3. Analysis, design concepts and principles

**OBJECTIVE**

- To introduce the analysis and design concepts.
- To introduce design heuristics and architectural design.
- To explain data design for design concepts.
- To explain about the User interface design.
- To define the R-T executives for the analysis and design process.
- To introduce the concept of data acquisition system.
- To know about the monitoring and control system and defining to implement them.

**TOPICS COVERED**

- Analysis Concepts
- Design Process And Concepts
- Architectural design
- Data Design
- User Interface Design
- Real Time Software Design
- System Design
- Real Time Executives
- Data Acquisition System
- Monitoring and Control System.
A. REQUIREMENTS ANALYSIS

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design. Requirements engineering activities result in the specification of software's operational characteristics (function, data, and behavior), indicate software's interface with other system elements, and establish constraints that software must meet. Requirements analysis allows the software engineer (sometimes called analyst in this role) to refine the software allocation and build models of the data, functional, and behavioral domains that will be treated by software.

Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs. Finally, the requirements specification provides the developer and the customer with the means to assess quality once software is built.

Software requirements analysis may be divided into five areas of effort:

1. Problem recognition,
2. Evaluation and synthesis,
3. Modeling,
4. Specification, and
5. Review.
B. REQUIREMENTS ELICITATION FOR SOFTWARE

Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process. A customer has a problem that may be amenable to a computer-based solution. A developer responds to the customer's request for help. Communication has begun. But, as we have already noted, the road from communication to understanding is often full of potholes.

**Initiating the Process**

The most commonly used requirements elicitation technique is to conduct a meeting or interview. The first meeting between a software engineer (the analyst) and the customer can be likened to the awkwardness of a first date between two adolescents. Neither person knows what to say or ask; both are worried that what they do say will be misinterpreted; both are thinking about where it might lead (both likely have radically different expectations here); both want to get the thing over with, but at the same time, both want it to be a success.

**Facilitated Application Specification Techniques**

**Introduction**

Too often, customers and software engineers have an unconscious "us and them" mindset. Misunderstandings abound, important information is omitted, and a successful working relationship is never established. It is with these problems in mind that a number of independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification called *facilitated application specification techniques (FAST)*.

**Description**

This approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements. FAST has been used predominantly by the information systems community, but the technique offers potential for improved communication in applications of all kinds.
Many different approaches to FAST have been proposed. The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal. FAST is not a panacea for the problems encountered in early requirements elicitation. But the team approach provides the benefits of many points of view, instantaneous discussion and refinement, and is a concrete step toward the development of a specification.

**Quality Function Deployment**

*Quality function deployment (QFD)* is a quality management technique that translates the needs of the customer into technical requirements for software.

**Normal requirements.** The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

**Expected requirements.** These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

**Exciting requirements.** These features go beyond the customer's expectations and prove to be very satisfying when present. For example, word processing software is requested with standard features. The delivered product contains a number of page layout capabilities that are quite pleasing and unexpected.

**Use-Cases**

As requirements are gathered as part of informal meetings, FAST, or QFD, the software engineer (analyst) can create a set of scenarios that identify a thread of usage or the system to be constructed. The scenarios, often called *use-cases*, provide a description of how the system will be used. To create a use-case, the analyst must first identify the
different types of people (or devices) that use the system or product. These *actors* actually represent roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. It is important to note that an actor and a user are not the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role. Once actors have been identified, use-cases can be developed. The use-case describes the manner in which an actor interacts with the system.

- What main tasks or functions are performed by the actor?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?

**C. ANALYSIS PRINCIPLES**

Investigators have identified analysis problems and their causes and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view.

All analysis methods are related by a set of operational principles:

- The information domain of a problem must be represented and understood.
- The functions that the software is to perform must be defined.
- The behavior of the software (as a consequence of external events) must be represented.
- The models that depict information function and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
- The analysis process should move from essential information toward implementation detail.

The various analysis principles are

- **The Information Domain**
- **Modeling**
- **Partitioning**
- **Essential and Implementation Views**
KEYWORDS

Requirements analysis
Facilitated application specification techniques (FAST).
Quality function deployment (QFD)
use-case
requirements elicitation technique

OBJECTIVE TYPE QUESTIONS:

1. __________ is a software engineering task that bridges the gap between system level requirements engineering and software design.
   a).FAST  b).Requirement analysis  c) QFD  d).Use case

2. Independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification called ______________.
   a).FAST  b).Requirement analysis  c) QFD  d).Use case

3. ______________ is a quality management technique that translates the needs of the customer into technical requirements for software.
   a).FAST  b).Requirement analysis  c) QFD  d).Use case

4. The use-case describes the manner in which an actor interacts with the system.

5. The most commonly used requirements elicitation technique is to conduct a meeting or interview.
Assessment Questions on analysis concepts:

1. What is Analysis modeling?
2. Who does Analysis modeling?
3. Why Analysis modeling is important?
4. What are the steps in Analysis modeling?
5. What is the work product in Analysis modeling?
6. How do I ensure that I've done it right?

