Design Goals & System Decomposition

Software Engineering I
Lecture 7

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

Where are we?

- We have covered Ch 1 - 3
- We are in the middle of Chapter 4
  - Functional modeling: Read again Ch 2, pp. 46 - 51
  - Structural modeling: Read again Ch 2, pp.52 - 59
- From use cases to class diagrams
  - Identify participatory objects in flow of events descriptions
    - Exercise: Apply Abbot’s technique to Fig. 5-7, p. 181
  - Identify entity, control and boundary objects
    - Heuristics to find these types: Ch 5, Section 5.4
- Notations for dynamic models:
  - Interaction-, Collaboration-, Statechart-, Activity diagrams
  - Read Ch. 2, pp. 59-67

Design is Difficult

- There are two ways of constructing a software design (Tony Hoare):
  - 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.”
- Corollary (Jostein Gaarder):
  - If our brain would be so simple that we can understand it, we would be too stupid to understand it.

Why is Design so Difficult?

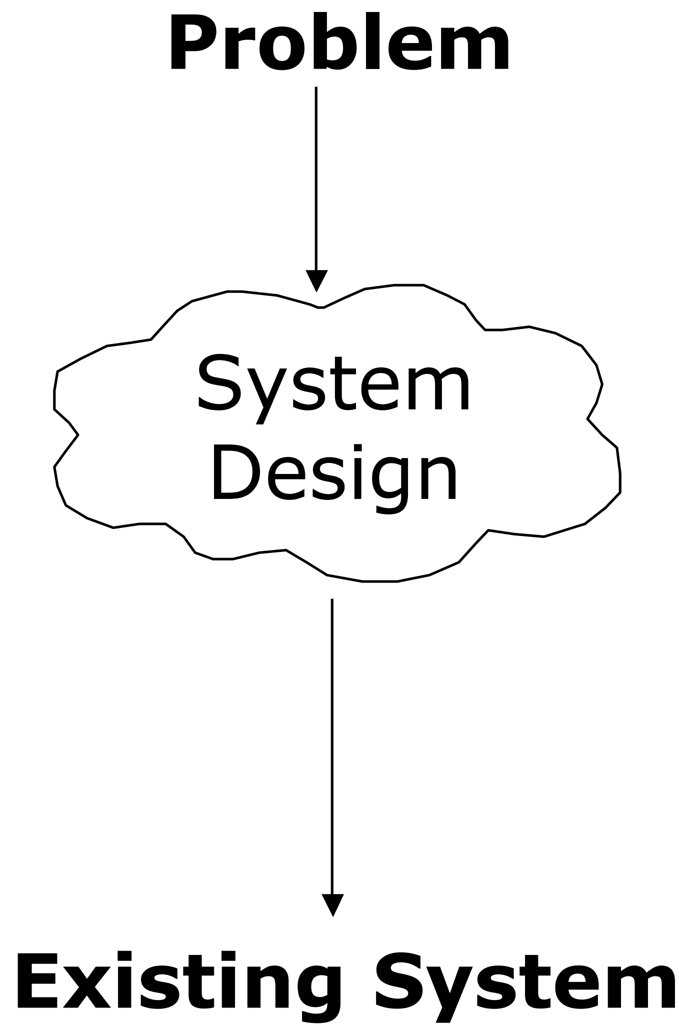
- Analysis: Focuses on the application domain
- Design: Focuses on the solution domain
  - The solution domain is changing very rapidly
    - Halftime knowledge in software engineering: About 3-5 years
    - Cost of hardware rapidly sinking
  - ➢ Design knowledge is a moving target

- Design window: Time in which design decisions have to be made.

The Scope of System Design

- Bridge the gap
  - between a problem and an existing system in a manageable way

- How?
- Use Divide & Conquer:
  1) Identify design goals
  2) Model the new system design as a set of subsystems
  3-8) Address the major design goals.



System Design: Eight Issues

System Design

1. Identify Design Goals
- Additional NFRs
- Trade-offs

2. Subsystem Decomposition
- Layers vs Partitions
- Coherence & Coupling

3. Identify Concurrency
- Identification of Parallelism (Processes, Threads)

4. Hardware/ Software Mapping
- Identification of Nodes
- Special Purpose Systems
- Buy vs Build
- Network Connectivity

5. Persistent Data Management
- Storing Persistent Objects
- Filesystem vs Database

6. Global Resource Handlung
- Access Control
- ACL vs Capabilities
- Security

7. Software Control
- Monolithic
- Event-Driven
- Conc. Processes

8. Boundary Conditions
- Initialization
- Termination
- Failure.

Overview

System Design I (This Lecture)

0. Overview of System Design
1. Design Goals
2. Subsystem Decomposition (identifying subsystems)

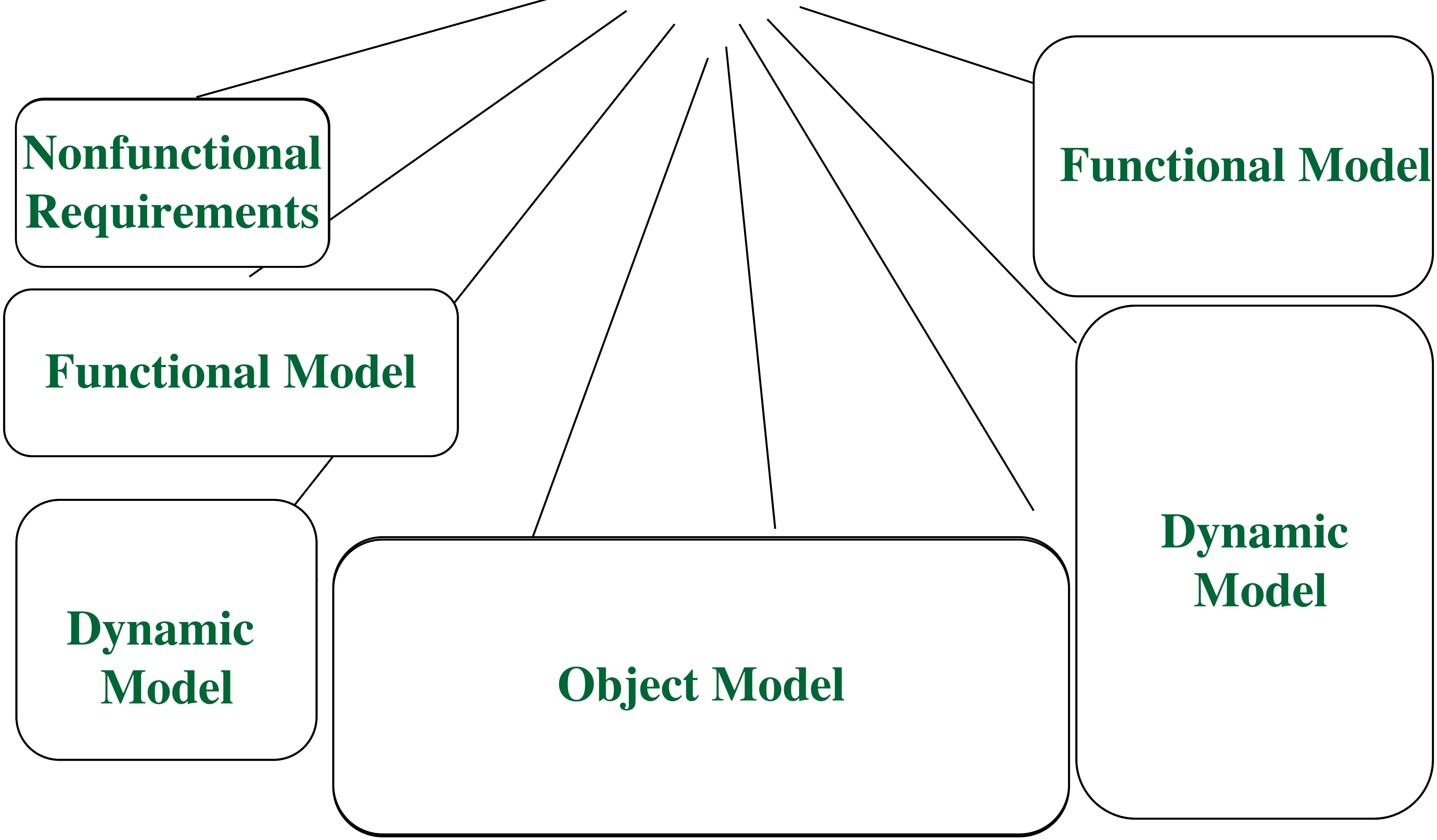
System Design II (Lecture 8:Addressing Design Goals)

