Software Engineering

System Design

Prof. Dr. Peter Müller

Chair of Programming Methodology

The slides in this section are partly based on the lecture
“Software Engineering I” by Prof. Bernd Brügge, TU München

Spring Semester 10

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

4. System Design

4.1 Overview

4.2 Subsystem Decomposition

4.3 Architectural Styles

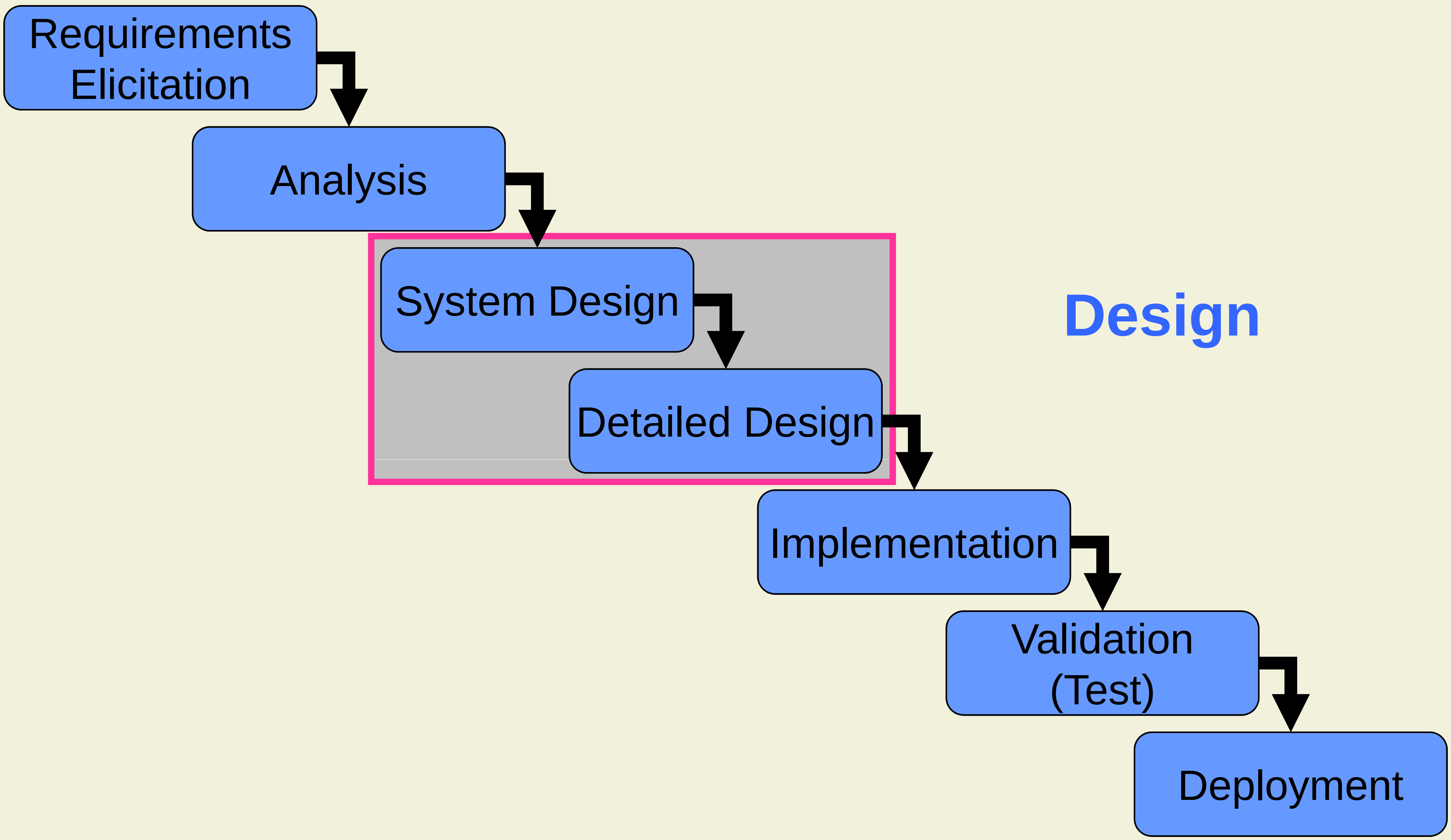
4.4 Specific System Design Issues

Software Design

"*There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.*"

[C.A.R. Hoare]

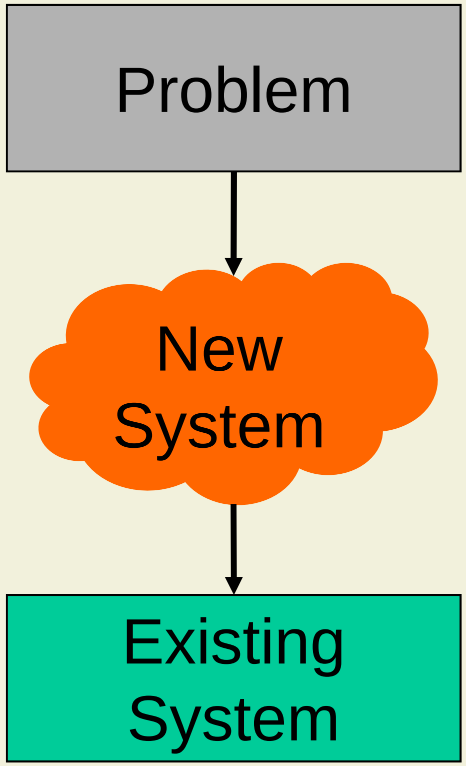
Waterfall Model of Project Life Cycle



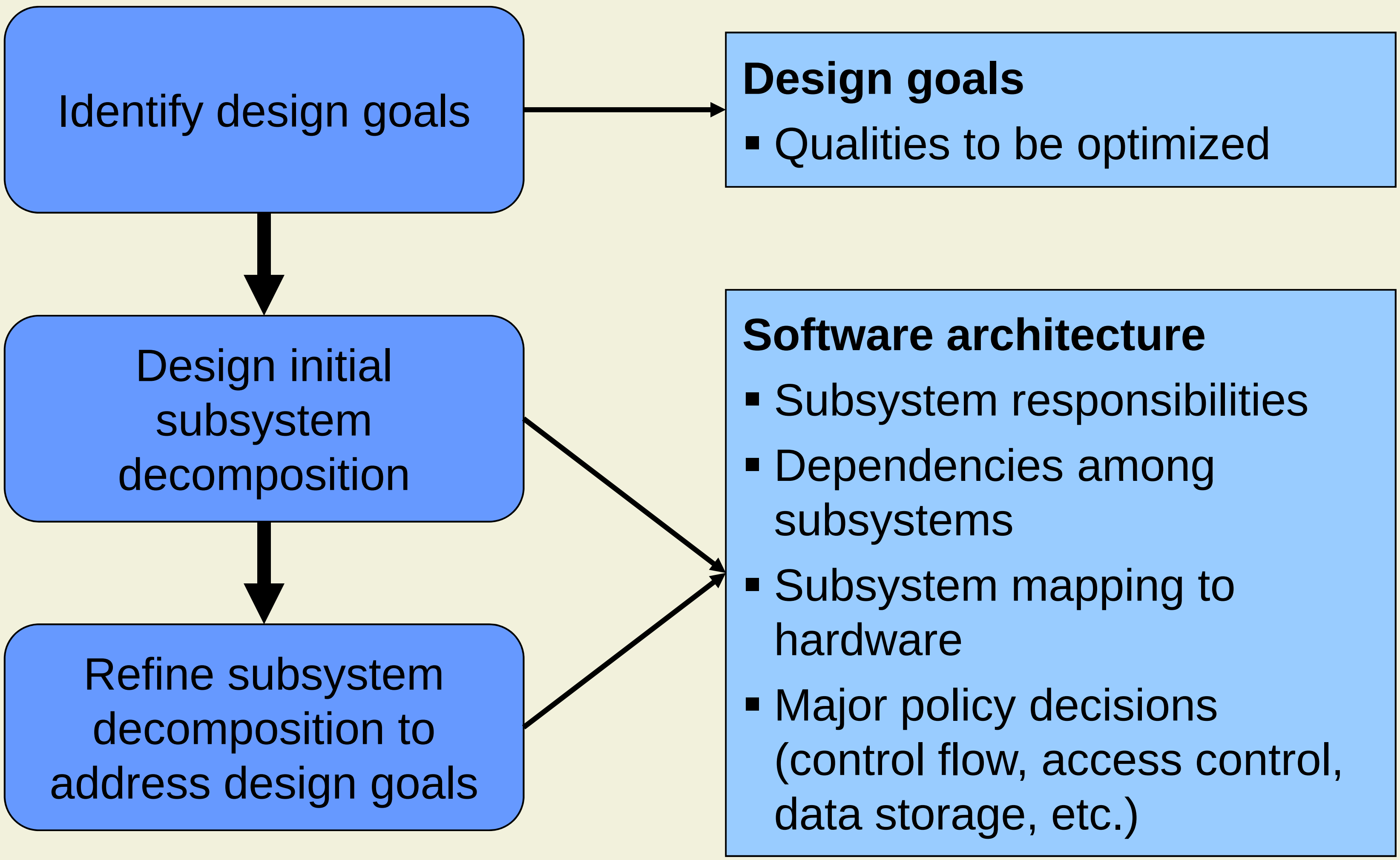
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Scope of System Design

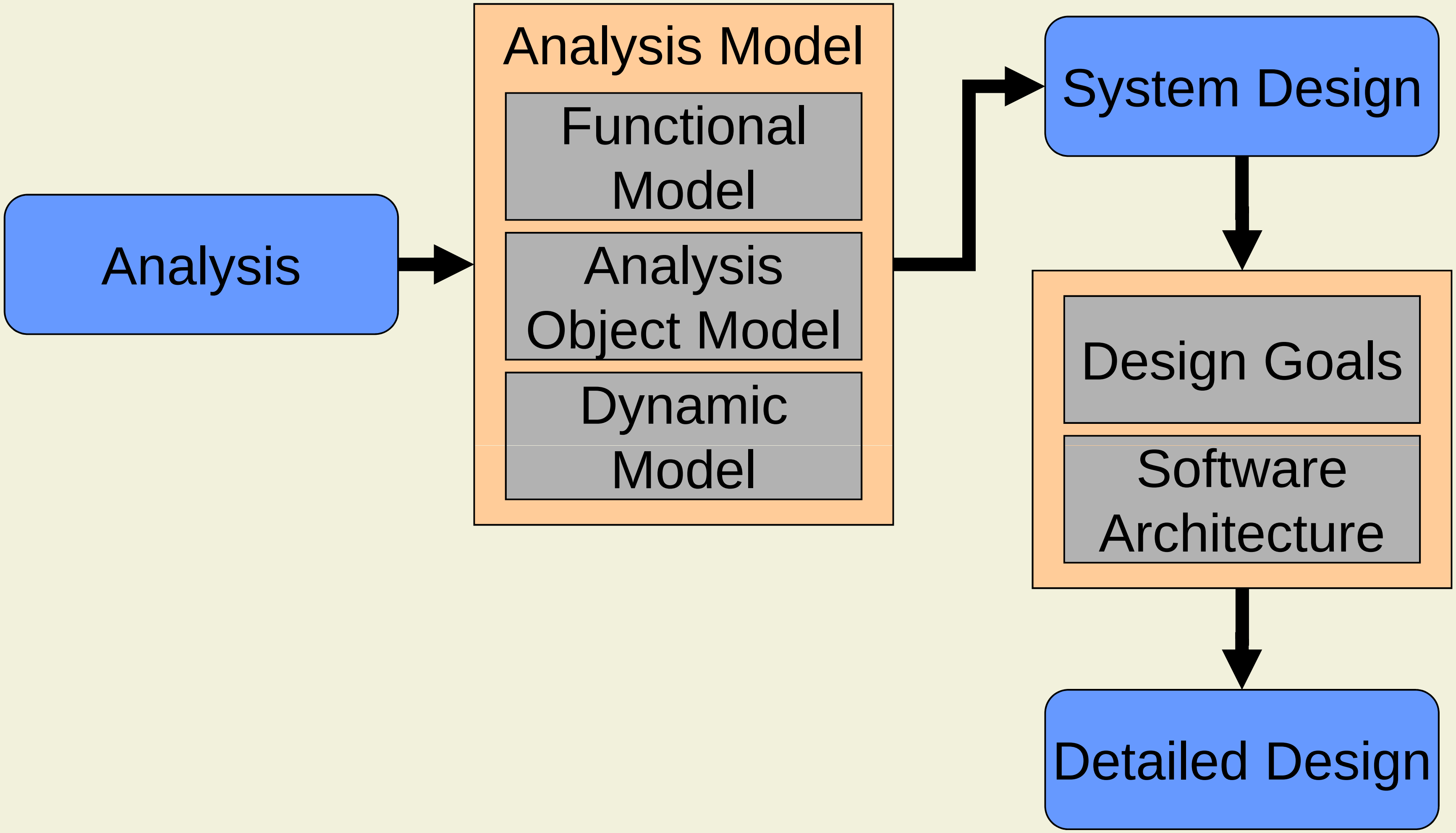
- Bridge the gap between a problem and an existing system in a manageable way
- Use divide and conquer: model the new system as a set of subsystems



Areas of System Design



From Analysis to System Design



Repetition: Representative Software Qualities

- Correctness
- Performance
- Robustness
- Scalability
- Reliability
- Usability
- Security
- Interoperability
- Maintainability
- Verifiability
- Understandability
- Reusability
- Evolvability
- Portability
- Repairability

Typical Design Trade-Offs

Functionality	Usability
Cost	Robustness
Performance	Portability
Rapid development	Functionality
Cost	Reusability
Backward Compatibility	Understandability

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

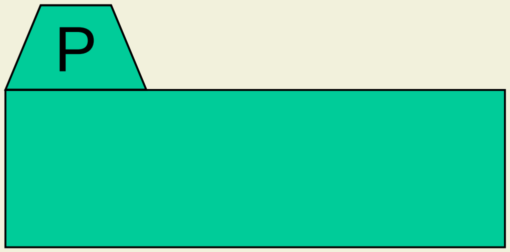
4. System Design

Why Decompose a System?

