

The background of the slide is a light blue sky. In the top right corner is a bright yellow sun with a thick orange border. There are three white, fluffy clouds: one large one in the center-left, and two smaller ones to its right. The bottom of the slide features a green landscape with rolling hills. On the left hill is a single green tree with a brown trunk. On the right hill are two green trees with brown trunks. Small pink flowers are scattered across the green grass.

# DESIGN CONCEPTS AND PRINCIPLES

Compiled by  
Er. Shree Krishna Yadav

# overview

- What is it?

Design is what almost every engineer wants to do. it is the place where creativity rules-where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software but unlike the requirements model(that focus on describing required data, function and behavior)the design model provides details about software architecture, data structures, interfaces, and components that are necessary to implement the system.



- Who does it?

Software engineers conduct each of the design tasks.

- Why it is important?

Design allows you to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted and end users become involved in large numbers. Design is the place where software quality is established.



- What are the steps?

Design depicts the software in a number of different ways.

First the architecture of the system or product must be represented.

Then, the interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are modeled.

Finally, the software components that are used to construct the system are designed.

Each of these views represents a different design action, but all must conform to set of basic design concepts that guide software design work.



- What is the work product?

A design model that encompasses architectural, interface, component level and deployment representations is the primary work product that is produced during software design.

- How do I ensure that I have done it right?

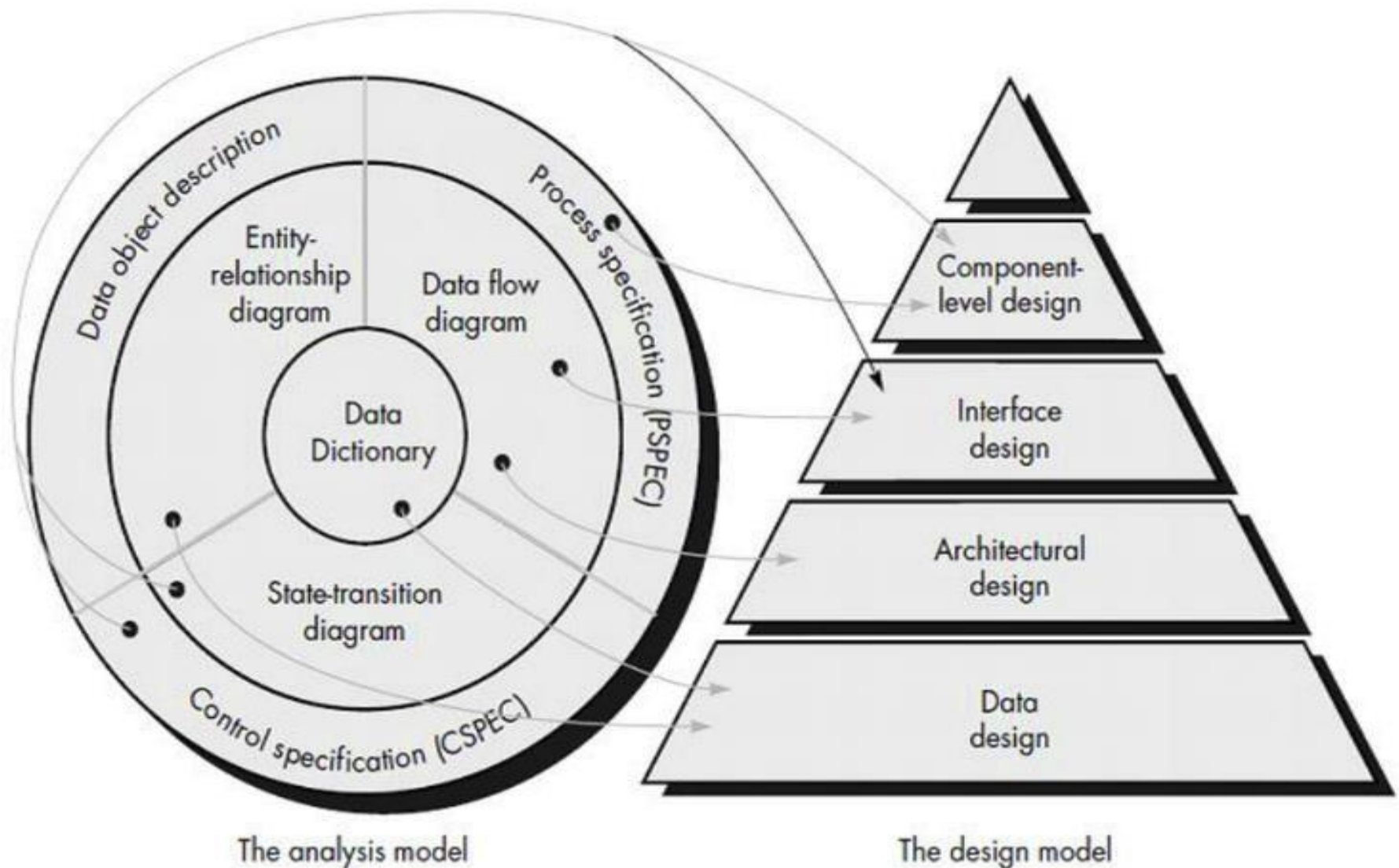
The design model is assessed by the software team in an effort to determine whether it contains errors, inconsistencies or omissions, whether better alternative exist and whether the model can be implemented within the constraints, schedule and cost that have been established.



- Once the requirements document for the software to be developed is available, the software design phase begins. While the requirement specification activity deals entirely with the problem domain, design is the first phase of transforming the problem into a solution. In the design phase, the customer and business requirements and technical considerations all come together to formulate a product or a system.
- Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Once software requirements have been analyzed and specified, software design is the first of three technical activities—design, code generation, and test—that are required to build and verify the software.
- Each of the elements of the analysis model provides information that is necessary to create the four design models required for a complete specification of design.
- Design is a meaningful engineering representation of something that is to be built. **It can be traced to a customer's requirements** and at the same time assessed for quality against a set of predefined criteria for “good” design. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces, and components.



# Translating the analysis model into a software design



- The **data design** transforms the information domain model created during analysis **into the datastructures** that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the data dictionary provide the basis for the data design activity. This is the foundation. *It transforms information from the Data Dictionary into a structured database.*

**Analysis Input:** Entity-Relationship Diagrams (ERDs), data object descriptions.

**Design Output:** Database schemas, data structures, necessary files/storage

- The **architectural design** defines the relationship between major structural elements of the software, the “design patterns” that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied. *This defines the overall framework of the software—how the main pieces fit together. It is derived largely from the Data Flow Diagrams.*

**Analysis Input:** Use Cases, Process Models (e.g., Activity Diagrams).

**Design Output:** High-level system structure, definition of major subsystems, selection of architectural styles (e.g., layered, microservices) and design patterns.

- The **interface design** describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design.



Component level design: The top of the pyramid. This is the most detailed part, where the specific logic for every single small part of the software is defined.

- The **component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC<sub>(process specification)</sub>, CSPEC<sub>(control specification)</sub>, and STD<sub>(state transition diagram)</sub> serve as the basis for component design.

**Analysis Input:** Functional models, Sequence Diagrams.

**Design Output:** Algorithms, Procedural logic for each component, class details (methods, attributes).

- During design we make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained.



# Why is design so important

- The importance of software design can be stated with a single word—quality.
- Design is the place where **quality is fostered** in software engineering.
- Design **provides us with representations of software** that can be assessed for quality.
- Design is the only way that we can **accurately translate a customer's requirements** into a finished software product or system.
- Software design **serves as the foundation** for all the software engineering and software support steps that follow.
- Without design, we risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.



# Basic of Software Design

- Software design is a phase in software engineering, in which a blueprint is developed to serve as a base for constructing the software system. **IEEE defines software design as 'both a process of defining, the architecture, components, interfaces, and other characteristics of a system or component and the result of that process.'**
- In the design phase, many critical and strategic decisions are made to achieve the desired functionality and quality of the system. These decisions are taken into account to successfully develop the software and carry out its maintenance in a way that the quality of the end product is improved.



# Design process

- Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software.
- The design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.
- As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.



# Design and software quality

- Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs. Three characteristics that serve as a guide for the evaluation of a good design suggested by McGlaughlin:
- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for those who test and support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation view.

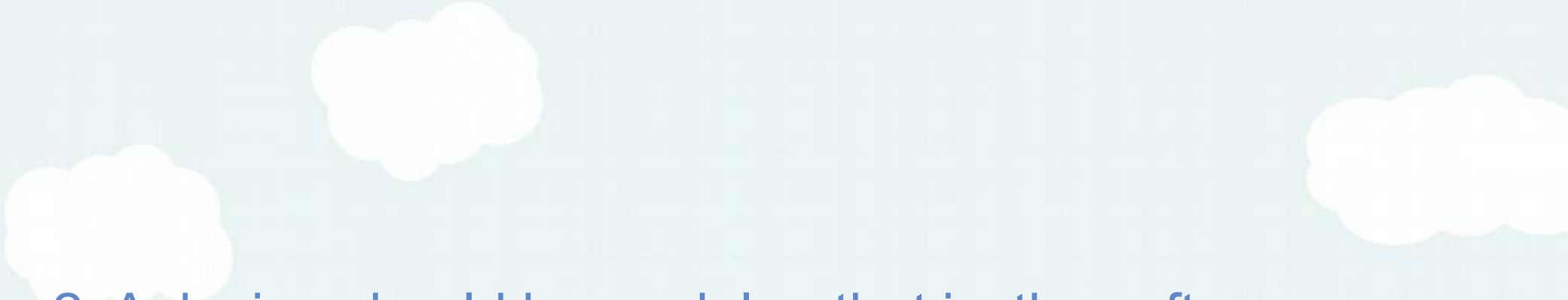


Each of the characteristics is actually a goal of the design process.


