Software Design in-the-large

Sandro Morasca and Dario Bertolino

Università degli Studi dell'Insubria

Dipartimento di Scienze Teoriche e Applicate

Via Ottorino Rossi 9 – Padiglione Rossi

21100 Varese

{sandro.morasca,dario.bertolino}@uninsubria.it

Design Goal

- Goal: The final goal of the design phase is to master the complexity of the problem of developing a software product

- How: Decompose the entire problem to solve into smaller, more manageable problems, so that
  - the “sum of the complexities” of the smaller problems is lower than the “complexity” of the entire problem

Specific Objectives: Examples

➢ Basic Concepts
Mechanisms
Styles

- Design for change
  - anticipate likely changes
  - do not concentrate on today's needs, think of the possible evolution
    - the case of evolutionary prototyping
- Program family
  - think of a program as a member of a family

Likely Changes

- Changes in algorithms
  - from bubblesort to quicksort
- Changes in data structures
  - folk data: 17% of maintenance costs
- Changes in the underlying abstract machine
  - hardware peripherals, OS, DBMS, …
    - new releases, portability problems
- Changes in the environment (e.g., EURO)
- Changes due to development strategy
  - evolutionary prototype

Program Family

- Think of the program and all of its variants as a member of a family
- The goal is to design the whole family, not each individual member of the family separately

Program Family: an Example

➢ Basic Concepts
Mechanisms
Styles

- A facility reservation system
  - for hotels: reserve rooms, restaurant, conference space, …, equipment (video beams, overhead projectors, …)
  - for a university
    - many functionalities are similar, some are different (e.g., facilities may be free of charge or not)

Design Result

- The design in-the-large phase produces the software architecture (or software design)

- *The architecture of a software system defines the system in terms of computational **components** and **interactions** among those components. (Garlan&Shaw1996)*

Components and Interactions

➢ Basic Concepts
Mechanisms
Styles

- Components and interactions can be defined
  - at two different levels of abstractions
  - from two different perspectives

- Mechanisms
  - What are the constituents and how are they aggregated and related?
- Styles
  - What kinds of software architecture can be used?

Mechanisms

Basic Concepts
➢ Mechanisms
Styles

- What are the modules?
- What is their interface?
- What are the useful relations among modules?

- Method issue
  - What are the criteria to decompose systems into modules?
- Documentation
  - How to document the catalog of modules and relations?

Style

Basic Concepts
➢ Mechanisms
Styles

- *Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components can be simple and familiar, such as procedure call and shared variable access. But they can be complex and semantically rich, such as client-server protocols, asynchronous event multicast, and piped streams. (Garlan&Shaw 1996)*

Mechanisms vs. Styles

- The mechanisms describe how an architecture is constructed
  - a car body as doors, hood, hinges, ...
  - constituents of the transmission system
- The style is what characterizes an architecture wrt to another
  - coupe vs van vs station wagon
- They are two VIEWS of the same world
- The distinction can be fuzzy

Mechanisms vs. Styles

- At each level one should be allowed to reason about the architecture and about properties of the system
- Both levels provide a mostly “static” description of the architecture

Key Design Concepts and Principles

- Key design concepts and design principles include:
  - Decomposition
  - Abstraction
  - Information Hiding
  - Modularity
  - Extensibility
  - Virtual Machine Structuring
  - Hierarchy
  - Program Families and Subsets
- Main goal of these concepts and principles is to:
  - Manage software system complexity
  - Improve software quality factors
  - Facilitate systematic reuse

Modules

- A module is a part of a system that provides a set of services to other modules
- Services are computational elements that other modules may use

Interfaces

Basic Concepts
➢ Mechanisms
Styles

- The set of services provided by a module (exported) constitutes the module's interface
- The interface defines a contract between the module and its users
- A module consists of its interface and its body (implementation, secrets)
- Users only know a module through its interface

Relations

- USES
  - a module uses the services exported by another
- IS_COMPONENT_OF
  - describes the aggregation of modules into higher level modules
- INHERITS
  - for object-oriented systems

Relations

- Let S be a set of modules

$$S = \{M_1, M_2, . . ., M_n\}$$

- A binary relation r on S is a subset of

$$S \times S$$

- If M_i and M_j are in S, $<M_i, M_j> \in r$ can be written as $M_i \; r \; M_j$