- Modularity is a software engineering **principle**
- **Management**
  - Partition the overall development effort
  - Clear assignment of requirements to modules, ideally one or more requirements mapped to one module
- **Modification**
  - Decouple parts of a system so that changes to one part do not affect other parts
- **Understanding**
  - Permit system to be understood as a composition of mind-sized chunks with one issue at a time

Subsystems

- Collection of classes, associations, operations, events and constraints that are closely interrelated with each other
- The objects and classes from the analysis object model are the "seeds" for the subsystems
- In UML, subsystems are modeled as packages



- In programming languages, subsystems are modeled as modules, packages, or by conventions

Services and Subsystem Interfaces

Subsystem

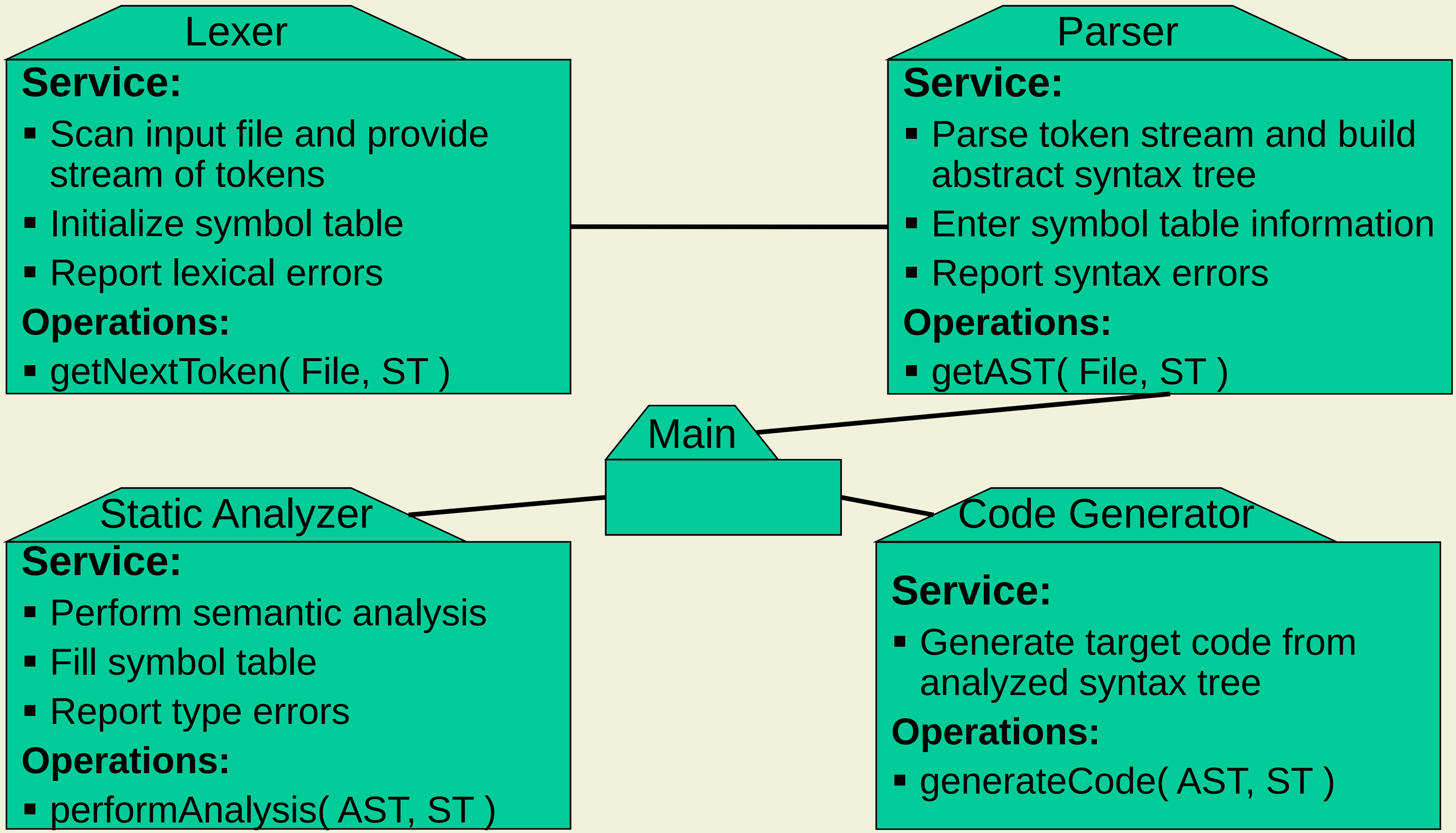
Subsystem interface: Set of fully-typed operations

- Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem
- Refinement of services
- Defined in detailed design

Service: Set of related operations

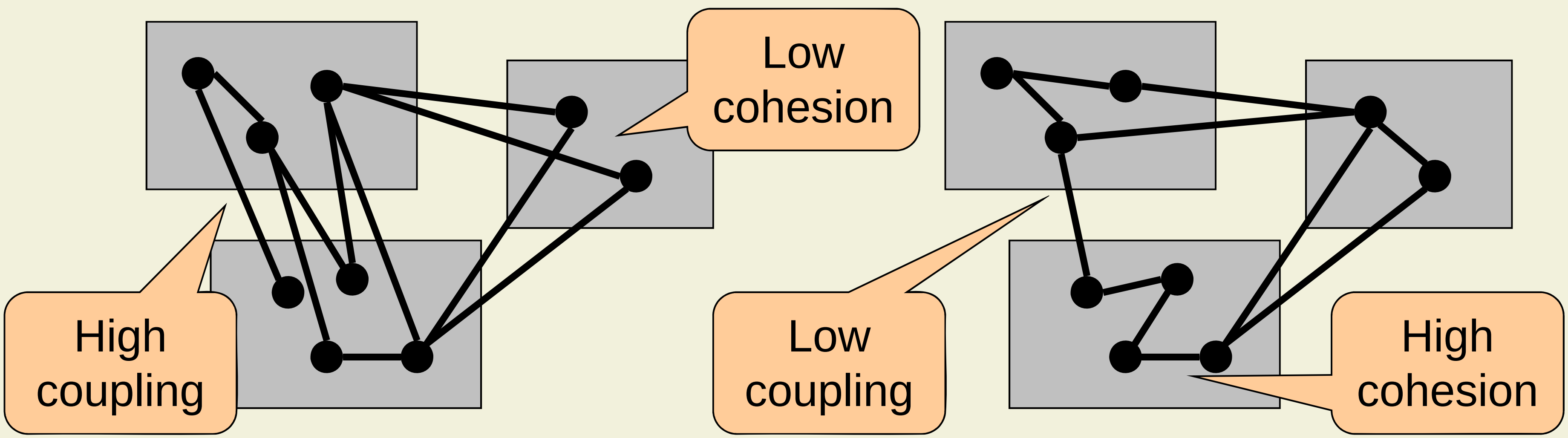
- Provided by the subsystem
- Share a common purpose
- Operations with parameters and high-level behavior defined during system design

Decomposition Example: Compiler



Repetition: Cohesion and Coupling

- Cohesion measures **interdependence** of the elements of **one module**
- Coupling measures **interdependence between** different **modules**
- Goal: **high cohesion** and **low coupling**



Achieving High Cohesion and Low Coupling

High cohesion

- Operations work on same data
- Operations implement a common abstraction (abstract data type)
- **Use object-orientation!**

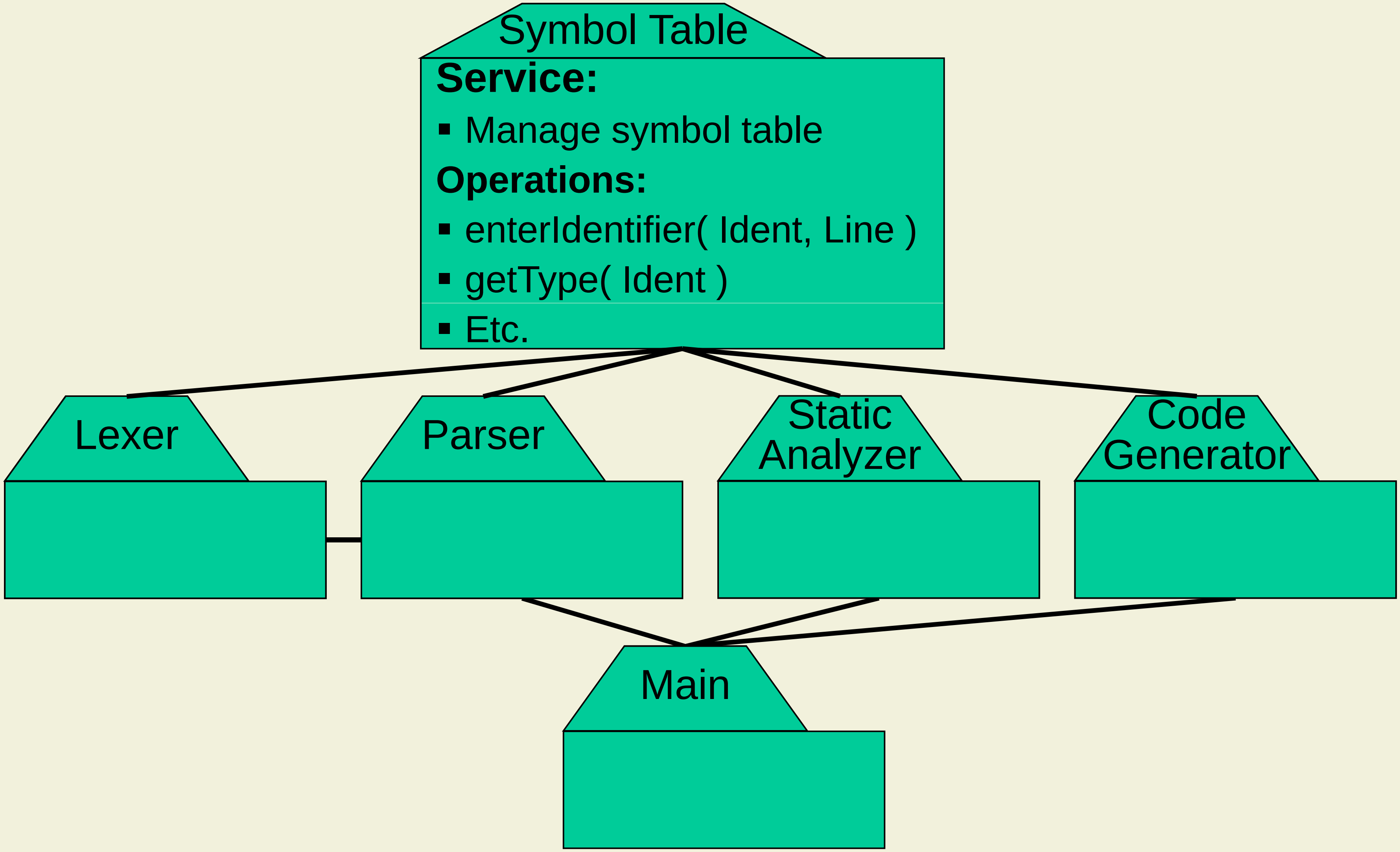
Low coupling

- Small interfaces
- Information hiding
- No global data
- Interactions are within subsystem rather than across subsystem boundaries
- **Use object-orientation!**

Cohesion and Coupling in Compiler Example

- Cohesion
  - Each subsystem has a clear responsibility
  - Very high cohesion in compiler
- Coupling
  - Small interfaces between subsystems
  - But: All subsystems read and update the symbol table (global data)
  - Changes of symbol table structure have effect on all subsystems
  - Coupling can be further reduced

Compiler Example Revisited



4. System Design

4.1 Overview

4.2 Subsystem Decomposition

4.3 Architectural Styles

4.4 Specific System Design Issues

Good Architecture

- Result of a consistent set of **principles** and **techniques**, applied **consistently** through all phases of a project
- **Resilient** in the face **of** (inevitable) **changes**
- Source of **guidance** throughout the product lifetime

- Reuse of established engineering knowledge
  - Application of architectural styles
  - Analogous to design patterns in detailed design

Architecture as an Art

- Inventing a novel architecture is a highly creative act
- Requires
  - Knowledge of existing work
  - Experience

Styles in Building Architecture


Ranch style


T-Ranch style


Raised Ranch style

- Customer picks an architectural style
  - Main components of the style are fixed
- Architect changes details according to requirements of the customer
- We apply the same approach to software

Elements of a Software Architecture

- Subsystems (components)
  - Computational units with specified interface
  - Examples: filters, databases, layers, objects
- Connectors
  - Interactions between components
  - Examples: method calls, pipes, event broadcasts, shared data
- See M. Shaw, D. Garlan: Software Architecture. Prentice Hall, 1996.

Architectural Styles: Overview

- Data flow systems
  - Batch sequential, pipe-and-filter
- Call-and-return system
  - Main program and subroutine
- Independent components
  - Interacting processes, event system
- Data-centered systems (repositories)
  - Databases, blackboards
- Hierarchical systems
  - Layers
  - Interpreters, rule-based systems
- Client-server systems
- Peer-to-peer systems

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

4. System Design

Data Flow Systems

- The **availability of data** controls the computation
- The structure is determined by the **orderly motion of data** from component to component
- Data flow is the only form of communication between components
- Variations
  - How control is exerted (e.g., push versus pull)
  - Degree of concurrency between processes
  - Topology

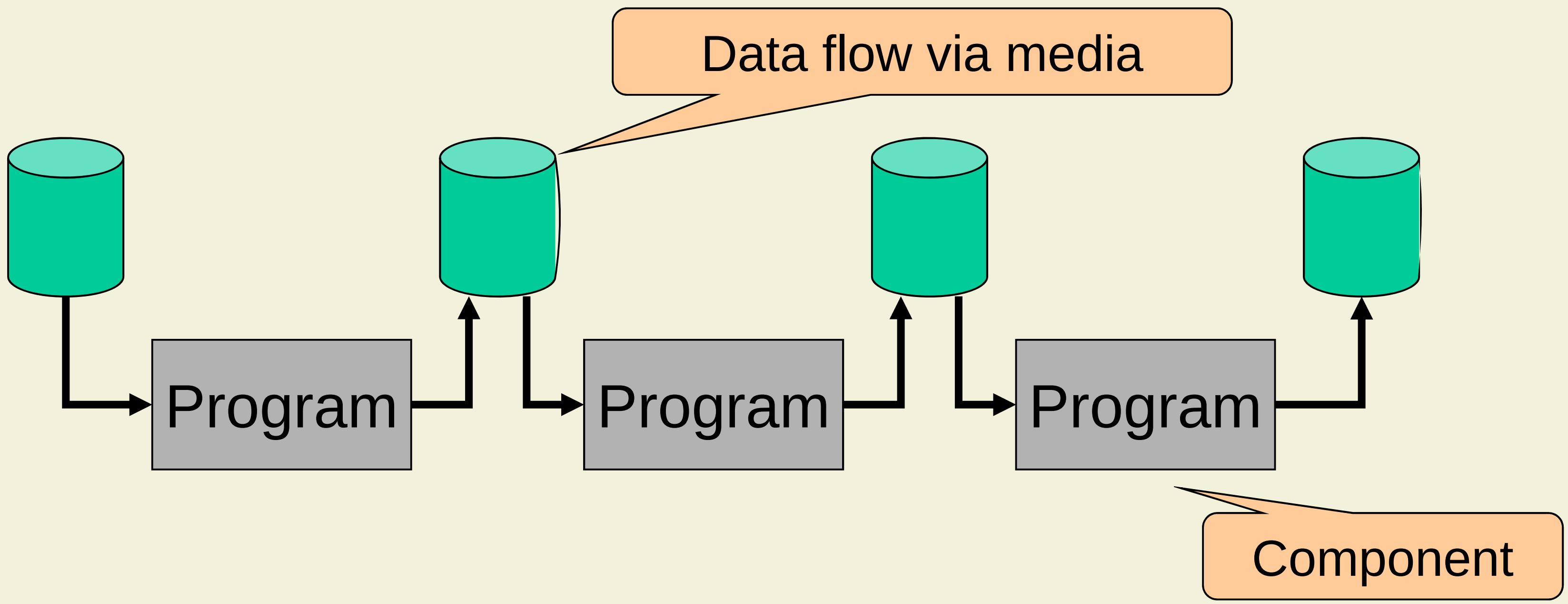
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Data Flow Systems (cont'd)

- Components: data flow components
  - Interfaces are input ports and output ports
  - Input ports read data; output ports write data
  - Computational model: read data from input ports, compute, write data to output ports
- Connectors: data streams
  - Uni-directional
  - Usually asynchronous, buffered
  - Computational model: transport data from writer to reader

Batch Sequential Style

- Components are **independent programs**
- Connectors are some type of **media**
- Each step **runs to completion** before next step begins

