UNIT-III

DESIGN ENGINEERING

Design engineering encompasses the **set of principals, concepts, and practices** that lead to the development of a high- quality system or product.

- ✓ Design principles establish an overriding philosophy that guides the designer in the work that is performed.
- ✓ Design concepts must be understood before the mechanics of design practice are applied and
- ✓ Design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

What is design:

Design is what virtually every engineer wants to do. It is the place where creativity rules – customer's requirements, business needs, and technical considerations all come together in the formulation of a product or a system. Design creates a representation or model of the software, but unlike the analysis model, the design model provides detail about software data structures, architecture, interfaces, and components that are necessary to implement the system.

Why is it important:

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

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, a designer must practice diversification and then convergence. Another **goal** of software design is to derive an architectural rendering of a system. The rendering serves as a framework from which more detailed design activities are conducted.

1) DESIGN PROCESS AND DESIGN QUALITY:

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.

Goals of design:

McGlaughlin 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.

Quality guidelines:

In order to evaluate the quality of a design representation we must establish technical criteria for good design. These are the following guidelines:

1. A design should exhibit an architecture that
   - **a.** has been created using recognizable architectural styles or patterns
   - **b.** is composed of components that exhibit good design characteristics and
   - **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 or subsystems.
3. A design should contain distinct representation of data, architecture, interfaces and components.
4. A design should lead to data structures that are appropriate for the classes 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 interface that reduce the complexity of connections between components 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.
8. A design should be represented using a notation that effectively communication its meaning.

These design guidelines are not achieved by chance. Design engineering encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.

Quality attributes:
The FURPS quality attributes represent a target for all software design:

- ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ***Usability*** is assessed by considering human factors, overall aesthetics, consistency and documentation.
- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, and the mean – time –to- failure (MTTF), the ability to recover from failure, and the predictability of the program.
- ***Performance*** is measured by processing speed, response time, resource consumption, throughput, and efficiency
- ***Supportability c***ombines the ability to extend the program (extensibility), adaptability, serviceability- these three attributes represent a more common term maintainability

Not every software quality attribute is weighted equally as the software design is developed.

One application may stress functionality with a special emphasis on security.

Another may demand performance with particular emphasis on processing speed.

A third might focus on reliability.

2) DESIGN CONCEPTS:

M.A Jackson once said:"The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it right." Fundamental software design concepts provide the necessary framework for "getting it right."

I. Abstraction: Many levels of abstraction are there.

- ✓ 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 detailed description of the solution is provided.

A ***procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. The name of procedural abstraction implies these functions, but specific details are suppressed.
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 operation, 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.**

II. Architecture:

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

One **goal** of software design is to derive an architectural rendering of a system. The rendering serves as a framework from which more detailed design activities are conducted.

The architectural design can be represented using one or more of a number of different models. ***Structured 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 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 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.

III. Patterns:

Brad Appleton defines a ***design pattern*** in the following manner: "a pattern is a named nugget of inside which conveys that essence of a proven solution to a recurring problem within a certain context amidst competing concerns." Stated in another way, a design pattern describes a design structure that solves a particular design within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine

1) Whether the pattern is capable to the current work,
2) Whether the pattern can be reused,
3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

IV. Modularity:

Software architecture and design patterns embody ***modularity*;** software is divided into separately named and addressable components, sometimes called ***modules*** that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

The "divide and conquer" strategy- it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to modularity and software. If we subdivide software indefinitely, the effort required to develop it will become negligibly small. The effort to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort associated with integrating the modules also grow.

Under modularity or over modularity should be avoided. We modularize a design so that development can be more easily planned; software increment can be defined and delivered; chamges can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

V. Information Hiding:

The principle of *information hiding* suggests that modules be "characterized by design decision that hides from all others."

Modules should be specified and designed so that information 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 entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and local data structure used by module.

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

VI. 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.

Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified. Independent sign or code modifications are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information hiding.

A cohesion module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should do just one thing.

Coupling is an indication 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. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagates throughout a system.

VII. Refinement:

Stepwise refinement is a top- down design strategy originally proposed by Niklaus wirth. A program is development by successively refining levels of procedural detail. A hierarchy is development by decomposing a macroscopic statement of function in a step wise fashion until programming language statements are reached.

Refinement is actually a process of elaboration. We begin with a statement of function 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 data. 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.

VIII. Refactoring :

Refactoring is a reorganization technique that simplifies the design of a component without changing its function or behavior. Fowler defines refactoring in the following manner: "refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. The designer may decide that the component should be refactored into 3 separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

IX. Design classes:

The software team must define a set of design classes that

1. Refine the analysis classes by providing design detail that will enable the classes to be implemented, and

2. Create a new set of design classes that implement a software infrastructure to support the design solution.

Five different types of design classes, each representing a different layer of the design architecture are suggested.

- **User interface classes:** define all abstractions that are necessary for human computer interaction. In many cases, HCL occurs within the context of a *metaphor* and the design classes for the interface may be visual representations of the elements of the metaphor.
- **Business domain classes:** are often refinements of the analysis classes defined earlier. The classes identify the attributes and services that are required to implement some element of the business domain.
- **Process classes** implement lower – level business abstractions required to fully manage the business domain classes.
- **Persistent classes** represent data stores that will persist beyond the execution of the software.
- **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the design model evolves, the software team must develop a complete set of attributes and operations for each design class. The level of abstraction is reduced as each analysis class is transformed into a design representation. Each design class be reviewed to ensure that it is "well-formed." They define **four characteristics of a well- formed design class.**

Complete and sufficient: A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

Primitiveness: Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.

High cohesion: A cohesive design class has a small, focused set of responsibilities and single- mindedly applies attributes and methods to implement those responsibilities.

Low coupling: Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of classes in other subsystems. This restriction, called the *law of Demeter*, suggests that a method should only sent messages to methods in neighboring classes.