Relations

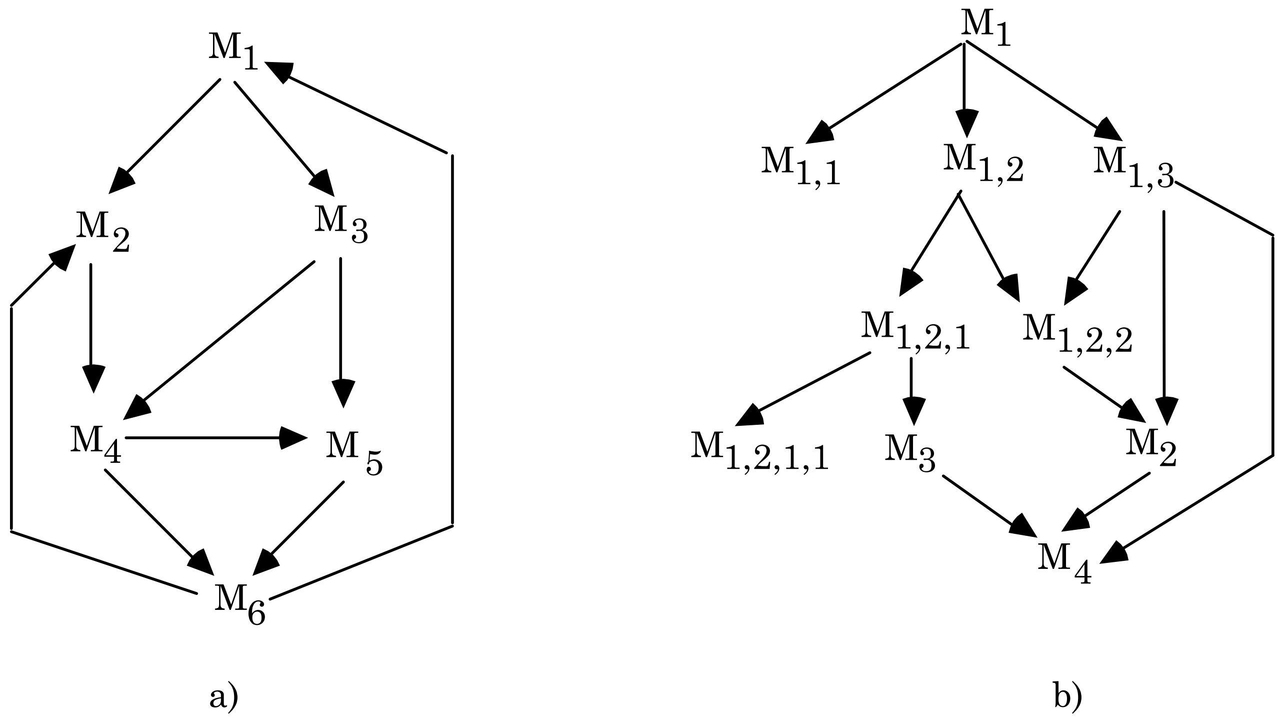
- Transitive closure r^+ of r
  - $M_i\ r^+\ M_j$ <u>iff</u>
    - $M_i\ r\ M_j$ or $\exists\ M_k$ in S such that $M_i\ r\ M_k$
    - and $M_k\ r^+\ M_j$

(We assume that our relations are irreflexive)

- r is a hierarchy iff for all M_i, M_j
  - $M_i\ r^+\ M_j \Rightarrow \neg M_j\ r^+\ M_i$

USES

- Relations can be represented as a graph
- A hierarchy is a DAG (Directed Acyclic Graph)



USES

- A uses B
  - A can access the services exported by B through its interface
  - it is “statically” defined
  - A depends on B to provide its services
    - example: A calls a routine exported by B
- A is a client of B
- A depends on B
  - B’s quality affects A’s quality

IS_COMPONENT_OF

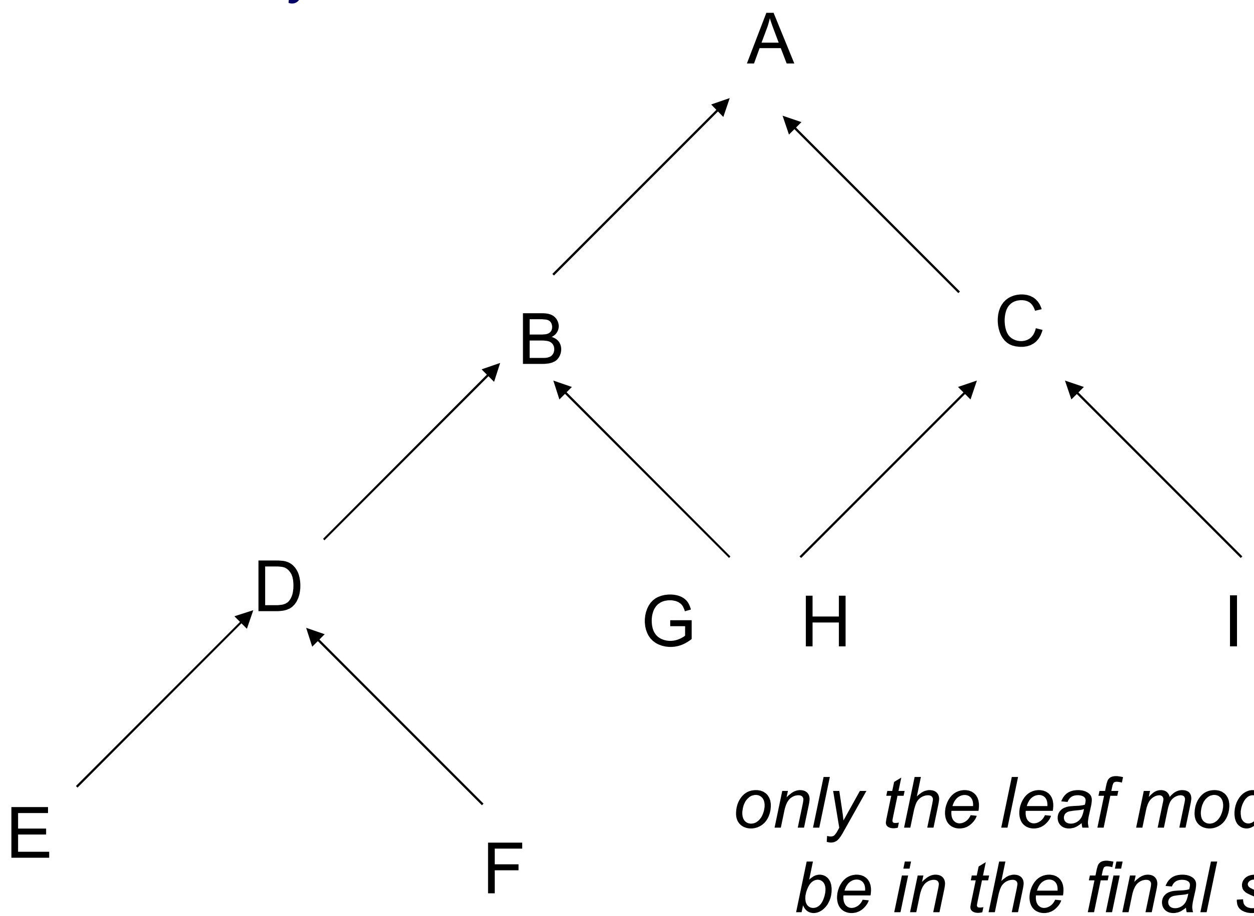
- Used to describe a higher level module as made up of a number of lower level modules
- A IS_COMPONENT_OF B
  - B consists of several modules, of which one is A
- B COMPRISES A (inverse relationship)
- $M_Z=\{M_k | M_k \in S \wedge M_k$ IS_COMPONENT_OF $Z\}$

