5.5 ACHIEVING QUALITY ATTRIBUTES

- Architectural styles provide general beneficial properties. To support specific quality attribute tactics are utilized:
  - Modifiability
  - Performance
  - Security
  - Reliability
  - Robustness
  - Usability
  - Business goals

ACHIEVING QUALITY ATTRIBUTES
SELF-MANAGING SOFTWARE

- In response to increasing demands that systems be able to operate optimally in different and sometimes changing environments, the software community is starting to experiment with self-managing software
  - Also referred to as autonomic, adaptive, dynamic, selfconfiguring, self-optimizing, self-healing, context-aware
- The essential idea is the same: the software system monitors its environment or its own performance, and changes its behavior in response to changes that it

ACHIEVING QUALITY ATTRIBUTES
SELF-MANAGING SOFTWARE (CONTINUED)

- Some examples of sensor changes:
  - Change the input sensors used, such as avoiding vision-based sensors when sensing in the dark
  - Change the Web servers that are queried, based on the results and performance of past queries
  - Move running components to different processors to balance processor load or to recover from a processor failure
- Obstacles to building self-managing software:
  - Few architectural styles
  - Monitoring nonfunctional requirements
  - Decision making

5.6 COLLABORATIVE DESIGN

- Usually the design of software systems is performed by a team of developers
- Several issues must be addressed by the team:
  - Who is best suited to design each aspect of the system
  - How to document all aspects
  - How to coordinate and integrate the software units
- Important to view group interaction in its cultural and ethical contexts
  - Fixed size of a dialogue box for different languages?
  - Colour scheme for an internationally usable design?

5.6 COLLABORATIVE DESIGN
OUTSOURCING

- Coordination becomes increasingly difficult
- Collaborative team may be distributed around the world
- Four stages in distributed development:
  - Project performed at single site with on-site developers from foreign countries
  - On-site analysts determine system requirements, which are in turn provided to off-site groups
  - Off-site developers build generic products and components that are used worldwide
  - Off-site developers build products that take advantage of their individual areas of expertise

5.7 ARCHITECTURE EVALUATION AND REFINEMENT

- Design is iterative: we propose design decisions, assess, make adjustments, and propose more decisions
- Many techniques to evaluate the design:
  - Trade-off analysis
  - Cost-benefit analysis
  - Prototyping

5.7 ARCHITECTURE EVALUATION AND REFINEMENT
TRADE-OFF ANALYSIS

- Often several alternative designs to consider
  - professional duty to explore design alternatives and not simply implement the first design that comes to mind
  - different members of design team may promote competing designs
  - need a measurement-based method for comparing design alternatives

ARCHITECTURE EVALUATION AND REFINEMENT
ONE SPECIFICATION, MANY DESIGNS

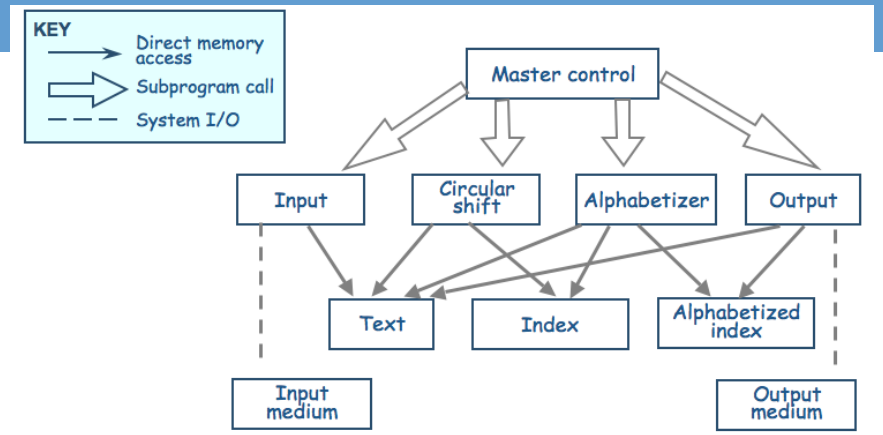
- **One specification, many designs**: to see how different designs can be used to solve the same problem

