MENOUFIA UNIVERSITY
جامعة المنوفية
منارة المعرفة فى قلب الدلتا

Software Engineering

Dr. Mohammed Badawy

mbmbadawy@yahoo.com
mohamed.badawi@el-eng.menofia.edu.eg
Room: 417, Ext. 7332

13 May 2016

Chapter 5

System Design

Contents

Introduction
System Architecture

- Any complex system is composed of subsystems that interact under the control of system design such that the system provides the expected behavior.
- When designing such a system, therefore, the logical approach is to identify the subsystems that should compose the system, the interfaces of these subsystems, and the rules for interaction between the subsystems.
- This is what software architecture aims to do.

Introduction

- Design activity begins when the requirements document for the software to be developed is available and the architecture has been designed
- Design focuses on the what we have called the module view
- A software system: A set of modules with clearly defined behavior which interact with each other in a defined manner to produce some behavior or services for its environment

Introduction

- The design process for software systems often has two levels:
  - System design or top-level design
    - deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected
  - Detailed design or logic design
    - the internal design of the modules, or how the specifications of the module can be satisfied

Introduction

- Software Design methods
  - Function-Oriented Methods (Structured Design Methodology)
    - In a function-oriented design approach, a system is viewed as a <u>transformation function</u>, transforming the inputs to the desired outputs
    - The purpose of the design phase is to specify the components for this transformation function
  - Object-Oriented Methods

Introduction

- The goal of the design process is not simply to produce a design for the system
- Instead, the goal is to find the ***best possible design*** within the limitations imposed by the requirements and the physical and social environment in which the system will operate

Design Objectives/Properties

- The various desirable properties or objectives of software design are:
  - Correctness (satisfies the requirements of the system)
  - Verifiability (how easily the correctness of the design can be checked)
  - Completeness (all the different components of the design should be verified)
  - Traceability (the entire design element be traceable to the requirements.)
  - Efficiency (the proper use of scarce resources by the system)
  - Simplicity

Design Principles

- The three design principles are as follows:
  - Problem partitioning
  - Abstraction
  - Top-down and Bottom-up design

Horizontal Partitioning

- The simplest approach to horizontal partitioning defines three partitions: Input, Data Transformation (often called processing), and Output
- Partitioning benefits:
    - Software that is easier to test
    - Software that is easier to maintain
    - Software that is easier to extend
    - Propagation of fewer side effects

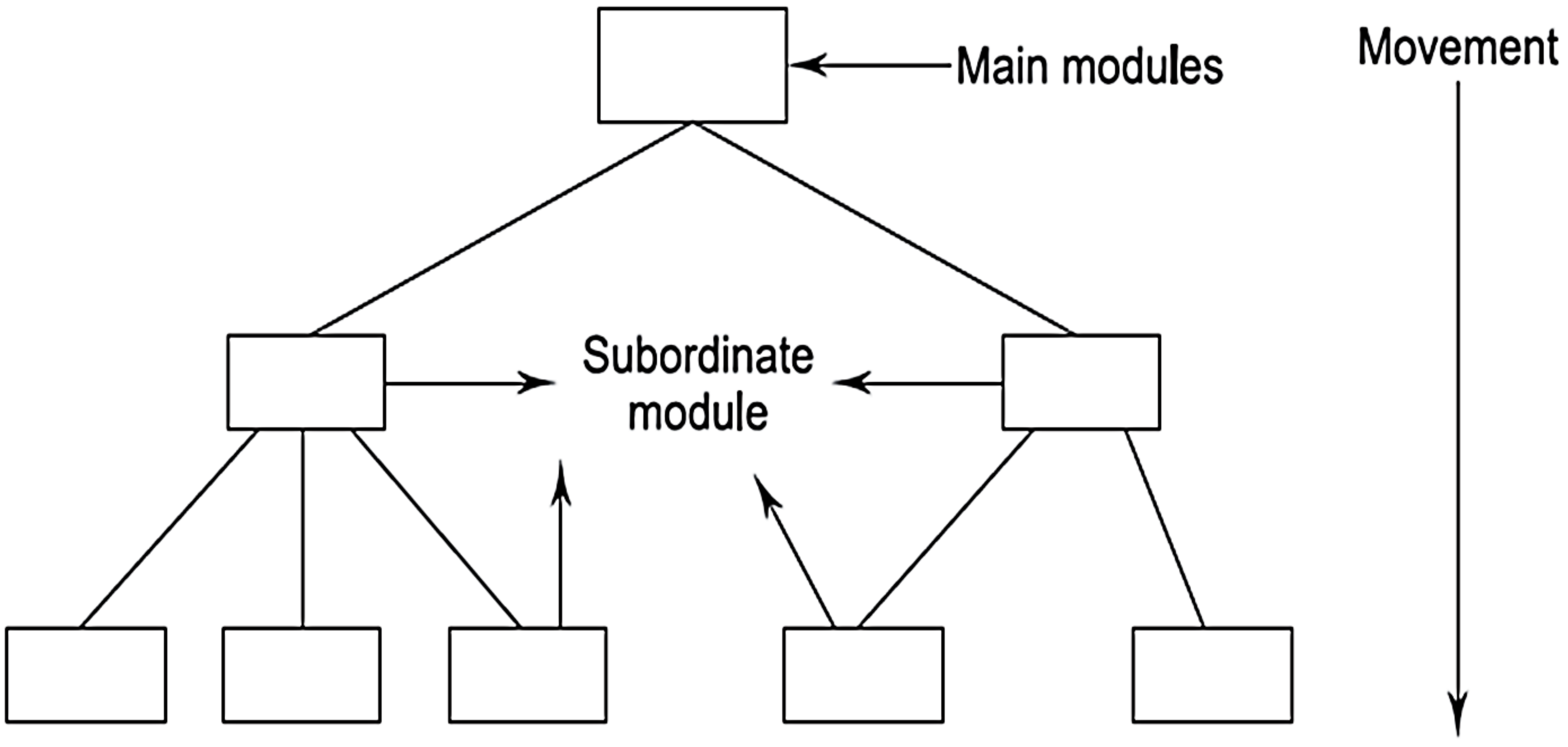
Vertical Partitioning

- Vertical partitioning, often called ***factoring***, suggests that control and work should be distributed from top-down in the program structure
- Top-level modules should perform control functions and do actual processing work
- Modules that reside low in the structure should be the workers, performing all input output tasks

Abstraction

- An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior
- Partitioning essentially is the exercise in determining the components of a system
- These components are not isolated from each other, but interact with other component
- Allowing the designer to concentrate on one component at a time, abstraction of other components is used

Top-Down Approach

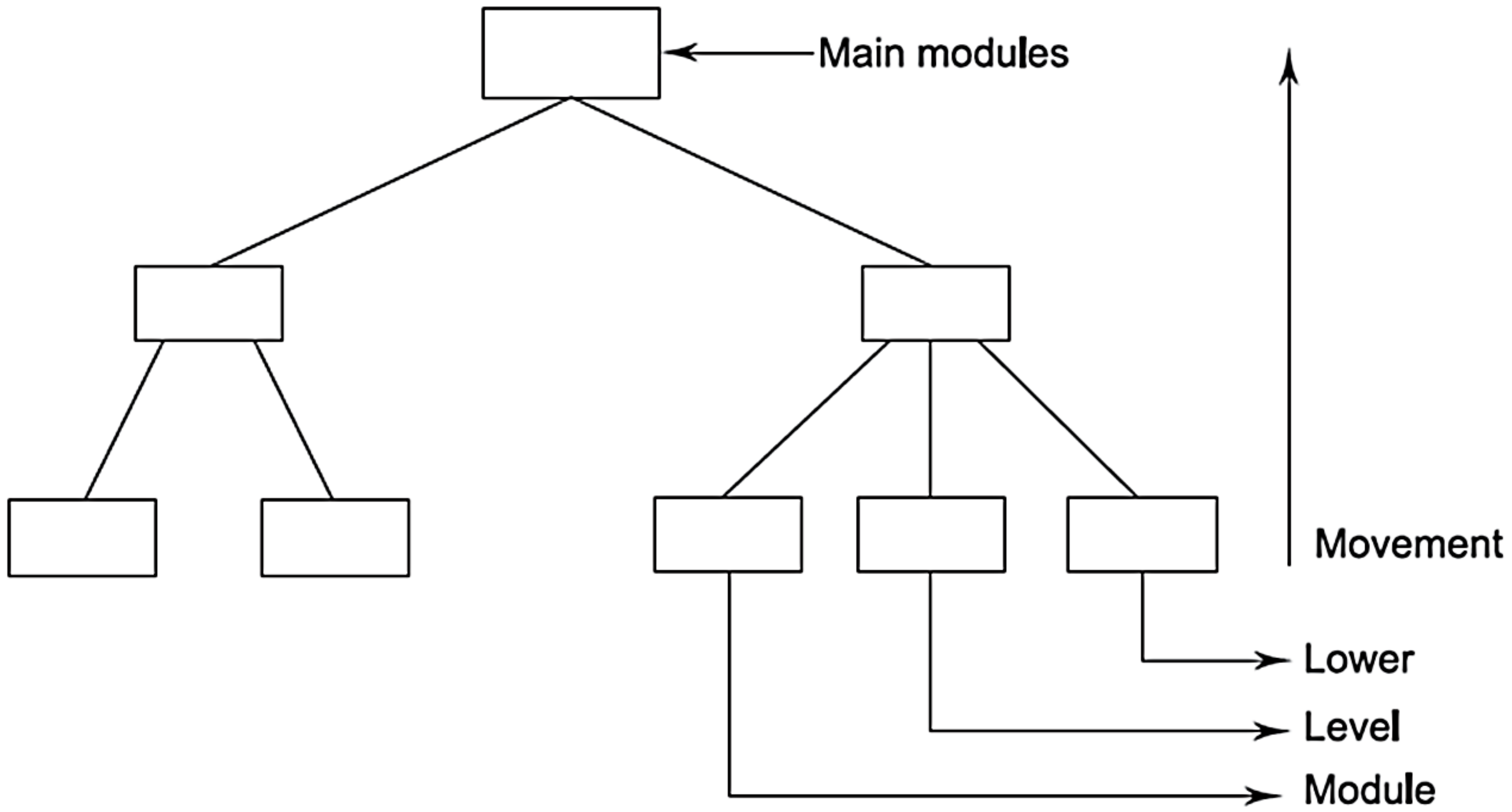


(Movement in top-down approach)

Top-Down Approach

- Components and iterating until the desired level of a top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level detail is achieved
- Top-down design methods often result in some form of stepwise refinement
- Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed
- Most design methodologies are based on the top-down approach

Bottom-Up Approach



(Movement in bottom-up approach)

Bottom-Up Approach

- A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components
- Bottom-up methods work with layers of abstraction. Starting from the very bottom, operations that provide a layer of abstraction are implemented