Following are the quality guidelines:

1. A design should exhibit an architectural structure that
  - (a) Has been created using recognizable design patterns,
  - (b) Is composed of components that exhibit good design characteristics.
  - (c) Can be implemented in an evolutionary fashion.



- 
2. A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and sub functions.
  3. A design should contain distinct representations of data, architecture, interfaces, and components (modules).
  4. A design should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns.



- 
5. A design should lead to components that exhibit independent functional characteristics.
  6. A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.
  7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

The software design process encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.



# Design principles

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built.

The design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer software. Creative skill, past experience, a sense of what makes “good” software and an overall commitment to quality are critical success factors for a competent design.

Davis suggests a set of principles for software design, which have been adapted and extended in the following list:



**The design process should not suffer from “tunnel vision.”** While designing the process, it should not suffer from “tunnel vision” which means that it should not only focus on completing or achieving the aim but on other effects also. A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job, and the design concepts.

**The design should be traceable to the analysis model.** The design process should be traceable to the analysis model which means it should satisfy all the requirements that software requires to develop a high-quality product. Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.

**The design should not reinvent the wheel.** The design process should not reinvent the wheel that means it should not waste time or effort in creating things that already exist. Due to this, the overall development will get increased. //Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.



## **The design should “minimize the intellectual distance”**

The design process should reduce the gap between real-world problems and software solutions for that problem meaning it should simply minimize intellectual distance.

## **The design should exhibit uniformity and integration.**

The design should display uniformity which means it should be uniform throughout the process without any change. Integration means it should mix or combine all parts of software i.e. subsystems into one system.

## **The design should be structured to accommodate change.**

The software should be designed in such a way that it accommodates the change implying that the software should adjust to the change that is required to be done as per the user's need.



**Design is not coding, coding is not design.** Design means describing the logic of the program to solve any problem and coding is a type of language that is used for the implementation of a design.

Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.

**The design should be assessed for quality as it is being created, not after the fact.**

A variety of design concepts and design measures are available to assist the designer in assessing quality.

**The design should be reviewed to minimize conceptual (semantic) errors.** The design should be reviewed which means that the overall evaluation should be done to check if there is any error present or if it can be minimized.

A design team should ensure that major conceptual elements of the design (omissions, ambiguity, and inconsistency) have been addressed before worrying about the syntax of the design model.

When these design principles are properly applied, the software engineer creates a design that exhibits both external and internal quality factors. External quality factors are those properties of the software that can be readily observed by users (e.g., speed, reliability, correctness, usability).<sup>2</sup> Internal quality factors are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts.



# DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the past four decades. Each provides the software designer with a foundation from which more sophisticated sign methods can be applied. Each helps the software engineer to answer the following questions:

- What criteria can be used to partition software into individual components?
  - How function or data is structure detail separated from a conceptual representation of the software?
  - What uniform criteria define the technical quality of a software design?
- Fundamental design Concepts provide the necessary framework for “getting Right”



# 1. ABSTRACTION

When we consider a modular solution to any problem, many levels of abstraction can be posed.

At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.

At lower levels of abstraction, a more procedural orientation is taken.

At the lowest level of abstraction, the solution is stated in a manner that can be directly Implemented. As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

A **procedural abstraction** is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

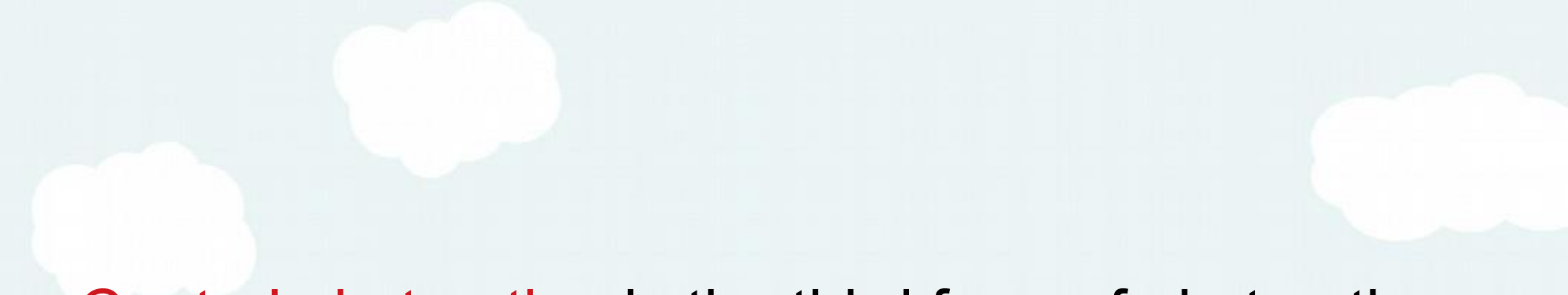


A **data abstraction** is a named collection of data that describes a data object. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door.

Many modern programming languages provide mechanisms for creating abstract data types. For example, the Ada package is a programming language mechanism that provides support for both data and procedural abstraction. The original abstract data type is used as a template or generic data structure from which other data structures can be instantiated.





**Control abstraction** is the third form of abstraction used in software design; control abstraction implies a program control mechanism without specifying internal details.

An example of a control abstraction is synchronization semaphore used to coordinate activities in an operating system.



# 2. Software Architecture

## 2. Architecture

- The architecture is the structure of program modules where they interact with each other in a specialized way.
- **Structural Properties:** Architectural design represent different types of components, modules, objects & relationship between these.
- **Extra-Functional Properties:** How design architecture achieve requirements of Performance, Capacity, Reliability, Security, Adaptability & other System Characteristics.
- **Families of related systems:** The architectural design should draw repeatable patterns. They have ability to reuse repeatable blocks.



# 3. REFINEMENT

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth . A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached. An overview of the concept is provided by Wirth: Refinement is actually a process of *elaboration*. Refinement causes the designer *to elaborate on the original statement*, providing more and more detail as each successive refinement occurs.

In each step (of the refinement), one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of any underlying computer or programming language . . . As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine the program and the data specifications in parallel.






Every refinement step implies some design decisions. It is important that . . . the programmer be aware of the underlying criteria (for design decisions) and of the existence of alternative solutions . . .

The process of program refinement proposed by Wirth is analogous to the process of refinement and partitioning that is used during requirements analysis. The difference is in the level of implementation detail that is considered, not the approach.

Refinement is actually a process of elaboration. We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.



# 4.MODULARITY

*“The degree to which software can be understood by examining its components independently of one another.”*

Software is divided into separately named and addressable components, often called *modules* that are integrated to satisfy problem requirements.

Modularity is the single attribute of software that allows a program to be intellectually manageable.

Monolithic software (i.e., **a large program composed of a single module**) cannot be easily grasped by a reader. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

To illustrate this point, consider the following argument based on observations of human problem solving.



Let  $C(x)$  be a function that defines the perceived complexity of a problem  $x$ , and  $E(x)$  be a function that defines the effort (in time) required to solve a problem  $x$ . For two problems,  $p1$  and  $p2$ , if

$$C(p1) > C(p2) \quad 1$$

it follows that

$$E(p1) > E(p2) \quad 2$$

As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

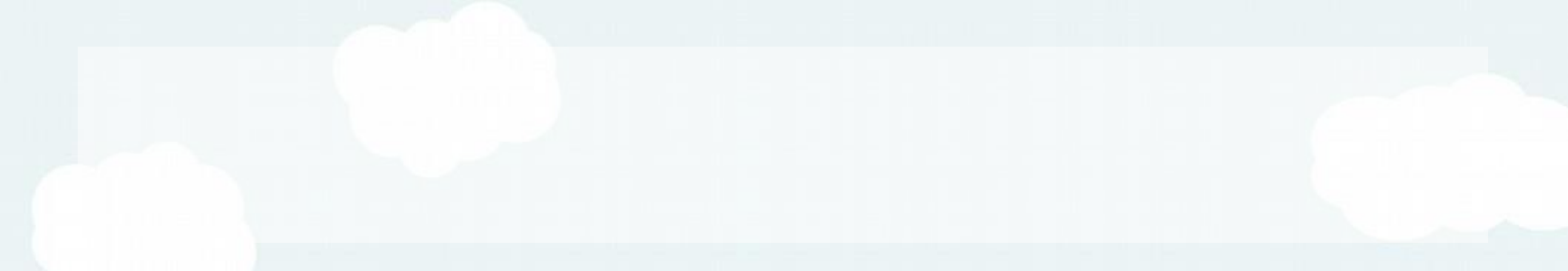
Another interesting characteristic has been uncovered through experimentation in human problem solving. That is,

$$C(p1 + p2) > C(p1) + C(p2) \quad 3$$

Expression 3 implies that the perceived complexity of a problem that combines  $p1$  and  $p2$  is greater than the perceived complexity when each problem is considered separately. Considering Expression 2 and the condition implied by Expressions 1, it follows that


$$E(p1 + p2) > E(p1) + E(p2) \quad 4$$





This leads to a "divide and conquer" conclusion—it's easier to solve a complex problem when you break it into manageable pieces. The result expressed in Expression 4 has important implications with regard to modularity and software. It is, in fact, an argument for modularity.