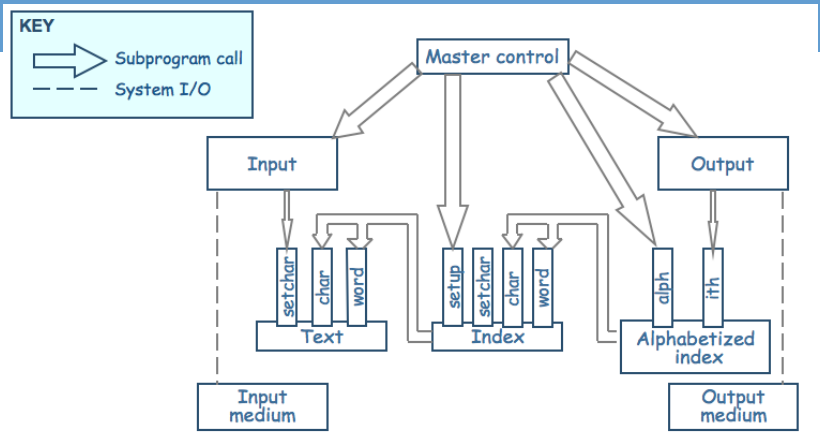
> The [key word in context] KWIC system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a list of all circular shifts of all lines in alphabetical order.

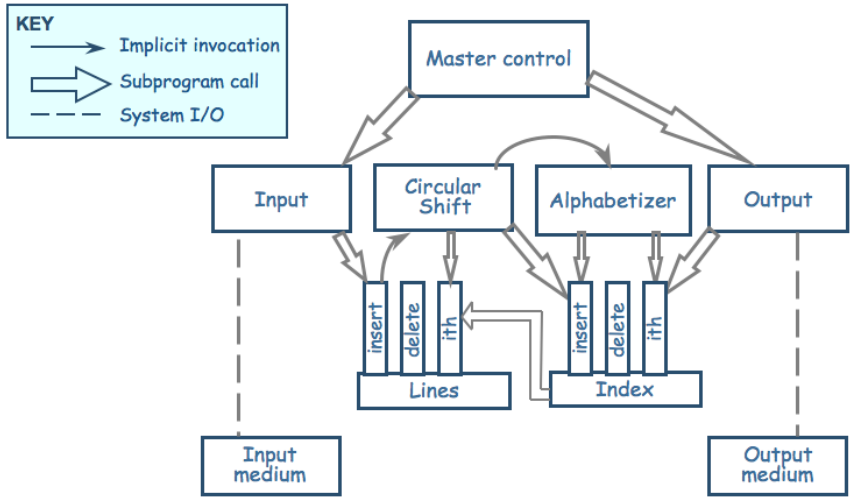
5.7 ARCHITECTURE EVALUATION AND REFINEMENT
ONE SPECIFICATION, MANY DESIGNS

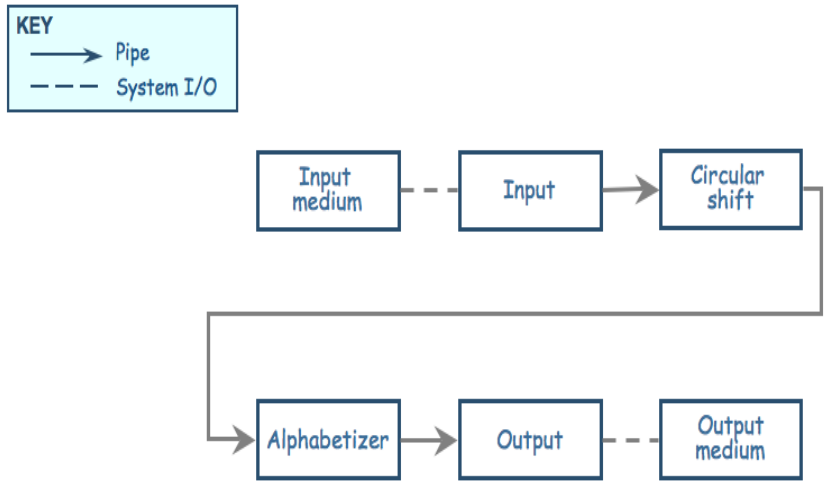
- **One specification, many designs**: to see how different designs can be used to solve the same problem
- Shaw and Garlan present four different architectural designs to implement KWIC (Key Word in Context problem)
  - shared data
  - abstract data type
  - implicit invocation
  - pipe and filter