Answers to assessment Questions:

What is it? Analysis modeling uses a combination of text and diagrammatic forms to depict requirements for data, function, and behaviour in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

Who does it? A software engineer (sometimes called an analyst) builds the model using requirements elicited from the customer.

Why is it important? To validate software requirements, you need to examine them from a number of different points of view. Analysis modeling represents requirements in three "dimensions" thereby increasing the probability that errors will be found, that inconsistency will surface, and that omissions will be uncovered.

What are the steps? Data, functional, and behavioural requirements are modeled using a number of different diagrammatic formats. Data modeling defines data objects, attributes, and relationships. Functional modeling indicates how data are transformed within a system. Behavioural modeling depicts the impact of events. Once preliminary models are created, they are refined and analysed to assess their clarity, completeness, and consistency. A specification incorporating the model is created and then validated by both software engineers and customers/users.

What is the work product? Data object descriptions, entity relationship diagrams, data flow diagrams, state transition diagrams, process specifications, and control specifications are created as part of the analysis modeling activity.

How do I ensure that I've done it right? Analysis modeling work products must be reviewed for correctness, completeness, and consistency.
Chapter 2       Design Process and Concepts

Introduction
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. The flow of information during software design is illustrated in Figure below. Software requirements, manifested by the data, functional, and behavioral models, feed the design task. Using one of a number of design methods, the design task produces a data design, an architectural design, an interface design, and a component 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 detailed data content depicted in 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 architectural design representation the framework of a computer-based system.

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.

**A. THE DESIGN PROCESS**

McLaughlin suggests three characteristics that serve as a guide for the evaluation of a good design:

- 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 subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

To achieve the above 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, thereby facilitating implementation and testing.
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.

B. 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. It is important to note, however, that the design process is not simply a cookbook. 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.

   The design process should not suffer from "tunnel vision."
   The design should be traceable to the analysis model.
   The design should not reinvent the wheel.
   The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world.
   The design should exhibit uniformity and integration.
   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.
   Design is not coding, coding is not design.
   The design should be assessed for quality as it is being created, not after the fact.
   The design should be reviewed to minimize conceptual (semantic) errors.

C. DESIGN CONCEPTS
A set of fundamental software design concepts has evolved over the past four decades. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied.

Abstraction
Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, software is allocated as an element of a computer-based system. During software requirements analysis, the software solution
is stated in terms "that are familiar in the problem environment." 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*.

**Refinement**

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. 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.

**Modularity**

The concept of modularity in computer software has been espoused for almost five decades. Software architecture embodies modularity; that is, software is divided into
separately named and addressable components, often called *modules*, that are integrated to satisfy problem requirements. 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. Meyer defines 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.

**Software Architecture**

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. In a broader sense, however, *components* can be generalized to represent major system elements and their interactions.

**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.

**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, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

**Structural Partitioning**

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. Referring to Figure 13.4a, *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
D. 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.

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. Stated another way, 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. 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 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. Moderate levels of cohesion are relatively close to one another in the degree of module independence. 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. High cohesion is characterized by a module that performs one distinct procedural task.

**Coupling**

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.

**SUMMARY**

Design is the technical kernel of software engineering. During design, progressive refinements of data structure, architecture, interfaces, and procedural detail of software components are developed, reviewed, and documented. Design results in representations of software that can be assessed for quality. A number of fundamental software design principles and concepts have been proposed over the past four decades. Design principles guide the software engineer as the design process proceeds. Design concepts provide basic criteria for design quality.

**References and further reading:**

[Design-Principles](#)
KEYWORDS

Cohesion
Coupling
The design process
The data design
Control hierarchy
Refinement
A data abstraction
Horizontal partitioning
The architectural design
The interface design
The component-level design

OBJECTIVE TYPE QUESTIONS

1. ______ is a measure of interconnection among modules in a software structure.

2. ______ is a natural extension of the information hiding concept.

3. ______ defines separate branches of the modular hierarchy for each major program function.

4. ______, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control.

5. ______ is actually a process of elaboration.

