1. Software Architecture

Software Architecture and Design

Architectural Styles

Harald Gall
http://seal.ifi.uzh.ch/ase

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Overview

Architectural Design

Architectural Styles and Patterns

Documenting Software Architecture

Standard Architectures

Software Architecture

Piazza del Campidoglio, Rome

Sears Tower, Chicago

Torii of Itsukushima, Japan

The Sydney Opera House

Facts and Figures:

Was designed by Danish architect Jørn Utzon

Was opened by Queen Elizabeth II on 20 October 1973

Presented, as its first performance, The Australian Opera's production of War and Peace by Prokofiev

Cost $AU 102.000.000 to build

Conducts 3.000 events each year

Includes 1.000 rooms

- Is 185 metres long and 120 metres wide
- Has 2.194 pre-cast concrete sections as its roof
- Has roof sections weighing up to 15 tons
- Has roof sections held together by 350 kms of tensioned steel cable
- Has over 1 million tiles on the roof
- Uses 6.225 square metres of glass and 645 kilometres of electric cable

Architecting ...

„Architecting, the planning and building of structures, is as old as human societies – and as modern as the exploration of the solar system.“

by Eberhardt Rechtin, 1991

Motivation

If the size and complexity of a software system increase,
the **global structure of the system** becomes more important than the selection of specific algorithms and data structures.

Goals

A framework to support the development of software

Integration platform for future enhancements

Interface definition for collaboration of components

Basic elements of a software architecture

Components

Connectors

Constraints

Rationale

Components

Decomposition of a system (multi-version, multi-person)

Criteria for component decomposition

- Modularization, encapsulation, information hiding, abstraction
- Functions as components (functional decomposition)
- Distribution/Parallelism
- Optimization of performance (e.g. distribution onto parallel processors)

What is a Software Connector?

by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy

Architectural element that models

- Interactions among components
- Rules that govern those interactions

Simple interactions

- Procedure calls
- Shared variable access

Complex & semantically rich interactions

- Client-server protocols
- Database access protocols
- Asynchronous event multicast

Each connector provides

- Interaction duct(s)
- Transfer of control and/or data

Why treating Connectors independently?

by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy

Connector ≠ Component

- Components provide application-specific functionality
- Connectors provide application-independent interaction mechanisms

Interaction abstraction and/or parameterization

Specification of complex interactions

- Binary vs. N-ary
- Asymmetric vs. Symmetric
- Interaction protocols

Software Connector Roles

by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy

Locus of interaction among set of components

Protocol specification (sometimes implicit) that defines its properties

- Types of interfaces it is able to mediate
- Assurances about interaction properties
- Rules about interaction ordering
- Interaction commitments (e.g., performance)

Roles

- Communication
- Coordination
- Conversion
- Facilitation

Connectors as Communicators

by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy

Main role associated that supports

- Different communication mechanisms
  - e.g. procedure call, RPC, shared data access, message passing
- Constraints on communication structure/direction
  - e.g. pipes
- Constraints on quality of service
  - e.g. persistence

Separates communication from computation

May influence non-functional system characteristics

- e.g. performance, scalability, security

Connectors as Coordinators

by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy

Determine computation control

Control delivery of data

Separates control from computation

Orthogonal to communication, conversion, and facilitation

- Elements of control are in communication, conversion and facilitation

Connectors as Converters

by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy

Enable interaction of independently developed, mismatched components

Mismatches based on interaction

- Type
- Number
- Frequency
- Order

Examples of converters

- Adaptors
- Wrappers

Connectors as Facilitators

by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy

Enable interaction of components intended to interoperate

- Mediate and streamline interaction

Govern access to shared information

Ensure proper performance profiles

- e.g., load balancing

Provide synchronization mechanisms

- Critical sections
- Monitors

Connector Types

Procedure call

Data access

Event

Stream

Linkage

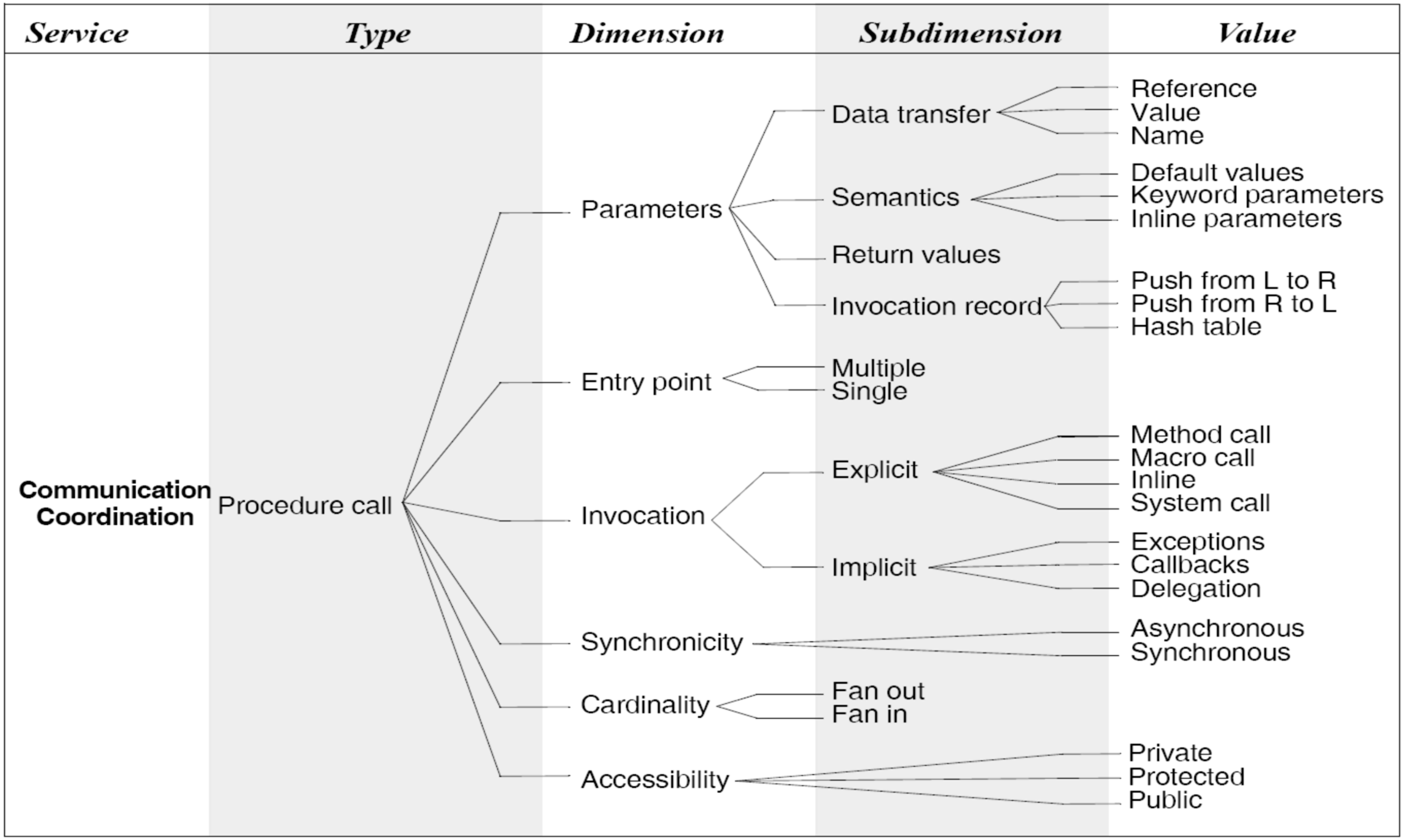
Distributor

Arbitrator

Adaptor

Procedure Call Connectors

by Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy



Service	*Type*	*Dimension*	*Subdimension*	*Value*
Communication Coordination	Procedure call	Parameters	Data transfer	Reference Value Name
			Semantics	Default values Keyword parameters Inline parameters
			Return values	
			Invocation record	Push from L to R Push from R to L Hash table
		Entry point	Multiple Single	
		Invocation	Explicit	Method call Macro call Inline System call
			Implicit	Exceptions Callbacks Delegation
		Synchronicity		Asynchronous Synchronous
		Cardinality	Fan out Fan in	
		Accessibility		Private Protected Public

Constraints

Components must be constrained to provide that

- the required functionality is achieved
- no functionality is duplicated
- the required performance is achieved
- the requirements are met
- modularity is realized (e.g. which modules interact with the operating system)

Assignment of functionality

Rationale (why?)