5.7 ARCHITECTURE EVALUATION AND REFINEMENT
ONE SPECIFICATION, MANY DESIGNS (CONTINUED)









ARCHITECTURE EVALUATION AND REFINEMENT
ONE SPECIFICATION, MANY DESIGNS (CONTINUED)

- Comparison of KWIC solutions on the basis of important attributes

Attribute	Shared Data	Data Abstraction	Implicit Invocation	Pipe and Filter
Easy to change Algorithm	-	-	+	+
Easy to Change Data	-	+	-	-
Easy to Add Functionality	+	-	+	+
Performance	-	-	+	+
Efficient Data Rep	+	+	+	-
Easy to Reuse	-	+	-	+

+ means that the design has the attribute
- means that the attribute is not an aspect of the design

ARCHITECTURE EVALUATION AND REFINEMENT
ONE SPECIFICATION, MANY DESIGNS (CONTINUED)

- Weighted comparison of KWIC solutions

Highest Priority of an attribute = 5, Lowest priority = 1.

For example, Reusability is the most desirable attribute here

Numbers in column 3 to 6 represent the extent to which a design satisfies the characteristic of the corresponding attribute.

1 being the lowest, 5 being the highest

Attribute	Priority	Shared data	Abstract data type	Implicit invocation	Pipe and filter
Easy to change algorithm	1	1	2	4	5
Easy to change data representation	4	1	5	2	1
Easy to change function	3	4	1	4	5
Good performance	3	5	4	2	2
Easy to reuse	5	1	4	2	5

ARCHITECTURE EVALUATION AND REFINEMENT
ONE SPECIFICATION, MANY DESIGNS (CONTINUED)

- Weighted comparison of KWIC solutions

Multiply the priorities with ratings and sum over the design to get score for each design

For example score for pipe and filter = 1x5 + 4x1 + 3x5 + 3x2 + 5x5 = 55.

Pick the design with the highest score. Here Abstract data type has the highest score (57)

Attribute	Priority	Shared data	Abstract data type	Implicit invocation	Pipe and filter
Easy to change algorithm	1	1	2	4	5
Easy to change data representation	4	1	5	2	1
Easy to change function	3	4	1	4	5
Good performance	3	5	4	2	2
Easy to reuse	5	1	4	2	5

5.7 ARCHITECTURE EVALUATION AND REFINEMENT
ONE SPECIFICATION, MANY DESIGNS (CONTINUED)

- Other attributes to consider
  - Modularity
  - Testability
  - Security
  - Ease of use
  - Ease of understanding
  - Ease of integration

LECTURE

DESIGN PRINCIPLES

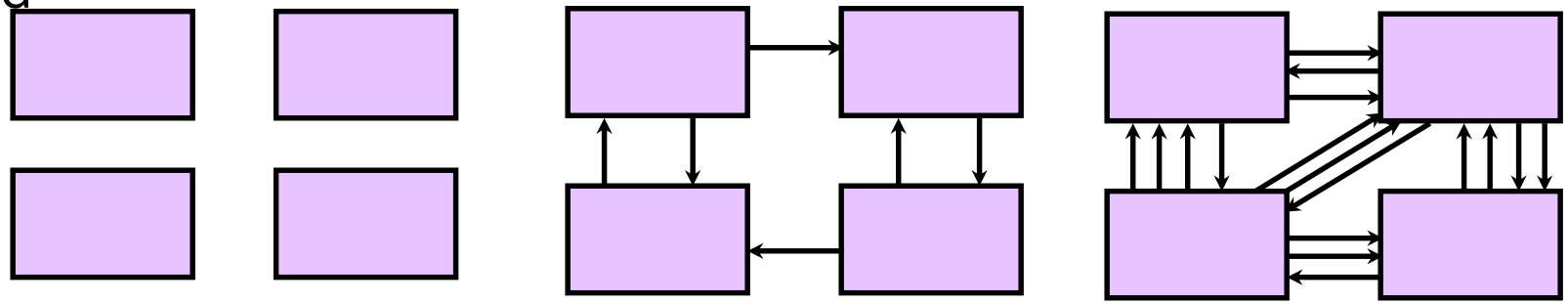
- **Design principles** are guidelines for decomposing a system's required functionality and behavior into modules
- The principles identify the criteria
  - for decomposing a system
  - deciding what information to provide (and what to conceal) in the resulting modules
- Six dominant principles:
  - Modularity
  - Interfaces
  - Information hiding
  - Incremental development
  - Abstraction
  - Generality