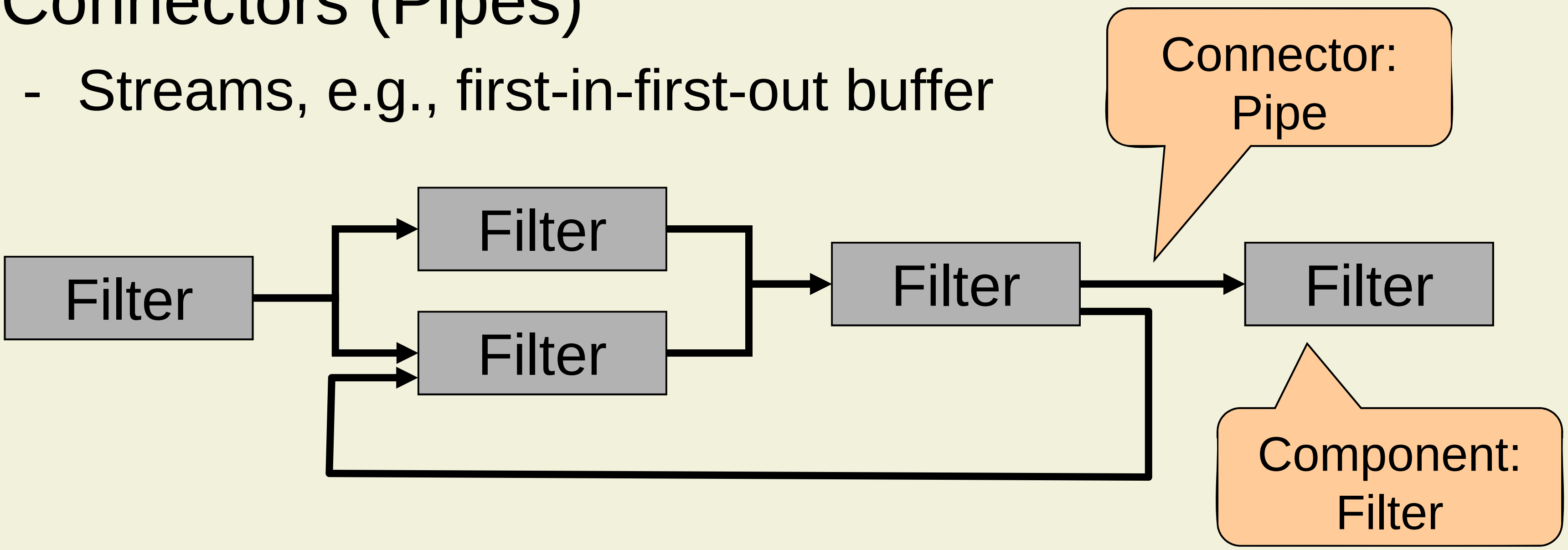


Batch Sequential Style: Properties

- History: Mainframes and magnetic tape
- Applications: Business data processing
  - Discrete transactions of predetermined type and occurring at periodic intervals
  - Creation of periodic reports based on periodic data updates
- Examples
  - Payroll computations
  - Tax reports

Pipe-and-Filter Style

- Components (Filters)
  - Read streams of input data
  - Locally transform input data
  - Produce streams of output data
- Connectors (Pipes)
  - Streams, e.g., first-in-first-out buffer

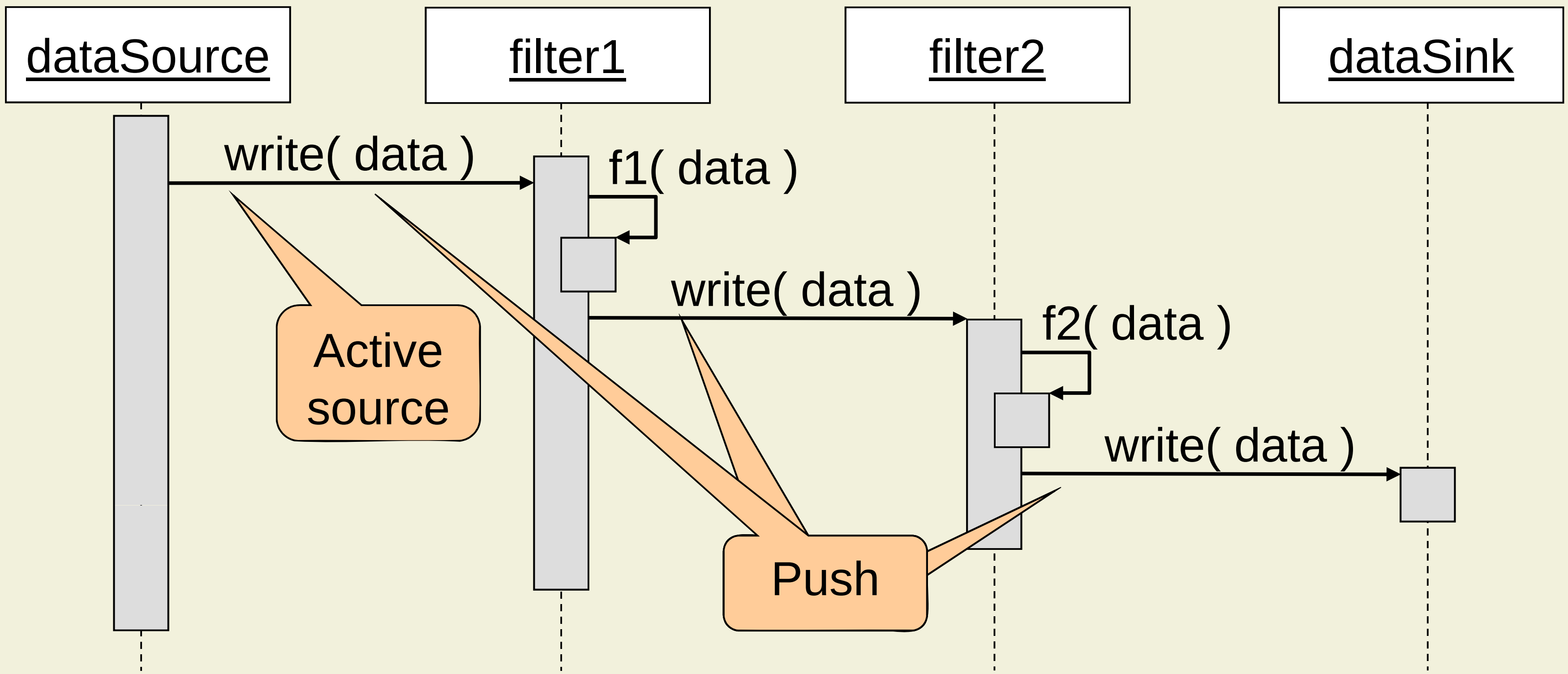


Pipe-and-Filter Style: Properties

- Data is processed **incrementally** as it arrives
- Output usually begins before input is consumed
- Filters must be **independent**, no shared state
- Filters don’t know upstream or downstream filters

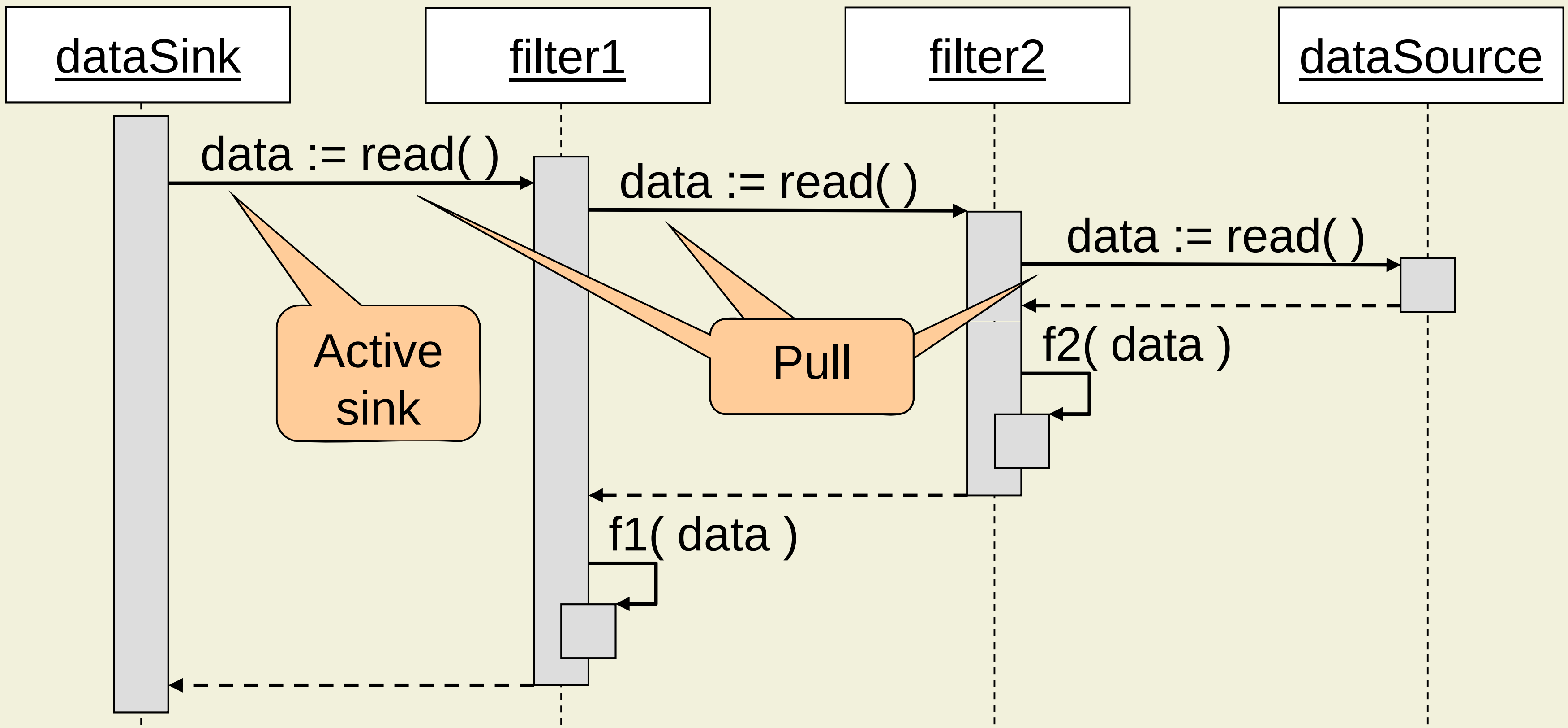
- Examples
  - lex/yacc-based compiler (scan, parse, generate code, …)
  - Unix pipes
  - Image / signal processing

Push Pipeline with Active Source



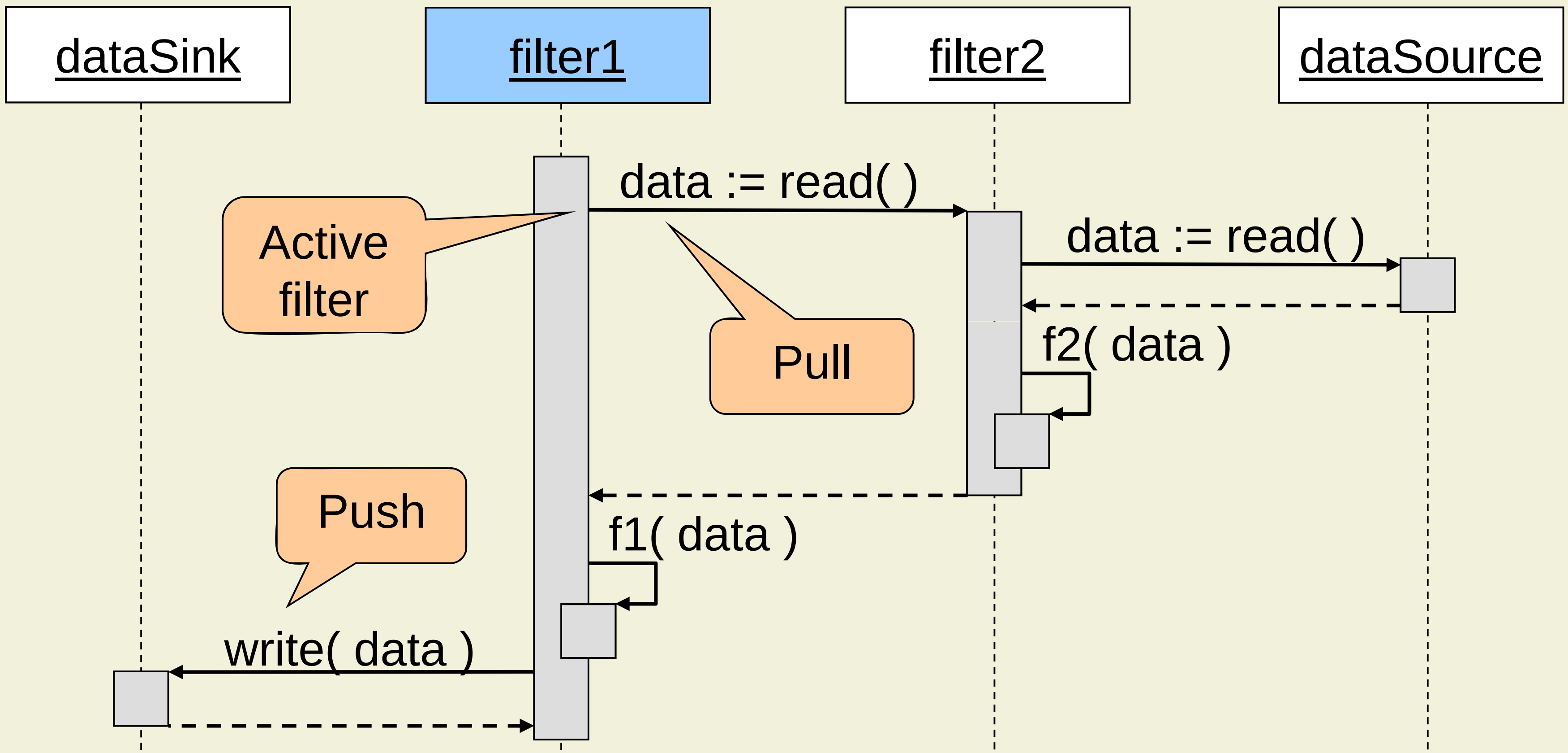
- Source of each pipe pushes data downstream
- Example: Unix pipes: grep pattern * ¦ wc

Pull Pipeline with Active Sink



- Sink of each pipe pulls data upstream
- Example: Compiler: lexer.getNextToken( )

Mixed Pipeline With Passive Source and Sink



- If more than one filter is pushing / pulling, synchronization is needed

Pipe-and-Filter Style: Discussion

Strengths

- Reuse: any two filters can be connected if they agree on that data format that is transmitted
- Ease of maintenance: filters can be added or replaced
- Potential for parallelism: filters implemented as separate tasks, consuming and producing data incrementally

Weaknesses

- Sharing global data is expensive or limiting
- Can be difficult to design incremental filters
- Not appropriate for interactive applications
- Error handling is Achilles heel, e.g., some intermediate filter crashes
- Often smallest common denominator on data transmission, e.g., ASCII in Unix pipes

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

4. System Design

Call-and-Return Style (Explicit Invocation)

- Components: Objects
- Connections: Messages (method invocations)
- Key aspects
  - Object preserves integrity of representation (encapsulation)
  - Representation is hidden from client objects
- Variations
  - Objects as concurrent tasks

Call-and-Return Style: Discussion

Strengths

- Change implementation without affecting clients
- Can break problems into interacting agents (distributed across multiple machines / networks)

Weaknesses

- Objects must know their interaction partners (in contrast to Pipe-and-Filter)
- When partner changes, objects that explicitly invoke it must change
- Side effects: if A uses B and C uses B, then C's effects on B can be unexpected to A

4. System Design

Event-Based Style (Implicit Invocation)

- Characterized by the style of communication between components
  - Component announces (broadcasts) one or more events
- Generalized Observer Design Pattern
- Components
  - May announce events
  - May register for events of other components with a callback
- Connectors
  - Bindings between event announcements and method calls (callbacks)

