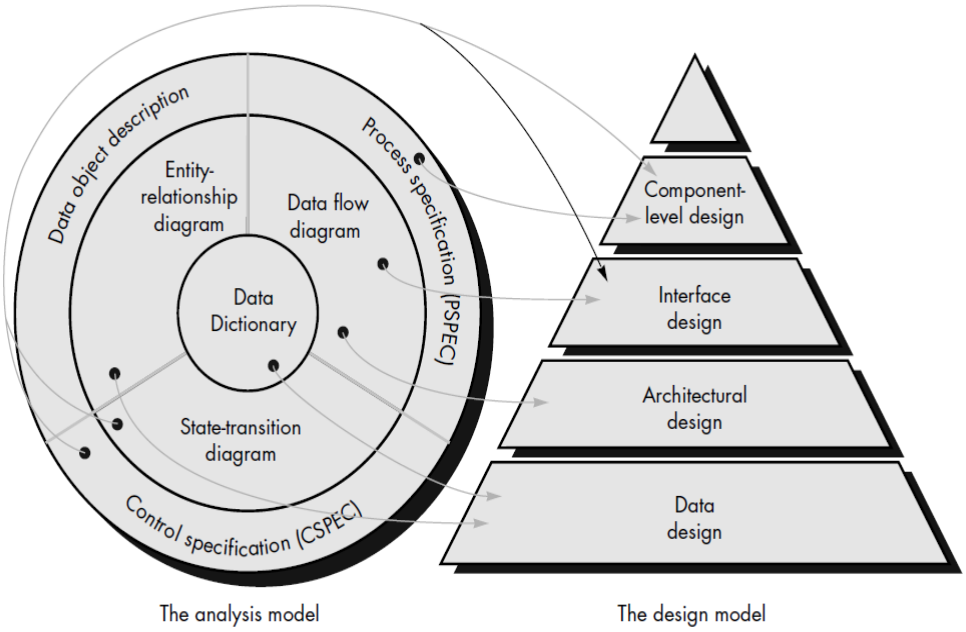
UNIT 3 DESIGN PROCESS AND CONCEPTS

Design process - Modular design - Design heuristic - Design model and document - Architectural design - Software architecture - Data design - Architecture data - Transform and transaction mapping - User interface design - User interface design principles.

3.1 Design process

Software Design and Software Engineering



- The **data design** transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity.

- The **architectural design** defines the relationship between major structural elements of the software, the “design patterns” that can be used to achieve the requirements that have been defined for the system, The architectural design representation the framework of a computer-based system can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis mode.

- The **interface design** describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. data and control flow diagrams provide much of the information required for interface design.

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

The Design Process

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

Design and Software Quality

- The quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs McGlaughlin 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 behavioural domains from an implementation perspective.

Design Principles

- Design process should not suffer from "tunnel vision"
- The design should be traceable to the analysis model
- The design should not reinvent the wheel; Time is short
- The design should "minimize intellectual distance" between the software and the problem in the real world
- The design should exhibit uniformity and integration
- The design should be structured to accommodate change
- The design should be structured to degrade gently.
- Design is not coding, coding is not design
- The design should be assessed for quality as it is being created, not after the fact
- The design should be reviewed to minimize conceptual errors

Design Concepts
Fundamental concepts which provide foundation to design correctly:

- Abstraction
- Refinement
- Modularity
- Software Architecture
- Control Hierarchy
- Structural Partitioning
- Data Structure
- Software Procedure
- Information Hiding

Abstraction

- Identifying important features for representation
- There are many levels of abstraction depending on how detailed the representation is required
- Data abstraction - representation of data objects
- Procedural abstraction - representation of instructions

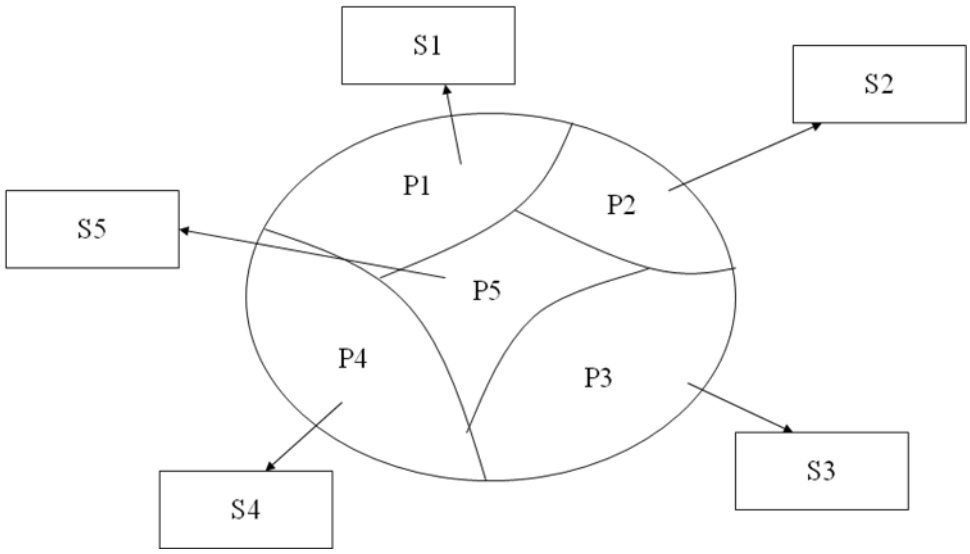
Refinement

- Stepwise refinement - top-down design strategy by Niklaus Wirth
- Refinement is actually a process of elaboration
- Starting at the highest level of abstraction, every step of refinement ‘decompose’ instructions into more detailed instructions
- Complementary to abstraction

Modularity

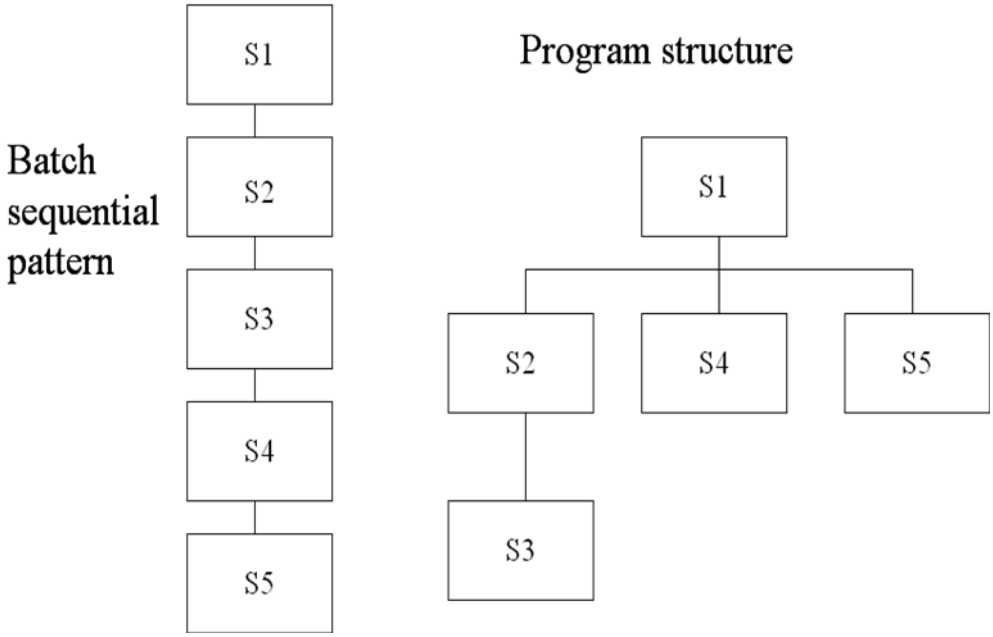
- Software is divided into separately named and addressable components, often called modules, that are integrated to satisfy problem requirements.
- “Divide and conquer” approach - problem is broken into manageable pieces
- Solutions for the separate pieces then integrated into the whole system