3. Concurrency (The more parallelism we can identify the better)
4. Hardware/Software Mapping: Mapping subsystems to processors
5. Persistent Data Management (Storing entity objects)
6. Global Resource Handling and Access Control (Who can access what?)
7. Software Control (Who is in control?)
8. Boundary Conditions (Administrative use cases).

How the Analysis Models influence System Design

- Nonfunctional Requirements
  => Definition of Design Goals
- Functional model
  => Subsystem Decomposition
- Object model
  => Hardware/Software Mapping, Persistent Data Management
- Dynamic model
  => Identification of Concurrency, Global Resource Handling, Software Control
- Finally: Hardware/Software Mapping
  => Boundary conditions

From Analysis to System Design

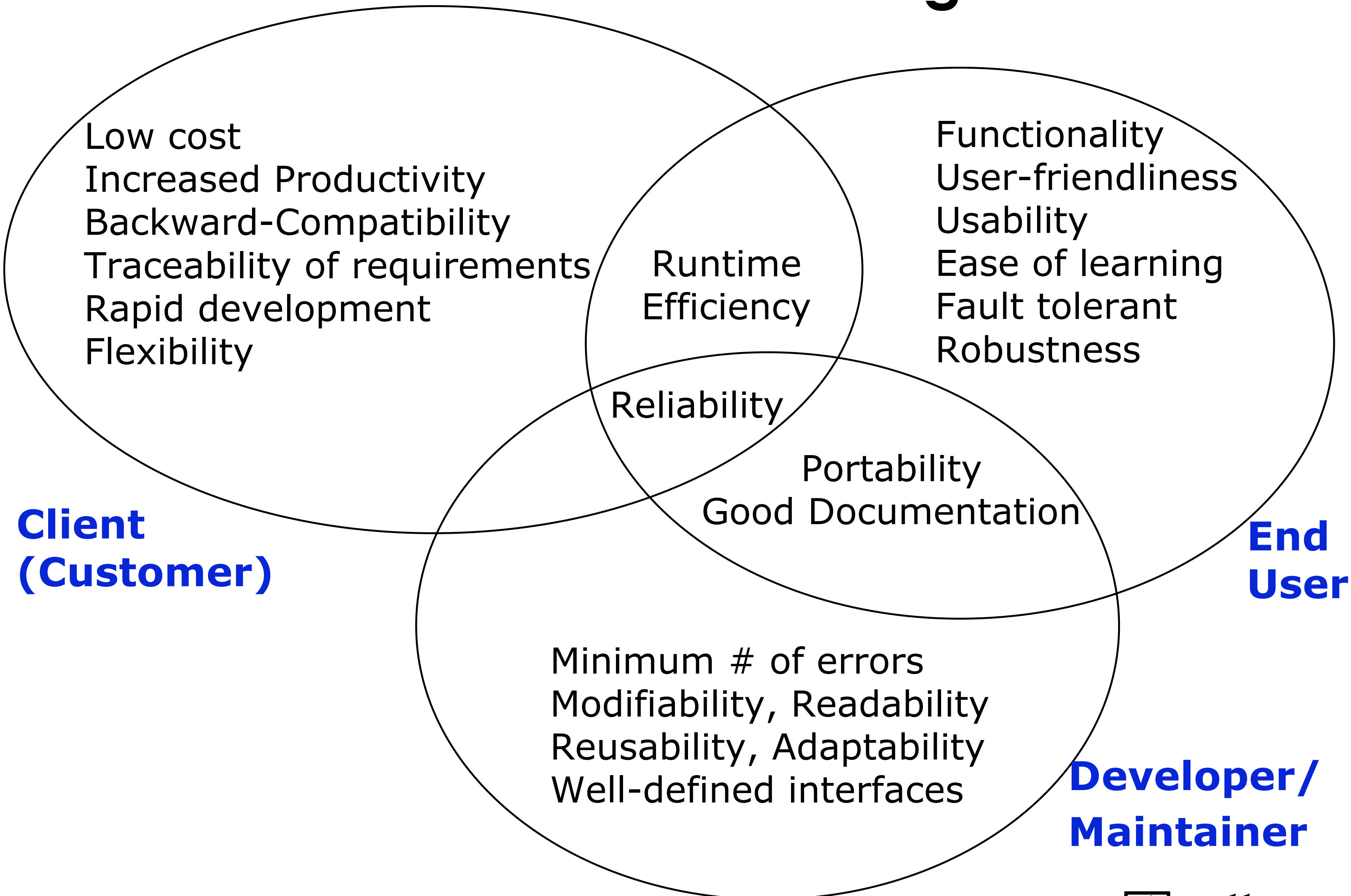


Example of Design Goals

- Reliability
- Modifiability
- Maintainability
- Understandability
- Adaptability
- Reusability
- Efficiency
- Portability
- Traceability of requirements
- Fault tolerance
- Backward-compatibility
- Cost-effectiveness
- Robustness
- High-performance

