On The Criteria …

CS580 Advanced Software Engineering
http://cs580.yusun.io
October 13, 2014

Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

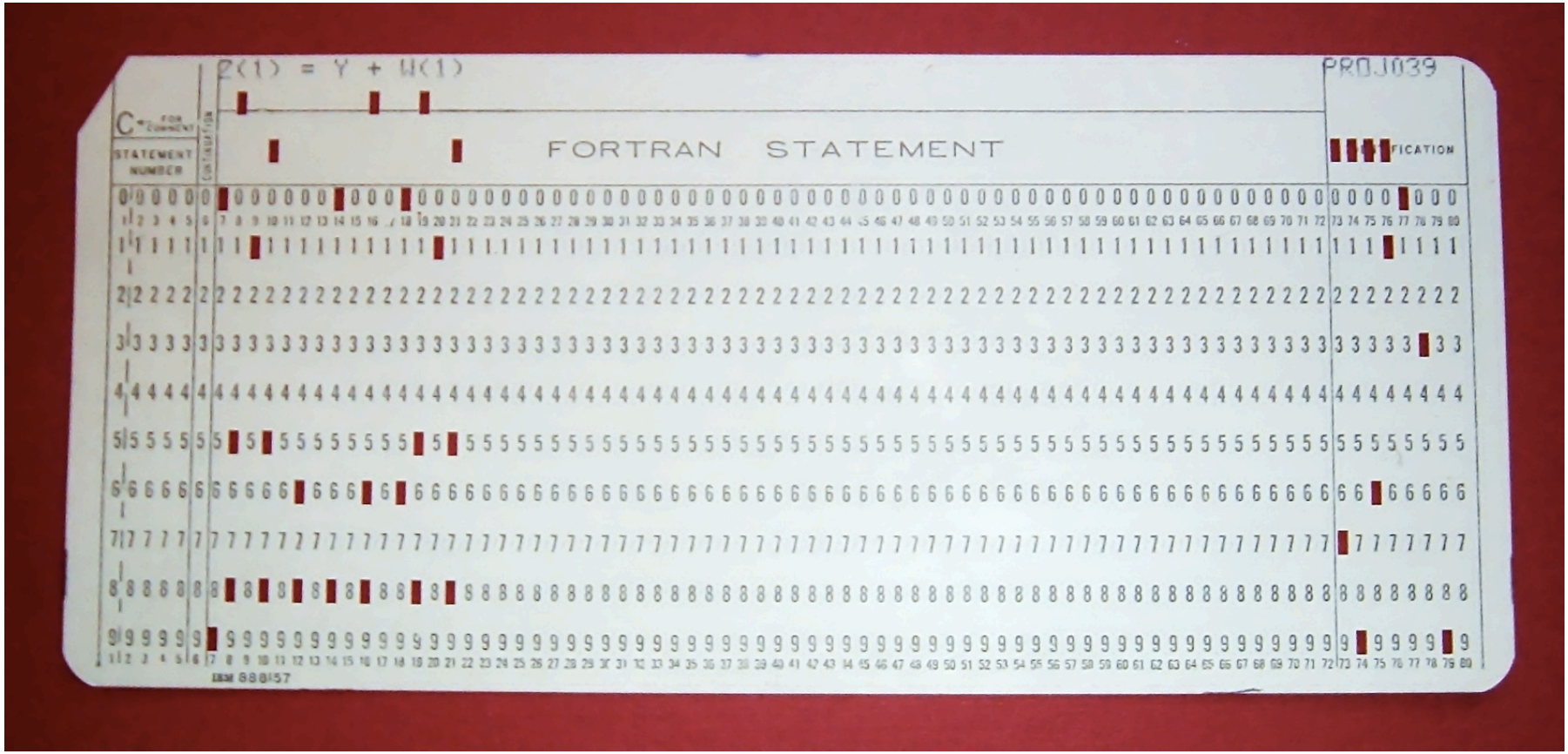
Announcement

- October 15, Wednesday 5:30-6:30pm
  - (New) Graduate Students Orientation
- Class Starts on 6:40pm
- October 17, Engineering/Hi-Tech Career Fair
  - Do your homework!

A Sketchy Evolution of Software Design

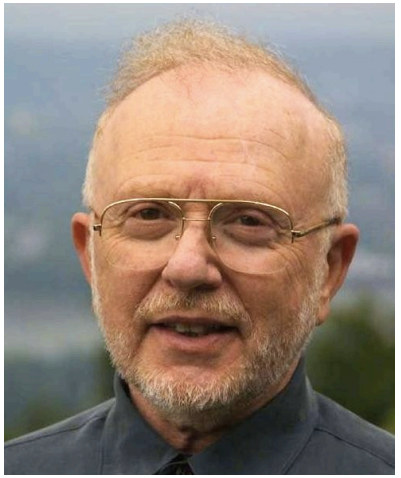
- 1960s
  - Structured Programming
    - ("Goto Considered Harmful", E.W.Dijkstra)
    - Emerged from considerations of formally specifying the semantics of programming languages, and proving programs satisfy a predicate
- 1970s
  - Structured Design
    - Methodology/guidelines for dividing programs into subroutines
- 1980s
  - Modular (object-based) programming
    - Ada, Modula, Euclid, …
    - Grouping of sub-routines into modules with data
- 1990s
  - Object-Oriented Languages started being commonly used (60s origin)
  - Object-Oriented Analysis and Design for guidance

Punched Card Programming



The Paper

- "On the Criteria To Be Used in Decomposing Systems into Modules"
- David Parnas
- Communications of the ACM, 1972
- Perhaps most popular paper in SE
  - Initially rejected, "Obviously Parnas does not know what he was talking about because nobody does it that way"