  we say that M_Z IMPLEMENTS Z

IS_COMPONENT_OF

Basic Concepts
➢ **Mechanisms**
Styles

- *A hierarchy*



only the leaf modules will be in the final system

The INHERITS_FROM Relation

- If the system is developed in an object-oriented style, the inheritance relation allows a component to extend another

- An heir can access (some) of the secrets of its ancestor
  - components are more strongly coupled via INHERITS_FROM than via USES

Design Principles

Basic Concepts
➢ **Mechanisms**
Styles

- How to identify modules?
- How to define module interfaces?
- How to define USE relations?

How to Select Modules

- A module is a self contained unit
- USE interconnections with other modules should be minimized
- PRINCIPLE:
  - maximize cohesion and minimize coupling

How to Select Modules & Interfaces

- Distinguish between what a module does for others and how it does that (its secrets)
- Minimize flow information to clients to maximize modifiability
- The interface is a contract with clients and must be stable
- GOLDEN PRINCIPLE: information hiding (Parnas 1974)
  - define what you wish to hide and design a module around it

Conclusions

- A module is a logical unit
- It is a firewall around its secrets
- Secrets are encapsulated and protected
- It filters access to its internals through the interface
- If changeable parts are in the secret part, their change does not affect clients

Example

- A table on which one can insert, delete, and print entries in some order (e.g., alphabetical)
- Put INSERT, DELETE and PRINT are in the interface
  - the data structure can be freely changed
  - the policy (keep ordered or order prior to printing) can be freely changed

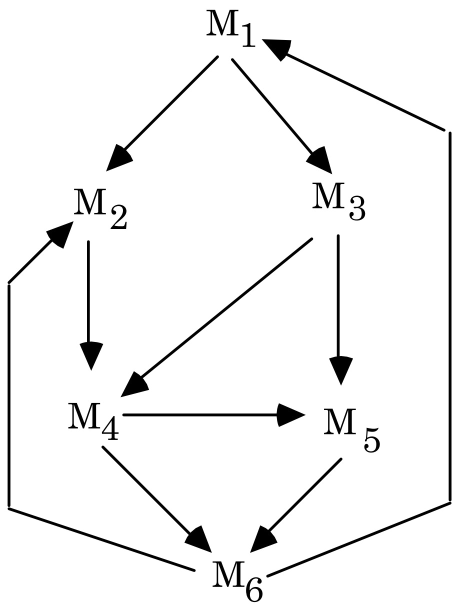
How to Define USES

Basic Concepts
➢ Mechanisms
Styles

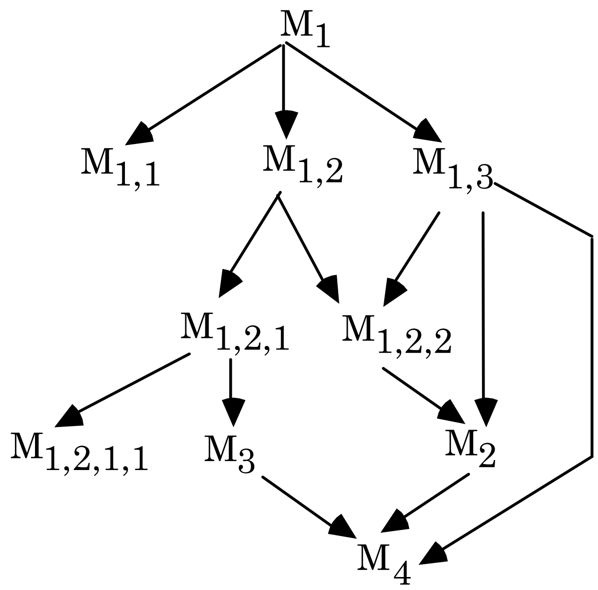
- Make it a hierarchy
  - easier to understand
    - can “read” the DAG from the leaves up
  - easier to verify and develop hierarchically
    - if it is not a hierarchy, we may end up with a system in which nothing works until everything works
- The hierarchy defines a system through “abstraction levels”