- Good documentation
- Well-defined interfaces
- User-friendliness
- Reuse of components
- Rapid development
- Minimum number of errors
- Readability
- Ease of learning
- Ease of remembering
- Ease of use
- Increased productivity
- Low-cost
- Flexibility

Stakeholders have different Design Goals



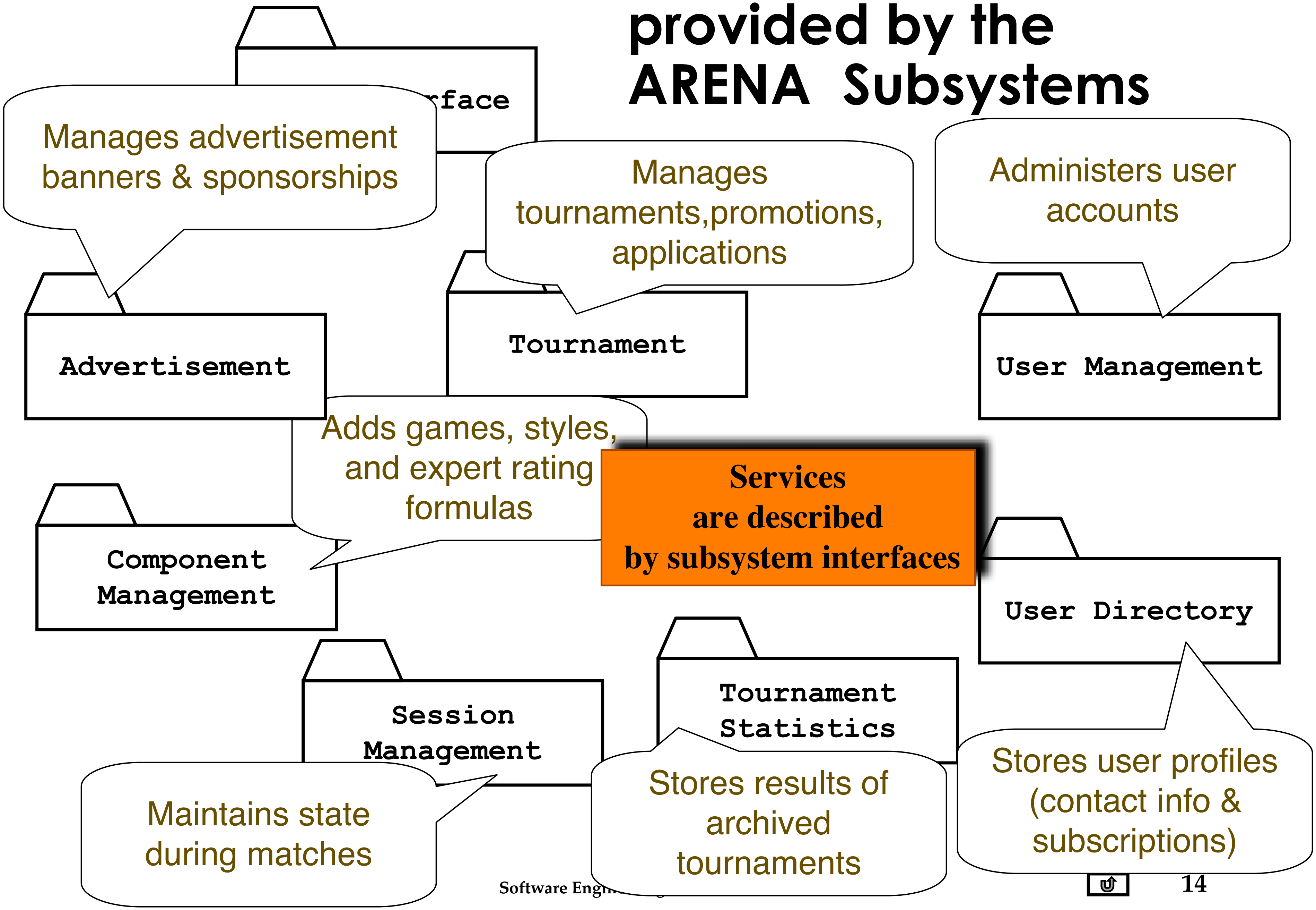
Typical Design Trade-offs

- Functionality v. Usability
- Cost v. Robustness
- Efficiency v. Portability
- Rapid development v. Functionality
- Cost v. Reusability
- Backward Compatibility v. Readability

Subsystem Decomposition

- Subsystem
  - Collection of classes, associations, operations, events and constraints that are closely interrelated with each other
  - The objects and classes from the object model are the "seeds" for the subsystems
  - In UML subsystems are modeled as packages
- Service
  - A set of named operations that share a common purpose
  - The origin ("seed") for services are the use cases from the functional model
  - Services are defined during system design.

Example: Services provided by the ARENA Subsystems



Subsystem Interfaces vs API

- Subsystem interface: Set of fully typed UML operations
  - Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem
  - Refinement of service, should be well-defined and small
  - *Subsystem interfaces are defined during object design*
- Application programmer's interface (API)
  - The API is the specification of the subsystem interface in a specific programming language
  - APIs are defined during implementation
- The terms subsystem interface and API are often confused with each other
  - *The term API should not be used during system design and object design, but only during implementation.*

Example: Notification subsystem

- Service provided by Notification Subsystem
  - LookupChannel()
  - SubscribeToChannel()
  - SendNotice()
  - UnscubscribeFromChannel()

- Subsystem Interface of Notification Subsystem in UML

  Left as an Exercise

- API of Notification Subsystem in Java

  Left as an Exercise

Subsystem Interface Object

- Good design: The subsystem interface object describes *all* the services of the subsystem interface

- Subsystem Interface Object
  - The set of public operations provided by a subsystem

Subsystem Interface Objects can be realized with the Façade pattern (=> lecture on design patterns).