6. ______ is a named collection of data that describes a data object.
7. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built.
   a) design       b) data       c) Architecture       d) Component level

8. The ______ design transforms the information domain model created during analysis into the data structures that will be required to implement the software.
   a) Coupling       b) data       c) Architecture       d) Horizontal partitioning

9. The ______ design defines the relationship between major structural elements of the software.
   a) Coupling       b) data       c) Architecture       d) Horizontal partitioning

10. The ______ design describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it.
    a) Interface       b) data       c) Architecture       d) Component level

11. The ______ design transforms structural elements of the software architecture into a procedural description of software components.
    a) Interface       b) data       c) Architecture       d) Component level

**Assessment Questions on design concepts:**

1. What are design concepts?
2. Who does design modeling?
3. Why a design concept is important?
4. What are the steps involved in design concepts?
5. What is the work product in design modeling?
6. How do I ensure that I've done it right?

**Answers to Assessment answers:**

**What is it?** 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.

Who does it? Software engineers design computer based systems, but the skills required at each level of design work are different. At the data and architectural level, design focuses on patterns as they apply to the application to be built. At the interface level, human ergonomics often dictate our design approach. At the component level, a "programming approach" leads us to effective data and procedural designs.

Why is it important? You wouldn't attempt to build a house without a blueprint, would you? You'd risk confusion, errors, a floor plan that didn't make sense, windows and doors in the wrong place . . . a mess. Computer software is considerably more complex than a house; hence, we need a blueprint— the design.

What are the steps? Design begins with the requirements model. We work to transform this model into four levels of design detail: the data structure, the system architecture, the interface representation, and the component level detail. During each design activity, we apply basic concepts and principles that lead to high quality.

What is the work product? Ultimately, a Design Specification is produced. The specification is composed of the design models that describe data, architecture, interfaces, and components. Each is a work product of the design process.

How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with the requirements and with one another.

Chapter 3

ARCHITECTURAL DESIGN

Objectives

To introduce architectural design and to discuss its importance
To explain the architectural design decisions that have to be made
To introduce three complementary architectural styles covering organization, decomposition and control
To discuss reference architectures are used to communicate and compare architectures
To explain why multiple models are required to document a software architecture
To discuss how domain-specific reference models may be used as a basis for product-lines and to compare software architectures.

**Topics covered**

Architectural design decisions
System organization
Decomposition styles
Control styles
Reference architectures

**Description:**

The design process for identifying the subsystems making up a system and the framework for sub-system control and communication is architectural design.

The output of this design process is a description of the software architecture.

**Architectural design**

An early stage of the system design process.
Represents the link between specification and design processes.
Often carried out in parallel with some specification activities.
It involves identifying major system components and their communications.

**Architectural models**

Different architectural models may be produced during the design process
Each model presents different perspectives on the architecture
Static structural model that shows the major system components
Dynamic process model that shows the process structure of the system
Interface model that defines sub-system interfaces
Relationships model such as a data-flow model
Architectural styles

The architectural model of a system may conform to a generic architectural model or style. An awareness of these styles can simplify the problem of defining system architectures. However, most large systems are heterogeneous and do not follow a single architectural style.

Data-centered architectures. 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. Client software accesses a central repository. 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. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client change.

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. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the working of its neighboring filters.
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. A number of sub styles exist within this category:

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 procedures call architectures. The components of main program subprogram architecture are distributed across multiple computers on a network.

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.

Layered architectures. A number 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. These architectural styles are only a small subset of those available to the software designer. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those
characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles might be designed and evaluated.

**SUMMARY**

Software architecture provides a holistic view of the system to be built. It depicts the structure and organization of software components, their properties, and the connections between them.

A number of different architectural styles and patterns are available to the software engineer. Each style describes a system category that encompasses a set of components that perform a function required by a system, a set of connectors that enable communication, coordination and cooperation among components, constraints that define how components can be integrated to form the system, and semantic models that enable a designer to understand the overall properties of a system.

Architectural design encompasses the initial set of design activities that lead to a complete design model of the software. In the chapters that follow, the design focus shifts to interfaces and components.

**FURTHER READINGS AND INFORMATION SOURCES:**

* [Architectural Design](#)
KEYWORDS
Data-flow architectures
Architectural design
Data-centered architectures
Call and return architectures

OBJECTIVE TYPE QUESTIONS

1. ______________ are applied when input data are to be transformed through a series of computational or manipulative components into output data.
   a).Data centered       b).Call and return     c).Data flow

2. ______________: 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.
   a).Data centered       b).Call and return     c).Data flow

3. ______________ architectures enables a software designer (system architect) to achieve a program structure
   a).Data centered       b).Call and return     c).Data flow

4. ______________ encompasses the initial set of design activities that lead to a complete design model of the software.
   a).Architectural design b). Data flow           c). Call and return

Assessment Questions in Architectural Design:

1. What is meant by architectural design?
2. Who does Architectural design?
3. Why Architectural design is important?
4. What are the steps in Architectural design?
5. What is the work product in Architectural design?
6. How do I ensure that I've done it right (Verification and Validation)?
**ANSWERS:**

**What is it?** Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

**Who does it?** Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style for the requirements derived during system engineering and software requirements analysis.

