UNIT 3

SOFTWARE DESIGN

SOFTWARE DESIGN

The IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990) defines software design as "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and "the result of [that] process".

- Good software design should exhibit:
- *Firmness:* A program should not have any bugs that inhibit its function.
- *Commodity:* A program should be suitable for the purposes for which it was intended.
- *Delight:* The experience of using the program should be pleasurable one.

DESIGN CONCEPTS

- A set of fundamental software design concepts has evolved over the history of software Engineering.
- Each helps you to answer the following questions

1. What criteria can be used for partition software into individual components?
2. How is function or data structure are separated from conceptual design of the software?
3. What uniform criteria define the technical quality of a software design?

DESIGN CONCEPTS

1. Abstraction
2. Architecture
3. Patterns
4. Modularity
5. Information Hiding
6. Functional Dependence
7. Refinement

8. Refactoring
9. Design Classes

Abstraction

- The process of hiding complex properties or characteristics from the software itself.
- High Level of Abstraction: Solution is stated in broad terms using the language of problem environment.
- Low Level of Abstraction : More detailed description of the solution is provided
- A Procedural Abstraction refers to a sequence of instructions that have a specific and limited function.
- A Data Abstraction is a named collection of data that describes a data object.

Architecture

- A Software Architecture alludes to *"the overall structure of the software and the ways in which that structure provides the conceptual integrity for a system."*
- Architecture is the structure or organization of program components the manner in which these components interact, and the structure of data that are used by the components.
- The architectural design can be represented using one or more of a number of different models.

a) **Structured models** represent architecture as an organized collection of program components

b) **Framework models** increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

c) **Dynamic models** address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function external events.

d) **Process models** focus on the design of the business or technical process that the system must accommodate.

e) **Functional models** can be used to represent the functional hierarchy of a system

Patterns

- A Design pattern describes a design structure that solves a particular design problem 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

  ➢ Whether the pattern is capable to the current work,

  ➢ Whether the pattern can be reused,

  ➢ Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

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.
- **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.
- 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 and changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

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.
- 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.

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
- Independence is assessed using two qualitative criteria:

➢ Cohesion is an indication of the relative functional strength of a module.

➢ Coupling is an indication of the relative interdependence among modules.

Refinement

- 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

Refactoring

- 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 structure or any other design failure that can be corrected to yield a better design.

Design Classes

- The software team must define a set of design classes that Refine the analysis classes by providing design detail that will enable the classes to be implemented.
- 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, HCI 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.

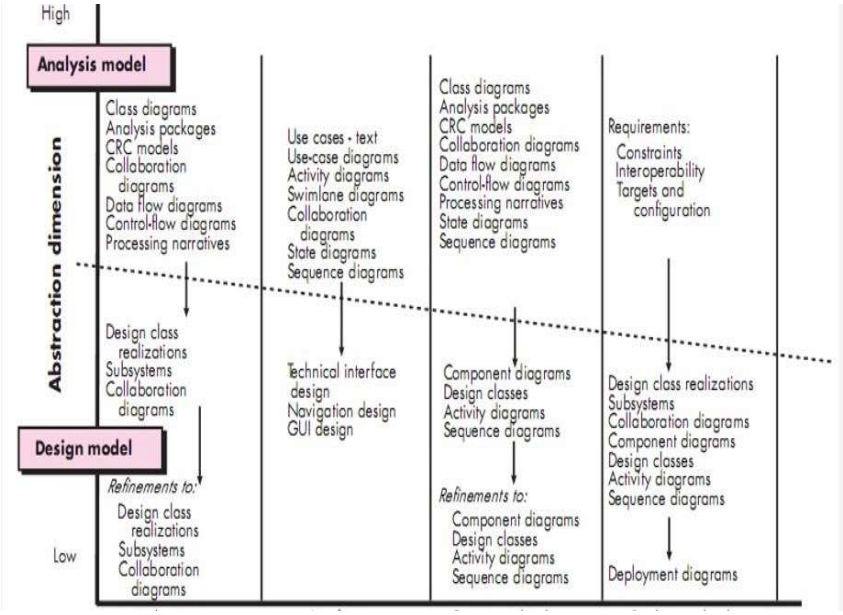
- Process classes implement lower-level business abstractions required to fully manage the business domain classes
- Persistent classes represent data stores that will persist beyond the execution of the software.
- System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

DESIGN MODEL

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

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



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.