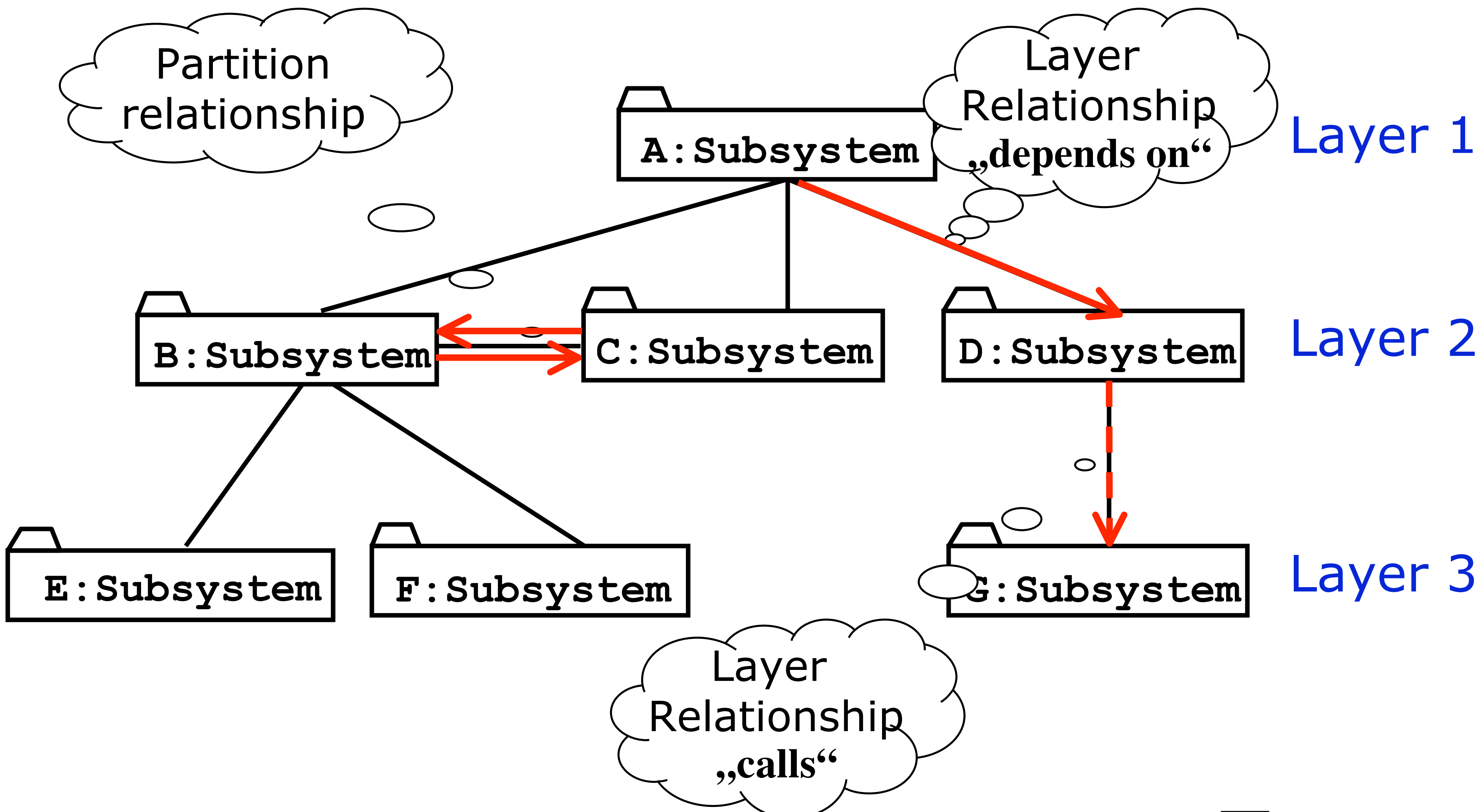
Properties of Subsystems: Layers and Partitions

- A layer is a subsystem that provides a service to another subsystem with the following restrictions:
  - A layer only depends on services from lower layers
  - A layer has no knowledge of higher layers
- A layer can be divided horizontally into several independent subsystems called partitions
  - Partitions provide services to other partitions on the same layer
  - Partitions are also called “weakly coupled” subsystems.

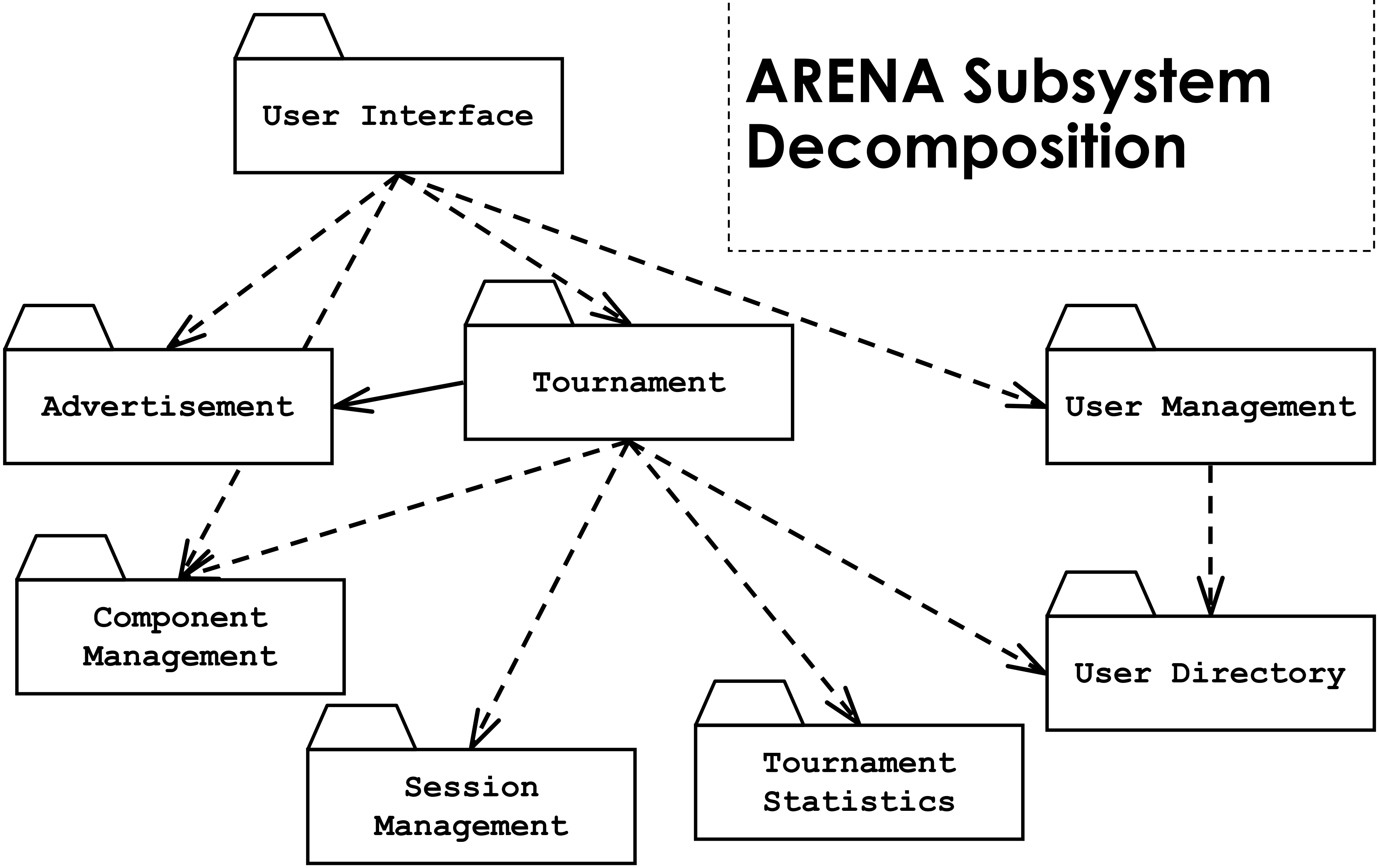
Relationships between Subsystems

- Layer relationships
  - Layer A "depends on" Layer B (compile time property)
    - Example: Build dependencies (make, ant, maven)
  - Layer A "calls" Layer B (runtime property)
    - Example: Client/Server dependency
    - Can the client and server layers run on the same machine?
    - Think about the layers, not about the hardware mapping!
- Partition relationship
  - The subsystems have mutual knowledge about each other
    - A can call services in B, B can call services in A
    - Example: Peer-to-Peer systems
- UML convention:
  - Runtime dependencies are associations with dashed lines
  - Compile time dependencies are associations with solid lines.