Divide and Conquer



Software Architecture

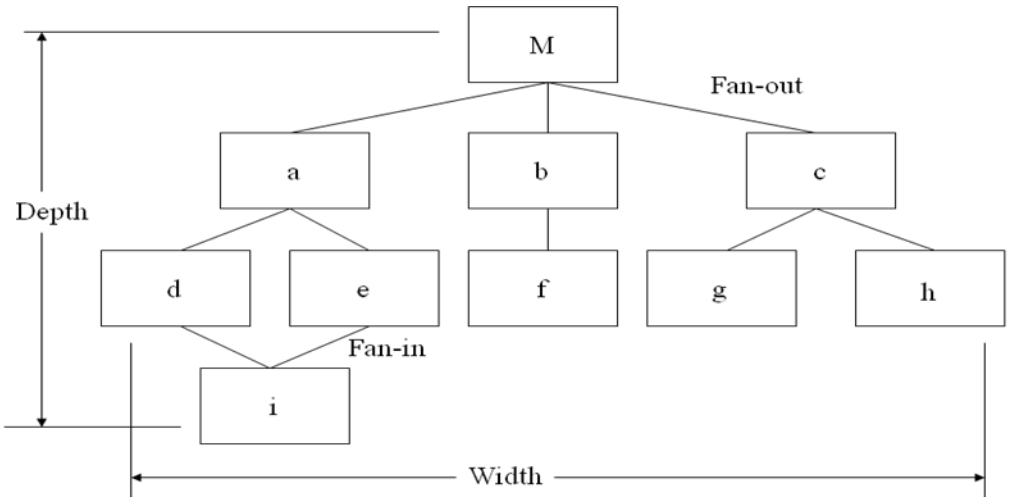
- Modules can be integrated in many ways to produce the system
- Software architecture is the overall structure of the software
- The hierarchy of components and how they interact, and the structure of data used by the components
- Use of framework models, and possible reuse of architectural patterns



Control Hierarchy

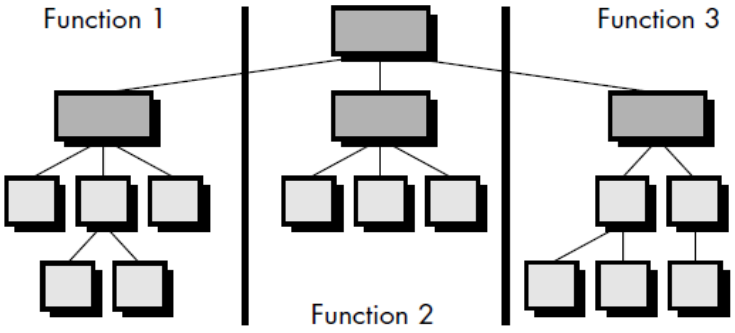
- Control hierarchy, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control.

- Hierarchy of modules representing the control relationships
- A super-ordinate module controls another module
- A subordinate module is controlled by another module
- Measures relevant to control hierarchy: depth, width, fan-in, fan-out
- Depth and width provide an indication of the number of levels of control and overall span of control.
- Fan-out is a measure of the number of modules that are directly controlled by another module. Fan-in indicates how many modules directly control a given module.

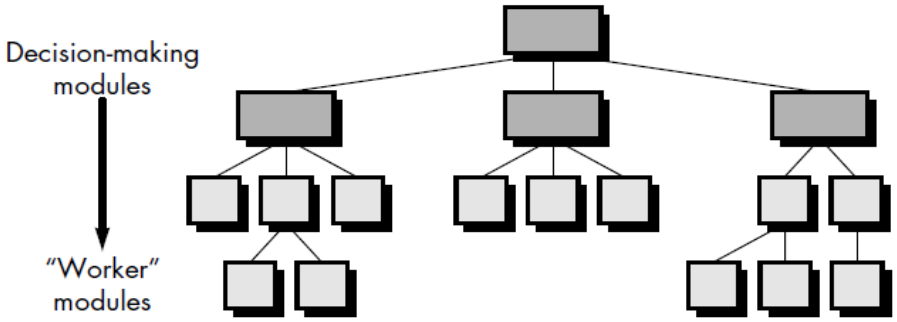


Structural Partitioning

- Program structure partitioned horizontally and vertically
- Horizontal partitioning defines separate branches for each major program function - input, process, output
- Vertical partitioning defines control (decision-making) at the top and work at the bottom



(a) Horizontal partitioning



(b) Vertical partitioning

Software Procedure

- Processing details of individual modules
- Precise specification of processing, including sequence of events, exact decision points, repetitive operations, and data organization/structure
- Procedure is layered - subordinate modules must be referenced in processing details

Information Hiding

- Information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information
- Effective modularity is achieved by independent modules, that communicate only necessary information
- Ease of maintenance - testing, modification localized and less likely to propagate

Data Structure

- Data structure is a representation of the logical relationship among individual elements of data.

3.2 Modular Design

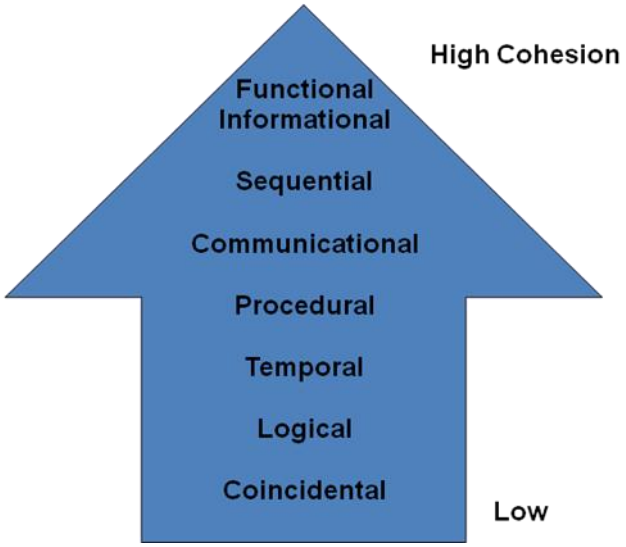
- **Functional Independence**
- Designing modules in such a way that each module has specific functional requirements. Functional independence is measured using two terms **cohesion** and **coupling**.

Cohesion

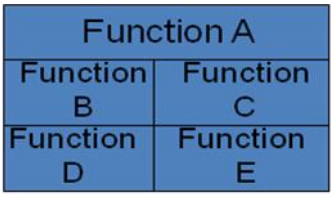
- Internal interaction of the module.
- **Cohesion** is a measure of relative functional strength of a module
- The degree to which all elements of a component are directed towards a single task and all elements directed towards that task are contained in a single component.

Types of cohesion

- Logical Cohesion
- Coincidental cohesion
- Temporal Cohesion
- Procedure Cohesion
- Communication Cohesion
- Sequential cohesion
- Informational cohesion
- Functional cohesion



Coincidental cohesion

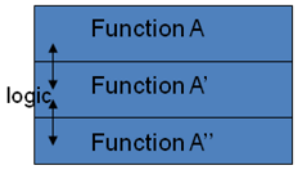


Coincidental
Parts unrelated

- Parts of a component are simply bundled together

- The result of randomly breaking the project into modules to gain the benefits of having multiple smaller files/modules to work on
- Usually worse than no modularization
- A module has coincidental cohesion if it performs multiple, completely unrelated actions

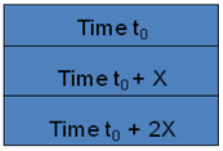
Logical Cohesion



Logical
Similar functions

- Elements of component are related logically and not functionally
- Results in hard to understand modules with complicated logic

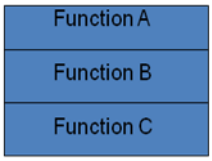
Temporal Cohesion



Temporal
Related by time

- Elements of a component are related by timing

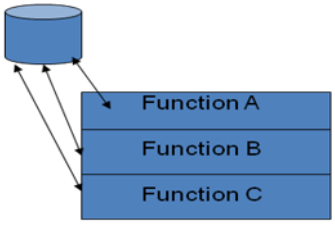
Procedural Cohesion



Procedural
Related by order of functions

- Elements of a component are related only to ensure a particular order of execution.

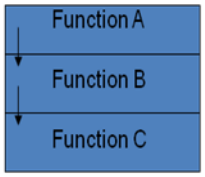
Communicational Cohesion



Communicational
Access same data

- Module performs a series of actions related by a sequence of steps to be followed by the product and all actions are performed on the same data

Sequential Cohesion



Sequential
Output of one is input to another

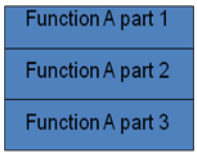
- The output of one component is the input to another.
- Occurs naturally in functional programming languages

Informational Cohesion

- Module performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data.