Abstraction Levels

Basic Concepts
➢ **Mechanisms**
Styles



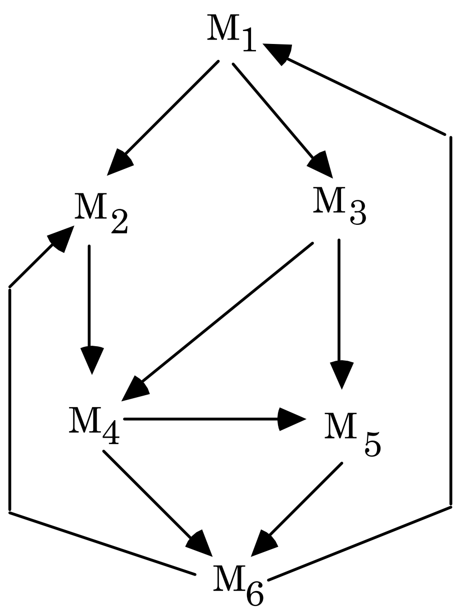
a)



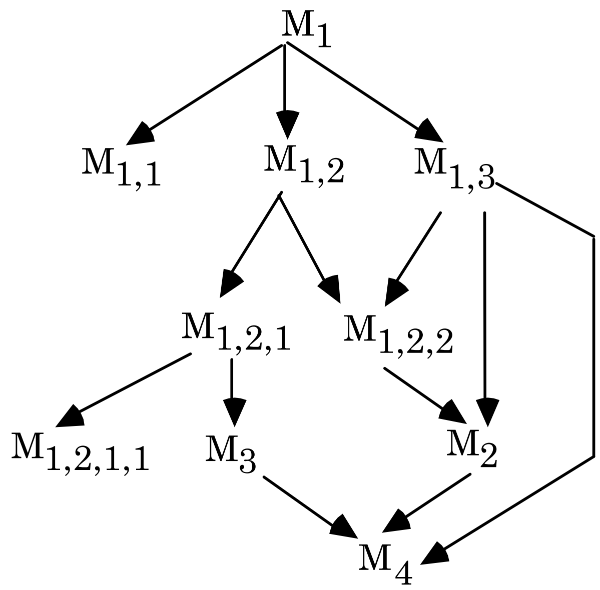
b)

Abstraction Levels

Basic Concepts
➢ **Mechanisms**
Styles



a)



b)

Object-oriented Design

Basic Concepts
➢ Mechanisms
Styles

- The module (class) is itself a resource
  - it is used by others to generate instances
- Introduces the *inheritance* relation, to factor a common part in a component
  - see the case of a program family
- Changes (variations) are deltas defined in the subcomponents
- Inheritance adds further interdependencies among modules

Design Styles

- Shared understanding of common design forms is typical of mature engineering fields
- Shared vocabulary of design idioms is codified in engineering handbooks
- Software is going in this direction
  - but there is less maturity

Components and Connectors

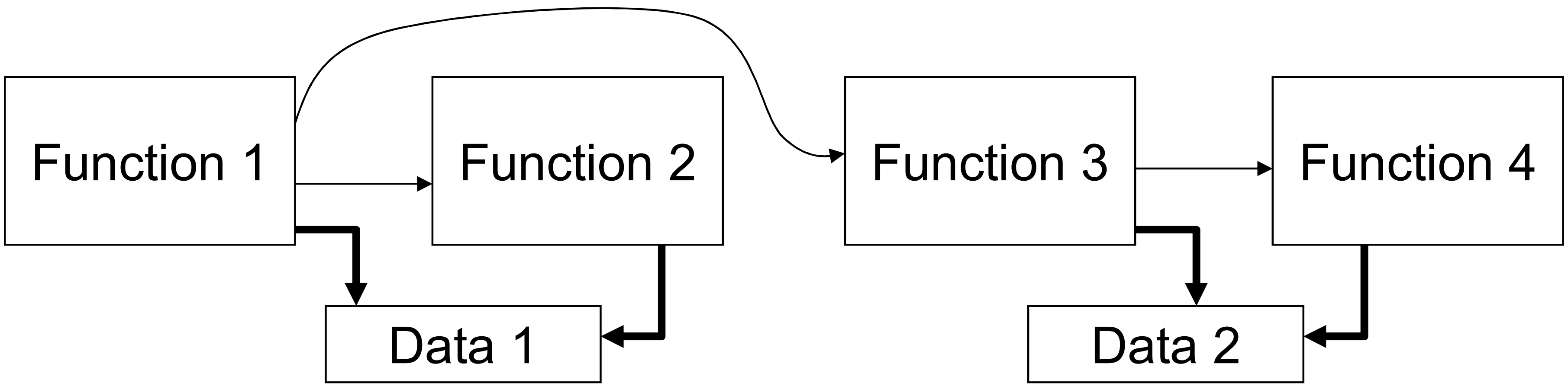
Basic Concepts
Mechanisms
➢ Styles

- Components
  - clients
  - servers
  - filters
  - layers
  - databases
  - ...
- Connectors
  - procedure call
  - event broadcast
  - database protocols
  - pipes
  - ...

