Chapter 6

System Design

System Design

- More creative than analysis
- Problem solving activity

WHAT IS DESIGN

System Design

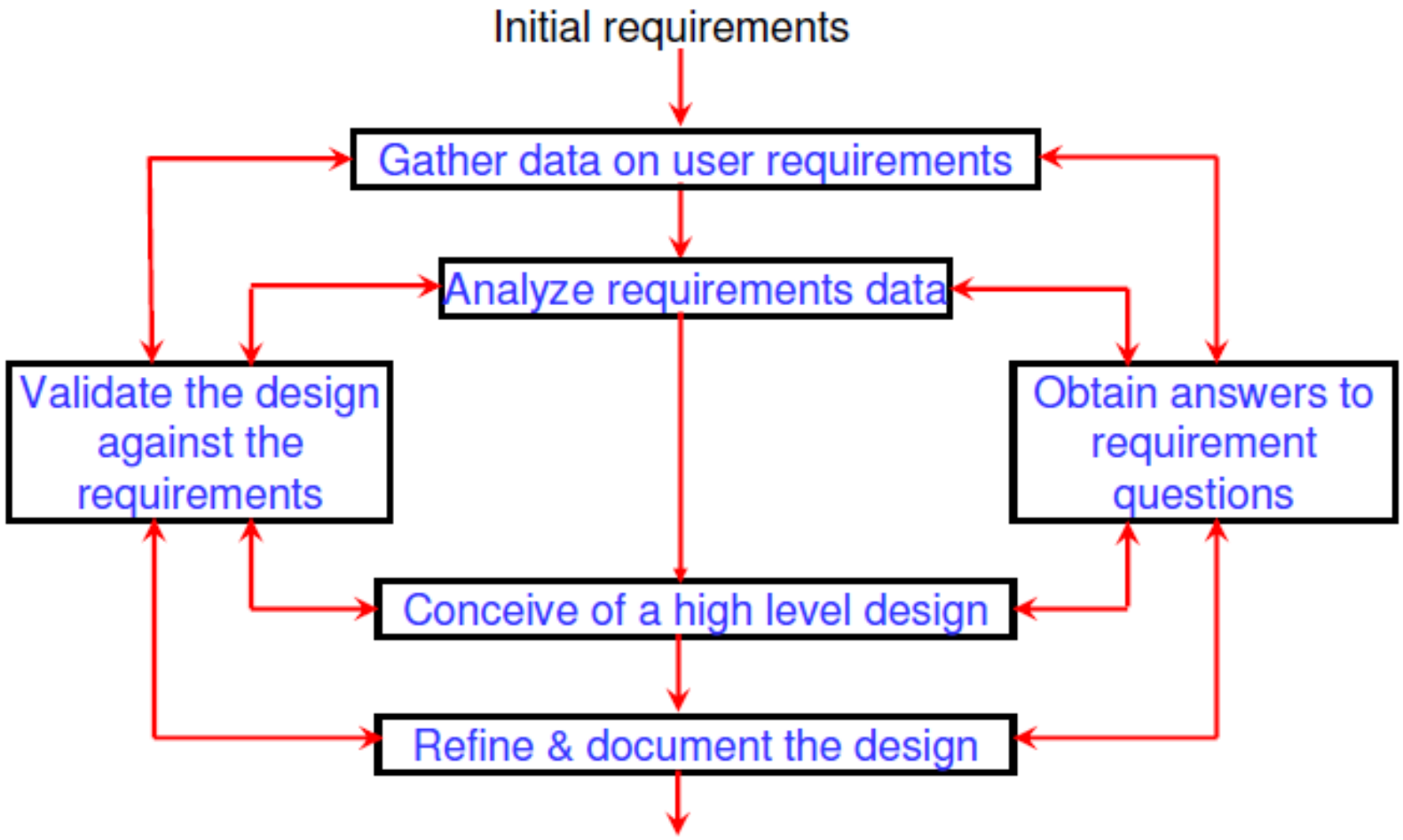
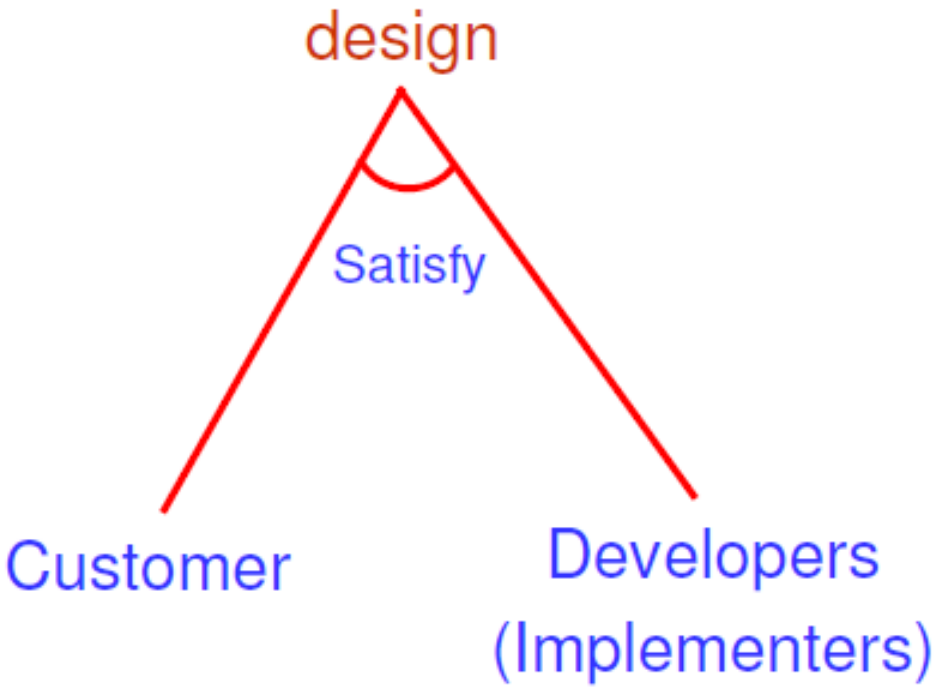


