DESIGN CONCEPTS AND PRINCIPLES (Chapter 7)

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.

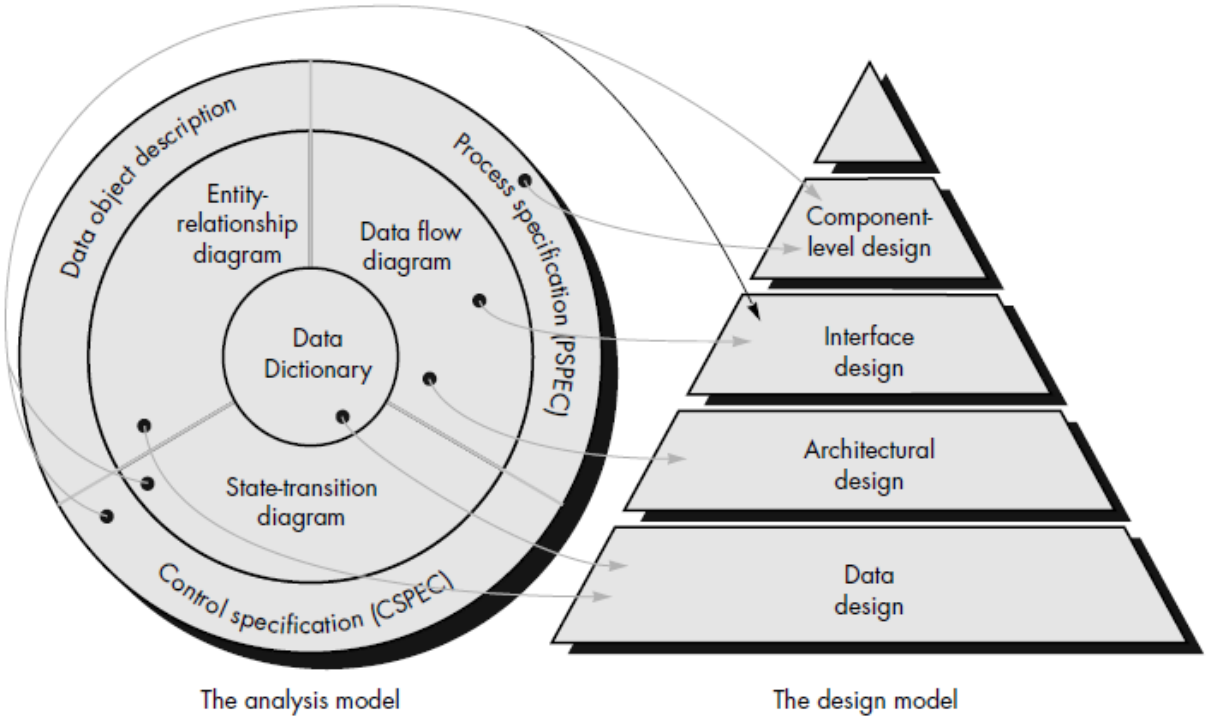


Fig: Translating the analysis model into a software design

The ***data design*** transforms the information domain model created during analysis into the data structures 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.

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.

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.

The ***component-level design*** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, and STD serve as the basis for component design.
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.

THE 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.
Following guidelines:
1. A design should exhibit an architectural structure that
(1) Has been created using recognizable design patterns,
(2) Is composed of components that exhibit good design characteristics.
(3) 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 and slowly refines the thing to provide guidance for constructing each detail. 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." 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. 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. 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" between the software and the problem as it exists in the real world.
That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
The design should exhibit uniformity and integration. A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
The design should be structured to accommodate change.
The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered. Well designed software should never "bomb." It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

Design is not coding, coding is not 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. There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. 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.

****

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.