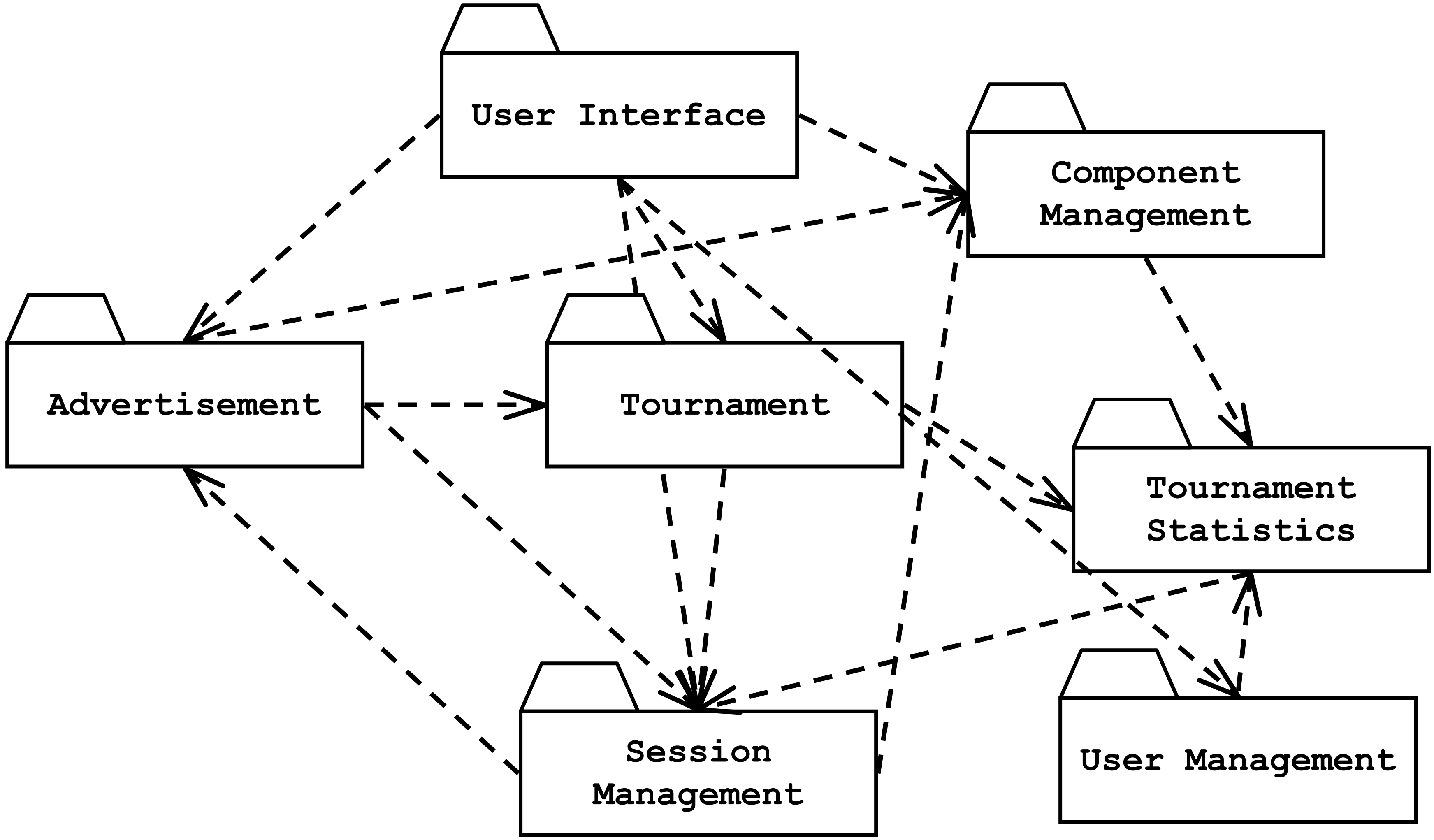
Example of a Subsystem Decomposition



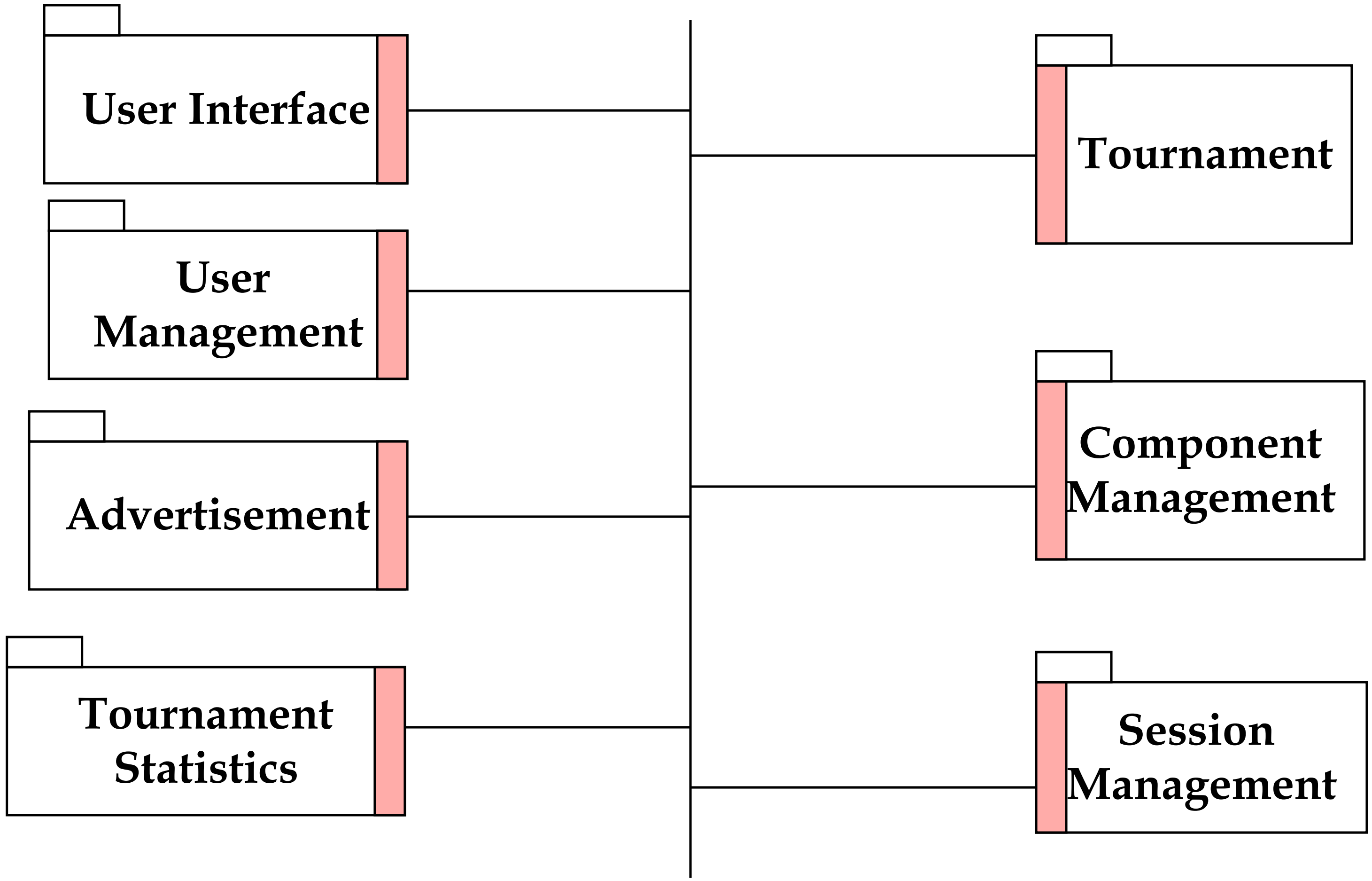
ARENA Subsystem Decomposition

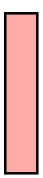


Example of a Bad Subsystem Decomposition



Good Design: The System as set of Interface Objects





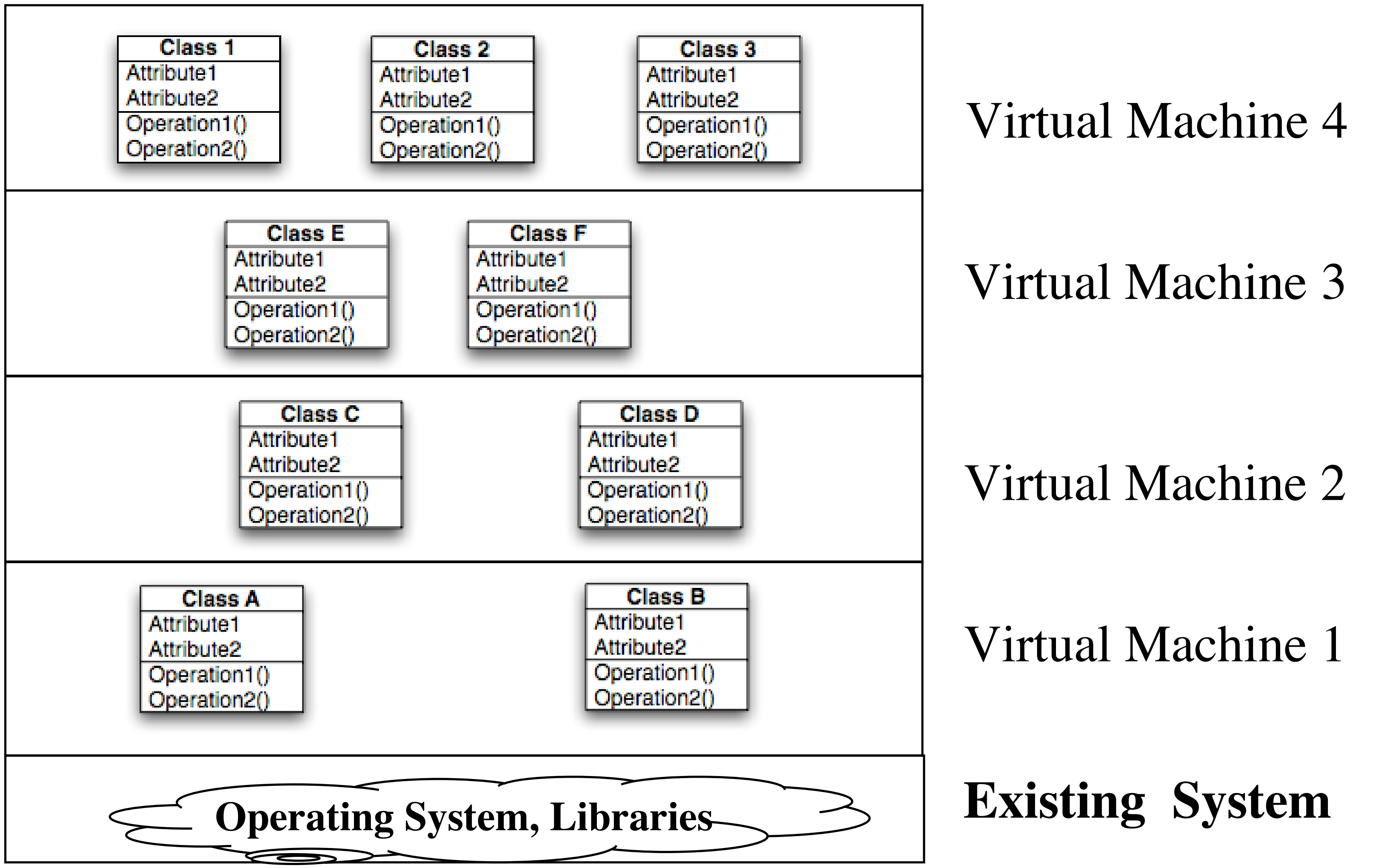
Subsystem Interface Objects

Virtual Machine

- The terms layer and virtual machine can be used interchangeably
  - Also sometimes called “level of abstraction”.
  - A virtual machine is an abstraction that provides a set of attributes and operations
- A virtual machine is a subsystem connected to higher and lower level virtual machines by "provides services for" associations.

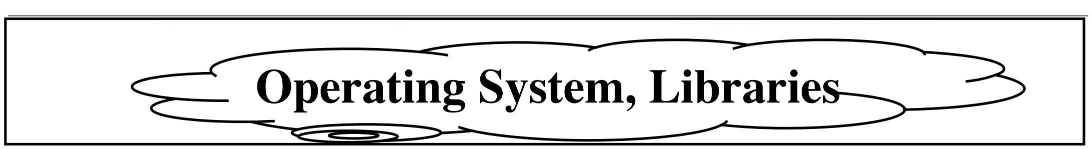
Building Systems as a Set of Virtual Machines

A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines.