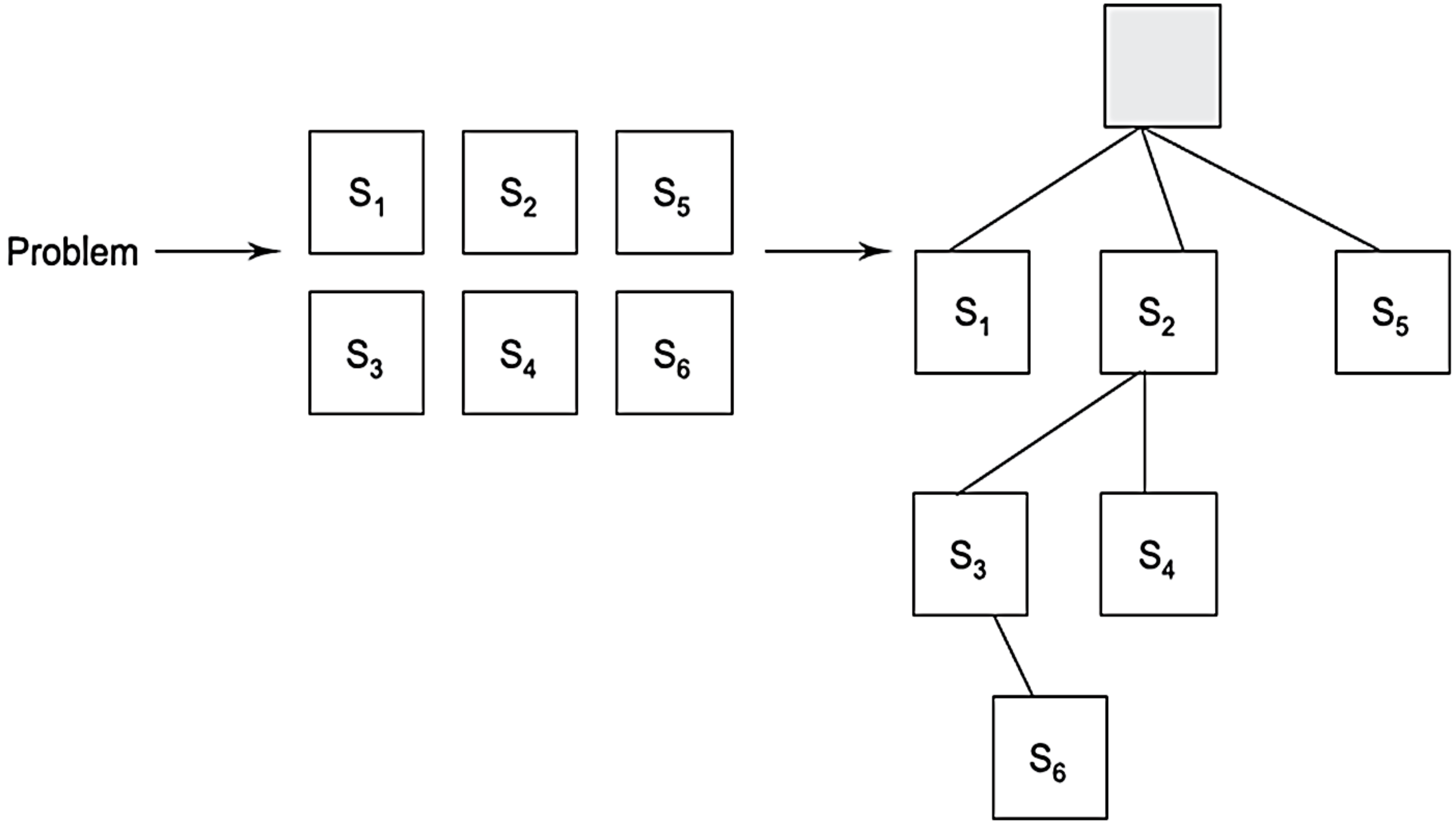
Bottom-Up versus Top-Down

- A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch
- However, if a system is to be built from an existing system, a bottom-up approach is more suitable, as it starts from some existing components

Architectural Design

- Large systems are always decomposed into subsystems that provide some related set of services
- Architectural design is the initial design process of identifying these subsystems and establishing a framework for subsystem control and communication
- The number of levels of a component in the structure is called ***depth*** and the number of components across the horizontal section is called ***width***

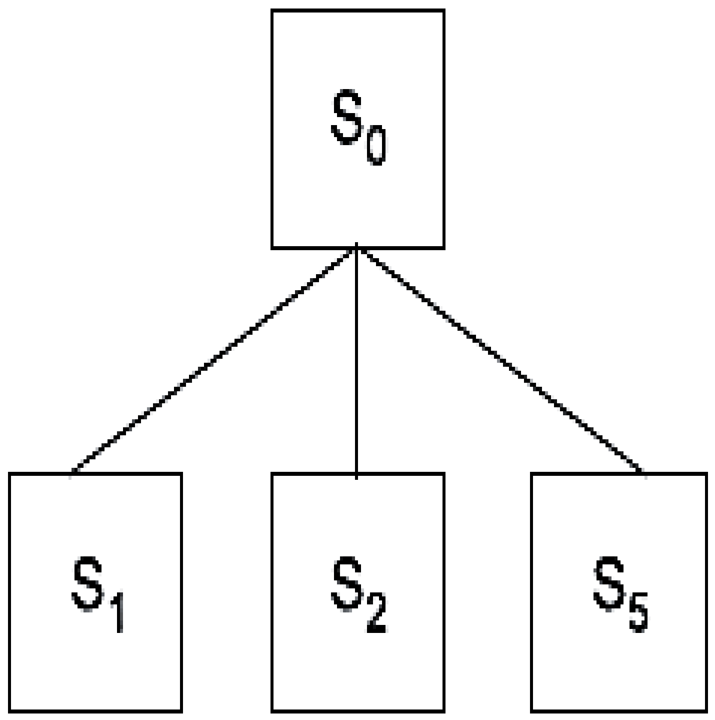
Objectives of Architectural Design

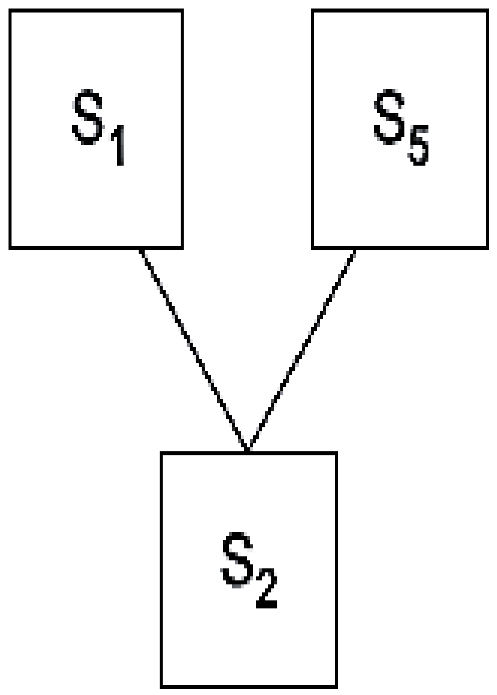


Architectural Design

- The number of components, which controls the component, is called fan-in, i.e., the number of incoming edges to a component
- The number of components that are controlled by the module is called ***fan-out***, i.e., the number of outgoing edges

Fan-in and Fan-out





Modularization

- System is considered modular if it consists of discrete components so that each component can be implemented separately, and a change to one component has minimal impact on other components

Advantages of Modular Systems

- Modular systems are <u>easier to understand</u> because their parts are functionally independent
- Modular systems are <u>easier to document</u> because each part can be documented as an independent unit
- <u>Programming individual modules is easier</u> because the programmer can focus on just one small, simple problem
- <u>Testing and debugging individual modules is easier</u>

Advantages of Modular Systems

- Bugs are easier to isolate and understand, and they can be fixed without fear of introducing problems outside the module
- Well-composed modules are more reusable

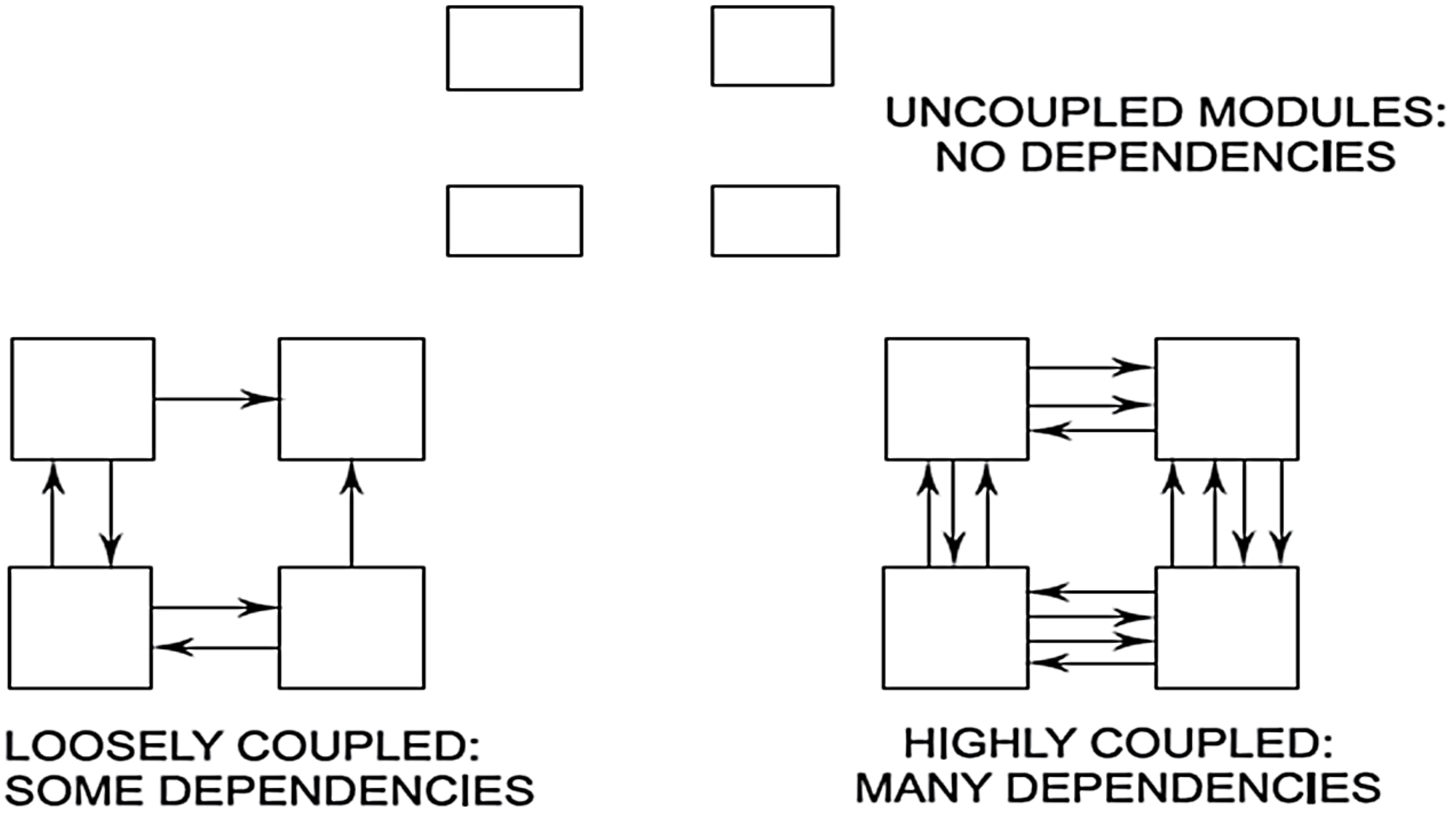
Coupling and Cohesion

- The coupling between two modules indicates the degree of interdependence between them
- The cohesion of a component is a measure of the closeness of the relationships between its components