Functional Cohesion

- Module with functional cohesion focuses on exactly one goal or “function”
- Every essential element to a single computation is contained in the component.
- Every element in the component is essential to the computation

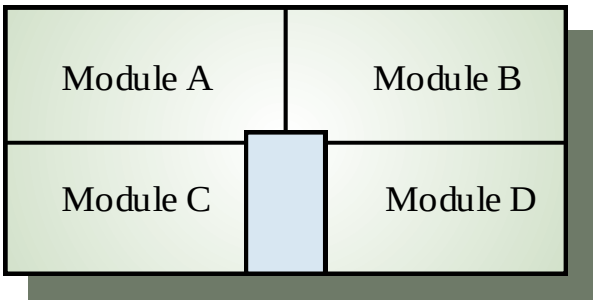


Functional
Sequential with complete, related functions

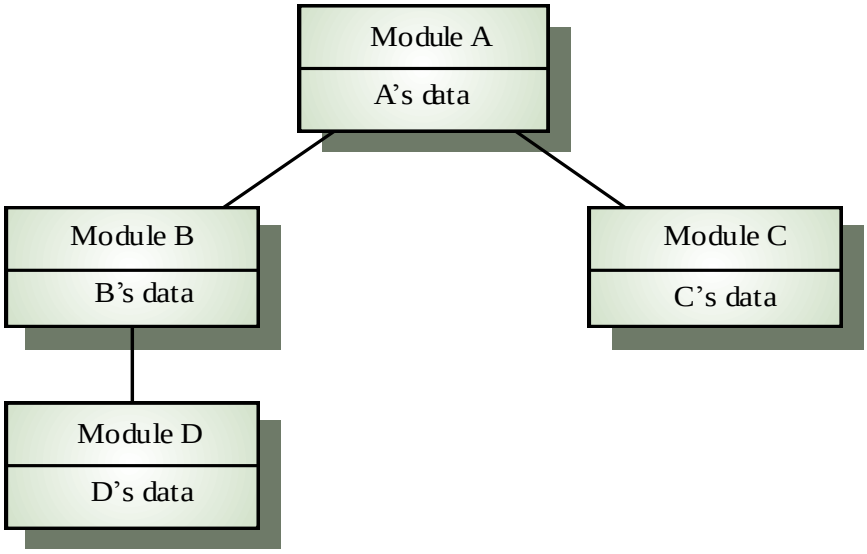
Coupling

- **Coupling** is a measure of relative independence among modules, that is it is a measure of interconnection among modules.
- **Loose coupling** means component changes are unlikely to affect other components.
- Shared variables or control information exchange lead to **tight coupling**.
- Loose coupling can be achieved by state decentralization (as in objects) and component communication via parameters or message passing.

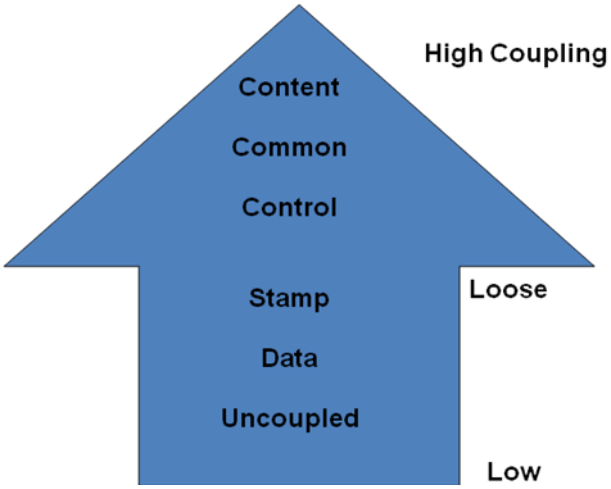
Tight Coupling



Loose Coupling



Range of Coupling



Types of coupling

- Content coupling
- Common Coupling

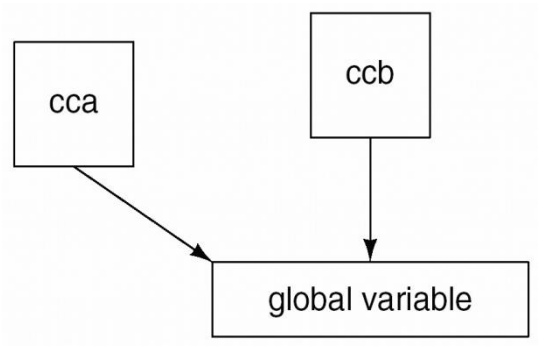
- Control Coupling
- Stamp Coupling
- Data Coupling

Content coupling

- One module directly refers to the content of the other
  - Module a modifies statement of module b

Common Coupling

- Common coupling exists when two or more modules have read and write access to the same global data.



Control Coupling

- Two modules are control-coupled if module 1 can directly affect the execution of module 2

Stamp Coupling

- It is a case of passing more than the required data values into a module
- Two modules are stamp coupled if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure

Data Coupling

- Two modules are data coupled if all parameters are homogeneous data items [simple parameters, or data structures all of whose elements are used by called module]

3.3 Design Heuristics for Effective Modularity

- Evaluate the first iteration of the program structure to reduce coupling and improve cohesion.
- Attempt to minimize structures with high fan-out; strive for fan-in as structure depth increases.
- Keep the scope of effect of a module within the scope of control for that module.

- Evaluate module interfaces to reduce complexity, reduce redundancy, and improve consistency.
- Define modules whose function is predictable and not overly restrictive (e.g. a module that only implements a single subfunction).
- Strive for controlled entry modules, avoid pathological connection (e.g. branches into the middle of another module)

3.4 Design model and document

The Design Model

- We want to create a software design that is stable. By establishing a broad foundation using data design, a stable mid-region with architectural and interface design, and a sharp point by applying component-level design, we create a design model that is not easily “tipped over” by the winds of change

Design Documentation

- First, the overall scope of the design effort is described. Much of the information presented here is derived from the System Specification and the analysis model (Software Requirements Specification).
- Next, the data design is specified. Database structure, any external file structures,internal data structures, and a cross reference that connects data objects to specific files are all defined
- The architectural design indicates how the program architecture has been derived from the analysis model. In addition, structure charts are used to represent the module hierarchy
- The design of external and internal program interfaces is represented and a detailed design of the human/machine interface is described. In some cases, a detailed prototype of a GUI may be represented.
- Components—separately addressable elements of software such as subroutines, functions, or procedures—are initially described with an English-language processing narrative.
- The processing narrative explains the procedural function of a component (module). Later, a procedural design tool is used to translate the narrative into a structured description.
- The Design Specification contains a requirements cross reference. The purpose of this cross reference is (1) to establish that all requirements are satisfied by the software

design and (2) to indicate which components are critical to the implementation of specific requirements.

- The final section of the Design Specification contains supplementary data. Algorithm descriptions, alternative procedures, tabular data, excerpts from other documents, and other relevant information

3.5 Architectural Design

- **Software Architecture**
  - Software architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution
  - The architecture of a software system defines that system in terms of computational components and interactions among those components.

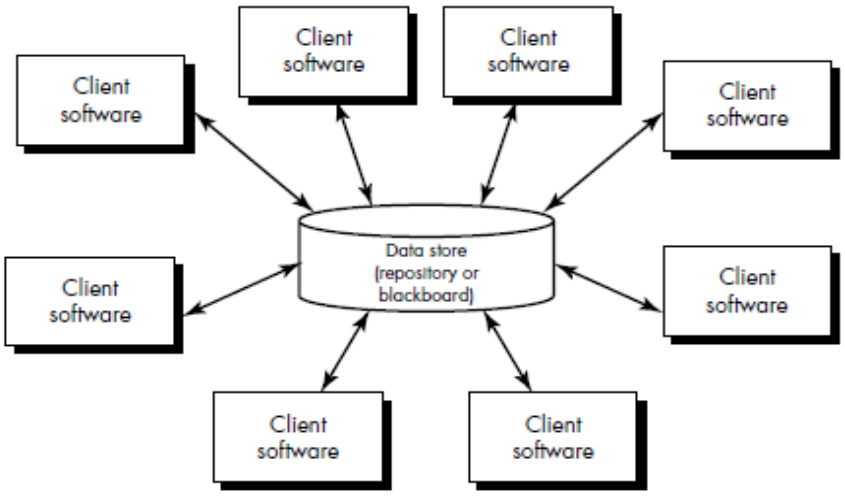
Architectural Styles

- An architectural style is a description of component and connector types and a pattern of their runtime control and/or data transfer.
- An architectural style, sometimes called an architectural pattern, is a set of principles—a coarse grained pattern that provides an abstract framework for a family of systems.
- Defines ways of selecting and presenting architectural building blocks

Benefits

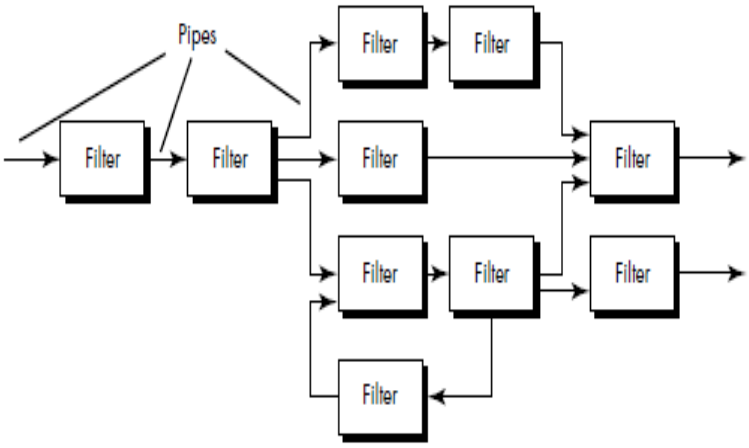
- Design Reuse
- Code Reuse (may be domain dependant)
- Communication among colleagues
- Interoperability
- System Analysis