For multi-version software its design rationales must be documented:

- Decomposition into components
- Connections between components
- Constraints on components and connections

Serves as plan for future enhancements

Serves as support/aid for maintainers

Commitment of the architecture

As one of the **early design decisions** it is difficult to change the architecture

The **organization** of the project is influenced substantially:

- teams, documentation, configuration, management, maintenance, integration and tests

The architecture must **not prevent** a beneficial implementation

Impact onto the life-cycle

The architecture substantially impacts performance and available system resources

The architecture determines the simplicity of future changes and adaptations

A successful architecture can be used to build similar systems:

- “product family” and
- “domain specific software architectures”

Requirements and Software Architecture

Fulfillment of functional requirements

- Input/Output behavior

Fulfillment of desired performance

- Timing, preciseness, stability
- Memory workload, other resources

Can be verified by observation of the running system

Non-functional Requirements

Software architectures must also fulfill the following requirements:

- Adaptability
- Flexibility
- Portability
- Interoperability
- Reusability within “related” projects

Static and dynamic structures

Module structure

- for configuration, non-existent at run-time

Distribution structure

- at run-time

Dynamic structures influence

- non-functional Requirements
- functional Requirements and system performance

... and their connection to the running system

VIEWS OF A SOFTWARE ARCHITECTURE

Views of a software architecture

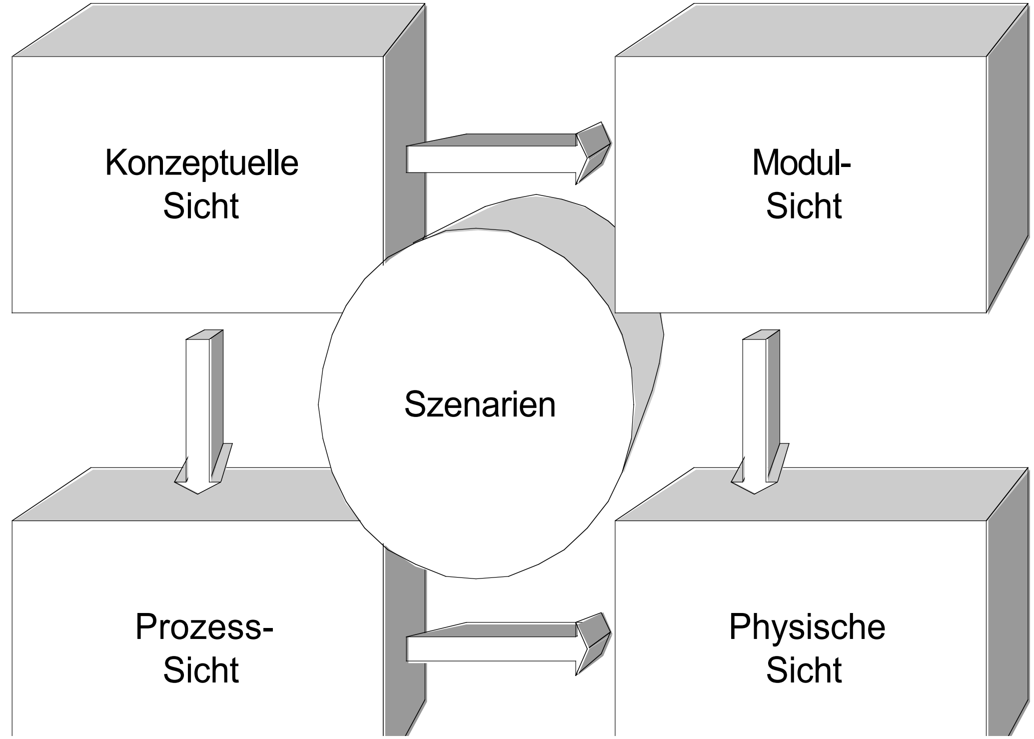
Problem

- ambiguous diagrams
- overloaded diagrams

Solution/approach

- Different perspectives
- Connection between single views

4+1 Views


Konzeptuelle Sicht
Modul-Sicht
Szenarien
Prozess-Sicht
Physische Sicht

Conceptual (Logical) View

Functional requirements

Orientation on problem domain

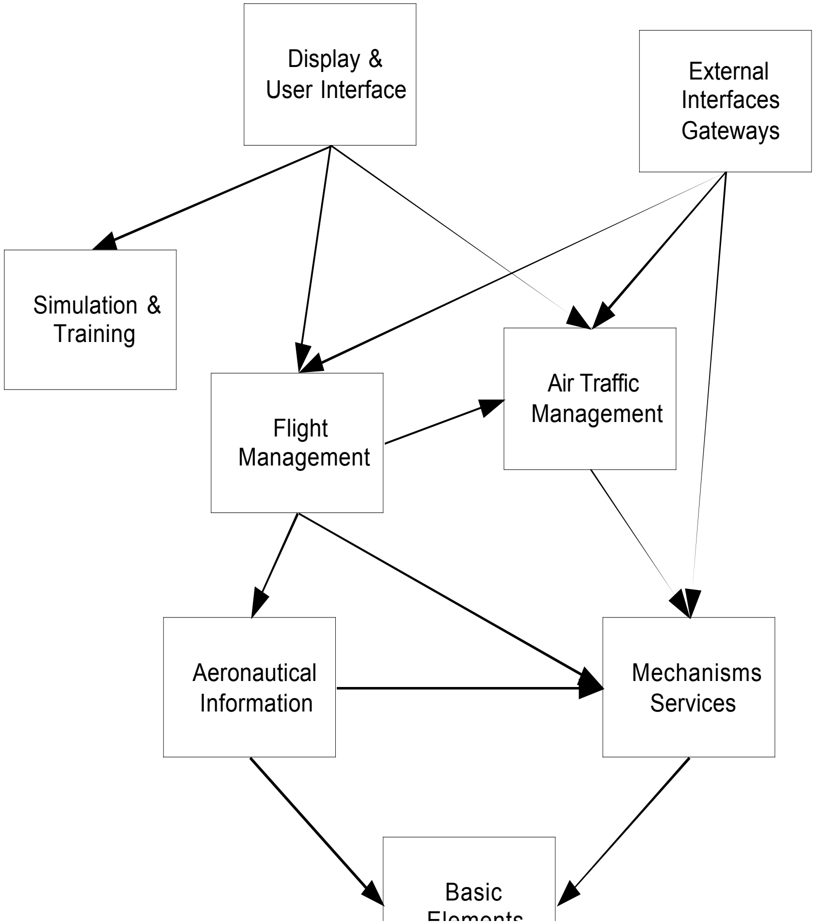
Communication with experts

Independence of implementation decisions

“Frameworks”

Conceptual view: example

Air traffic management system



Module (Development-) View

Organization of modules

- Subsystems
- Coherent parts in the development
- Allocation of effort (development, maintenance)

Organization in hierarchical layers

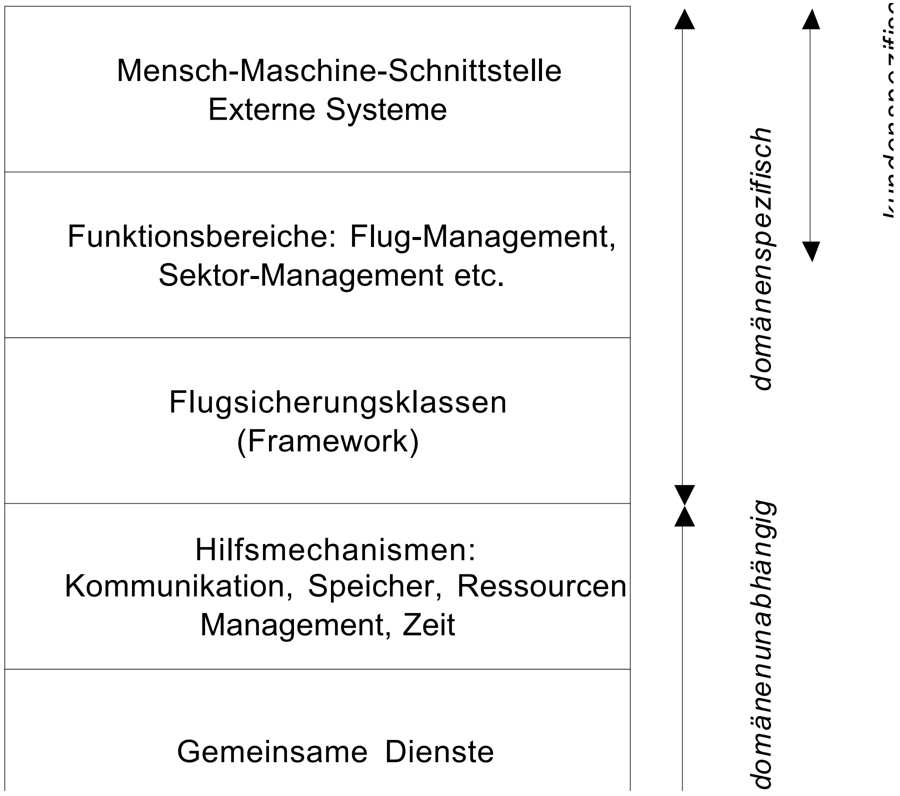
- OSI communication protocols

Compile-time structure

- marginal for the operation of the system

Module view: example

Air traffic management system



Mensch-Maschine-Schnittstelle
Externe Systeme

Funktionsbereiche: Flug-Management,
Sektor-Management etc.

Flugsicherungsklassen
(Framework)

Hilfsmechanismen:
Kommunikation, Speicher, Ressourcen
Management, Zeit

Gemeinsame Dienste

domänenspezifisch

domänenunabhängig

kundenspezifisc

Process and Coordination View

Dynamic aspects of the run-time processes

- Process creation
- Synchronization
- Concurrency

Components of this view are processes: instructions and separate execution logic

At run-time different reconfigurations can be done

Estimates for process allocation etc.

Physical View

Mapping of software on existing/available hardware

- e.g. distribution of computations in a distributed system

Impact on

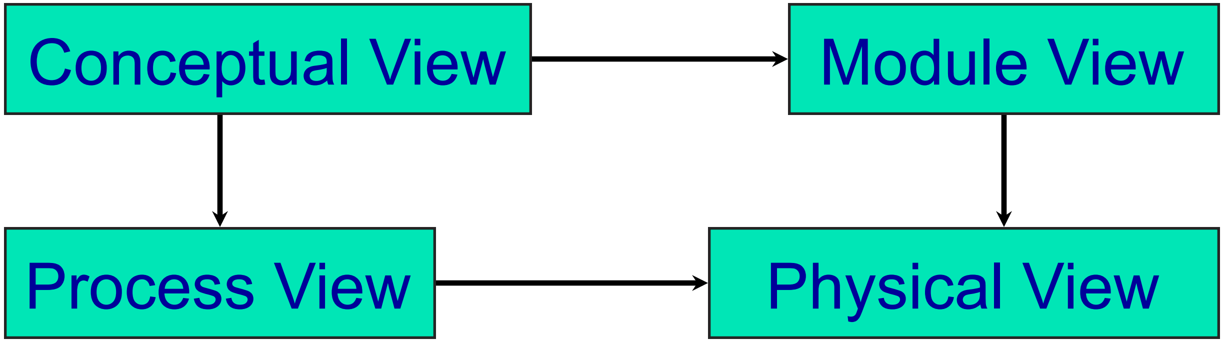
- availability, reliability, performance and scalability