Coupling

- **Highly Coupled**: When the modules are highly dependent on each other
- **Loosely Coupled**: When the modules are dependent on each other but the interconnection among them is weak
- **Uncoupled**: When the different modules have no interconnection among them

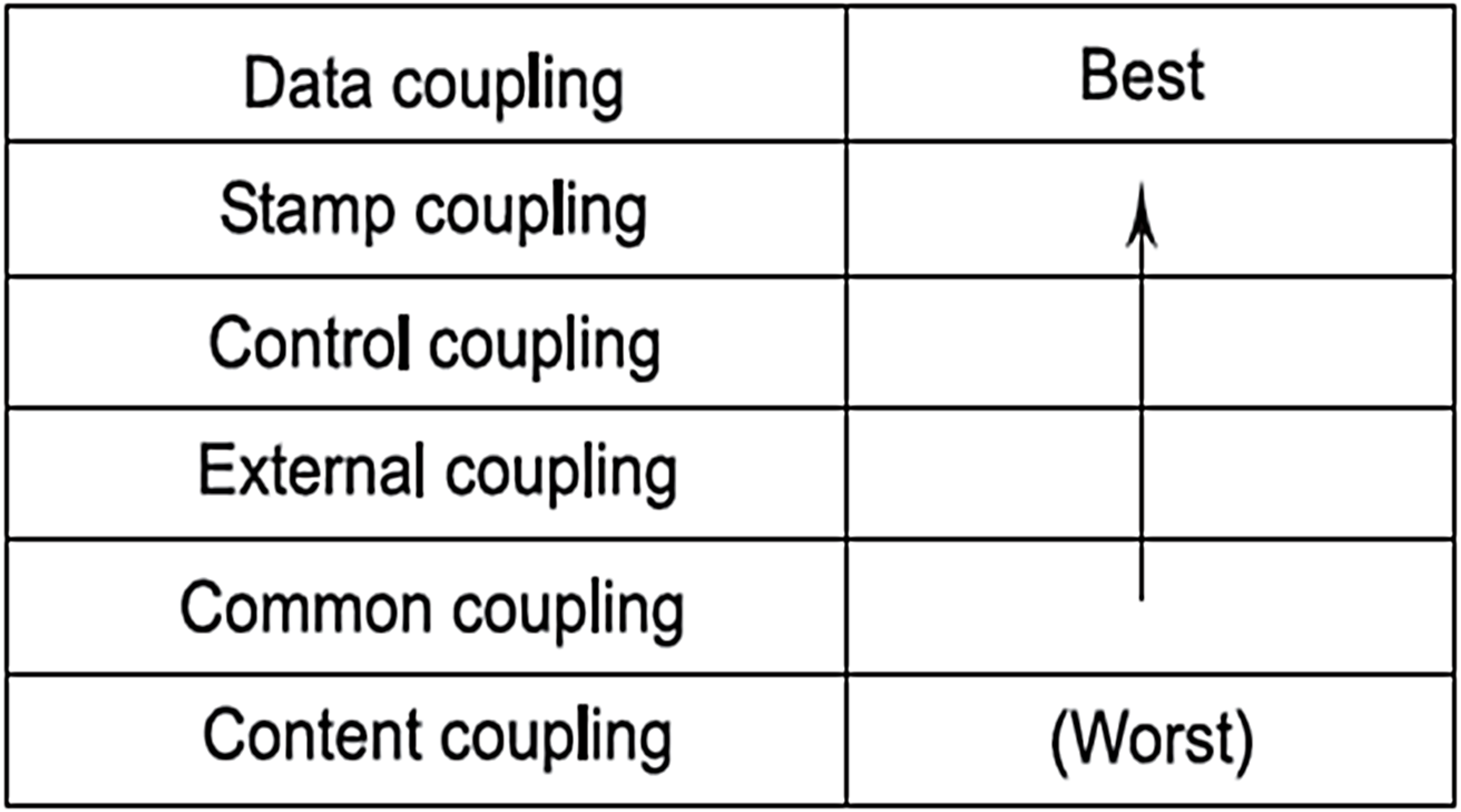
Coupling


UNCOUPLED MODULES:
NO DEPENDENCIES
LOOSELY COUPLED:
SOME DEPENDENCIES
HIGHLY COUPLED:
MANY DEPENDENCIES

Types of Coupling

- Different types of couplings include content, common, external, control, stamp, and data
- The strength of a coupling from the lowest coupling (best) to the highest coupling (worst) is given in the figure below

Types of Coupling



Data coupling	Best
Stamp coupling	↑
Control coupling	
External coupling	
Common coupling	
Content coupling	(Worst)

Data Coupling

- Two modules are data coupled if they communicate using an elementary data item that is passed as a parameter between the two; for example, an integer, a float, etc.

Stamp Coupling

- Two modules are stamp coupled if they communicate using a composite data item, such as a record, structure, etc.
- For example, passing a record in PASCAL or a structure variable in C or an object in C++ language to a module

Control Coupling

- Control coupling exists between two modules if data from one module is used to direct the order of instruction execution in another

External Coupling

- It occurs when modules are tied to an environment external to software
- An example of external coupling is a program where one part of the code reads a specific file format that another part of the code wrote
- Both pieces need to know the format so when one changes, the other must as well

Common Coupling

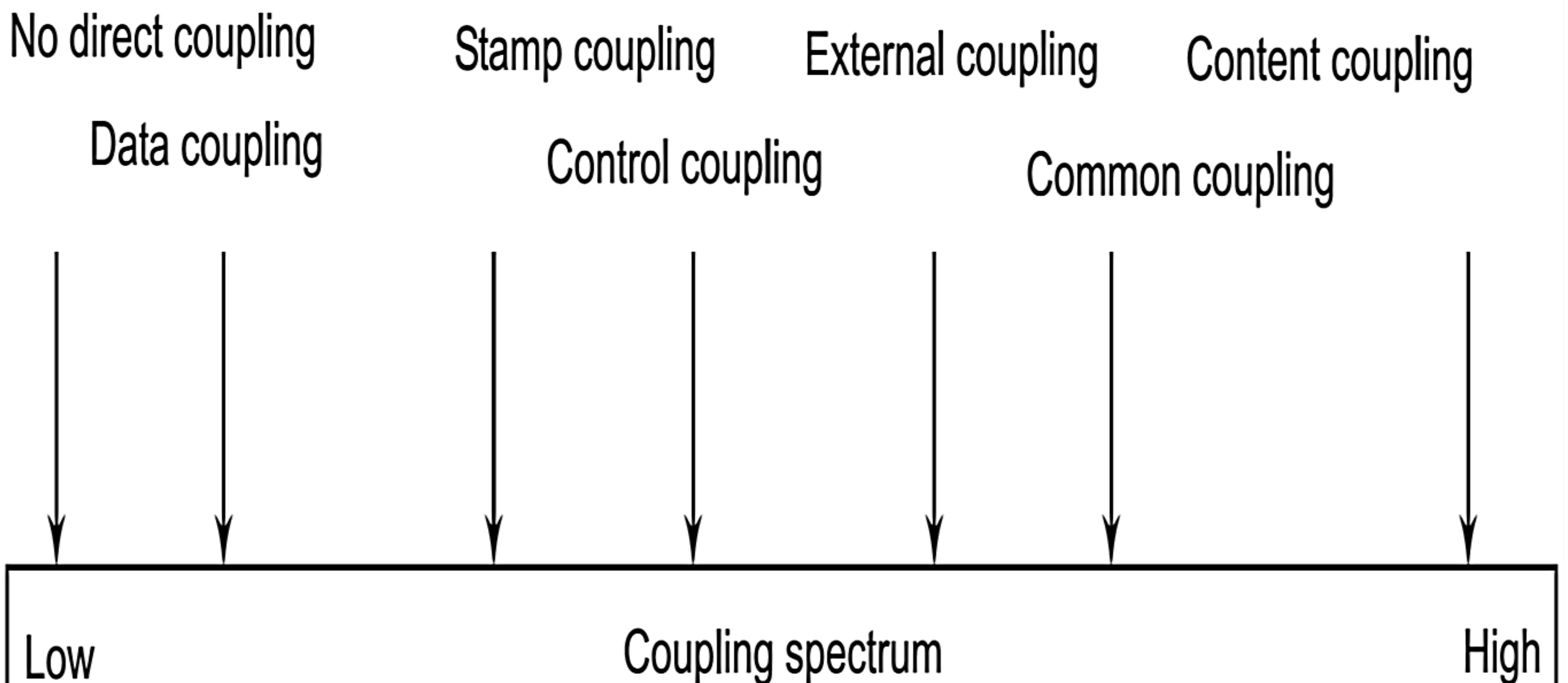
- Two modules are common coupled if they share some global data items (e.g., Global variables)
- Diagnosing problems in structures with considerable common coupling is time-consuming and difficult

Content Coupling

- Content coupling exists between two modules if their code is shared; for example, a branch from one module into another module
- It is when one module directly refers to the inner workings of another module. Modules are highly interdependent on each other