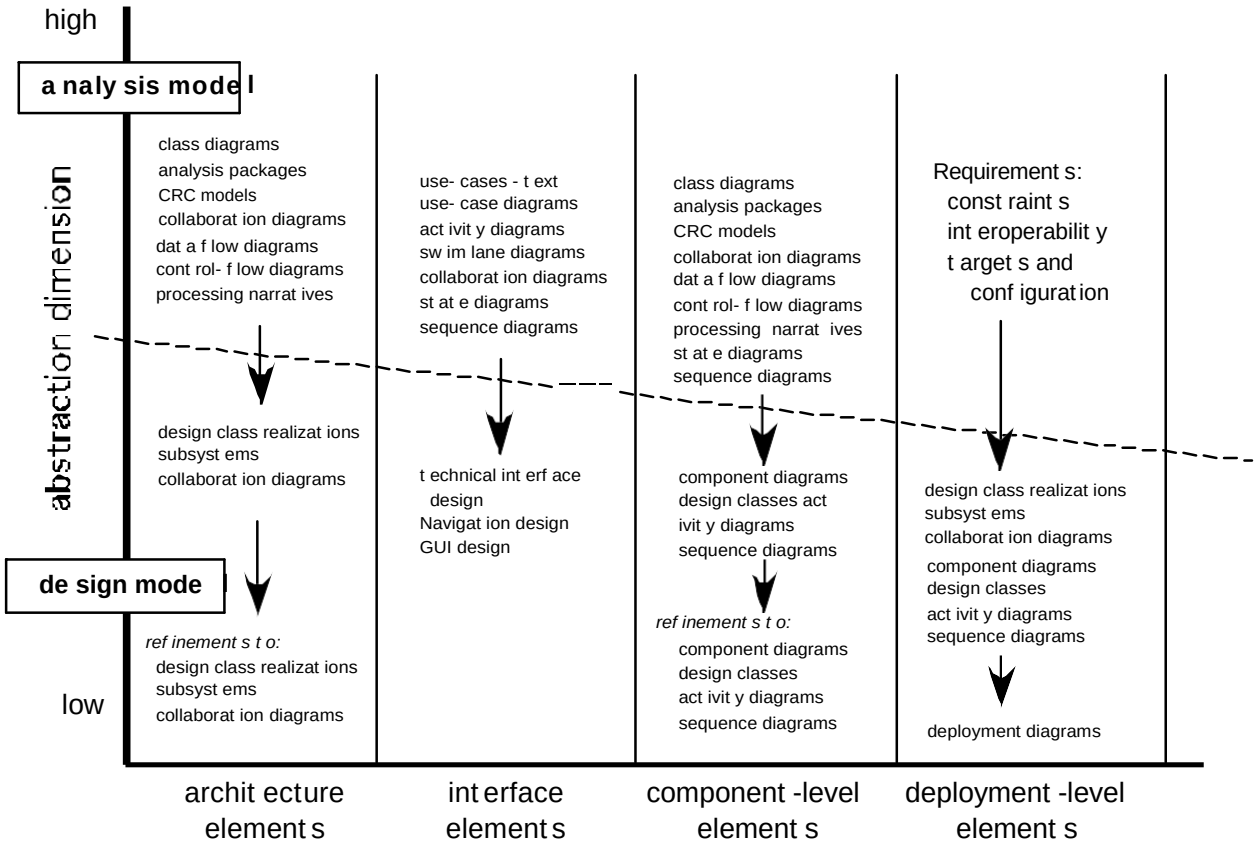
THE DESIGN MODEL:

- The design model can be viewed into different dimensions.
- The process dimension indicates the evolution of the design model as design tasks are executed as a part of the software process.

The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

The elements of the design model use many of the same UML diagrams that were used in the analysis model. The difference is that these diagrams are refined and elaborated as a path of design; more implementation- specific detail is provided, and architectural structure and style, components that reside within the architecture, and the interface between the components and with the outside world are all emphasized.

It is important to mention however, that model elements noted along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model us usually delayed until the design has been fully developed.



i. Data design elements:

Data design sometimes referred to as data architecting creates a model of data and/or information that is represented at a high level of abstraction. This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.
The structure of data has always been an important part of software design.

- ✓ At the **program component level**, the design of data structures and the associated algorithms required to manipulate them is essential to the criterion of high-quality applications.
- ✓ At **the application level**, the translation of a data model into a database is pivotal to achieving the business objectives of a system.
- ✓ At the **business leve**l, 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.

ii. Architectural design elements:

The *architectural design* for software is the equivalent to the floor plan of a house.
The architectural model is derived from three sources.

1) Information about the application domain for the software to be built.
2) Specific analysis model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
3) The availability of architectural patterns

iii. Interface design elements:

The *interface design* for software is the equivalent to a set of detailed drawings for the doors, windows, and external utilities of a house.

The interface design elements for software tell how information flows into and out of the system and how it is communicated among the components defined as part of the architecture. There are 3 important elements of interface design:

1) The user interface(UI);
2) External interfaces to other systems, devices, networks, or other produces or consumers of information; and
3) Internal interfaces between various design components.

These interface design elements allow the software to communicated externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design is a major software engineering action.

The design of a UI incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. The design of external interfaces should incorporate error checking and appropriated security features.

UML defines an *interface* in the following manner:"an interface is a specifier for the externally-visible operations of a class, component, or other classifier without specification of internal structure."

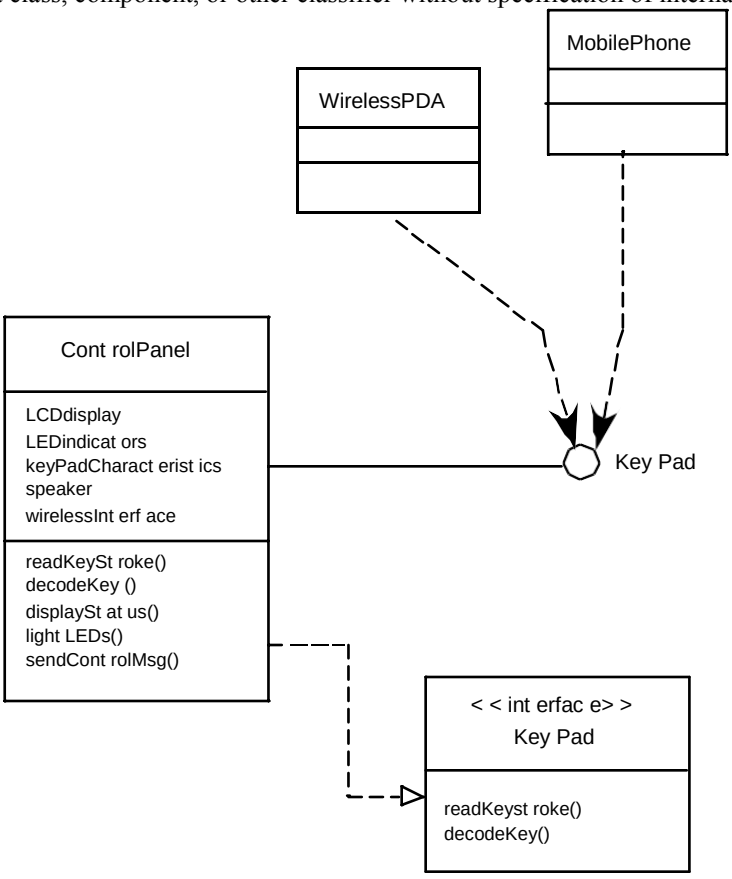
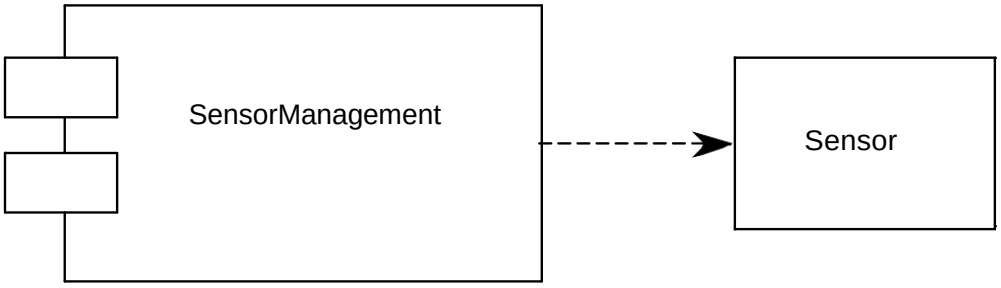