Event-Based Style: Example

Purse - Microsoft SpecSharp [design] - BagBroken.ssc

File Edit View Project Build Debug Tools Window Help

BagBroken.ssc

Bag | elems

```
public class Bag
{
  int[]! elems;
  int count; // Make this field [SpecPublic]!
  invariant 0 <= count && count <= elems.Length;

  public Bag(int[] initialElements) {
    this.count = initialElements.Length; // Guard this dereference!
    int[] e = new int[initialElements.Length]; // Make initialElements a non-null paramete
    initialElements.CopyTo(e, 0); // Use the less restrictive Copy method!
    this.elems = e;
    base();
  }

  public Bag(int[]! initialElements, int start, int howMany)
    requires 0 <= start && 0 <= howMany;
    requires start + howMany <= initialElements.Length;
  {
    this.count = howMany;
    int[] e = new int[howMany];
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Event-Based Style: Properties

- **Announcers** of events **do not know** which **components** will be affected by those events
- Components **cannot make assumptions about ordering of processing**, or what processing will occur as a result of their events

- Examples
  - Programming environment tool integration
  - User interfaces (Model-View-Controller)
  - Syntax-directed editors to support incremental semantic checking

Event-Based Style: Example

- Integrating tools in a shared environment

- Editor announces it has finished editing a module
  - Compiler registers for such announcements and automatically re-compiles module
  - Editor shows syntax errors reported by compiler
- Debugger announces it has reached a breakpoint
  - Editor registers for such announcements and automatically scrolls to relevant source line

Event-Based Style: Discussion

Strengths

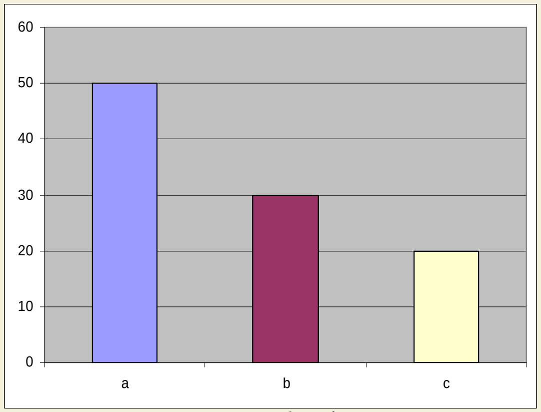
- Strong support for reuse: plug in new components by registering it for events
- Maintenance: add and replace components with minimum effect on other components in the system

Weaknesses

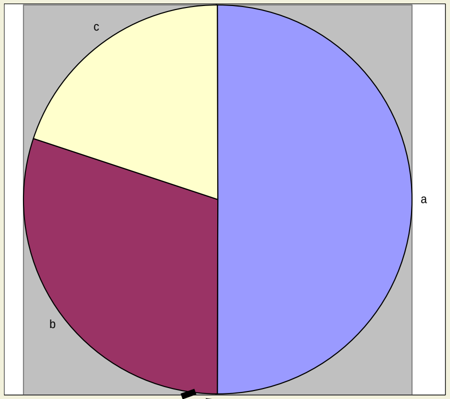
- Loss of control
  - What components will respond to an event?
  - In which order will components be invoked?
  - Are invoked components finished?
- Ensuring correctness is difficult because it depends on context in which invoked

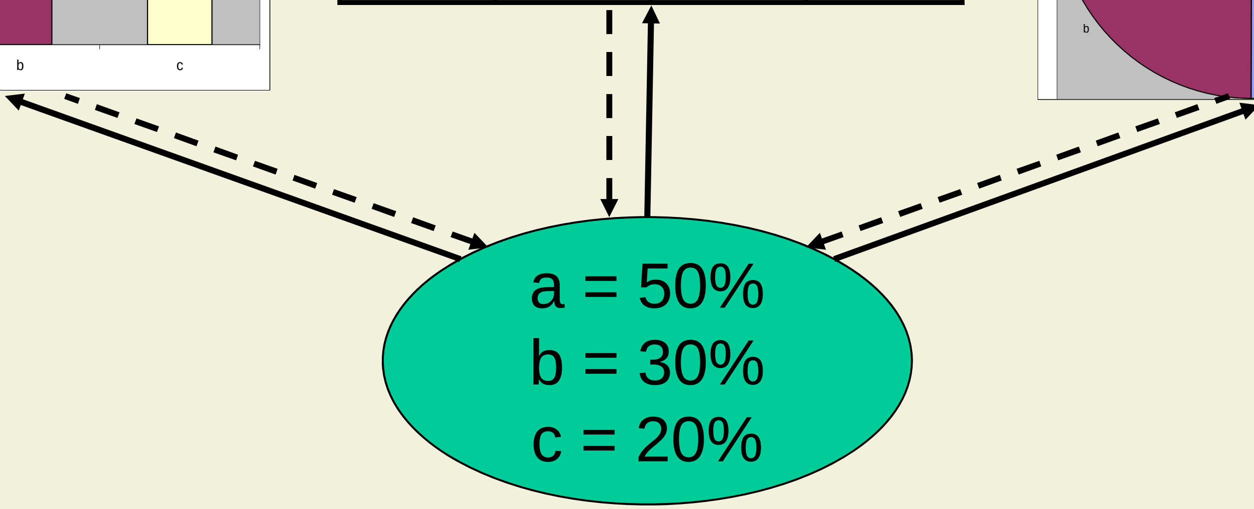
- In practice, call-and-return style and event-based style are combined

Model-View-Controller Example



	a	b	c
X	60	30	10
Y	50	30	20
Z	80	10	10

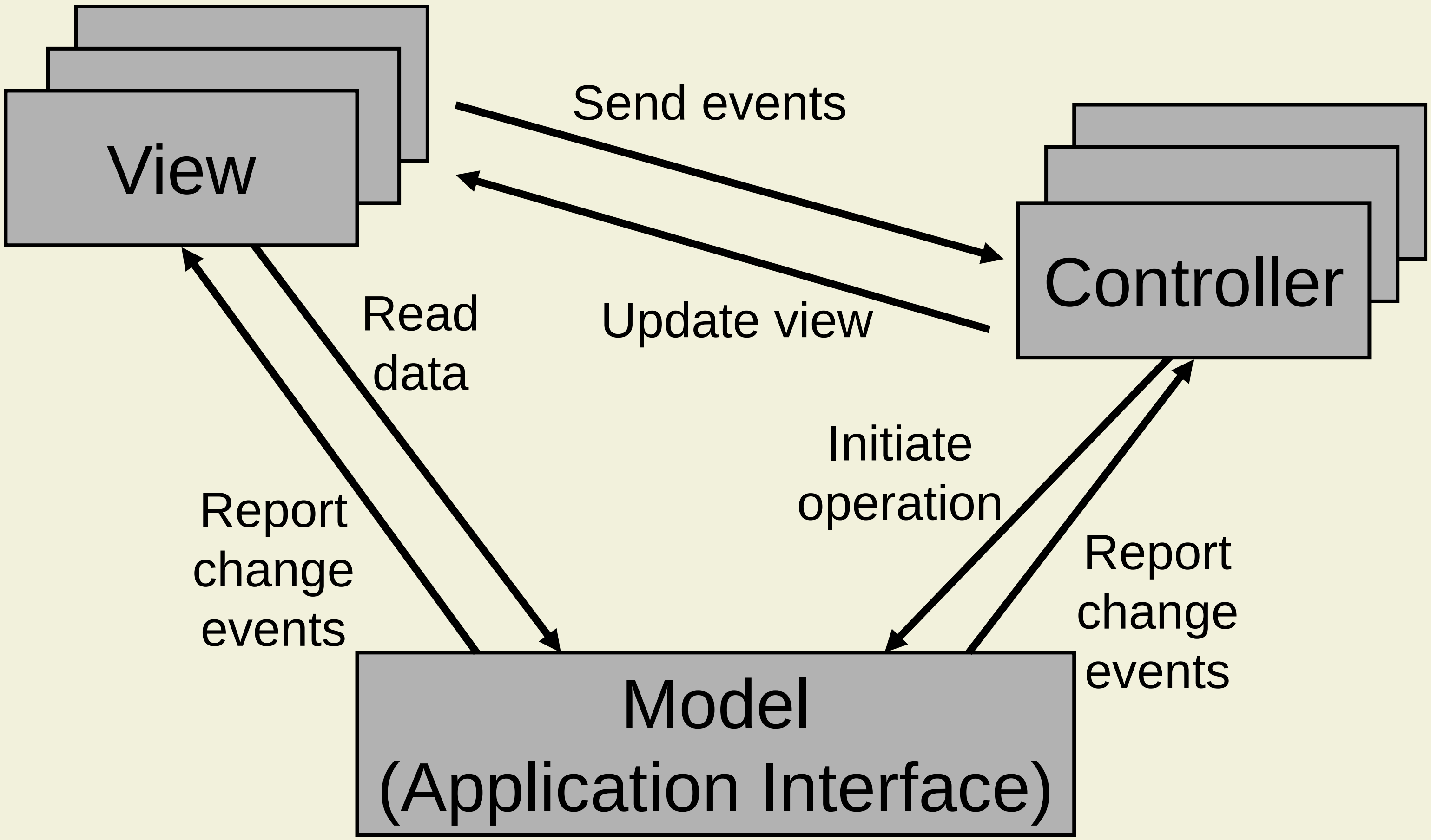




——→ Change notification

- - - - -→ Requests, modifications

Model-View-Controller Architecture

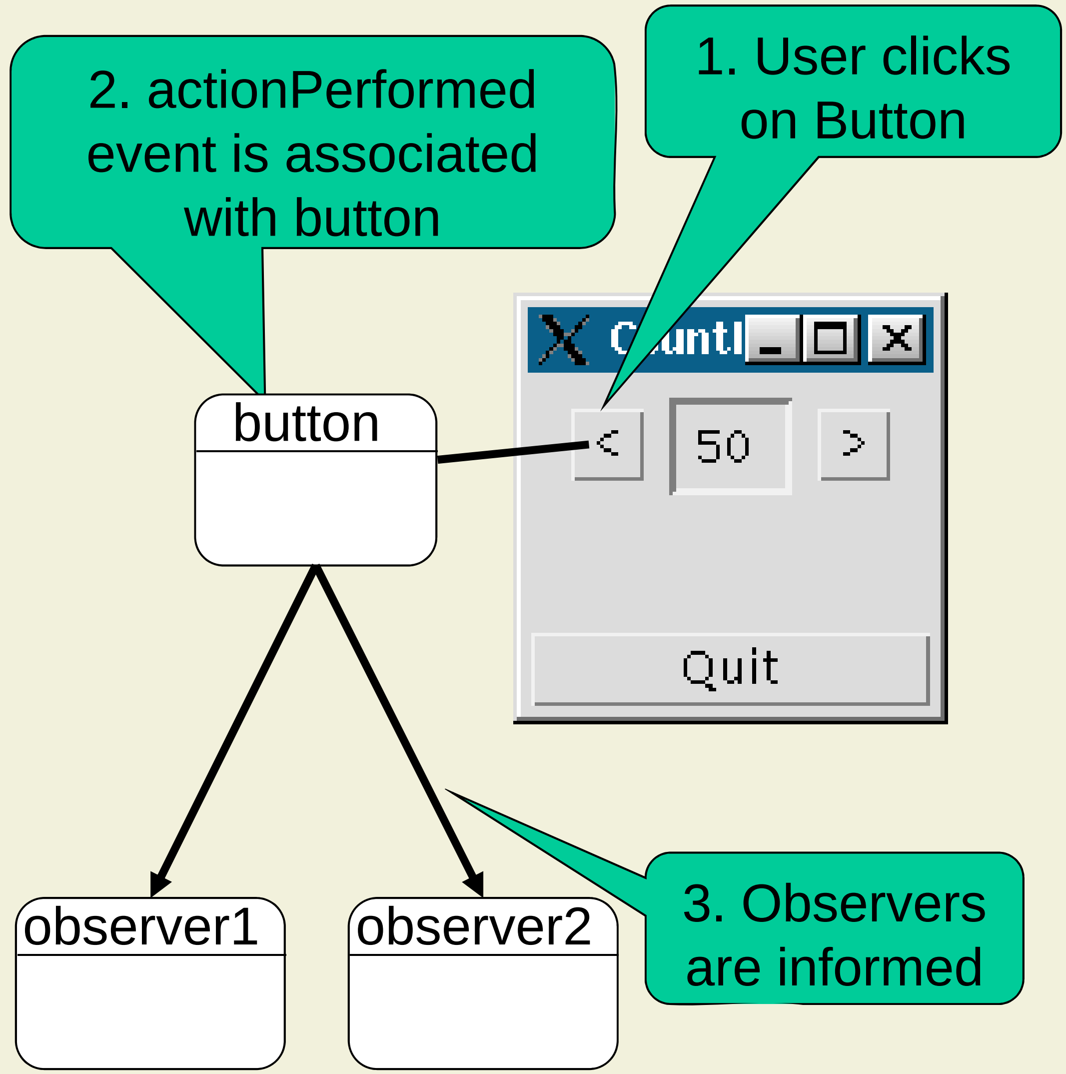


Model-View-Controller Architecture

- Components
  - Model contains the core functionality and data
  - One or more views display information to the user
  - One or more controllers handle user input
- Communication
  - Change-propagation mechanism via events ensures consistency between user interface and model
  - If the user changes the model through the controller of one view, the other views will be updated automatically

Model-View-Controller in Java

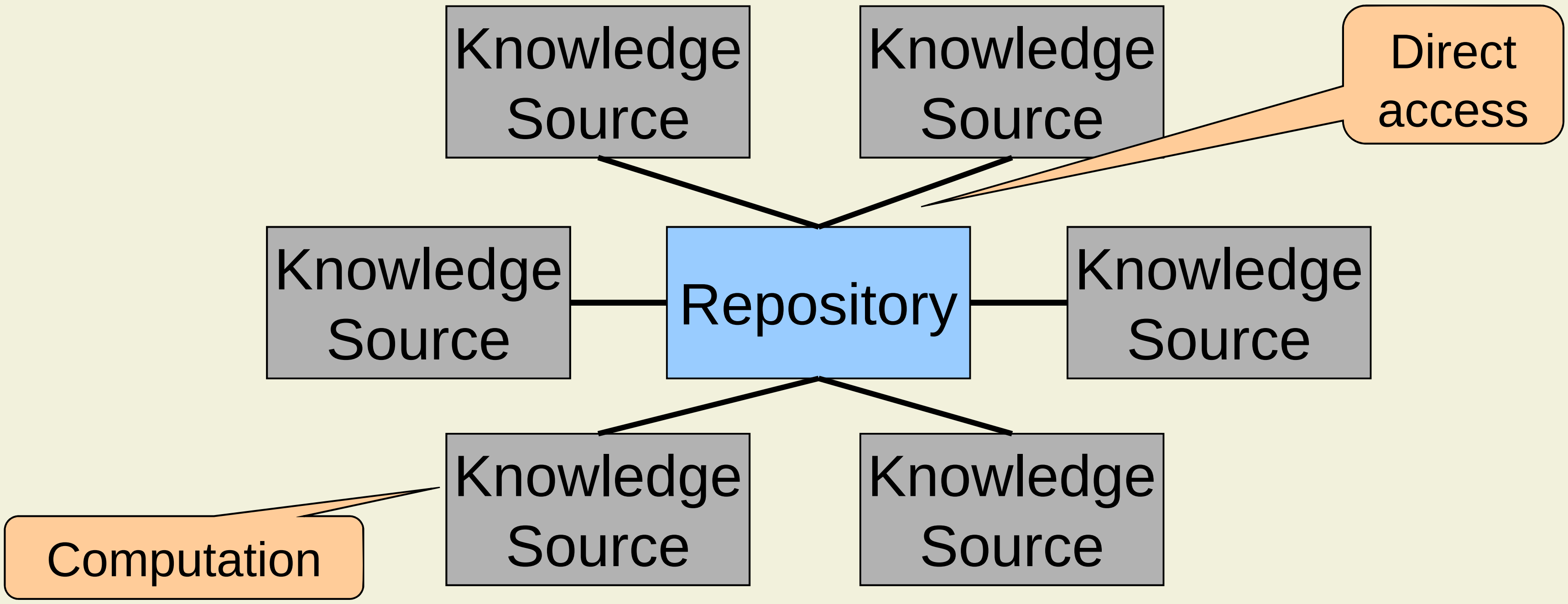
- Objects can register with a GUI component as observer for one or several event types
- Upon occurrence of an event, the event source informs all registered objects by invoking a method



4. System Design

Data-Centered Style (Repository Style)

- Components
  - Central data store component represents system state
  - Independent components operate on the data store



Special Case: Blackboard Architectures

- **Interactions** among knowledge sources **solely through repository**
- Knowledge sources make changes to the shared data that lead incrementally to solution
- Control is driven entirely by the state of the blackboard

- Example
  - Repository: modern compilers act on shared data: symbol table, abstract syntax tree
  - Blackboard: signal and speech processing

Data-Centered Style: Discussion

Strengths

- Efficient way to share large amounts of data
- Data integrity localized to repository module

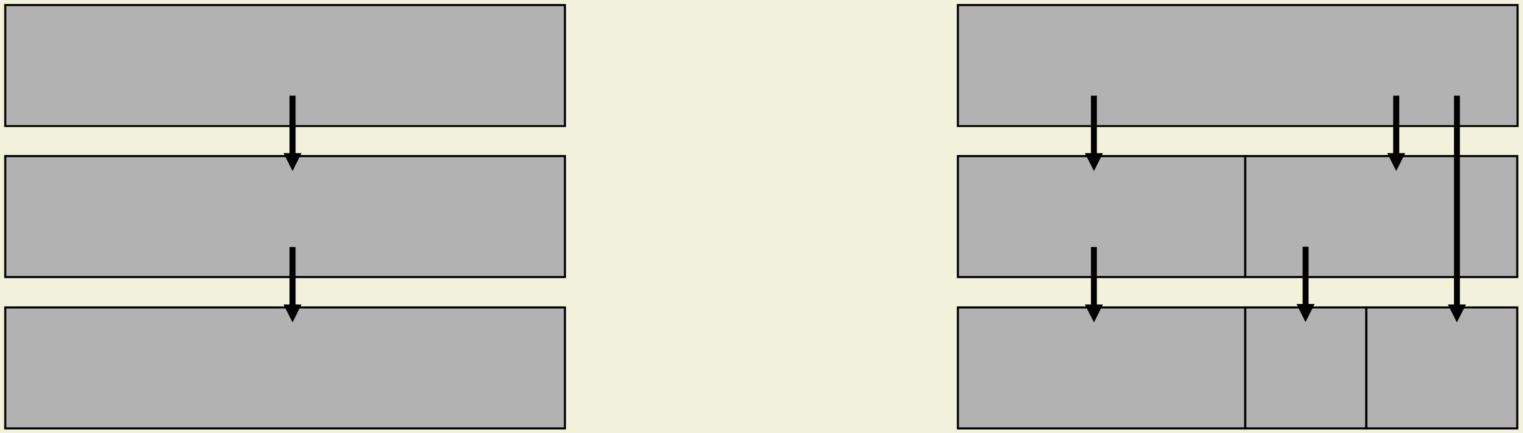
Weaknesses

- Subsystems must agree (i.e., compromise) on a repository data model
- Schema evolution is difficult and expensive
- Distribution can be a problem

4. System Design

Hierarchical Style (Layered Style)

- Components
  - Group of subtasks which implement an abstraction at some layer in the hierarchy
- Connectors