Coupling

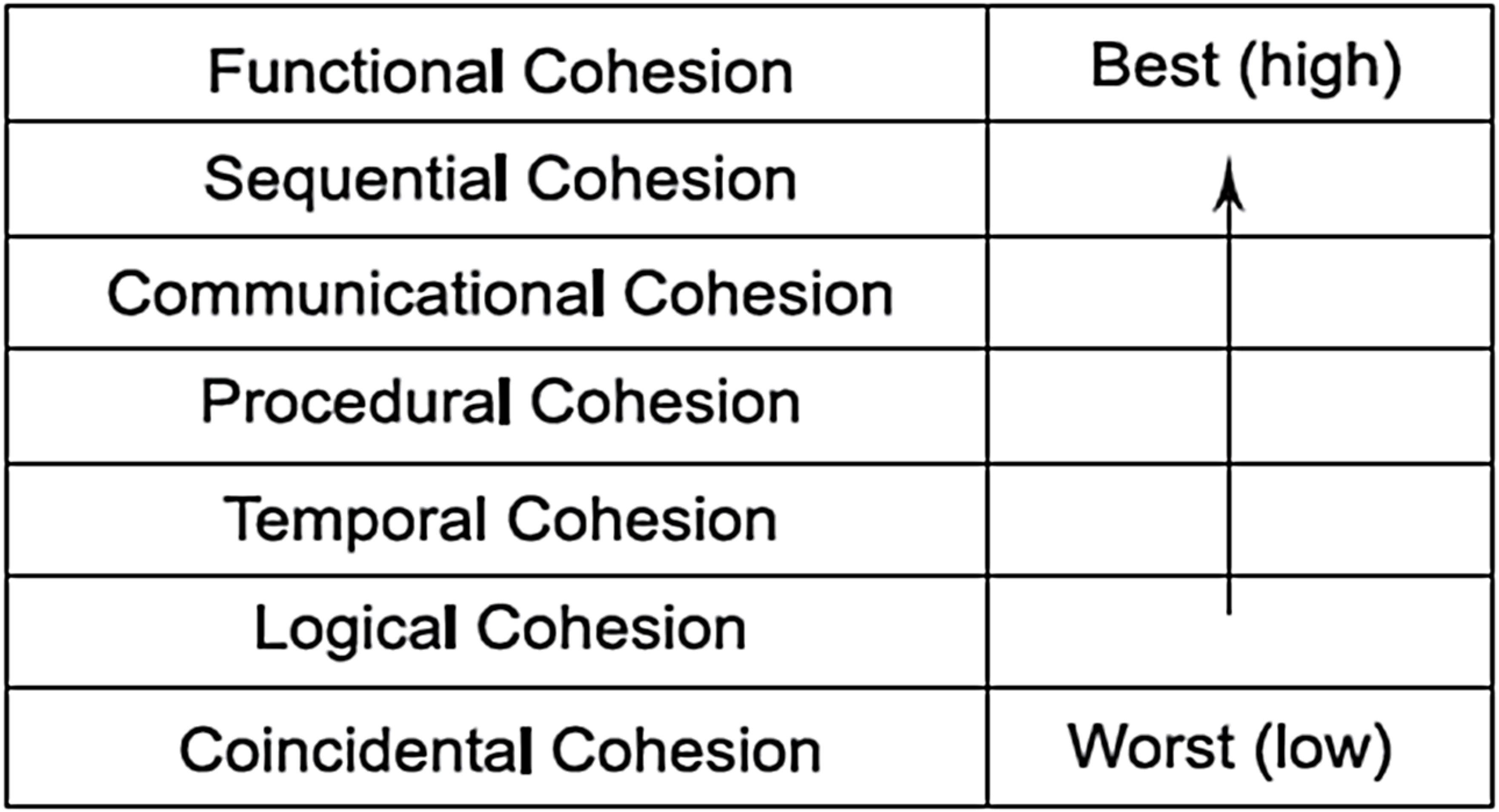
- having high coupling cannot be developed indepHigh coupling among modules not only makes a design difficult to understand and maintain, but it also increases development effort as the modules endently by different team members



Cohesion

- A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program
- An important design objective is to maximize the module cohesion and minimize the module coupling

Types of Cohesion



Functional Cohesion	Best (high)
Sequential Cohesion	
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Worst (low)

Types of Cohesion

- **Functional Cohesion**
  - Functional cohesion is said to exist if <u>different elements of a module cooperate to achieve a single function</u>
  - When a module displays functional cohesion, we can describe it using a single sentence
- **Sequential Cohesion**
  - A module is said to possess sequential cohesion if the <u>elements of a module form the parts of a sequence</u>, where the output from one element of the sequence is input to the next

Types of Cohesion

- **Communication Cohesion**
  - A module is said to have communicational cohesion if all the functions of the module <u>refer to or update the same data structure</u>; for example, the set of functions defined on an array
- **Procedural Cohesion**
  - A module is said to possess procedural cohesion if the <u>set of functions of the module are all part of a procedure</u> (algorithm) in which a certain sequence of steps has to be carried out for achieving an objective

Types of Cohesion

- **Temporal Cohesion**
  - When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion
  - The set of functions responsible for initialization, start-up, shutdown of some process, etc., exhibit temporal cohesion

Types of Cohesion

- **Logical Cohesion**
  - A module is said to be logically cohesive if all elements of the module perform similar operations; for example, error handling, data input, data output, etc.
  - An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module
- **Coincidental Cohesion**
  - A module is said to have coincidental cohesion if it performs a set of tasks that relate to each other very loosely
  - In this case, the module contains a random collection of functions

How does one determine the cohesion level of a module ?

- There is no mathematical formula that can be used
- A useful technique for determining if a module has functional cohesion is to write a sentence that describes, fully and accurately, the function or purpose of the module
- If the sentence contains a comma, or it has more than one verb, the module is probably performing more than one function, and it probably has sequential or communicational cohesion

How does one determine the cohesion level of a module ?

- If the sentence contains words relating to time, like "first," "next," "when," and "after" the module probably has sequential or temporal cohesion
- If the predicate of the sentence does not contain a single specific object following the verb (such as "edit all data") the module probably has logical cohesion
- Words like "initialize" imply temporal cohesion
- Modules with functional cohesion can always be described by a simple sentence

Relationship between Coupling and Cohesion

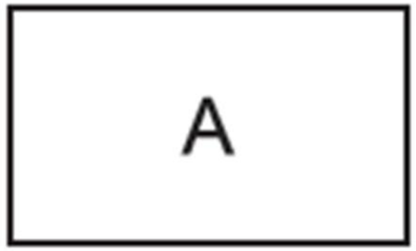
- A software engineer must design the modules with the goal of high cohesion and low coupling

Structure Charts

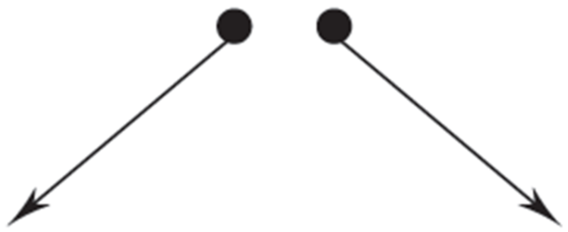
- The structure chart is one of the most commonly used methods for system design
- <u>Structure charts are used during architectural design</u>
- The structure of a program is made up of the modules of that program together with the interconnections between modules
- The focus is on representing the hierarchy of modules

Basic Building Blocks of a Structure Chart

- A rectangular box represents a module
- Usually a rectangular box is annotated with the name of the module it represent

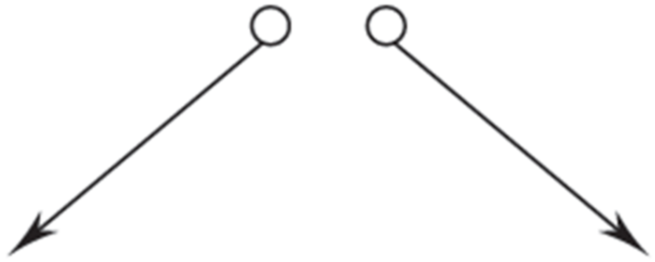


- An arrow connecting two modules implies that during program execution, control is passed from one module to the other in the direction of the connecting arrow

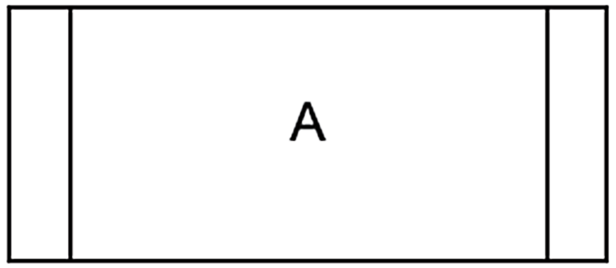


Basic Building Blocks of a Structure Chart

- Data-flow arrows represent that the named data passes from one module to the other in the direction of the arrow

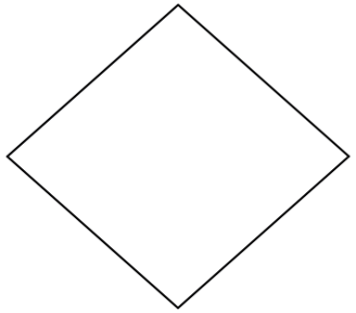


- Library modules are usually represented by a rectangle with double edges

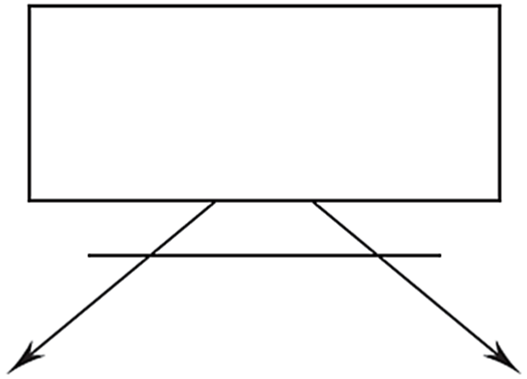


Basic Building Blocks of a Structure Chart

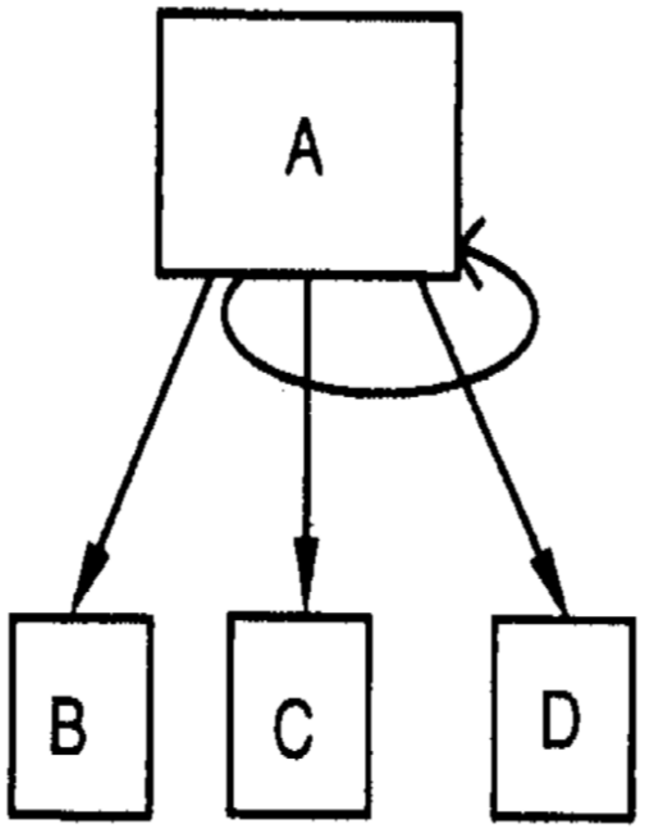
- The diamond symbol represents that one module out of several modules connected with the diamond symbol are invoked depending on the condition satisfied, which is written in the diamond symbol