Figure : UML interface representation for Control Pane l

iv. **Component- level design elements:** The component-level design for software is equivalent to a set of detailed drawings.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations.



v. **Deployment-level design elements:** Deployment-level design elements indicated how software functionality and subsystems will be allocated within the physical computing environment that will support the software

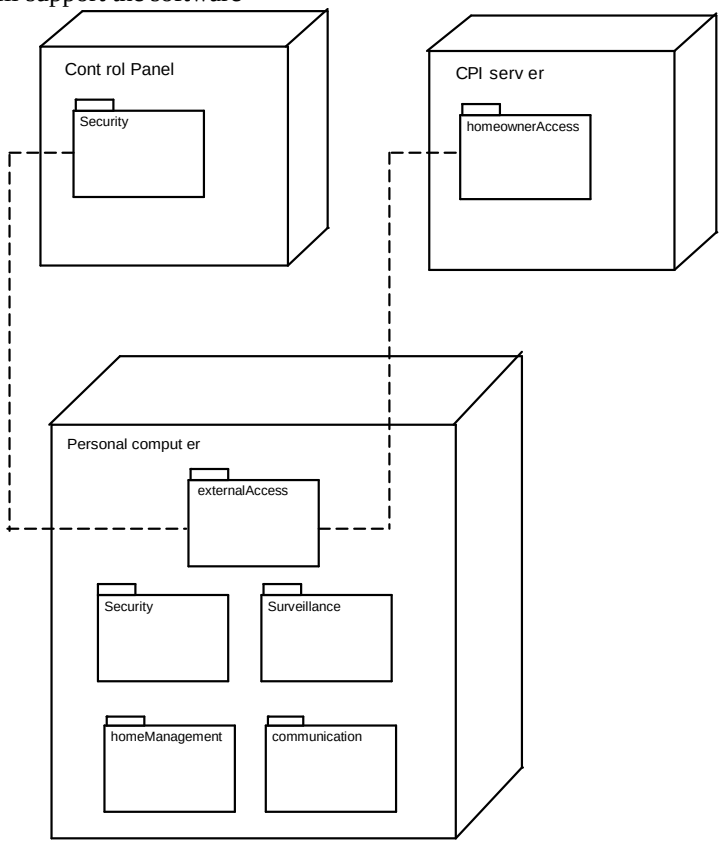


Figure 9 . 8 UML deploy m ent diagram for *SafeHom e*

ARCHITECTURAL DESIGN

1) SOFTWARE ARCHITECTURE:

What Is Architecture?

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.

The architecture 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.

The design of software architecture considers two levels of the design pyramid
- data design
- architectural design.

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

Why Is Architecture Important?
Bass and his colleagues [BAS98] identify three key reasons that software architecture is important:
- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- 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.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

2) DATA DESIGN:
The data design activity translates data objects as part of the analysis model into data structures at the software component level and, when necessary, a database architecture at the application level.

➢ 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.
➢ At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.
➢ 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.

2.1) Data design at the Architectural Level:
The challenge for a business has been to extract useful information from this data environment, particularly when the information desired is cross functional.

To solve this challenge, the business IT community has developed ***data mining*** techniques, also called ***knowledge discovery in databases*** (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information. An alternative solution, called a ***data warehouse***, adds an additional layer to the data architecture. a data warehouse is a large, independent database that encompasses some, but not all, of the data that are stored in databases that serve the set of applications required by a business.

2.2) 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. The following set of 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.

3) ARCHITECTURAL STYLES AND PATTERNS:

The builder has used an ***architectural style*** as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod).

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, coordinations 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.

An ***architectural pattern,*** like an architectural style, imposes a transformation the design of architecture. However, a pattern differs from a style in a number of fundamental ways:

(1) The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety.

(2) A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level.

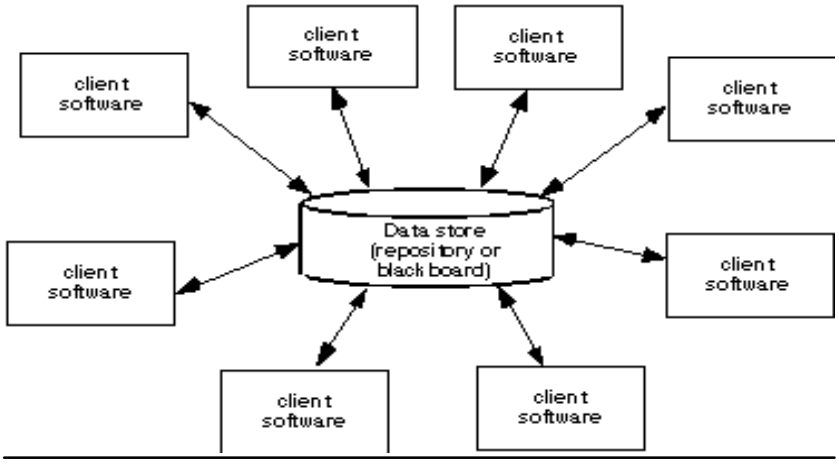
(3) Architectural patterns tend to address specific behavioral issues within the context of the architectural.

3.1) A Brief Taxonomy of Styles and Patterns

Data-centered architectures:

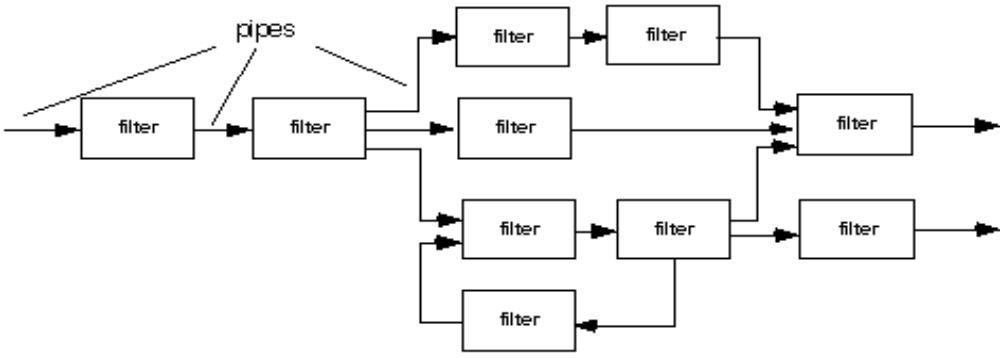
A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. A variation on this approach transforms the repository into a "blackboard" that sends notification to client software when data of interest to the client changes

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 (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism

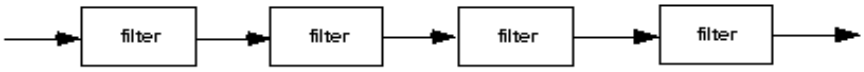


Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A ***pipe and filter pattern*** has a set of components, called ***filters***, connected by pipes that transmit data from one component 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 of a specified form.