  - Protocols that define how the layers interact



Hierarchical Style: Properties

- Each layer provides **service to the layer above** it and acts as a client of the layer below
- Each layer collects services at a particular level of **abstraction**
- A layer depends only on lower layers
  - Has no knowledge of higher layers
- Example
  - Communication protocols
  - Operating systems

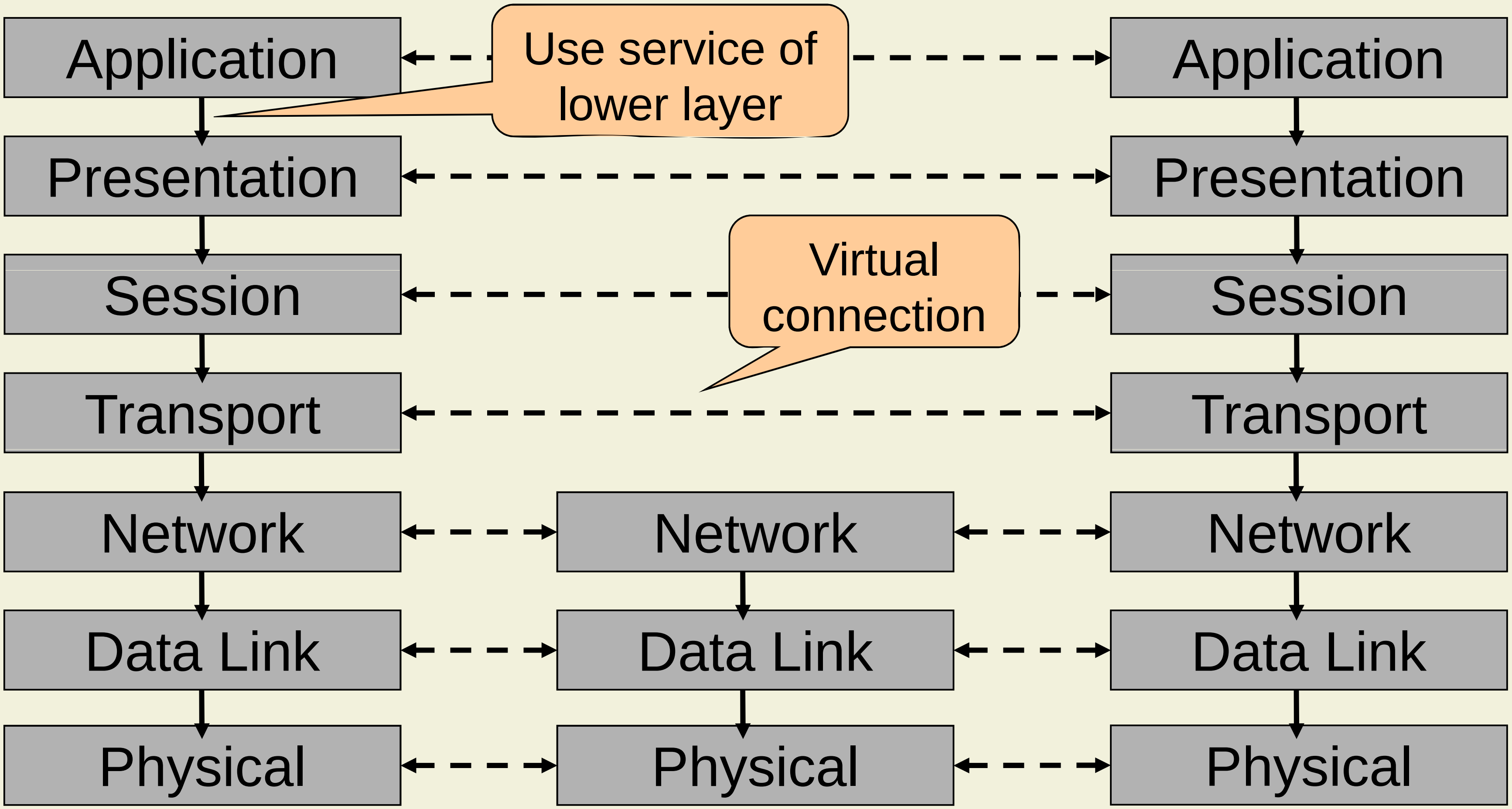
Hierarchical Style: Example

- The OSI Networking Model
  - Each level supports communication at a level of abstraction
  - Protocol specifies behavior at each level of abstraction
  - Each layer deals with specific level of communication and uses services of the next lower level
- Layers can be exchanged
  - Example: Token Ring for Ethernet on Data Link Layer

OSI Model Layers and Their Responsibilities

Application	The system you are designing
Presentation	Performs data transformation services, such as byte swapping and encryption
Session	Initializes a connection, including authentication
Transport	Reliably transmits messages
Network	Transmits and routes data within the network
Data Link	Sends and receives frames without error
Physical	Sends and receives bits over a channel

Hierarchical Style: Example (cont'd)



Hierarchical Style: Discussion

Strengths

- Increasing levels of abstraction as we move up through layers: partitions complex problems
- Maintenance: in theory, a layer only interacts with layer below (low coupling)
- Reuse: different implementations of the same level can be interchanged

Weaknesses

- Performance: communicating down through layers and back up, hence bypassing may occur for efficiency reasons

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

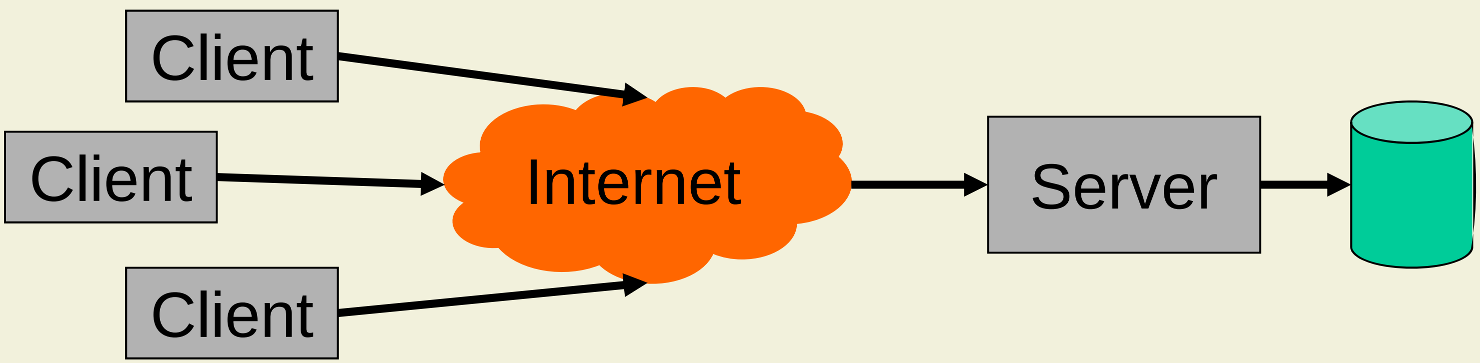
Interpreters

- Architecture is based on a **virtual machine** produced in software
- Special kind of a **layered architecture** where a layer is implemented as a true language interpreter
- Components
  - “Program” being executed and its data
  - Interpretation engine and its state
- Example: Java Virtual Machine
  - Java code translated to platform independent bytecode
  - JVM is platform specific and interprets the bytecode

4. System Design

Client Server Style

- Components
  - Subsystems are independent processes
  - Servers provide specific services such as printing, etc.
  - Clients use these services
- Connectors
  - Data streams, typically over a communication network



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Client Server Style Example: Databases

- Front-end: User application (client)
  - Customized user interface
  - Front-end processing of data
  - Initiation of server remote procedure calls
  - Access to database server across the network
- Back-end: Database access and manipulation (server)
  - Centralized data management
  - Data integrity and database consistency
  - Database security
  - Concurrent operations (multiple user access)
  - Centralized processing (for example archiving)

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Client Server Style: Variants

- Thick / fat client
  - Does as much processing as possible
  - Passes only data required for communications and archival storage to the server
  - Advantages: less network bandwidth, fewer server requirements
- Thin client
  - Has little or no application logic
  - Depends primarily on the server for processing activities
  - Advantages: lower IT admin costs, easier to secure, lower hardware costs.

Client Server Style: Discussion

Strengths

- Makes effective use of networked systems
- May allow for cheaper hardware
- Easy to add new servers or upgrade existing servers
- Availability (redundancy) may be straightforward

Weaknesses

- Data interchange can be hampered by different data layouts
- Communication may be expensive
- Data integrity functionality must be implemented for each server
- Single point of failure

4. System Design

Peer-to-Peer Style

- Similar to client-server style, but **each component is both client and server**
- Pure peer-to-peer style
  - No central server, no central router
- Hybrid peer-to-peer style
  - Central server keeps information on peers and responds to requests for that information
- Examples
  - File sharing applications, e.g., Napster, Gnutella, Kazaa
  - Communication and collaboration, e.g., Skype

Peer-to-Peer Style: Discussion

Strengths

- Efficiency
