Introduction to Software Design

Sandro Morasca

Università degli Studi dell'Insubria

Dipartimento di Scienze Teoriche e Applicate

Via Mazzini 5

I-21100 Como, Italy

The Goal

- The design activity 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**
- A Notation
- Contracts
- Styles
- Case Study

- Can be defined at different levels of abstractions
- We identify two:
  - level 1: mechanisms
    - what are the constituents and how are they aggregated and related?
  - level 2: styles

➢ Basic Concepts
A Notation
Contracts
Styles
Case Study

- What are modules?
- What are their interfaces?
- Which 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
- A Notation
- Contracts
- Styles
- Case Study

- *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

- ➢ **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- 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 with respect to another
  - coupe vs van vs station wagon
- They are two VIEWS of the same world
- The distinction can be fuzzy

Mechanisms vs. Styles

➢ **Basic Concepts**
A Notation
Contracts
Styles
Case Study

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

Mechanisms: Modules

- ➢ **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- 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

Mechanisms: Interfaces

➢ **Basic Concepts**
A Notation
Contracts
Styles
Case Study

- 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

Mechanisms: Relations

➢ **Basic Concepts**
A Notation
Contracts
Styles
Case Study

- 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

➢ **Basic Concepts**
A Notation
Contracts
Styles
Case Study

- Let S be a set of modules

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

- A binary relation r on S is a subset of

$$S \times S$$

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

Relations

- ➢ **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- 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 $M_k \; r^+ \; M_j$

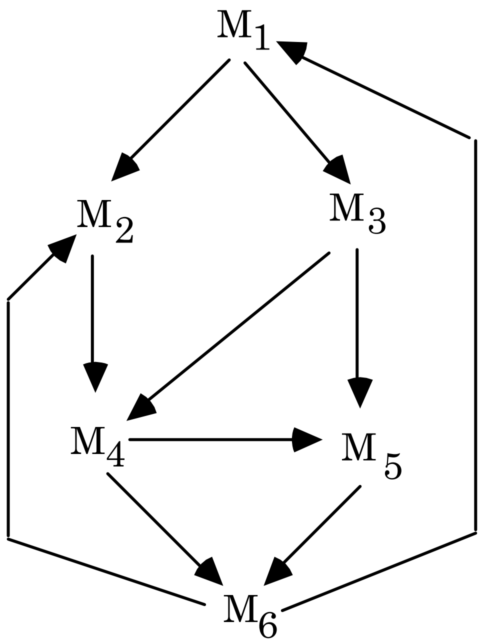
(We assume that our relations are irreflexive)

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

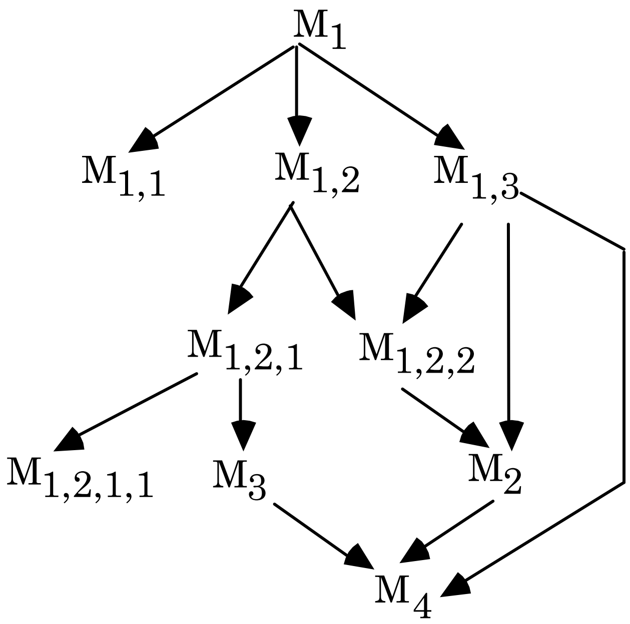
Relations

- **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- Relations can be represented as a graph
- A hierarchy is a DAG



a)



b)

USES

- ➢ **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- 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

IS_COMPONENT_OF

- Basic Concepts
- A Notation
- Contracts
- Styles
- Case Study

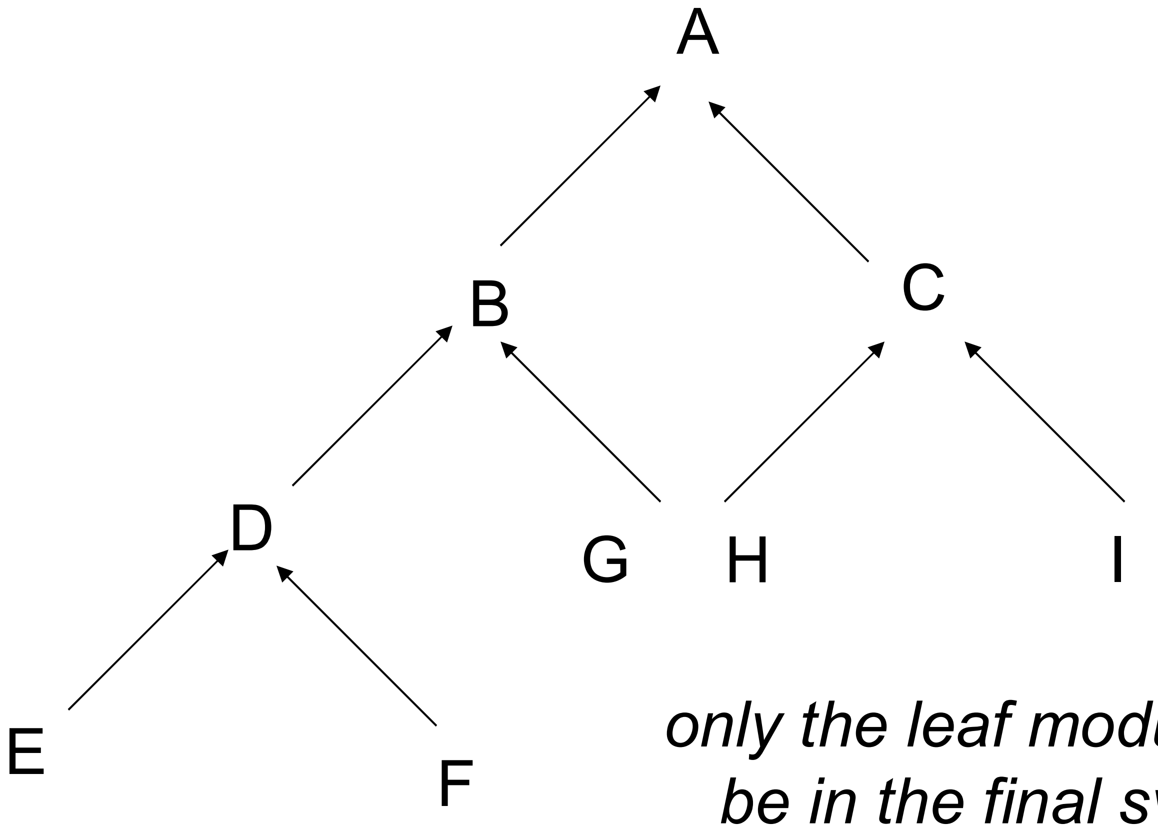
- Used to describe a higher level module as constituted by a number of lower level modules
- A IS_COMPONENT_OF B
  - B consists of several modules, of which one is A
- B COMPRISES A
- $M_Z=\{M_k|M_k \in S \wedge M_k$ IS_COMPONENT_OF $Z\}$

  we say that M_Z IMPLEMENTS Z

IS_COMPONENT_OF

- **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- A hierarchy



only the leaf modules will be in the final system

The INHERITS_FROM Relation

- Basic Concepts
- A Notation
- Contracts
- Styles
- Case Study

- 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 Goals

- ➢ **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- 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

