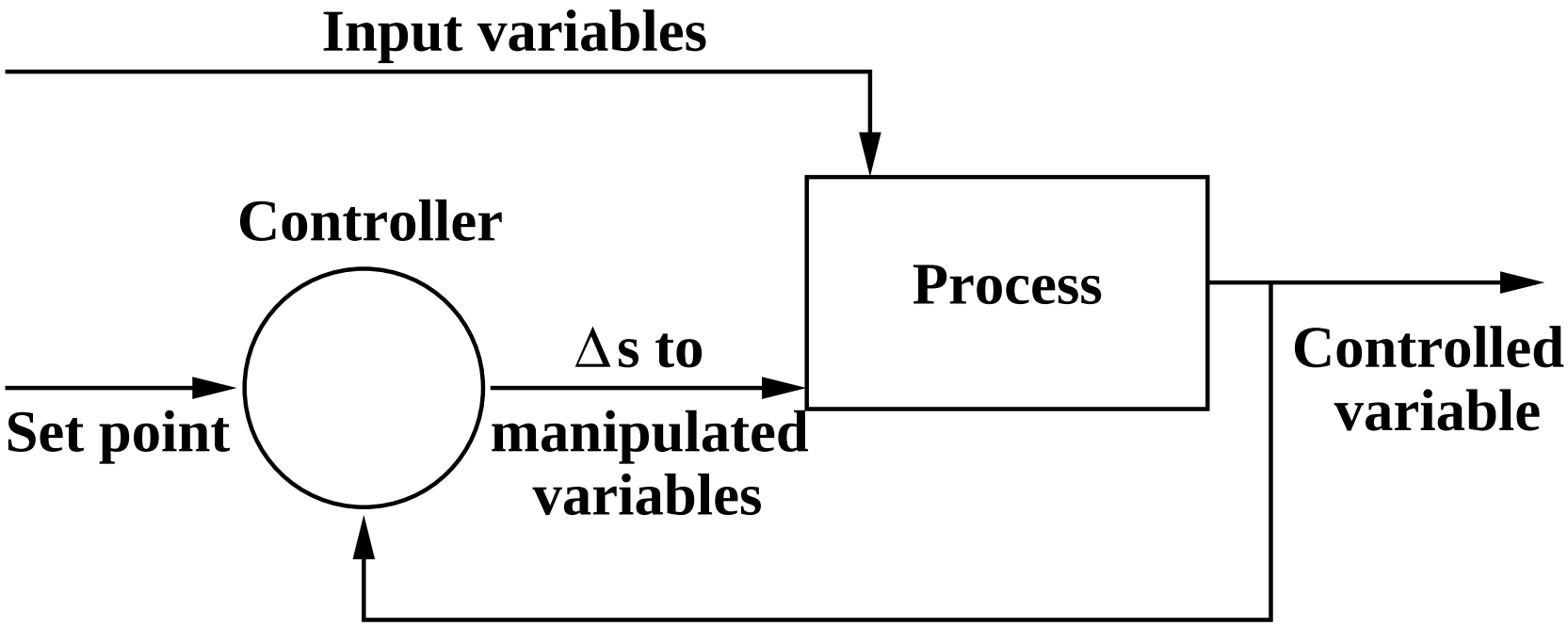
Open-Loop Temperature Control



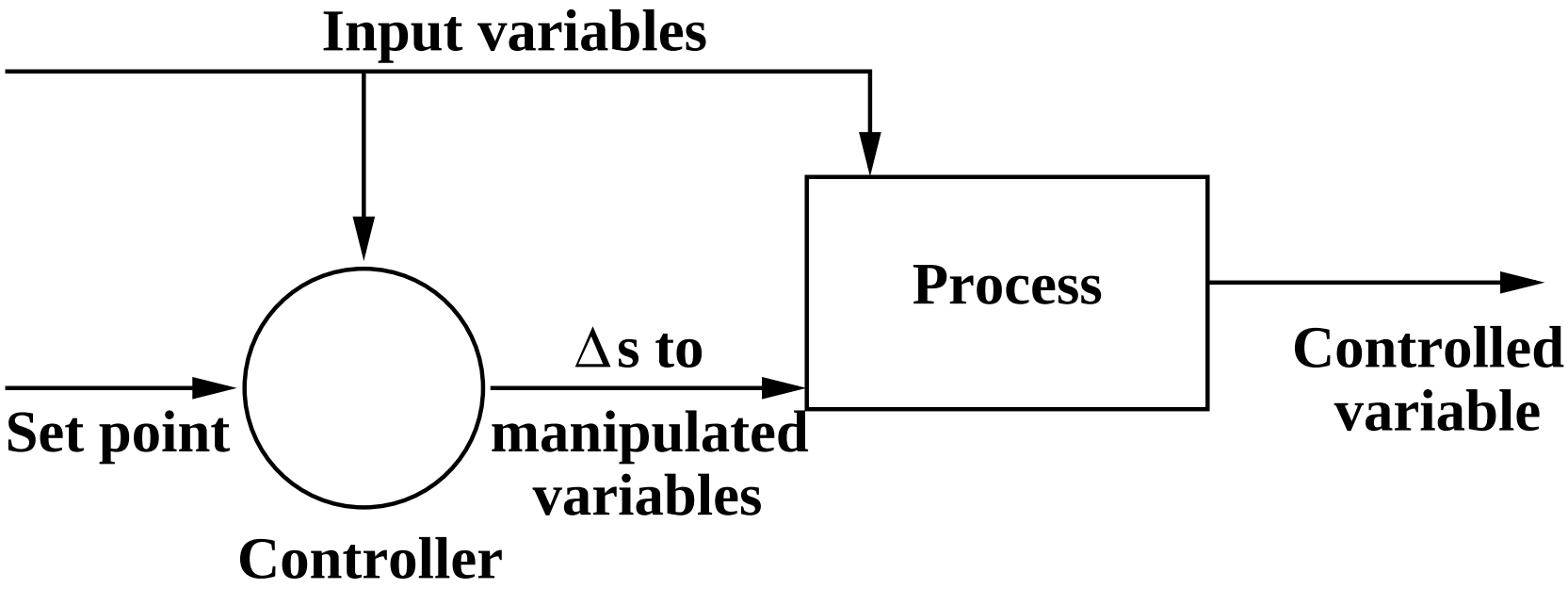
Closed-Loop Temperature Control



Feedback Control



Feedforward Control



5.1.9 Further Architectural Styles

- Distributed processes
  - topological features
  - interprocess protocols
  - client-server organization
- Main program/subroutine: mirroring the programming language
- Domain specific SW architectures
  - tailored to family of applications
  - Examples: avionics, vehicle management, ...
- State transition systems
- Combinations of architectural styles
  - hierarchically
  - mixture of connectors

5.2 Developing a Software Architecture

- The choice of a software architecture is a far reaching decision that can influence the effort required to change the system later on.
- Criteria for decomposition
- Two case studies from Shaw and Garlan
- Formalization of an architecture

5.2.1 Criteria used for decomposition

⇒ major processing step

⇒ Information hiding: encapsulate a design decision
e.g., input format, data layout, choice of algorithm, computed data vs. stored data, ...

- Cohesion: qualitative measure of dependency of items within a single component

⇒ Maximize cohesion: all elements of a component should contribute to the performance of a single function

- Coupling: qualitative measure of interdependence of a collection of components

⇒ Minimize coupling: component only receives data essential for performing its function

Kinds of Cohesion

- Coincidental Cohesion : (Worst) Component performs multiple unrelated actions
- Logical Cohesion : Elements perform similar activities as selected from outside component
- Temporal Cohesion : Elements are related in time (*e.g.* `initialization()` or `FatalErrorShutdown()`)
- Procedural Cohesion : Elements involved in different but sequential activities
- Communicational Cohesion : Elements involved in different activities based on same input info
- Sequential Cohesion : output from one function is input to next (pipeline)
- Informational Cohesion : independent actions on same data structure
- Functional Cohesion : all elements contribute to a single, well-defined task

Kinds of Coupling

- Content Coupling : (worst) component directly references data in another
- Control Coupling : 2 components communicating with a control flag
- Common Coupling : 2 components communicating via global data
- Stamp Coupling : Communicating via a data structure passed as a parameter. The data structure holds more information than the recipient needs.
- Data Coupling : (best) Communicating via parameter passing. The parameters passed are only those that the recipient needs.
- No coupling : independent components.