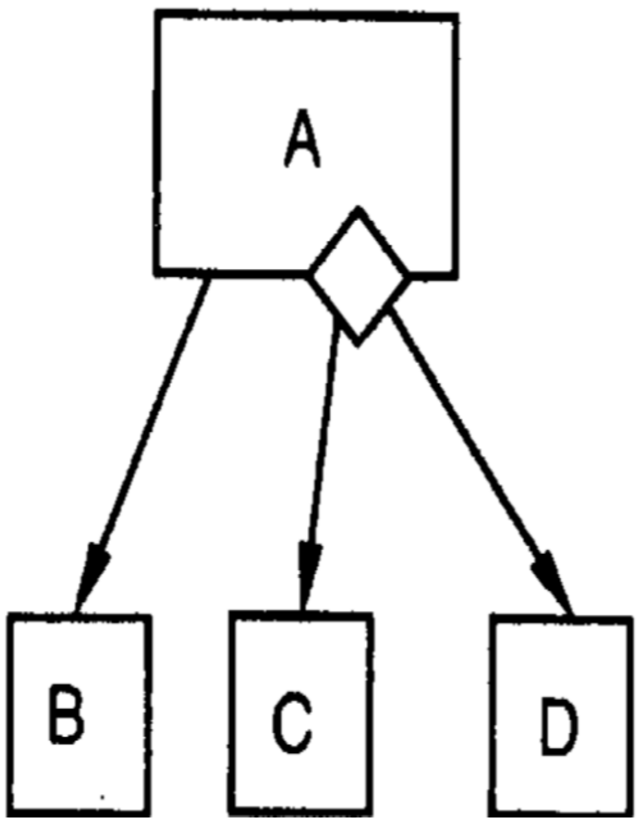


- loop around the control-flow arrows denotes that the respective modules are invoked repeatedly

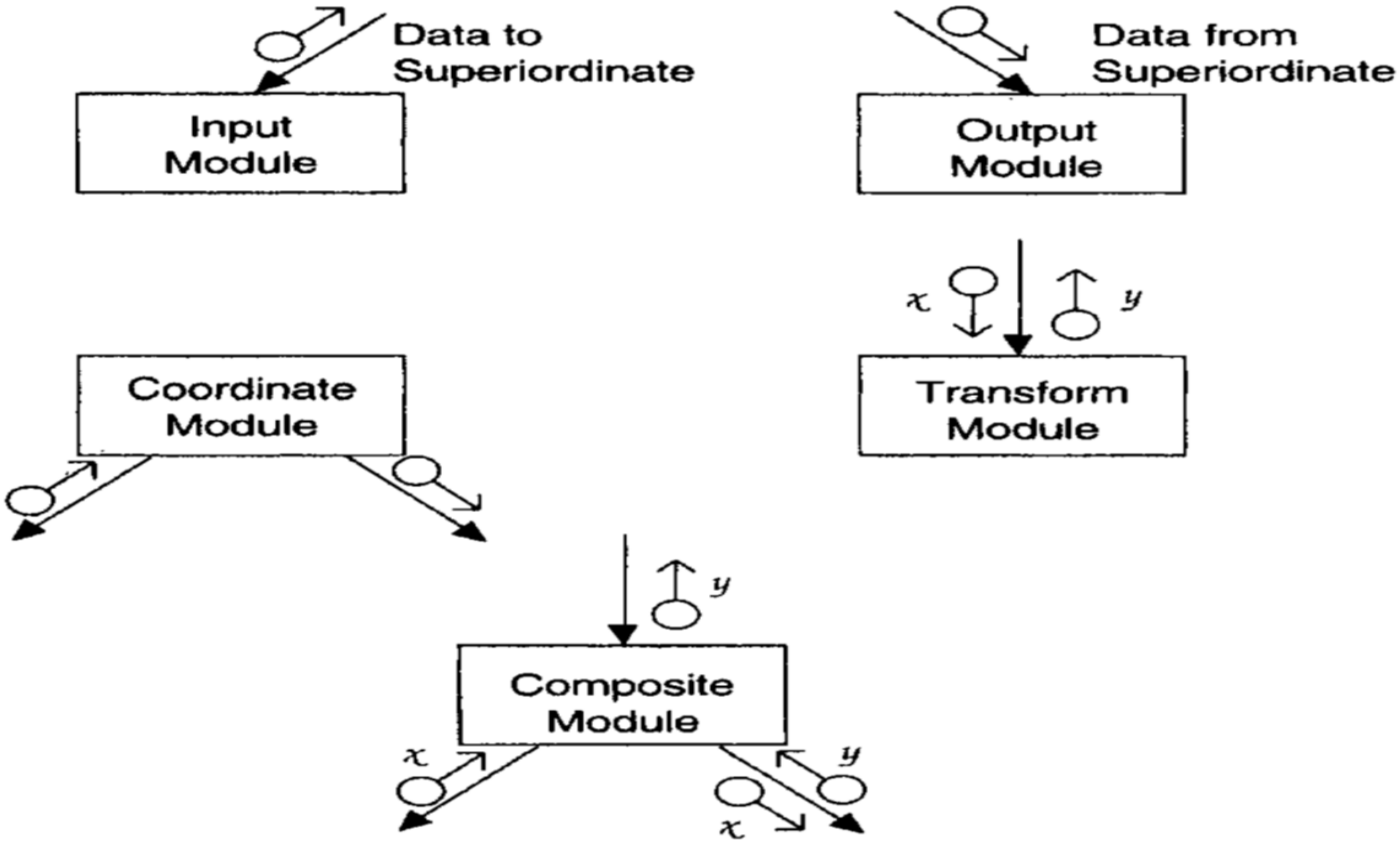


Loops and Decisions



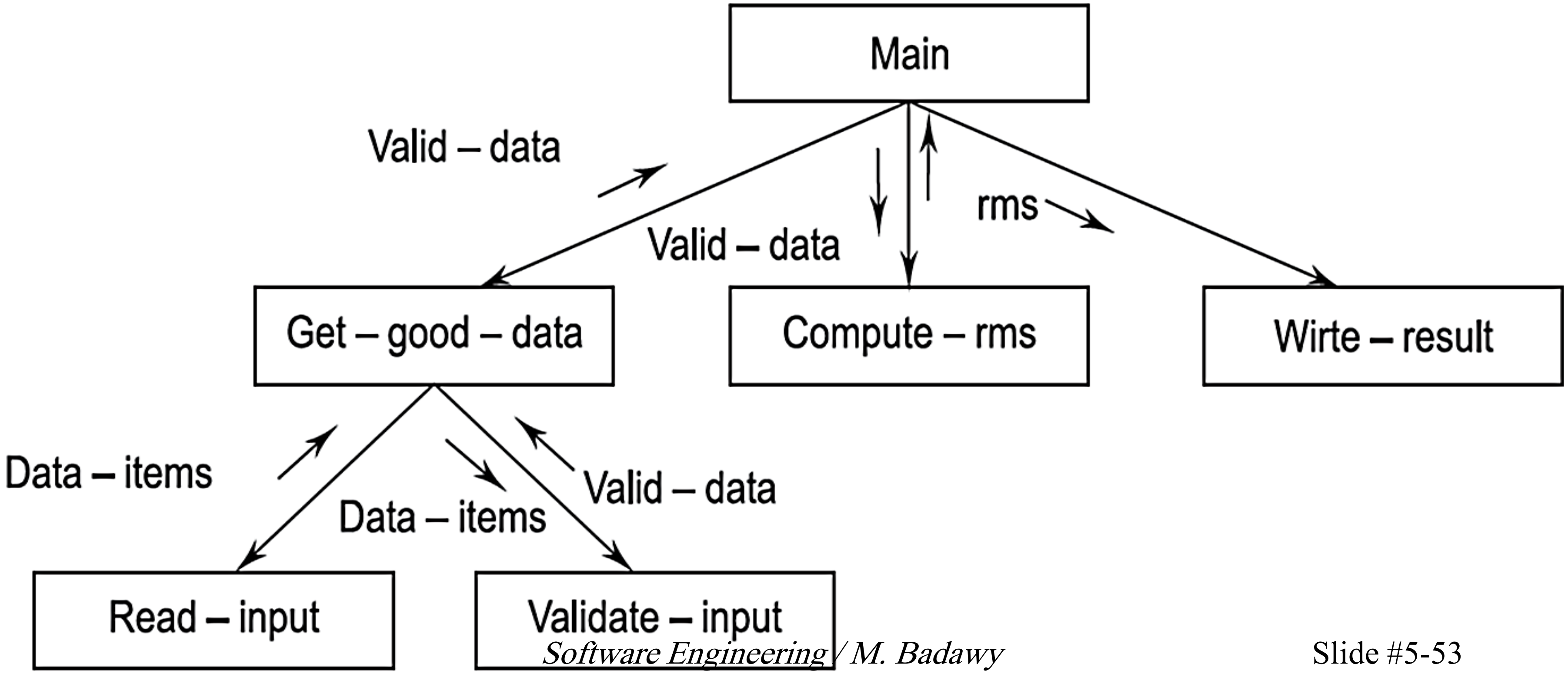


Different Types of Modules



Structure Chart Example

- A software system called RMS calculating software reads three integral numbers from the user in the range between –1000 and +1000 and determines the root mean square (rms) of the three input numbers and then displays it



Structure Chart

- Structure charts <u>are not sufficient for representing</u> the final design, as it <u>does not give all the information</u> needed about the design
- Design is generally supplemented with textual specifications to help the implementer

Specification

- Structure charts are inadequate when the design is to be communicated
- Module specification is the major part of system design specification
- All modules in the system should be identified when the system design is complete, and these modules should be specified in the document
- The design document must specify:
  - The interface of the module
  - The abstract behavior of the module
  - All other modules used by the module being specified

Structured Design Methodology

- Structured design methodology (SDM) views every software system as having some inputs that are converted into the desired outputs by the software system
- It is primarily function-oriented and relies heavily on functional abstraction and functional decomposition
- The aim is to design a system so that programs implementing the design would have a hierarchical structure, with functionally cohesive modules and as few interconnections between modules as possible

Structured Design Methodology

- Factoring is the process of decomposing a module so that the bulk of its work is done by its subordinates.
- A system is said to be completely factored if all the actual processing is accomplished by bottom-level atomic modules and if non-atomic modules largely perform the jobs of control and coordination.
- SDM attempts to achieve a structure that is close to being completely factored. The overall strategy is to identify the input and output streams and the primary transformations that have to be performed to produce the output.