A Functional Architecture

Basic Concepts
Mechanisms
➢ Styles

- The system is decomposed into abstract operations
- Operations know (and name) each other
- Connectors = operation call/return
- Additional connectors via shared data



A Functional Architecture

Basic Concepts
Mechanisms
➢ Styles

- “Traditional” system development
  - functions are subroutines of monolithic programs
  - data are “common” data among the routines
- Object-oriented system development
  - functions are methods of a class
  - data are the data of the class

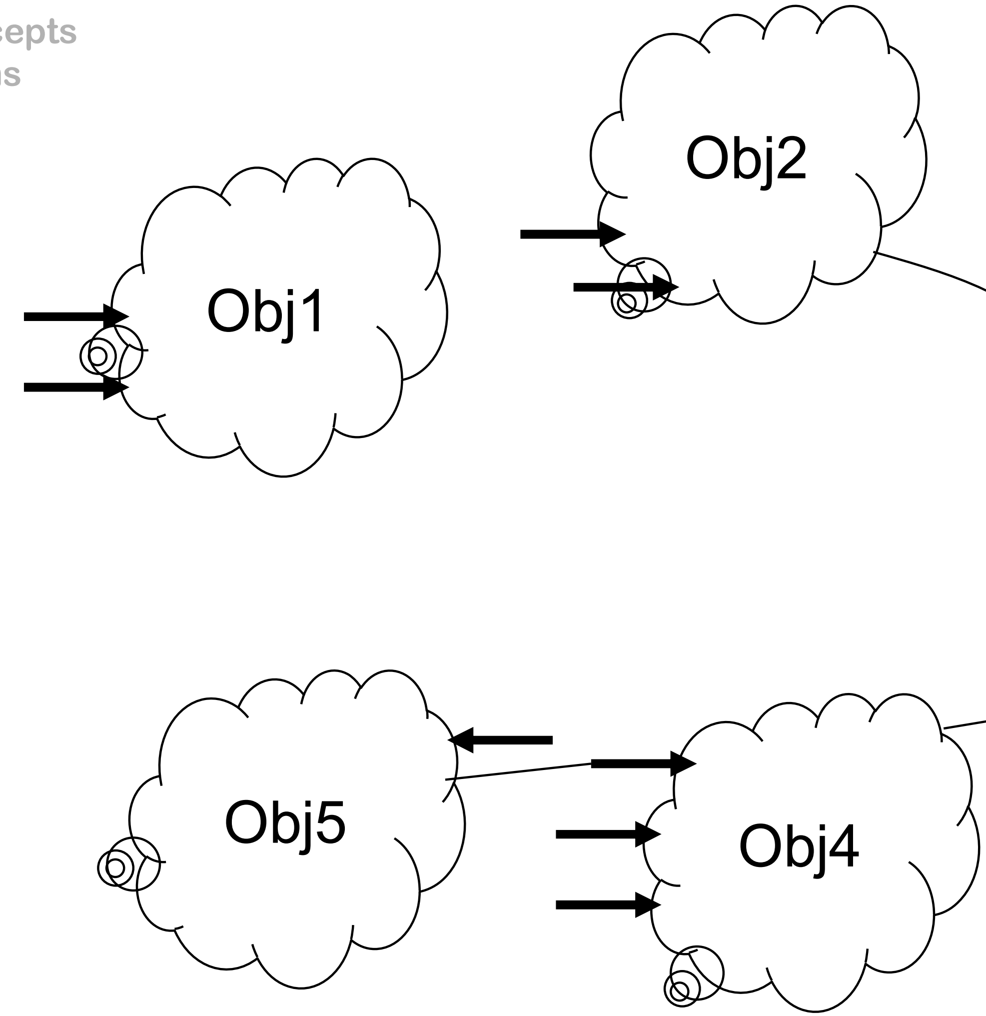
A Functional Architecture

calls from functions to functions

use of common data

```
public class SetOfIntegers
{ private final static int SIZE = 10;
  private static int n;
  private static int list[] = new int [SIZE];
  public static void insert(int number)
  { if (!isFull() && (!belongs(number)))
    { list[n] = number;
      n++; } }
  public static boolean belongs(int number)
  { return search(number) != -1; }
  private static int cardinality() { return n; }
  ...
  public static void main()
  { number = read(); //suppose a read method exists
    insert( number );
    number = read(); insert( number );
    number = read(); delete( number );
  }
}
```