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.



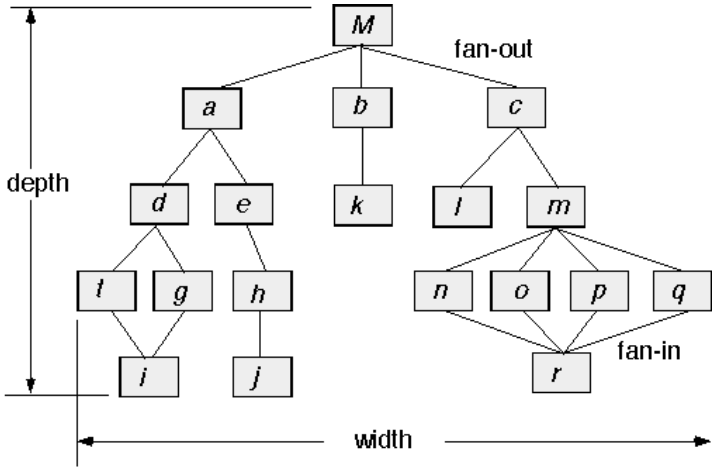
(a) pipes and filters



(b) batch sequential

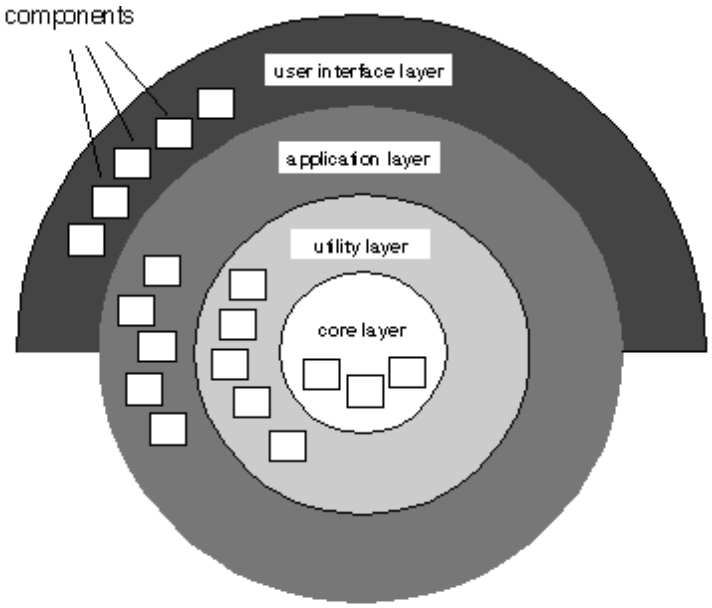
Call and return architectures. This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. A number of substyles [BAS98] exist within this category:

- ***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. Figure 13.3 illustrates an architecture of this type.
- ***Remote procedure call architectures***. The components of a 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. The basic structure of a layered architecture is illustrated in Figure 14.3. 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.



3.2) Architectural Patterns:

An ***architectural pattern,*** like an architectural style, imposes a transformation the design of architecture. However, a pattern differs from a style in a number of fundamental ways:

(1) The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety.

(2) A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level.

(3) Architectural patterns tend to address specific behavioral issues within the context of the architectural.

The architectural patterns for software define a specific approach for handling some behavioral characteristics of the system

Concurrency—applications must handle multiple tasks in a manner that simulates parallelism

- *operating system process management* pattern
- *task scheduler* pattern

Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:

- a ***database management system*** pattern that applies the storage and retrieval capability of a DBMS to the application architecture
- an ***application level persistence*** pattern that builds persistence features into the application architecture

Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment

- A ***broker*** acts as a 'middle-man' between the client component and a server component**.**

Organization and Refinement:
The design process often leaves a software engineer with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions provide insight into the architectural style that has been derived:

Control.

- ✓ How is control managed within the architecture?
- ✓ Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?
- ✓ How do components transfer control within the system?
- ✓ How is control shared among components?

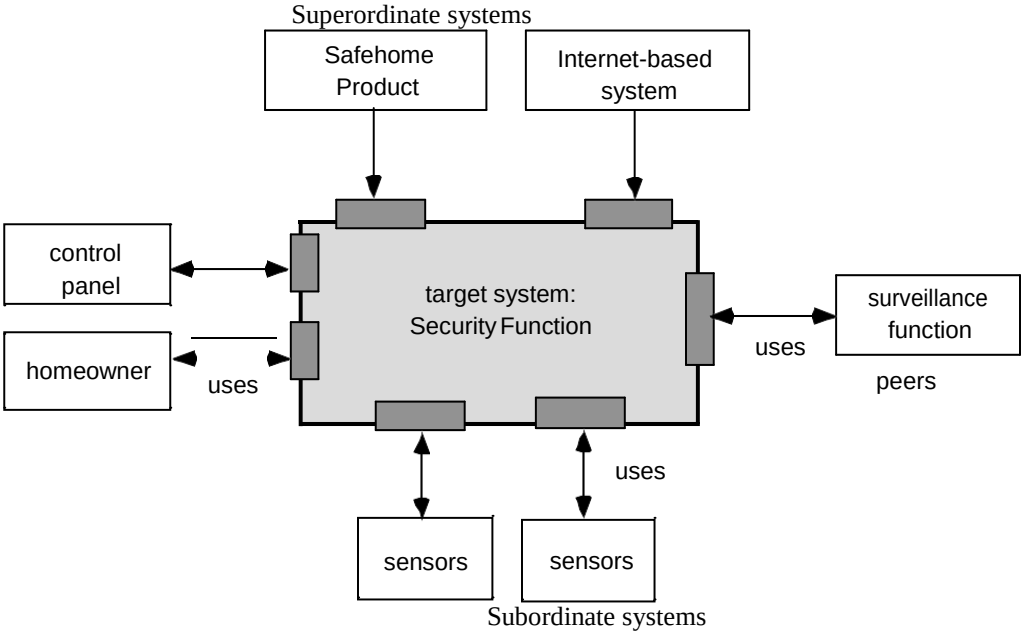
Data.