**Why is it important?** In the Quick Look for the last chapter, we asked: "You wouldn't attempt to build a house without a blueprint, would you?" You also wouldn't begin drawing blueprints by sketching the plumbing layout for the house. You'd need to look at the big picture—the house itself—before you worry about details. That's what architectural design does—it provides you with the big picture and ensures that you've got it right.

**What are the steps?** Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method.

**What is the work product?** An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.

**How do I ensure that I've done it right?** At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.
Chapter 4  
User Interface Design

OBJECTIVES
To suggest some general design principles for user interface design
To explain different interaction styles and their use
To explain when to use graphical and textual information presentation
To explain the principal activities in the user interface design process
To introduce usability attributes and approaches to system evaluation

DESCRIPTION
The blueprint for a house (its architectural design) is not complete without a representation of doors, windows, and utility connections for water, electricity, and telephone (not to mention cable TV). The "doors, windows, and utility connections" for computer software make up the interface design of a system. Interface design focuses on three areas of concern:
1. The design of interfaces between software components,
2. The design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities), and
3. The design of the interface between a human (i.e., the user) and the computer.

TOPICS COVERED
The Golden Rules
The user interface design process
Summary

A. THE GOLDEN RULES
Theo Mandel coins three "golden rules":
1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.
These golden rules actually form the basis for a set of user interface design principles that guide this important software design activity.

1. Place the User in Control

   Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
   Provide for flexible interaction.
   Allow user interaction to be interruptible and undoable.
   Streamline interaction as skill levels advance and allow the interaction to be customized.
   Hide technical internals from the casual user.
   Design for direct interaction with objects that appear on the screen.

2. Reduce the User's Memory Load

   The more a user has to remember, the more error-prone will be the interaction with the system. It is for this reason that a well-designed user interface does not tax the user's memory.

   Reduce demand on short-term memory.
   Establish meaningful defaults.
   Define shortcuts that are intuitive.
   The visual layout of the interface should be based on a real world metaphor.
   Disclose information in a progressive fashion.

3. Make the Interface Consistent

   Allow the user to put the current task into a meaningful context.
   Maintain consistency across a family of applications.
   If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.
B. THE USER INTERFACE DESIGN PROCESS

The design process for user interfaces is iterative and can be represented using a spiral model. Referring to Figure below, the user interface design process encompasses four distinct framework activities:

1. User, task, and environment analysis and modeling
2. Interface design
3. Interface construction
4. Interface validation

1. The initial analysis activity focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited.

2. Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral).

3. The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

4. The implementation activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.
5. Validation focuses on
   a. the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;
   b. the degree to which the interface is easy to use and easy to learn; and
   c. The users' acceptance of the interface as a useful tool in their work.

SUMMARY

The user interface is arguably the most important element of a computer-based system or product. If the interface is poorly designed, the user's ability to tap the computational power of an application may be severely hindered. In fact, a weak interface may cause an otherwise well-designed and solidly implemented application to fail.

Three important principles guide the design of effective user interfaces:
   Place the user in control,
   Reduce the user's memory load, and
   Make the interface consistent.

User interface design begins with the identification of user, task, and environmental requirements. Task analysis is a design activity that defines user tasks and actions using either an elaborative or object-oriented approach.
Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. A variety of implementation tools are used to build a prototype for evaluation by the user. The user interface is the window into the software. In many cases, the interface molds a user's perception of the quality of the system. If the "window" is smudged, wavy, or broken, the user may reject an otherwise powerful computer-based system.

ASSESSMENT QUESTIONS FOR THE USER INTERFACE DESIGN

1. What Is Meant By User Interface Design?
2. Who Does User Interface Design?
3. Why User Interface Design Is Important?
4. What Are The Steps In User Interface Design?
5. What Is The Work Product In User Interface Design?
6. How Do I Ensure That I've done It Right (Verification And Validation)?

ASSESSMENT ANSWERS

What is it? User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Who does it? A software engineer designs the user interface by applying an iterative process that draws on predefined design principles.

Why is it important? If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits or the functionality it offers. Because it molds a user's perception of the software, the interface has to be right.