Structuring should have little or no influence on the implementation of the components

Integration of Views

End-user
- Functionality

Programmer
- Software management



Conceptual View

Module View

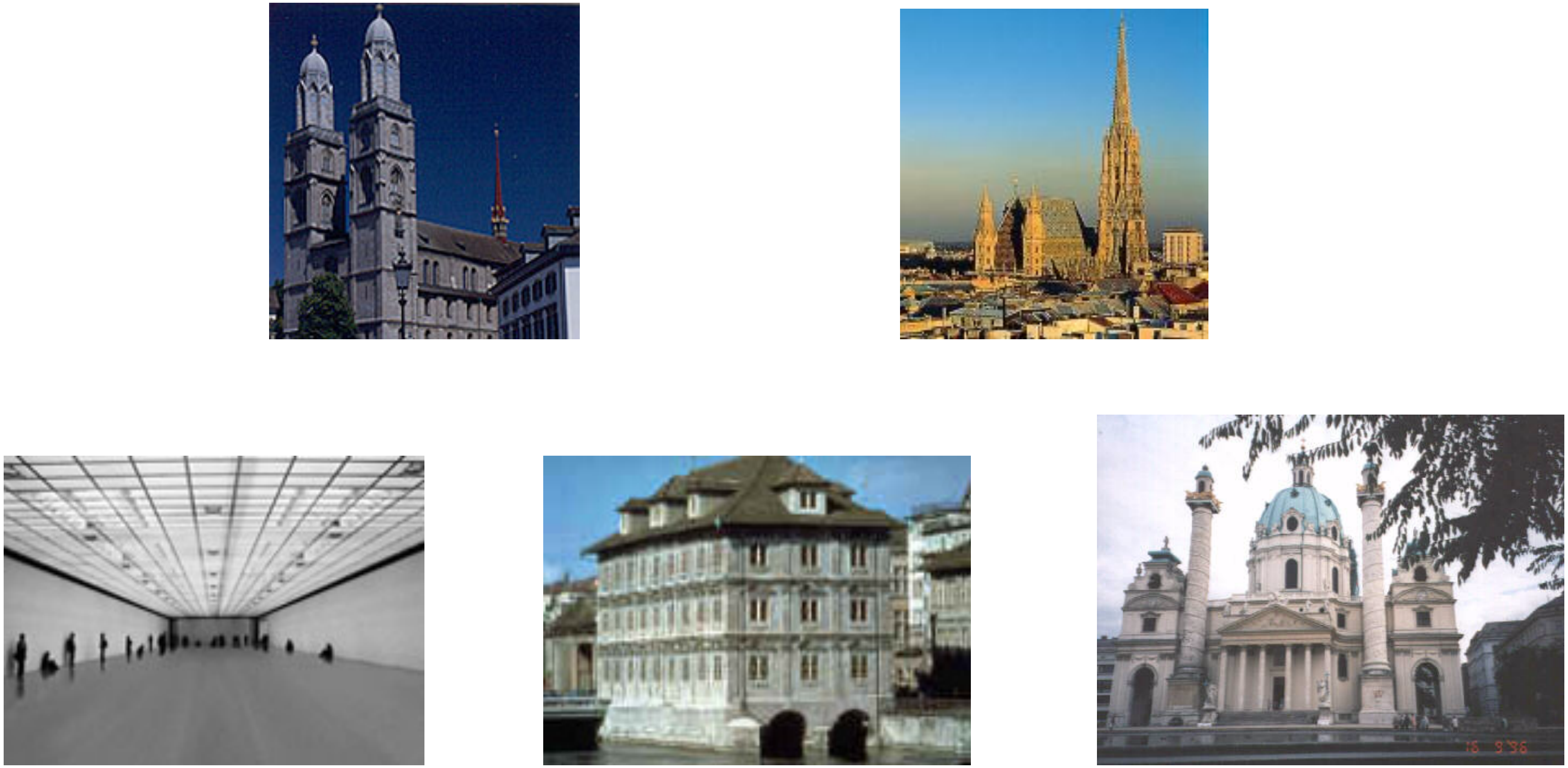
Process View

Physical View

System Integrator
- Performance
- Scalability
- Throughput

System Developer
- System Topology
- Completion
- Installation

Software Architecture Styles



What is an Architectural Style?

A Design Language for a class (family) of systems

- Vocabulary for design elements, e.g., pipes, filters, server, parser, DBs
- Design rules and design constraints, e.g., <100 clients per server per time unit
- Semantic interpretation of architectural elements
- Analysis for checking conformance of an architectural design, e.g., deadlock detection, schedulability analysis

Definition of Architectural Style

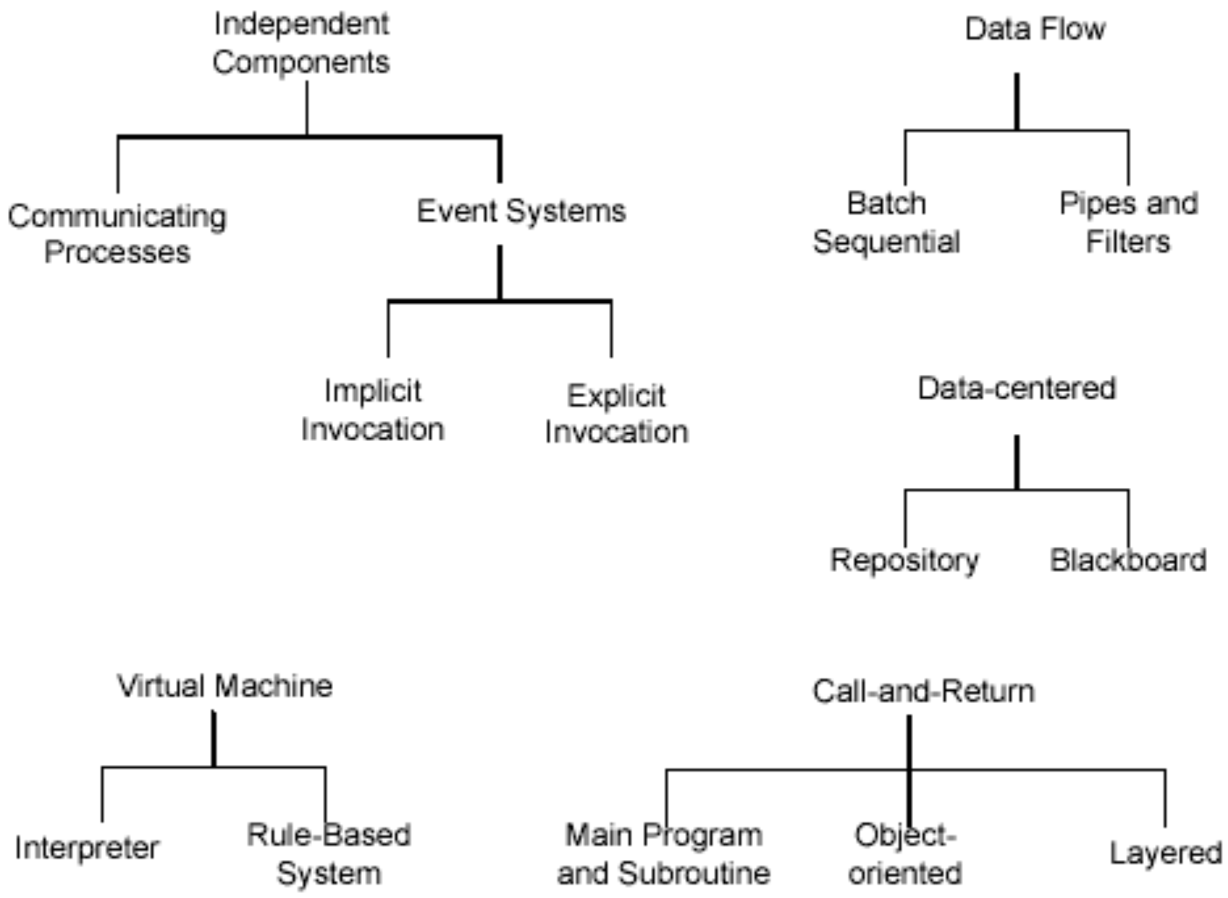
An architectural style defines a family of software systems and their structural organization

It defines components, connectors, and configurations as well as constraints for their application in concrete applications

It also defines design rules and constraints for developing instances of a software system

[Perry u. Wolf 1992]