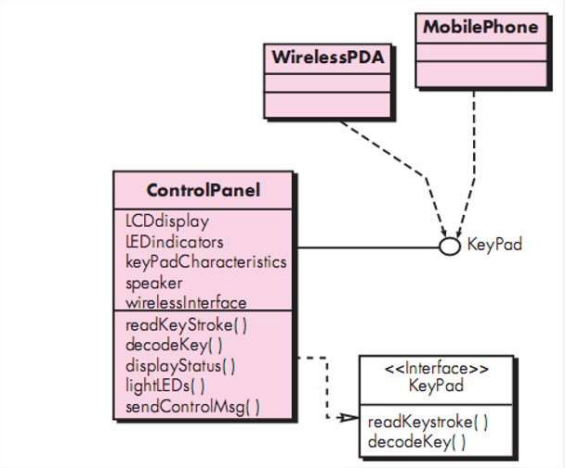
At **the application level**, the translation of a data model into a database is pivotal to achieving the business objectives of a system.

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.

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.

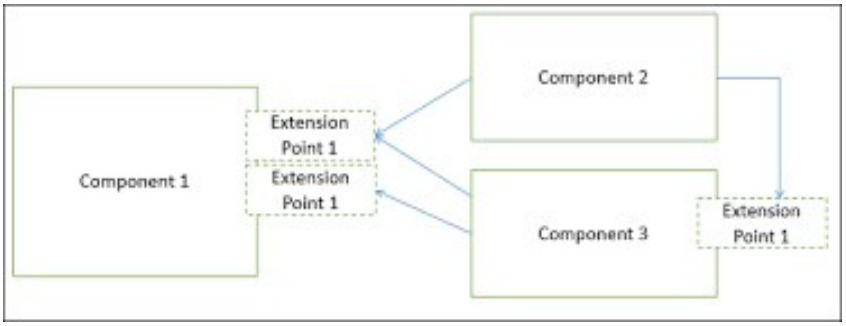
The interface design elements for software tell how information flows into and out of the system and how it is communicated among the components defined as part of the architecture. There are 3 important elements of interface design: The **user interface**(UI); **External interfaces** to other systems, devices, networks, or other produces or consumers of information; and **Internal interfaces** between various design components.



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



Deployment 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

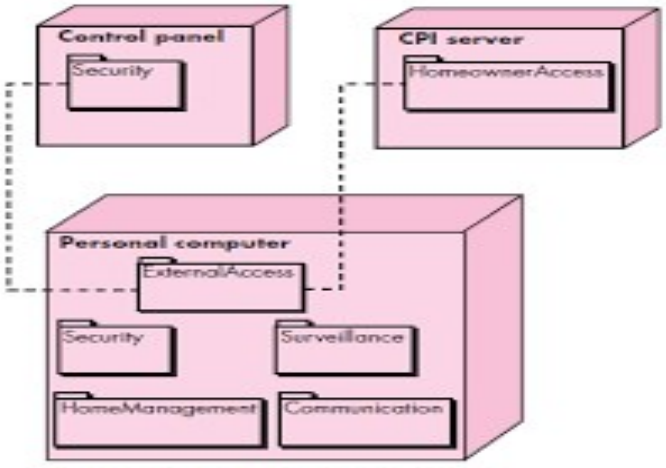


Fig 8.7: A UML deployment diagram

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

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.

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:

- The systematic analysis principles applied to function and behavior should also be applied to data. All data structures and the operations to be performed on each should be identified. A data dictionary should be established and used to define both data and program design. Low-level data design decisions should be deferred until late in the design process.
- The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure. A library of useful data structures and the operations that may be applied to them should be developed. A software design and programming language should support the specification and realization of abstract data types.