SDM Major Steps

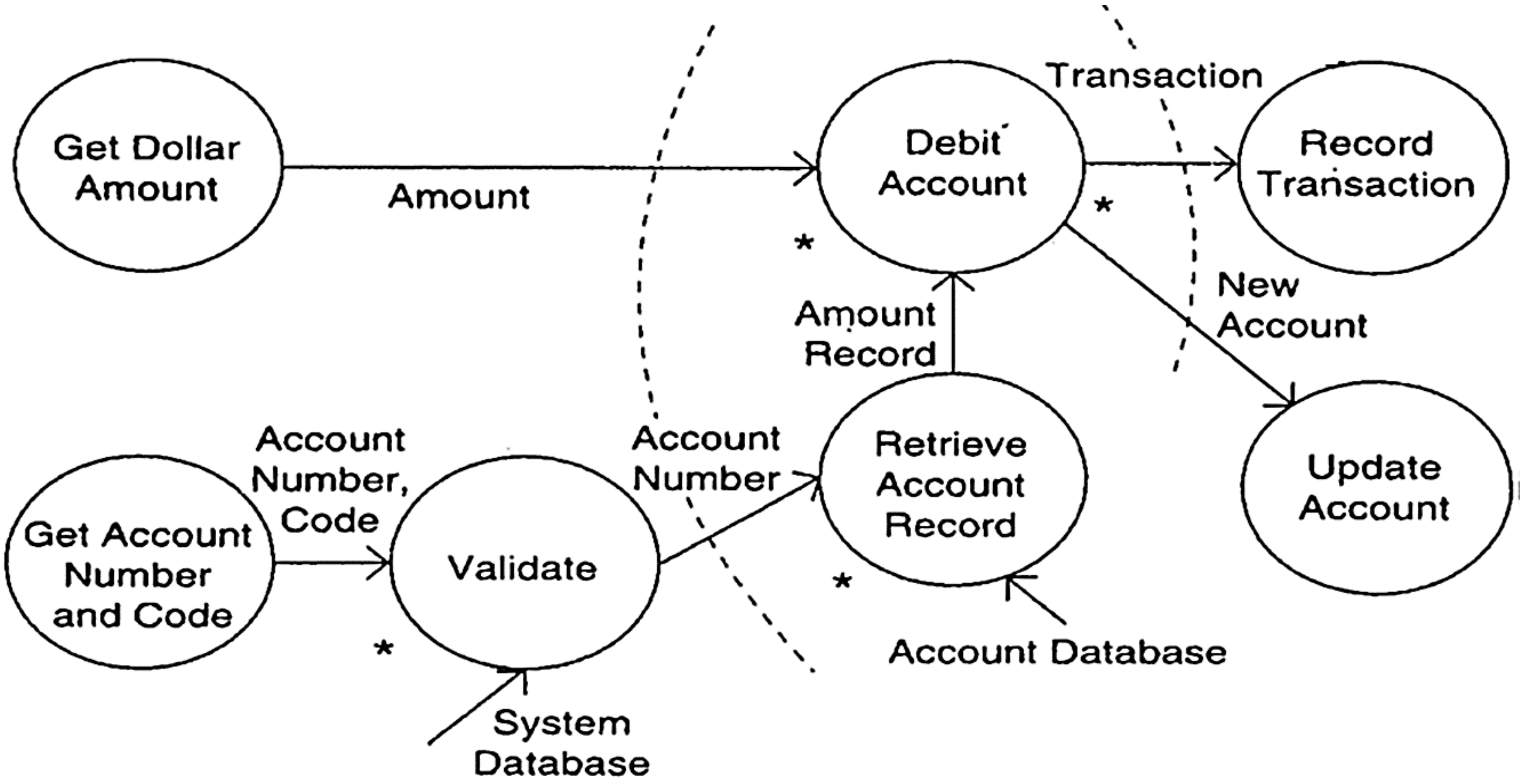
- There are four major steps in this strategy:
  1. Restate the problem as a data flow diagram
  2. Identify the input and output data elements
  3. First-level factoring
  4. Factoring of input, output, and transform branches
- For illustrating each step of the methodology as we discuss them, we consider the following problem: <u>There is a text file containing words separated by blanks or new lines</u>.

Restate as a Data Flow Diagram

- Drawing a DFD for design is a very creative activity in which the designer visualizes the eventual system and its processes and data flows
- As the system does not yet exist, the designer has complete freedom in creating a DFD that will solve the problem stated in the SRS

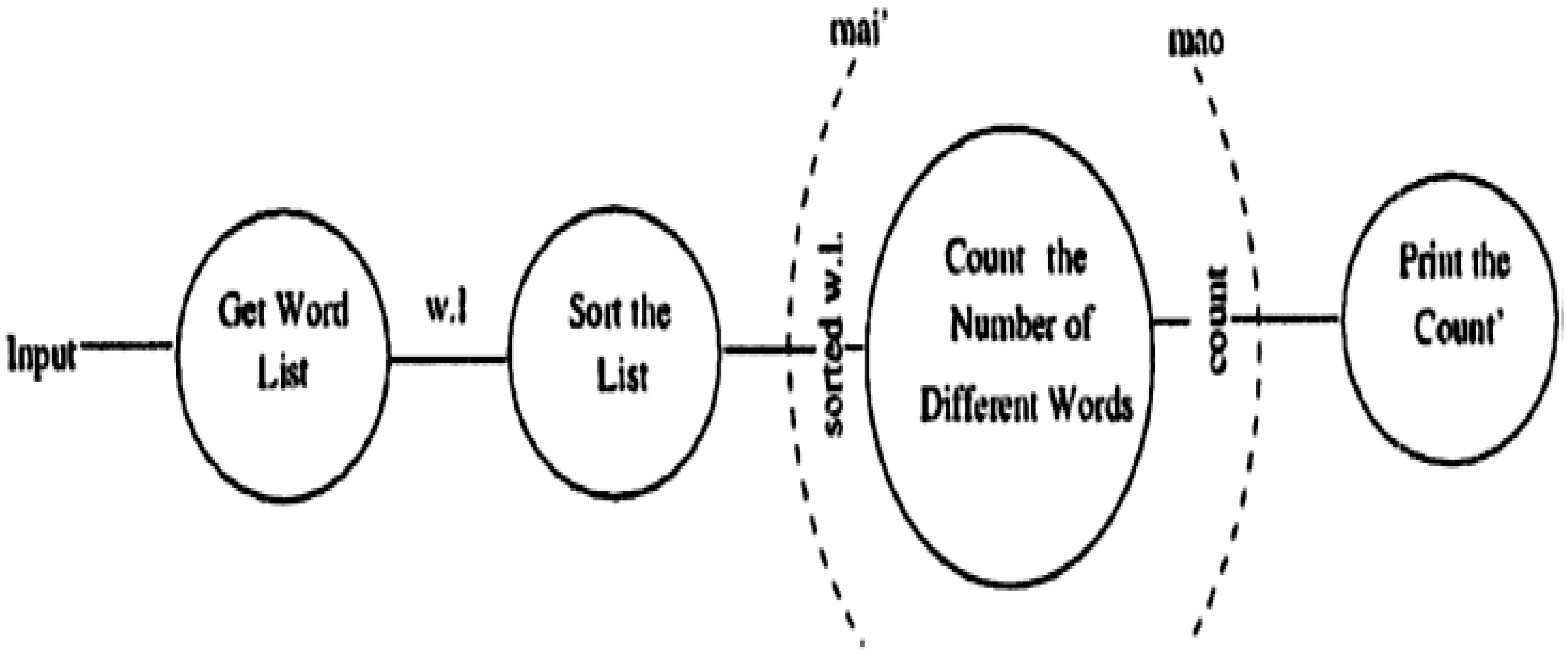
Restate as a Data Flow Diagram

A DFD for an ATM Machine



Restate as a Data Flow Diagram

Count Different Words in an Input File



Most abstract input and output data elements

- Most abstract input (MAI) data are those data elements that are farthest removed from the physical inputs but can still be considered inputs to the system
- MAIs obtained after operations like error checking, data validation, proper formatting, and conversion are complete

First Level Factoring

- Having identified the central transforms and the MAI and MAO items, we are ready to identify some modules for the system.

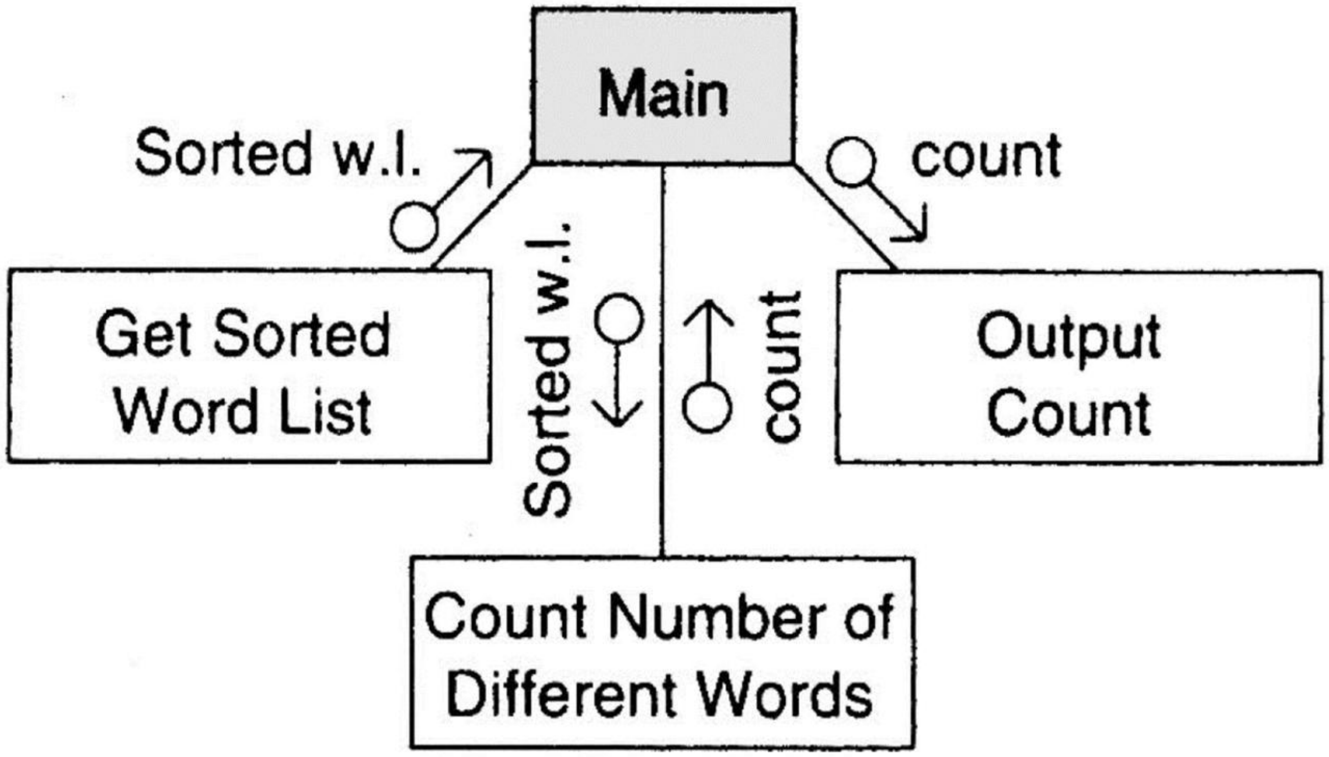
1. A main module, whose purpose is to invoke the subordinates. The main module is therefore a coordinate module.
2. For each of the most abstract input data items, an immediate subordinate module to the main module is specified.
3. Each of these modules is an input module, whose purpose is to deliver to the main module the most abstract data item for which it is created.
4. Similarly, for each most abstract output data item, a subordinate module that is an output module that accepts data from the main module is specified.