- ✓ How are data communicated between components?
- ✓ Is the flow of data continuous, or are data objects passed to the system sporadically?
- ✓ What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)?
- ✓ Do data components (e.g., a blackboard or repository) exist, and if so, what is their role?
- ✓ How do functional components interact with data components?
- ✓ Are data components *passive* or *active* (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

4) ARCHITECTURAL DESIGN:

I Representing the System in Context:

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in the figure



Superordinate systems – those systems that use the target system as part of some higher level processing scheme.

Subordinate systems - those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

Peer-level systems - those systems that interact on a peer-to-peer basis

Actors -those entities that interact with the target system by producing or consuming information that is necessary for requisite processing

II Defining Archetypes:

An archetype is a class or pattern that represents a core abstraction that is critical to the design of architecture for the target system. In general, a relative small set of archetypes is required to design even relatively complex systems.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the analysis model. In safe home security function, the following are the archetypes:

- **Node:** Represent a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors, and (2) a variety of alarm indicators.
- **Detector:** An abstraction that encompasses all sensing equipment that feeds information into the target system
- **Indicator:** An abstraction that represents all mechanisms for indication that an alarm condition is occurring.
- **Controller:** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

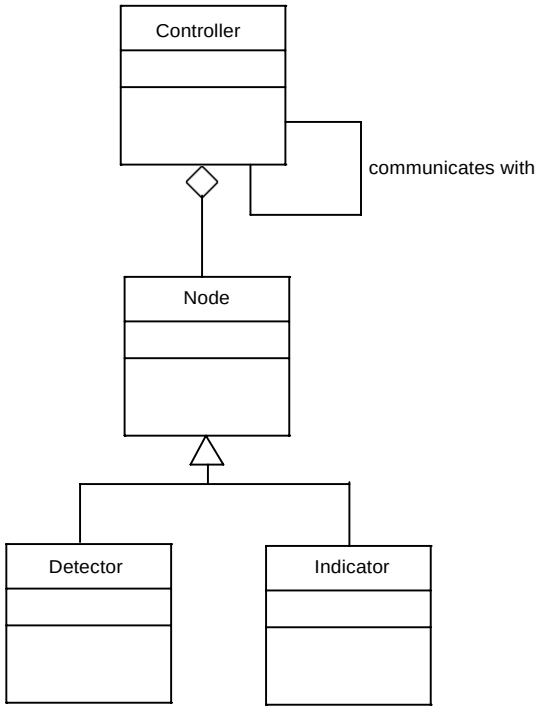


Figure 10.7 UML relat ionships for SafeHome security function archetypes (adapted f rom [BOS00])

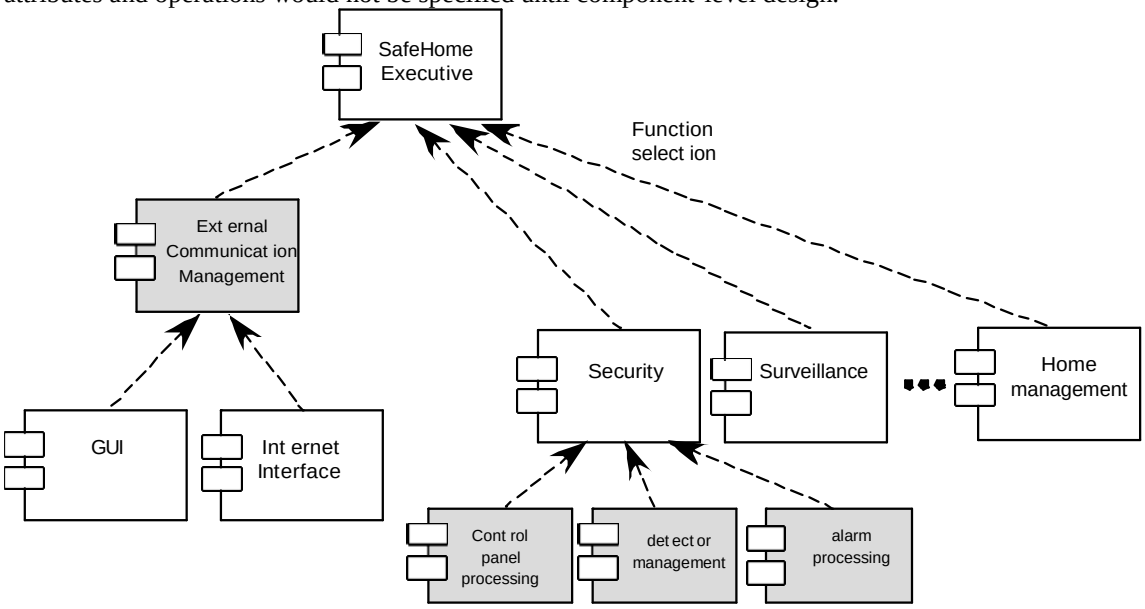
III Refining the Architecture into Components:

As the architecture is refined into components, the structure of the system begins to emerge. The architectural designer begins with the classes that were described as part of the analysis model. These analysis classes represent entities within the application domain that must be addressed within the software

architecture. Hence, the application domain is one source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application domain.
For eg: memory management components, communication components database components, and task management components are often integrated into the software architecture.
In the *safeHome* security function example, we might define the set of top-level components that address the following functionality:

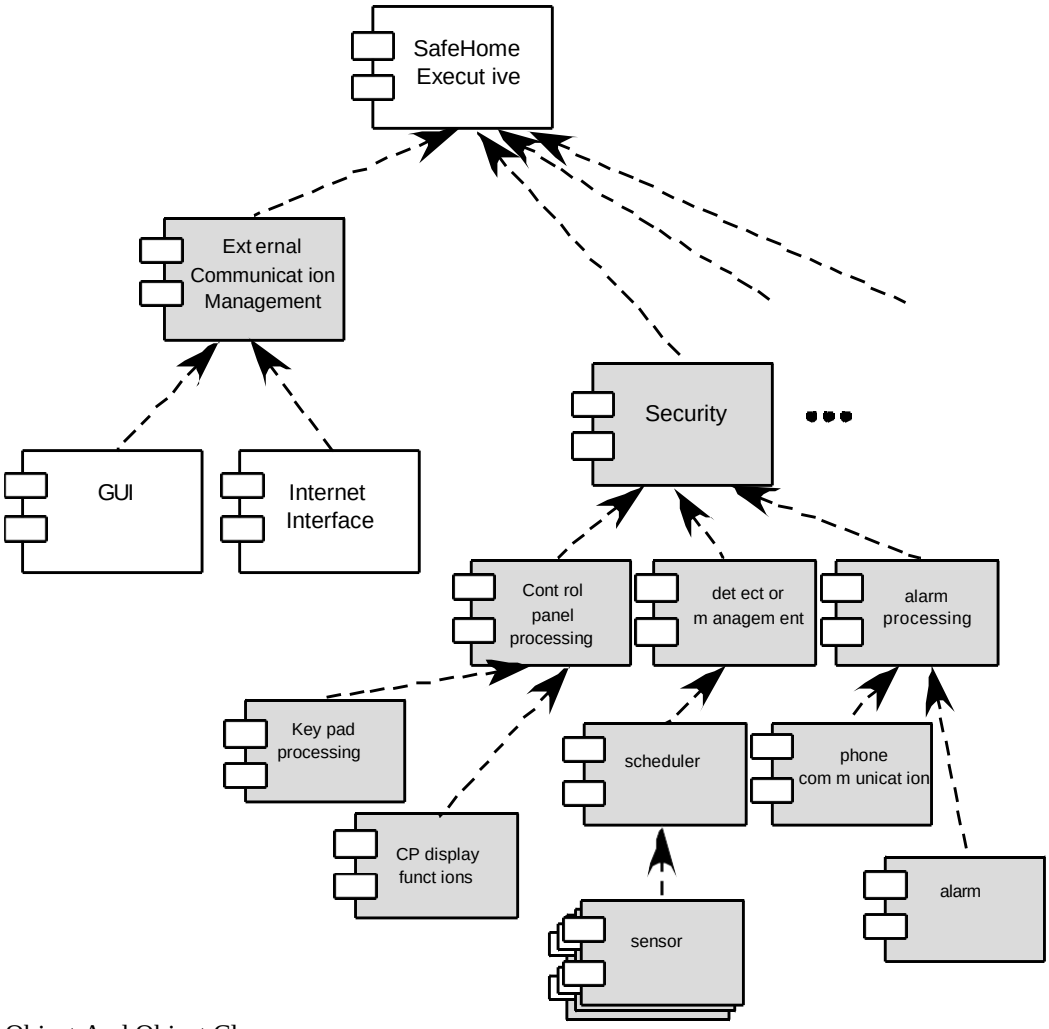
- *External communication management*- coordinates communication of the security function with external entities
- *Control panel processing*- manages all control panel functionality.
- *Detector management*- coordinates access to all detectors attached to the system.
- *Alarm processing*- verifies and acts on all alarm conditions.

Design classes would be defined for each. It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design.



Component Structure

IV **Describing Instantiations of the System:** An actual instantiation of the architecture means the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.



Object And Object Classes

- Object : An object is an entity that has a state and a defined set of operations that operate on that state.
- An obect class defination is both a type specification and a template for creating obects.
- It includes declaration of all the attributes and operations that are associated with object of that class.

Object Oriented Design Process

- There are five stages of object oriented design process

1)Understand and define the context and the modes of use of the system.

2)Design the system architecture

3)Identify the principle objects in the system.

4)Develop a design models

5)Specify the object interfaces

Systems context and modes of use

- It specify the context of the system.it also specify the relationships between the software that is being designed and its external environment.
- If the system context is a static model it describe the other system in that environment.
- If the system context is a dynamic model then it describe how the system actually interact with the environment.

System Architecture

- Once the interaction between the software system that being designed and the system environment have been defined
- We can use the above information as basis for designing the System Architecture.

Object Identification

- This process is actually concerned with identifying the object classes.
- We can identify the object classes by the following

1)Use a grammatical analysis
2)Use a tangible entities
3)Use a behaviourial approach
4)Use a scenario based approach

Design model

- Design models are the bridge between the requirements and implementation.
- There are two type of design models

1)Static model describe the relationship between the objects.
2)Dynamic model describe the interaction between the objects

- Object Interface SpecificationIt is concerned with specifying the details of the interfaces to an objects.
- Design evolution

The main advantage OOD approach is to simplify the problem of making changes to the design.
Changing the internal details of an obect is unlikely to effect any other system object.

USER INTERFACE DESIGN

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.

What is User Interface Design?

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.

Why is User Interface Design 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.

1.1 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.

Place the User in Control

Mandel [MAN97] defines a number of design principles that allow the user to maintain control:

- **Define interaction modes in a way that does not force a user into unnecessary or undesired actions**. Word processor – spell checking – move to edit and back; enter and exit with little or no effort
- **Provide for flexible interaction**. Several modes of interaction – keyboard, mouse, digitizer pen or voice recognition, but not every action is amenable to every interaction need. Difficult to draw a circle using keyboard commands.
- **Allow user interaction to be interruptible and undoable**. User stop and do something and then resume where left off. Be able to undo any action.
- **Streamline interaction as skill levels advance and allow the interaction to be customized**. Perform same actions repeatedly; have macro mechanism so user can customize interface.
- **Hide technical internals from the casual user**. Never required to use OS commands; file management functions or other arcane computing technology.
- **Design for direct interaction with objects that appear on the screen.** User has feel of control when interact directly with objects; stretch an object.

Reduce the User's Memory Load:

- ✓ The more a user has to remember, the more error-prone interaction with the system will be.
- ✓ Good interface design does not tax the user's memory
- ✓ System should remember pertinent details and assist the user with interaction scenario that assists user recall.

Mandel defines design principles that enable an interface to reduce the user's memory load:

- **Reduce demand on short-term memory**. Complex tasks can put a significant burden on short term memory. System designed to reduce the requirement to remember past actions and results; visual cues to recognize past actions, rather than recall them.
- **Establish meaningful defaults**. Initial defaults for average user; but specify individual preferences with a reset option.
- **Define shortcuts that are intuitive**. Use mnemonics like Alt-P.
- **The visual layout of the interface should be based on a real world metaphor**. Bill payment – check book and check register metaphor to guide a user through the bill paying process; user has less to memorize
- **Disclose information in a progressive fashion**. Organize hierarchically. High level of abstraction and then elaborate. Word underlining function – number of functions, but not all listed. User picks underlining then all options presented

Make the Interface Consistent

Interface present and acquire information in a consistent fashion.

1. All visual information is organized to a design standard for all screen displays
2. Input mechanisms are constrained to limited set used consistently throughout the application
3. Mechanisms for navigation from task to task are consistently defined and implemented

Mandel [MAN97] defines a set of design principles that help make the interface consistent:

- **Allow the user to put the current task into a meaningful context.** Because of many screens and heavy interaction, it is important to provide indicators – window tiles, graphical icons, consistent color coding so that the user knows the context of the work at hand; where came from and alternatives of where to go.
- **Maintain consistency across a family of applications**. For applications or products implementation should use the same design rules so that consistency is maintained for all interaction
- **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.** Unless a compelling reason presents itself don't change interactive sequences that have become de facto standards. (alt-S to scaling)

1.2 USER INTERFACE DESIGN

1.2.1 Interface Design Models

Four different models come into play when a user interface is to be designed.