Data-centered architectures



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

Data-flow architectures

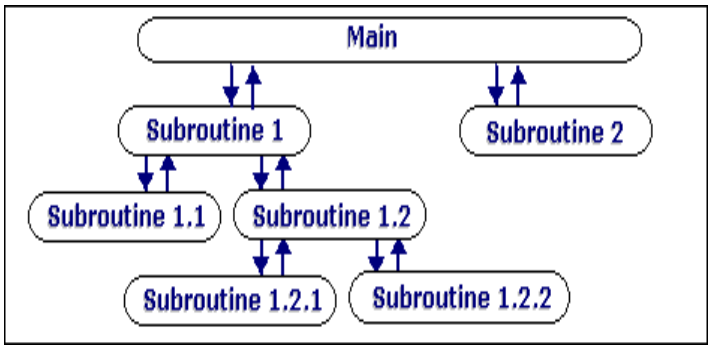


- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.

- A **pipe and filter** pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- The filter does not require knowledge of the working of its neighboring filters.

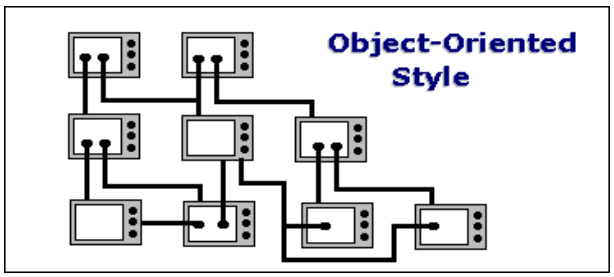
Call and return architectures

- This architectural style enables a software designer to achieve a program structure that is relatively easy to modify and scale.
- **Main program/subprogram architectures**. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.
- **Remote procedure call architectures**. The components of a main program/ subprogram architecture are distributed across multiple computers on a network

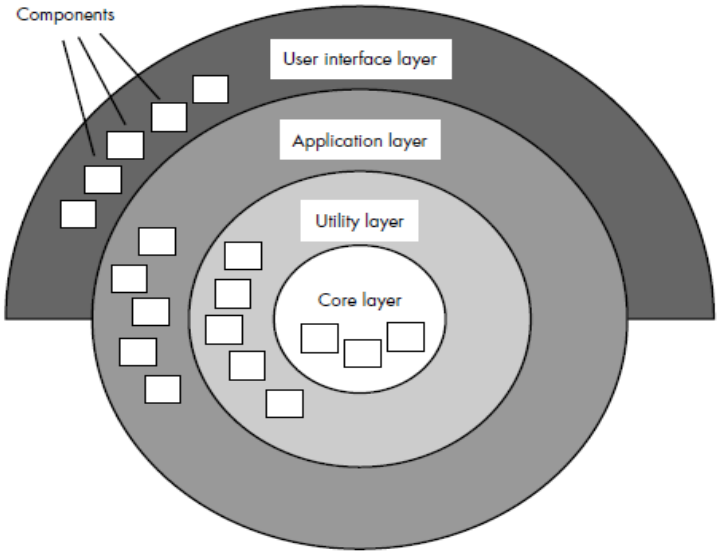


Object-oriented architectures

- The components of a system encapsulate data and the operations that must be applied to manipulate the data.
- Communication and coordination between components is accomplished via message Passing.



Layered architectures

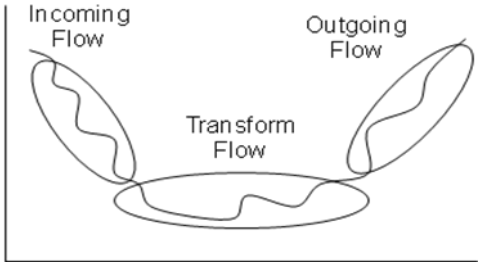


- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.

3.6 Transform and transaction mapping

Transform Flow

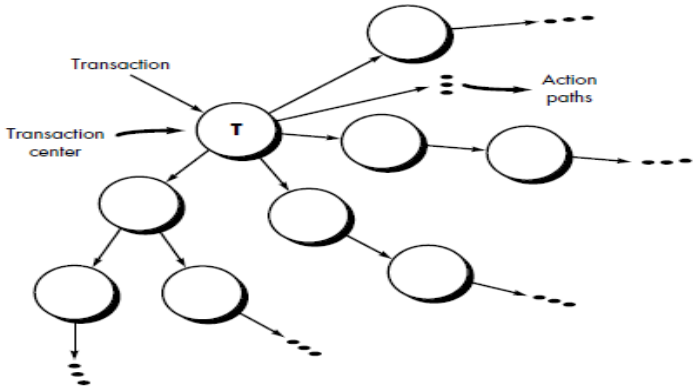
- Data “continuously” moves through a collection of incoming flow processes, transform center processes, and finally outgoing flow processes.



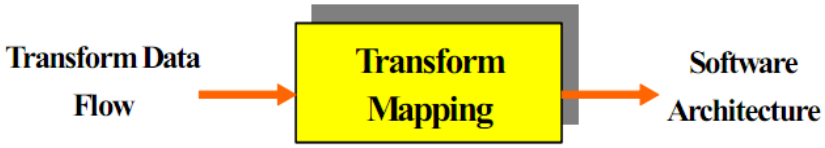
- **Incoming flow:** Information enters the system along paths that transform external data into internal data
- **Transform flow:** Internal data is processed
- **Outgoing flow:** Internal data are transformed into external data

Transactional Flow

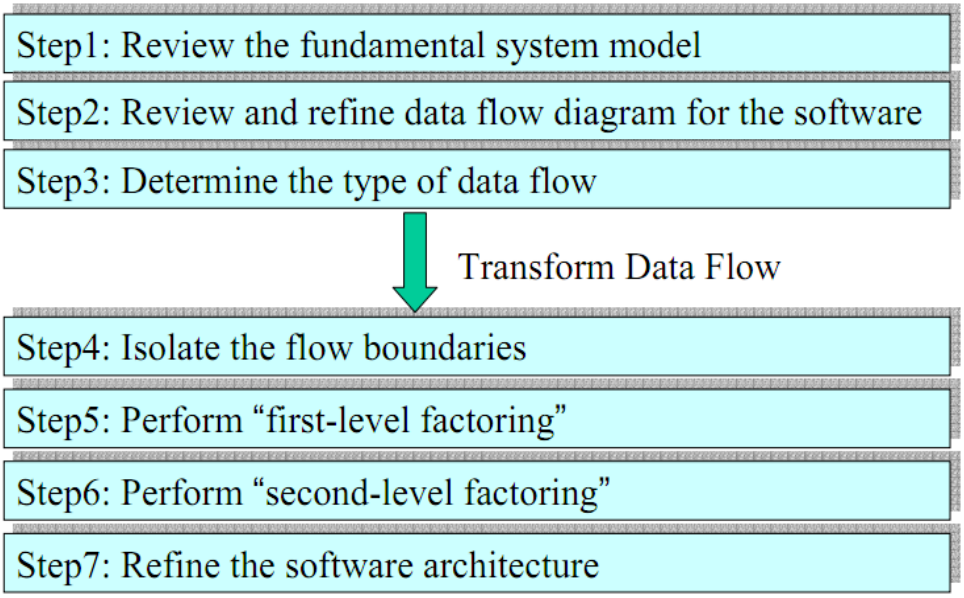
- Data “continuously” moves through a collection of incoming flow processes, reaches a particular transaction center process, and then follows one of a number of actions paths. Each action path is again a collection of processes.