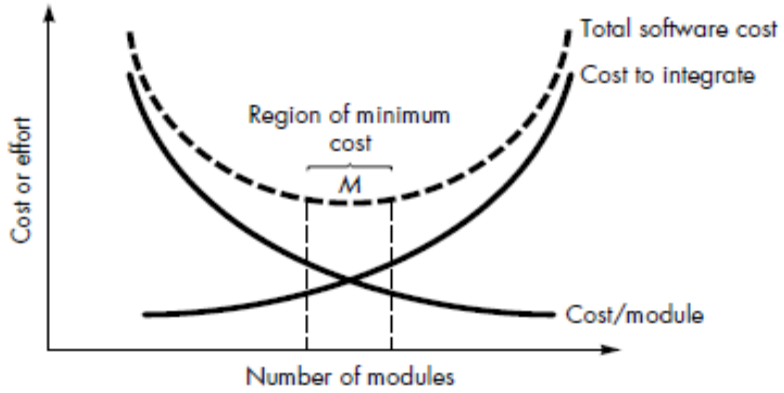
Control abstraction is the third form of abstraction used in software design; control abstraction implies a program control mechanism without specifying internal details.
An example of a control abstraction is *synchronization semaphore* used to coordinate activities in an operating system.

2. REFINEMENT

A program is developed by successively refining levels of procedural details.
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.
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.

3. 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.
If we subdivide software indefinitely, the effort required to develop it will become negligibly small. Undermodularity or overmodularity 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:



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.

4. Software Architecture

Software architecture refers to *"the overall structure of the software and the ways in which that structure provides conceptual integrity for a system"*
Architecture is the *hierarchical structure of program components* (modules), the *manner in which these components interact and the structure of data* that are used by the components.
A set of properties that should be specified as part of an architectural design:
Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.
Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.

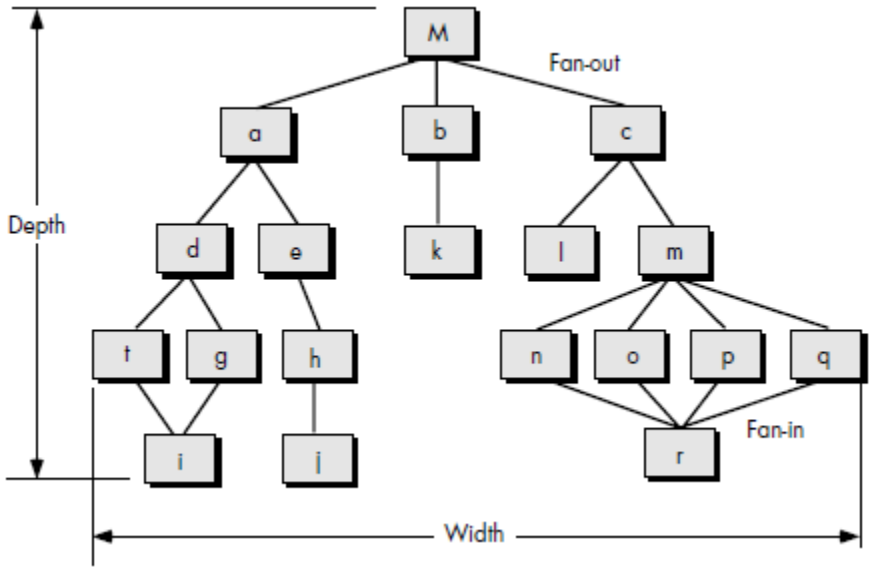
Structural models represent architecture as an organized collection of program components.
Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.
Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
Process models focus on the design of the business or technical process that the system must accommodate.
Functional models can be used to represent the functional hierarchy of a system.
A number of different *architectural description languages* (ADLs) have been developed to represent these models. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

5. Control Hierarchy

Control hierarchy, also called *program structure*, represents the organization of program components (modules) and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes.

Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation.

The most common is the treelike diagram that represents hierarchical control for call and return architectures.



Depth and *width* provide an indication of the number of levels of control and overall *span of control*, respectively. *Fan-out* is a measure of the number of modules that are directly controlled by another module. *Fan-in* indicates how many modules directly control a given module.

The control hierarchy also represents two 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.