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)

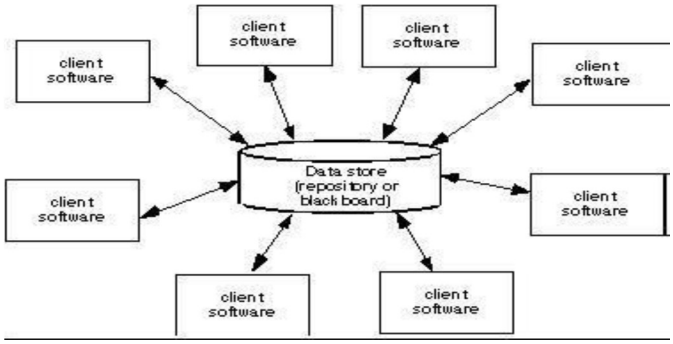
- The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses A set of components (e.g., a database, computational modules) that perform a function required by a system.
- A set of connectors that enable ―communication, coordination's 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 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:

- The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety. A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level.
- Architectural patterns tend to address specific behavioral issues within the context of the architectural.

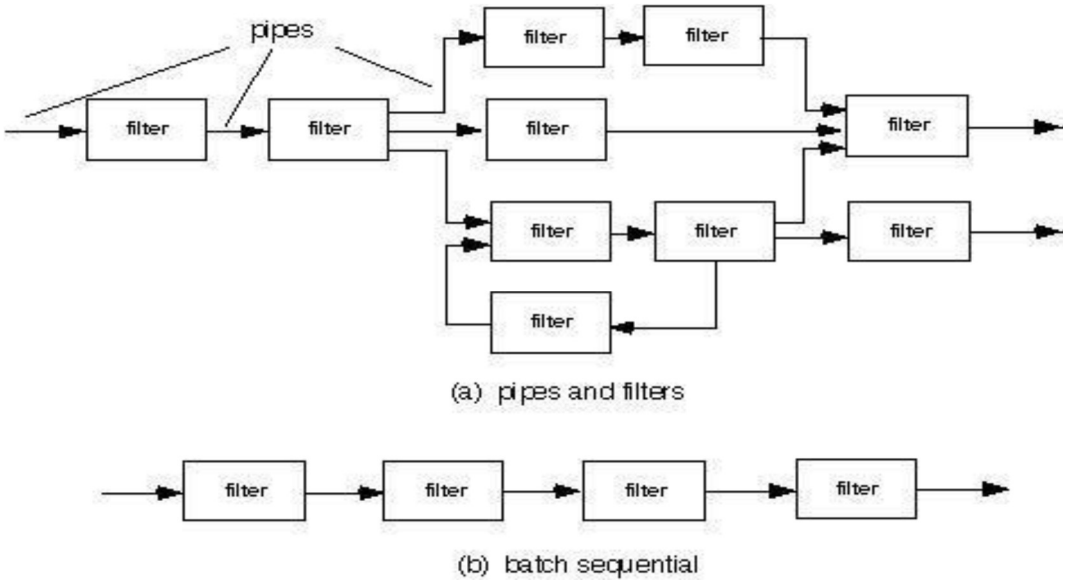
A Brief Taxonomy of Styles and Patterns Data-centered architectures:

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.



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.

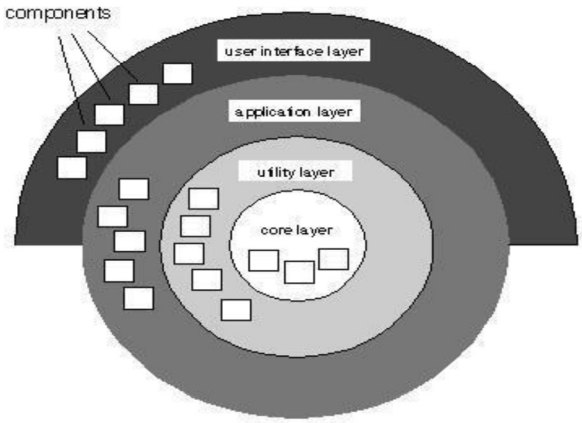


Object-oriented architectures.