5.2.2 Key Word in Context [KWIC]

The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

David L. Parnas.
On the criteria to be used in decomposing systems into modules.
Communications of the ACM, 15(12):1053-1058, December 1972

- Classical problem with practical applications
- Here: four different designs
- Assessment

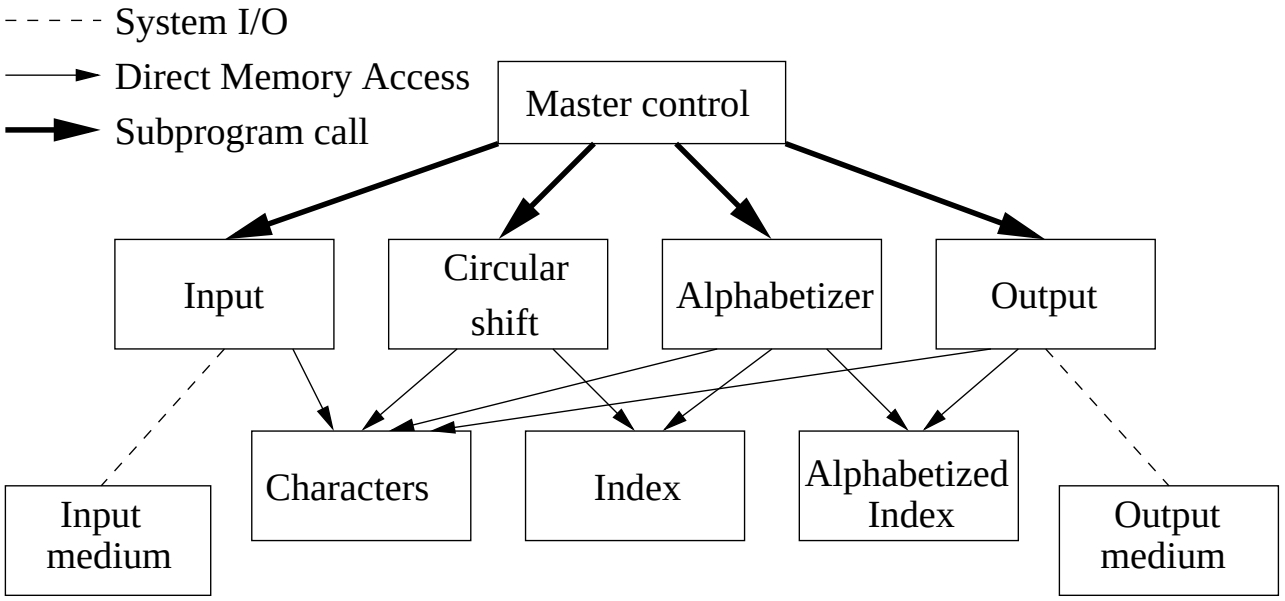
Guidelines for Assessment

Is the architecure amenable to . . .

- Changes in processing algorithm
  Example: line shifting
- Changes in data representation
- Enhancement to system function
  Example: noise words, interactive
- Reuse
- Good performance

Solution 1: Main program/subroutine with shared data

- Four basic functions: input, shift, alphabetize, and output
- subroutines coordinated by main program
- shared storage with unconstrained access
  (why does this work?)