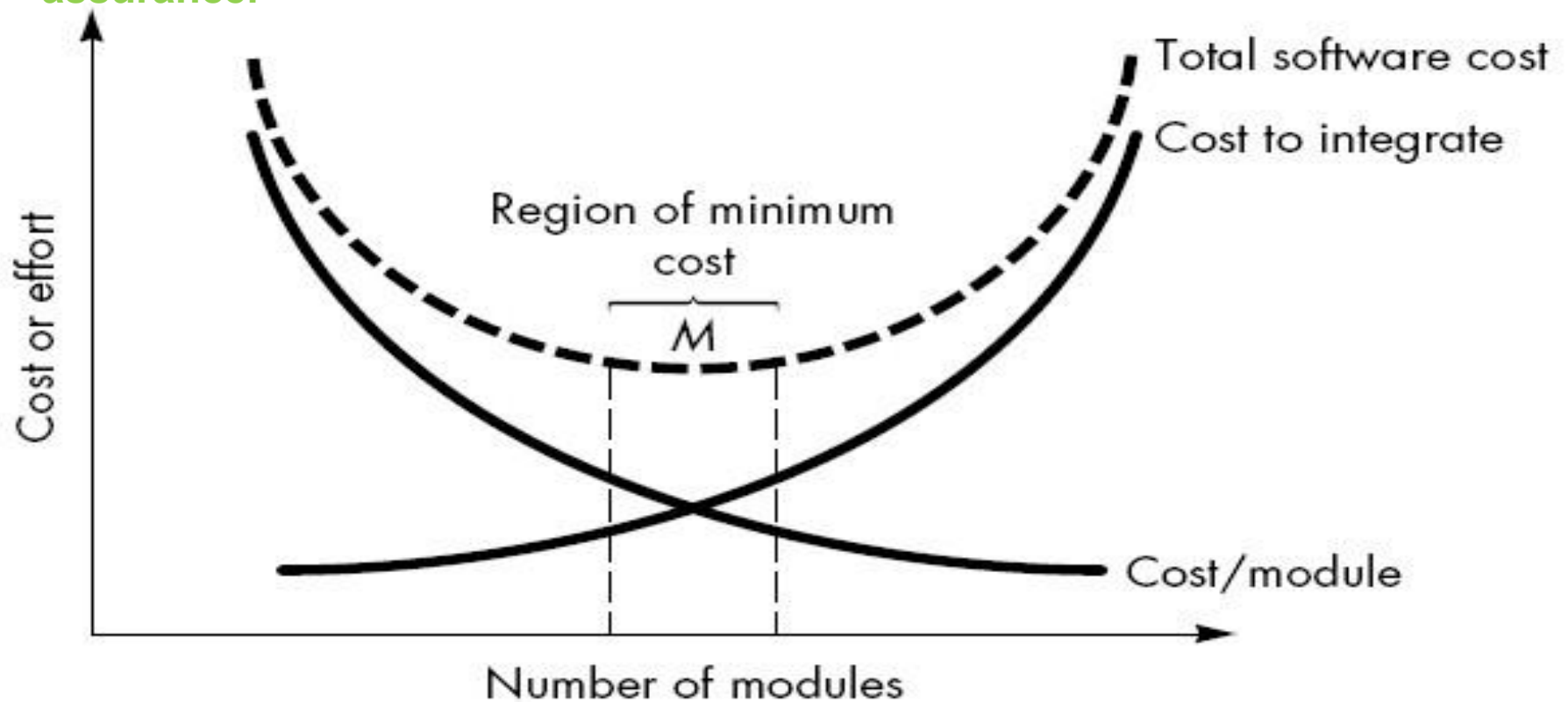
It is possible to conclude from Expression 4 that, if we subdivide software indefinitely, the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to the figure, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows.

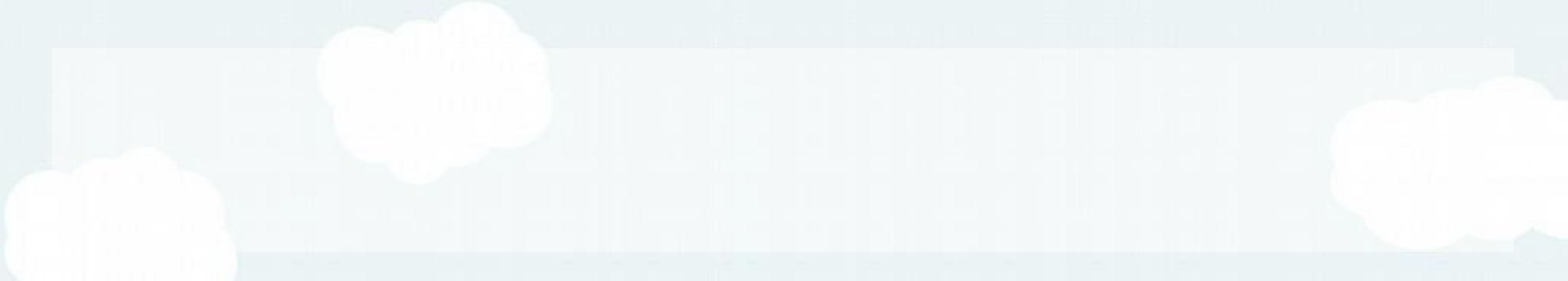


If we subdivide software indefinitely, the effort required to develop it will become negligibly small. Under modularity or over modularity should be avoided.

As the number of modules grows, the effort (cost) associated with integrating the module also grows. These characteristics lead to a total cost or effort curve shown in the figure:

There is a number,  $M$ , of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict  $M$  with assurance.

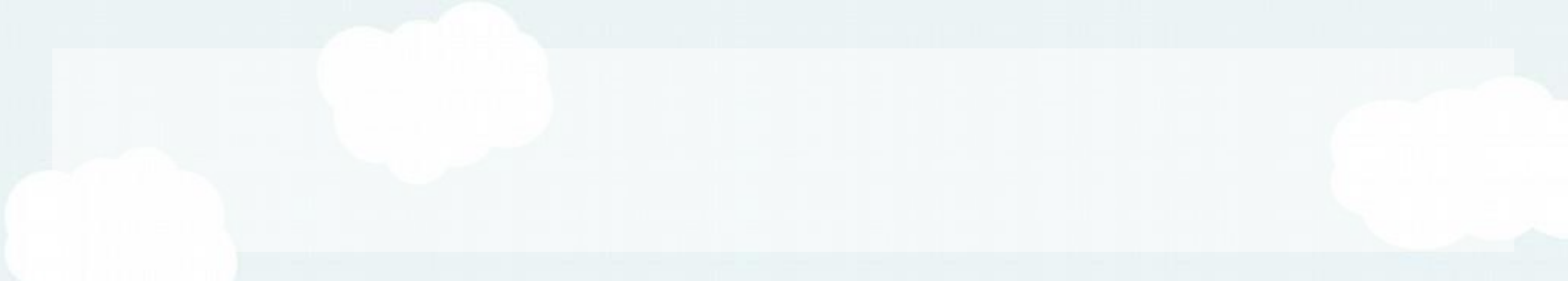




The curves shown in figure provide useful guidance when modularity is considered. We should modularize, but care should be taken to stay in the vicinity of  $M$ . Undermodularity or overmodularity should be avoided. But how do we know "the vicinity of  $M$ "? How modular should we make software?

Another important question arises when modularity is considered. How do we define an appropriate module of a given size? The answer lies in the method(s) used to define modules within a system.



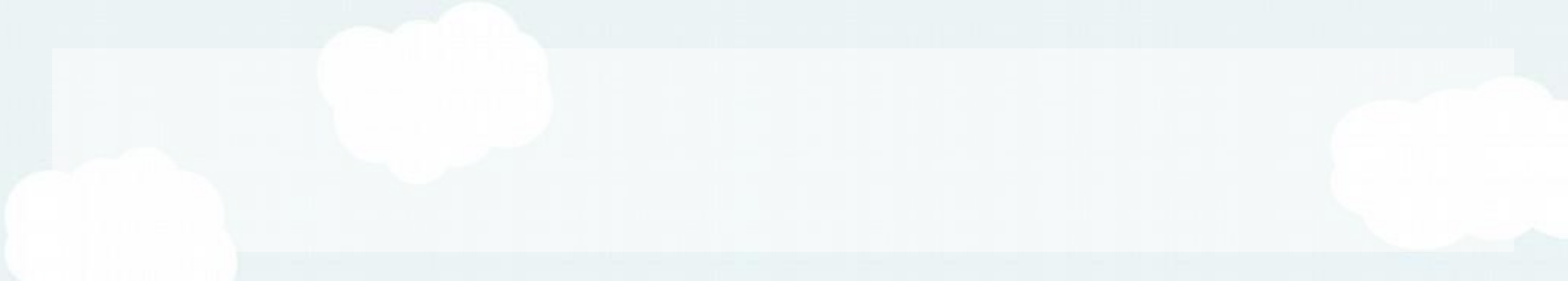


Five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

**Modular decomposability:** If a design method provides a systematic mechanism **for decomposing the problem into sub problems**, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

**Modular composability:** If a design method **enables existing (reusable) design components to be assembled into a new system**, it will yield a modular solution that does not reinvent the wheel.