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 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.



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

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

a) Concurrency—applications must handle multiple tasks in a manner that simulates parallelism operating system process management pattern o task scheduler pattern.

b**) 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.

c) 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.

d) 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.

ARCHITECTURAL DESIGN:

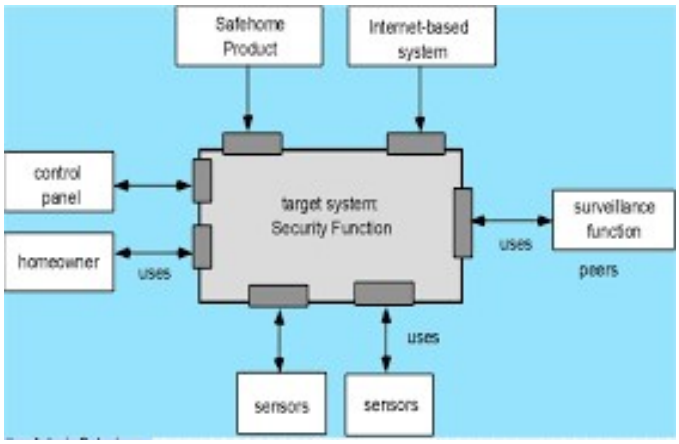
At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries.

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.



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.

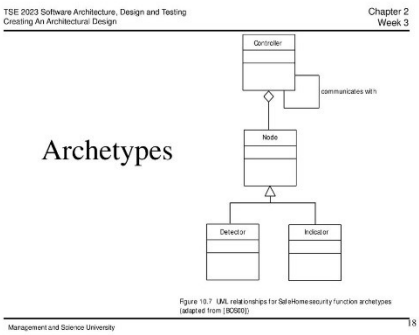
In many cases, archetypes can be derived by examining the analysis classes defined as part of the analysis model. In safe home security function, the following are the archetypes:

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.



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 object class definition is both a type specification and a template for creating objects. 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.

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

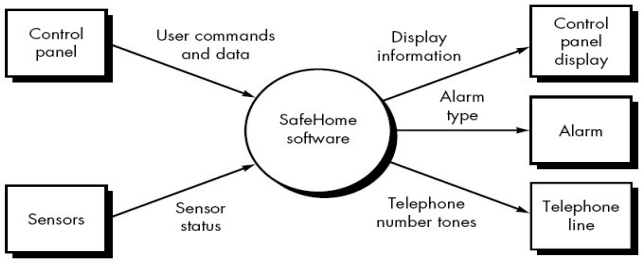
ARCHITECTURAL MAPPING USING DATA FLOW

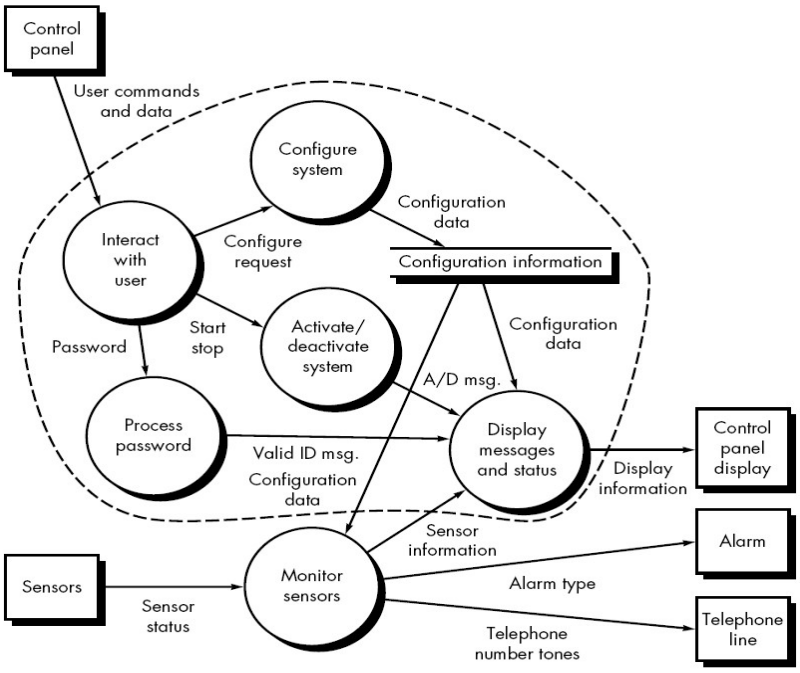
There is no practical mapping for some architectural styles, and the designer must approach the translation of requirements to design for these styles in using the techniques. To illustrate one approach to architectural mapping, consider the call and return architecture—an extremely common structure for many types of systems.