MODULARITY

- **Modularity** is the principle of keeping the unrelated aspects of a system separate from each other,
  - each aspect can be studied in isolation (also called separation of concerns)
- If the principle is applied well, each resulting module will have a single purpose and will be relatively independent of the others
  - each module will be easy to understand and develop
  - easier to locate faults
    - because there are fewer suspect modules per fault
  - Easier to change the system
    - because a change to one module affects relatively few other modules
- To determine how well a design separates concerns, we use two concepts that measure module independence: coupling and cohesion

MODULARITY: COUPLING

- Two modules are **tightly coupled** when they depend a great deal on each other
- **Loosely coupled** modules have some dependence, but their interconnections are weak
- **Uncoupled** modules have no interconnections at all; they are completely unrelated



Uncoupled -
no dependencies

Loosely coupled -
some dependencies

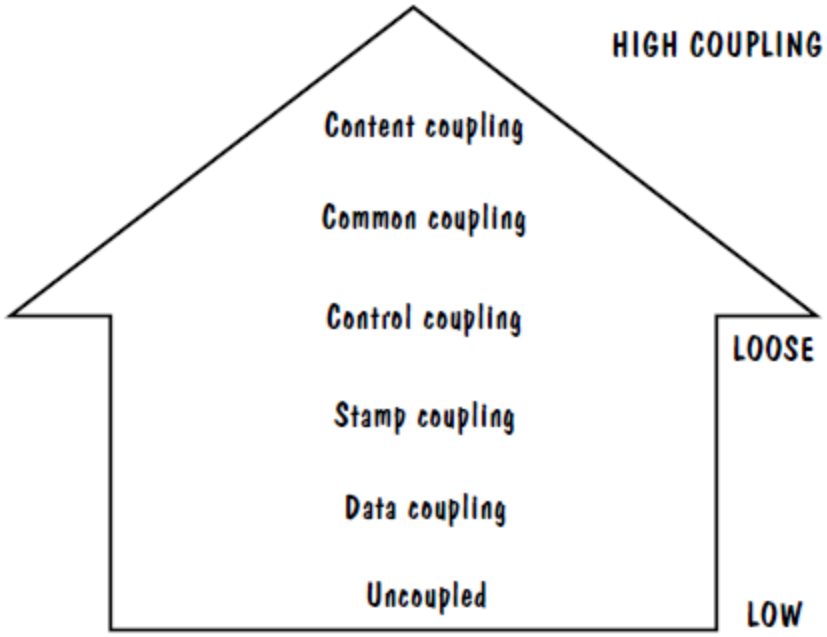
Tightly coupled -
many dependencies

MODULARITY: COUPLING

- There are many ways that modules can depend on each other:
  - The references made from one module to another
  - The amount of data passed from one module to another
  - The amount of control that one module has over the other
- Coupling can be measured along a spectrum of dependence