First Level Factoring

- Finally, for each central transform, a module subordinate to the main one is specified.
- These modules will be transform modules, whose purpose is to accept data from the main module, and then return the appropriate data back to the main module.
- The data items coming to a transform module from the main module are on the incoming arcs of the corresponding transform in the data flow diagram.
- The data items returned are on the outgoing arcs of that transform.

First Level Factoring

- The structure after the first-level factoring of the word-counting problem is shown in the next Figure

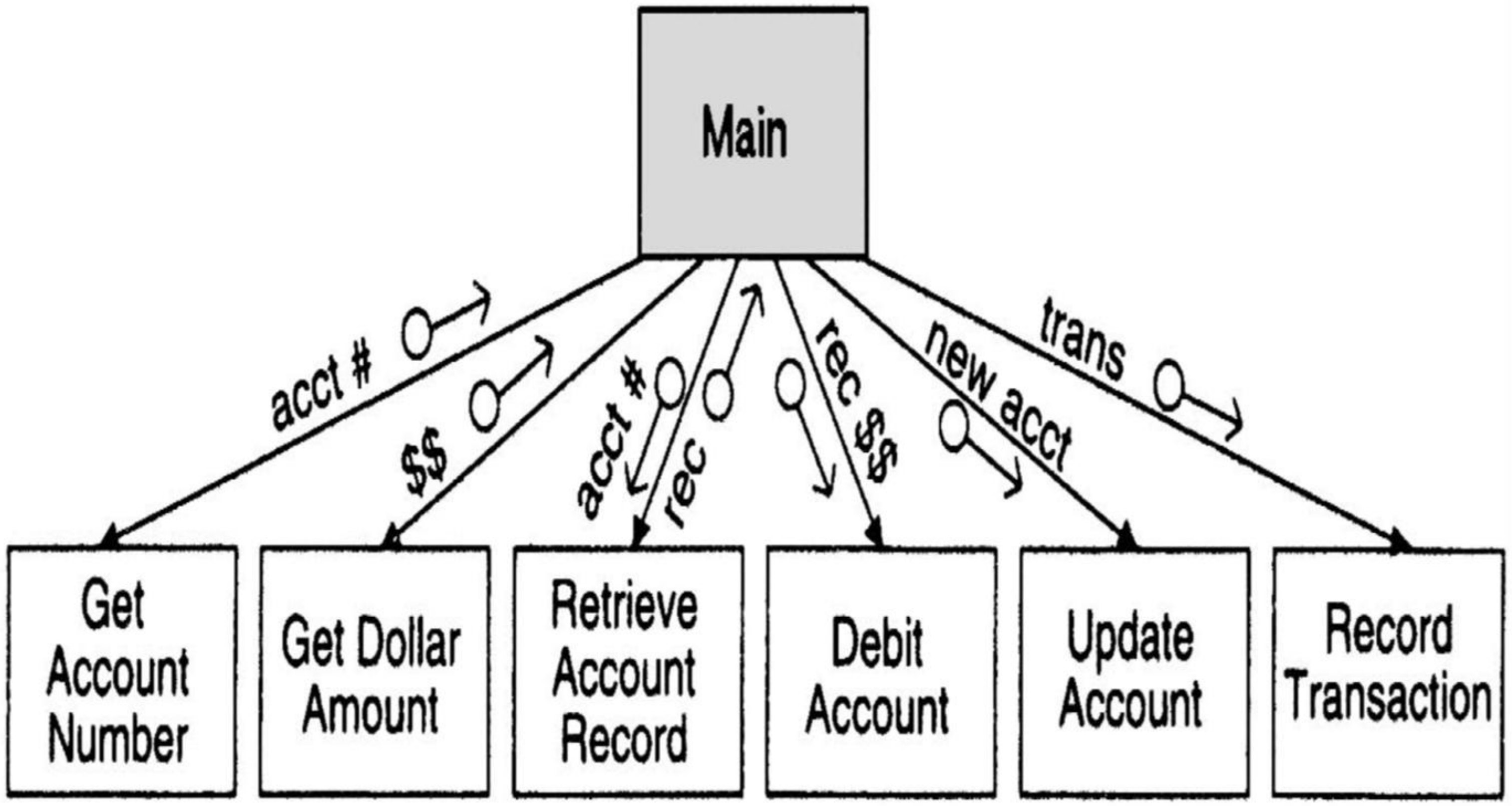


First Level Factoring

- In this example, there is one input module, which returns the sorted word list to the main module.
- The output module takes from the main module the value of the count. There is only one central transform in this example, and a module is drawn for that.
- The data items travelling to and from this transformation module are the same as the data items going in and out of the central transform.

First Level Factoring

- For the data flow diagram of the ATM.
- It has two most abstract inputs, two most abstract outputs, and two central transforms.



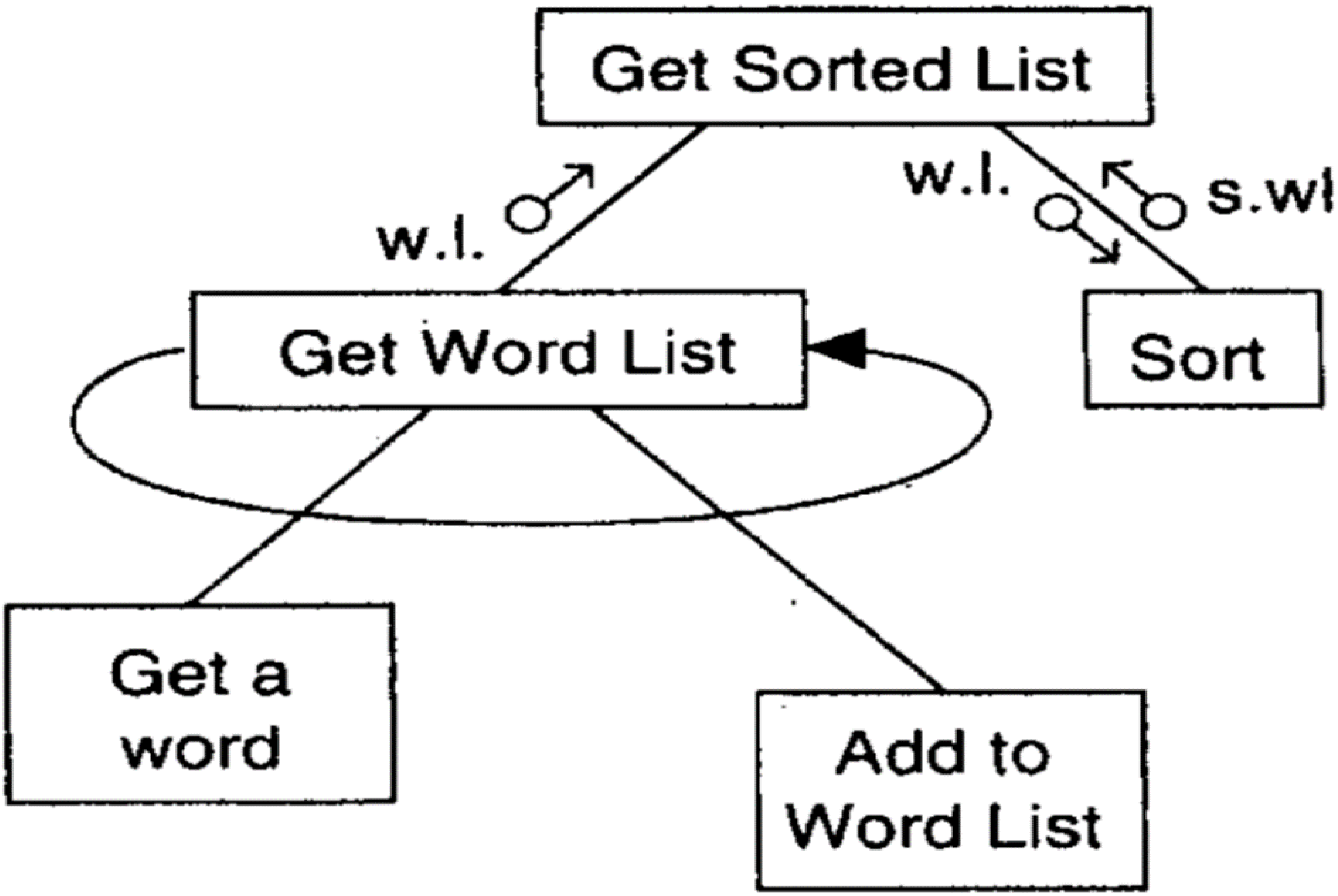
First Level Factoring

- The main module is the overall control module, which will form the main program or procedure in the implementation of the design.
- It is a coordinate module that invokes the input modules to get the most abstract data items, passes these to the appropriate transform modules, and delivers the results of the transform modules to other transform modules until the most abstract data items are obtained. These are then passed to the output modules.