Solution 1: Assessment

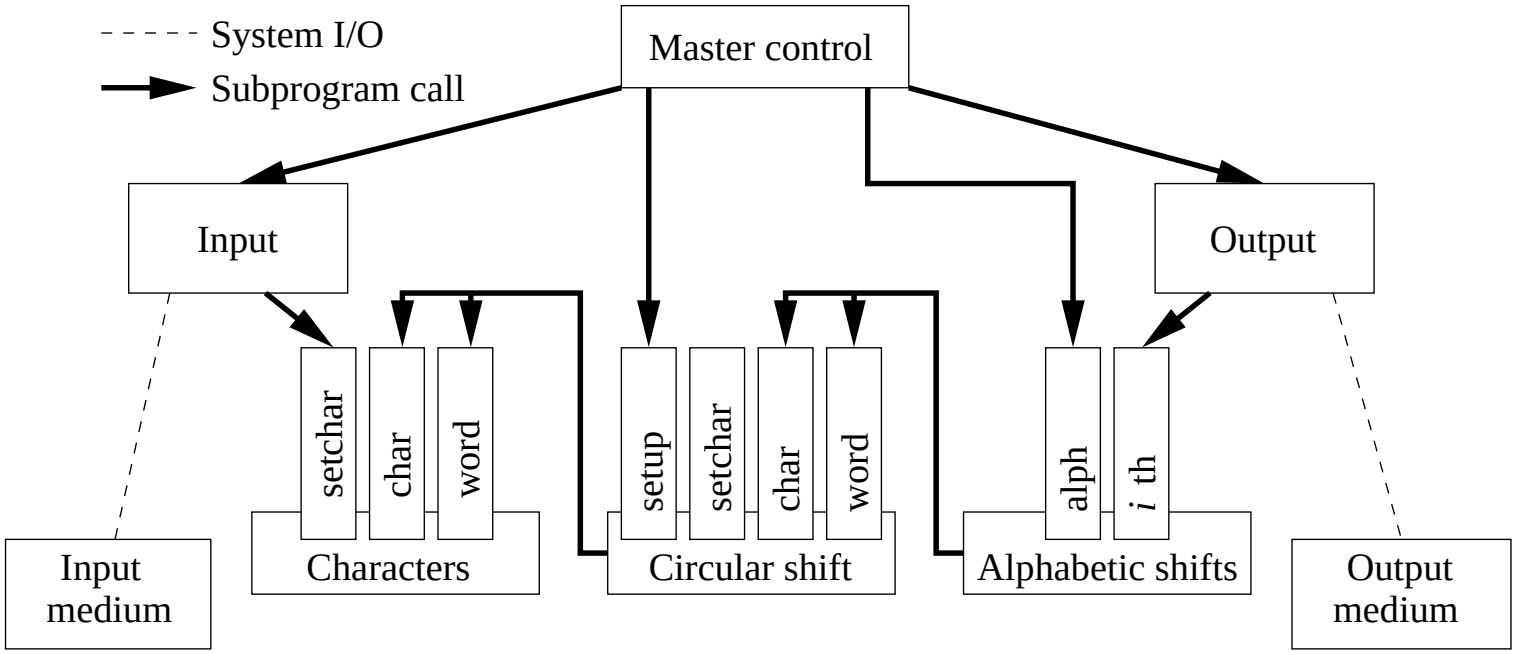
+ efficient data representation
+ distinct computational aspects are isolated in different modules

but serious drawbacks in terms of its ability to handle changes

– change in data storage format will affect almost all of the modules
– similarly: changes in algorithm and enhancements to system function
– reuse is not well-supported because each module of the system is tied tightly to this particular application

Solution 2: Abstract data types

- decomposition into five modules
- data no longer shared
- access through procedural interfaces