- The call and return architecture can reside within other more sophisticated architectures. For example, the architecture of one or more components of a client-server architecture might be call and return.
- A mapping technique, called structured design, is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture.
- The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six step process:
  (1) the type of information flow is established,
  (2) flow boundaries are indicated,
  (3) the DFD is mapped into the program structure,
  (4) control hierarchy is defined,
  (5) the resultant structure is refined using design measures and heuristics, and
  (6) the architectural description is refined and elaborated.

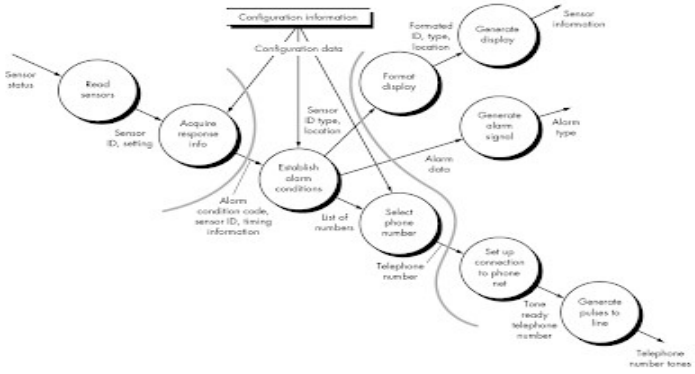
Transform Mapping: Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style.

Step 1. Review the fundamental system model. The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. Figure depicts a level 0 context model, a shows refined data flow for the security function.





Step 2. Review and refine data flow diagrams for the software. Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for monitor sensors is examined, and a level 3 data flow diagram is derived as shown in figure . The data flow diagram exhibits relatively high cohesion.



Step 3. Determine whether the DFD has transform or transaction flow characteristics. Evaluating the DFD in the Figure , we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for information flow.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. Incoming data flows along a path in which information is converted from external to internal form;

outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure 9.13. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary. The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

Step 5. Perform "first-level factoring." The program architecture derived using this mapping results in a top-down distribution of control. Factoring leads to a program structure in which top level components perform decision making and lowlevel components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work.

Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture.

MODELLING COMPONENT LEVEL DESIGN

- The purpose of component-level design is to define data structures, algorithms, interface characteristics, and communication mechanisms for each software component identified in the architectural design.
- Component-level design occurs after the data and architectural designs are established.
- The component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built.
- The work product produced is a design for each software component, represented using graphical, tabular, or text-based notation.

Component Definitions

- "A component is a modular building block for computer software."
- "A component is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."
- Components populate the software architecture and, as a consequence, play a role in achieving the objectives and requirements of the system to be built.

- Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.
- There are three important views of what a component is and how it is used as design modeling proceeds

Three views of Component

1. **An Object-Oriented View:**
   - A component contains a set of collaborating classes.
   - Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.
   - As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined.