Fig. 1 : Design framework

System Design



System Design

Conceptual Design and Technical Design

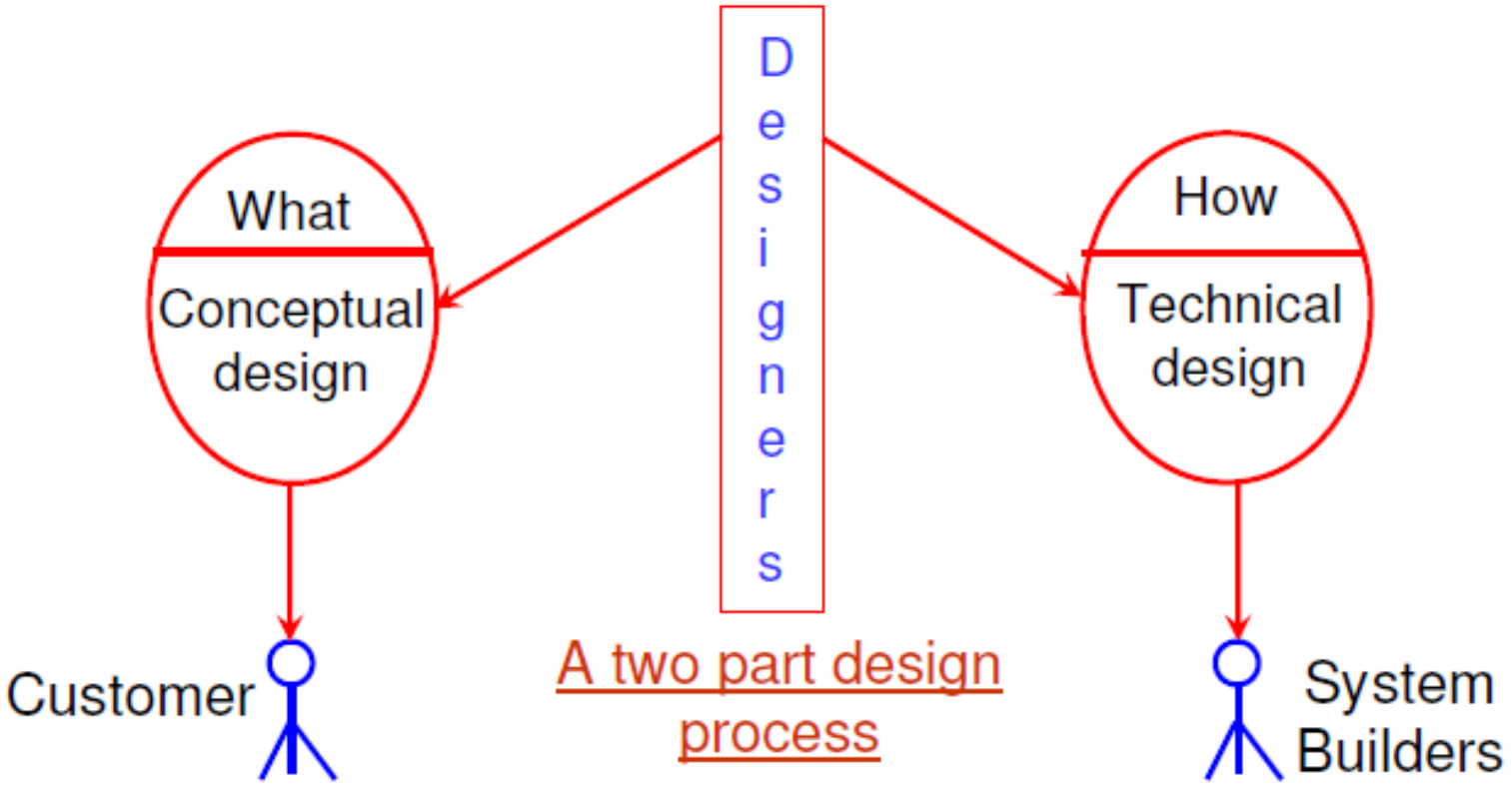


Fig. 2 : A two part design process

System Design

- Conceptual design answers :
  - Where will the data come from ?
  - What will happen to data in the system?
  - How will the system look to users?
  - What choices will be offered to users?
  - What is the timings of events?
  - How will the reports & screens look like?