What are the steps? User interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and
specification of major and minor menu items. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

**What is the work product?** User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion.

**How do I ensure that I've done it right?** The prototype is "test driven" by the users and feedback from the test drive is used for the next iterative modification of the prototype.

---

**Chapter 5**

**Real time Software Design**

**OBJECTIVES**

To explain the concept of a real-time system and why these systems are usually implemented as concurrent processes

To describe a design process for real-time systems

To explain the role of a real-time operating system

To introduce generic process architectures for monitoring and control and data acquisition systems

**TOPICS COVERED**

1. System design
2. Real-time executives
3. Data acquisition systems
4. Monitoring and control systems

**1. SYSTEM DESIGN**

System design develops the architectural detail required to build a system or product.

The system design process encompasses the following activities:

- Partition the analysis model into subsystems.
- Identify concurrency that is dictated by the problem.
- Allocate subsystems to processors and tasks.
Develop a design for the user interface.
Choose a basic strategy for implementing data management.
Identify global resources and the control mechanisms required to access them.
Design an appropriate control mechanism for the system, including task management.
Consider how boundary conditions should be handled.
Review and consider trade-offs.
Design both the hardware and the software associated with system.
Partition functions to either hardware or software.
Design decisions should be made on the basis on non-functional system requirements.
Hardware delivers better performance but potentially longer development and less scope for change.

2. REAL TIME EXECUTIVE

Introduction
Real-time executives are specialised operating systems which manage the processes in the RTS
Responsible for process management and resource (processor and memory) allocation
May be based on a standard RTE kernel which is used unchanged or modified for a particular application
Does not include facilities such as file management

Executive components
Real-time clock
  Provides information for process scheduling.
Interrupt handler
  Manages a periodic request for service.
Scheduler
  Chooses the next process to be run.
Resource manager
  Allocates memory and processor resources.
Despatcher
  Starts process execution.
Process priority

The processing of some types of stimuli must sometimes take priority

Interrupt level priority. Highest priority which is allocated to processes requiring a very fast response

Clock level priority. Allocated to periodic processes

Within these, further levels of priority may be assigned

Interrupt servicing

Control is transferred automatically to a pre-determined memory location

This location contains an instruction to jump to an interrupt service routine

Further interrupts are disabled, the interrupt serviced and control returned to the interrupted process

Interrupt service routines MUST be short, simple and fast Process switching

The scheduler chooses the next process to be executed by the processor. This depends on a scheduling strategy which may take the process priority into account

The resource manager allocates memory and a processor for the process to be executed
The dispatcher takes the process from ready list, loads it onto a processor and starts execution.

**The R-T system design process**
- Identify stimuli and associated responses.
- Define the timing constraints associated with each stimulus and response.
- Allocate system functions to concurrent processes.
- Design algorithms for stimulus processing and response generation.
- Design a scheduling system which ensures that processes will always be scheduled to meet their deadlines.

**Scheduling strategies**
- Non pre-emptive scheduling
  - Once a process has been scheduled for execution, it runs to completion or until it is blocked for some reason (e.g. waiting for I/O)
- Pre-emptive scheduling
  - The execution of an executing processes may be stopped if a higher priority process requires service
- Scheduling algorithms
  - Round-robin
  - Rate monotonic
  - Shortest deadline first

**Summary**
- Real-time system correctness depends not just on what the system does but also on how fast it reacts.
- A general RT system model involves associating processes with sensors and actuators.
- Real-time systems architectures are usually designed as a number of concurrent processes.
3. DATA ACQUISITION SYSTEMS

Data acquisition — Tools that acquire data to be used during testing.
Collect data from sensors for subsequent processing and analysis.
Data collection processes and processing processes may have different periods and deadlines.
Data collection may be faster than processing e.g. collecting information about an explosion.
Circular or ring buffers are a mechanism for smoothing speed differences.

5. MONITORING AND CONTROL SYSTEMS

Introduction
Important class of real-time systems
Continuously check sensors and take actions depending on sensor values
Monitoring systems examine sensors and report their results
Control systems take sensor values and control hardware actuators.
Example:

Define a monitoring and control system for Burglar alarm system.

Solution:

A system is required to monitor sensors on doors and windows to detect the presence of intruders in a building.
When a sensor indicates a break-in, the system switches on lights around the area and calls police automatically.
The system should include provision for operation without a mains power supply.

Sensors

Movement detectors, window sensors, door sensors.

50 window sensors, 30 door sensors and 200 movement detectors

Voltage drop sensor

Actions

When an intruder is detected, police are called automatically.
Lights are switched on in rooms with active sensors.
An audible alarm is switched on.
The system switches automatically to backup power when a voltage drop is detected.