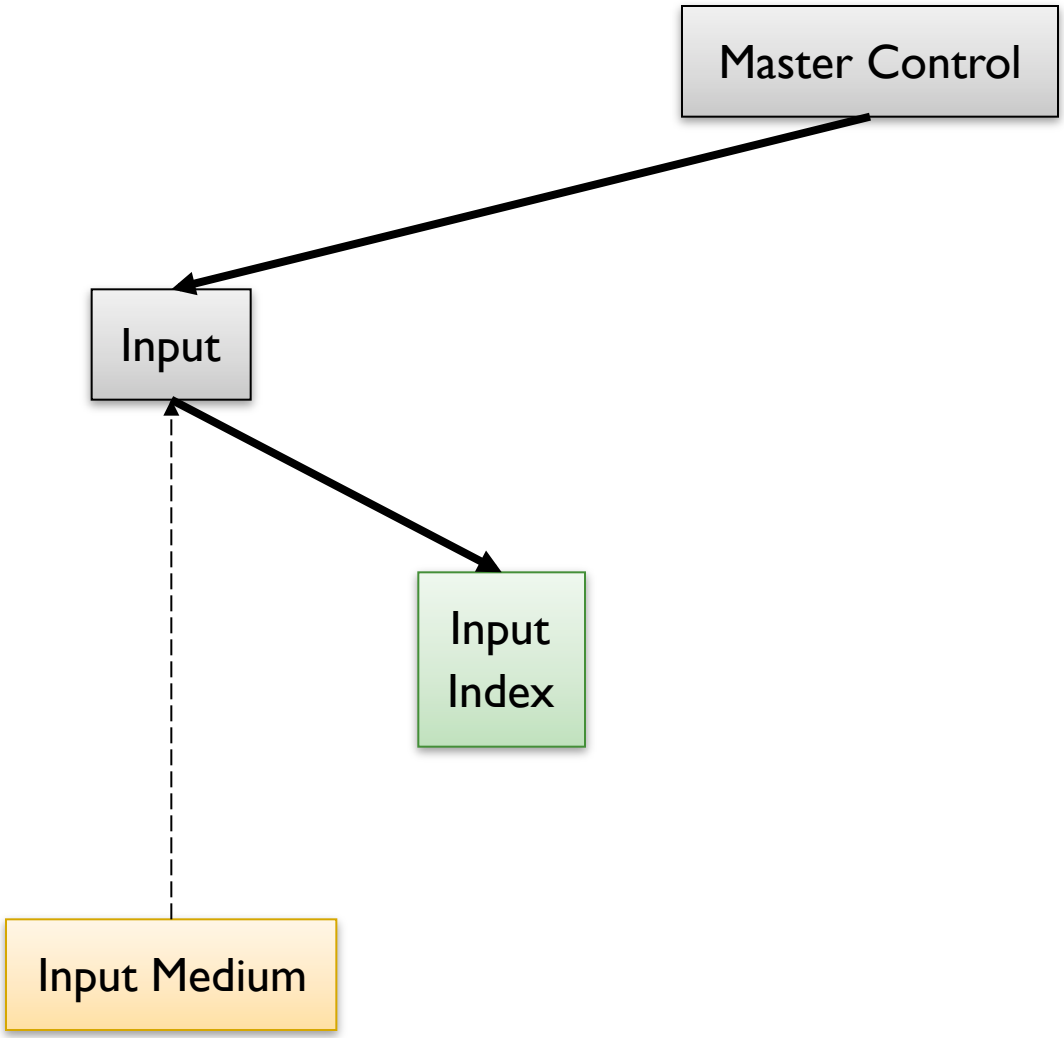
Module Structure

- Discusses "modularization"
  - Module = a collection of subroutines and data elements
  - Critique of Procedural Design
    - Pointing the way to object-based and OO design
- Describes two ways to modularize a program that generates KWIC (Key Word in Context) indices
  - Modularization 1 – Based on the sequence of steps to perform
  - Modularization 2 – Based on principle of "information hiding"

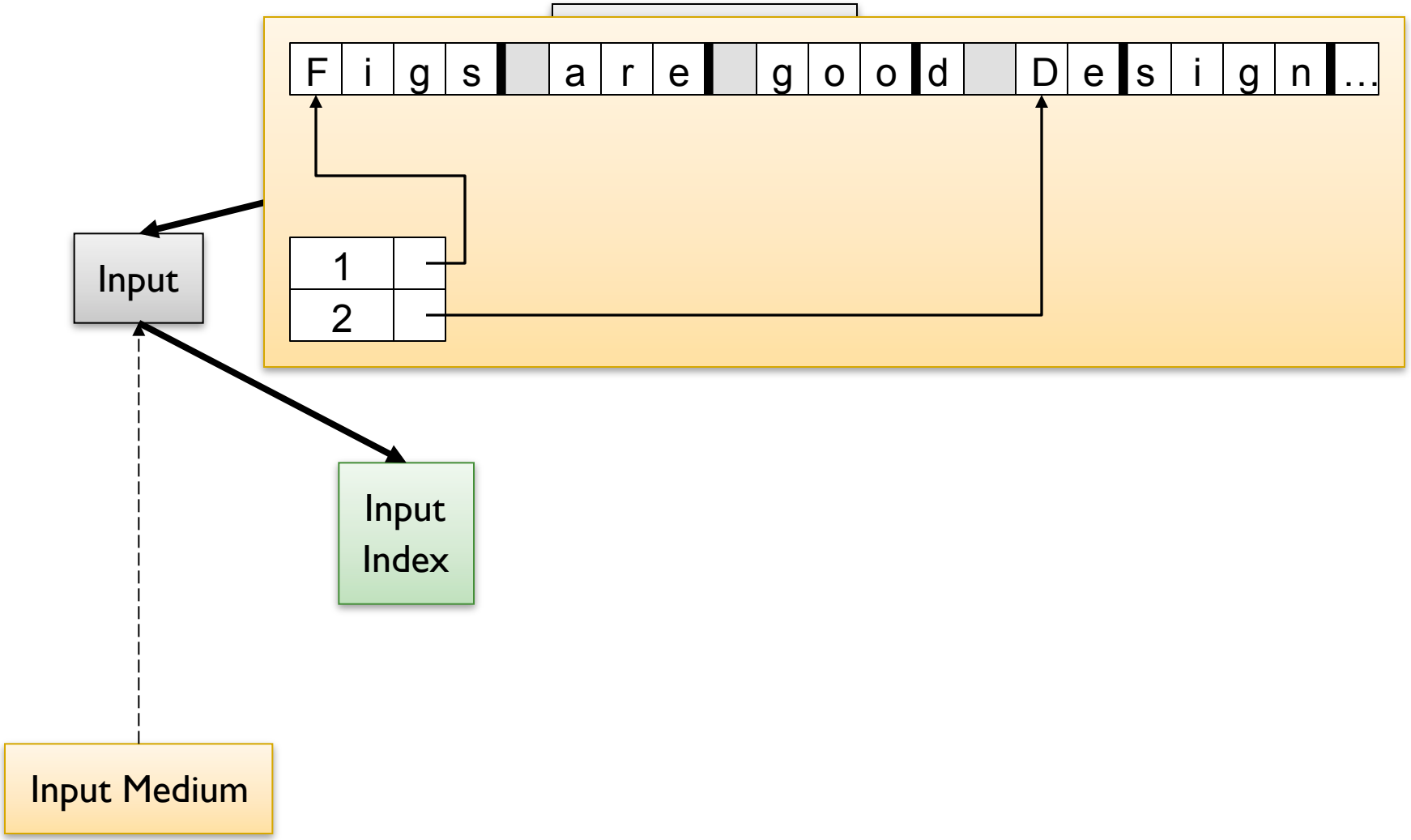
KWIC

- Input
  - Figs are Good
  - Designing Software for Ease of Construction
- Output
  - are Good Figs
  - Construction Designing Software for Ease of
  - Designing Software for Ease of Construction
  - Ease of Construction Designing Software for
  - Figs are Good
  - for Ease of Construction Designing Software
  - Good Figs are
  - of Construction Designing Software for Ease
  - Software for Ease of Construction Designing