System Design

- Technical design describes :
  - Hardware configuration
  - Software needs
  - Communication interfaces
  - I/O of the system
  - Software architecture
  - Network architecture
  - Any other thing that translates the requirements in to a solution to the customer's problem.

System Design

- The design needs to be
  - Correct & complete
  - Understandable
  - At the right level
  - Maintainable

System Design

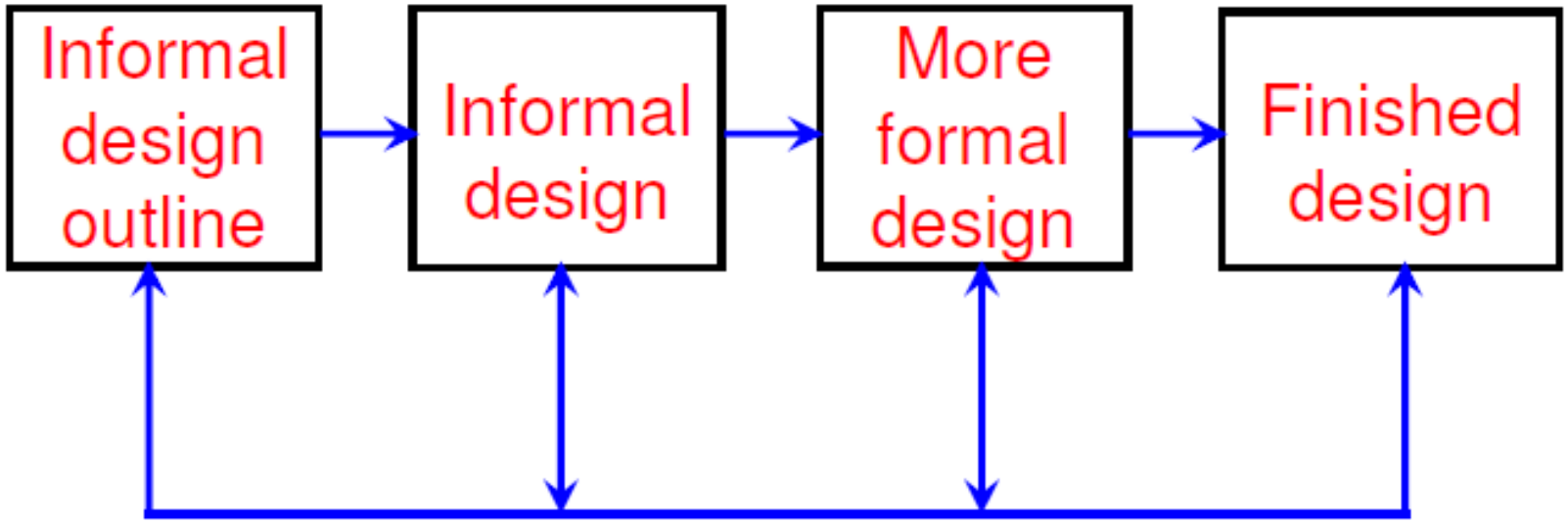


Fig. 3 : The transformation of an informal design to a detailed design.

Problem Partitioning and Hierarchy

- When solving a small problem, the entire problem can be tackled at once. The complexity of large problems and the limitations of human minds do not allow large problems to be treated as huge monoliths. For solving larger problems, the basic principle is the time-tested principle of "divide and conquer." Clearly, dividing in such a manner that all the divisions have to be conquered together is not the intent of this wisdom. This principle, if elaborated, would mean, "divide into smaller pieces, so that each piece can be conquered separately."
- For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately. It is this restriction of being able to solve each part separately that makes dividing into pieces a complex task and that many methodologies for system design aim to address. The basic rationale behind this strategy is the belief that if the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the cost of solving all the pieces.

Problem Partitioning and Hierarchy

- However, the different pieces cannot be entirely independent of each other, as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem. This communication adds complexity, which arises due to partitioning and may not have existed in the original problem. As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning. It is at this point that no further partitioning needs to be done. The designer has to make the judgment about when to stop partitioning.