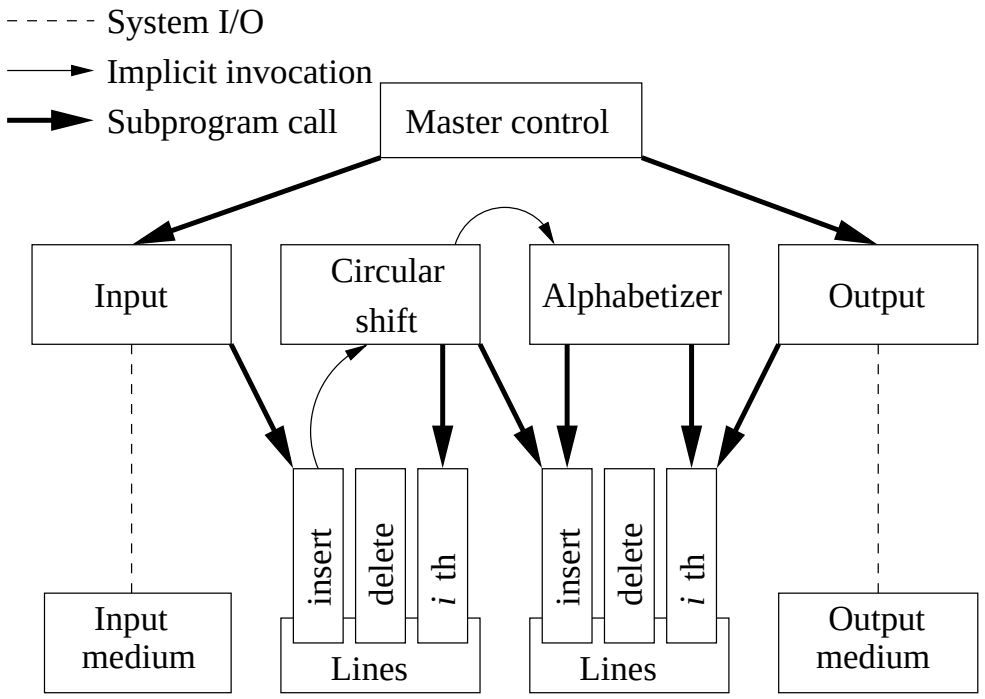
Solution 2: Assessment

Same processing modules as the first solution, but better amenable to change.

+ algorithms and data representations can be changed in individual modules without affecting others
+ reuse is better supported because modules make fewer assumptions about the others with which they interact
– not well suited to enhancements: to add new functionality
  – modify the existing modules—compromising their simplicity and integrity—or
  – add new modules that lead to performance penalties.

Solution 3: Implicit Invocation

- component integration based on shared data
- but abstract access to data
- operations invoked implicitly as data is modified

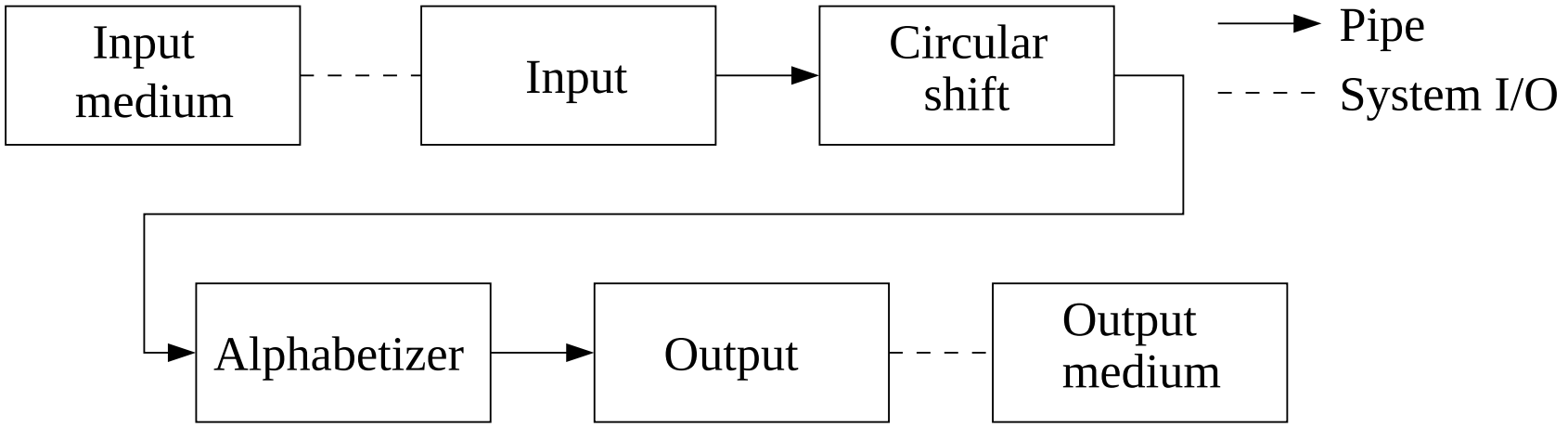


Solution 3: Assessment

+ functional enhancements easy: register additional modules
+ computations insulated from changes in data representation
+ supports reuse since modules only rely on externally triggered events
– processing order difficult to control
– requires more space than previous decompositions

Solution 4: Pipes and Filters

- four filters: input, shift, alphabetize, output
- distributed control
- data sharing limited to pipes



Solution 4: Assessment

+ intuitive flow of processing
+ supports reuse: each filter usable in isolation
+ supports enhancements: new filters are easily incorporated
+ amenable to modification: each filter is independent of the others
– impossible to support an interactive system
– inefficient use of space
– overhead for parsing and unparsing data