  - All clients provide resources
- Scalability
  - System capacity grows with number of clients
- Robustness
  - Data is replicated over peers
  - No single point of failure in the system (in pure peer-to-peer style)

Weaknesses

- Architectural complexity
- Resources are distributed and not always available
- More demanding of peers (compared to client-server)
- New technology not fully understood

Architectural Style Case Study

- 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.

- We discuss and evaluate different system designs

KWIC Example

Input

- Star Wars
- The Empire Strikes Back
- The Return of the Jedi

Output

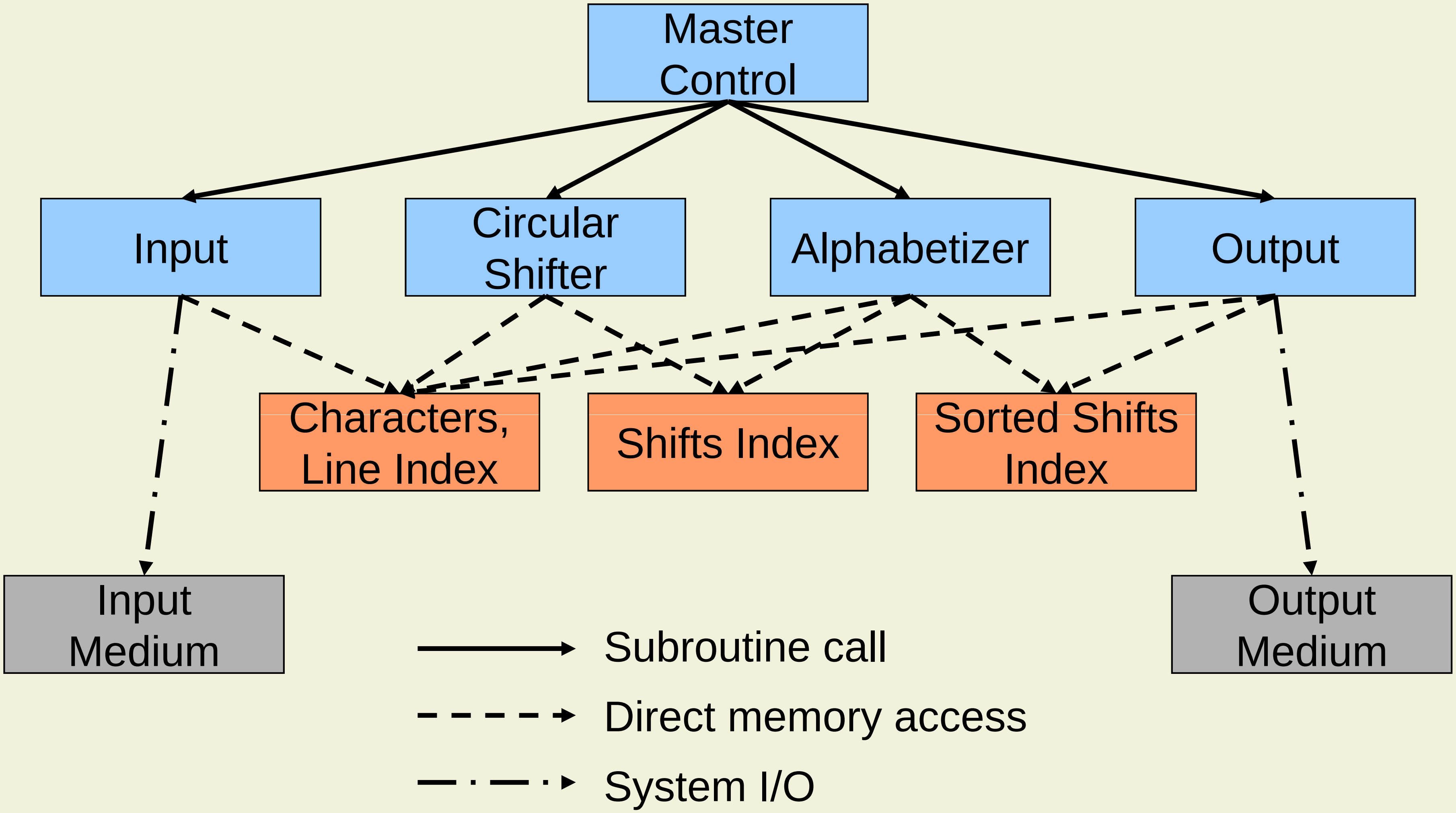
- Back The Empire Strikes
- Empire Strikes Back The
- Jedi The Return of the
- Return of the Jedi The
- Star Wars
- Strikes Back The Empire
- The Empire Strikes Back
- The Return of the Jedi
- Wars Star
- of the Jedi The Return
- the Jedi The Return of

Evaluation Criteria

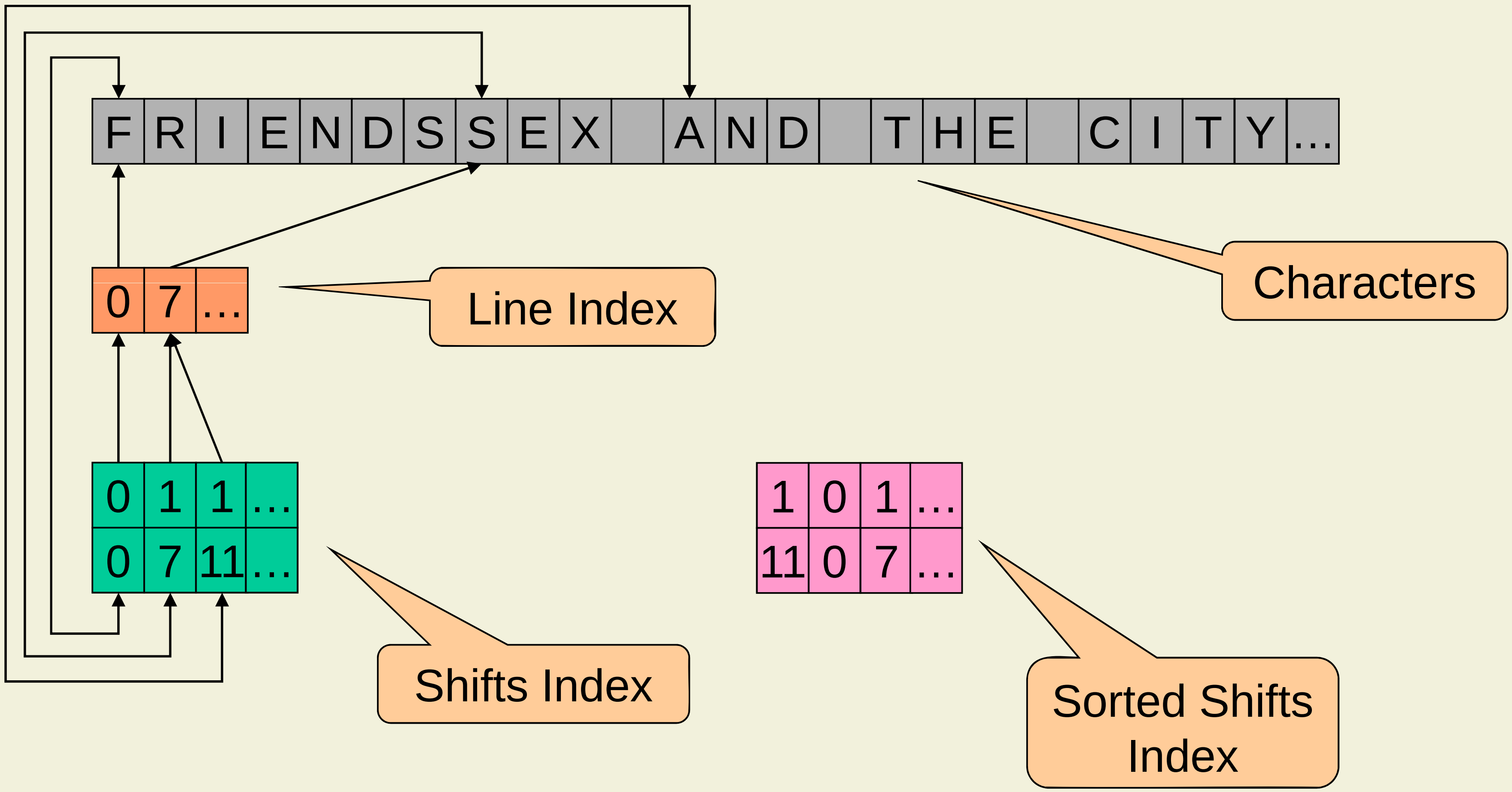
- Changes in algorithm
  - Line shifting on each line as it is read, on all the lines after they are read, or on demand when the alphabetization requires a new set of shifted lines
- Changes in data representation
  - Lines and circular shifts can be stored in various ways
- Enhancement to system function
  - Elimination of certain noise words ("a", "an", "and", etc.)
  - Interaction
- Performance: space and time
- Reuse

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Solution 1: Subroutines with Shared Data



Solution 1: Data Representation

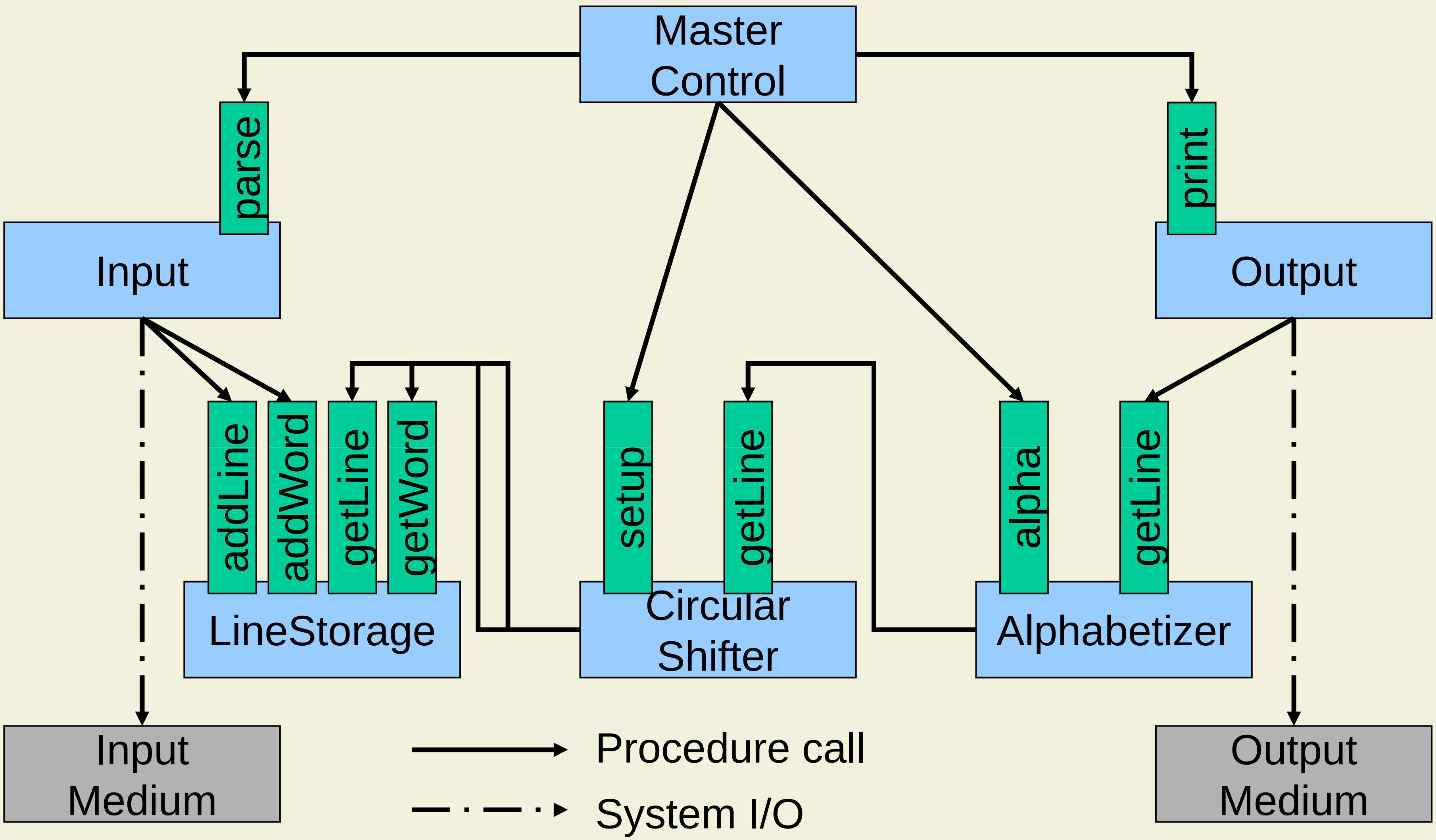


Solution 1: Discussion

- Pros
  - **Efficient data representation** (data stored only once)
  - Distinct computational aspects are isolated in different modules
- Cons
  - **Change** in data storage format **affects all modules**
  - Similarly: changes in algorithm and enhancements to system function
  - **Reuse is not well-supported** (each module is tightly tied to this particular implementation)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

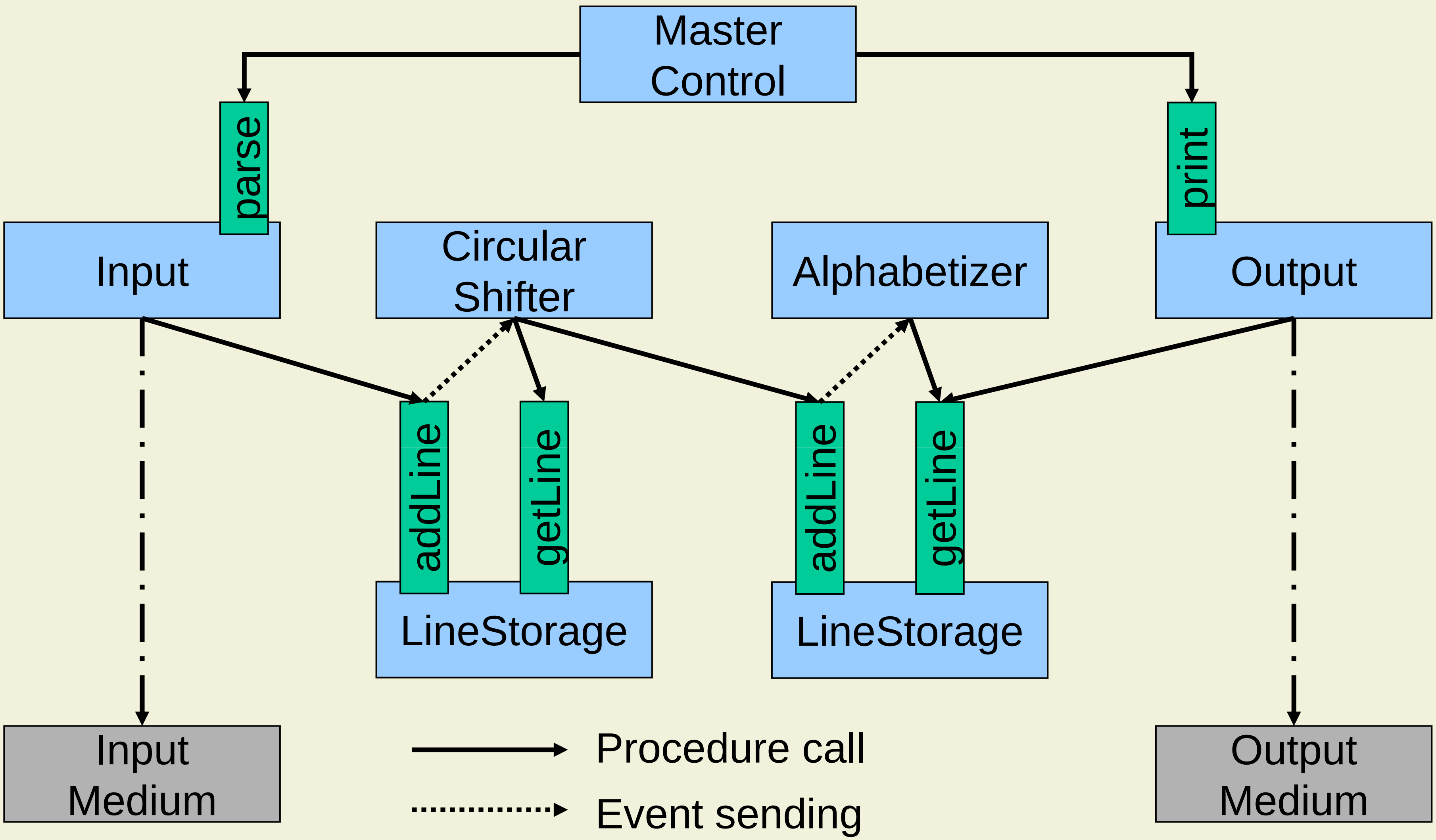
Solution 2: Abstract Data Types



Solution 2: Discussion

- Same processing modules as first solution, but better amenable to change
- Data not directly shared by components
- Pros
    - **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
- Cons
    - Not particularly well-suited to enhancements

Solution 3: Implicit Invocation (Event-Based)

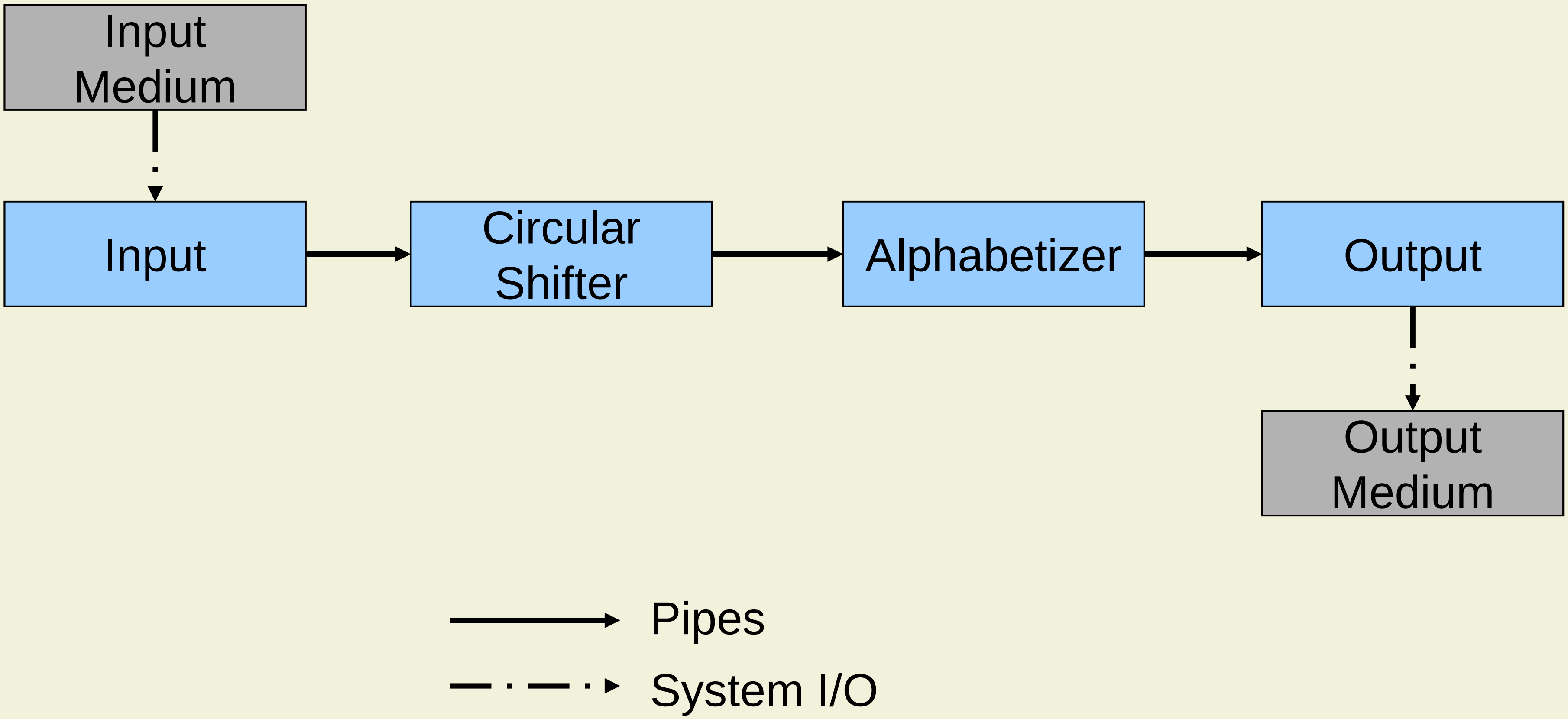


Solution 3: Discussion

- Pros
  - Supports **functional enhancements**: additional modules can be attached to the system by registering them to be invoked on certain events
  - **Data representations can be changed**
  - **Reuse**: implicitly invoked modules only rely on the existence of certain externally triggered events
- Cons
  - Difficult to change the **order of processing**
  - Uses **more space** than solutions 1 and 2

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Solution 4: Pipe-and-Filter



Solution 4: Discussion

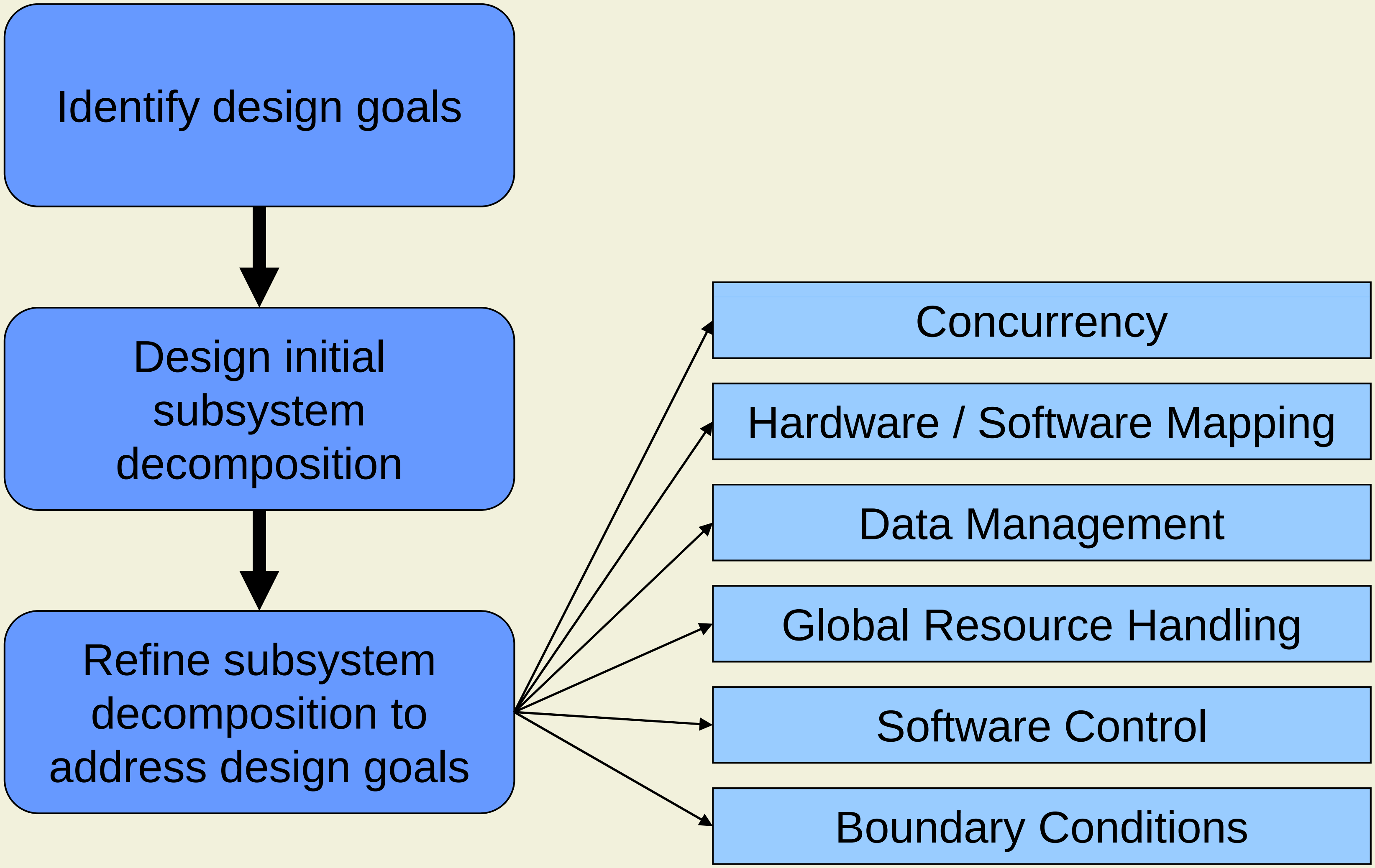
- Pros
  - **Intuitive** flow of processing
  - **Reuse**: each filter can function in isolation
  - **New functions** are easily added to the system by inserting filters at the appropriate point in the processing sequence
- Cons
  - Difficult (impossible) to support an **interactive system**
  - **Inefficient in terms of space**, since each filter must copy all of the data to its output ports

KWIC Case Study: Summary

	Subroutines	Abstract Data Types	Implicit Invocation	Pipe-and-Filter
Change in Algorithm	-	-	+	+
Change in Data Rep.	-	+	+	-
Change in Function	+	-	+	+
Performance	+	o	-	-
Reuse	-	+	+	+

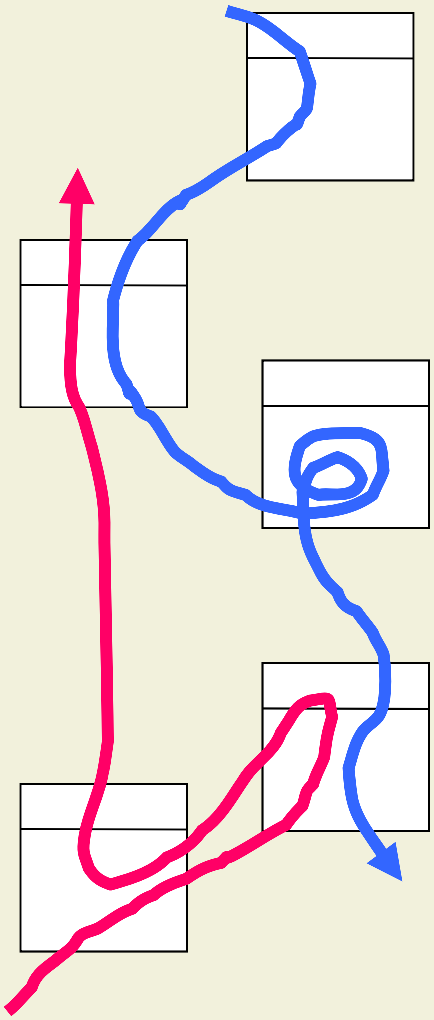
4. System Design

Areas of System Design: Specific Issues



Concurrency: Threads

- Execution threads are **sequences of atomic actions** during a program execution
- Concurrent programs can have **more than one thread**
- Execution of threads can be **parallel** (on several processors) or **virtually parallel** (on one processor)
- Design goal: response time, performance



Concurrency Questions

- Which objects of the object model are **independent**?
  - Candidates for separate threads
- Does the system support **multiple users**?
  - Example: Client-server architecture with several clients
- Can a single request to the system be **decomposed** into multiple requests? Can these requests be **handled** in **parallel**?