- Mapping the transform data flow diagram into software architecture design model
- Input: transform data flow diagram
- Output: software architecture



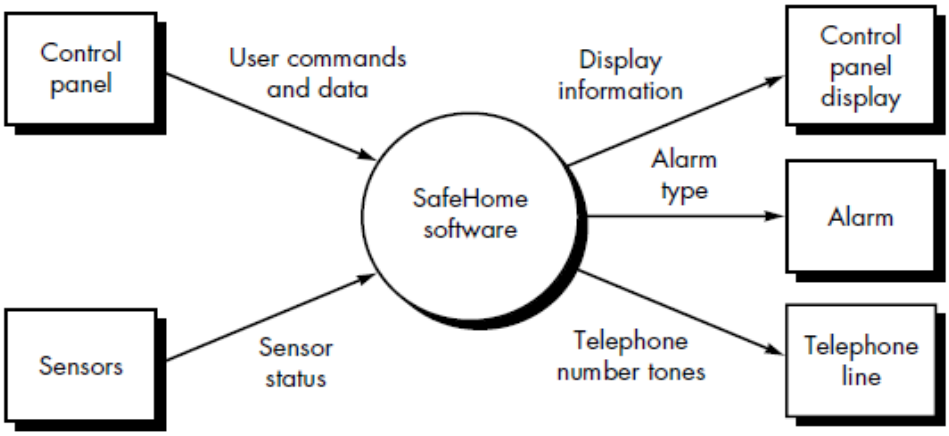
Process of Transform Mapping



Step1:Review the fundamental system model

- What is fundamental system model?
  - Top-level or 0-level data flow diagram
- Why reviewing the fundamental system model?
  - To evaluate the SRS in order to guarantee that the system model conforms to the real system

Level 0 DFD

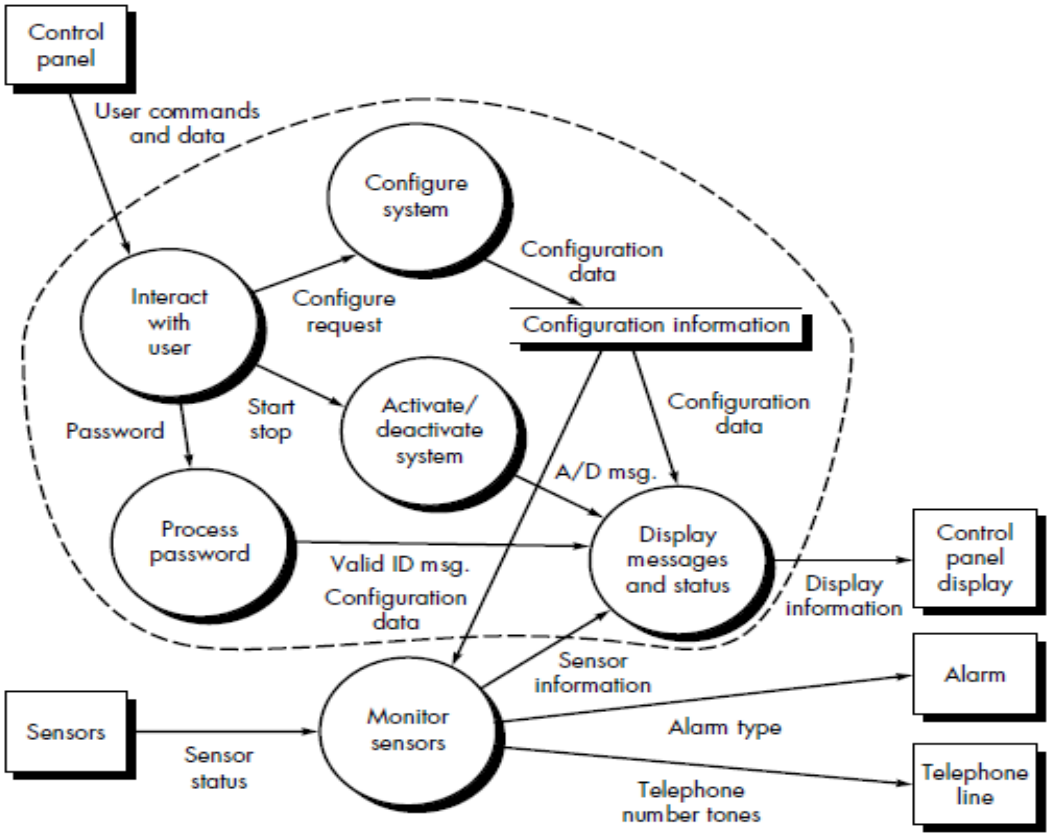


Step2: Review and Refine Data Flow Diagram for the Software

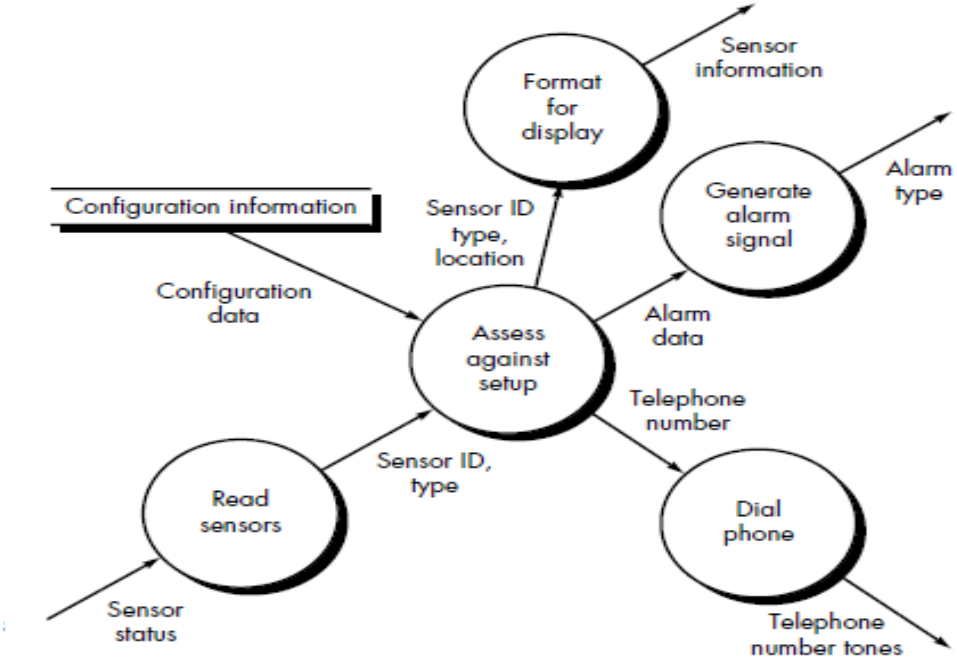
- DFD is correct
- Produce greater detail

- Each transform in the data flow diagram exhibits relatively high cohesion that can be implemented as a component in software

Level 1 DFD



Level 2 DFD Refine monitor sensor process



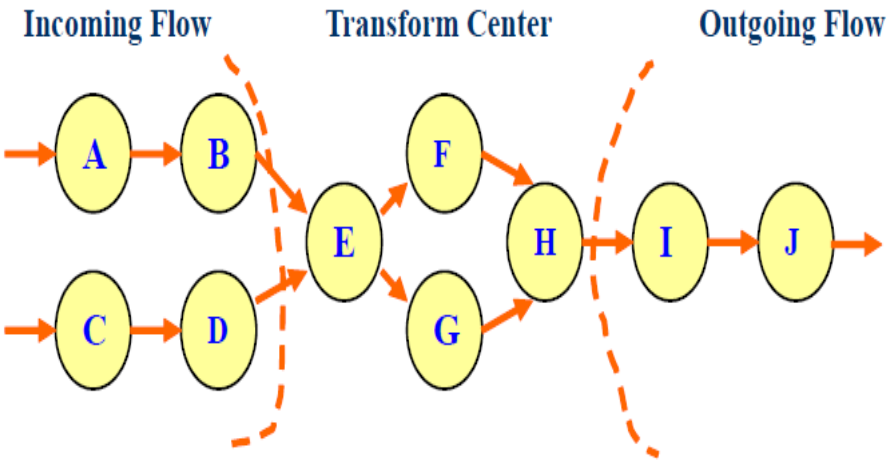
Step 3: Determine the type of data flow

- Transform flow or transaction flow
- Different type of data flow corresponds to different mapping approach