Horizontal partitioning defines separate branches of the modular hierarchy for each major program function.

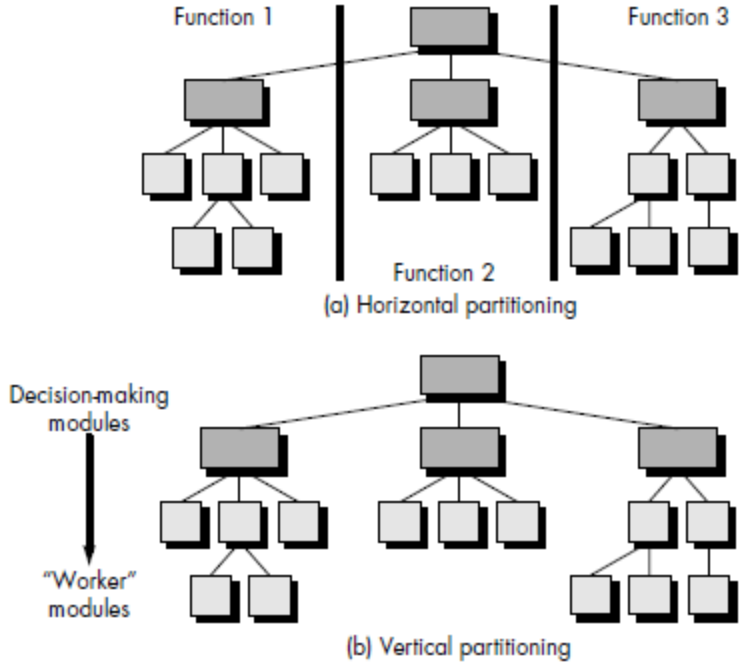
Control modules, *(represented* in a darker shade in figure) are used to coordinate communication between and execution of the functions.

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.

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.



Vertical partitioning, often called *factoring*, suggests that control (decision-making) and work should be distributed top-down in the program structure.

Top-level 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.

The nature of change in program structures justifies the need for vertical partitioning. 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.

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. When scalar items are organized as a list or contiguous group, a ***sequential vector*** is formed.

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.

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.

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.

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.

9. Information Hiding

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.

The use of information hiding 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.

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 is a measure of the relative functional strength of a module. *Coupling* is a measure of the relative interdependence among modules.

Cohesion

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

Cohesion is a natural extension of the information hiding concept.

A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program.

A Cohesive module should (ideally) do just one thing.

We always strive for high cohesion, although the mid-range of the spectrum is often acceptable.

The scale for cohesion is nonlinear. That is, low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion.

At the low (undesirable) end of the spectrum, we encounter a module that performs a set of tasks that relate to each other loosely, if at all. Such modules are termed ***coincidentally cohesive.***

A module that performs tasks that are related logically (e.g., a module that produces all output regardless of type) is ***logically cohesive***.

When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits ***temporal cohesion.***

When processing elements of a module are related and must be executed in a specific order, ***procedural cohesion*** exists.

When all processing elements concentrate on one area of a data structure, ***communicational cohesion*** is present.

It is important to strive for high cohesion and recognize low cohesion so that software design can be modified to achieve greater functional independence.

High Cohesion is characterized by a module that performs one distinct procedural task.

Coupling

Coupling is a measure of the *relative interdependence among modules.*
Coupling is a measure of interconnection among modules in a software structure.
Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
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.
Data coupling:

> Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

Stamp coupling (Data-structured coupling):

> Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it).

Control coupling:

> Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag)

External coupling:

> External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface. This is basically related to the communication to external tools and devices.

Common coupling:

> Common coupling is when two modules share the same global data (e.g., a global variable). Changing the shared resource implies changing all the modules using it.

The highest degree of coupling, *content coupling,* occurs when one module makes use of data or control information maintained within the boundary of another module. Secondarily, content coupling occurs when branches are made into the middle of a module. This mode of coupling can and should be avoided.

❖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.