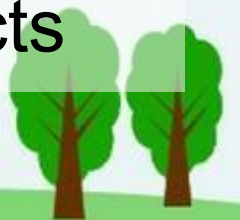


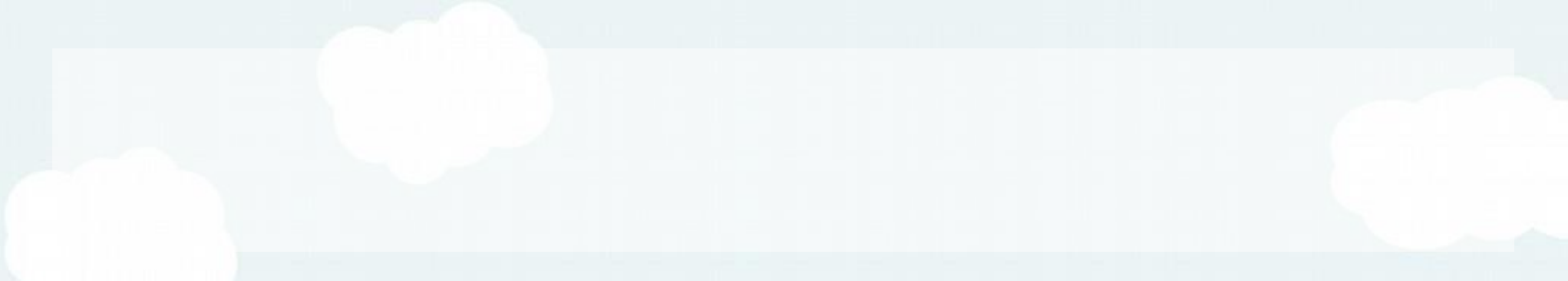


**Modular understandability:** If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.

**Modular continuity:** If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized.

**Modular protection:** If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.





Finally, it is important to note that a system may be designed modularly, even if its implementation must be "monolithic." There are situations (e.g., real-time software, embedded software) in which relatively minimal speed and memory overhead introduced by subprograms (i.e., subroutines, procedures) is unacceptable.

In such situations, software can and should be designed with modularity as an overriding philosophy. Code may be developed "in-line." Although the program source code may not look modular at first glance, the philosophy has been maintained and the program will provide the benefits of a modular system.



# 5. Control Hierarchy

In software engineering, **control hierarchy** (or program structure) defines the organizational relationship between modules without detailing the logic of how they execute.

To evaluate the complexity and organization of a control hierarchy, the following measures are commonly used:

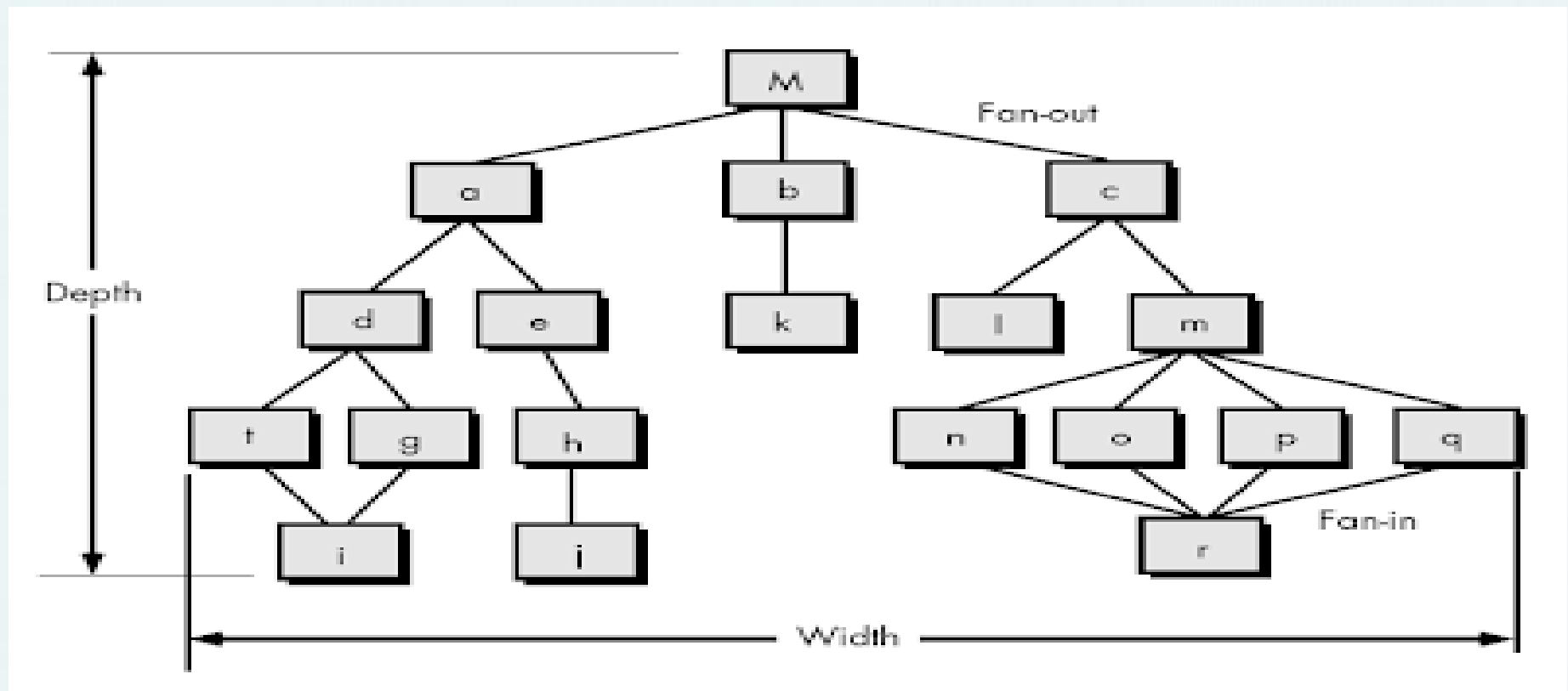
**Depth:** The total number of levels within the control hierarchy .

**Width:** The horizontal span of the architecture, representing the maximum number of modules at any single level .

**Fan-out:** The number of modules directly controlled by a specific module (also known as the "out-degree") .

**Fan-in:** The number of modules that directly control a specific module, indicating how many components "call" or reuse it





The control relationship among modules is expressed in the following way: A module that controls another module is said to be **superordinate** to it, and conversely, a module controlled by another is said to be **subordinate** to the controller. For example, referring to figure, module M is superordinate to modules a, b, and c. Module h is subordinate to module e and is ultimately subordinate to module M. Width-oriented relationships (e.g., between modules d and e) although possible to express in practice, need not be defined with explicit terminology.

The control hierarchy also represents two subtly different characteristics of the software architecture: visibility and connectivity. **Visibility** indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. For example, a module in an object-oriented system may have access to a wide array of data objects that it has inherited, but makes use of only a small number of these data objects. All of the objects are visible to the module. Connectivity indicates the set of components that are directly invoked or used as data by a given component. For example, a module that directly causes another module to begin execution is connected to it.



# 6. Structural Partitioning

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically.

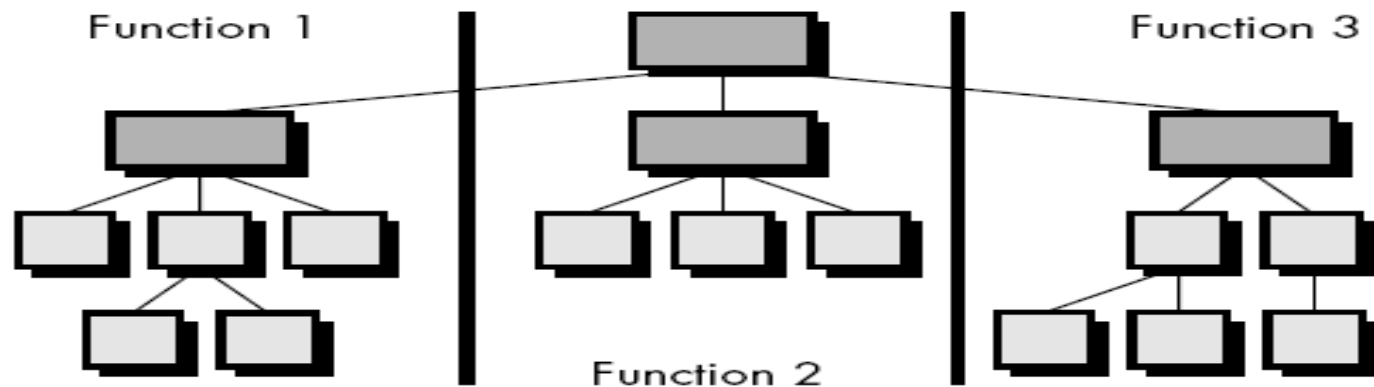
Referring to figure horizontal partitioning defines separate branches of the modular hierarchy for each major program function.

Control modules, represented in a darker shade are used to coordinate communication between and execution of the functions. The simplest approach to horizontal partitioning defines three partitions—input, data transformation (often called processing) and output. Partitioning the architecture horizontally provides a number of distinct benefits:

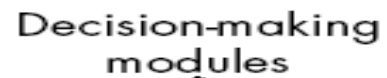
- software that is easier to test
- software that is easier to maintain
- propagation of fewer side effects
- software that is easier to extend



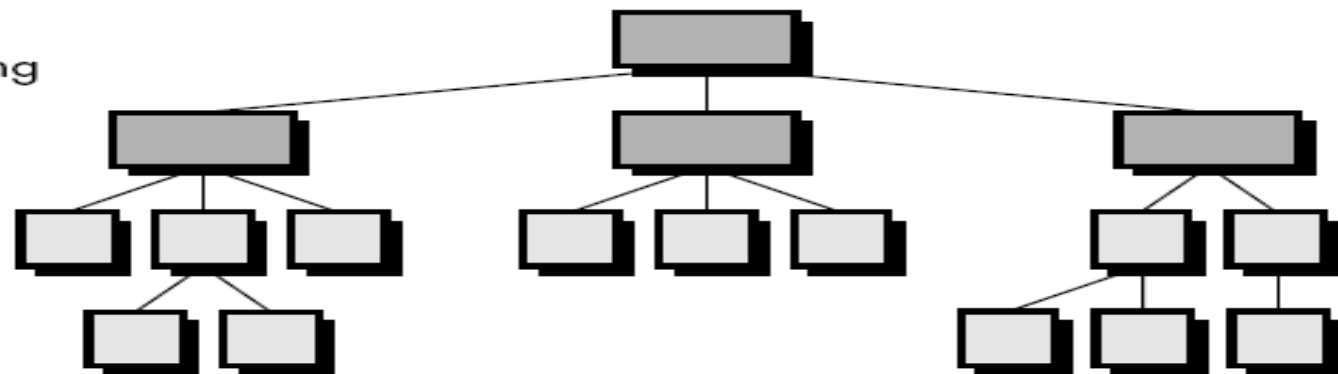
(Input,process,output)