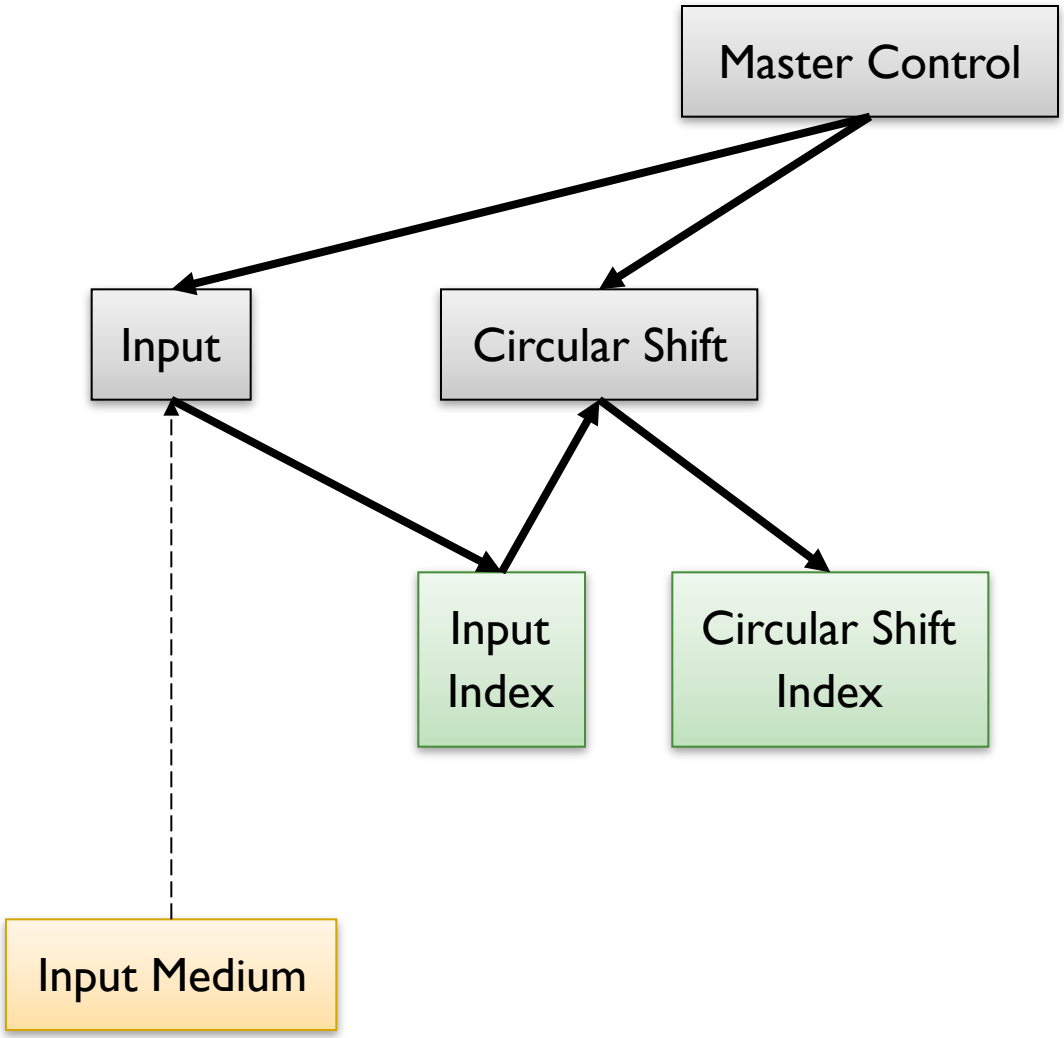
Modularization 1



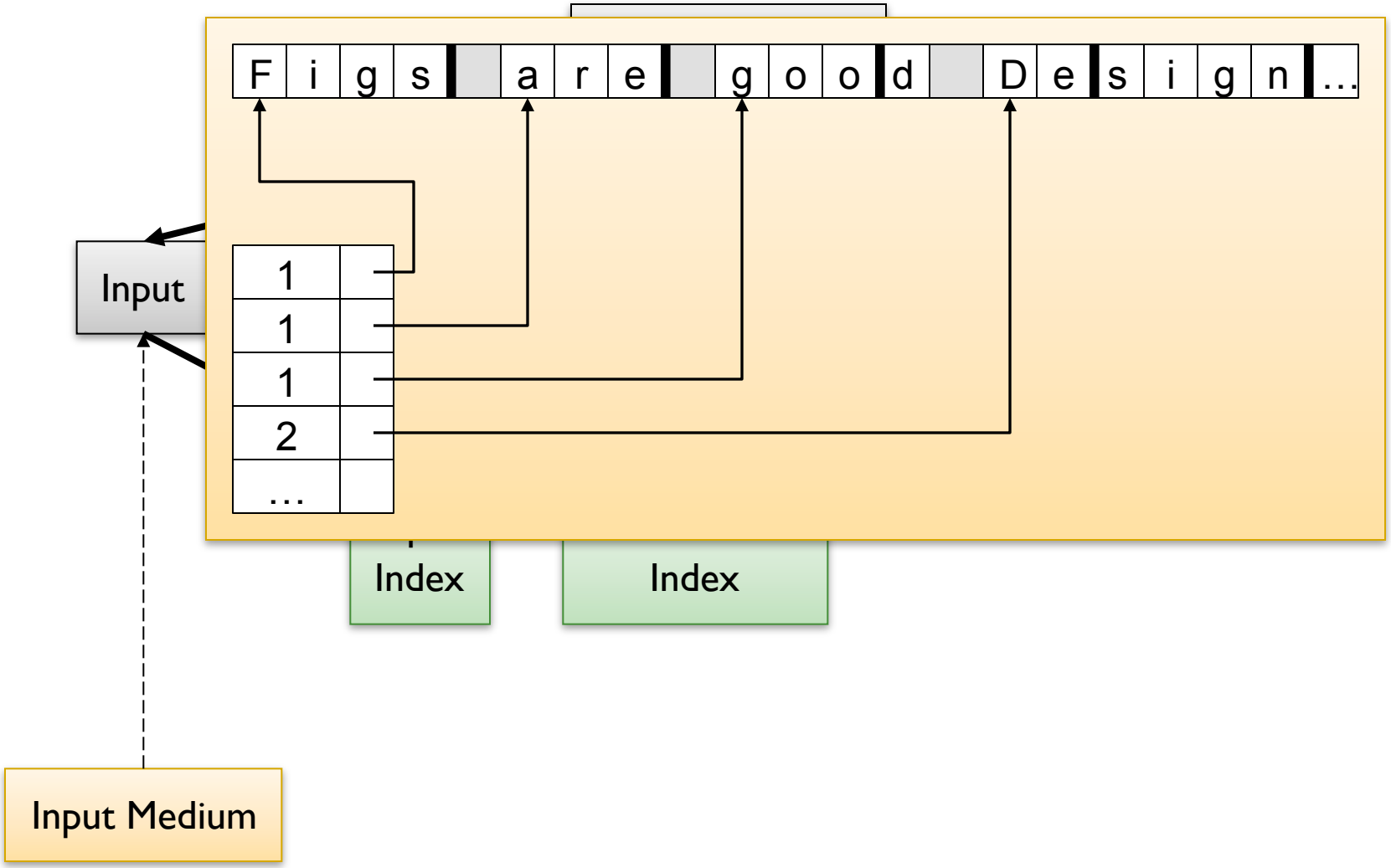
Modularization 1



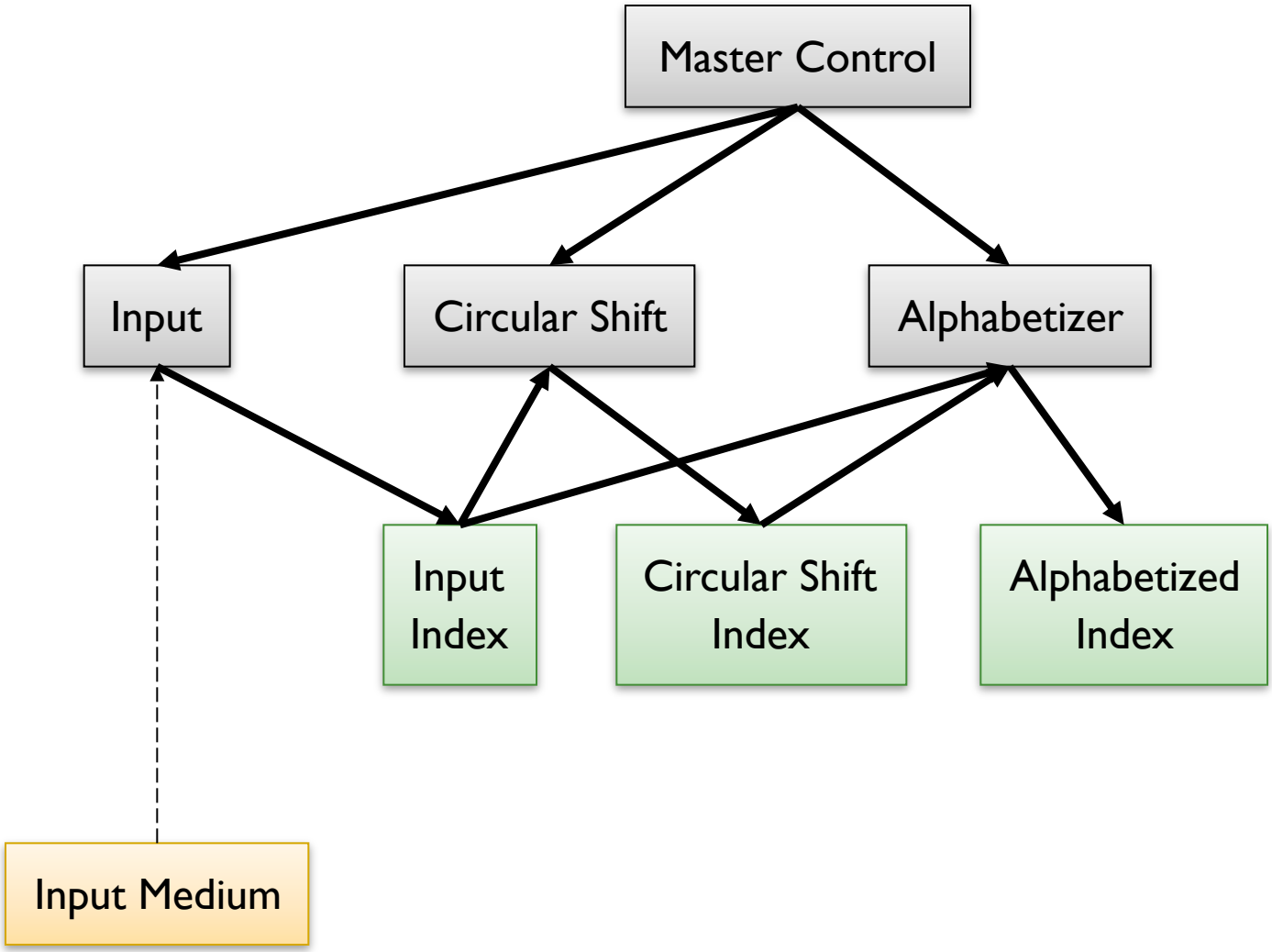
Modularization 1



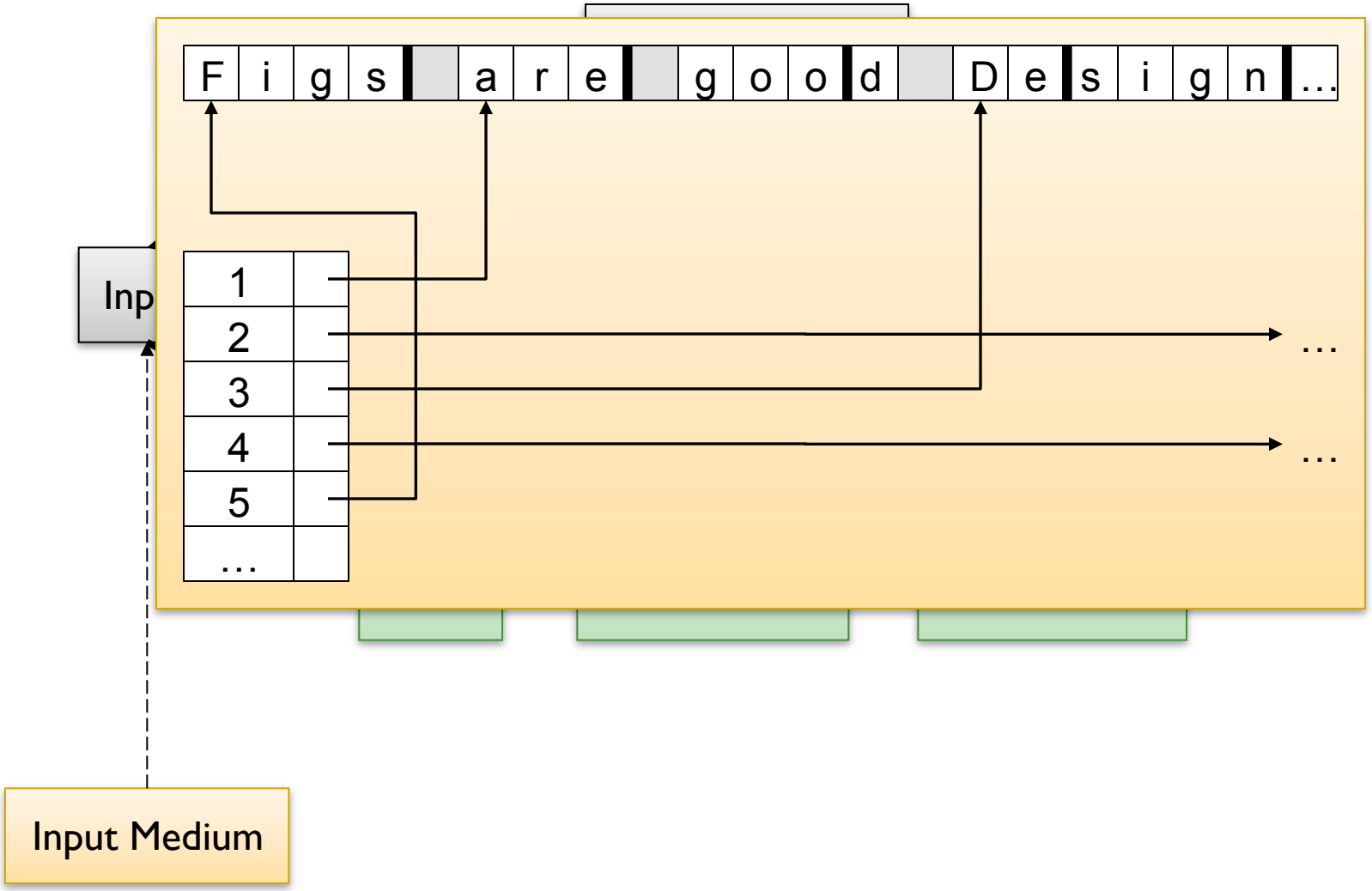
Modularization I



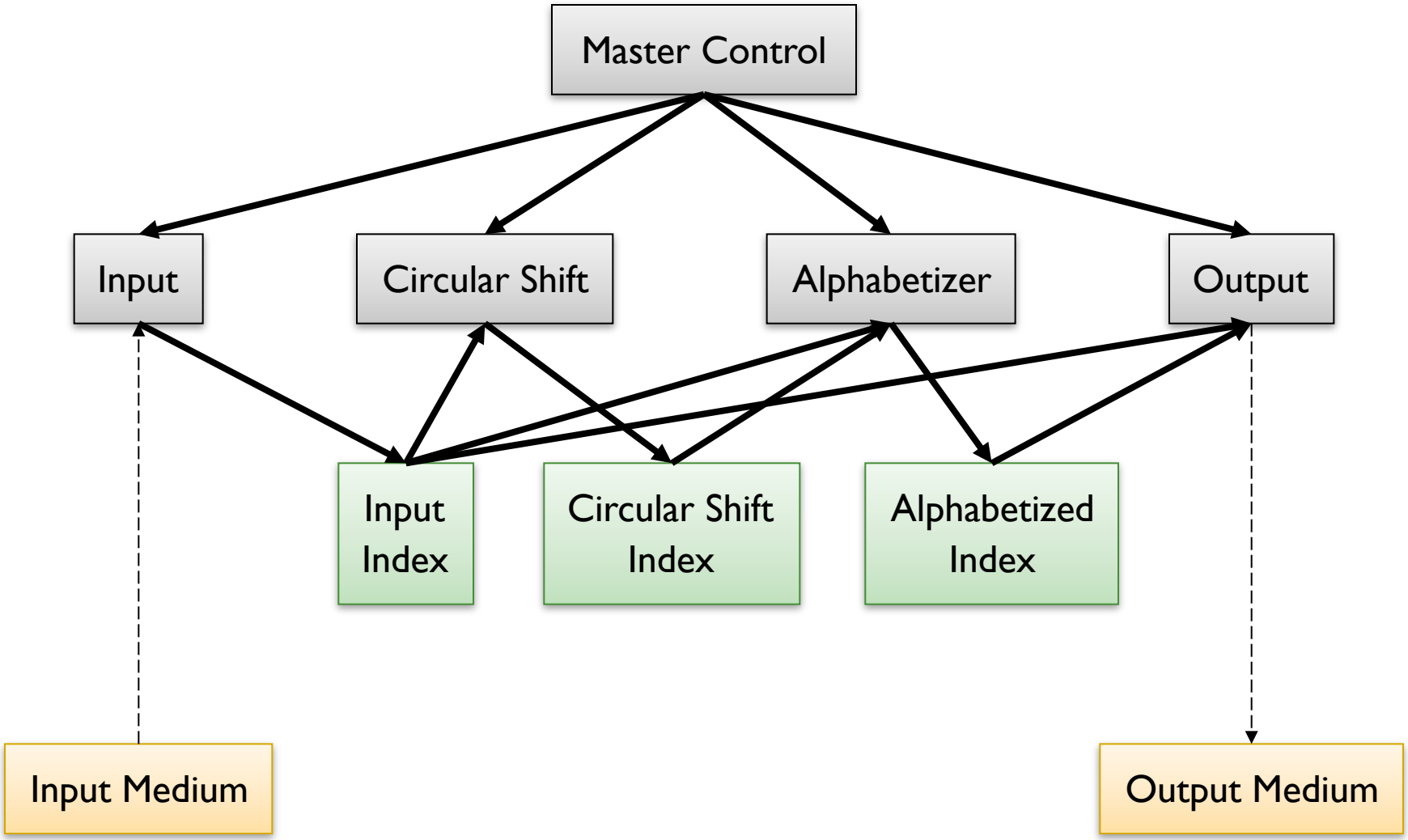
Modularization 1



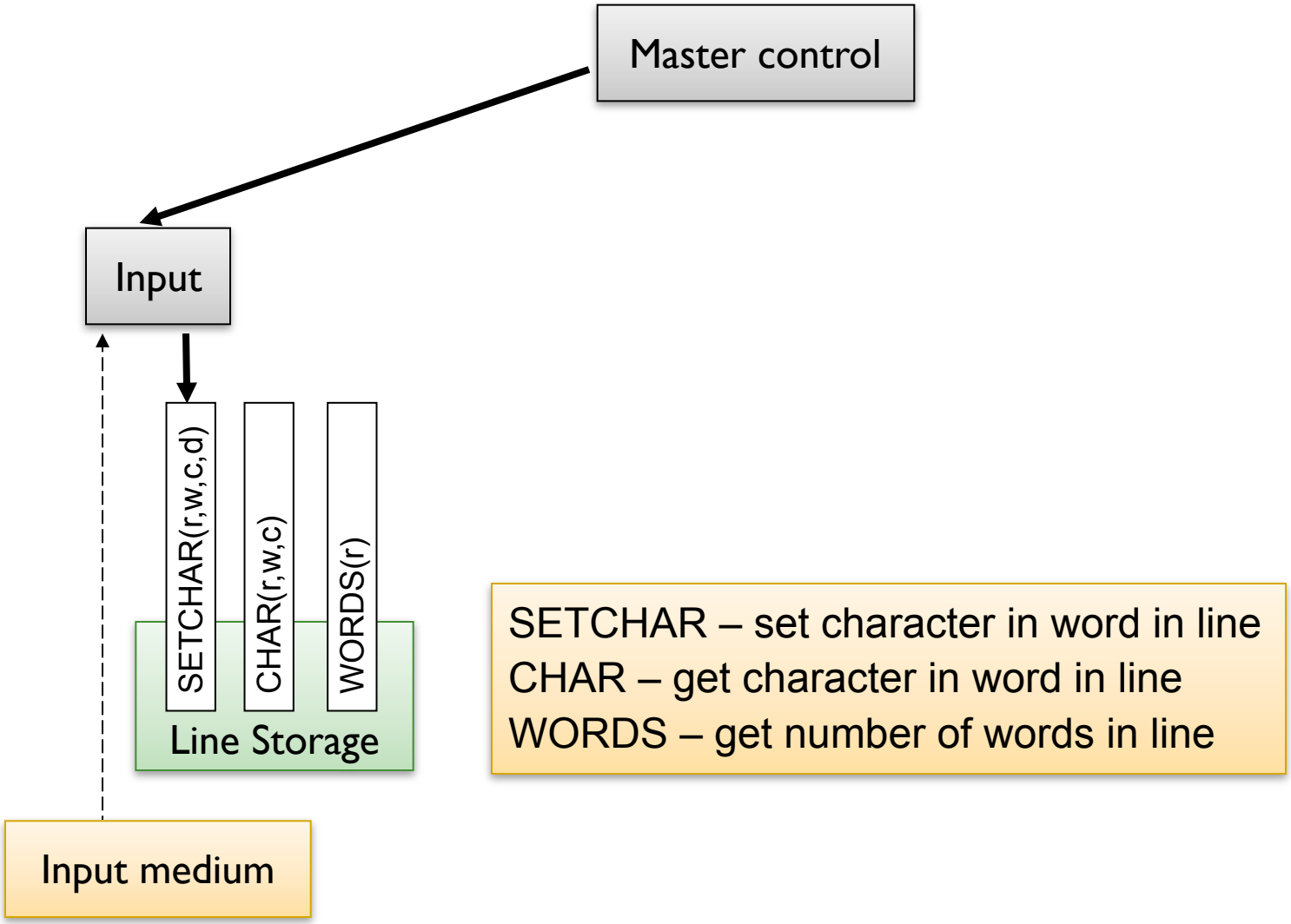
Modularization 1



Modularization 1

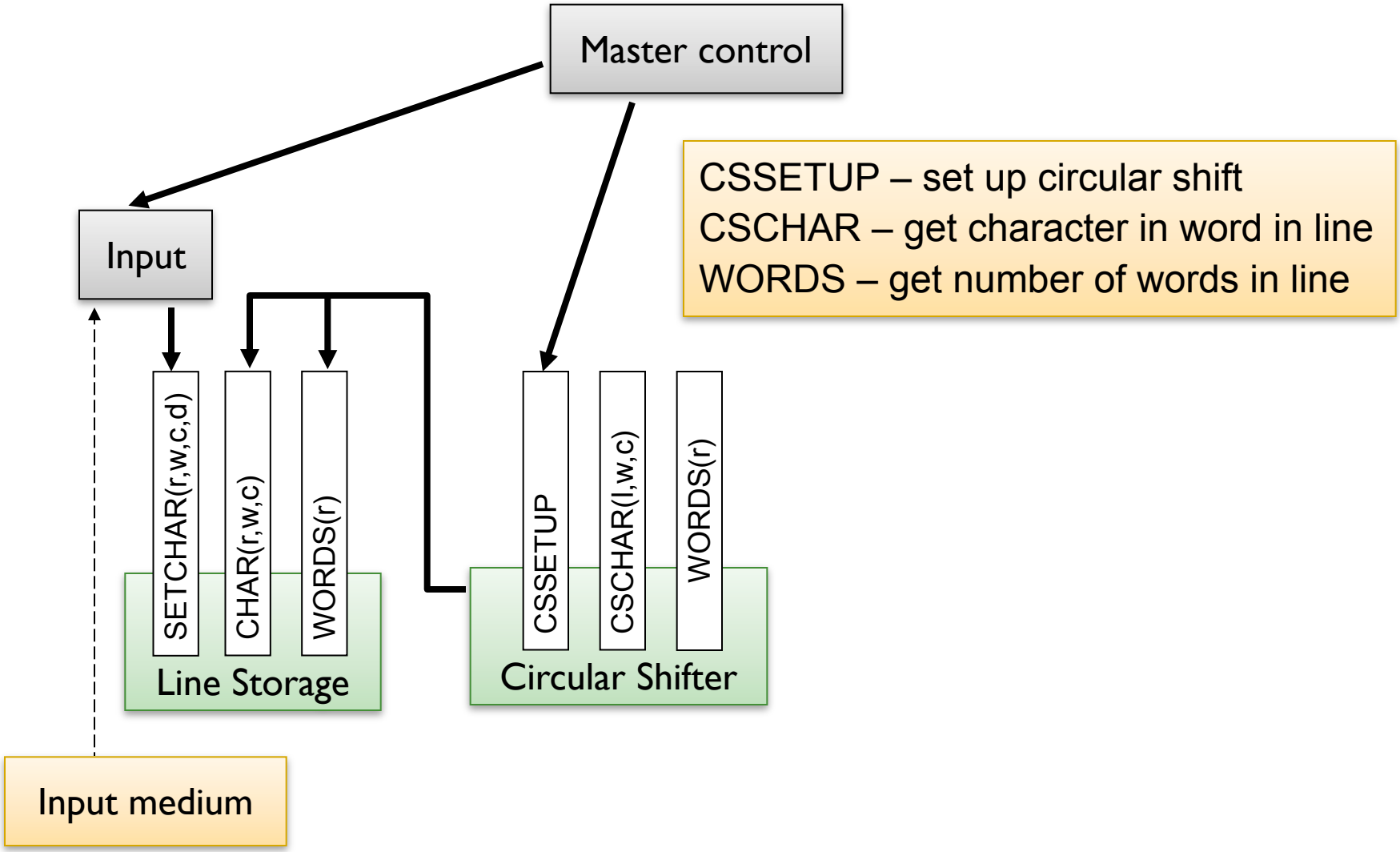


Modularization 2

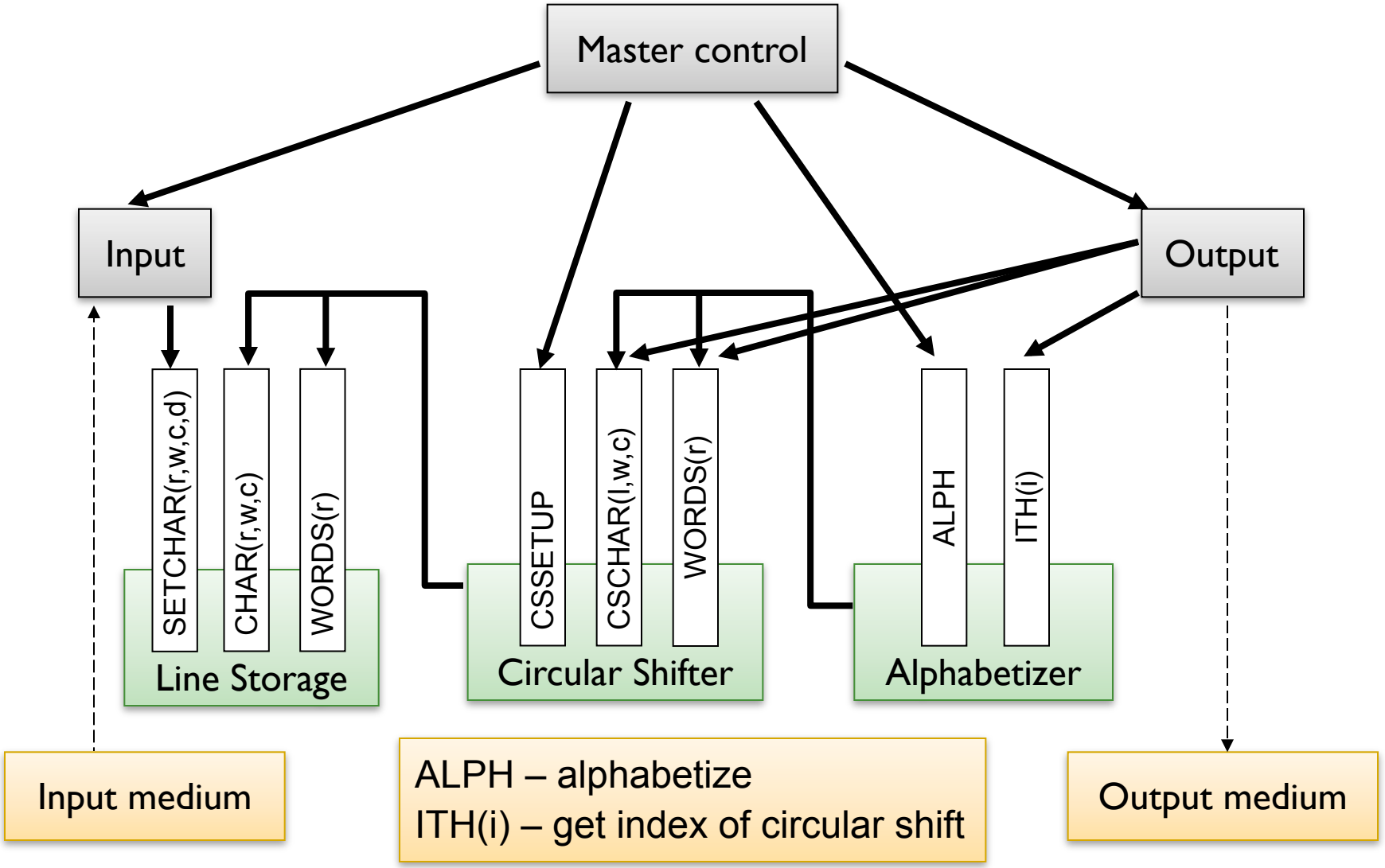


SETCHAR – set character in word in line
CHAR – get character in word in line
WORDS – get number of words in line

Modularization 2



Modularization 2



General Comparison

- General
  - Note: both systems might share the same data structures and the same algorithms
  - "Both will reduce the programming to the relatively independent programming of a number of small, manageable, programs"
    - Differences are in the way they are divided into work assignments
  - Systems are substantially different even if identical in the executable representation and result