Summary

	Shared Data	Abstract Data Type	Implicit Invocation	Pipe and Filter
Change in Algorithm	–	–	+	+
Change in Data Rep	–	+	–	–
Change in Function	+	–	+	+
Performance	+	o	–	–
Reuse	–	+	+	+

5.2.3 Instrumentation Software

- task: develop reusable system architecture for oscilloscopes
- an oscilloscope . . .
  - samples electrical signals (at a rate of up to 2.5 GHz) and displays pictures of them on a screen
  - performs measurements on the signals and displays them

Ingredients of an Oscilloscope

- a very fast AD converter
- quite complex software
  - many different measurements
  - internal storage
  - network interfaces
- sophisticated user interface (touch screen, etc)

Design Requirements

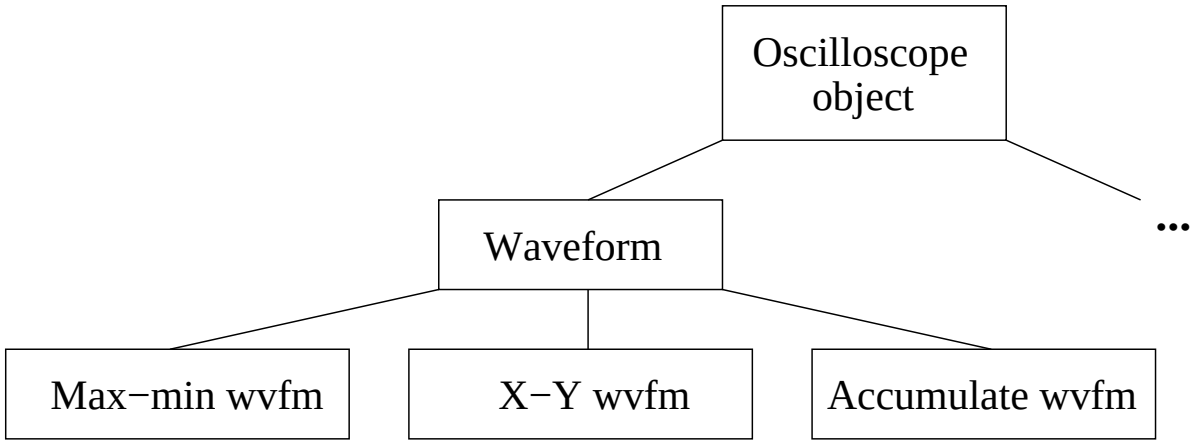
- Reuse
  - rapid changes in user interface and hardware
  - specialized markets
- Performance
  - switching between different configurations

$\Rightarrow$ domain specific software architecture

The design process considered several models.

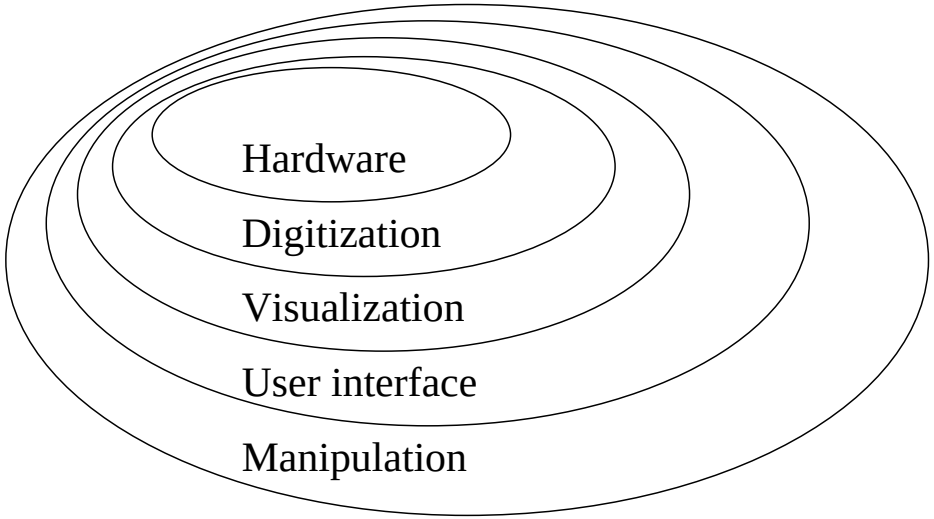
An Object-Oriented Model

- OO model clarified data types: waveforms, signals, measurements, trigger modes, ...
- did not explain how the types fit together
- result: confusion about partitioning of functionality