(a) Horizontal partitioning

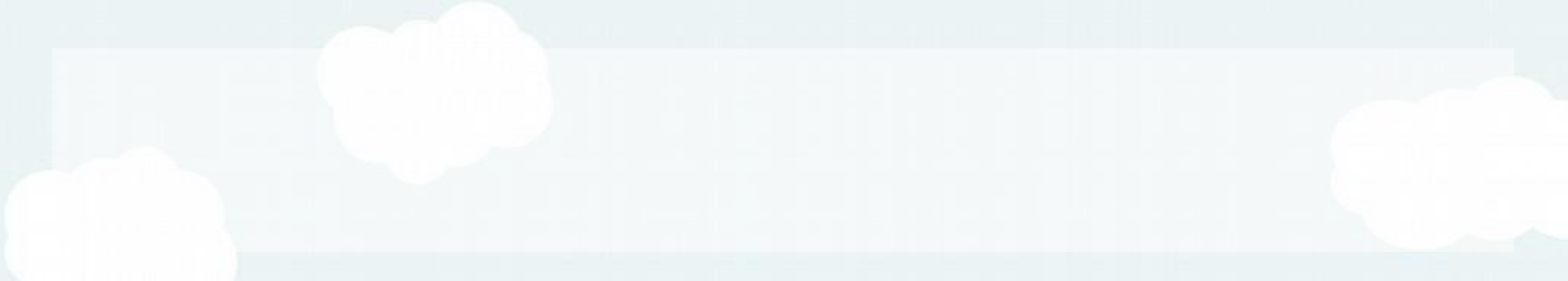


“Worker”  
modules



(b) Vertical partitioning

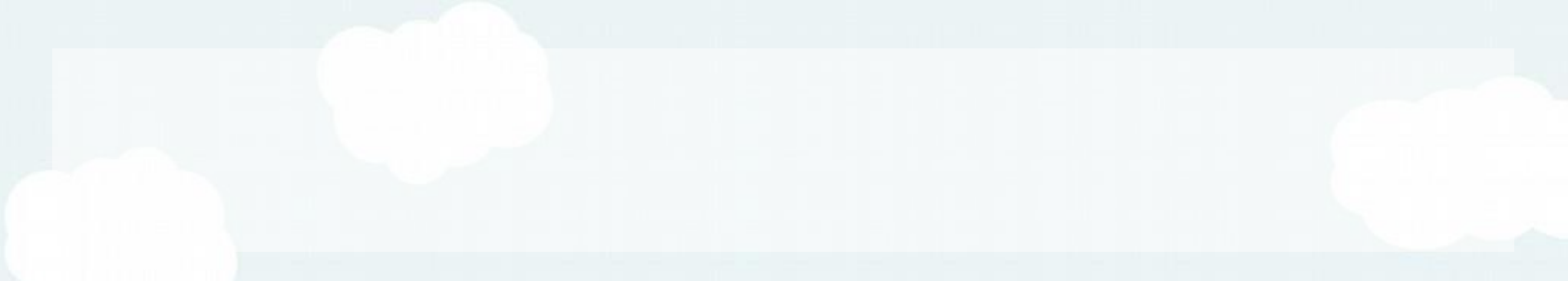




Because major functions are decoupled from one another, change tends to be less complex and extensions to the system (a common occurrence) tend to be easier to accomplish without side effects. On the negative side, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow (if processing requires rapid movement from one function to another).

Vertical partitioning (Figure (b)), often called factoring, suggests that control (decision making) and work should be distributed top-down in the program structure. Toplevel modules should perform control functions and do little actual processing work. Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

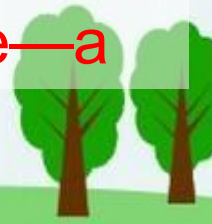




The nature of change in program structures justifies the need for vertical partitioning. Referring to figure(b), it can be seen that a change in a control module (high in the structure) will have a higher probability of propagating side effects to modules that are subordinate to it.

A change to a worker module, given its low level in the structure, is less likely to cause the propagation of side effects. In general, changes to computer programs revolve around changes to input, computation or transformation, and output.

The overall control structure of the program (i.e., its basic behavior is far less likely to change). For this reason vertically partitioned structures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable—a key quality factor.



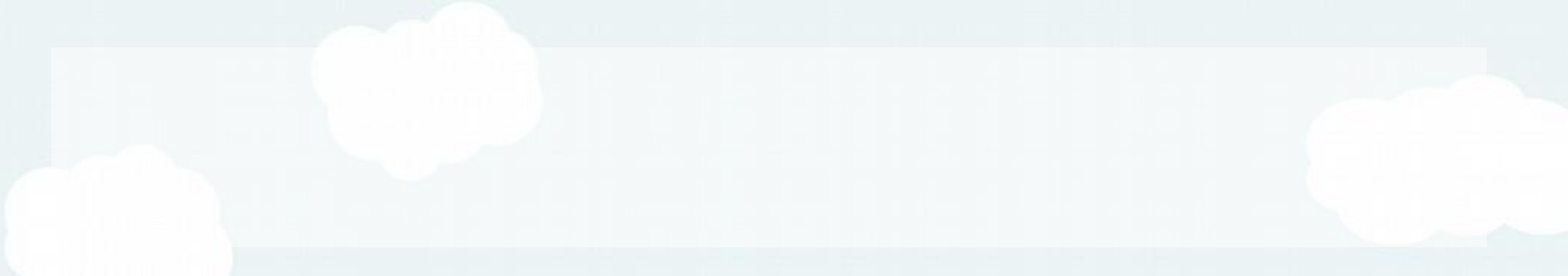
# 7.Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the representation of software architecture.

Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. It is important to understand the classic methods available for organizing information and the concepts that underlie information hierarchies. The organization and complexity of a data structure are limited only by the ingenuity of the designer. There are, however, a limited number of classic data structures that form the building blocks for more sophisticated structures.

A scalar item is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in memory. The size and format of a scalar item may vary within bounds that are dictated by a programming language. For example, a scalar item may be a logical entity one bit long, an integer or floating point number that is 8 to 64 bits long, or a character string that is hundreds or thousands of bytes long.



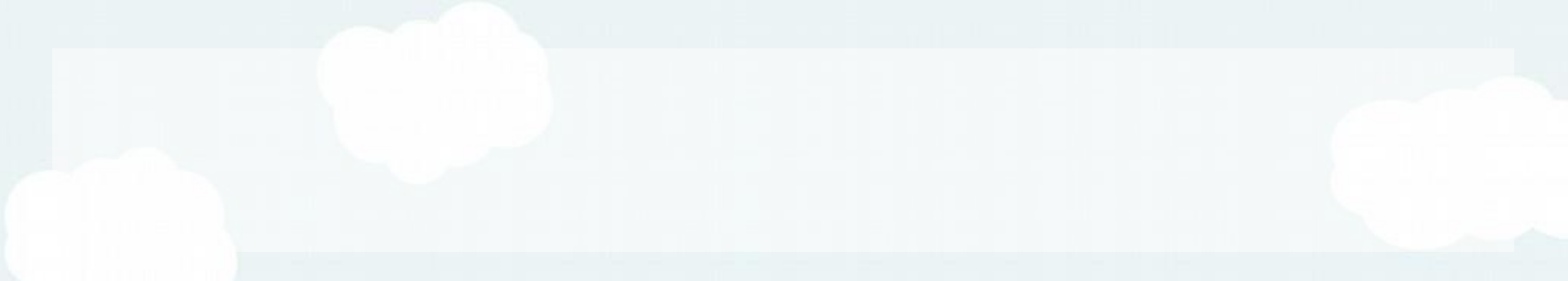


When scalar items are organized as a list or contiguous group, a sequential vector is formed. Vectors are the most common of all data structures and open the door to variable indexing of information.

When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an  $n$ -dimensional space is created. The most common  $n$ -dimensional space is the two-dimensional matrix. In many programming languages, an  $n$  dimensional space is called an array.


Items, vectors, and spaces may be organized in a variety of formats. A linked list is a data structure that organizes noncontiguous scalar items, vectors, or spaces in a manner (called nodes) that enables them to be processed as a list. Each node contains the appropriate data organization (e.g., a vector) and one or more pointers that indicate the address in storage of the next node in the list. Nodes may be added at any point in the list by redefining pointers to accommodate the new list entry.





Other data structures incorporate or are constructed using the fundamental data structures just described. For example, a hierarchical data structure is implemented using multilinked lists that contain scalar items, vectors, and possibly, n-dimensional spaces. A hierarchical structure is commonly encountered in applications that require information categorization and associativity.

It is important to note that data structures, like program structure, can be represented at different levels of abstraction. For example, a stack is a conceptual model of a data structure that can be implemented as a vector or a linked list. Depending on the level of design detail, the internal workings of a stack may or may not be specified.