MODULARITY: COUPLING: TYPES OF COUPLING

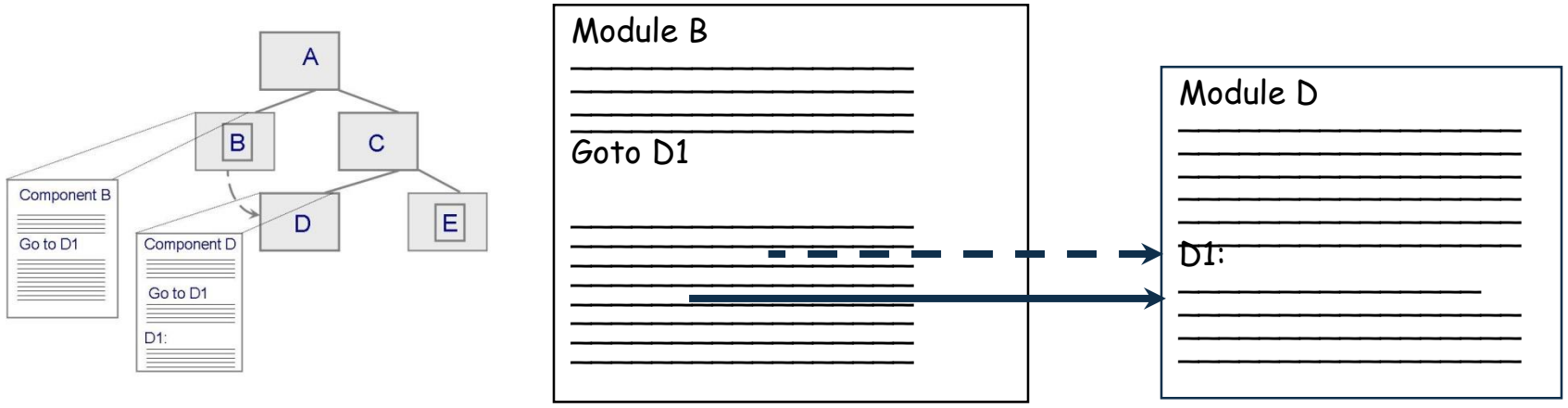
- Content coupling
- Common coupling
- Control coupling
- Stamp coupling
- Data coupling



High coupling is not desired

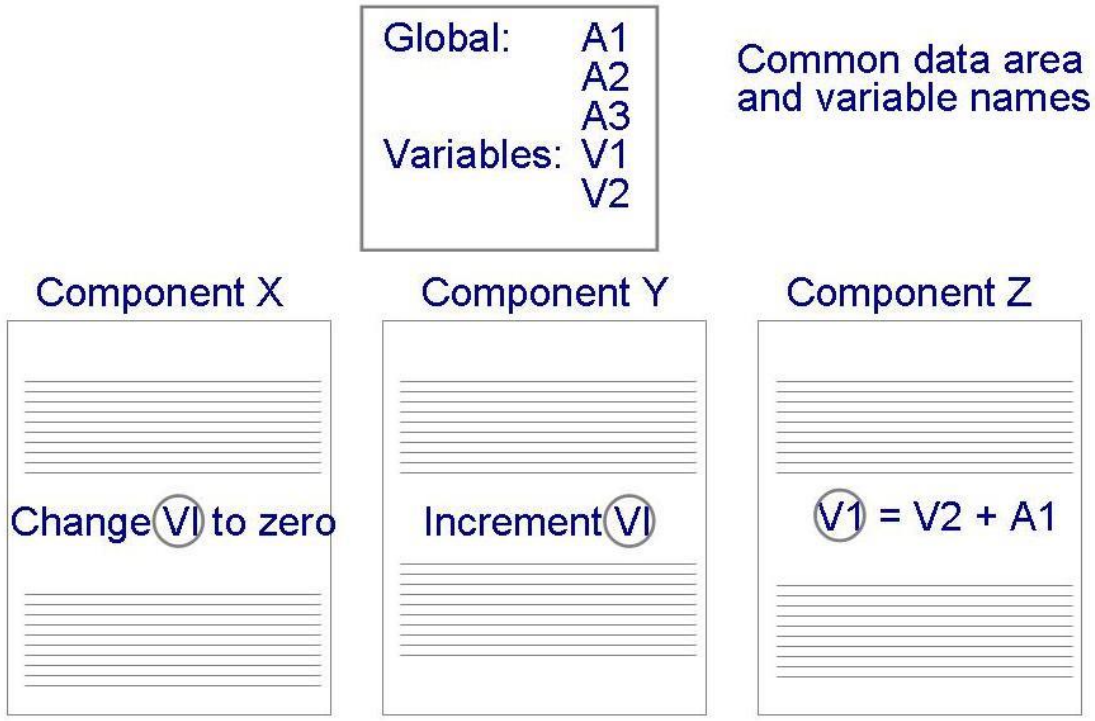
MODULARITY: COUPLING: CONTENT COUPLING

- One component modifies an internal data item of another component, a component (e.g. B) branches into another component (e.g. D)