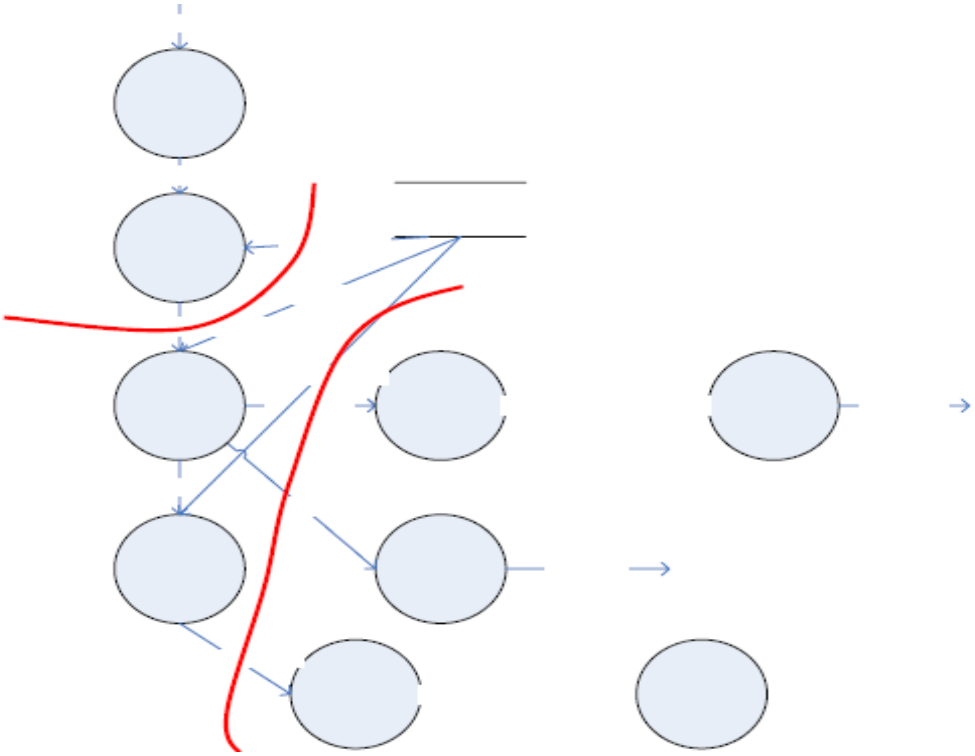
Step 4:Isolate the flow boundaries

- Incoming flow
- Transform center
- Outgoing flow
- Different designers may select slightly different points in flow as boundary location, and therefore have different design

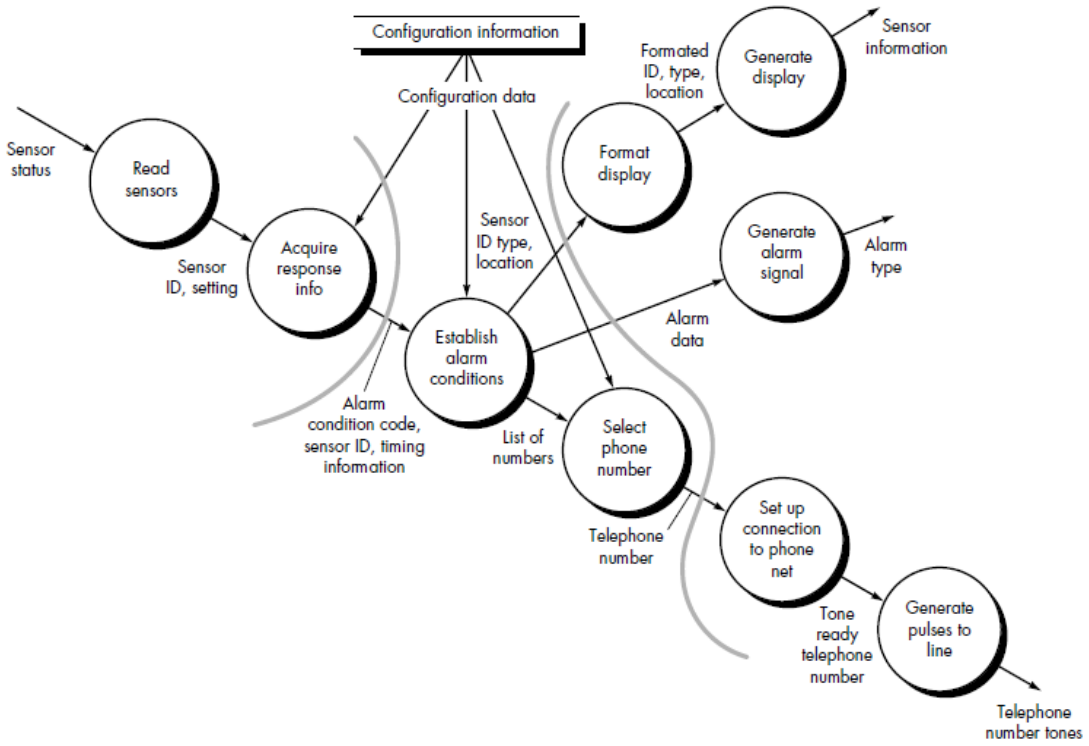
Isolating Boundary



Isolating Boundary



Level 3 DFD for monitor sensors with flow boundaries

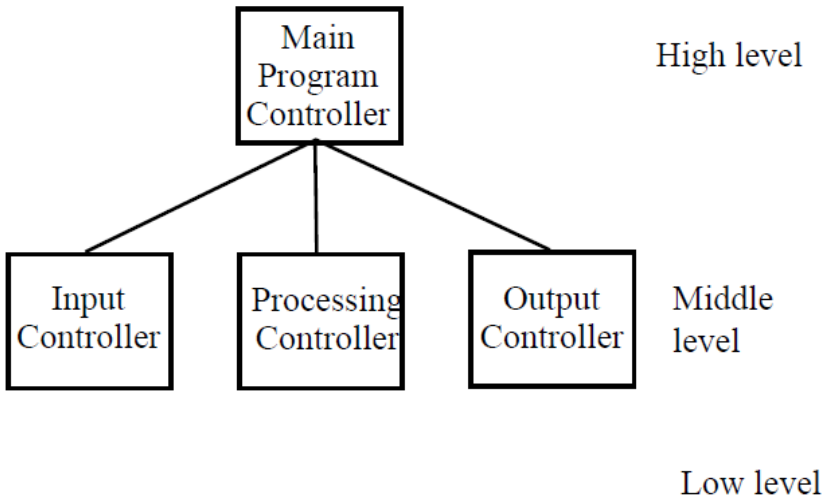


Step5. Perform First-level Factoring

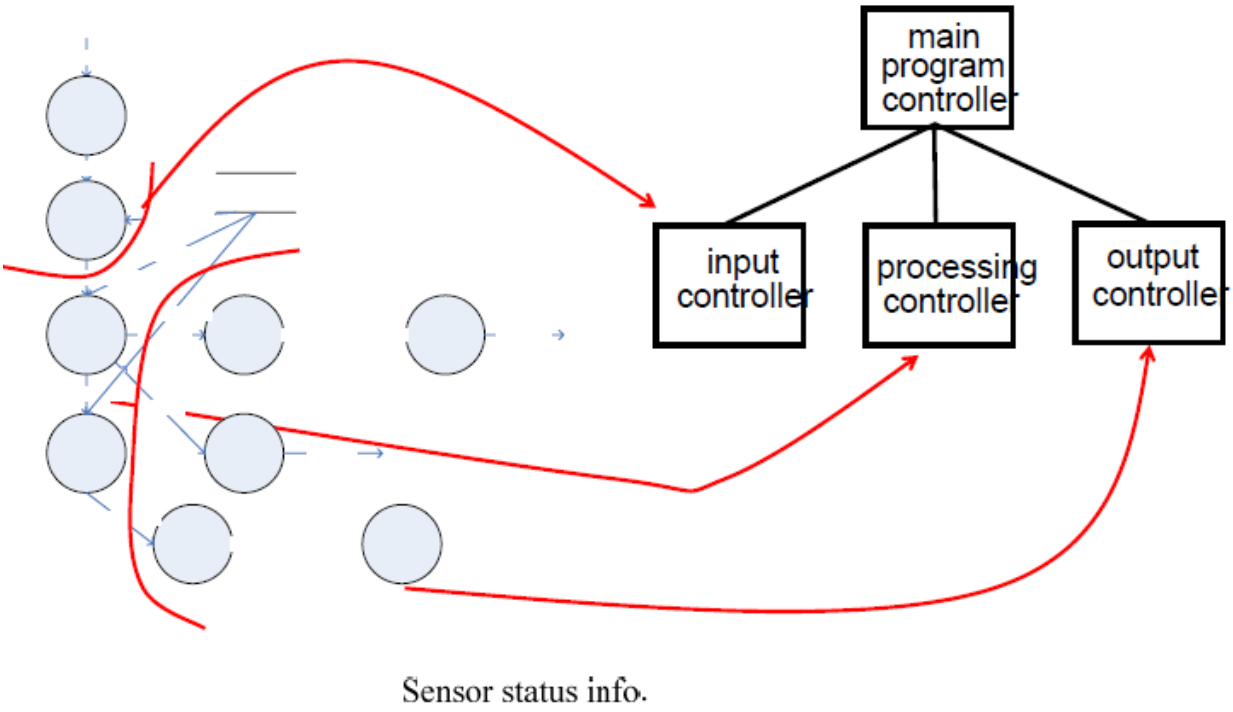
- **Top-level modules:** decision making
- **Middle-level modules:** some control and some work
- **Low-level modules:** perform most input, computational, and output work