An Object-oriented Architecture

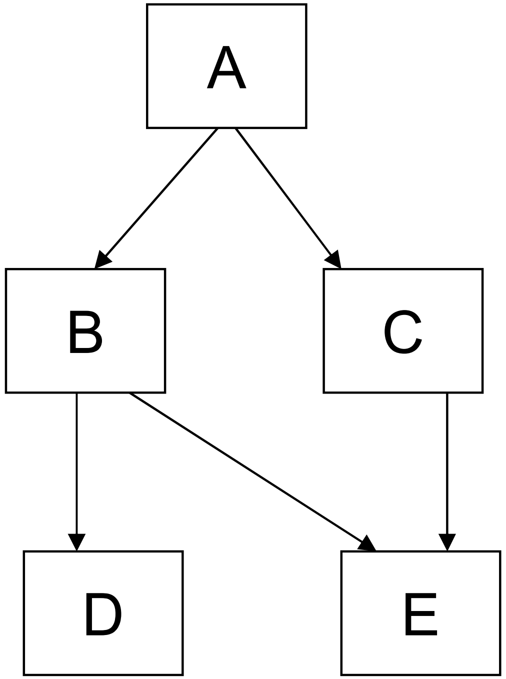


the run-time view

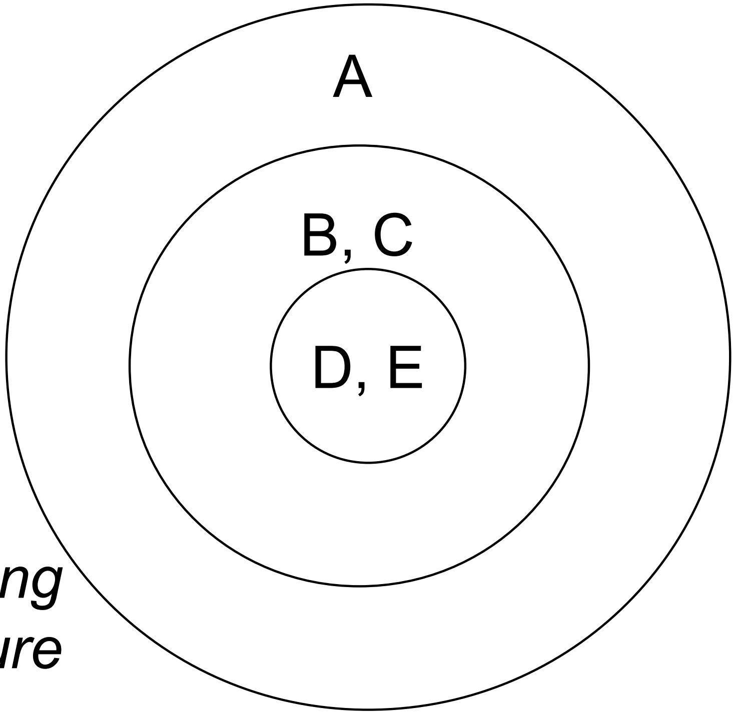
Objects know (and name) each other

Layered System

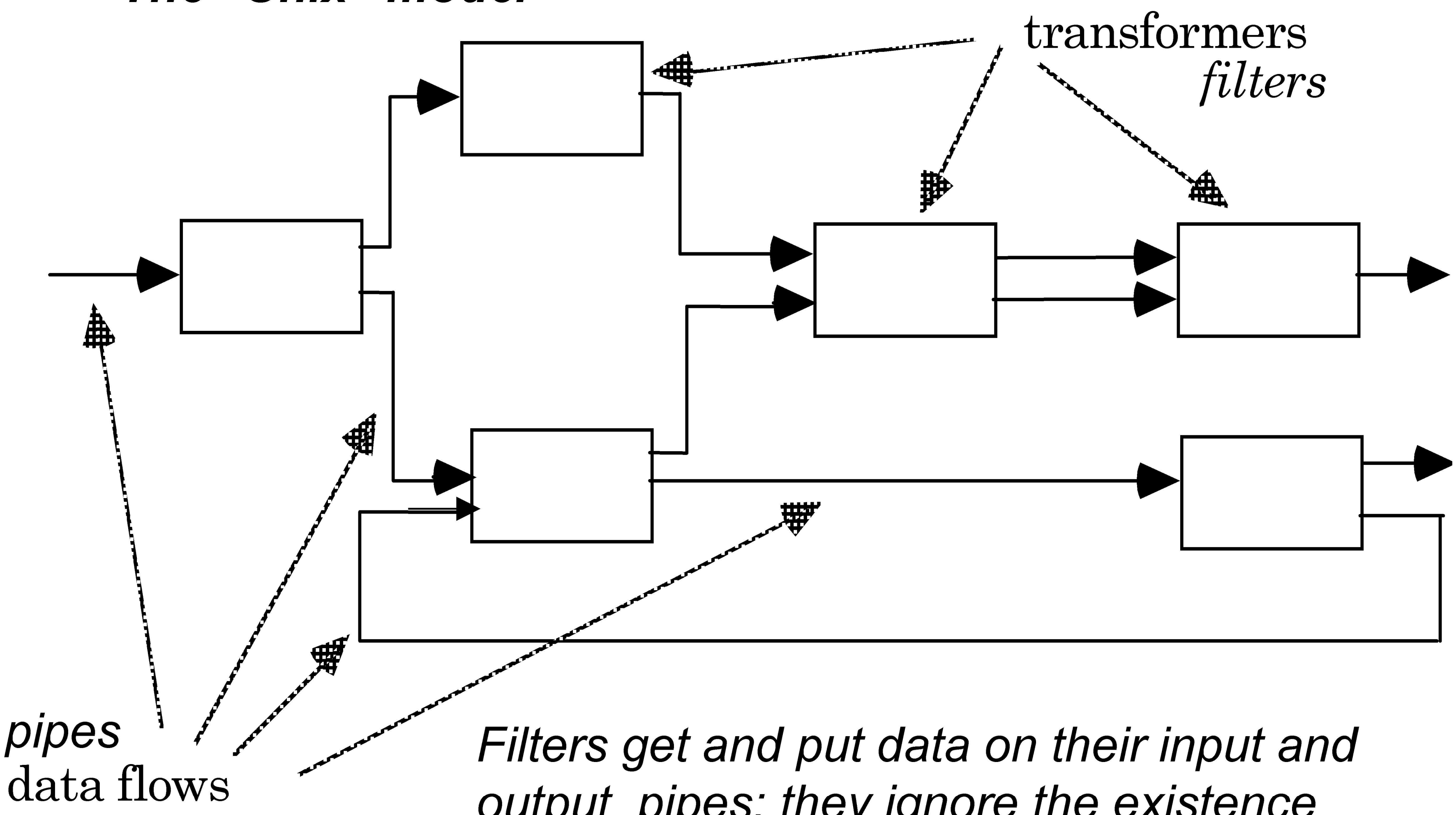
- The system is organized through abstraction levels, as a hierarchy of abstract machines
- Hierarchy is given by the USE relation