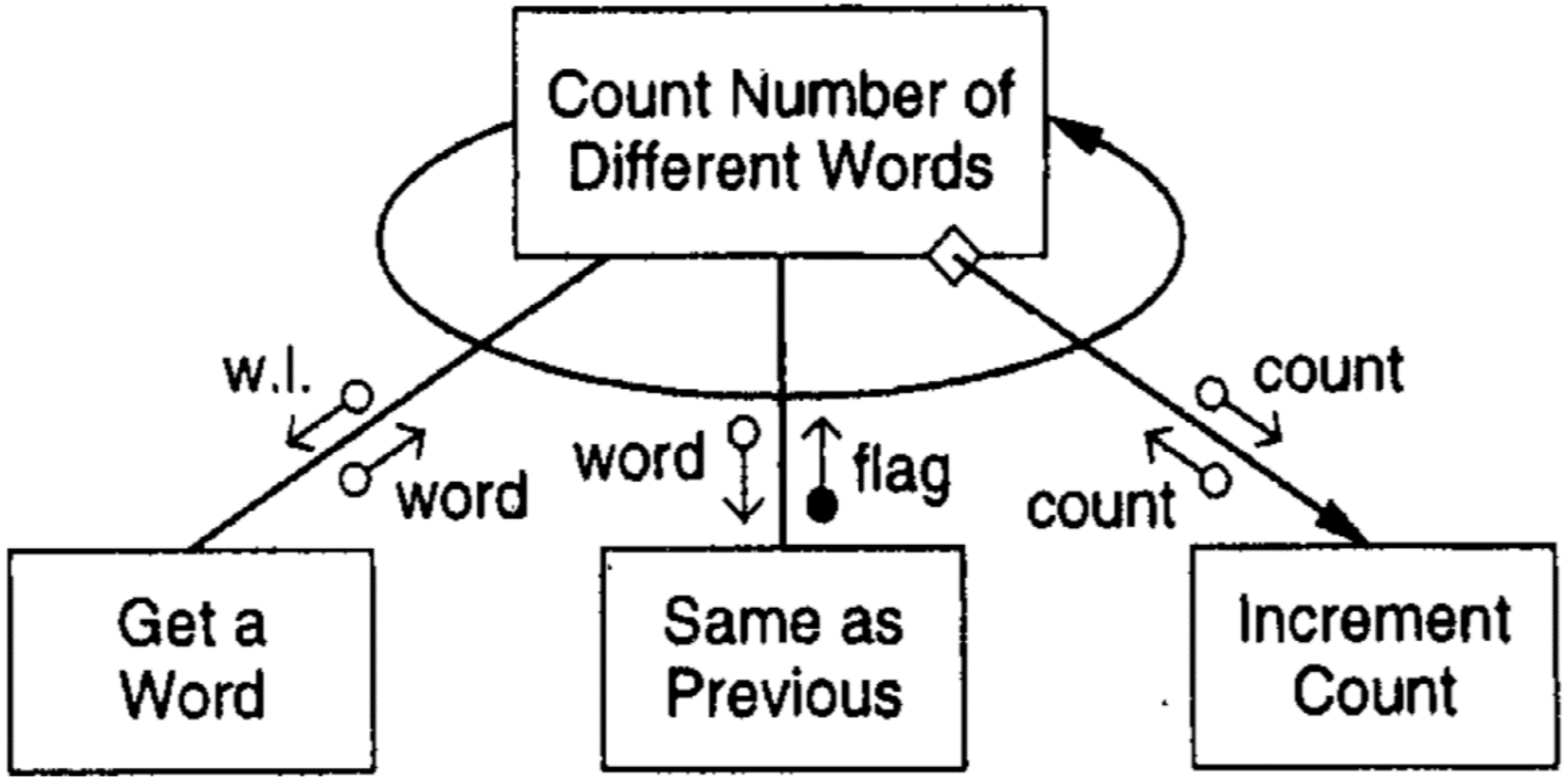
Factoring the Input, Output, and Transform Branches

- The first-level factoring results in a very high-level structure, where each subordinate module has a lot of processing to do
- To simplify these modules, <u>they must be factored into subordinate modules that will distribute the work of a module</u>

Factoring the input module



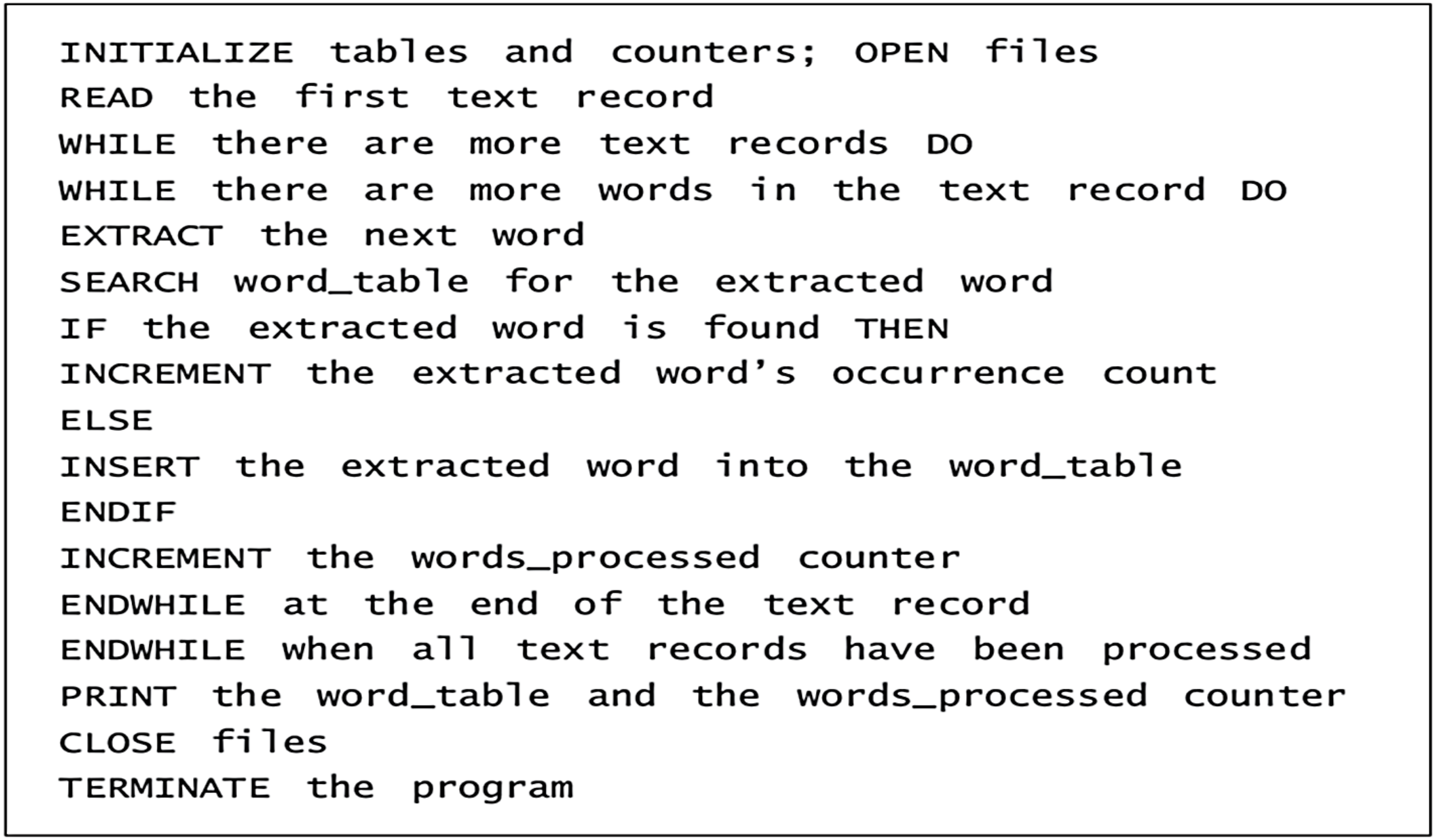
Factoring the central transform



Pseudo-Code

- Pseudo-code notation can be <u>used in both the preliminary and detailed design phases</u>
- Using pseudo-code, the designer <u>describes system characteristics using short, concise English language phrases that are structured by keywords, such as If-Then-Else, While-Do, and End</u>
- It is written using simple phrases and avoids cryptic symbols
- It is independent of high-level languages and is a very good means of expressing an algorithm
- It is written in a structured manner and indentation is used to increase clarity

Pseudo-Code Design Specification Example



```
INITIALIZE tables and counters; OPEN files
READ the first text record
WHILE there are more text records DO
WHILE there are more words in the text record DO
EXTRACT the next word
SEARCH word_table for the extracted word
IF the extracted word is found THEN
INCREMENT the extracted word's occurrence count
ELSE
INSERT the extracted word into the word_table
ENDIF
INCREMENT the words_processed counter
ENDWHILE at the end of the text record
ENDWHILE when all text records have been processed
PRINT the word_table and the words_processed counter
CLOSE files
TERMINATE the program
```

Advantages of Pseudo-Code

- Converting a pseudo-code to a programming language is much easier compared to converting a flowchart or decision table
- Compared to a flowchart, it is easier to modify the pseudo-code of program logic
- Writing of pseudo-code involves much less time and effort than the equivalent flowchart
- Pseudo-code is easier to write than writing a program in a programming language because pseudo-code as a method has only a few rules to follow

Disadvantages of Pseudo-Code

- There are no standard rules to follow in using pseudo-code
- Problems occur due to lack of standardization
- For a beginner, it is more difficult to follow the logic or write the pseudo-code as compared to flowcharting

Verification for Design

- If the design is expressed in some formal notation, then through tools it can be checked for internal consistency
- If the design is not specified in a formal, executable language, it cannot be processed through tools
- Other means for verification have to be used. The most common approach for verification is design review or inspections

Verification for Design

- The review group must include a member of
  - both the system design team and the detailed design team,
  - the author of the requirements document,
  - the author responsible for maintaining the design document,
  - an independent software quality engineer
- The most significant design error is <u>omission</u> or <u>misinterpretation</u> of specified requirements

A Sample design Checklist

- Is each of the functional requirements taken into account?
- Are there analysis to demonstrate that performance requirements can be met?
- Are there any constraints on the design beyond those in the requirements?
- Are external specifications of each module completely specified?
- Have exceptional conditions been handled?
- Are all the data formats consistent with the requirements?

Design Metrics

- Size is always a product metric of interest, as size is the single most influential factor deciding the cost of the project
- Another metric of interest is complexity
- Reducing the complexity will directly improve the testability and maintainability
- Complex modules are often more error-prone
- We will describe some of the metrics that have been proposed to quantify the complexity of design

Network Complexity Metrics

- Network metrics for design focus on the call graph component of the structure chart and define some metrics of how "good" the structure or network is in an effort to quantify the complexity of the call graph
- As coupling of a module increases if it is called by more modules, <u>a good structure is considered one that has exactly one caller</u>
- That is, the call graph structure is simplest if it is a pure tree

Graph Impurity = n (nodes) - e (edges) - 1

Stability Metrics

- Maintenance activity is hard and error-prone as changes in one module require changes in other modules to maintain consistency, which require further changes, and so on
- It is desirable to minimize this ripple effect of performing a change
- Stability of a design is a metric that tries to quantify the resistance of a design to the potential ripple effects that are caused by changes in modules.
- The higher the stability of a program design, the better the maintainability of the program.

Information Flow Metrics

- If we want a metric that is better at quantifying coupling between modules
- The information flow metrics attempt to define the <u>complexity in terms of the total information flowing through a module</u>.
- In one of the earliest work on information flow metrics, the complexity of a module is considered as depending on the intra module complexity and the inter module complexity.
- The intra module complexity is approximated by the size of the module in lines of code

Information Flow Metrics

- The inter module complexity of a module depends on the total information flowing in the module (inflow) and the total information flowing out of the module (outflow).
- The module design complexity, Dc, is defined as

$$Dc = size * (inflow * outflow)^2$$

- The term (inflow * outflow) refers to the total number of combinations of input source and output destination.
- This is based on the common experience that the <u>modules with more interconnections are harder to test or modify compared to other similar-size modules</u> with fewer interconnections.

Information Flow Metrics

- A variant of this was proposed based on the hypothesis that the module complexity depends not only on the information flowing in and out, but also on the number of modules to or from which it is flowing.
- The module size is considered an insignificant factor, and complexity Dc for a module is defined as:

  Dc = fan_in* fan_out + inflow * outflow

- Where fan in represents the number of modules that call this module and fan_out is the number of modules this module calls.