MODULARITY: COUPLING: COMMON COUPLING

- Dependency due to common data. Which component is responsible to set a particular value for the common variable?
  - Tracing back to all components that access those data to evaluate the effect of the change



MODULARITY: COUPLING: CONTROL COUPLING

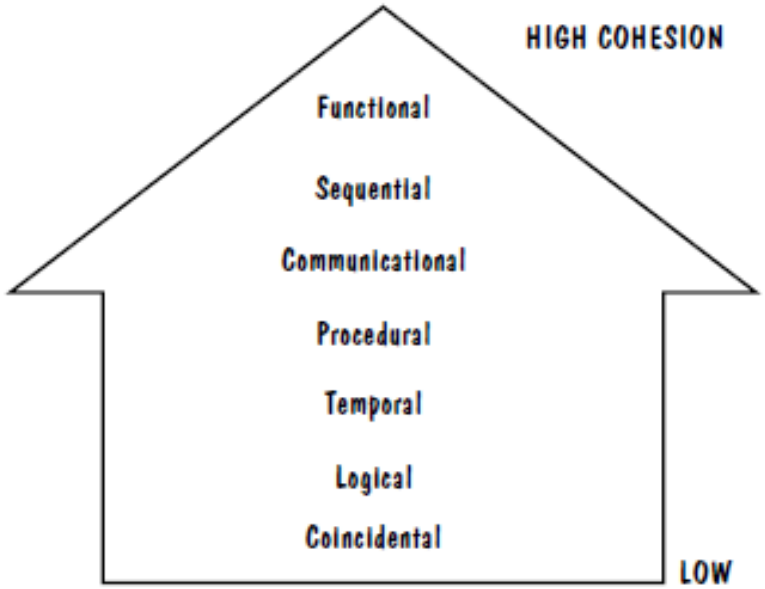
- One component passes parameters to control the activity of another component
- The controlled component cannot function without direction from the controlling component
- Each component performs only one function (Advantage)

MODULARITY: COUPLING: STAMP AND DATA COUPLING

- **Stamp coupling** occurs when complex data structures are passed between modules
  - Stamp coupling represents a more complex interface between modules, because the modules have to agree on the data's format and organization
- If only data values, and not structured data, are passed, then the modules are connected by **data coupling**
  - Data coupling is simpler and less likely to be affected by changes in data representation
  - Only data value is passed from one component to another
  - Easiest to trace data and make changes