Problem Partitioning and Hierarchy

- Two of the most important quality criteria for software design are simplicity and understandability. It can be argued that maintenance is minimized if each part in the system can be easily related to the application and each piece can be modified separately. If a piece can be modified separately, we call it independent of other pieces. If module A is independent of module B, then we can modify A without introducing any unanticipated side effects in B. Total independence of modules of one system is not possible, but the design process should support as much independence as possible between modules. Dependence between modules in a software system is one of the reasons for high maintenance costs. Clearly, proper partitioning will make the system easier to maintain by making the design easier to understand.

Problem Partitioning and Hierarchy

- Problem partitioning also aids design verification. Problem partitioning, which is essential for solving a complex problem, leads to hierarchies in the design. That is, the design produced by using problem partitioning can be represented as a hierarchy of components. The relationship between the elements in this hierarchy can vary depending on the method used. For example, the most common is the "whole-part of" relationship. In this, the system consists of some parts; each part consists of subparts, and so on. This relationship can be naturally represented as a hierarchical structure between various system parts. In general, hierarchical structure makes it much easier to comprehend a complex system. Due to this, all design methodologies aim to produce a design that has nice hierarchical structures.

Problem Partitioning and Hierarchy

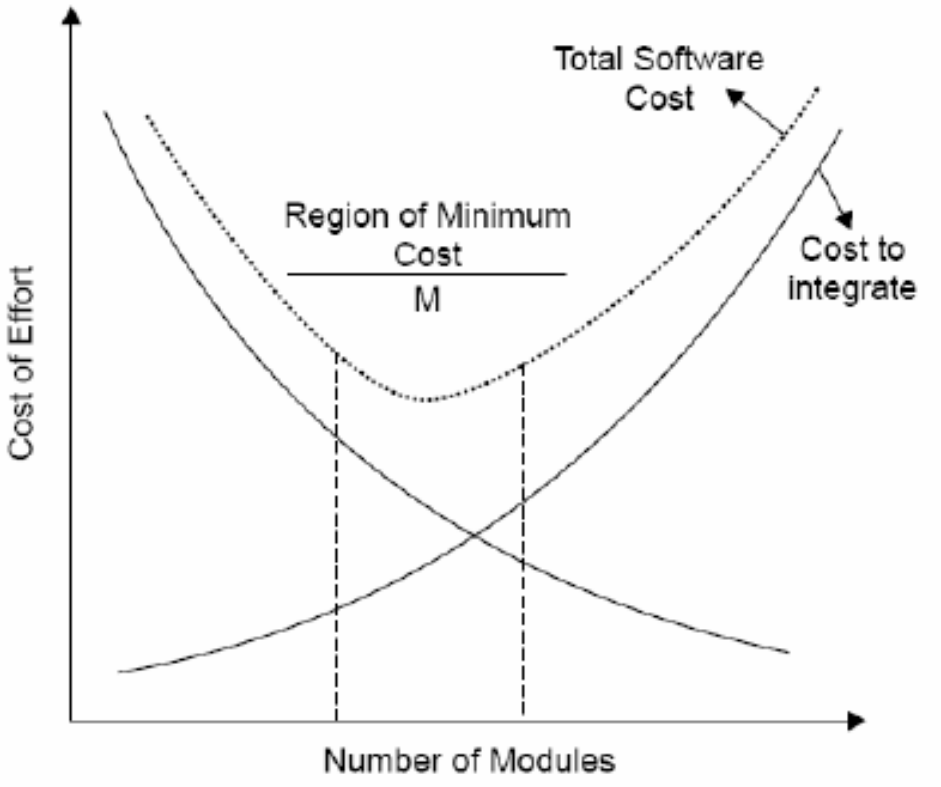
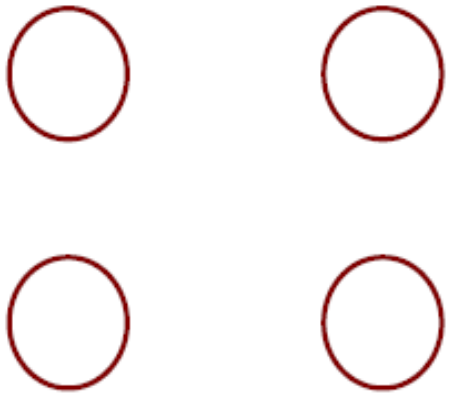


Fig. 4 : Modularity and software cost

Problem Partitioning and Hierarchy

Module Coupling

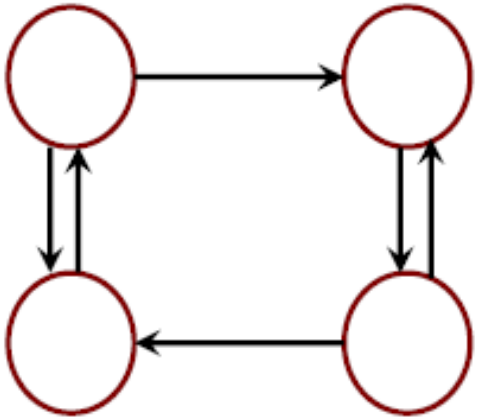
Coupling is the measure of the degree of interdependence between modules.