ARCHITECTURE DESIGN

SOFTWARE ARCHITECTURE

Design has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements.

Architectural design represents the structure of data and program components that are required to build a computer-based system. Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:-

1. Analyze the effectiveness of the design in meeting its stated requirements.
2. Consider architectural alternatives at a stage when making design changes is still relatively easy.
3. Reducing the risks associated with the construction of the software.

Software architecture considers two levels of the design Pyramid:

1. **Data design:** data design enables us to represent the data component of the architecture.
2. **Architectural design:** Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

Three key **reasons/ significance** that software architecture is important:

1. Representations of software architecture enable *communication between all parties (stakeholders) interested in the development* of a computer-based system.
2. The architecture *highlights early design decisions* that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
3. Architecture *"constitutes a relatively small, intellectually graspable model of how the system is structured* and how its components work together"

DATA DESIGN:

Data design creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data).

The structure of data has always been an important part of software design. Data design plays an important role in following cases:-

1. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.

2. At the application level, the translation of a data model into a database is pivotal to achieving the business objectives of a system.
3. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

Data Modeling, Data Structures, Databases, and the Data Warehouse:

The data objects defined during software requirements analysis are modeled using entity/relationship diagrams and the data dictionary.
The data design activity translates these elements of the requirements model into data structures at the software component level and, when necessary, a database architecture at the application level.
Data mining techniques, also called knowledge discovery in databases (KDD) that navigate through existing databases in an attempt to extract appropriate business-level information.
Data warehouse encompasses all data that are used by a business. The intent is to enable access to "knowledge" that might not be otherwise available.
Characteristics that differentiate a data warehouse from the typical database:-
1. **Subject orientation:** A data warehouse is organized by major business subjects, rather than by business process or function.
2. **Integration:** Regardless of the source, the data exhibit consistent naming conventions, units and measures, encoding structures, and physical attributes, even when inconsistency exists across different application-oriented databases.
3. **Time variancy**: For a data warehouse, however, data can be accessed at a specific moment in time (e.g., customers contacted on the date that a new product was announced to the trade press). The typical time horizon for a data warehouse is five to ten years.
4. **Nonvolatility:** Unlike typical business application databases that undergo a continuing stream of changes (inserts, deletes, and updates), data are loaded into the warehouse, but after the original transfer, the data do not change.

Data Design at the Component Level:

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components.
Principles for data specification:
1. The **systematic analysis** principles applied to function and behavior should also be applied to data.
2. All **data structures and the operations** to be performed on each should be identified.
3. A **data dictionary** should be established and used to define both data and program design.
4. **Low-level data design** decisions should be deferred until late in the design process.
5. The **representation of data structure** should be known only to those modules that must make direct use of the data contained within the structure.
6. A **library of useful data structures** and the operations that may be applied to them should be developed.
7. A **software design and programming language** should support the specification and realization of abstract data types.
These principles form a basis for a component-level data design approach that can be integrated into both the analysis and design activities.

ARCHITECTURAL STYLES

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses:-

1. **A set of *components*** (e.g., a database, computational modules) that perform a function required by a system;
2. **A set of *connectors*** that enable "communication, co ordinations and cooperation" among components;
3. ***Constraints*** that define how components can be integrated to form the system; and
4. ***Semantic models*** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

Commonly used architectural pattern:

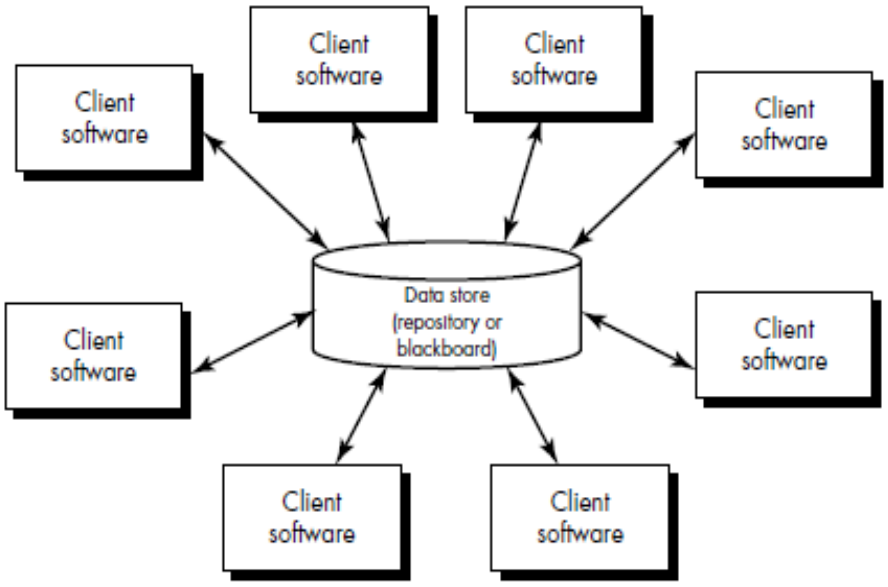
1 .Data Centered Architecture:

A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Here client software accesses the data independent of any changes to the data or the actions of other client software.

In some cases the data repository is *passive.* That is, client software accesses the data independent of any changes to the data or the actions of other client software.

Data can also be passed among clients using blackboard mechanism that is sends notifications to client software when data of interest to the client change.

Data-centered architectures promote *integrability.* That is, existing components can be changed and new client components can be added to the architecture without concern about other clients.

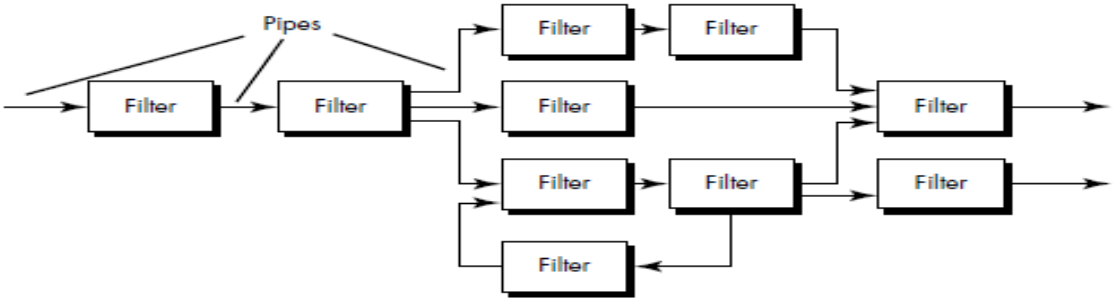


2. Data-flow architectures:

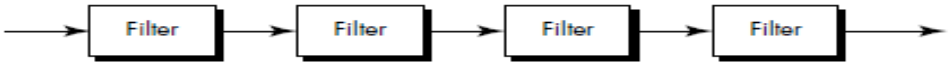
This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.

A ***pipe and filter pattern*** has a set of components, called *filters,* connected by pipes that transmit data from one component to the next and each filter works independently.

However, the filter does not require knowledge of the working of its neighboring filters.
If the data flow degenerates into a single line of transforms, it is termed ***batch sequential***.
This pattern accepts a batch of data and then applies a series of sequential components (filters) to transform it.



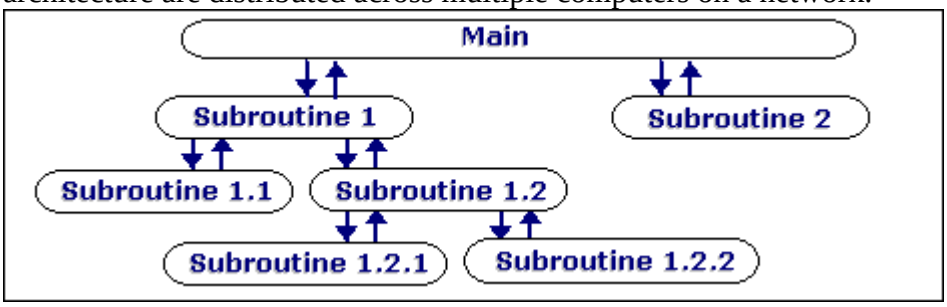
(a) Pipes and filters



(b) Batch sequential

3. Call and return architectures: This architectural style enables a software designer (system architect) to achieve a program structure that is *relatively easy to modify and scale*.
Sub styles of this type:-

- **Main program/subprogram architectures:** This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.
- **Remote procedure call architectures:** The components of a main program/subprogram architecture are distributed across multiple computers on a network.