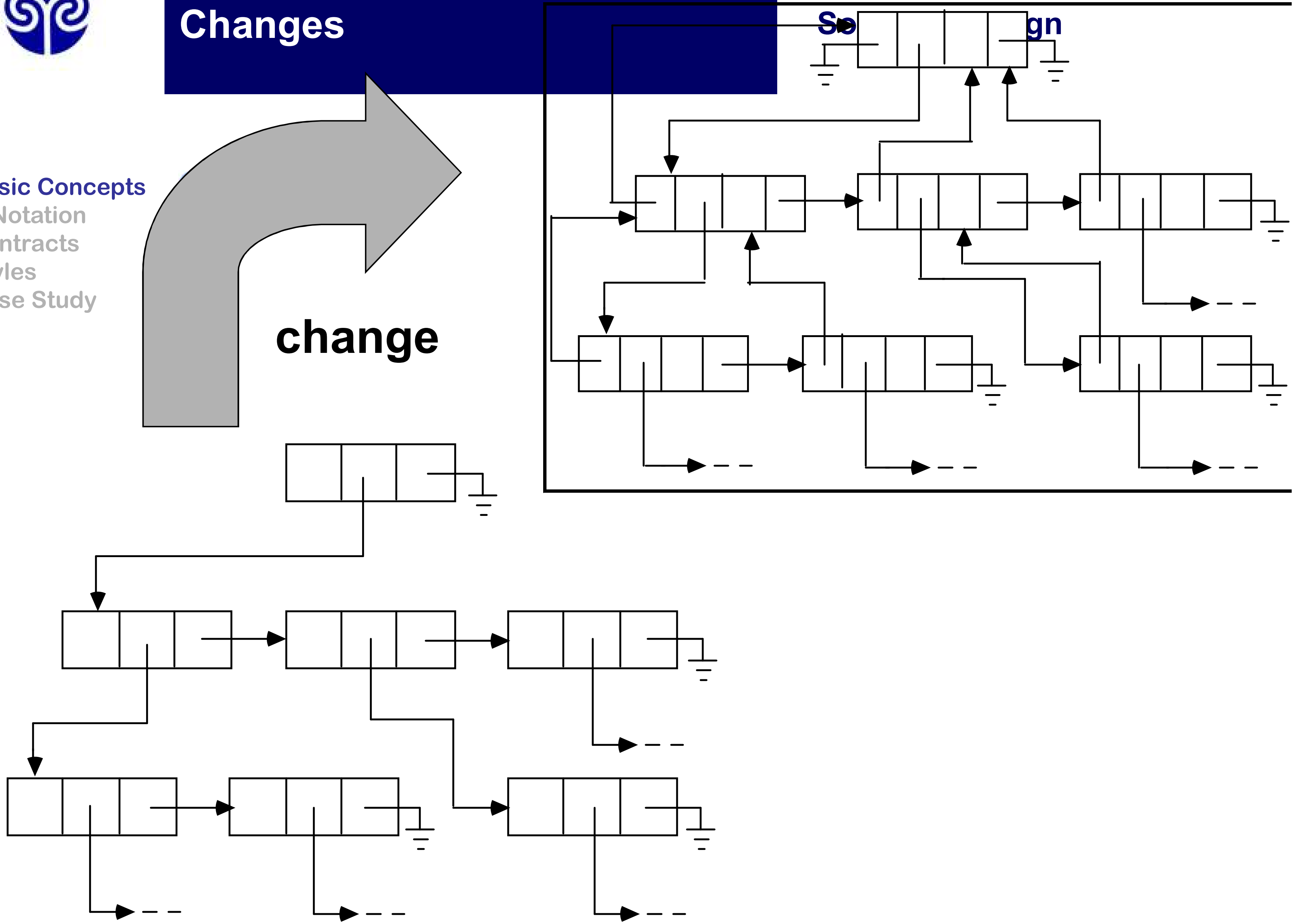
Likely Changes

- **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- 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

Changes

- **Basic Concepts**
  - A Notation
  - Contracts
  - Styles
  - Case Study



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**
A Notation
Contracts
Styles
Case Study

- 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 Principles

➢ **Basic Concepts**
A Notation
Contracts
Styles
Case Study

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

How to Select Modules

- ➢ **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- 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

- ➢ **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- 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

- **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- 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

- ➢ **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

- 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

Key Design Concepts and Principles

➢ **Basic Concepts**
A Notation
Contracts
Styles
Case Study

- 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

Sample Types of Modules

➢ Basic Concepts
A Notation
Contracts
Styles
Case Study

- Abstract operation
- Abstract object (abstract state machine)
    - e.g. the TABLE module
  - a module that encapsulates a data structure
  - exports a set of operations
  - application of operation changes the state of the encapsulated object
- Abstract data type
  - a module that allows abstract objects to be instantiated

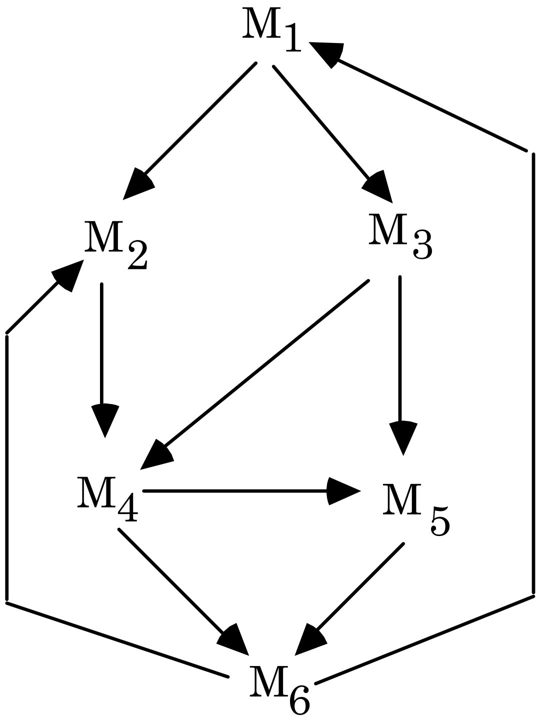
How to Define USES

- **Basic Concepts**
- A Notation
- Contracts
- Styles
- Case Study

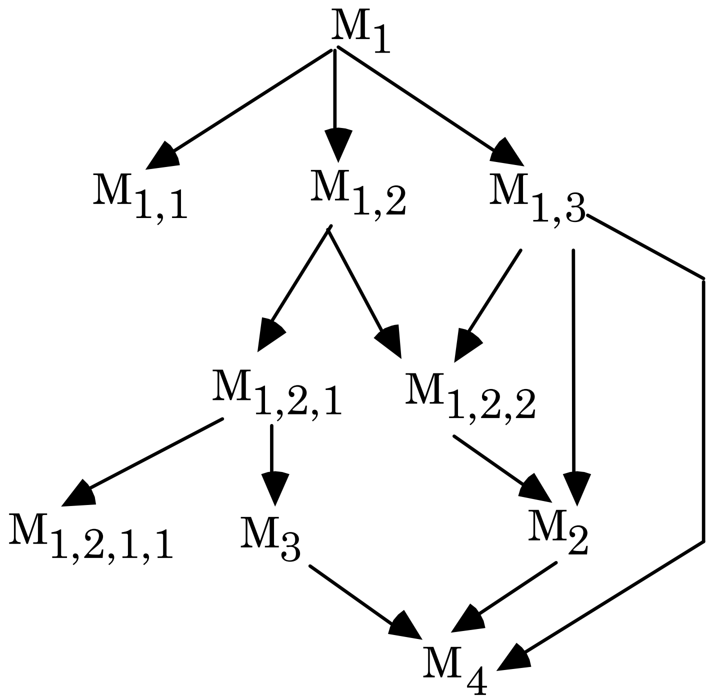
- Make it a hierarchy
  - easier to understand
    - can “read” the DAG from the leaves up
  - easier to verify and develop hierarchically
    - if it is not a hierarchy, we may end up with a system in which nothing works until everything works
- The hierarchy defines a system through “abstraction levels”

Abstraction Levels

➢ **Basic Concepts**
A Notation
Contracts
Styles
Case Study



a)



b)

A Possible Design Notation

Basic Concepts
➢ A Notation
Contracts
Styles
Case Study

module X
uses Y, Z
exports var A : integer;
    type B : **array** (1. .10) **of** real;
    procedure C (D: **in out** B; E: **in** integer; F: **in** real);
    here is an optional natural language description of what A, B, and C actually are, along with possible constraints or properties that clients need to know; for example we might specify that objects of type B se to procedure C should be initialized by the client, and should never contain all zeroes
implementation
    if needed, here are general comments about the rationale of the modularization, hints on the implementation, etc.
    is composed of R, T;
end X