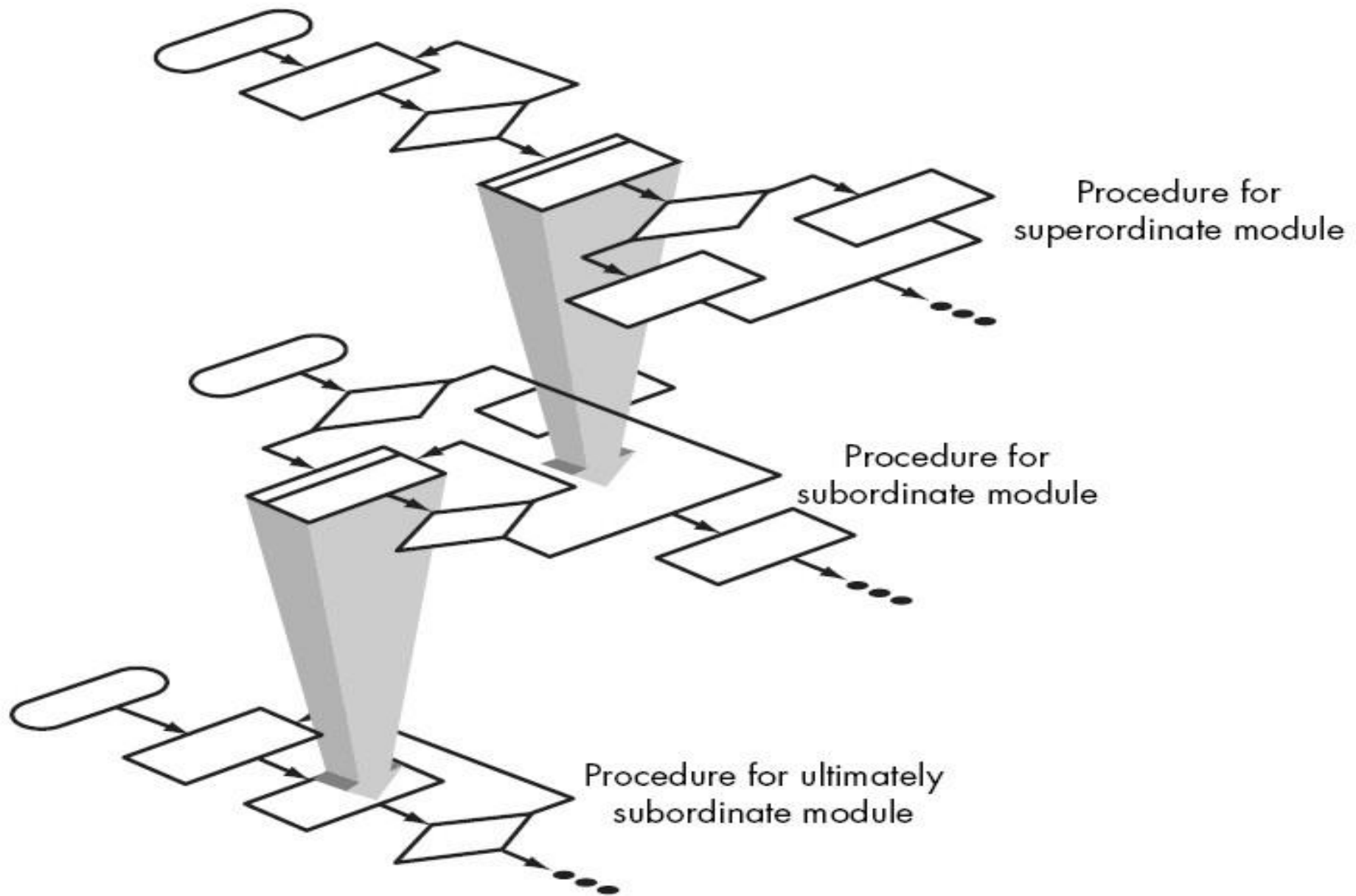


# 8. Software Procedure

Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.

There is, of course, a relationship between structure and procedure. The processing indicated for each module must include a reference to all modules subordinate to the module being described. That is, a procedural representation of software is layered as illustrated in figure.





# 9. Information Hiding

The concept of modularity leads every software designer to a fundamental question: "How do we decompose a software solution to obtain the best set of modules?" The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module .



# 9. Information Hiding

**Sign Up**

It's free and only takes a minute

First Name

Last Name

Email

Password

Confirm Password

[Submit](#)

By clicking the Sign Up button, you agree to our [Terms and Condition](#) and [Policy/Privacy](#)

Already have an account? [Login here](#)

**Login**

Email

Password

[Submit](#)

Not have an account? [Sign Up Here](#)



The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.



# Effective modular design

A modular design reduces complexity, facilitates change (a critical aspect of software maintainability), and results in easier implementation by encouraging parallel development of different parts of a system. Various concept of modular design includes:

## 1. Functional Independence

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.

We want to design software so that *each module addresses a specific sub-function of requirements* and has a simple interface when viewed from other parts of the program structure.



It is fair to ask **why independence is important.**

- ❑ Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified.
- ❑ Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.
- ❑ Functional independence is a key to good design, and design is the key to software quality.

**Independence is measured using two qualitative criteria: cohesion and coupling.**

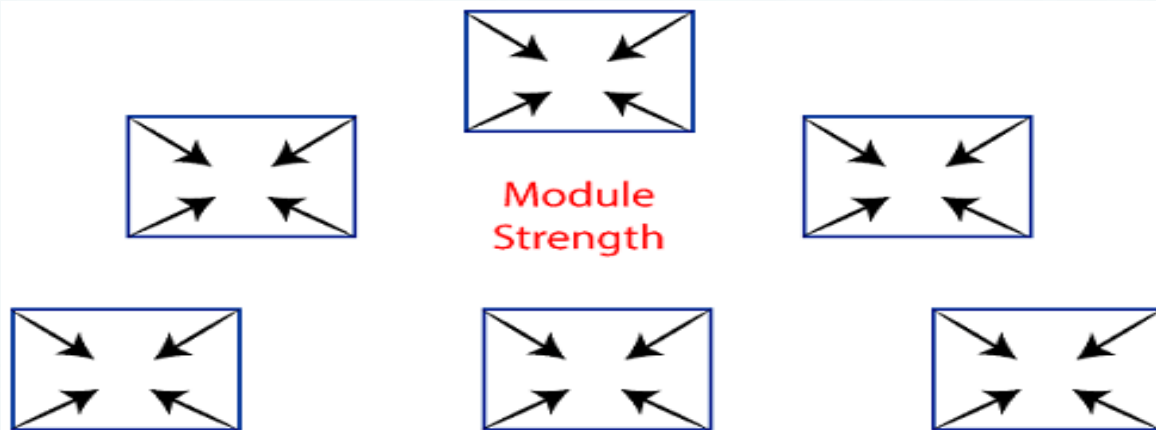


# cohesion

***Cohesion is a measure of the relative functional strength of a module.***

***We always strive for high cohesion***

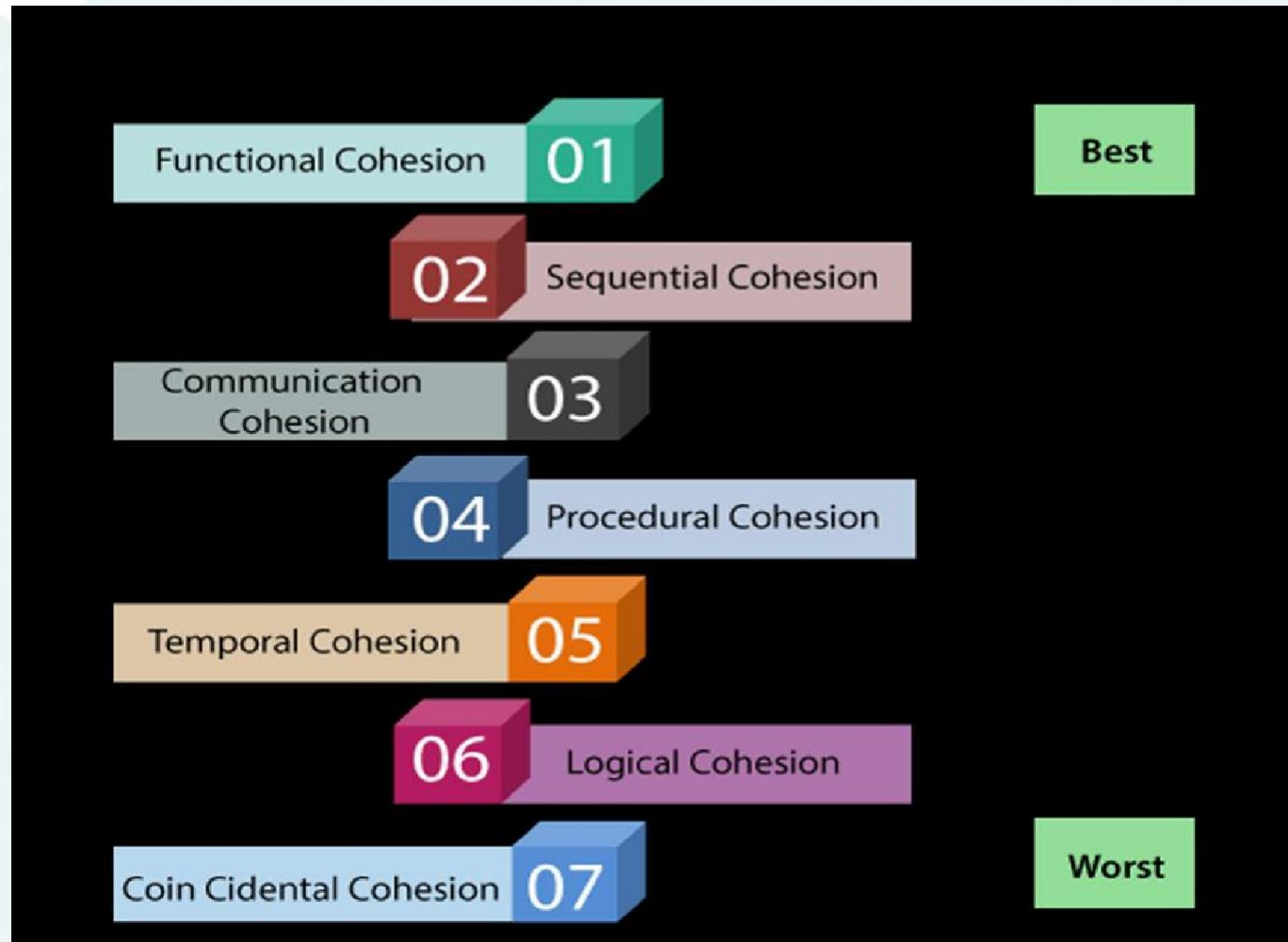
cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