Building Systems as a Set of Virtual Machines

A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines.

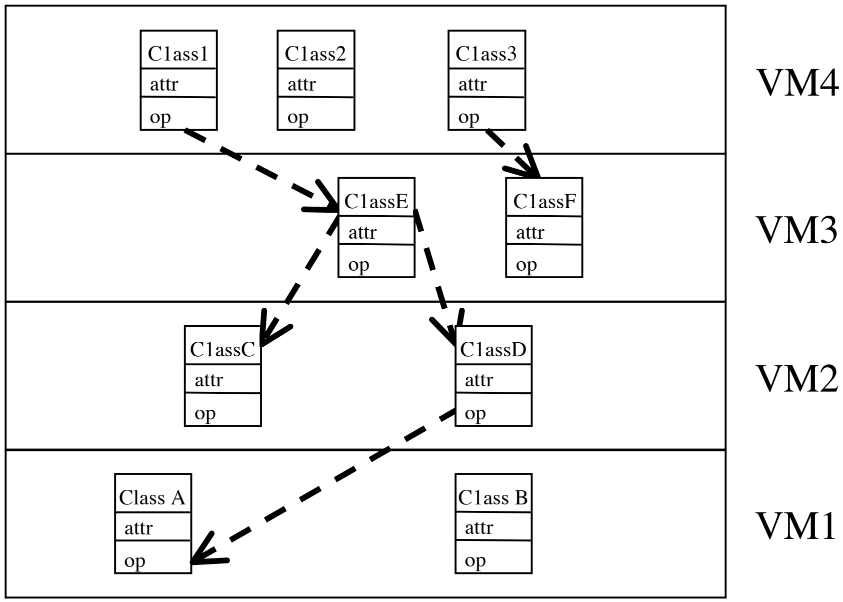


Existing System

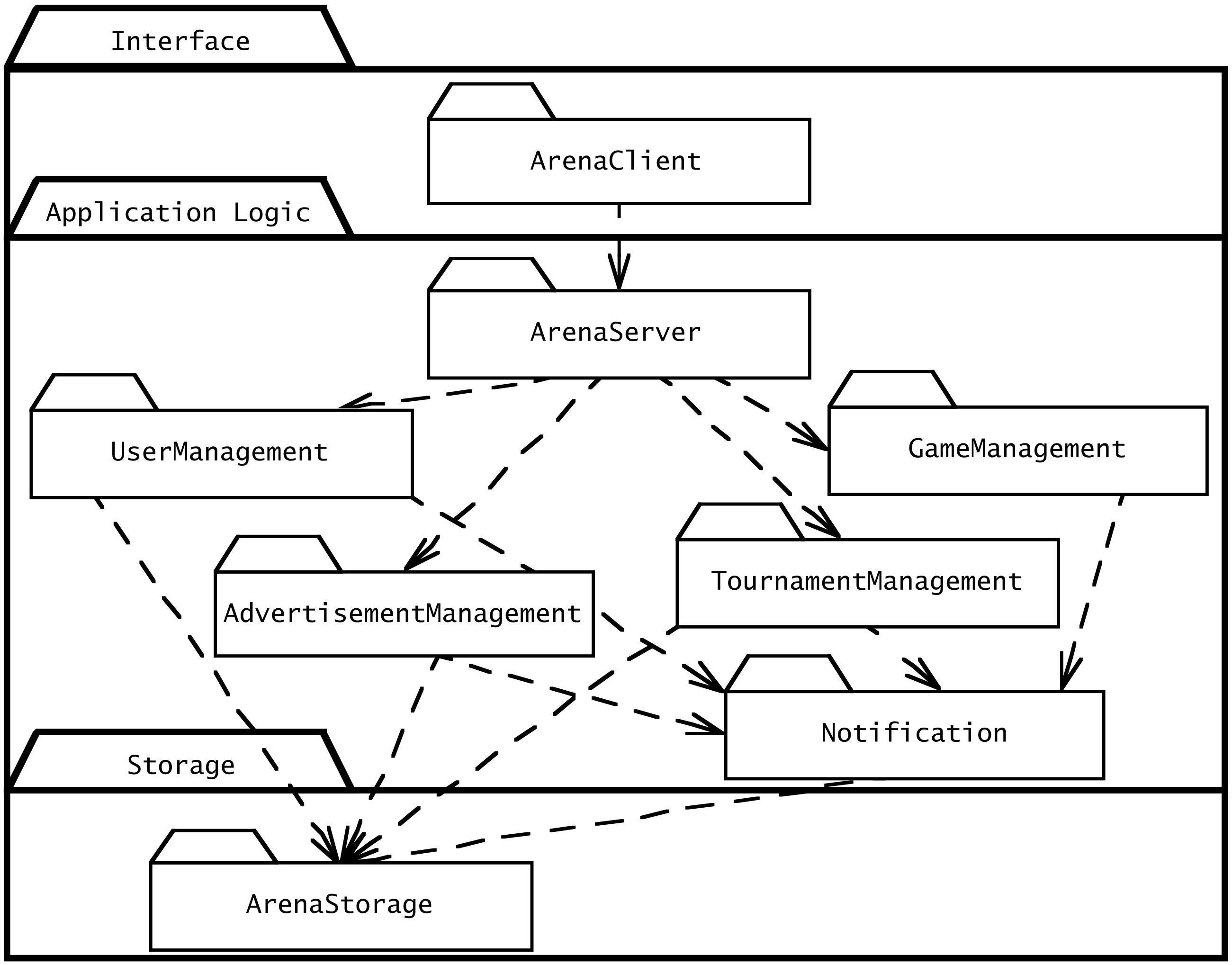
Closed Architecture (Opaque Layering)

- Each virtual machine can only call operations from the layer below

Design goals: Maintainability, flexibility.