- The software engineer creates a *design model,*
- a human engineer (or the software engineer) establishes a *user model,*
- the end-user develops a mental image that is often called the *user's model* or the *system perception,* and
- the implementers of the system create a *implementation model.*

The role of interface designer is to reconcile these differences and derive a consistent representation of the interface.

User Model: The user model establishes the profile of end-users of the system. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, sex, physical abilities, education, cultural or ethnic background, motivation, goals and personality" [SHN90]. In addition, users can be categorized as

- ✓ Novices.
- ✓ Knowledgeable, intermittent users.
- ✓ Knowledgeable, frequent users.

Design Model: A design model of the entire system incorporates data, architectural, interface and procedural representations of the software.

Mental Model: The user's mental model (system perception) is the image of the system that end-users carry in their heads.

Implementation Model: The implementation model combines the outward manifestation of the computer-based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describe system syntax and semantics.

These models enable the interface designer to satisfy a key element of the most important principle of user interface design: **"Know the user, know the tasks."**

1.2.2 The User Interface Design Process: (steps in interface design)

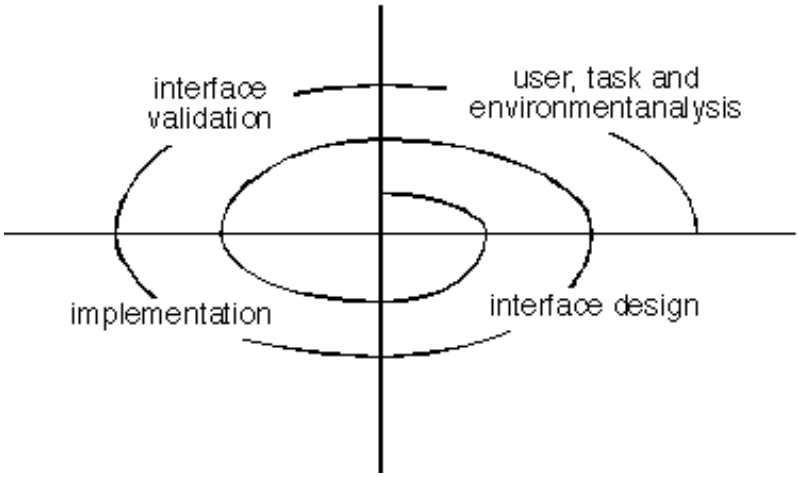
The user interface design process encompasses four distinct framework activities :

1. User, task, and environment analysis and modeling

2. Interface design

3. Interface construction

4. Interface validation



User Interface Design Process

(1) User Task and Environmental Analysis:

The interface 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. In essence, the software engineer attempts to understand the system perception (Section 15.2.1) for each class of users.
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

The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to theinterface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences.

(2) Interface Design:
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.

(3) Interface Construction(implementation)
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 (Section 15.5) may be used to complete the construction of the interface.

(4) Interface Validation:
Validation focuses on
(1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;

(2) the degree to which the interface is easy to use and easy to learn; and

(3) the users' acceptance of the interface as a useful tool in their work.

12.3 INTERFACE ANALYUSIS
A Key tenet of all software engineering process models is this: you better understand the problem before you attempt to design a solution. In the case of user interface design, understanding the problem means understanding (1) The people who will interact with the system through the interface; (2) the tasks that tend-users must perform to do their work, (3) the content that is presented as part of the inter face, an (4) the environment in which these tasks will be conducted. In the sections that follow, we examine each of these elements of interface analysis with the intent of establishing a solid foundation for the design tasks that follow.

12.3.1 User analysis
Earlier we noted that each user has a mental image or system perception of the software that may be different from the mental image developed by other users.

User Interviews. The most direct approach, interviews involve representatives from the software team who meet with end-users to better understand their needs, motivations work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.

Sales input. Sales people meet with customers an users on regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

Marketing input. Market analysis can be invaluable in definition of market segments while providing an understanding of how each segment might use the software in subtly different ways.

Support input. Support staff talk with users on a daily basis, making them the most likely soured of information on what works an what doesn't, what users like and what they dislike, what features generate questions, and what features are easy to use.

The following set of questions (adapted form (HAC98)) will help the interface designer better understand the users of a system:

- Are user trained professionals, technicians, clerical or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they ecpressed a desire of classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours, or do they work until the job is done.
- Is the software to be an integral part of the work users do, or will it be used only occasionally?

- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter the is addressed by the system?
- Do users want to know about the technology that sits behind the interface?

The answers to these an similar questions will allow the designer to understand who the end-users are, what is likely to motivate and please them, how they can be grouped into different user classes or profiled, what their mental models of the system are, and how the user interface must be characterized to meet their needs.

12.3.2 Task Analysis and Modeling

The goal of talk analysis is to answer the following questions:

- What work will the user perform in specific circumstances?
- What specific problem domain objects will the user manipulate as work is performed?
- What is the sequence of work tasks-the workflow?
- What is the hierarchy of tasks?

To answer these questions, the software engineer must draw upon analysis techniques discussed in Chapters 7 and 8, but in this instance, these techniques are applied to the user interface.

In earlier chapter we noted that the use-case describe the manner in which an actor (in the context of user interface design, an actor is always a person) interacts with a system.

The use-case provides a basic description of one important work task for the computer-aided design system. From, it, the software engineer can extract tasks, objects, and the overall flow of the interaction.

Task elaboration. Task analysis of interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate. To understand the tasks that must be performed to accomplish the goal of the activity, a human engineer must understand the tasks that humans currently perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface. Alternatively, the human engineer can study an existing specification for computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers. By observing an interior designer at work, the engineer notices that interior design comprises a number of major activities: further layout (note the use-case discussed earlier), fabric and material selection, wall and window coverings selection, presentation (to the customer), costing, and shopping. Each of these major tasks can be elaborated into subtasks. For example, using information contained in the use-case, furniture layout can be refined into the following tasks: (1) draw a floor plan based on room dimensions; (2) place windows and doors at appropriate locations;(3a) use furniture templates to draw scaled accents on floor plan(4) move furniture outlines;(6) draw dimensions to show location;(7) draw perspective rendering view for customer. A similar approach could be used for each of the other major tasks.

Object elaboration. The software engineer extracts the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide the designer with a list of operations. For example, the furniture template might translate into a class called **Furniture** with attributes that might include **size, shape, location** and others. The interior designer would select the object from the **Furniture** class, move it to a position on the floor plan (another object in this context), draw the furniture outline, and so forth. He tasks select, move, and draw are operations. The user interface analysis model would not provide a literal implementation for each of these operation for each of these operations. How ever, as the design is elaborated, the details of each operation are defined.

Workflow analysis. When a number of different users, each playing different roles, makes uses of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply workflow analysis. This technique allows a software engineer to understand how a work process is completed when several people are involved.

The flow of events (shown in the figure) enable the interface designer to recognize three day interface characteristics.

1. Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different form the one defined for pharmacists or physicians.

2. The interface design for pharmacists and physicians must accommodate access to and display of information form secondary information sources(e.g., access to inventory of the pharmacist and access to information about alternative medications for the physician)
3. Many of the activities noted in the swimlane diagram can be further elaborated using talk analysis and /or object elaboration(e.g., fills prescription could imply a mail-order deliver, a visit to a pharmacy, or a visit to a special drug distribution center.

Hierarchical representation. As the interface is analyzed, a process of elaboration occurs. Once workflow has been established, a task hierarchy can e defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the user task requests that a prescription be refilled. The following task hierarchy is developed:
Request that a prescription be refilled

- Provide identifying information
- Specify name
- Specify userid
- Specify PIN and password
- Specify prescription number
- Specify date refill is required

To complete the request that a prescription be refilled tasks, three subtasks are defined. One of these subtasks, provide indentifying information, is further elaborated in three additional sub-subtasks.

12.3.3 Analysis of Display Content

System response time is measured from the point at which the user performs some control action(e.g., hits the return key or clicks a mouse)until the software responds with the desired output or action.

System response time has two important characteristics: length and variability. If system response is is too long, user frustration and stress is the inevitable result. Variability refers to the deviation form average response time, and, in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds. When variability is significant, the user is always off balance, always wondering whether something "defferent"has occurred behind the scenes.

Help facilities. Modern software provides on-line help facilities that enable a user to get a question answered or resolve a problem without leaving the interface. A number of design issues must be addressed when a help facility is considered:

- Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.
- How will the user request help? Options include a help menu, a special function day, or a HELP command.
- How will help be represented? Options include a separate window, a reference to a printed document, or a one-or two-line suggestion produced in a fixed screen location.
- How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.
- How will help information be structured? Options include a "flat" structure in which all information is accessed through a keyword, a layered hierarchy or information that provides increasing detail as the user proceeds into the structure, or the user of hypertext.

In general, every error message or warning produced by an interactive system should have the following characteristics:

- The message should describe the problem in language the user can understand.
- The message should provide constructive advice for recovering form the error.
- The message should indicate any negative consequences of the error(e.g., potentially corrupted data files)so that the user can check to ensure that they have not occurred.
- The message should be nonjudgmental. That is, the wording should never place blame on the user.

But an-effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

A number of design issues arise when typed commands or menu labels are provided as mode of interaction:

- Will every menu option have a corresponding command?

- What form will commands take? Options include a control sequence (e.g., alt-p), function keys, or a typed word.
- How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?

Application accessibility .Accessibility for users and software engineers) who may be physically challenged is an imperative for moral, legal, and business reasons. A variety of accessibility guidelines many designed for Web applications but often applicable to all types of software-provide detailed suggestions for designing interfaces that achieve vary8ing levels of accessibility. Others provide specific guidelines or "assistive technology" that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

Internationalization. The challenge should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. Localization features enable the interface to be customized for a specific market.

A variety of internationalization guidelines are available to software engineers. These guidelines address broad design issues and discrete implementation issues. The Unicode standard has been developed to address the daunting challenge of managing dozens of natural languages with hundred of characters and symbols.

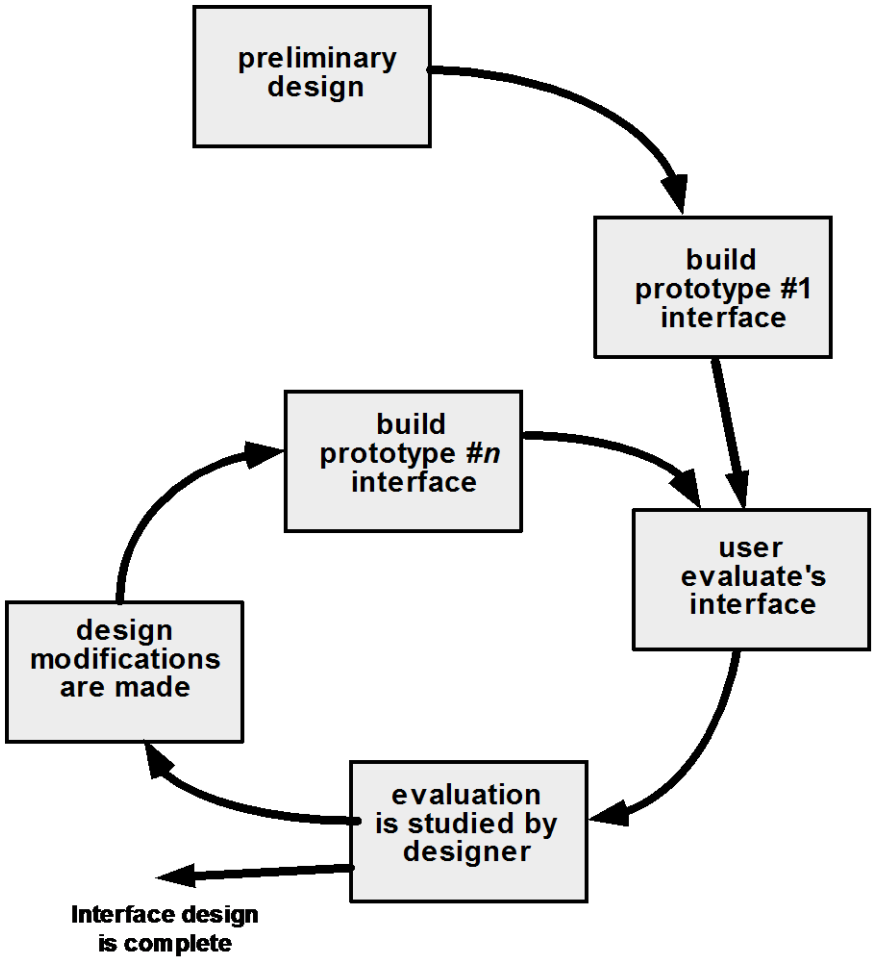
12.5 DESIGN EVALUATION

After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user, who provides the designer with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used e.g., questionnaires, rating sheets), the designer may extract information form these data (e.g., 80percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary. If a design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews:

1. The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by user of the system.
2. The number of user tasks specified and the average number of actions per task provide an indication on interaction time and the overall efficiency of the system.
3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.
4. Interface styles, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built, the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect 2qualitaive data, questionnaires can be distributed to users of the prototype. Questions can be (1) simple yes/no response, (2) numeric response, (3) scaled (subjective) response,(4) Likert scales(e.g., strongly.

Users are observed during interaction, and data-such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent "looking" at the display, number and types of errors, error recovery time, time spent using help, and number of help references per standard time period-are collected and used as a guide for interface modification.


preliminary design
build prototype #1 interface
build prototype #n interface
user evaluate's interface
design modifications are made
evaluation is studied by designer
Interface design is complete