Catalogue of Architectural Styles



Software Architecture Styles

Dataflow systems

- Batch sequential
- Pipes and filters

Call-and-return systems

- Main program and subroutine
- Hierarchical layers
- OO systems

Independent components

- Communicating processes
- Event systems

Virtual machines

- Interpreters
- Rule-based systems

Data centered systems

- Databases
- Hypertext systems
- Blackboards

PIPES AND FILTERS

Pipes & Filters

Filters are the components

- Read an input data stream and transform it into an output data stream

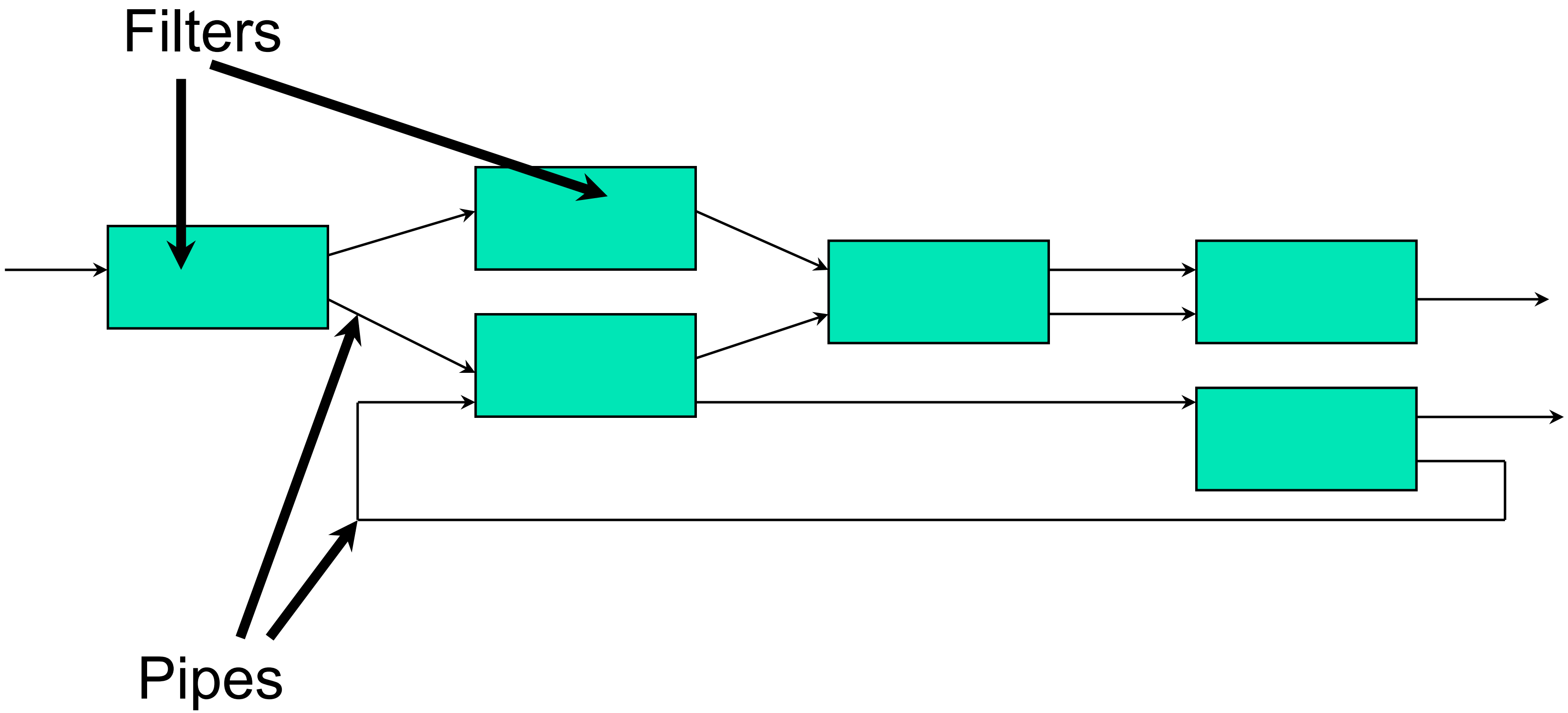
Pipes are the connectors

- Provide the output of a filter as input to another filter

Example:

- Unix shell: piping of components (commands) via "|“
- cat {myfile} | grep “architecture” | sort … | more

Pipes & Filters System Design



Pipes & Filters

Filters are independent components that

- do not share status with other components
- do not know the identity of their neighbors (input/output)

Pipelines

- Constrain the topology to a linear configuration of filters

Bounded Pipes

- Constrain the amount of data that a pipe can store temporarily

Typed Pipes

- Constraint the type of data stream that a pipe must have

Advantages

A designer can define the input/output behavior of the whole system as combination of single filters

Simple; no complex component interactions

Filters as black-box and, therefore, substitutable

Reusability

- Two filters can be arranged arbitrarily, as long as they support the same data format / stream

Advantages

Maintenance

- Integration of new filters
- Substitution of existing/integrated filters

Hierarchical structures are easy to compose

Analysis of

- Throughput and potential deadlocks

Concurrent execution

- Filters are synchronized by the data transfer

Disadvantages

Batch processing characteristic but not apt for interactive applications

Handling of independent data streams

Filter require a common data format

Parsing/Unparsing: if the data stream is analyzed by tokens, every filter has to parse and unparse the data separately

Filter memory: if a filter has to fully parse, e.g. a file, before computation, memory requirements arise (buffering)

Process overhead: if each filter is run in a separate process, this requires processing overhead

LAYERED ARCHITECTURE

Layered Architecture

Hierarchically organized system

- Layers are components
- Interfaces and protocols are the connectors

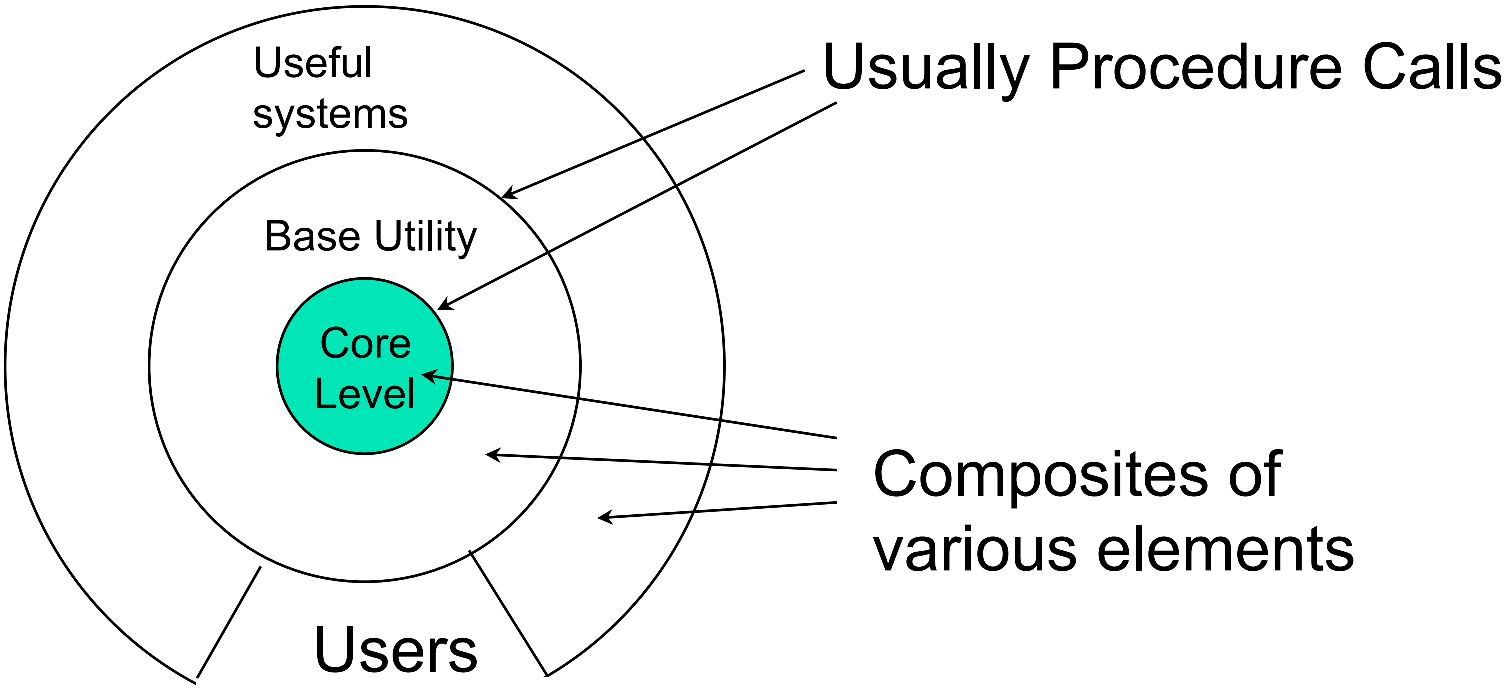
Abstraction:

- Each layer represents and implements an abstract virtual machine

Architectural constraints:

- Each layer can only interact with the directly connected upper and lower layers

Layered Systems Design



Advantages

Support of abstraction levels by layering

- A larger problem is decomposed into several smaller ones

Changes in one layer affect at most the two neighboring layers (interface, protocol)

Reusability

- Standard interfaces can be reused often
- Different implementations of the same layer and their substitution

Disadvantages

Not all systems can be decomposed into layers

- check for violations of architectural constraints (communication direction and protocols, dependencies)

Communication between neighbors

- Sometimes communication between non-neighboring layers can be necessary
- Skipping of several layers can cause difficulties

Comprehension

- Abstractions of some layers can be difficult to comprehend

OBJECT-ORIENTED ARCHITECTURE

Object-oriented Organization

Data abstraction and information hiding

Encapsulation of data and corresponding operations

- Attributes and methods

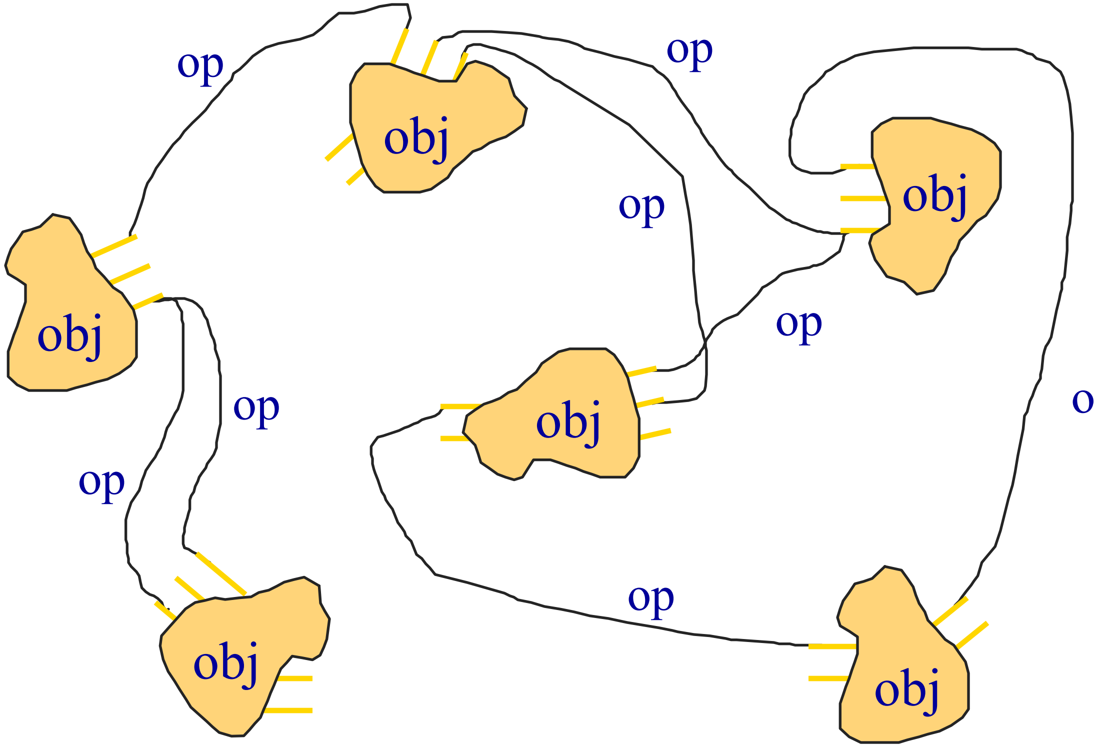
Objects ensure consistency of their data