- The generated software structure can be specified by hierarchy diagram or structure diagram

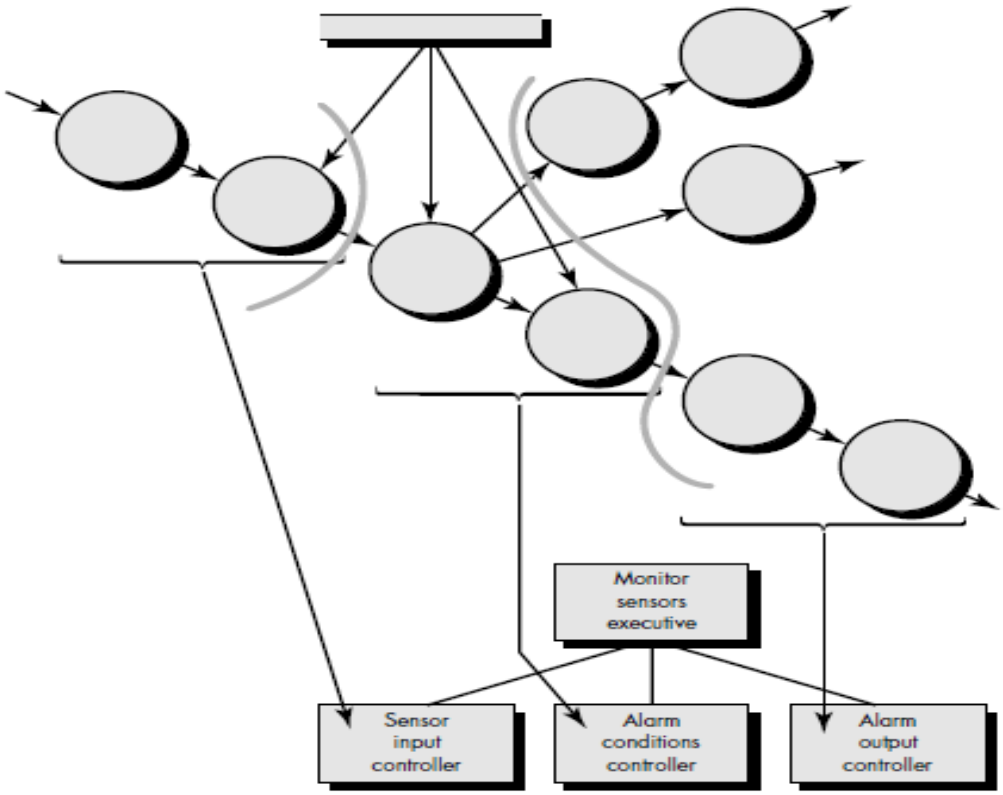
First-level Factoring



First-level Factoring



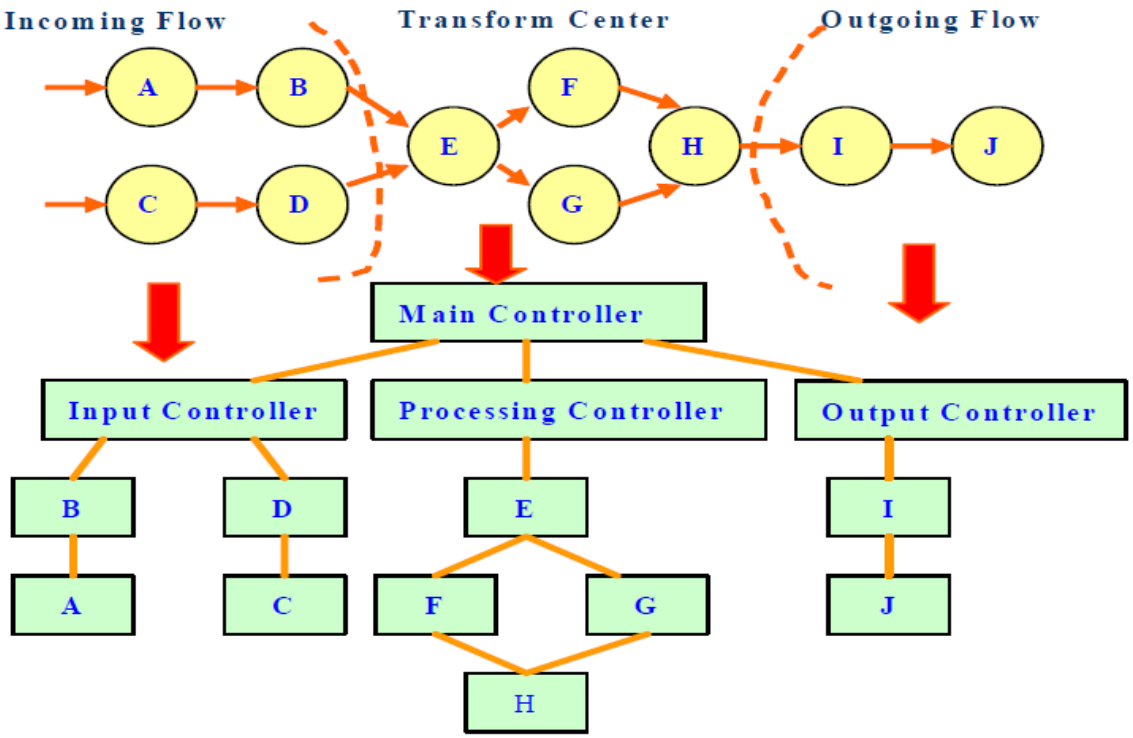
First-level Factoring of monitor sensor



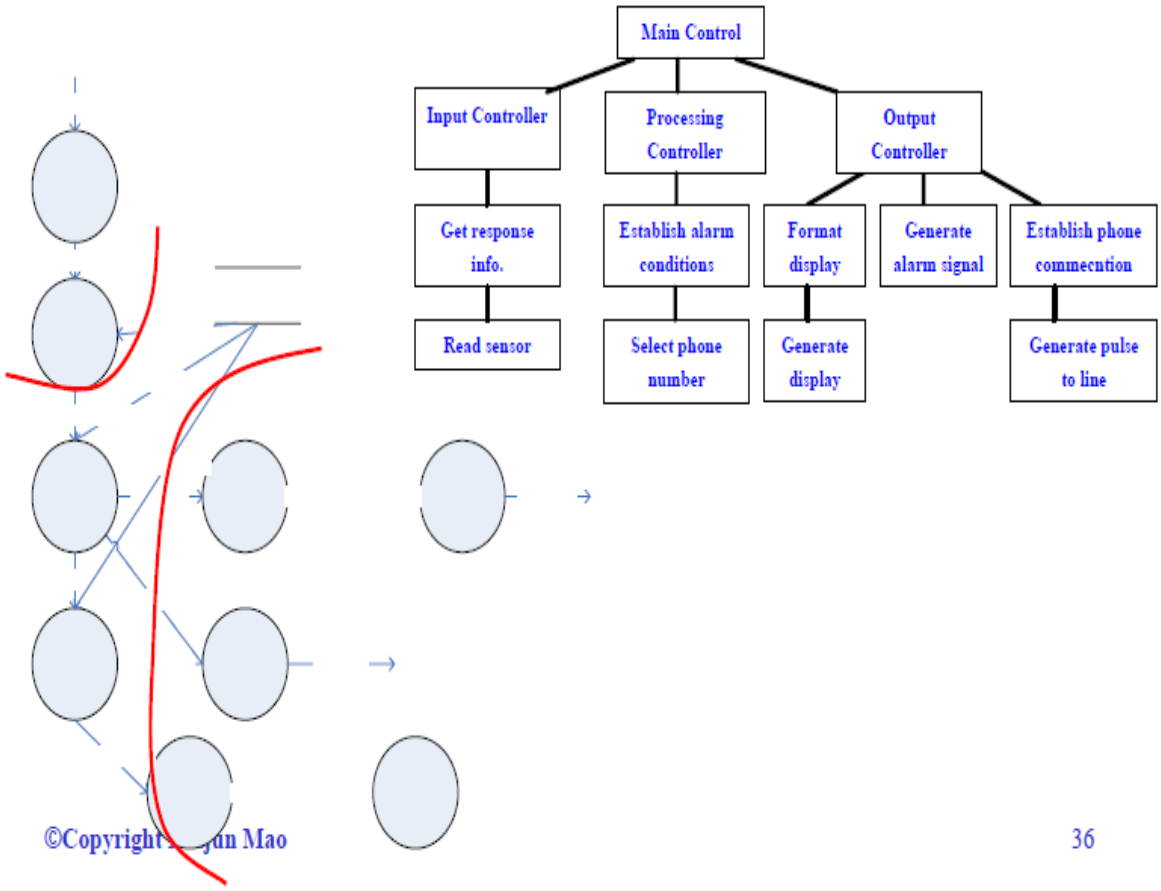
Step 6. Perform second-level Factoring

- Mapping individual transforms of a DFD into aappropriate modules with the architecture
- Approach
- Beginning at the transform center boundary and moving outward along incoming and then outgoingpaths, transforms are mapped into sub-ordinate levels of the software architecture, transforms aremapped into sub-ordinate levels of the software structure