4. Object-oriented architectures: The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.

5. Layered architectures: In this architectural style, numbers of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations.

At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

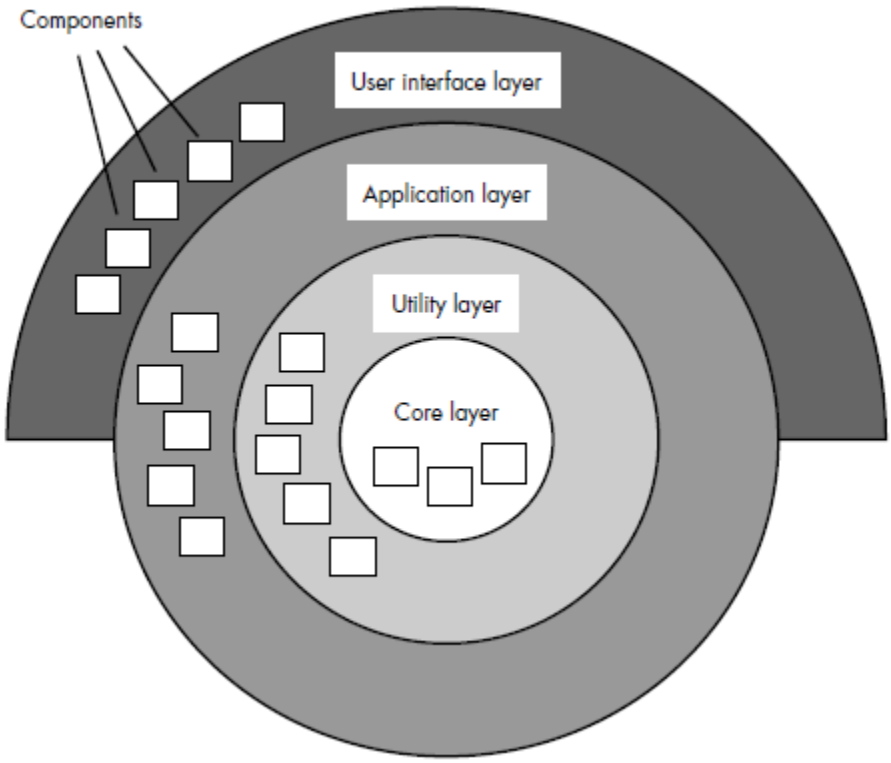


Fig: Layered Architecture

MAPPING REQUIREMENTS INTO A SOFTWARE ARCHITECTURE

Software requirements can be mapped into various representations of the design model.
The mapping technique, sometimes called *structured design*, has its origins in earlier design concepts that stressed modularity, top-down design, and structured programming.
Structured design is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture. The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six-step process:

1. The type of information flow is established;
2. Flow boundaries are indicated;
3. The DFD is mapped into program structure;
4. Control hierarchy is defined;
5. Resultant structure is refined using design measures and heuristics; and
6. The architectural description is refined and elaborated.

Transform Flow

Recalling the fundamental system model (level 0 data flow diagram), information must enter and exit software in an "external world" form.

For example, *data typed on a keyboard, tones on a telephone line, and video images in a multimedia application are all forms of external world information*.