(Uncoupled : no dependencies)

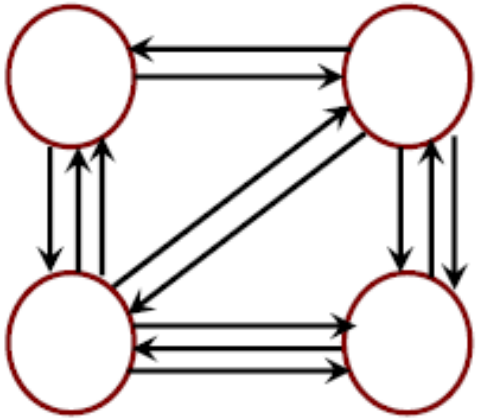
(a)

Problem Partitioning and Hierarchy



Loosely coupled:
some dependencies

(B)



Highly coupled:
many dependencies

(C)

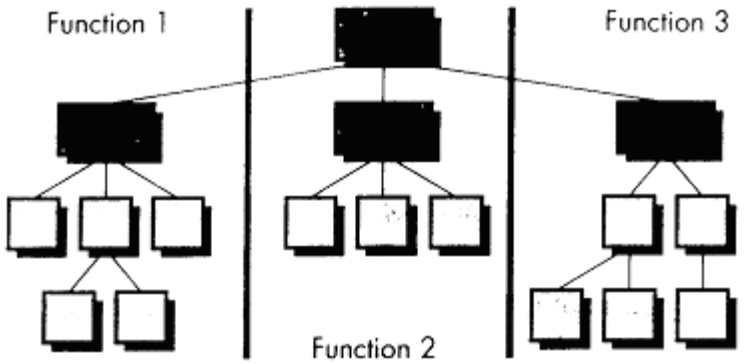
Fig. 5 : Module coupling

Problem Partitioning and Hierarchy

- This can be achieved as:
  - Controlling the number of parameters passed amongst modules.
  - Avoid passing undesired data to calling module.
  - Maintain parent / child relationship between calling & called modules.
  - Pass data, not the control information.

Structural Partitioning

- If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. Referring to Figure (a) 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.



(a) Horizontal partitioning

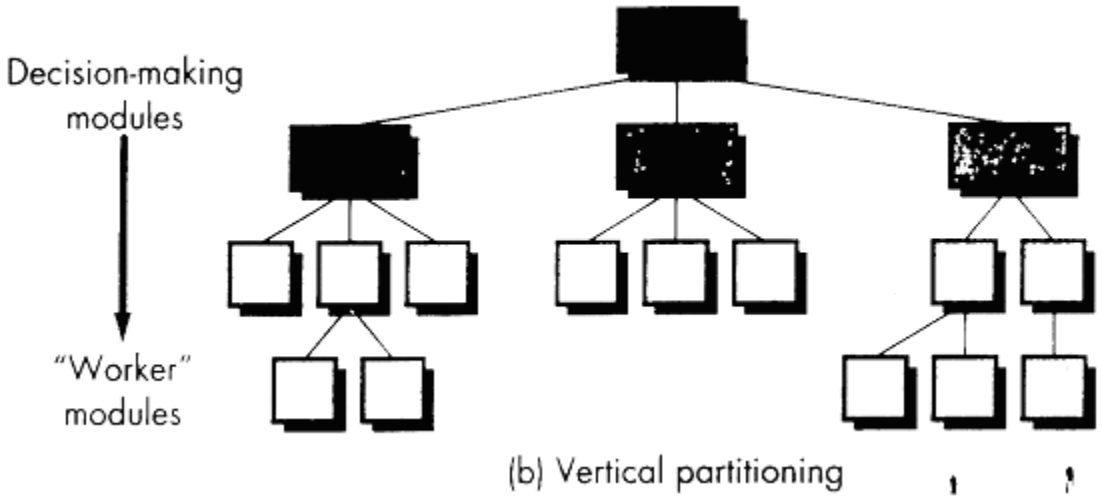
Structural Partitioning

- 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).

Structural Partitioning

- 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.

Structural Partitioning



(b) Vertical partitioning

Abstraction

- Abstraction is the intellectual tool that allows us to deal with concepts apart from particular instances of those concepts. During requirements definition and design, abstraction permits separation of the conceptual aspects of a system from the (yet to be specified) implementation details. We can for example, specify the FIFO property of a queue or the LIFO property of a stack without concern for the representation scheme to be used in implementing the stack or queue. Similarly, we can specify the functional characteristics of the routines that manipulate data structures (e.g. NEW, PUSH, POP, TOP, EMPTY) without concern for the algorithmic details of the routines.