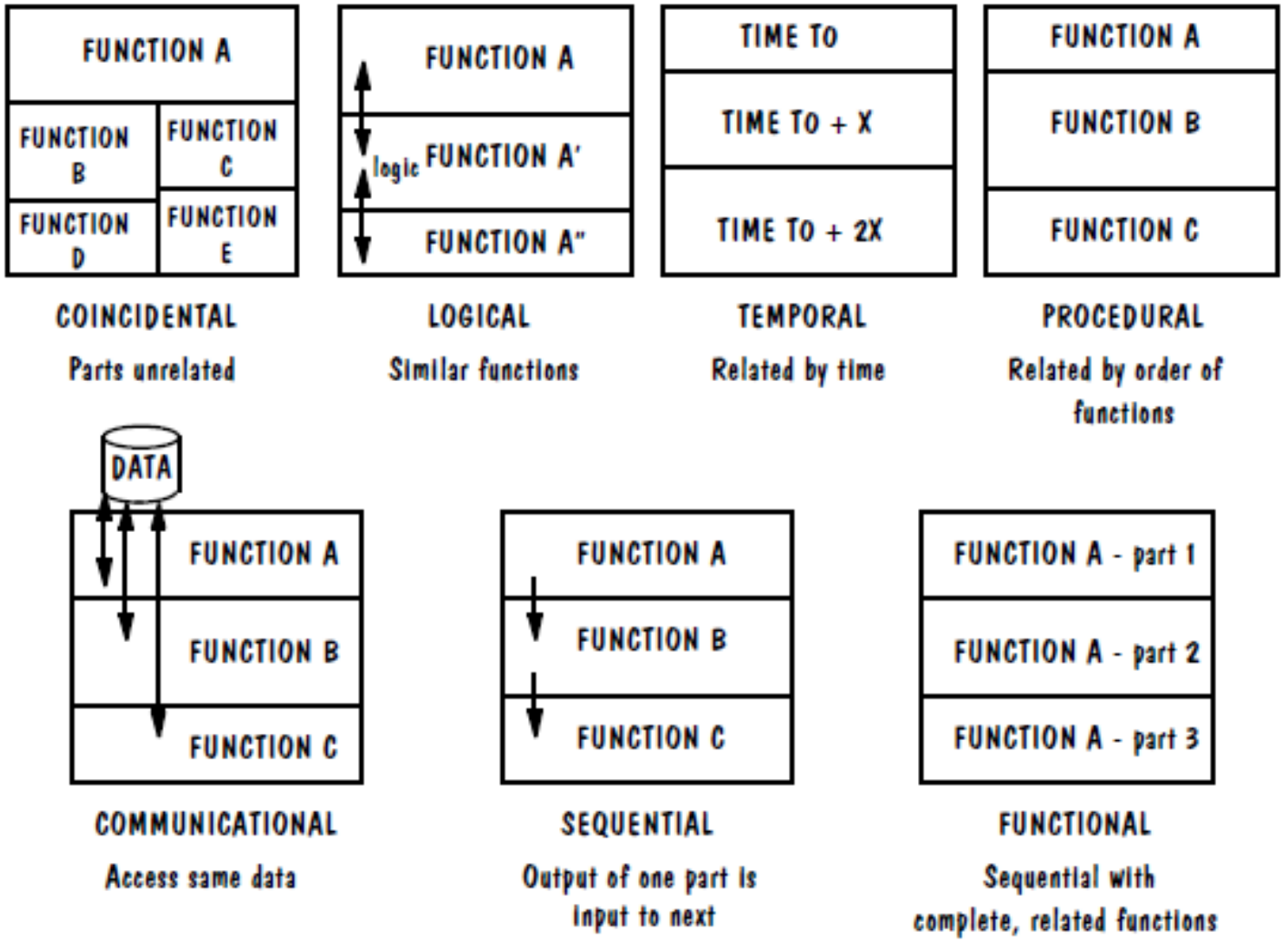
MODULARITY: COHESION: TYPES OF COHESION

- **Cohesion** refers to the dependence within and among a module's internal elements (e.g., data, functions, internal modules)



Low cohesion is not desired

MODULARITY: COHESION: TYPES OF COHESION



LECTURE

5.7 ARCHITECTURE EVALUATION AND REFINEMENT
COST-BENEFIT ANALYSIS

- Consider a proposal to improve KWIC performance because the number of KWIC indices have increased
  - Eliminate noise word indices?
  - Change representation of indices to bin of indices?
  - Increase server capacity?
- A cost–benefit analysis is a widely used business tool for estimating and comparing the costs and benefits of a proposed change

5.7 ARCHITECTURE EVALUATION AND REFINEMENT
COST-BENEFIT ANALYSIS

- A cost-benefit analysis contrasts financial benefits with financial costs, both in Dollars
  - Costs
    - Development
    - Operational
  - Benefits
    - Reduced Operational Costs
    - Increased Earnings
- Payback period
  - the length of time before accumulative benefits recover the costs of implementation

COST BENEFIT ANALYSIS OF DESIGN PROTOCOLS

TABLE 5.4 Cost–Benefit Analysis of Design Proposals

	Eliminate Noise Words	Store Indices in Bins	Add Second Server
Benefits			
Search time	0.015 sec	0.002 sec	0.008 sec
Throughput	72 requests/sec	500 requests/sec	115 requests/sec
Added value	$24,000/yr	$280,000/yr	$110,000/yr
Costs			
Hardware			$5,000
Software	$50,000	$300,000	$200,000
Business losses	$28,000+/yr		
Total costs first year	$78,000	$300,000	$205,000