- Objects are self dependent for their integrity (invariant)
- Internal representation of data is hidden
  (no direct manipulation of data)

Objects can have different interfaces (role and client dependent)

Objects

O-O



Advantages

O-O

Hiding of implementation, only the interface is visible for the client

Changes to one object do not affect other objects (as long as the interface remains unchanged)

Objects are a good design tool

- Data and access operations are put together

Disadvantages

O-O

To communicate an object has to know the identity of the other object

If IDs change all “clients” must be adapted accordingly

Side-effects and mutual influence in case of concurrent object access

EVENT-BASED SYSTEMS

Event-based Systems

Functions are not executed through a direct procedure/method call

Publisher

- Components raise an event (publisher)

Subscriber

- Other "interested" components (subscribers) are notified and react accordingly

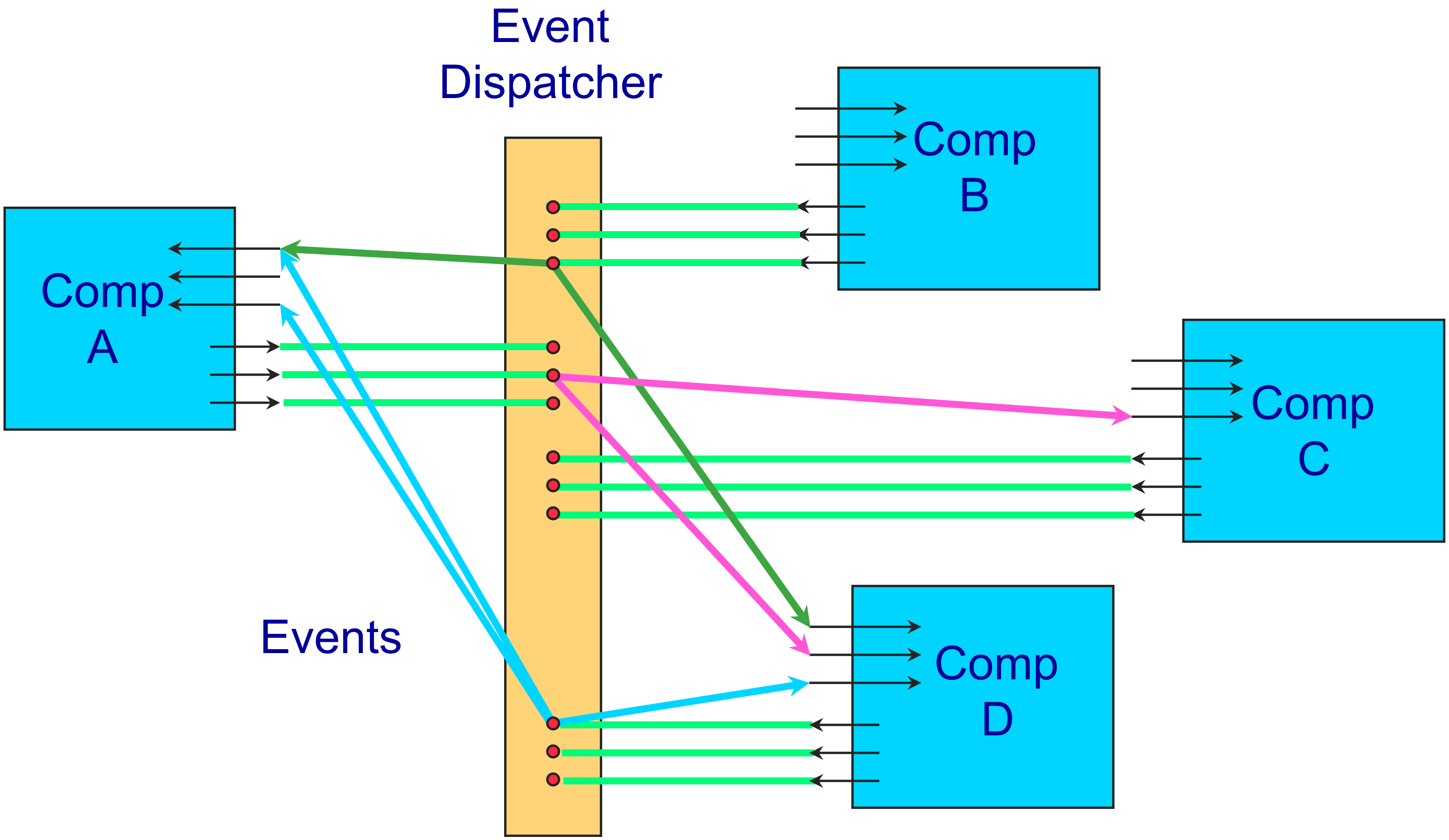
Event Dispatcher

- Distributes published event to the subscribers

Relation of events and event handling is unknown to the components

Components

event-based



Advantages

event-based

Extensibility and Reusability

- A new component can be easily integrated into the system
- Subsequent registration for other events and announcement of its own events

Exchangeability of components

- Without influence on the interfaces of other components

Disadvantages

event-based

If an event is published, it is not assured that it is being handled by others

- processing sequence

Data exchange other than with events is problematic

Behavior of components is tightly coupled with the execution environment (e.g. event model)

SHARED DATA

Shared Data

Two kinds of components:

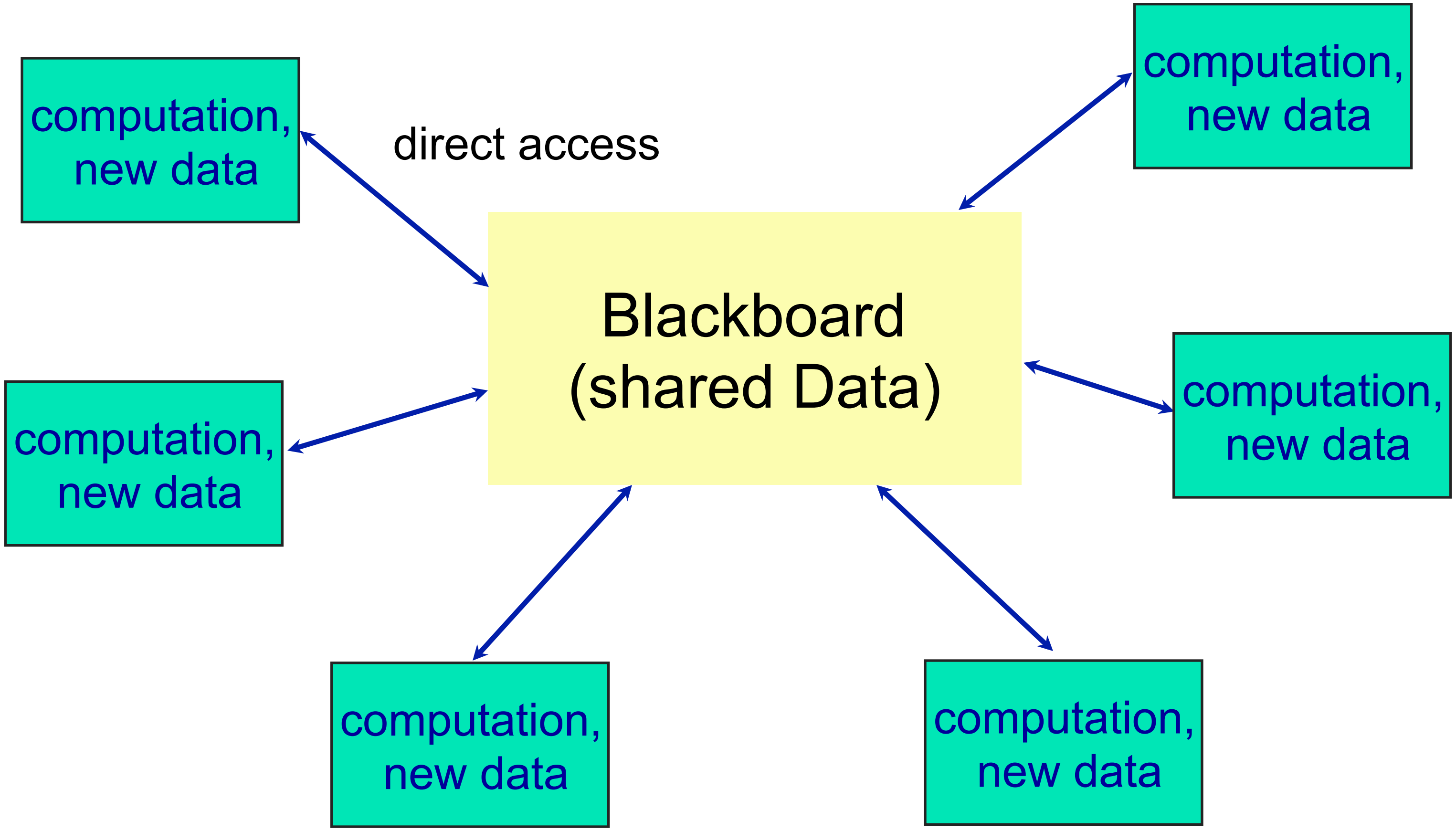
- Central data management
- Independent components for computation

Activation of computation

- When inserting (storing) new data (database trigger)
- Trough the actual state

Blackboard

Shared Data



Pros/Cons

Control can be realized in different parts of the architecture

This style can also be used to model batch processing with a shared database

Software Architecture

System decomposition and Modular Structure

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

System Decomposition

Modularization as mechanism for improving the flexibility and comprehensibility of a system

modular programming

- write one module with little knowledge of the code in another module
- reassemble and replace modules without reassembly of the whole system

especially important for large systems!