Once this is completed, the following steps are performed

I. Provide further elaboration of each attribute, operation, and interface.
II. Specify the data structure appropriate for each attribute.
III. Design the algorithmic detail required to implement the processing logic associated with each operation.
IV. Design the mechanisms required to implement the interface to include the messaging that occurs between objects.

2. **The Traditional View:**

   A component is viewed as a functional element (i.e., a module) of a program that incorporates
   - The processing logic
   - The internal data structures
   - An interface that are required to implement the processing logic that enables the component to be invoked and data to be passed to it

A component serves one of the following roles

- A control component that coordinates the invocation of all other problem domain components
- A problem domain component that implements a complete or partial function that is required by the customer
- An infrastructure component that is responsible for functions that support the processing required in the problem domain

3. **Process-Related view**
    - Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
    - As the software architecture is formulated, components are selected from the library and used to populate the architecture

    Because the components in the library have been created with reuse in mind, each contains the following:

    - A complete description of their interface
    - The functions they perform
    - The communication and collaboration they require

Component-level Design Principles

Open-closed principle (OCP)

- A module or component should be open for extension but closed for modification
- The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications of the component to the existing parts

Liskov substitution principle (LSP)

- Subclasses should be substitutable for their base classes
- A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead

Dependency inversion principle (DIP)

- Depend on abstractions (i.e., interfaces); do not depend on concretions
- The more a component depends on other concrete components (rather than on the interfaces) the more difficult it will be to extend

Interface segregation principle (ISP)

- Many client-specific interfaces are better than one general purpose interface
- For a server class, specialized interfaces should be created to serve major categories of clients
- Only those operations that are relevant to a particular category of clients should be specified in the interface

Release reuse equivalency principle (REP)

- The granularity of reuse is the granularity of release
- Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created

Common closure principle (CCP)

- Classes that change together belong together
- Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change

Common reuse principle (CRP)

- Classes that aren't reused together should not be grouped together
- Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

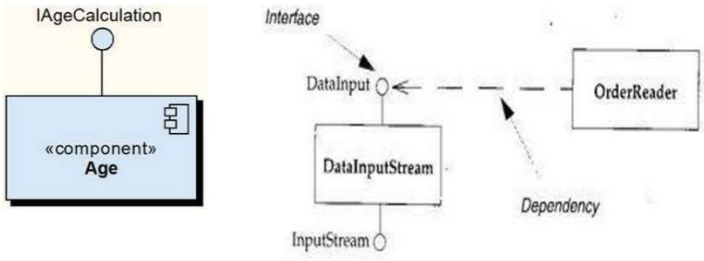
Component-Level Design Guidelines:

Components

- Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
- Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
- Use infrastructure meaning component names that reflect their implementation-specific(e.g., Stack)

Interfaces

- Interfaces provide important information about communication and collaboration
- lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach
- only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available
- These recommendations are intended to simplify the visual nature of UML component diagrams.



Dependencies and inheritance in UML

- Model any dependencies from left to right bottom and inheritance from top (base class) rather than to bottom (derived classes)
- Consider modeling any component dependencies as interfaces representing them as a direct component-to-component dependency

Cohesion:

Cohesion is the "single-mindedness' of a component

It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself

The objective is to keep cohesion as high as possible

The kinds of cohesion can be ranked in order from highest to lowest

Functional

A module performs one and only one computation and then returns a result

Layer

- Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

Communicational

- All operations that access the same data are defined within one class
- Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.

Coupling

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible
- **Content coupling.** Occurs when one component "secretly modifies data that is internal to another component". This violates information hiding— a basic design concept.
- **Common coupling.** Occurs when a number of components all make use of a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.
- **Control coupling.** Occurs when operation A() invokes operation B() and passes a control flag to B. The control flag then "directs" logical flow within B. The problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes. If this is overlooked, an error will result.
- **Stamp coupling.** Occurs when Class B is declared as a type for an argument of an operation of ClassA. Because Class B is now a part of the definition of Class A, modifying the system becomes more complex.