COST BENEFIT ANALYSIS OF DESIGN PROTOCOLS

For simplicity, suppose that every *additional* request per second that the system can process, up to 200 requests/second, would save the company $2000 per year, based on retained customers and reduced calls to technical support. Given this value function, eliminating noise words would save the company $24,000 per year, calculated as

$$(72 \text{ requests/second} - 60 \text{ requests/second}) \times \$2000/\text{year} = \$24{,}000/\text{year}$$

Adding a second server would save the company $110,000 per year, calculated as

$$(115 \text{ requests/second} - 60 \text{ requests/second}) \times \$2000/\text{year} = \$110{,}000/\text{year}$$

The second design option would improve the system's throughput beyond what will be needed (the system will receive at most 200 requests per second). Therefore, the value added by changing to bin-based indexing is the maximum possible value:

$$(200 \text{ requests/second} - 60 \text{ requests/second}) \times \$2000/\text{year} = \$280{,}000/\text{year}$$

If there are multiple attributes to consider (e.g., the time it takes to update, reindex, and re-sort the catalogue), then a design's total financial added value is the sum of the added values ascribed to each of the attributes—some of whose added values may be negative if a design improves one attribute at the expense of a conflicting attribute.

5.7 ARCHITECTURE EVALUATION AND REFINEMENT COST-BENEFIT ANALYSIS

- Return on Investment (ROI)
  - ROI = Benefits/Cost, ROI > 1 is desired
  - %age gain ROI = (Benefits – Cost) / Cost x 100
  - Example
    - After five years
      - Cost = $ 2249559
      - Benefits = $ 6122893
      - ROI = 2.72 i.e. Total benefits are approx. 3 times the total costs
      - %age gain ROI = 172.19 % i.e. Your earnings are 172.19 % of the total costs

5.7 ARCHITECTURE EVALUATION AND REFINEMENT COST-BENEFIT ANALYSIS

- Return on Investment (ROI) Table

	1 year	5 year	10 year
Eliminate Noise Words	24000/78000 = 0.3076	(24000*5)/{50000+ (28000*5)} = 0.6316	(24000*10)/{50000 +(28000*10)} = 0.7273
Store Indices in Bins	280000/300000 = 0.93	(280000*5)/ 300000 = 4.667	(280000*10)/ 300000 = 9.33
Add Second Server	110000/205000 = 0.53	(110000*5)/ 205000 = 2.683	(110000*10)/ 205000 = 5.366

5.7 ARCHITECTURE EVALUATION AND REFINEMENT COST-BENEFIT ANALYSIS

- % gain on ROI Table

	1 year	5 year	10 year
Eliminate Noise Words	{(24000-78000) /78000}*100 = -69.23		
Store Indices in Bins	{(280000-300000)/300000} * 100 = -6.667		
Add Second Server	{(110000-205000) /205000}*100 = -46.34		

5.7 ARCHITECTURE EVALUATION AND REFINEMENT
COST-BENEFIT ANALYSIS

Another useful measure is the **payback period**: the length of time before accumulated benefits recover the costs of implementation. In our example, the payback period for restructuring the sorted-index module (design 2) is

$$\$300{,}000/\$280{,}000 = 1.07 \text{ of a year} = \text{approximately 13 months}$$

5.7 ARCHITECTURE EVALUATION AND REFINEMENT
COST-BENEFIT ANALYSIS

- Payback Period Table

		Years
Eliminate Noise Words	78000/24000 =	3.25 {it is wrong because 28000 is also added annually}
Store Indices in Bins	300000/280000 =	1.07
Add Second Server	205000/110000 = 0.53	1.864

SPECIAL THANK FOR THE MATERIALS

- Mr. Haroon Abdul Waheed
- Prof. Zeeshan Ali Rana
- Pfleeger's Book slides from UCF
- Software Fundamentals