  - Search in a distributed database
  - Image recognition by decomposing the image into stripes

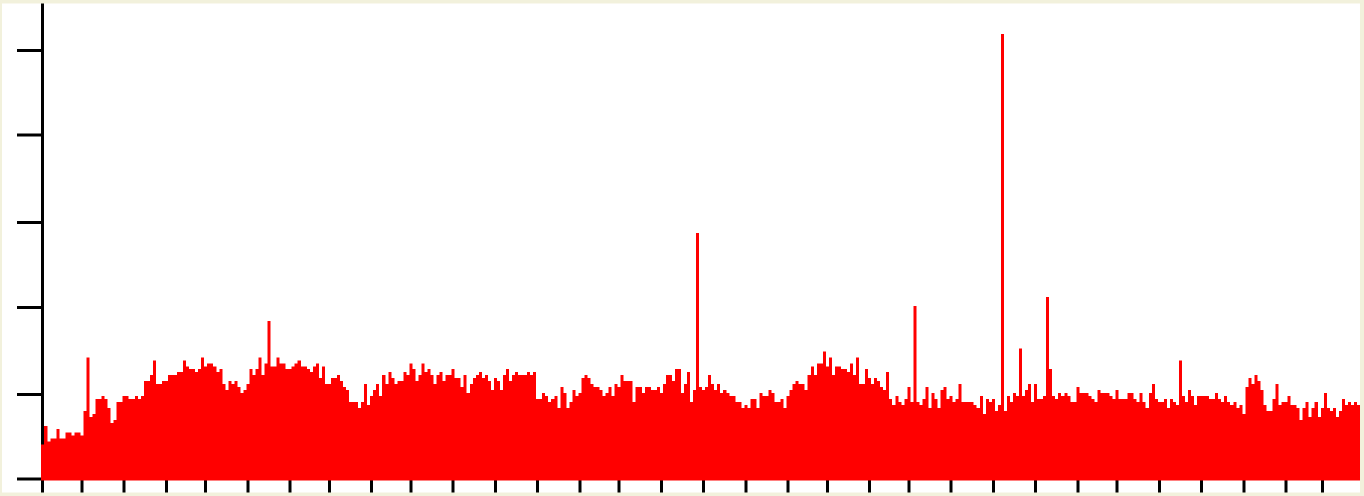
Hardware / Software Mapping

- This activity addresses two questions:
  - How shall we realize the subsystems: **with hardware or with software**?
  - How do we **map the object model** on the chosen hardware and software?
- Much of the difficulty of designing a system comes from meeting externally-imposed hardware and software constraints

Mapping the Objects

- Processor issues
  - Is the computation rate too **demanding** for a single processor?
  - Can we get a **speedup** by distributing tasks across several processors?
  - How many processors are required to **maintain steady state load**?
- Memory issues
  - Is there enough memory to buffer **bursts of requests**?

Mapping the Objects (cont'd)



- Example: stock trading
  - Usually steady rate of stock orders per day
  - Extreme peaks for important IPOs
- Bank is liable for loss of orders
  - System must be able to handle peak load

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Mapping the Associations

- Which of the **client-supplier relationships** in the analysis / design model correspond to **physical connections**?

- Describe the **logical connectivity** (subsystem associations)

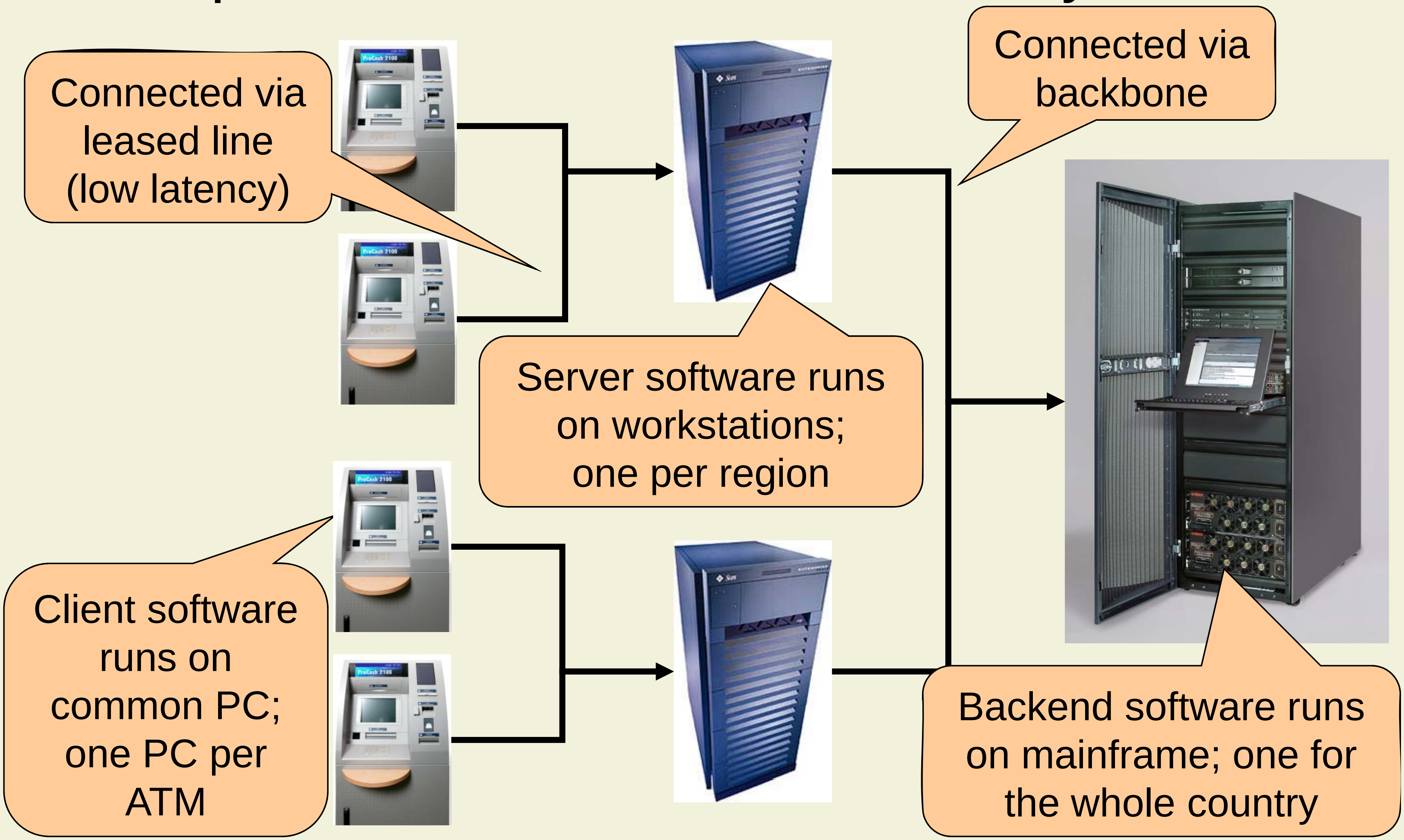
- Identify associations that do not directly map into physical connections
  - How should these associations be implemented?

ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Hardware / Software Mapping Questions

- What is the **connectivity** among physical units?
  - Tree, star, matrix, ring
- What is the appropriate **communication protocol** between the subsystems?
  - Function of required bandwidth, latency and desired reliability, desired quality of service (QoS)
- Is certain **functionality** already **available in hardware**?
- General system **performance** question
  - What is the desired response time?

Example: ATM Machine and Host System



Data Management

- Some objects in the models need to be **persistent**
- Persistency is achieved by **files** and **databases**
- Files
  - Cheap, simple, permanent storage
  - Low level (read, write)
  - Applications must add code to provide suitable level of abstraction
- Database
  - Powerful, easy to port
  - Supports multiple writers and readers

File or Database?

- When should you choose a file?
  - Is the data **voluminous** (bit maps)?
  - Do you have lots of **raw data** (core dump, event trace)?
  - Do you need to keep the data only for a **short time**?
- When should you choose a database?
  - Does the data require access by **multiple users**?
  - Must the data be ported across multiple platforms **(heterogeneous systems)**?
  - Do **multiple application programs** access the data?
  - Does the data **management** require a lot of **infrastructure** (e.g., indexing, transactions)?

Database Management System

- Contains mechanisms for **describing** data, **managing** persistent storage and for providing a **backup** mechanism

- Provides **concurrent access** to the stored data

- Contains information about the data ("meta-data")
  - Also called data schema

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Object-Oriented Databases

- An object-oriented database **supports** all the fundamental **object modeling** concepts
  - Classes, Attributes, Methods, Associations, Inheritance
- Mapping an object model to an OO-database
  - Determine which objects are persistent
  - Perform normal requirement analysis and detailed design
  - Do the mapping specific to commercially available product
- Suitable for **medium-sized** data set, **irregular associations** among objects

Relational Databases

- Data is presented as **two-dimensional tables**
- Tables have a specific number of columns and arbitrary numbers of rows
  - **Primary key**: Combination of attributes that uniquely identify a row in a table
  - **Foreign key**: Reference to a primary key in another table
- SQL is the standard language for defining and manipulating tables
- Suitable for **large** data set, **complex queries** over attributes

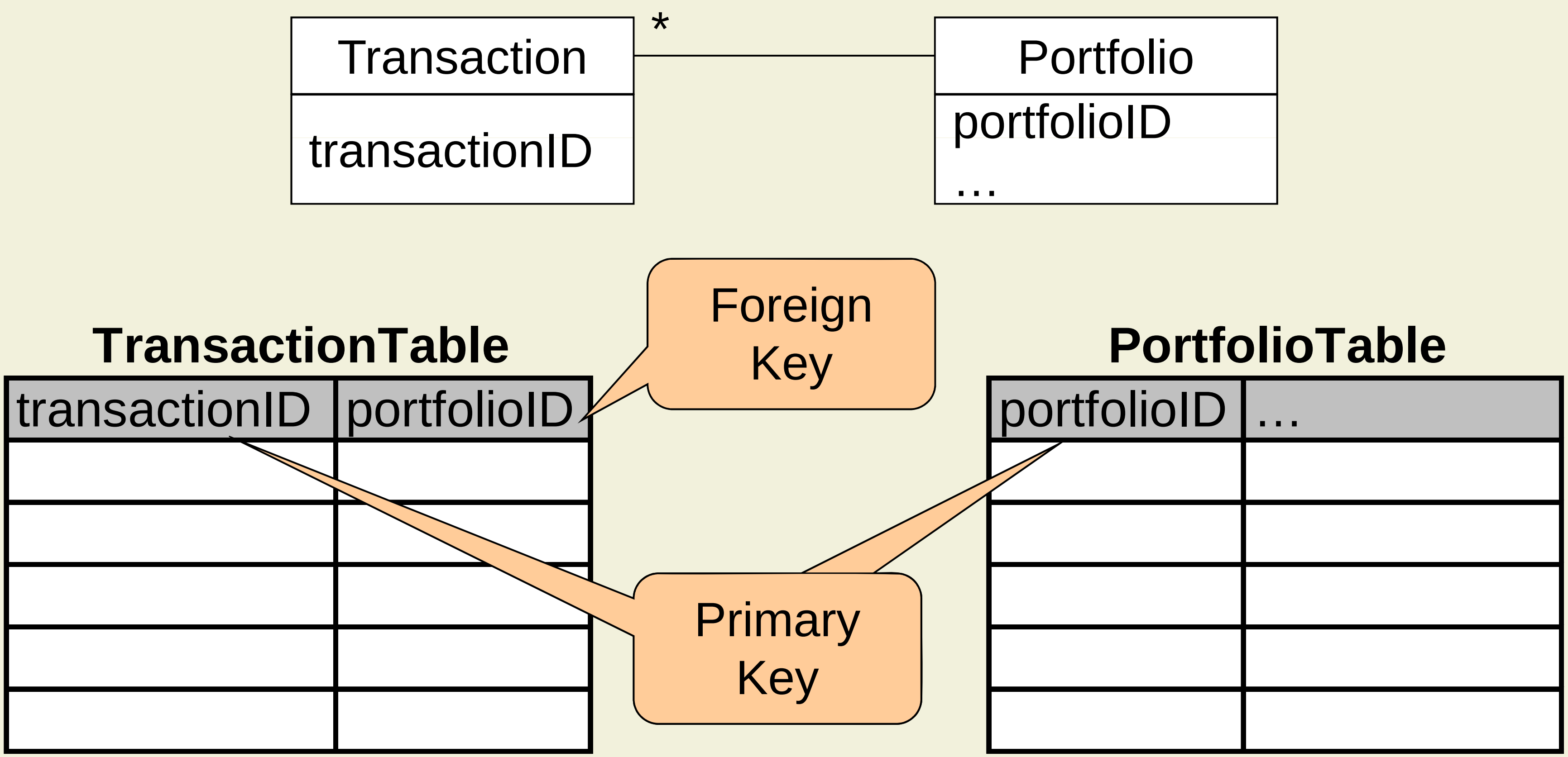
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Mapping an Object Model to a Relational DB

- UML object models can be mapped to relational databases
- UML mappings