A Possible Design Notation

Basic Concepts
➢ A Notation
Contracts
Styles
Case Study

module R
uses Y
exports var K : **record** . . . **end;**
        type B : **array** (1. .10) **of** real;
        procedure C (D: **in out** B; E: **in** integer; F: **in** real);
implementation
    . . .
end R

module T
uses Y, Z, R
exports var A : integer;
implementation
    . . .
end T

An Example: a Compiler

Basic Concepts
➢ A Notation
Contracts
Styles
Case Study

```
module COMPILER
exports procedure MINI (PROG: in file of char;
                        CODE: out file of char);
    MINI is called to compile the program stored in
    PROG and produce the object code in file CODE
implementation
  It is a conventional compiler implementation.
  ANALYZER performs both lexical and syntactic
  analysis and produces an abstract tree as well as
  entries in the symbol table; CODE_GENERATOR
  generates code starting from the abstract tree and
  information stored in the symbol table. Module
MAIN  acts as a job coordinator.
  is composed of ANALYZER, SYMBOL_TABLE,
                 ABSTRACT_TREE_HANDLER,
                 CODE_GENERATOR, MAIN
end COMPILER
```

An Example: a Compiler

Basic Concepts
➢ A Notation
Contracts
Styles
Case Study

module ANALYZER
uses SYMBOL_TABLE, ABSTRACT_TREE_HANDLE
exports procedure ANALYZE (SOURCE: in file of char);

SOURCE is analyzed; an abstract tree is produc by using the services provided by the handler an recognized entities, with their attributes, are stored in the symbol table

. . .

end ANALYZER

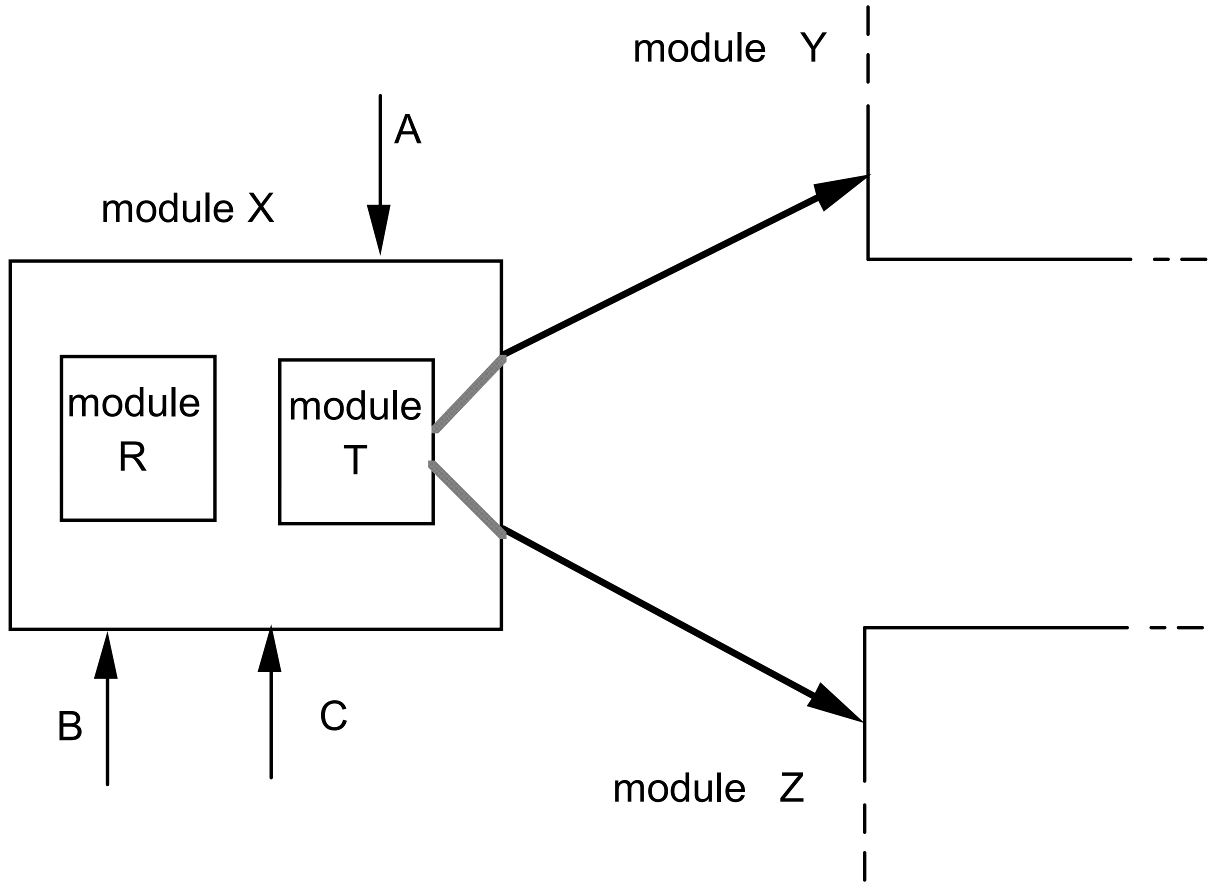
An Example: a Compiler

Basic Concepts
➢ A Notation
Contracts
Styles
Case Study


```
module CODE_GENERATOR
uses SYMBOL_TABLE, ABSTRACT_TREE_HANDLE
exports procedure CODE (OBJECT: out file of char)
        the abstract tree is traversed using the operatioı
        exported by the ABSTRACT_TREE_HANDLER aı
        accessing the information stored in the symbol
        table in order to generate code in the output file
            . . .
end CODE_GENERATOR
```

A Sample Graphical Notation

Basic Concepts
➢ A Notation
Contracts
Styles
Case Study



Basic Concepts
A Notation
➢ Contracts
Styles
Case Study

- 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 by Contract

Basic Concepts
A Notation
➢ **Contracts**
Styles
Case Study

- It refines a known design principle, especially suitable for OO design:
  - The module (class) interface defines a contract
- What is contract?
  - An agreement between a *client* and a *contractor*
  - Defines *obligations* to achieve *benefits*

An Example

Basic Concepts
A Notation
➢ Contracts
Styles
Case Study

	Obligations	Benefits
Client	Provide 1 hectare of land	Get building at least 3 stories high for a given amount of money
Contractor	Build building at least 3 stories high for a given amount of money	No need to to do anything if the 1 hectare land is not provided

Contract for an OO Module

- Define what each method requires (obligation for the client)

  precondition

- and what each method provides (obligation for the contractor)

  postcondition

Preconditions and postconditions may be expressed using logic

Example

Basic Concepts
A Notation
➢ Contracts
Styles
Case Study

- Operation *insert(element)* in a table

  Precondition
  - no_elements < size

  Postcondition
  - element is in table
  - no_elements'=no_elements +1
    - no_elements' denotes the value after the operation

Why Preconditions?