the onion-ring structure



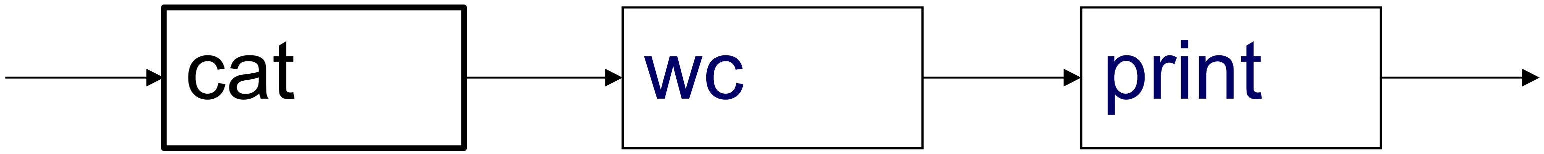
The "Unix" model



Filters get and put data on their input and output pipes; they ignore the existence (and identity) of other filters.

Pipes & Filters

- This is a pipeline



prompt

comment

```
$cat  > shakespeare1 #concatenate to file shakespeare1
$To be #text to be input in shakespeare1
$<ctrl-D> #end of file
$cat > shakespeare2
$or not to be
$<ctrl-D>
```

wordcount

sequence of commands

```
$cat shakespeare1 shakespeare2 > shakespeare
$wc -c shakespeare > shakechar
$print shakechar myprinter
```