Criteria for Decomposition

- Modularization 1
  - Each major step in the processing was a module
  - Flowchart

- Modularization 2
  - Information hiding
    - Each module has one or more "secrets”
    - Each module is characterized by its knowledge of design decisions which it hides from all others
  - Lines
    - How characters/lines are stored
  - Circular Shifter
    - Algorithm for shifting, storage for shifts
  - Alphabetizer
    - Algorithm for alpha, laziness of alpha

Weiss Quote

- "The way to evaluate a modular decomposition, particularly one that claims to rest on information hiding, is to ask what changes it accommodates." – [Hoffman and Weiss, 2001]

Changeability Comparison

- In Modularization 1, changes affected multiple modules
  - In some cases all modules
- In Modularization 2, changes affeteced only one module

Change	Modularization	
	1	2
Input format	Input	Input
All lines/characters stored in memory	All	Line Storage
Pack characters 4 to a word	All	Line Storage
Make an index for circular shifts rather than store them	Circular Shift Alphabetizer Output	Circular Shifter
Alphabetize once, rather than either 1) search for each item as needed, or 2) partially alphabetize, partially search	Alphabetizer Output	Alphabetizer

Independent Development

- Modularization 1
  - Must design all data structures before parallel work can proceed
  - Complex descriptions needed
- Modularization 2
  - Must design interfaces before parallel work can begin
  - Simple descriptions only

- Comprehensibility
  - Modularization 2 is better
    - Parnas subjective judgment

Benefits of Good Modular Design

- Independent Development
  - Since each module is independent, they can be developed independently at the same time
  - Shortened Development Time!

- Changeability, Product Flexibility & Reusability
  - Modules can be easily modified without affecting the rest of them
  - Modules can be easily replaced to add, enhance or change product capabilities

Benefits of Good Modular Design

- Comprehensibility
  - It is easier for programmers to fully understand the design of the entire product by individually studying the modules

Comprehensibility Quote

- "In many pieces of code the problem of disorientation is acute. People have no idea what each component of the code is for and they experience considerable mental stress as a result." – [Gabriel, 1995]

Historical Content in Present Context

- Paper is over 40 years old, but only some details might make this fact apparent:
  - Terminology
  - Previous concerns
  - Past design processes (flowcharts)
  - Code reuse (not a major point)
  - “could be developed in a week or two”

Terminology

- Parnas uses some terms that are not used anymore, or are used nowadays with different meanings, such as:
  - CORE
    - Then: main memory, general storage space
    - Now: internal functionality, internals
  - JOB
    - Then: Implied batch processing
    - Now: ???
  - Nowadays, we speak of memory in a more abstract way (data structures, etc). "Memory" was more often understood as referring to physical storage (addresses, records…)