Basic Concepts
A Notation
➢ Contracts
Styles
Case Study

- Should a routine be prepared to handle all possible inputs?
  - NO
    - WEAK precondition (TRUE means no constraints at all); all complications delegated to routine
    - STRONG precondition (FALSE means it cannot be invoked at all)
  - The choice of the precondition is a design decision; there is no absolute rule
    - preferable to write simple routines that satisfy a well-defined contract rather than a routine that tries to attempt every imaginable situation

Preconditions and Postconditions

Basic Concepts
A Notation
➢ Contracts
Styles
Case Study

- Precondition
  - The client must guarantee the property
    - If p is the precondition for method m, either we write (for any object x)
      - if (x.p) x.m(…)

        else …*special treatment*

        }
    - or we ensure that p holds before the call by reasoning on the program
- Postcondition
  - The contractor must guarantee it in the method's implementation

Internal Properties of a Class

Basic Concepts
A Notation
➢ Contracts
Styles
Case Study

- We can specify a property that all instances should satisfy as an *invariant*
- The invariant is true after creation and before and after each operation
  - e.g. $0 \leq$ no_elements $\leq$ size
- The invariant defines an additional proof obligation
  - the class implementation must satisfy it

Basic Concepts
A Notation
➢ Contracts
Styles
Case Study

- Creation operation
  - $\{pre_c\}$ constructor $\{INV'\}$

- Any other operation
  - $\{pre_{operation} \wedge INV\}$ operation $\{post_{operation} \wedge INV'\}$

Preconditions

Basic Concepts
A Notation
➢ Contracts
Styles
Case Study

The client is responsible for their truth …however…

- it may be impossible to evaluate applicability before application of operation
  - overflow, input/output, …
- frequent operations which almost never fail
  - new and memory exhausted
- there are errors that cause invocations that do not satisfy the precondition

The Role of Exceptions

Basic Concepts
A Notation
➢ Contracts
Styles
Case Study

- They should be raised if one of the following conditions are violated
  - precondition
  - postcondition
  - invariant
- When control leaves a routine, either its postcondition and the invariant are true or an exception is returned
  - returned exceptions should be listed in the interface

Inheritance

Basic Concepts
A Notation
➢ Contracts
Styles
Case Study

- Subclasses can add attributes and methods
- Can redefine methods
  - syntactic constraints
    - covariance of result and countervariance of parameters
  - semantic constraints
    - $pre_{class} \rightarrow pre_{subclass}$
    - $post_{subclass} \rightarrow post_{class}$

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

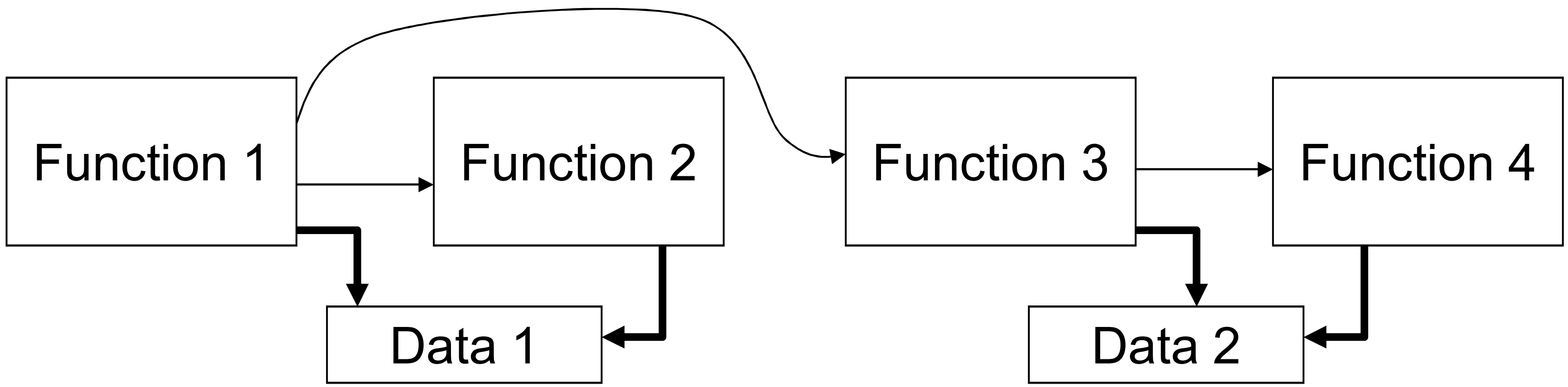
Basic Concepts
A Notation
Contracts
➢ Styles
Case Study

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

A Functional Architecture

Basic Concepts
A Notation
Contracts
➢ Styles
Case Study

- 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
A Notation
Contracts
➢ Styles
Case Study

- "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

Basic Concepts
A Notation
Contracts
➢ Styles
Case Study

calls from functions to functions

use of common data

```
public class InsiemeDiInteri
{ private final static int CAPACITA = 10;
  private static int n;
  private static int elenco[] = new int [CAPACITA];
  public static void inserimento(int numero)
  { if (!ePieno() && (!appartiene(numero)))
    { elenco[n] = numero;
      n++; } }
  public static boolean appartiene(int numero)
  { return ricerca(numero) != -1; }
  private static int cardinalita() { return n; }
  ...
  public static void main()
  { numero = read(); //supponiamo che esista la read
    inserimento( numero );
    numero = read(); inserimento( numero );
    numero = read(); eliminazione( numero );
  }
}
```

Basic Concepts
A Notation
Contracts
➤ **Styles**
Case Study

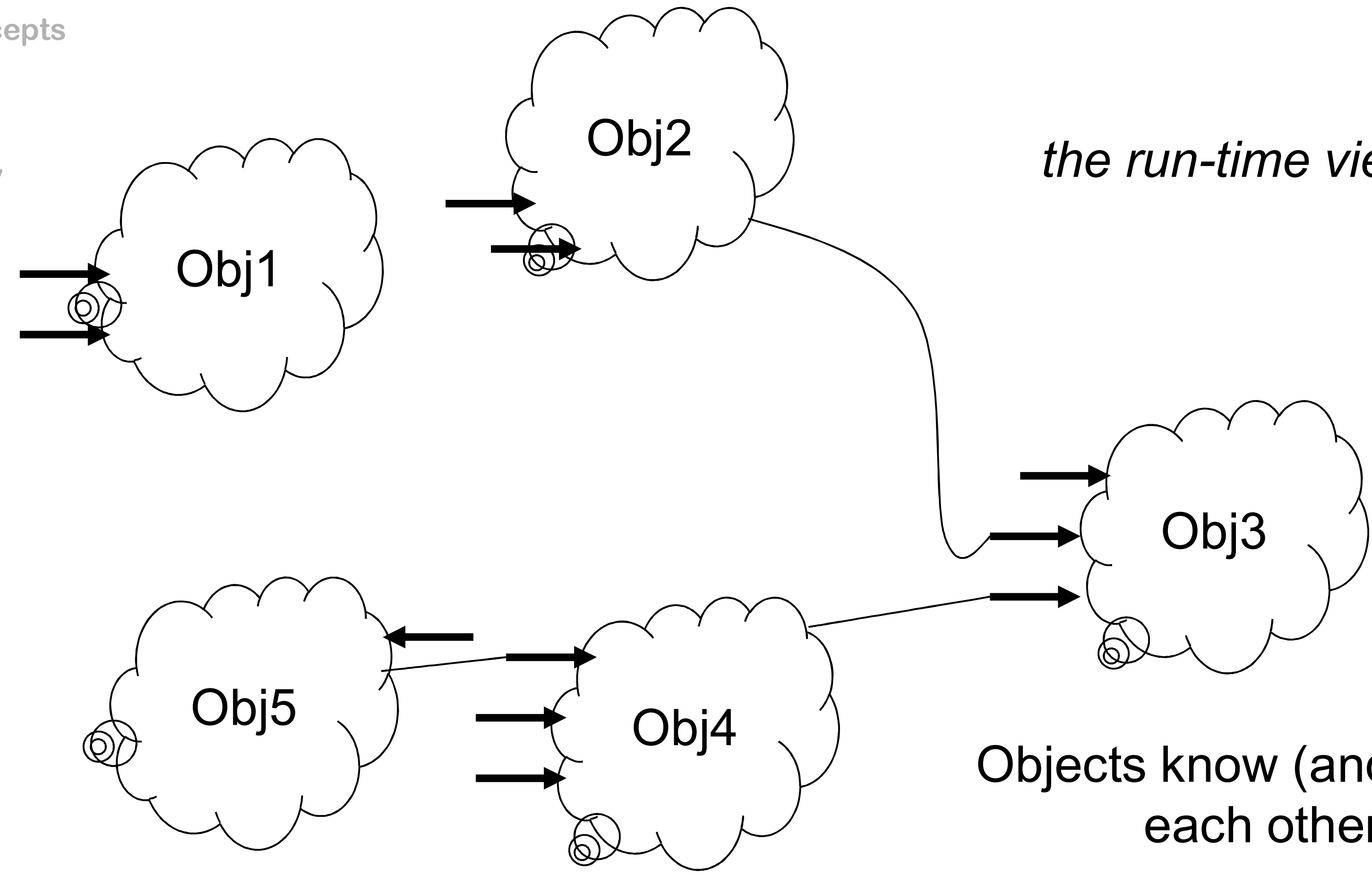
calls from functions
to functions

use of common data

```
public class InsiemeDiInteri
{ private final static int CAPACITA = 10;
  private int n;
  private int elenco[] = new int [CAPACITA];
  public void inserimento(int numero)
  { if (!ePieno() && (!appartiene(numero)))
    { elenco[n] = numero;
      n++; } }
  public boolean appartiene(int numero)
  { return ricerca(numero) != -1; }
  private int cardinalita() { return n; }
  ...
  public static void main()
  { InsiemeDiInteri = insieme new InsiemeDiInteri();
    numero = read(); //supponiamo che esista la read
    insieme.inserimento( numero );
    numero = read(); insieme.inserimento( numero );
    numero = read(); insieme.eliminazione( numero );
  }
}
```