pipeline

```
$cat shakespeare1 shakespeare2 | wc -c | print myprinter
```

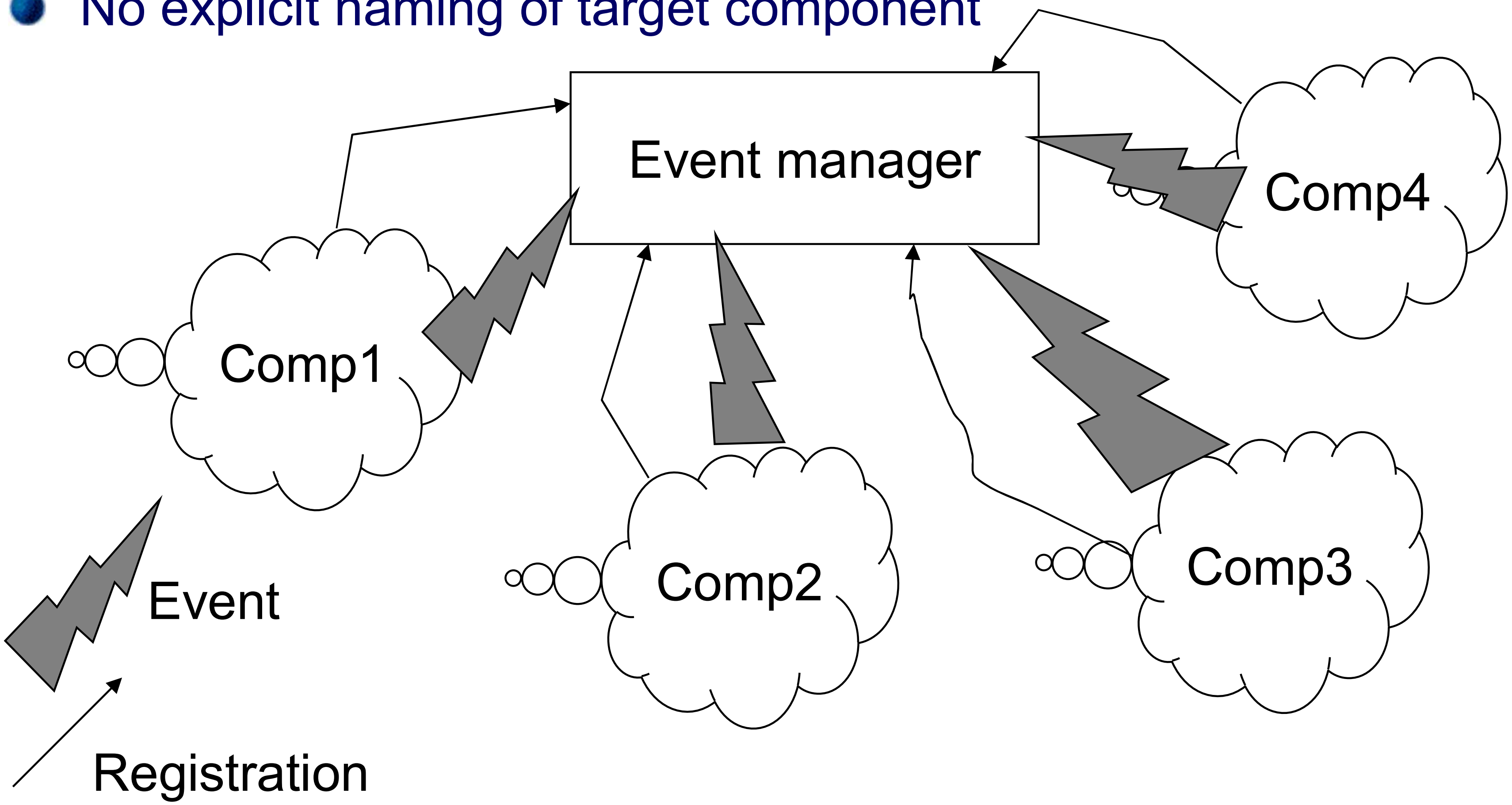
Pipes & Filters

- Various control regimes are possible
  - sequential batch vs. concurrent
- Pro's
  - compositional
    - overall behavior as composition of individual behaviors
  - reuse oriented
    - any two filters can be put together in principle
  - modifications are easy
    - can add/replace filters
- Con's
  - no persistency
  - tendency to batch organization

Event-based Systems

Basic Concepts
Mechanisms
➢ Styles

- Events are broadcast to all registered components
- No explicit naming of target component



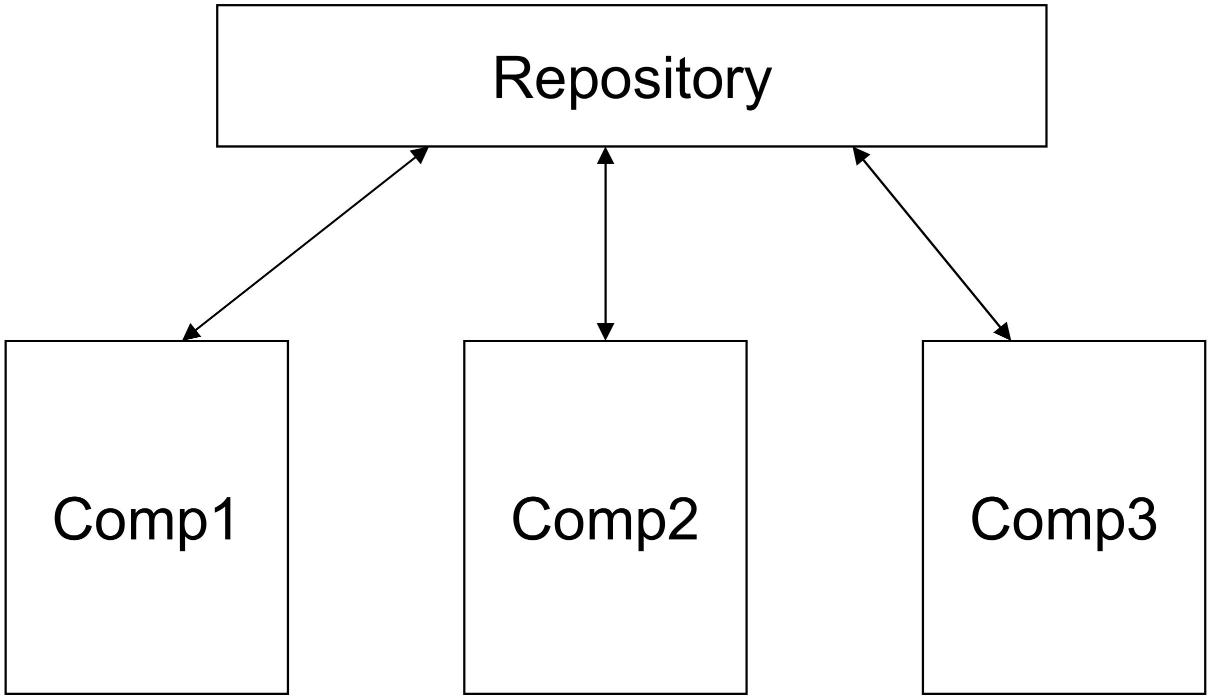
Event-based Systems

Basic Concepts
Mechanisms
➢ Styles

- Pro’s
  - Events are broadcast to all registered components
  - No explicit naming of target component
  - Increasingly used for modern integration strategies
  - Easy addition/deletion of components
- Con’s
  - Ordering of events
- Examples
  - graphical interfaces
  - concurrent, distributed stimulus-response systems

Repository-based Systems

- Components communicate only through a repository



The Database Case

Basic Concepts
Mechanisms
➢ Styles

- Components are active; repository is passive
- A further component (transaction handler) reads input transactions and calls appropriate functions

The Blackboard Case

Basic Concepts
Mechanisms
➢ Styles

- Components read and write into the blackboard
- Blackboard state changes trigger activation of components
  (a blackboard is an active database)