Abstraction

- Abstraction is a very powerful concept that is used in all-engineering disciplines. It is a tool that permits a designer to consider a component at an abstract level without worrying about the details of the implementation of the component. Any component or system provides some services to its environment. An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior. Presumably, the abstract definition of a component is much simpler than the component itself.

Abstraction

- Abstraction is an indispensable part of the design process and is essential for problem partitioning. Partitioning, essentially, is the exercise of determining the components of a system. However, these components are not isolated from each other; they interact with each other, and the designer has to specify how a component interacts with other components. To decide how a component interacts with other components, the designer has to know, at the very least, the external behavior of other components. If the designer has to understand the details of the other components to determine their external behavior, we have defeated the purpose of partitioning-isolating a component from others. To allow the designer to concentrate on one component at a time, abstraction of other components is used.

Abstraction

- Abstraction is used for existing components as well as components that are being designed. Abstraction of existing components plays an important role in the maintenance phase. To modify a system, the first step is understanding what the system does and how. The process of comprehending an existing system involves identifying the abstractions of sub-systems and components from the details of their implementations. Using these abstractions, the behavior of the entire system can be understood. This also helps determine how modifying a component affects the system.

Abstraction

- As we move through different levels of abstraction, we work to create procedural and data abstractions. 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. In the context of the procedural abstraction open, we can define a data abstraction called door. 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.

Abstraction

- Control abstraction is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the synchronization semaphore used to coordinate activities in an operating system.
- During the design process, abstractions are used in the reverse manner than in the process of understanding a system. During design, the components do not exist, and in the design, the designer specifies only the abstract specifications of the different components. The basic goal of system design is to specify the modules in a system and their abstractions. Once the different modules are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is, essentially, to implement the modules so that the abstract specifications of each module are satisfied.

Abstraction

- There are two common abstraction mechanisms for software systems: functional abstraction and data abstraction. In functional abstraction, a module is specified by the function it performs. For example, a module to compute the log of a value can be abstractly represented by the function log. Similarly, a module to sort an input array can be represented by the specification of sorting. Functional abstraction is the basis of partitioning in function-oriented approaches. That is, when the problem is being partitioned, the overall transformation function for the system is partitioned into smaller functions that comprise the system function. The decomposition of the system is in terms of functional modules.

Abstraction

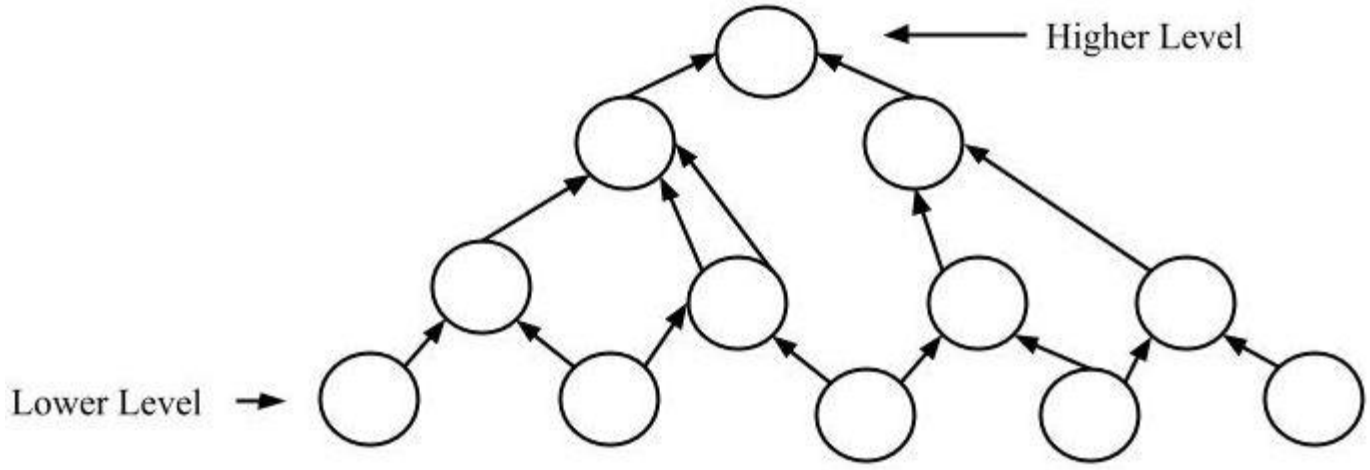
- The second unit for abstraction is data abstraction. Any entity in the real world provides some services to the environment to which it belongs. Often the entities provide some fixed predefined services. The case of data entities is similar. Certain operations are required from a data object, depending on the object and the environment in which it is used. Data abstraction supports this view. Data is not treated simply as objects, but is treated as objects with some predefined operations on them. The operations defined on a data object are the only operations that can be performed on those objects. From outside an object, the internals of the object are hidden; only the operations on the object are visible. Data abstraction forms the basis for object-oriented design. In using this abstraction, a system is viewed as a set of objects providing some services. Hence, the decomposition of the system is done with respect to the objects the system contains.