An Object-oriented Architecture

Basic Concepts
A Notation
Contracts
➢ **Styles**
Case Study

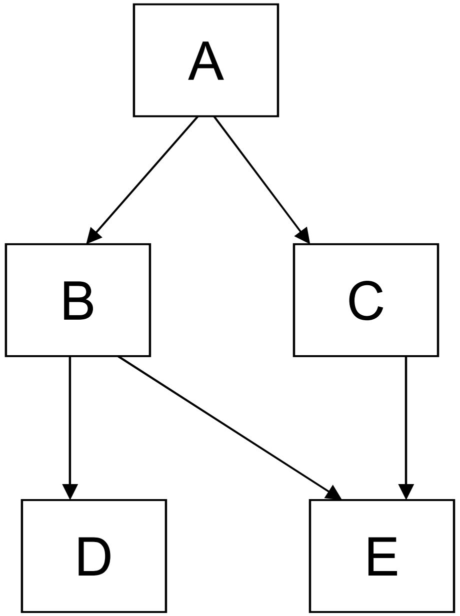


the run-time view

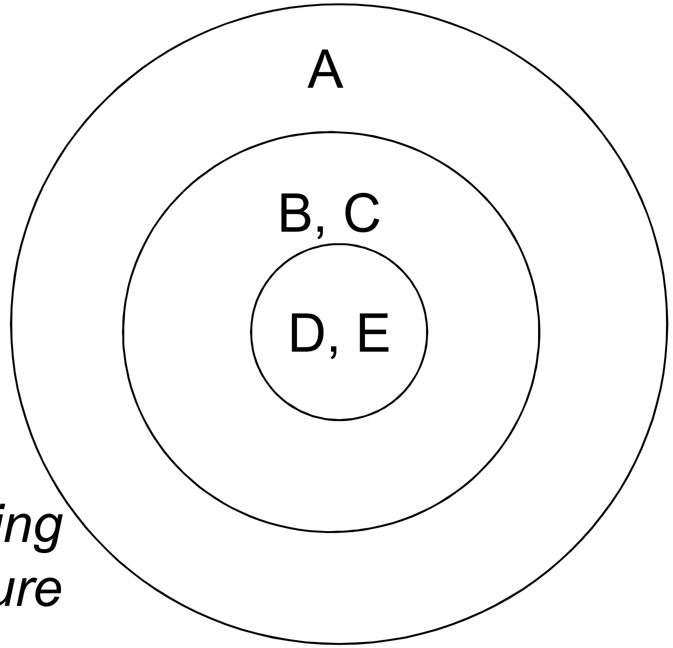
Objects know (and name) each other

Layered System

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

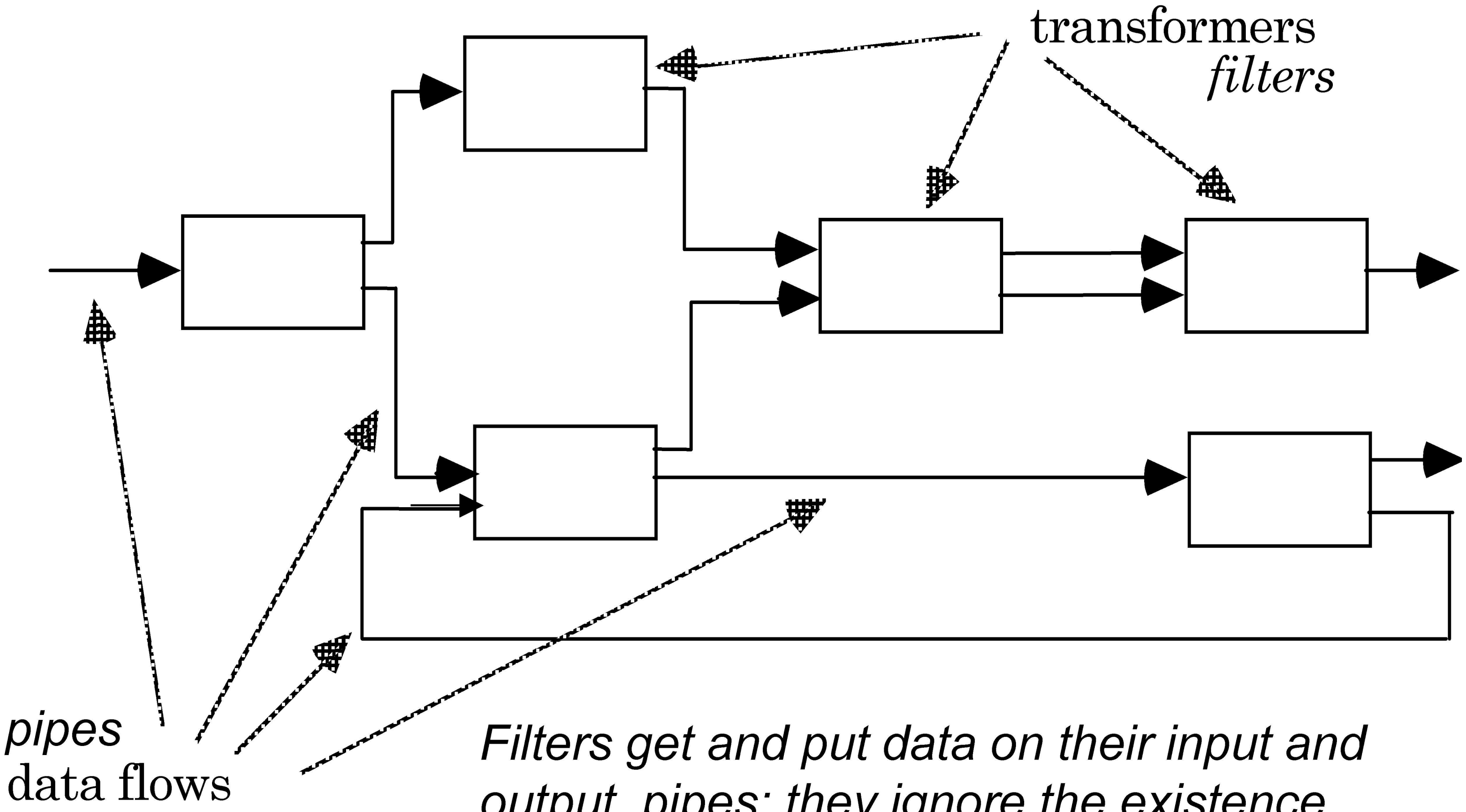


the onion-ring structure



Basic Concepts
A Notation
Contracts
➢ Styles
Case Study

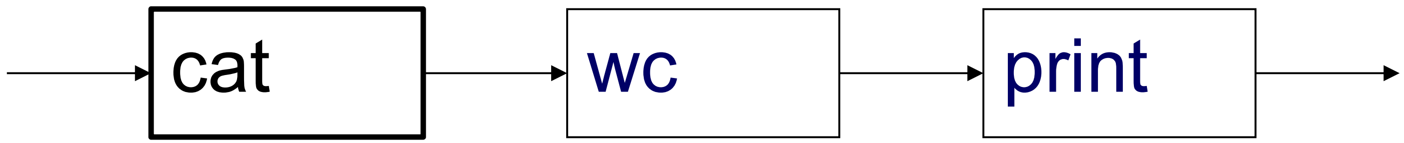
The "Unix" model



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

Basic Concepts
A Notation
Contracts
➢ **Styles**
Case Study

- 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
```

Basic Concepts
A Notation
Contracts
➢ Styles
Case Study

- A script to delete a file or a directory