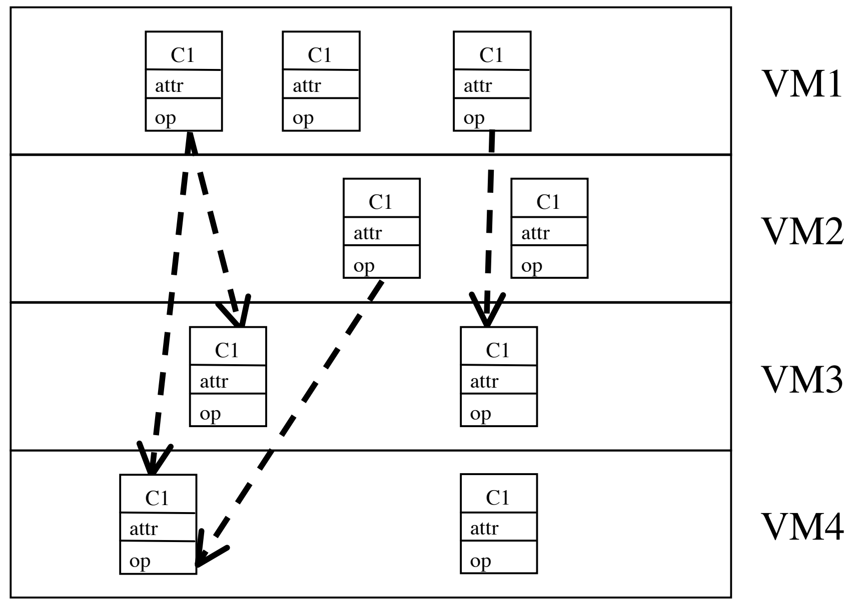


Opaque Layering in ARENA



Open Architecture (Transparent Layering)

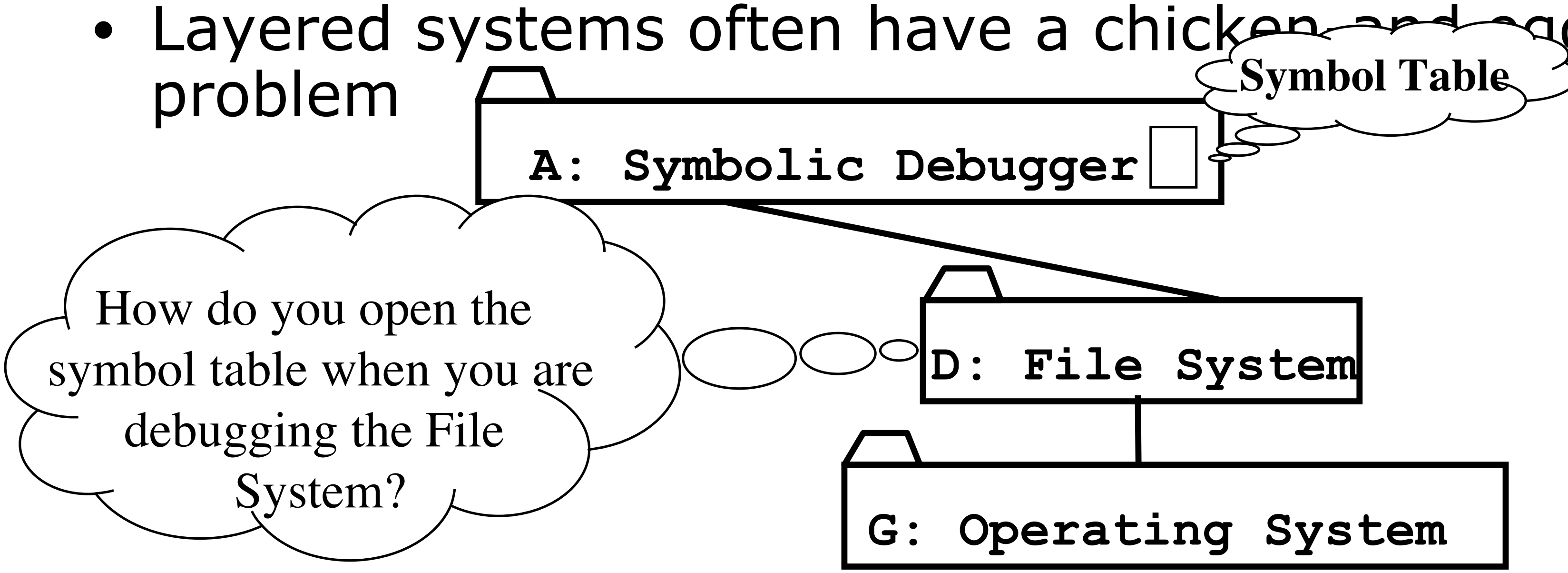
- Each virtual machine can call operations from any layer below



Design goal: Runtime efficiency

Properties of Layered Systems

- Layered systems are hierarchical. This is a desirable design, because the hierarchy reduces complexity
  - low coupling
- Closed architectures are more portable
- Open architectures are more efficient
- Layered systems often have a chicken-and-egg problem



Coupling and Coherence of Subsystems

- Goal: Reduce system complexity while allowing change
- Coherence measures dependency among classes
  - High coherence: The classes in the subsystem perform similar tasks and are related to each other via many associations
  - Low coherence: Lots of miscellaneous and auxiliary classes, almost no associations
- Coupling measures dependency among subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem
  - Low coupling: A change in one subsystem does not affect any other subsystem.

Coupling and Coherence of Subsystems

- Goal: Reduce system complexity while allowing change
- Coherence measures dependency among classes
  - High coherence: The classes in the subsystem perform similar tasks and are related to each other via associations
  - Low coherence: Lots of miscellaneous and auxiliary classes, no associations
- Coupling measures dependency among subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem
  - Low coupling: A change in one subsystem does not affect any other subsystem

Coupling and Coherence of Subsystems

Good Design

- Goal: Reduce system complexity while allowing change
- Coherence measures dependency among classes
  - High coherence: The classes in the subsystem perform similar tasks and are related to each other via associations
  - Low coherence: Lots of miscellaneous and auxiliary classes, no associations
- Coupling measures dependency among subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem
  - Low coupling: A change in one subsystem does not affect any other subsystem

How to achieve high Coherence

- High coherence can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- Questions to ask:
  - Does one subsystem always call the other for the service?
  - Can the subsystems be hierarchically ordered (in layers)?
  - Which of the subsystems call each other for services?
    - Can this be avoided by restructuring the subsystems or changing the subsystem interface?

How to achieve Low Coupling

- Low coupling can be achieved if a calling class does not know about the internals of the called class
- Questions to ask:
  - Does the calling class really have to know any attributes of classes in the lower layers?
  - Is it possible that the calling class calls only operations of the lower level classes?

Principle of information hiding (Parnas)

Additional Readings

- E.W. Dijkstra,
  - “The structure of the T.H.E Multiprogramming system, Communications of the ACM, 18(8), pp. 453-457, 1968
- D. Parnas
  - “On the criteria to be used in decomposing systems into modules, CACM, 15(12), pp. 1053-1058, 1972.

Summary

- System Design
  - Reduce gap between problem and an existing machine
  - Decomposes the overall system into manageable parts
  - Uses the principles of cohesion and coherence
- Design Goals Definition
  - Describes the important system qualities
  - Defines the values against which options are evaluated
- Subsystem Decomposition
  - Results into a set of loosely dependent parts which make up the system
  - Layers and Partitions
  - Virtual machine
  - High coherence and low coupling