Benefits of modular programming

managerial

- shorten development time because separate groups would work on each module with little need for communication

product flexibility

- changes to one module without a need to change others

comprehensibility

- study the system one module at a time

What is Modularization?

“module” is considered a responsibility assignment rather than a subprogram.

modularizations include the design decisions that must be made before the work on independent modules can begin

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.

KWIC example

used in D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", CACM, 1972.

Text:

Wikipedia, The Free Encyclopedia

KWIC is an acronym for Key Word In Context, the most common format for concordance lines.

→

KWIC is an acronym for Key Word In Context, ...	page 1
... Key Word In Context, the most common format for concordance lines.	page 1
... the most common format for concordance lines.	page 1
... is an acronym for Key Word In Context, the most common format ...	page 1
Wikipedia, The Free Encyclopedia	page 0
... In Context, the most common format for concordance lines.	page 1
Wikipedia, The Free Encyclopedia	page 0
KWIC is an acronym for Key Word In Context, the most ...	page 1
KWIC is an acronym for Key Word ...	page 1
... common format for concordance lines.	page 1
... for Key Word In Context, the most common format for concordance ...	page 1
Wikipedia, The Free Encyclopedia	page 0
KWIC is an acronym for Key Word In Context, the most common ...	page 1

example taken from Wikipedia.org

Considerations

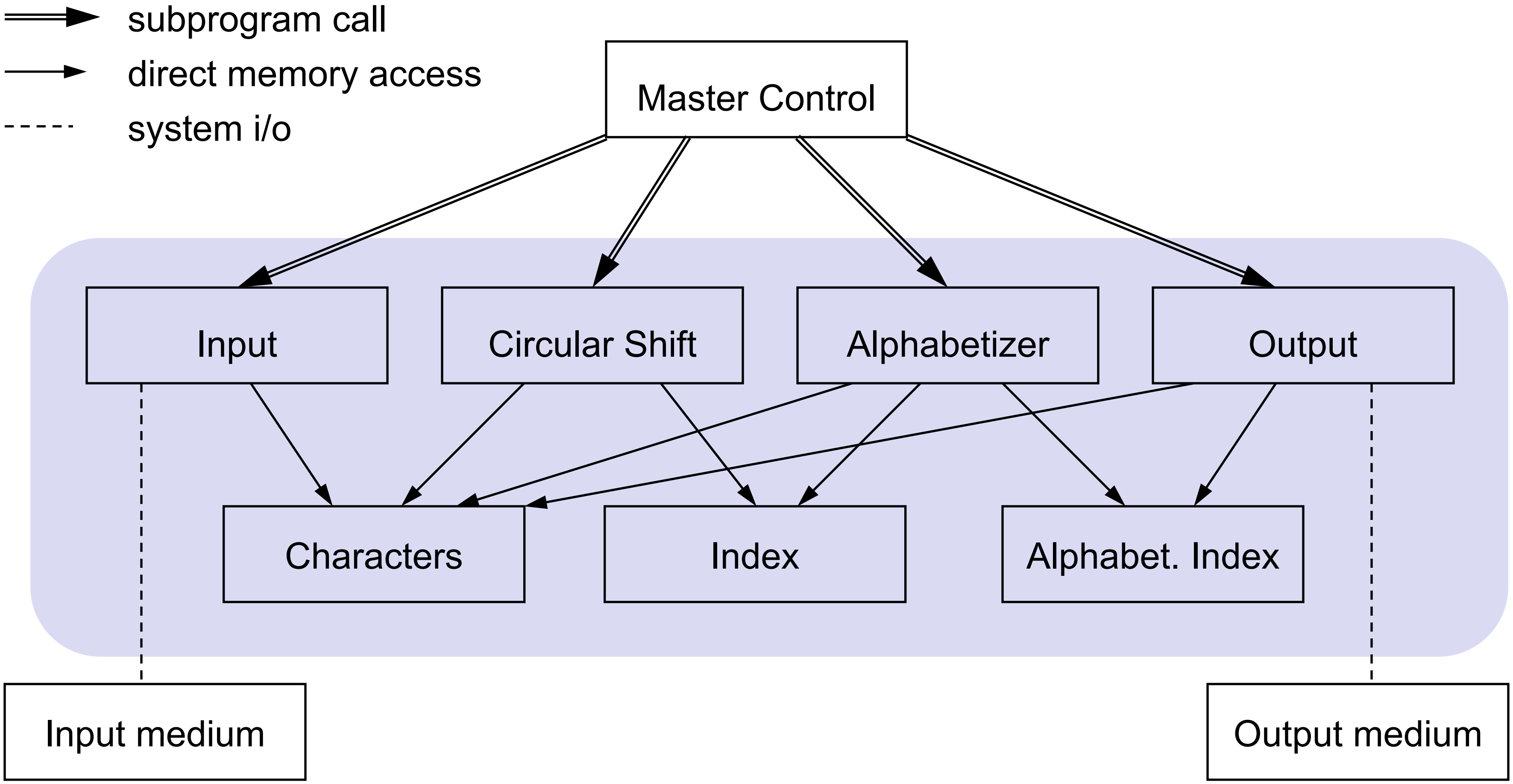
What are the components?

What architectural style shall be used?

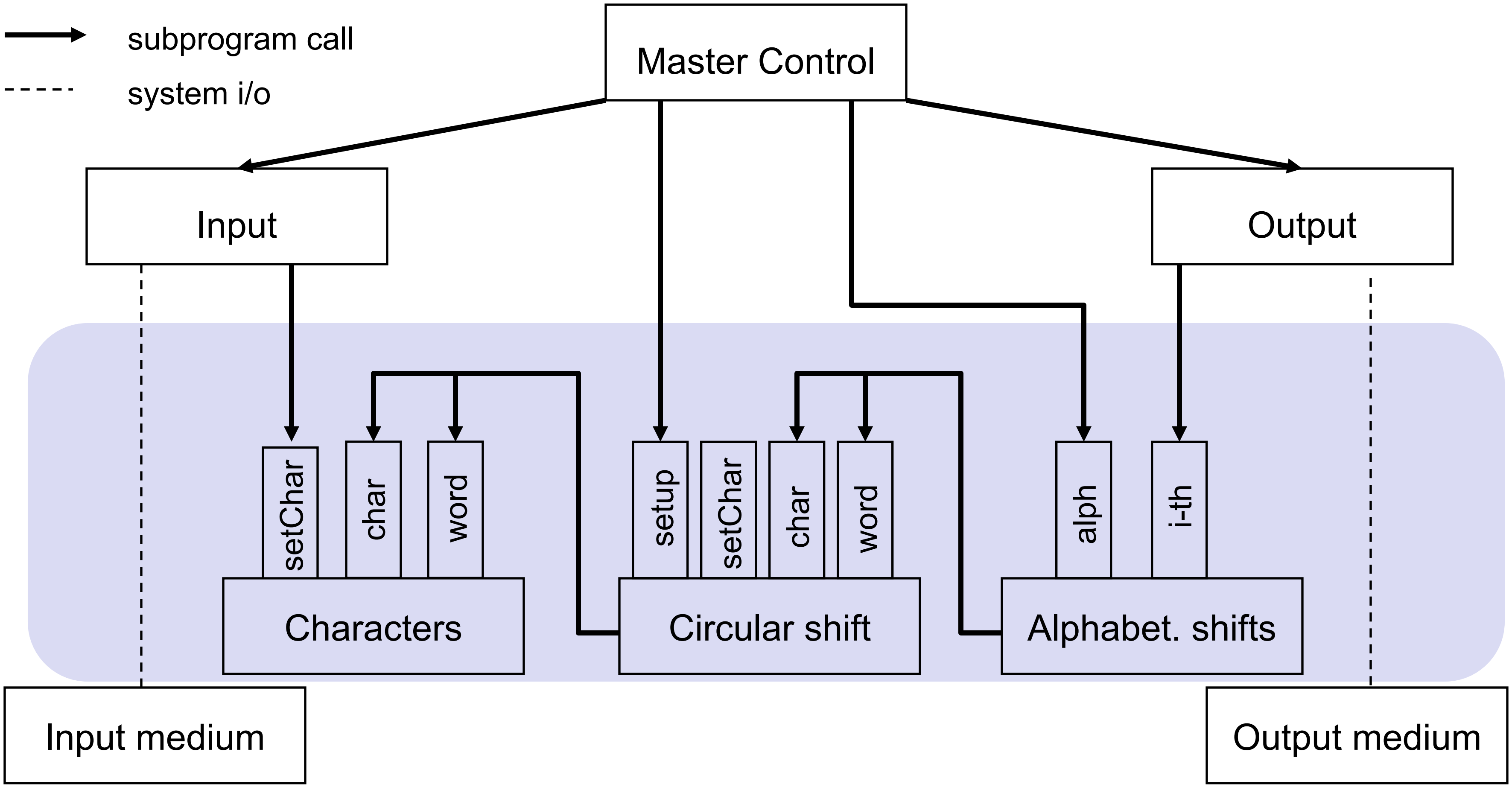
What about principles of encapsulation, changeability, information hiding?