```
foreach name ($argv)
  if ( -f $name ) then
    echo -n "delete the file '${name}' (y/n/q)?"
  else
    echo -n "delete the entire directory '${name}' (y/n/q)?"
  endif
  set ans = $<
  switch ($ans)
  case n:
    continue
  case q:
    exit
  case y:
    rm -r $name
    continue
  endsw
end
```

does $name identify an ordinary file or a directory one?

do not print the newline character

input from the keyboard

go the top of enclosing loop

recursively remove the entire subtree

Pipes & Filters

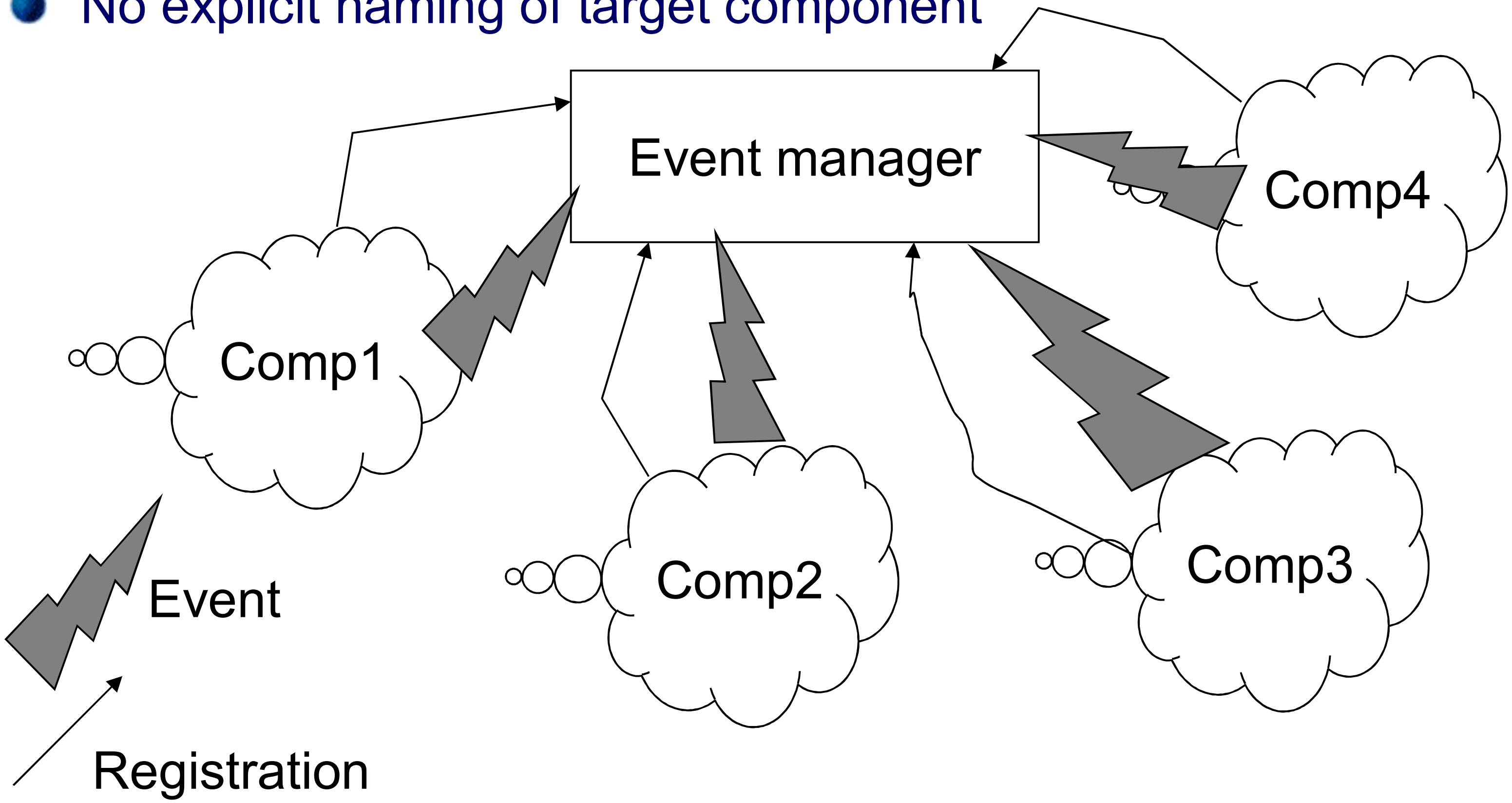
Basic Concepts
A Notation
Contracts
➢ **Styles**
Case Study

- 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
A Notation
Contracts
➢ Styles
Case Study

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



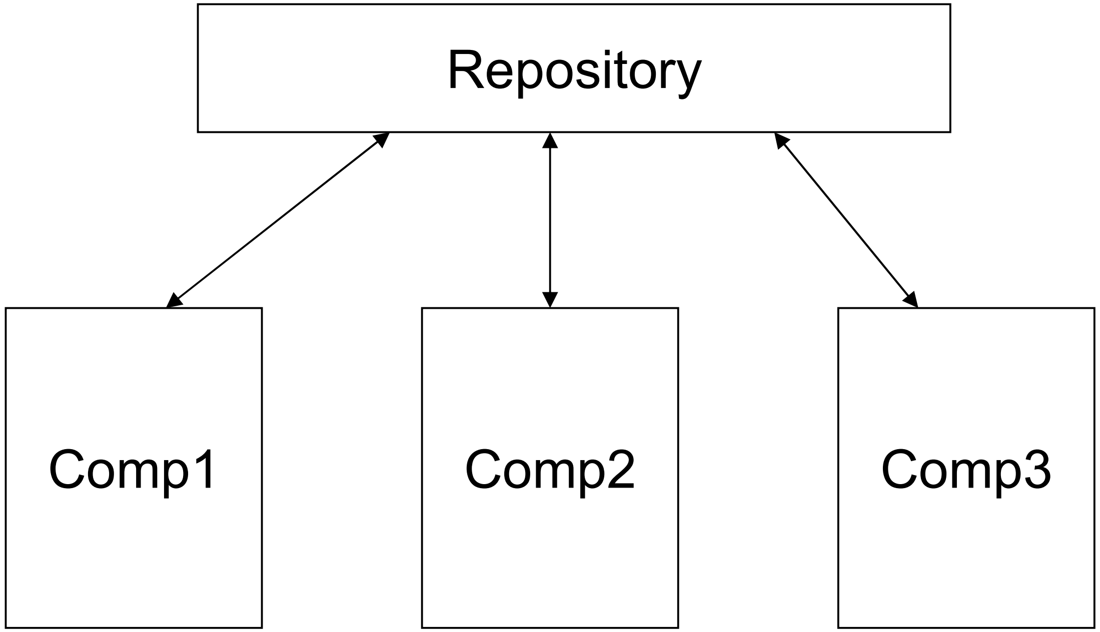
Event-based Systems

Basic Concepts
A Notation
Contracts
➢ **Styles**
Case Study

- 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
  - Potential scalability problems
  - Ordering of events
- We will discuss event-based systems in the context of a graphical interfaces
  - concurrent, distributed stimulus-response systems are an even more complex case

Basic Concepts
A Notation
Contracts
➢ Styles
Case Study

- Components communicate only through a repository



The Database Case

Basic Concepts
A Notation
Contracts
➢ **Styles**
Case Study

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

The Blackboard Case

Basic Concepts
A Notation
Contracts
➢ Styles