A Layered Model

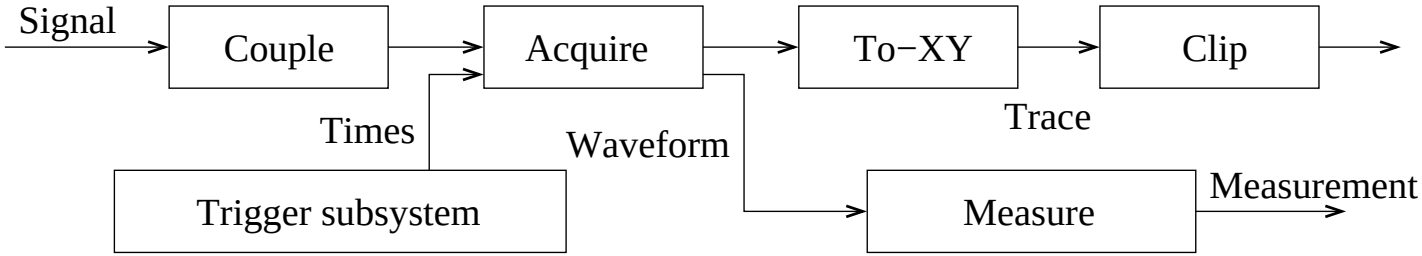
- Layers for: signal manipulation (hardware), waveform acquisition (digitization), waveform manipulation (measurement, addition, FT), display functions, user interaction



- intuitively appealing, but wrong for this domain!
- abstraction boundaries conflict with needs for interaction, *e.g.*, user interaction at all levels

A Pipe-and-Filter Model

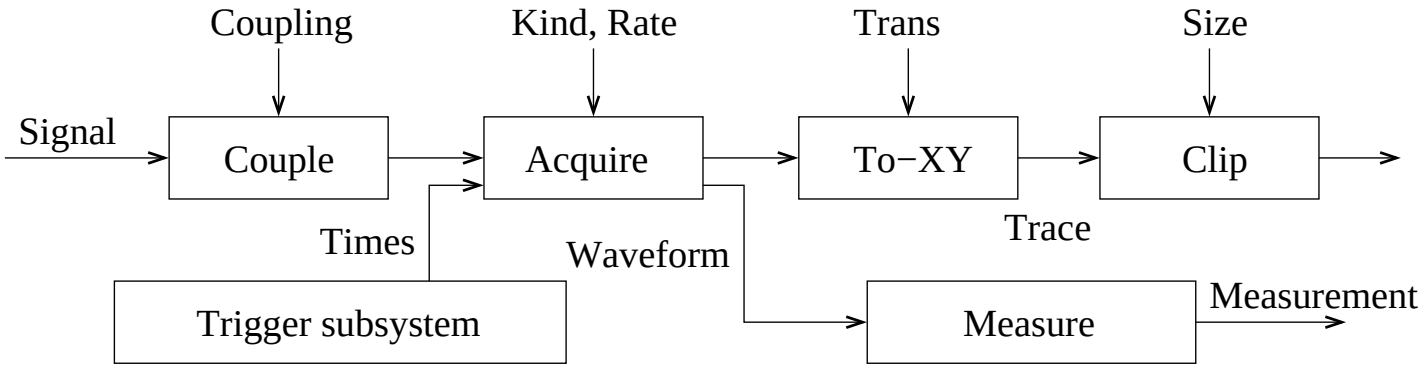
- oscilloscope functions as incremental transformers of data:
  signal transformers, acquisition transformers, display transformers



- significant improvement
- problem: user interaction

A Modified Pipe-and-Filter Model

- each filter has an external control interface



- solves user interface problem
  - what aspects can be modified dynamically by the user
  - decouples signal processing from user interface