System Design Strategy

- A good system design is to organize the program modules in such a way that are easy to develop and change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program.
- **Importance :**
  - If any pre-existing code needs to be understood, organized and pieced together.
  - It is common for the project team to have to write some code and produce original programs that support the application logic of the system.
- There are many strategies or techniques for performing system design.

Bottom-up approach

- The design starts with the lowest level components and subsystems. By using these components, the next immediate higher level components and subsystems are created or composed. The process is continued till all the components and subsystems are composed into a single component, which is considered as the complete system. The amount of abstraction grows high as the design moves to more high levels. By using the basic information existing system, when a new system needs to be created, the bottom up strategy suits the purpose.



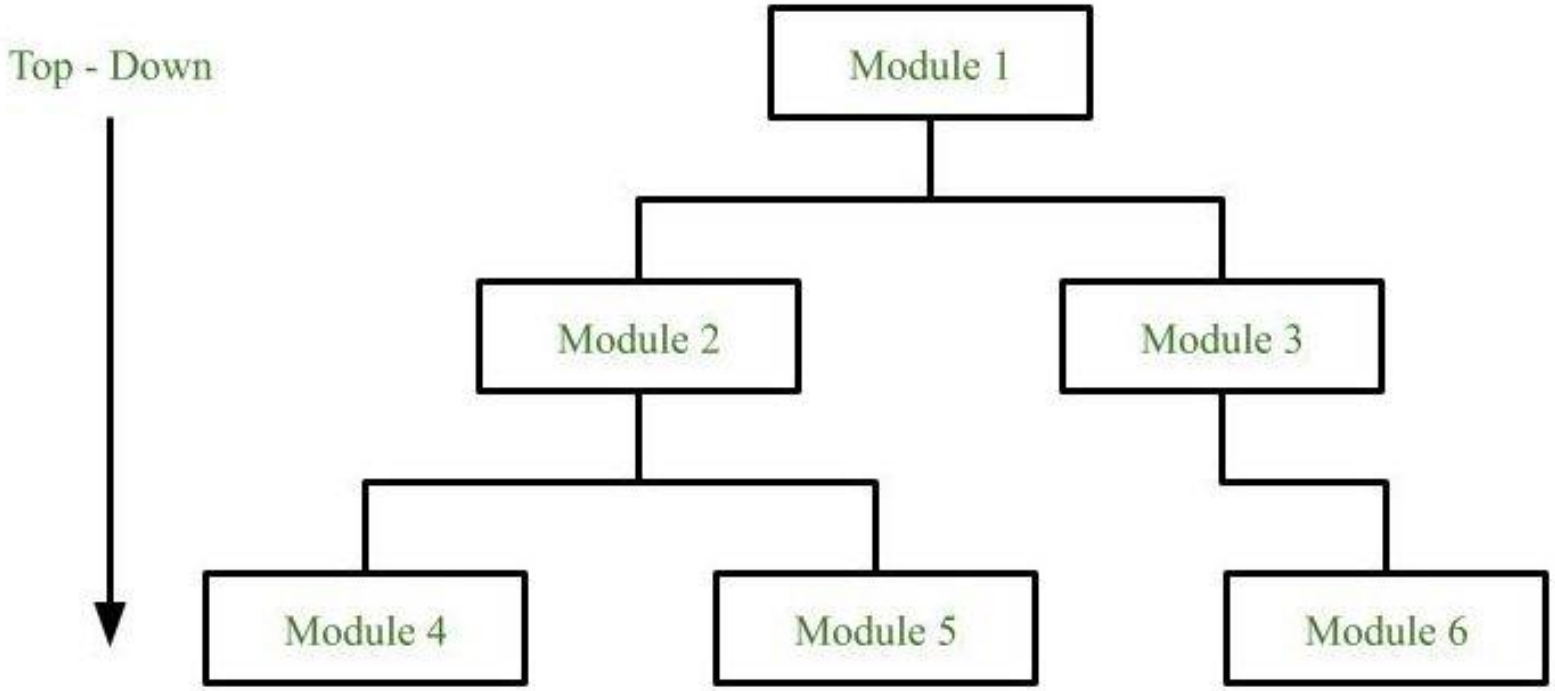
Bottom-up approach

- **Advantages:**
  - The economics can result when general solutions can be reused.
  - It can be used to hide the low-level details of implementation and be merged with top-down technique.
- **Disadvantages:**
  - It is not so closely related to the structure of the problem.
  - High quality bottom-up solutions are very hard to construct.
  - It leads to proliferation of ‘potentially useful’ functions rather than most approprite ones.