Case Study

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

Basic Concepts
A Notation
Contracts
Styles
➢ Case Study

Keywords in context

The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, 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.

(Parnas 1972)

Example

- Basic Concepts
- A Notation
- Contracts
- Styles
- **Case Study**

Nel mezzo del cammin di nostra vita

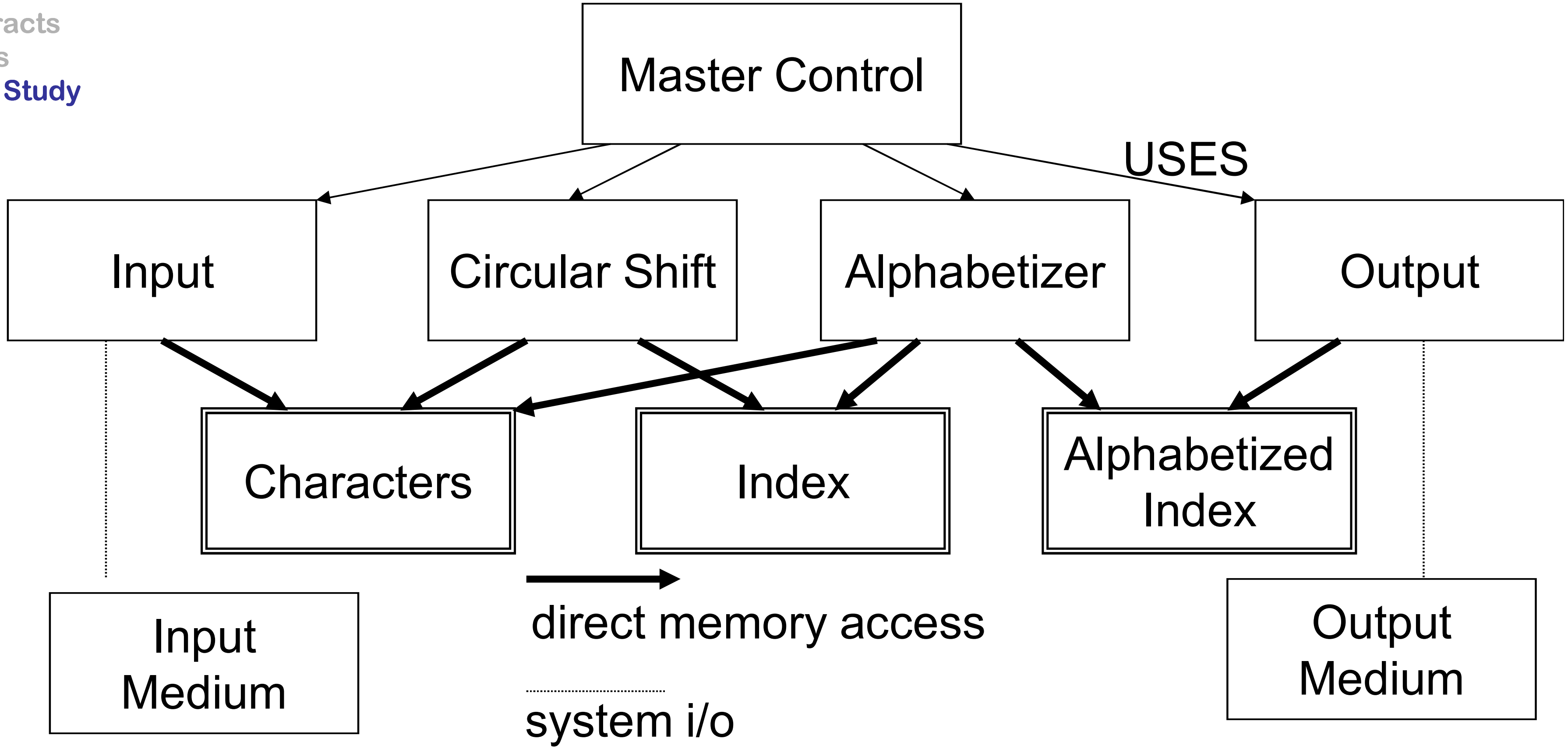
mezzo del cammin di nostra vita nel
del cammin di nostra vita nel mezzo
cammin di nostra vita nel mezzo del
di nostra vita nel mezzo del cammin
nostra vita nel mezzo del cammin
vita nel mezzo del cammin di nostra

The Problem

- Basic Concepts
- A Notation
- Contracts
- Styles
- **Case Study**

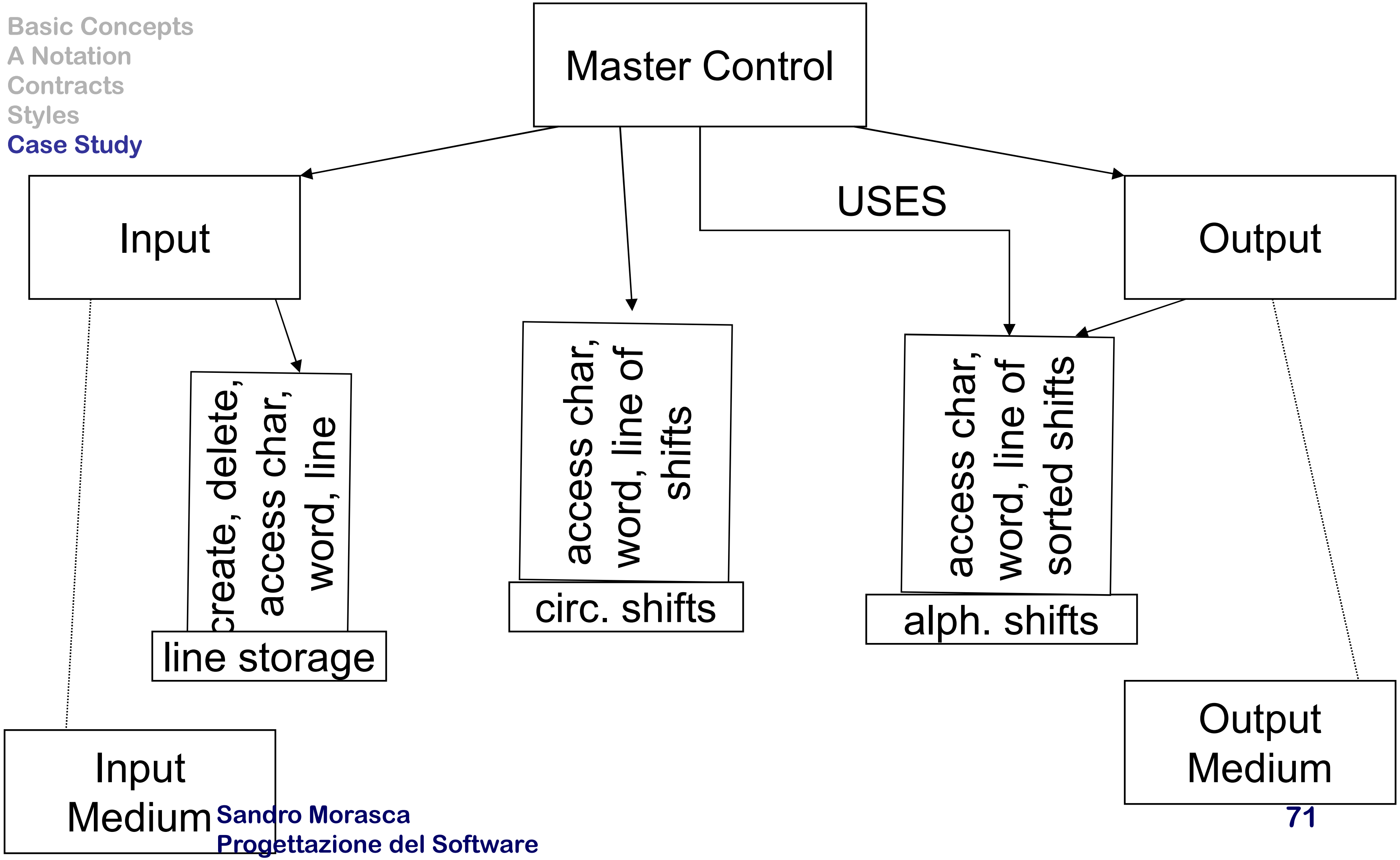
- Is simple but realistic
  - UNIX MAN pages are an example
- Many possible changes can be anticipated
  - algorithm
    - shifting performed on each line as it is read
    - on all lines after they are read
    - etc.
  - data representation
    - how lines are stored
    - how circular shifts are store
  - enhancements
    - elimination of noise words (“a”, “an”, “the”, …)

- Paradigm: program/subroutines with shared data



Case 2: Parnas' Solution

- Basic Concepts
- A Notation
- Contracts
- Styles
- **Case Study**



Potential Changes

- Basic Concepts
- A Notation
- Contracts
- Styles
- **Case Study**

- Easy to deal with changes in the implementation (with no changes in functional specs)