Next we present alternative solutions following different architectural styles.

Solution 1: Main program/subroutine with shared data



Solution 2: Abstract data types (ADTs)



Assessing Changeability /1

Differences are in the way that they are divided into work assignments, and the interfaces between modules.

Algorithms might be identical.

Changeability (design decisions that are likely to change):

- input format
- have all lines stored
- pack the characters four to a word
- make an index for the circular shifts rather than store them as such
- alphabetize the list once (search for item when needed or partially alphabetize)

Assessing Changeability /2

input format

- confined to one module in S1/ShaD & S2/ADT

all lines stored and characters packed

- S1/ShaD: changes in every module!

In S1/ShaD the format of the line storage must be used by all programs

In S2/ADT the exact way that the lines are stored is entirely hidden from all but module 1.

Assessing Changeability /3

index of circular shifts

- S1/ShaD: alphabetizer and output routines affected
- S2/ADT: confined to circular shift

alphabetize once

- S1/ShaD: output module will expect the index to have been completed
- S2/ADT: alphabetizer locally

Assessing Independent development

Interfaces in S1/ShaD

- complex formats and table organizations
- table structure and organization are essential to the efficiency
- complex; joint effort of all development groups

Interfaces in S2/ADT

- more abstract
- consist primarily of function names and the numbers and types of the parameters
- simple decisions and independent development much earlier

Assessing Comprehensibility

in S1/ShaD

- to understand the output module, it is necessary to understand the alphabetizer, the circular shifter, and the input module
- aspects of the tables used by the output module: constraints on the structure of the tables due to algorithms used in other modules
- system is comprehensible only as a whole

not in S2/ADT

Assessing Decomposition

S1/ShaD: make each major step in processing a module

- flowchart approach not sufficient for large systems

S2/ADT: information hiding

- line storage module is used in almost every action
- circular shift might not make any table at all but calculate each character as demanded
- every module is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.

Assessing Hierarchical structure

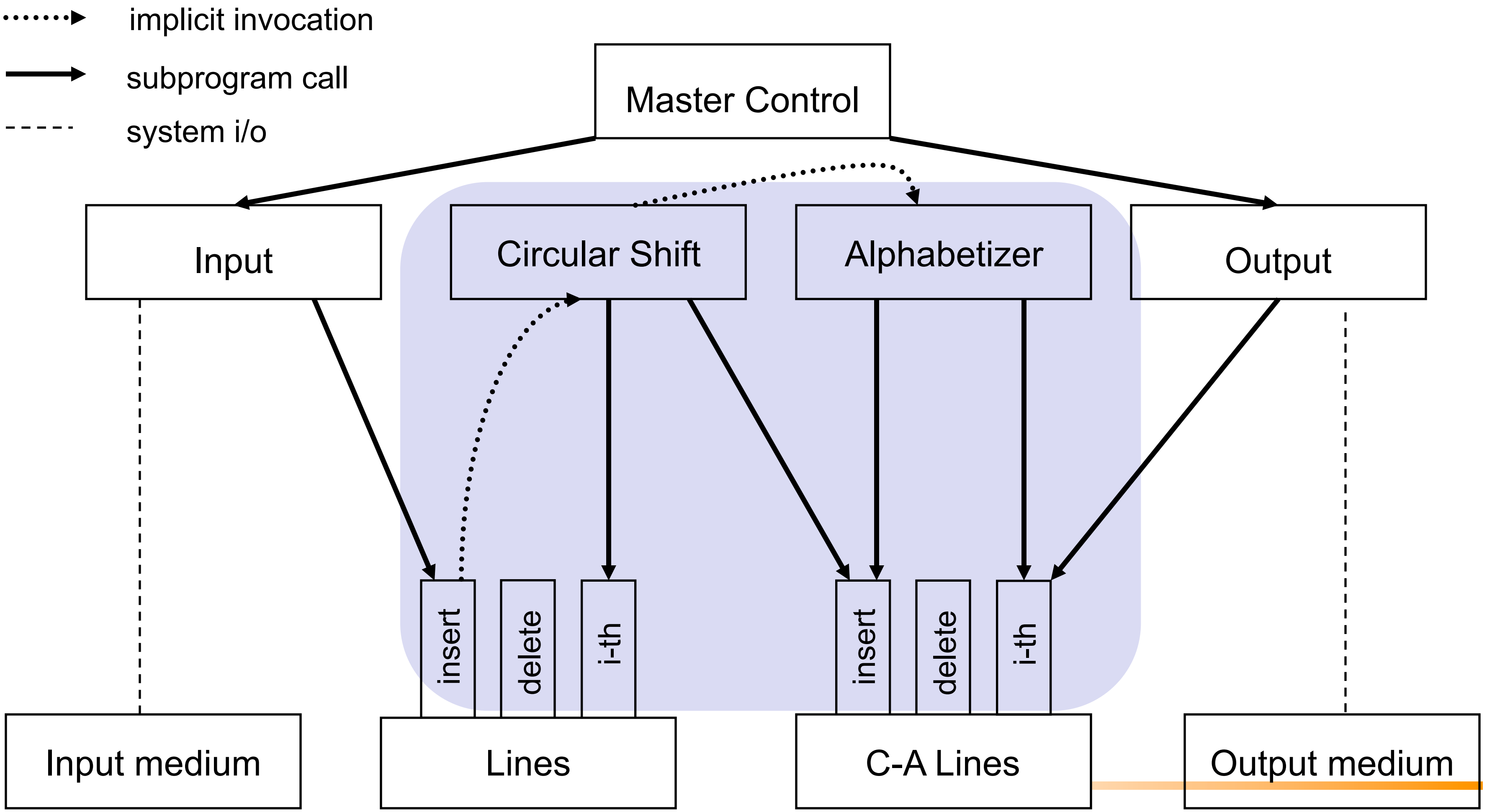
It is easy to confuse the benefits of a good decomposition with those of a hierarchical structure!

Concerned with a partial ordering relation "uses" or "depends"

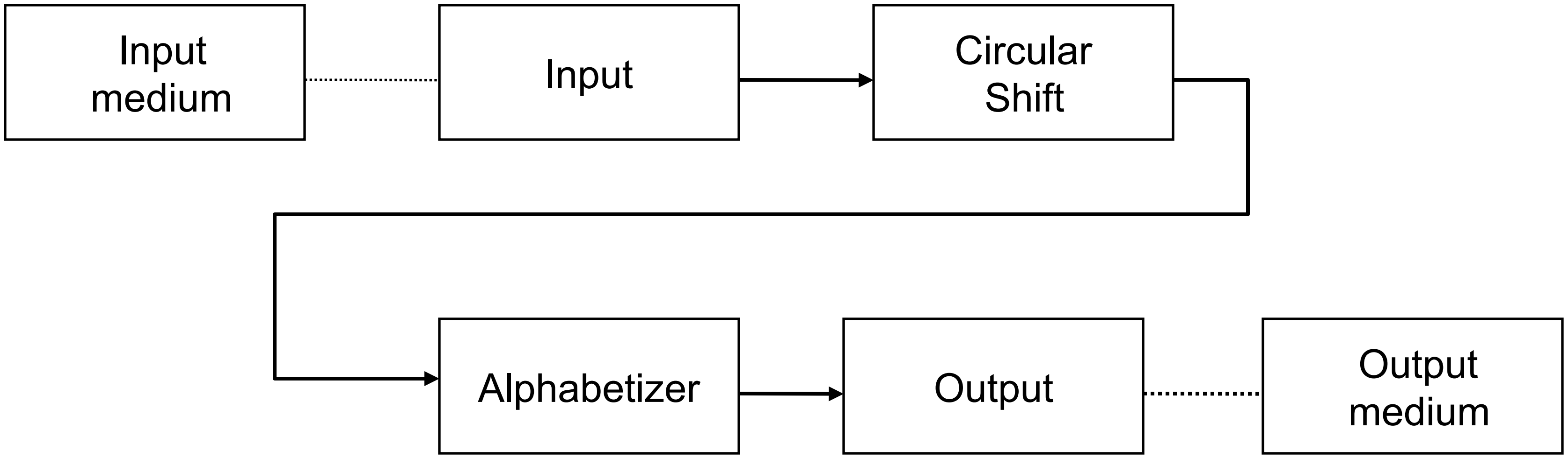
- parts of the system are simplified because they use services of the lower levels
- able to cut off the upper levels and still have a usable and useful product (e.g. symbol table)
- start "new tree on the old trunk"

Start with a list of difficult design decisions or design decisions that are likely to change!