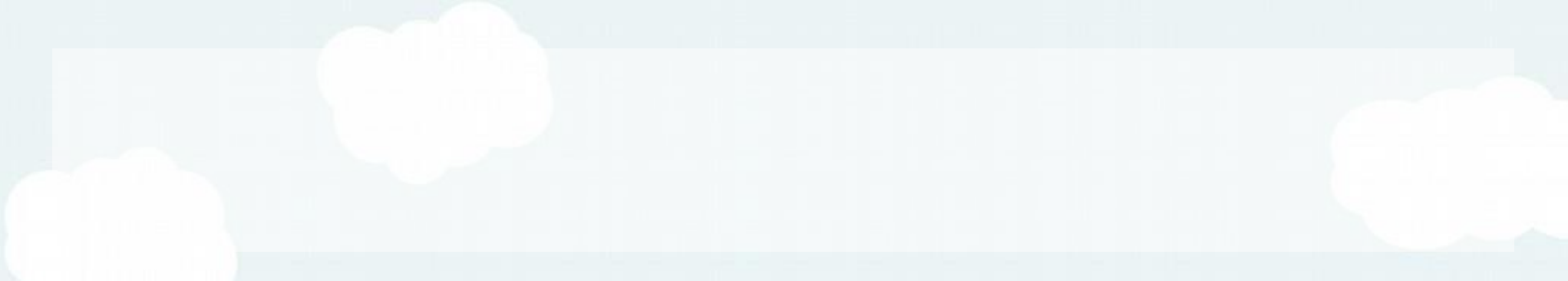


Cohesion= Strength of relations within Modules



# Types of Modules Cohesion






**Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

**Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.

**Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.

**Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

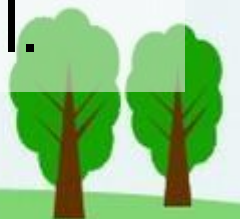




**Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.

**Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.

**Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

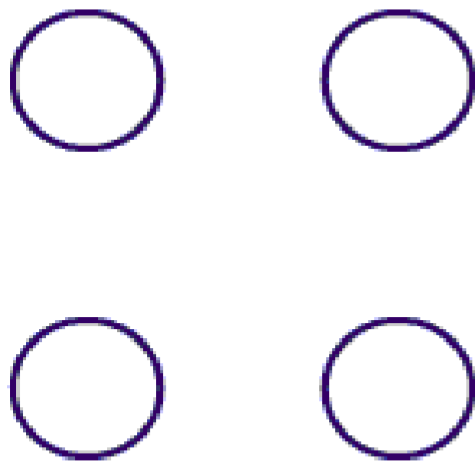


# Coupling

- *Coupling* is a measure of the *relative interdependence among modules*.
- Coupling is a measure of interconnection among modules in a software structure.
- The coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. **Uncoupled modules** have no interdependence at all within them.
- In software design, *we strive for lowest possible coupling*.
- Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.

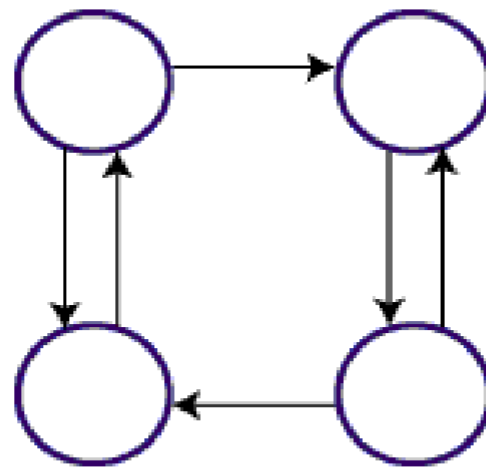


## Module Coupling



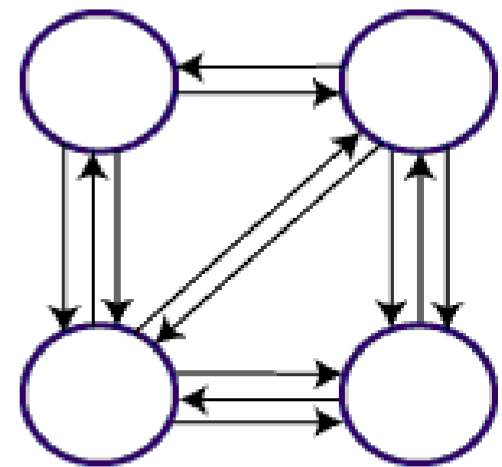
Uncoupled: no dependencies

(a)



Loosely Coupled: Some dependencies

(b)



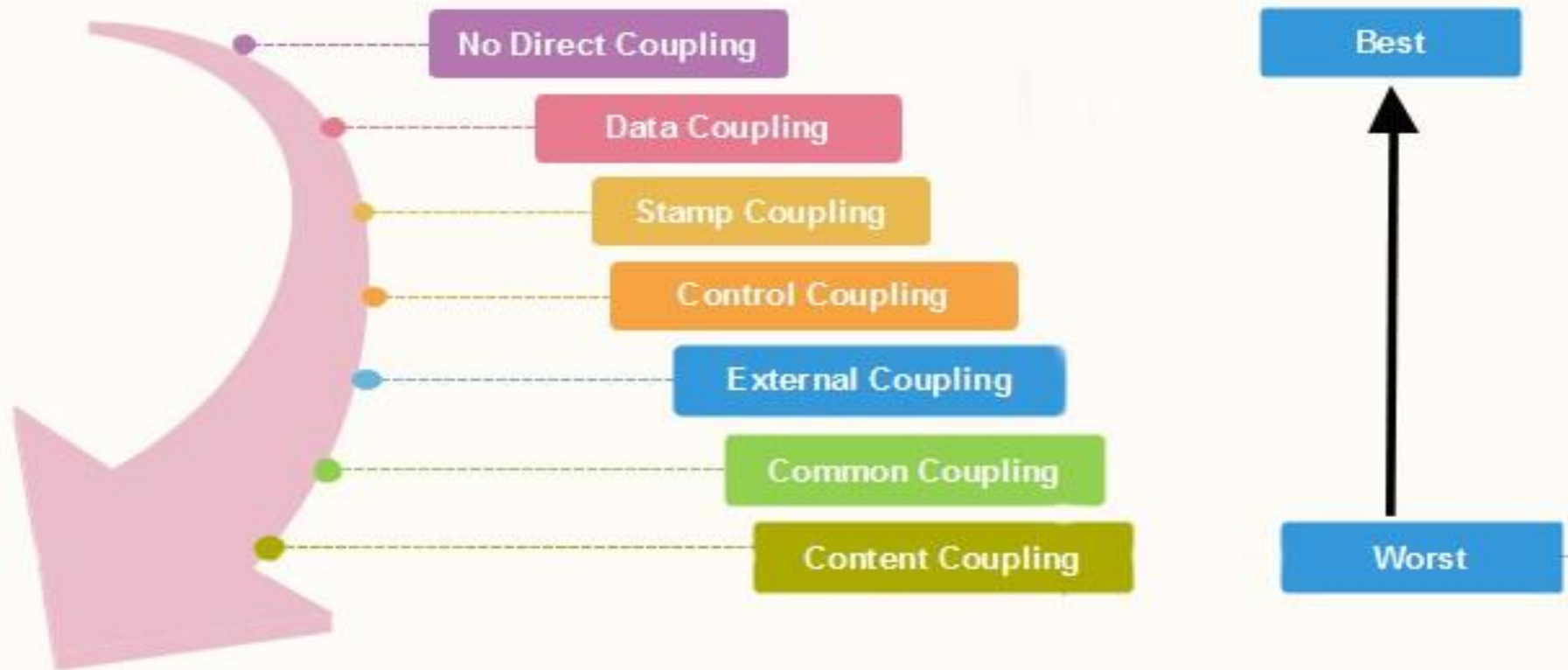
Highly Coupled: Many dependencies

(c)