  - How characters and words are represented
  - Explicit vs. implicit representation of shifts and alphabetization

How about Other Changes?

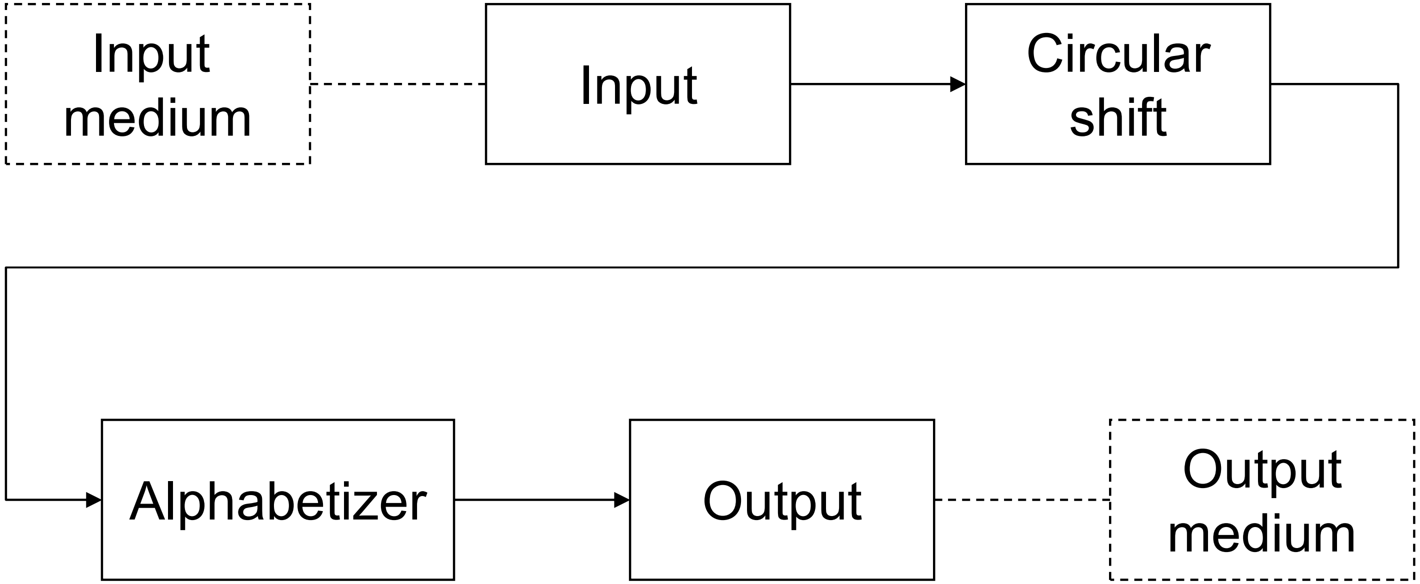
- Basic Concepts
- A Notation
- Contracts
- Styles
- **Case Study**

- Omit shifts that start with a noise word
  - adding a filter in output module is straightforward, but inefficient (unnecessary alphabetization)
  - modify circular shifter (OK, but if other changes are also made, then the module becomes too complicated)
- Include only shifts starting from a word in a given set
  - *as above*

Case 3: Pipes & Filters

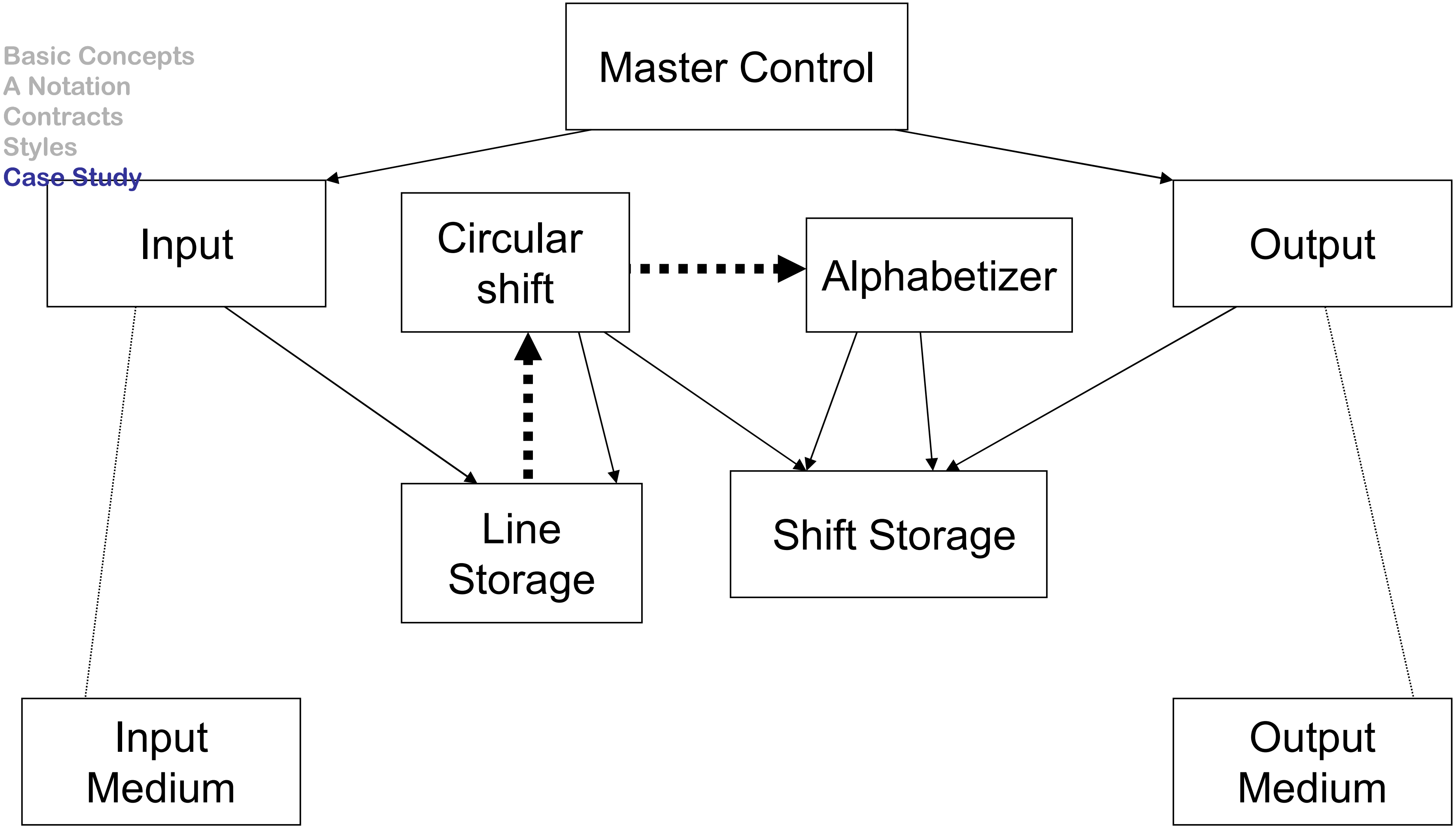
- Basic Concepts
- A Notation
- Contracts
- Styles
- Case Study

- No persistency
  - *to delete a line, an extra filter (and pipe generation) is needed*



Case 4: Event-based Integration

- Basic Concepts
- A Notation
- Contracts
- Styles
- Case Study



Event-based Integration

- “Circular shift” registers for the event “new line inserted” (alternatively, “all lines inserted”, in batch situation)
- “Alphabetizer” registers for completion of shifter activities (e.g., a shift produced for a line, all shifts for a line, all shifts for all lines)

A Comparison

- Basic Concepts
- A Notation
- Contracts
- Styles
- **Case Study**

	Functional decomposition	Parnas's solution (ADTs)	Pipe&filter	Event based invocation
Change in algorithm	-	-	+	+
Change in data rep	-	+	-	-
Change in function	+	-	+	+
Performance	+	+	-	-
Reuse	-	+	+	-