Top-down approach

- Each system is divided into several subsystems and components. Each of the subsystem is further divided into set of subsystems and components. This process of division facilitates in forming a system hierarchy structure. The complete software system is considered as a single entity and in relation to the characteristics, the system is split into sub-system and component. The same is done with each of the sub-system. This process is continued until the lowest level of the system is reached. The design is started initially by defining the system as a whole and then keeps on adding definitions of the subsystems and components. When all the definitions are combined together, it turns out to be a complete system.
- For the solutions of the software need to be developed from the ground level, top-down design best suits the purpose.

Top-down approach



Top-down approach

- **Advantages:**
  - The main advantage of top down approach is that its strong focus on requirements helps to make a design responsive according to its requirements.
- **Disadvantages:**
  - Project and system boundaries tends to be application specification oriented. Thus it is more likely that advantages of component reuse will be missed.
  - The system is likely to miss, the benefits of a well-structured, simple architecture.

Hybrid Design

- It is a combination of both the top – down and bottom – up design strategies. In this we can reuse the modules.

Modularity

- The real power of partitioning comes if a system is partitioned into modules so that the modules are solvable and modifiable separately. It will be even better if the modules are also separately compliable (then, changes in a module will not require recompilation of the whole system). A system is considered modular if it consists of discreet components so that each component can be implemented separately, and a change to one component has minimal impact on other components.
- Modularity is a clearly a desirable property in a system. Modularity helps in system debugging. Isolating the system problem to a component is easier if the system is modular. In system repair, hanging a part of the system is easy as it affects few other parts and in system building, a modular system can be easily built by “putting its modules together.”

Modularity

- A software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs to support a well-defined abstraction and have a clear interface through which it can interact with other modules. Modularity is where abstraction and partitioning come together. For easily understandable and maintainable systems, modularity is clearly the basic objective; partitioning and abstraction can be viewed as concepts that help achieve modularity.

Coupling

- A fundamental goal of software design is to structure the software product so that the number and complexity of interconnections between modules is minimized. An appealing set of heuristics, for achieving this goal, involves the concepts of coupling and cohesion.
- Two modules are considered independent if one can function completely without the presence of other. Obviously, if two modules are independent, they are solvable and modifiable separately. However, all the modules in a system cannot be independent of each other, as they must interact so that together, they produce the desired external behavior of the system. The more connections between modules, the more dependent they are in the sense that more knowledge about one module is required to understand or solve the other module. Hence, the fewer and simpler the connections between modules, the easier it is to understand one without understanding the other. The notion of coupling attempts to capture this concept of "how strongly" different modules are interconnected.

Coupling

- Coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules. In general, the more we must know about module A, in order to understand module B, the more closely connected A is to B. "Highly coupled" modules are joined by strong interconnections, while "loosely coupled" modules have weak interconnections. Independent modules have no interconnections. To solve and modify a module separately, we would like the module to be loosely coupled with other modules. The choice of modules decides the coupling between modules. Because the modules of the software system are created during system design, the coupling between modules is largely decided during system design and cannot be reduced during implementation.

Coupling

- Coupling is an abstract concept and is not easily quantifiable. So, no formulas can be given to determine the coupling between two modules. However, some major factors can be identified as influencing coupling between modules. Among them the most important are, the type of connection between modules, the complexity of the interface, and the type of information flow between modules.
- Coupling increases with the complexity and obscurity of the interface between modules. To keep coupling low, we would like to minimize the number of interfaces per module and the complexity of each interface. An interface of a module is used to pass information to and from other modules. Coupling is reduced if only the defined entry interface of a module is used by other modules (for example, passing information to and from a module, exclusively through parameters). Coupling would increase if a module were used by other modules via an indirect and obscure interface, like directly using the internals of a module or using shared variables.

Coupling

- Complexity of the interface is another factor affecting coupling. The more complex each interface is, the higher will be the degree of coupling. For example, complexity of the entry interface of a procedure depends on the number of items being passed as parameters and on the complexity of the items. Some level of complexity of interfaces is required to support the communication needed between modules. However, often more than this, minimum is used. For example, if a field of a record is needed by a procedure, often the entire record is passed, rather than just passing that field of the record. By passing the record, we are increasing the coupling unnecessarily. Essentially, we should keep the interface of a module as simple and small as possible.

Coupling

- The type of information flow along the interfaces is the third major factor-affecting coupling. There are two kinds of information that can flow along an interface: data or control. Passing or receiving control information means that the action of the module will depend on this control information, which makes it more difficult to understand the module and provide its abstraction. Transfer of data information means that a module passes some data as input to another module and gets in return some data as output. This allows a module to be treated as a simple input - output function that performs some transformation on the input data to produce the output data. In general, interfaces with only data communication result in the lowest degree of coupling, followed by interfaces that only transfer control data. Coupling is considered highest if the data is hybrid, that is, some data items and some control items are passed between modules.

End of Chapter 6

Questions?