  - Each class is mapped to a table
  - Each class attribute is mapped onto a column in the table
  - An instance of a class represents a row in the table
  - A one-to-many association is implemented as foreign key
  - A many-to-many association is mapped into its own table
- Methods are not mapped

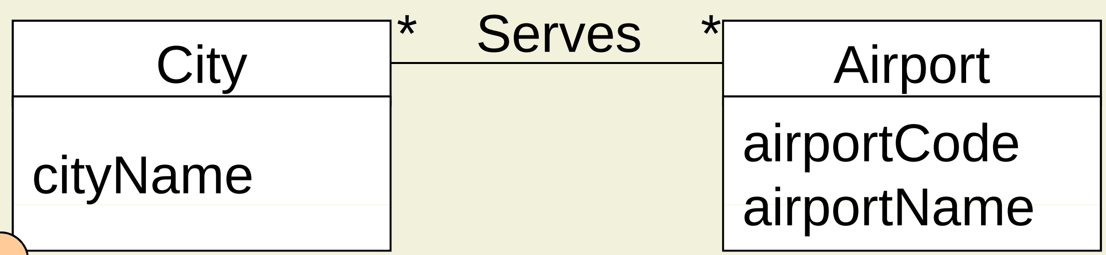
Mapping 1:n and n:1 Associations

- Buried Foreign Keys



Mapping Many-to-Many Associations

- Separate table for association



Primary Key

Separate Table

CityTable

cityName
Houston
Albany
Munich
Hamburg

AirportTable

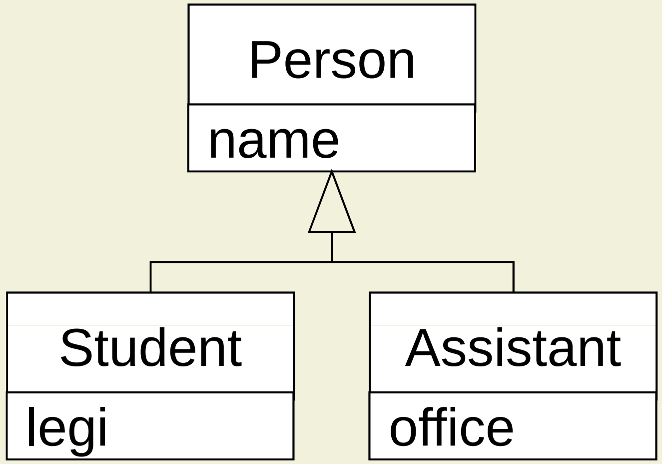
airportCode	airportName
IAH	Intercontinental
HOU	Hobby
ALB	Albany County
MUC	Munich Airport
HAM	Hamburg Airport

ServesTable

cityName	airportCode
Houston	IAH
Houston	HOU
Albany	ALB
Munich	MUC
Hamburg	HAM

Mapping Inheritance

- Option 1: separate table



StudentTable

id	legi
56	123456

AssistantTable

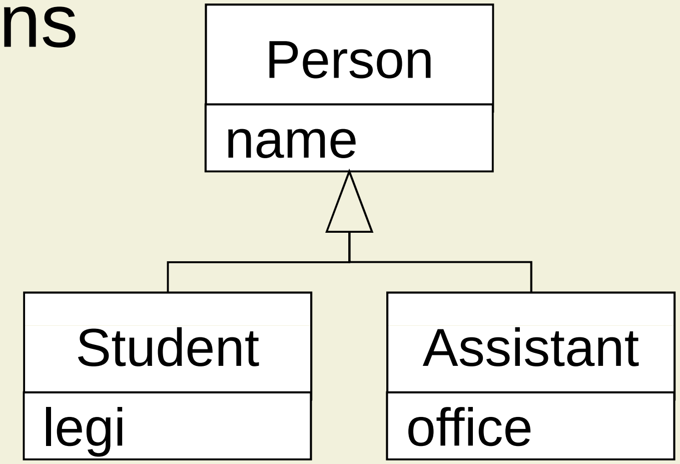
id	office
79	RZ F02

PersonTable

id	name
56	Urs
79	Sile

Mapping Inheritance (cont‘d)

- Option 2: duplicating columns



StudentTable

id	legi	name
56	123456	Urs

AssistantTable

id	office	name
79	RZ F02	Sile

Separate Tables vs. Duplicated Columns

- Trade-off between modifiability and response time
  - How likely is a change of the superclass?
  - What are the performance requirements for queries?
- Separate table mapping
  - Pro: Adding attributes to the superclass is easy (adding a column to the superclass table)
  - Con: Searching for the attributes of an object requires a join operation
- Duplicated columns
  - Con: Modifying the database schema is more complex and error-prone
  - Pro: Individual objects are not fragmented across a number of tables (faster queries)

Data Management Questions

- Should the data be **distributed**?
- Should the database be **extensible**?
- **How often** is the database **accessed**?
- What is the expected **request rate**? In the worst case?
- What is the **size of** typical and worst case **requests**?
- Does the data need to be **archived**?
- Does the system design try to hide the location of the databases **(location transparency)**?
- Is there a need for a **single interface** to access the data?
- What is the **query format**?
- Should the database be **relational** or **object-oriented**?

Boundary Conditions

- Most of the system design effort is concerned with the steady-state behavior described in the analysis phase
- Additional **administration use cases** describe:
- **Initialization** ("startup use cases”)
- **Termination** ("termination use cases")
  - What resources are cleaned up and which systems are notified upon termination
- **Failure** (“failure use cases”)
  - Many possible causes: Bugs, errors, external problems
  - Good system design foresees fatal failures

Boundary Condition Questions

- Initialization
  - How does the system start up?
  - What data needs to be accessed at startup time?
  - What services have to be registered?
  - What does the user interface do at start up time?
  - How does it present itself to the user?
- Termination
  - Are single subsystems allowed to terminate?
  - Are other subsystems notified if a single subsystem terminates?
  - How are local updates communicated to the database?

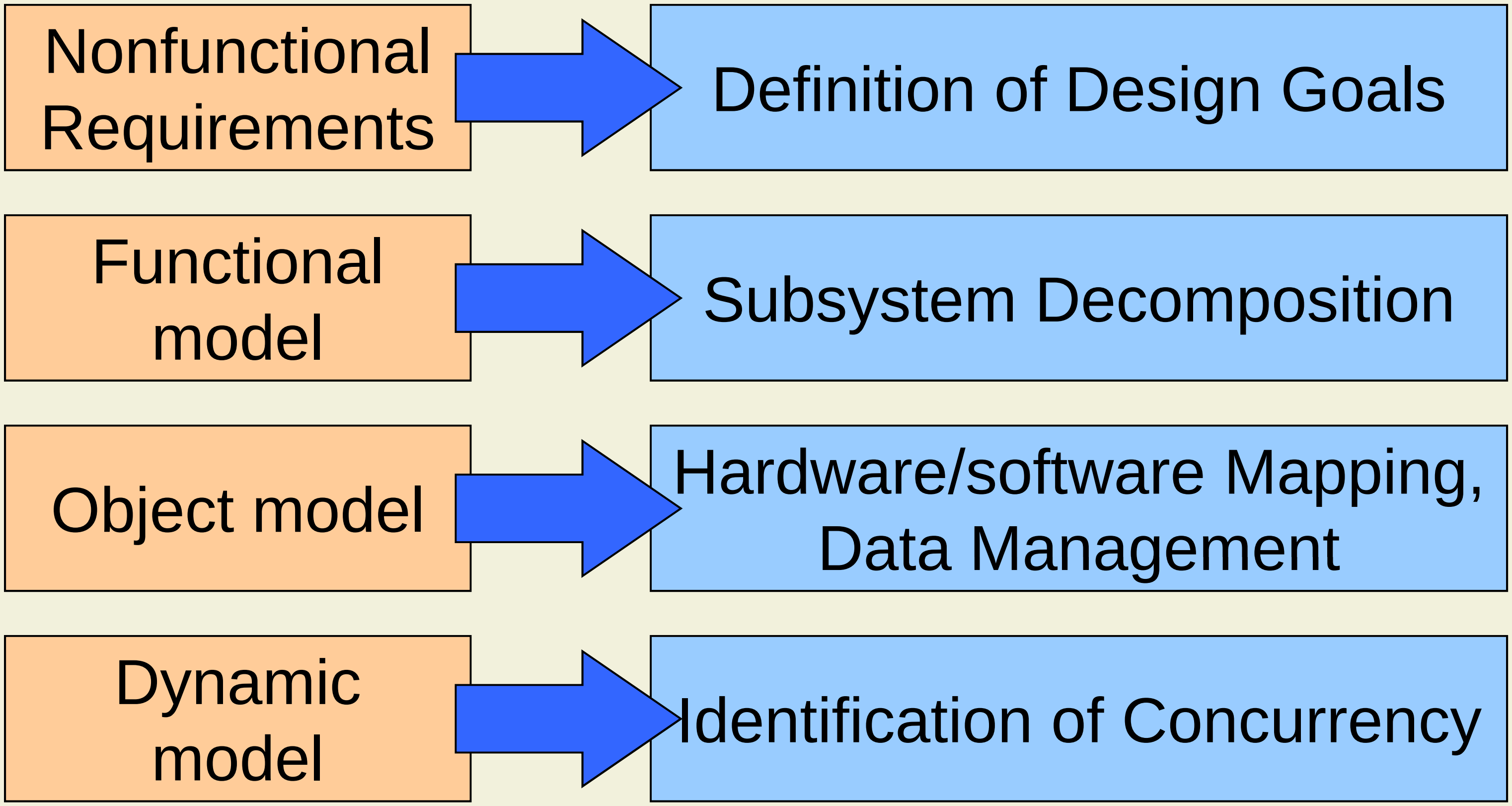
Boundary Condition Questions (cont'd)

- Failure
  - How does the system behave when a node or communication link fails? Are there backup communication links?
  - How does the system recover from failure? Is this different from initialization?

Modeling Boundary Conditions

- **Boundary conditions** are best **modeled as use cases** with actors and objects
- Actor: often the system administrator
- Interesting use cases:
  - Start up of a subsystem
  - Start up of the full system
  - Termination of a subsystem
  - Error in a subsystem or component, failure of a subsystem or component

Influences from Requirements Analysis



- Finally: The subsystem decomposition influences boundary conditions

Summary: System Design

- **Design goals definition**
  - Describes and prioritizes the qualities that are important for the system
- **Subsystem decomposition**
  - Decomposes the overall system into manageable parts by using the principles of cohesion and coherence
- **Architectural style**
  - A pattern of a typical subsystem decomposition
- **Software architecture**
  - An instance of an architectural style