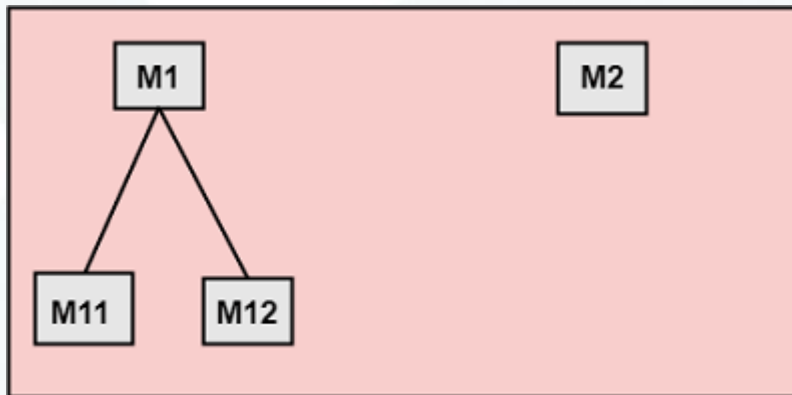


## Types of Modules Coupling

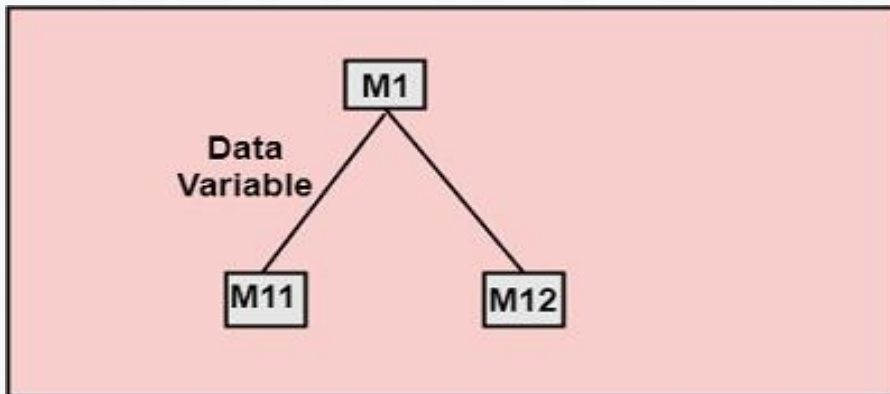
There are various types of module Coupling are as follows:

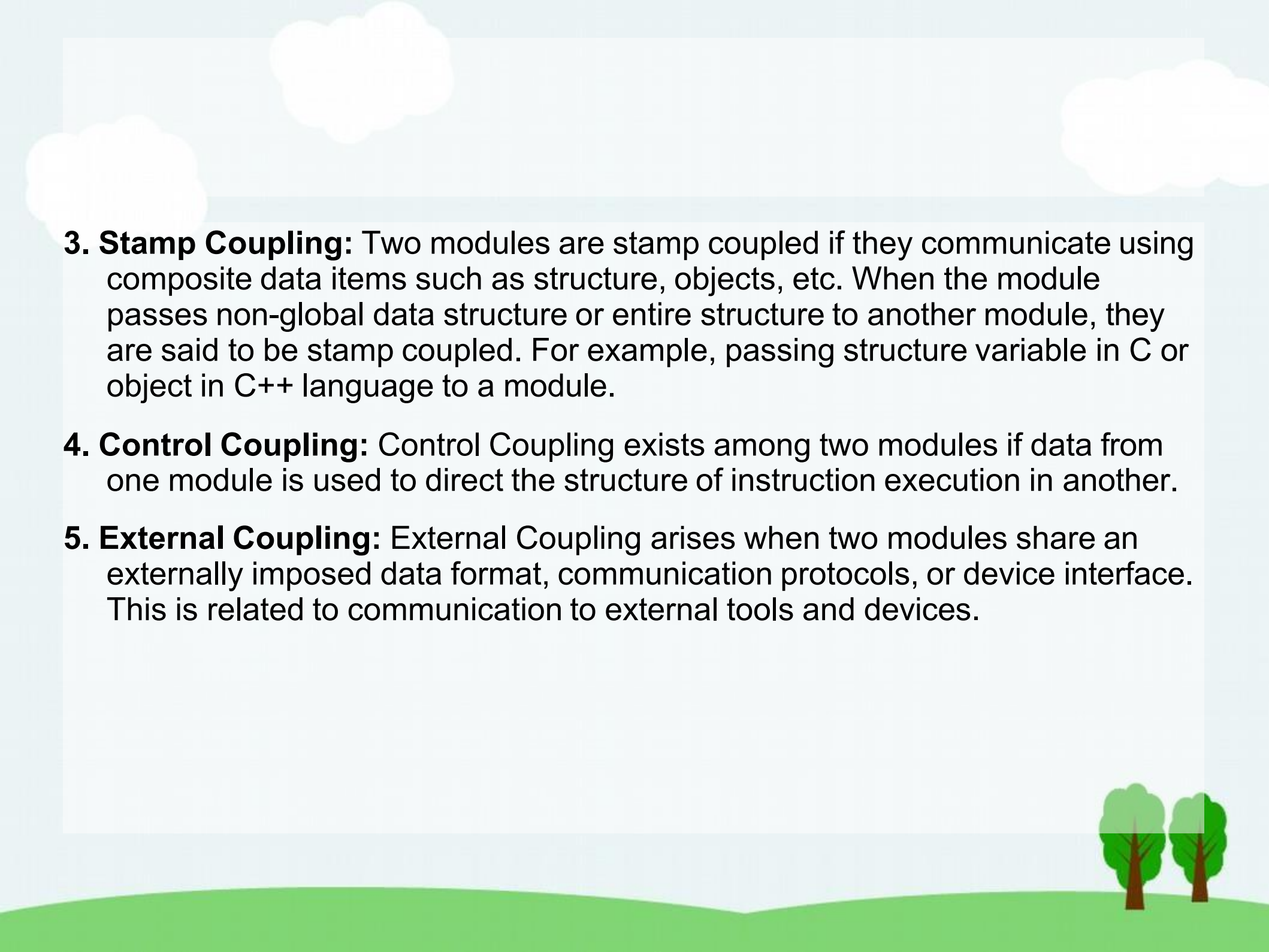


**1. No Direct Coupling:** There is no direct coupling between M1 and M2.

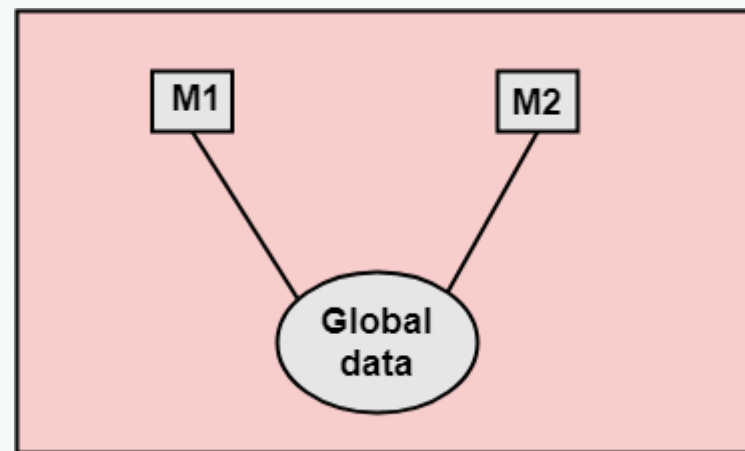


**2. Data Coupling:** When data of one module is passed to another module, this is called data coupling.



- 
3. **Stamp Coupling:** Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.
  4. **Control Coupling:** Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.
  5. **External Coupling:** External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

**6. Common Coupling:** Two modules are common coupled if they share information through some global data items.



**7. Content Coupling:** Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.



# DESIGN HEURISTICS FOR EFFECTIVE MODULARITY

Once program structure has been developed, effective modularity can be achieved by applying the design concepts introduced earlier in this chapter. The program structure can be manipulated according to the following set of heuristics:

1. *Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.*
2. *Attempt to minimize structures with high fan-out; strive for fan-in as depth increases.*
3. *Keep the scope of effect of a module within the scope of control of that module.*
4. *Evaluate module interfaces to reduce complexity and redundancy and improve consistency.*
5. *Define modules whose function is predictable, but avoid modules that are overly restrictive. .*
6. *Strive for "controlled entry" modules by avoiding "pathological connections."*



# THE DESIGN MODEL

The design principles and concepts discussed in this chapter establish a foundation for the creation of the design model that encompasses representations of data, architecture, interfaces, and components.

The design model is represented as a pyramid. The symbolism of this shape is important. A pyramid is an extremely stable object with a wide base and a low center of gravity.

Like the pyramid, we want to create a software design that is stable. By establishing a broad foundation using data design, a stable mid-region with architectural and interface design, and a sharp point by applying component-level design, we create a design model that is not easily “tipped over” by the winds of change.

But some programmers continue to design implicitly, conducting component-level design as they code. This is similar to taking the design pyramid and standing it on its point—an extremely unstable design results. The smallest change may cause the pyramid (and the program) to topple.



# DESIGN DOCUMENTATION

The *Design Specification* addresses different aspects of the design model and is completed as the designer refines his representation of the software.

First, the **overall scope of the design** effort is described. Much of the information presented here is derived from the *System Specification* and the analysis model (*Software Requirements Specification*).

Next, the **data design** is specified. Database structure, any external file structures, internal data structures, and a cross reference that connects data objects to specific files are all defined.


The **architectural design** indicates how the program architecture has been derived from the analysis model. In addition, *structure charts are used to represent the module hierarchy*.

The *design of external and internal program interfaces* is represented and a detailed design of the human/machine interface is described

*Components*—separately addressable elements of software such as subroutines, functions, or procedures—are initially described with an English-language processing narrative. Later, a procedural design tool is used to translate the narrative into a structured description.

The *Design Specification* contains a *requirements cross reference*. The purpose of this cross reference (usually represented as a simple matrix) is *to establish that all requirements are satisfied by the software design and to indicate which components are critical to the implementation of specific requirements*.





The first stage in the development of test documentation is also contained in the design document.

Once *program structure and interfaces* have been established, we can develop guidelines for testing of individual modules and integration of the entire package.

***Design constraints***, such as physical memory limitations or the necessity for a specialized external interface, may dictate special requirements for assembling or packaging of software.

Special considerations caused by the necessity for program overlay, virtual memory management, high-speed processing, or other factors may cause modification in design derived from information flow or structure.

The final section of the *Design Specification contains supplementary data*. Algorithm descriptions, alternative procedures, tabular data, excerpts from other documents, and other relevant information are presented as a special note or as a separate appendix. It may be advisable to develop a *Preliminary Operations/Installation Manual* and include it as an appendix to the design document.