Such externalized data must be converted into an internal form for processing. Information enters the system along paths that transform external data into an internal form. These paths are identified as *incoming flow.* At the kernel of the software, a transition occurs. Incoming data are passed through a *transform center* and begin to move along paths that now lead "out" of the software. Data moving along these paths are called *outgoing flow.* The overall flow of data occurs in a sequential manner and follows one, or only a few, "straight line" paths.

When a segment of a data flow diagram exhibits these characteristics, *transform flow* is present.

Transaction Flow

The fundamental system model implies transform flow; therefore, it is possible to characterize all data flow in this category. However, information flow is often characterized by a single data item, called a *transaction* that triggers other data flow along one of many paths. When a DFD takes the form, *transaction flow* is present. Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated and, based on its value; flow along one of many *action paths* is initiated. The hub of information flow from which many action paths emanate is called a *transaction center.* It should be noted that, within a DFD for a large system, both transform and transaction flow may be present. For example, in a transaction-oriented flow, information flow along an action path may have transform flow characteristics.

TRANSFORM MAPPING

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style**.**

Design Steps:

Step 1

Review the fundamental system model. The fundamental system model encompasses the level 0 DFD and supporting information. In actuality, the design step begins with an evaluation of both the System Specification and the Software Requirements Specification.

Step 2

Review and refine data flow diagrams for the software. Information obtained from analysis models contained in the *Software Requirements Specification* is refined to produce greater detail.

Step 3

Determine whether the DFD has transform or transaction flow characteristics. In this step, the designer selects global (software wide) flow characteristics based on the prevailing nature of the DFD. In addition, local regions of transform or transaction flow are isolated. These *subflows* can be used to refine program architecture derived from a global characteristic described.

Step 4

Isolate the transform center by specifying incoming and outgoing flow boundaries.

The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a

boundary (e.g, an incoming flow boundary separating *read sensors* and *acquire response info* could be proposed). The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

Step 5

Perform "first-level factoring." Program structure represents a top-down distribution of control. Factoring results in a program structure in which top-level modules perform decision making and low-level modules perform most input, computation, and output work. Middle-level modules perform some control and do moderate amounts of work. When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing.

Step 6

Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture.

Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. Two or even three bubbles can be combined and represented as one module (recalling potential problems with cohesion) or a single bubble may be expanded to two or more modules. Practical considerations and measures of design quality dictate the outcome of second level factoring. Review and refinement may lead to changes in this structure, but it can serve as a "first-iteration" design.

Step 7

Refine the first-iteration architecture using design heuristics for improved software quality. First-iteration architecture can always be refined by applying concepts of module independence. Modules are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

TRANSACTION MAPPING

In many software applications, a single data item triggers one or a number of information flows that affect a function implied by the triggering data item. In this section we consider design steps used to treat transaction flow.

1. Review the fundamental system model.

2. Review and refine data flow diagrams for the software.

3. Determine whether the DFD has transform or transaction flow characteristics.

(Steps 1, 2, and 3 are identical to corresponding steps in transform mapping.)

Step 4

Identify the transaction center and the flow characteristics along each of the action paths. The location of the transaction center can be immediately discerned from the DFD. The transaction center lies at the origin of a number of actions paths that flow radially from it.

Step 5

Map the DFD in a program structure amenable to transaction processing.

Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch. The structure of the incoming branch is developed in much the same way as transform mapping. Starting at the transaction center, bubbles along the incoming path are mapped into

modules. The structure of the dispatch branch contains a dispatcher module that controls all subordinate action modules. Each action flow path of the DFD is mapped to a structure that corresponds to its specific flow characteristics.

Step 6

Factor and refine the transaction structure and the structure of each action path.

Each action path of the data flow diagram has its own information flow characteristics. We have already noted that transform or transaction flow may be encountered. The action path-related "substructure" is developed using the design steps discussed in this and the preceding section.

Step 7

Refine the first-iteration architecture using design heuristics for improved software quality. This step for transaction mapping is identical to the corresponding step for transform mapping. In both design approaches, criteria such as module independence, practicality (efficacy of implementation and test), and maintainability must be carefully considered as structural modifications are proposed.