Solution 3: Implicit invocation



Solution 4: Pipe-and-Filters



KWIC: Comparisons of solutions

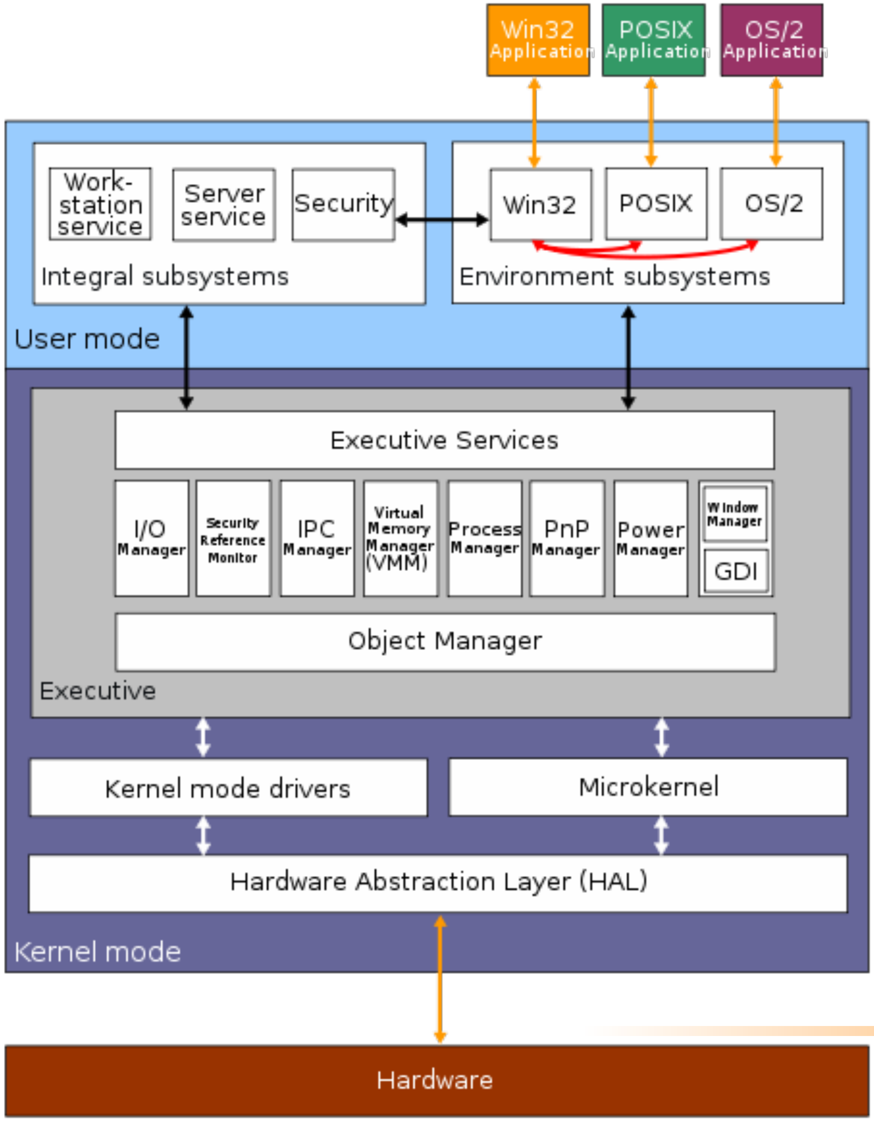
Next time

Patterns of Software Architecture

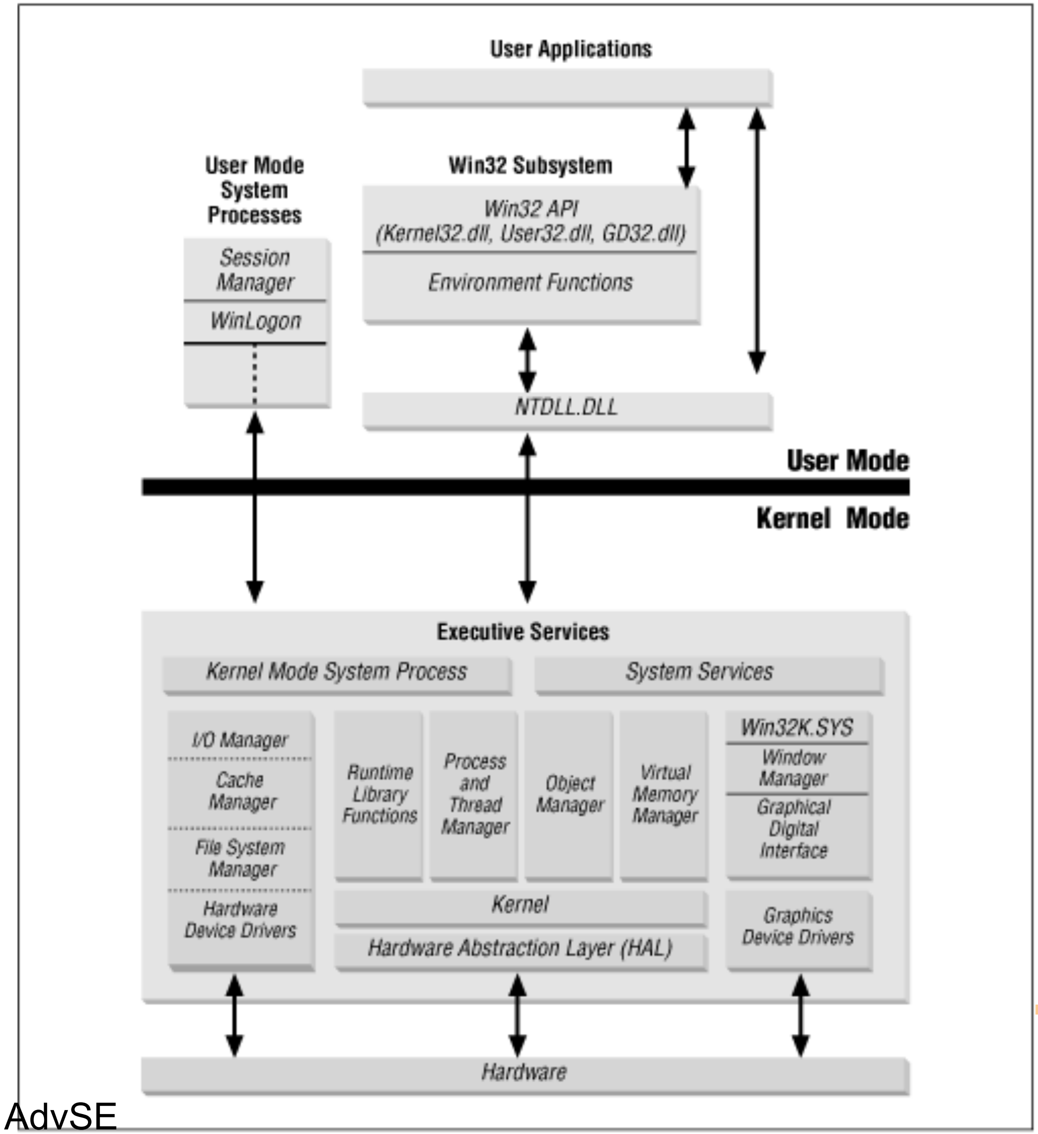
Architecture Description Languages

SOME STANDARD ARCHITECTURES

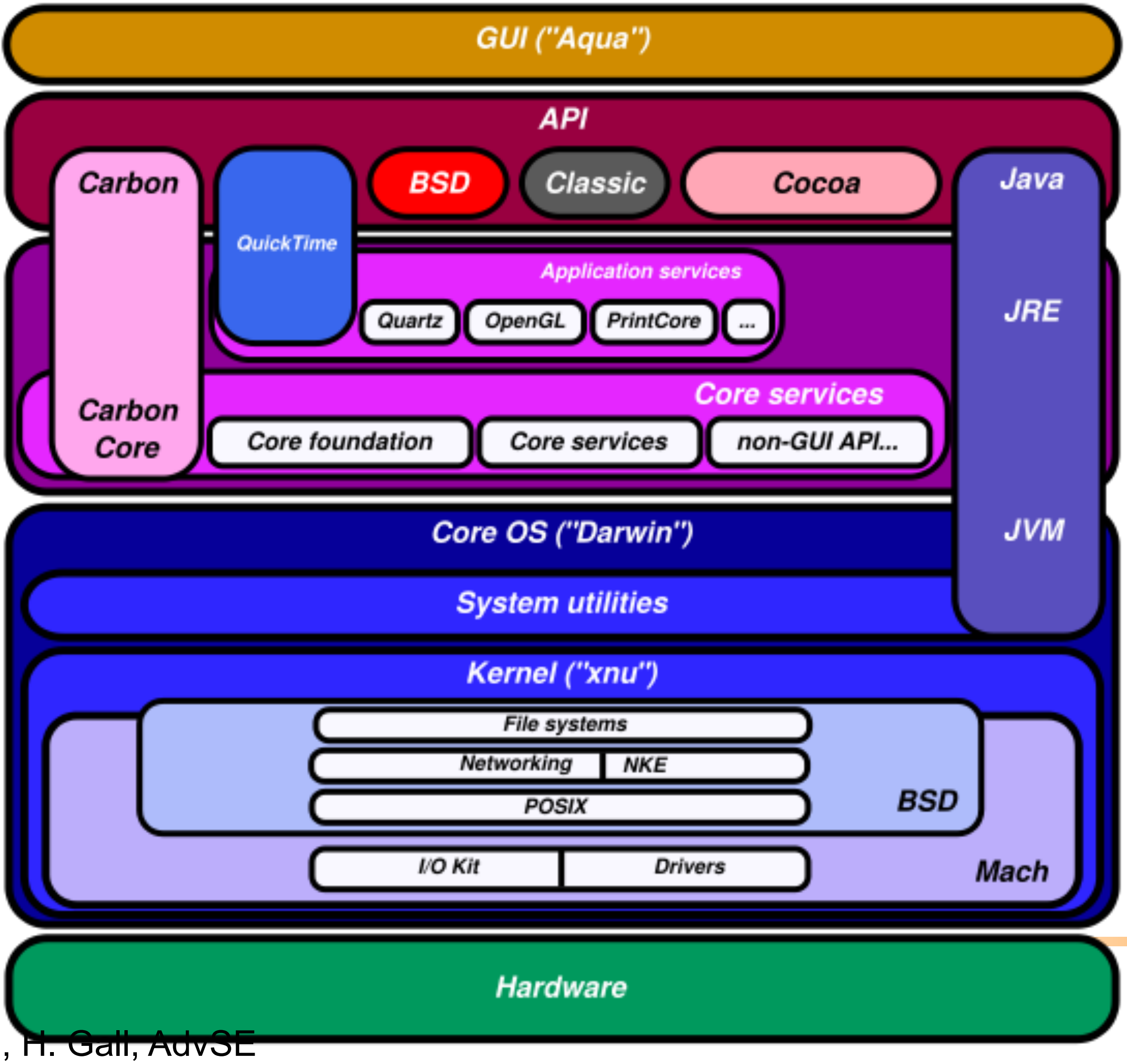
Architecture of Windows 2000



Architecture of Windows



Architecture of Mac OS X



Architecture of JBoss