Further Refinement

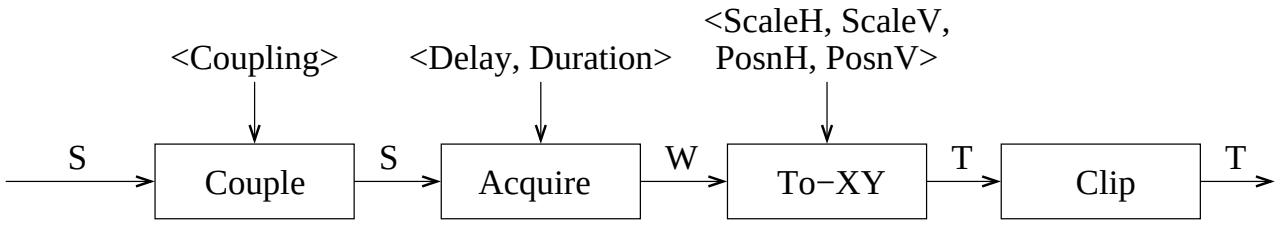
- Pipe-and-filter led to poor performance
  - copying of waveforms impractical
  - synchronization of fast filters with slow filters unacceptable
- Solution: colored pipes
  - without copying
  - ignore incoming data (protect slow filters)
- increased stylistic vocabulary allowed better design

5.2.4 Formally Specified Architecture

- Up to now: informal descritions of architecture
- programming languages not appropriate for describing connectors

$\Rightarrow$ formal specification language (Z) can help

Overview and Datatypes



- Signals (S): inputs to the oscilloscope
- Waveforms (W): internally stored data
- Traces (T): pictures shown on screen
- functions of time, voltage, screen coordinates

$$Signal == AbsTime \rightarrow Volts$$
$$Waveform == AbsTime \nrightarrow Volts$$
$$Trace == Horiz \nrightarrow Vert$$

Component Couple

- purpose: subtract DC offset from signal
- user configuration: choice of kind of input signal

$$Coupling ::= DC \mid AC \mid GND$$

- component modeled as a function that maps a user configuration into a signal transformer

$$Couple : Coupling \to Signal \to Signal$$

$$\begin{aligned}
&\forall\, s : Signal \bullet \\
&\quad Couple\ DC\ s = s \\
&\quad Couple\ AC\ s = \lambda\, t : AbsTime \bullet s(t) - dc(s) \\
&\quad Couple\ GND\ s = \lambda\, t : AbsTime \bullet 0
\end{aligned}$$

Component Acquire

- purpose: extract a time slice from a signal
- user configuration: *delay* and *duration*
- further input: trigger event

$$TriggerEvent == AbsTime$$

- waveform is only defined for a time interval of length *duration* starting *delay* units after a trigger event

$$Acquire : RelTime \times RelTime \rightarrow TriggerEvent \rightarrow Signal \rightarrow Waveform$$

$$\forall\, delay, dur : RelTime, trig : TriggerEvent, s : Signal \bullet$$
$$Acquire\ (delay, dur)\ trig\ s\ =$$
$$\{t : AbsTime \mid trig + delay \leq t \leq trig + delay + dur\} \lhd s$$

Packaging the user configuration

$$
\begin{array}{|l}
\hline
ChannelParameters \\
c : Coupling \\
delay, dur : RelTime \\
scaleH : RelTime \\
scaleV : Volts \\
posnH : Horiz \\
posnV : Vert \\
\hline
\end{array}
$$

assuming the remaining components as

$$
\begin{aligned}
&WaveformToTrace : RelTime \times Volts \times Horiz \times Vert \\
&\qquad \rightarrow Waveform \rightarrow Trace \\
&Clip : Trace \rightarrow Trace
\end{aligned}
$$

System Description

$$
\begin{array}{l}
ChannelConfiguration : ChannelParameters \to TriggerEvent \\
\qquad \to Signal \to Trace \\
\hline
\forall\, p : ChannelParameters \bullet \\
\quad ChannelConfiguration\ p = \lambda\, trig : TriggerEvent \bullet \\
\qquad Clip \circ \\
\qquad WaveformToTrace(p.scaleH, p.scaleV, p.posnH, p.posnV) \circ \\
\qquad Acquire(p.delay, p.dur) \circ \\
\qquad Couple\ p.c
\end{array}
$$

What have we gained?

- precise characterization of system
- component = parameterized data transformer
- data sharing only via connections
- external parameters must be available