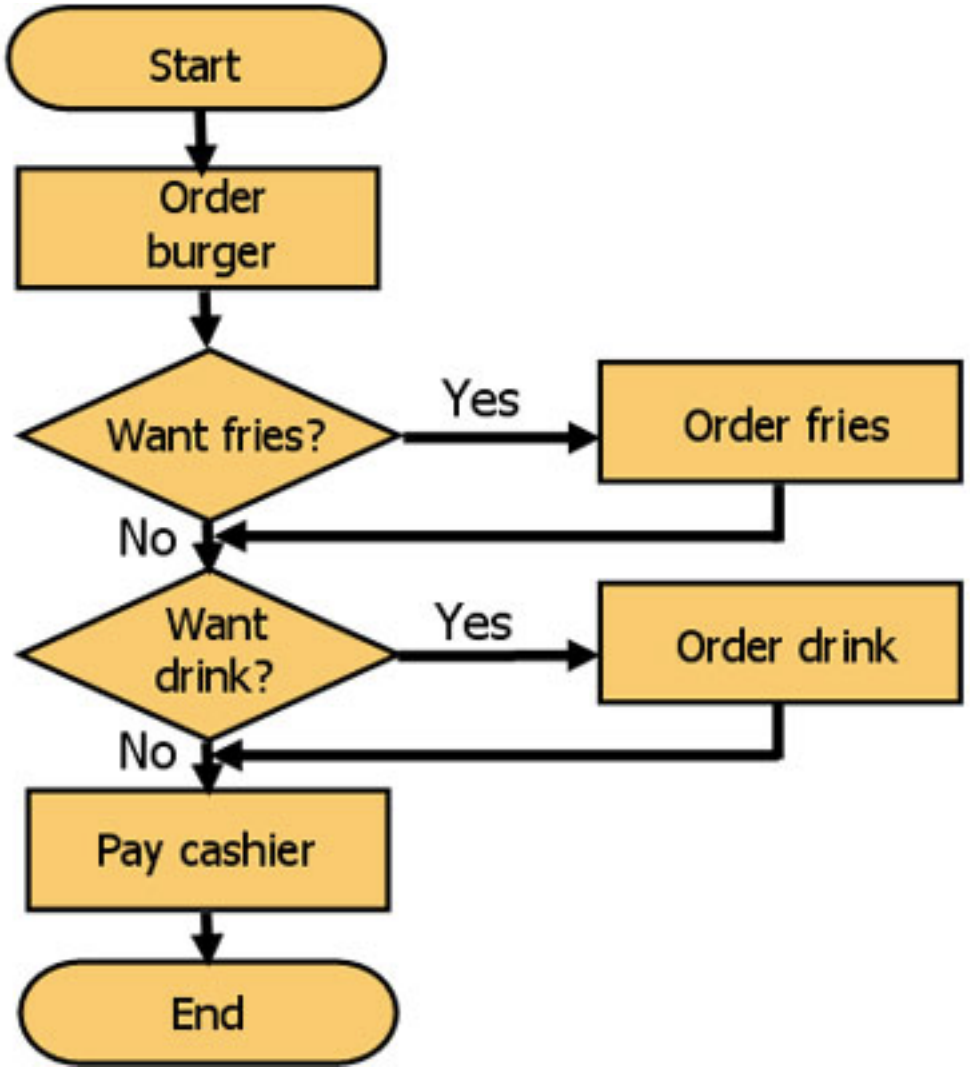
Previous Concerns

- Parnas mentions as "major advancements in the area of modular programming"…
  - The development of ASSEMBLERS
- Nowadays, we could mention higher level languages, mainly object-oriented languages that better:
  - "(1) allow one module to be written with little knowledge of the code in another module, and
  - (2) allow modules to be reassembled and replaced without reassembly of the whole system"
- Aspect Languages

Past Design Processes

- Use of flowcharts
  - When paper was written, the use of flowchart by programmers before was almost mandatory. With a flowchart in hand, the programmer would "move from there to a detailed implementation." This caused modularizations like the first one to be created
  - Parnas could see the problem with this approach and condemned it; a flowchart would work okay for a small system, but not with a larger on

Flowchats


Start
Order burger
Want fries?
Yes
Order fries
No
Want drink?
Yes
Order drink
No
Pay cashier
End

Code Reuse

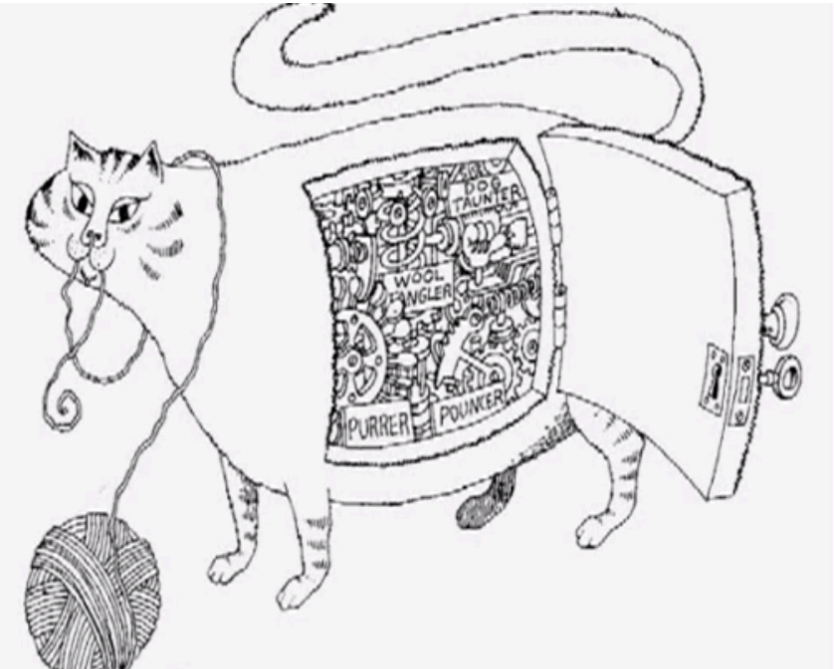
- Parnas does not emphasize code reuse so much in this paper. The reason might be the nature of programs written in assembly or lower-level languages programmers (not very portable/reusable)
- If the paper were to be reviewed by Parnas, reuse would likely be a point he would emphasize more
- It is important to notice that these points do not disturb the current relevance of Parnas' ideas

Effects on Current Programming

- "Fathered" key ideas supporting OOP
  - Information hiding
  - Encapsulation before functional relations
  - Easier understandability/maintainability
- Design more important than implementation
  - Good design leads to good implementation
  - Proper design allows for different implementations (easily modifiable)

What is called well-designed software?

- The most important factor to distinguish a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details from other modules.



Encapsulation hides the details of the implementation of an object.

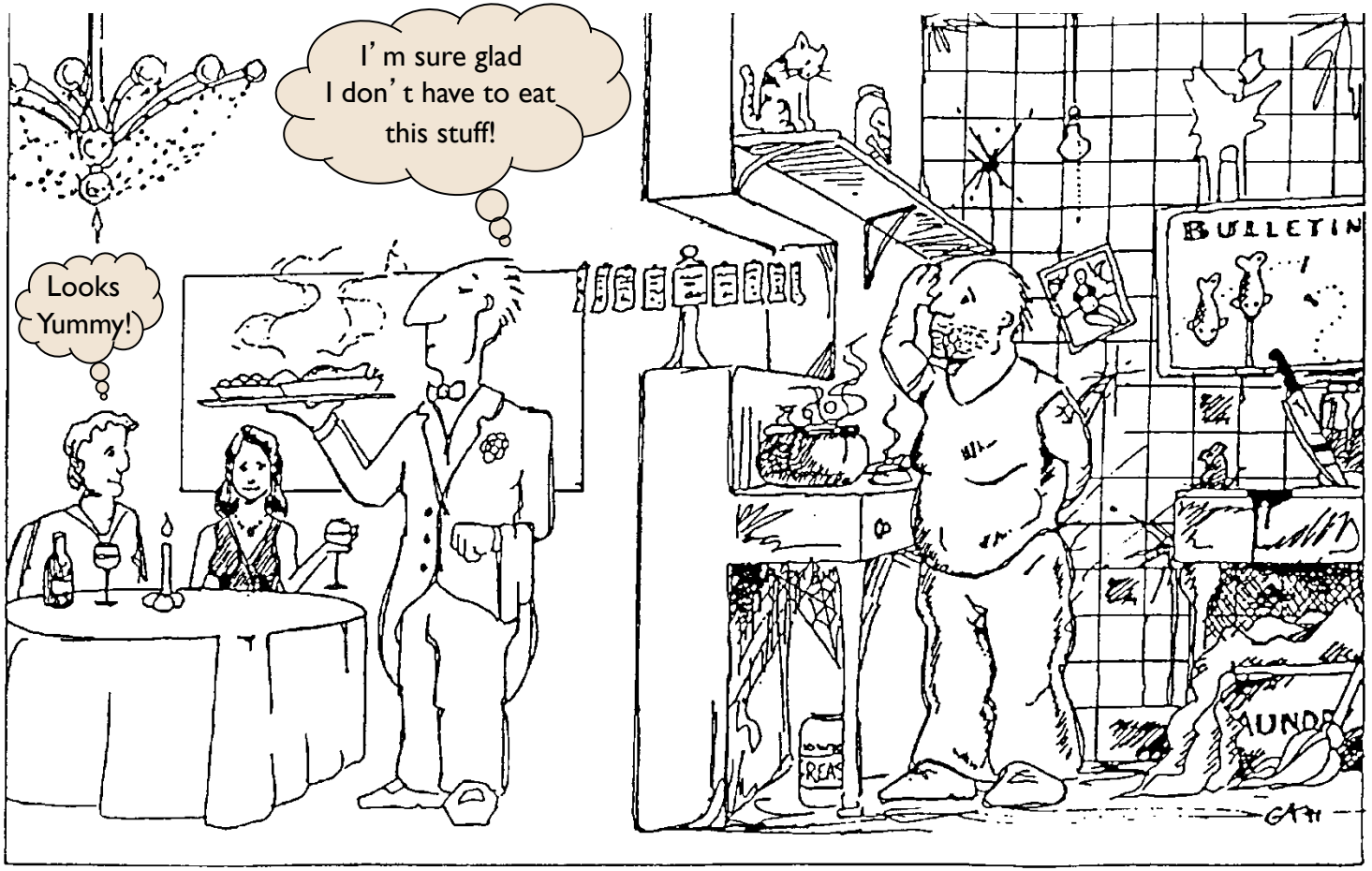
Information Hiding

- Hides the design decisions and implementation details from other modules

- "The second decomposition was made using 'information hiding' ... as a criterion. The modules no longer correspond to steps in the processing. ... Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings." – [Parnas, 1972b]