- **Data coupling.** Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.
- **Routine call coupling.** Occurs when one operation invokes another. This level of coupling is common and is often quite necessary.
- **Type use coupling.** Occurs when component A uses a data type defined in component B (e.g., this occurs whenever "a class declares an instance variable or a local variable as having another class for its type"). If the type definition changes, every component that uses the definition must also change.
- **Inclusion or import coupling.** Occurs when component A imports or includes a package or the content of component B.
- **External coupling.** Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

USER INTERFACE DESIGN

Interface design focuses on three areas of concern: the design of interfaces between software components, the design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities), and 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.

THE GOLDEN RULES

Theo Mandel coins three ―golden rules:

- **Place the user in control.**
    1. Provide for flexible interaction. Different users have different interaction preferences therefore provide choices . Save icon, ctrl S , File save. software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.
    2. Allow user interaction to be interruptible and undoable(UPS). Allow user to put something on hold (interrupt) for sometime and come back to it later without losing what he has already done. Allow undo action.
    3. Allow interaction to be customized. Design a macro mechanism for advanced users.
    4. Hide technical internals from the casual user. The user does not need to know about the operating system or file structure. (e.g., a user should never be required to type operating system commands from within application software)
    5. Design for direct interaction with objects that appear on the screen. Allow dragging of corners to increase / decrease size.
    6. Avoid unnecessary actions. Do not force the user to do unnecessary actions i.e. allow him to make corrections through spell check.
- **Reduce the user's memory load.**
    1. Reduce demand on short term memory. Give visual cues to recognize past actions. New windows as you select subfolders in Win NT.
    2. Establish meaningful defaults and allow customization and reset.
    3. Define shortcuts that are intuitive. Ctrl S for saving and Ctrl P for printing. Mnemonics.
    4. The visual layout of the interface should be based on a real world metaphor. For example use exact layout of cheque for making payment. Attendance sheet.
    5. Disclose information in a progressive manner. The interface should be organized hierarchically. Information about a task, object etc. is presented first at a high level of abstraction. More detail should be presented after the user indicates interest in details.

- **Make the interface consistent.**

1. Maintain consistency across a family of applications. Appearance, options, mnemonics (shortcuts).
2. Do not make changes unless it is absolutely necessary. If ctrl s is used for Saving do not change it to Scaling.

These golden rules actually form the basis for a set of user interface design principles that guide this important software design activity.

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.

a) 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

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

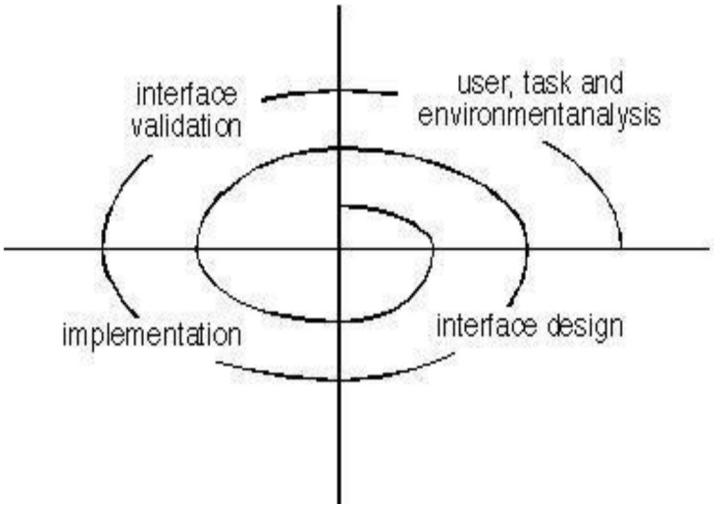
c) Mental Model: The user's mental model (system perception) is the image of the system that end-users carry in their heads.

d) 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."

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

The user interface design process encompasses four distinct framework activities: User, task, and environment analysis and modeling Interface design Interface construction Interface validation.



(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

(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 the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements the degree to which the interface is easy to use and easy to learn; and the users‘ acceptance of the interface as a useful tool in their work.