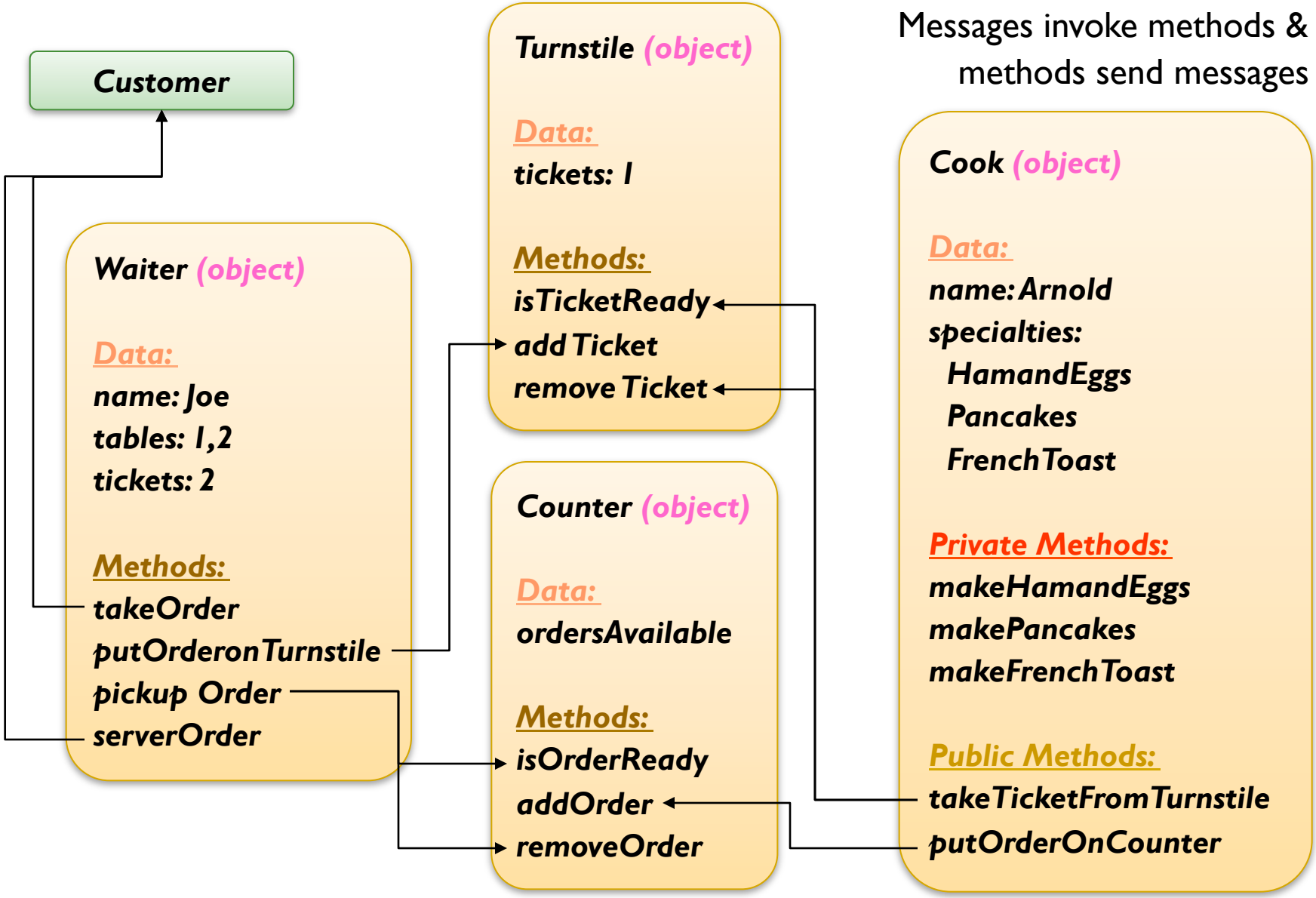
- "... the purpose of hiding is to make inaccessible certain details that should not affect other parts of a system." – [Ross et al., 1975]

The Restaurant

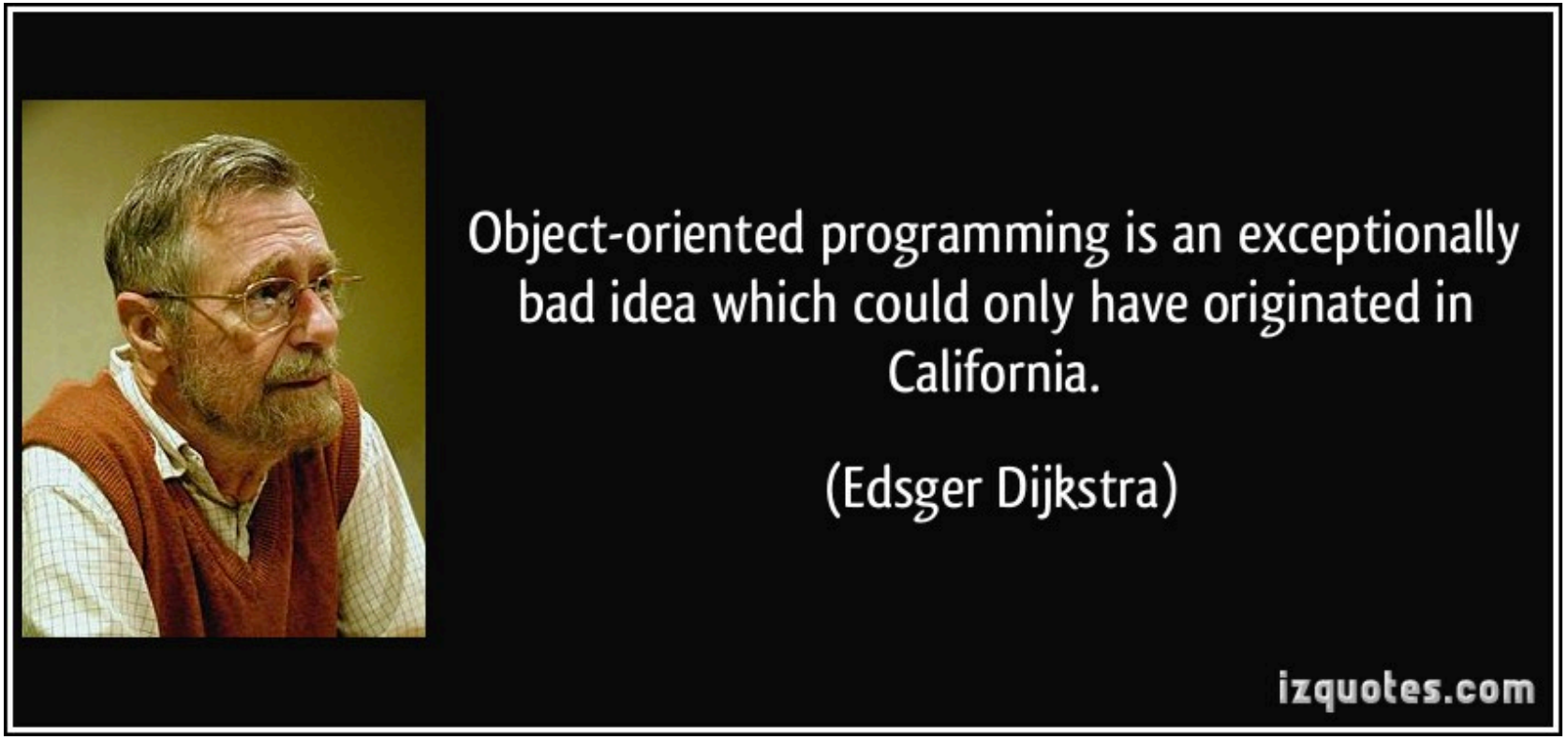


Once a message has been passed to an object, objects outside can't know and don't care how processing takes place.

The Restaurant



Disadvantages of OOP



Disadvantages of OOP

- Not every problem can be considered as objects
  - e.g., Procedure is not an object
- Steep learning curve
  - PP (Top-down) is more straightforward for us to think
- Large program size
- Slower programs / Less efficient