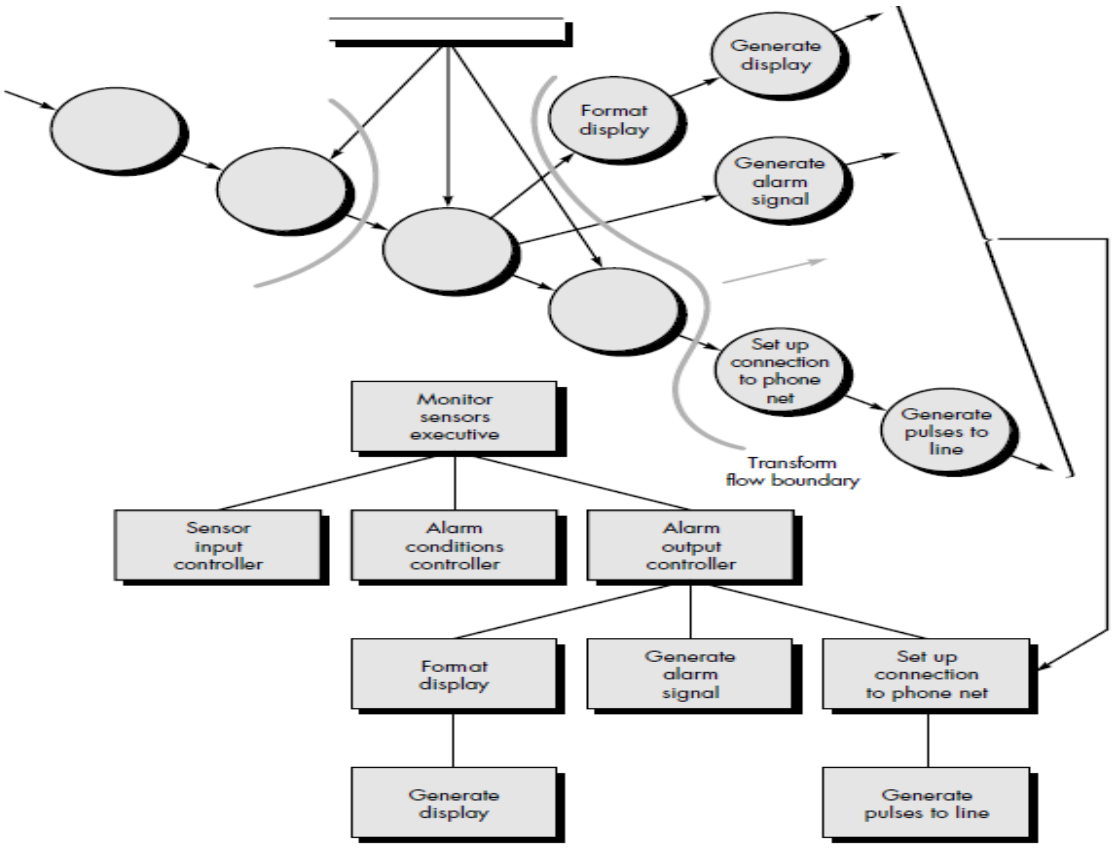
second-level Factoring



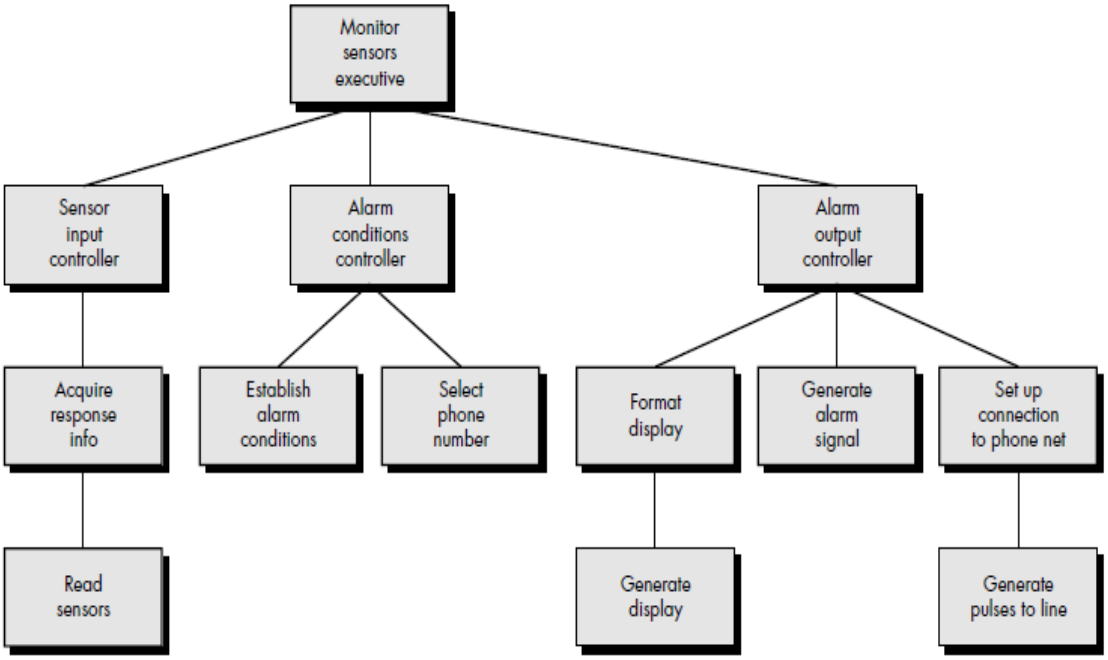
second-level Factoring



Second level factoring of monitor sensor



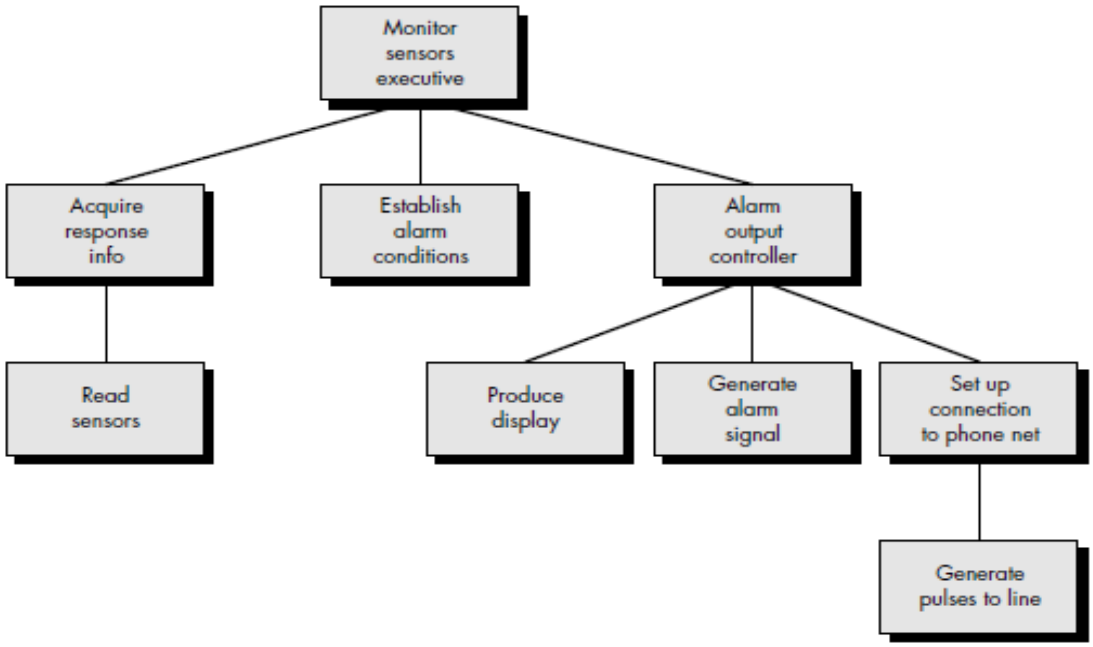
First iteration program structure for monitor sensor



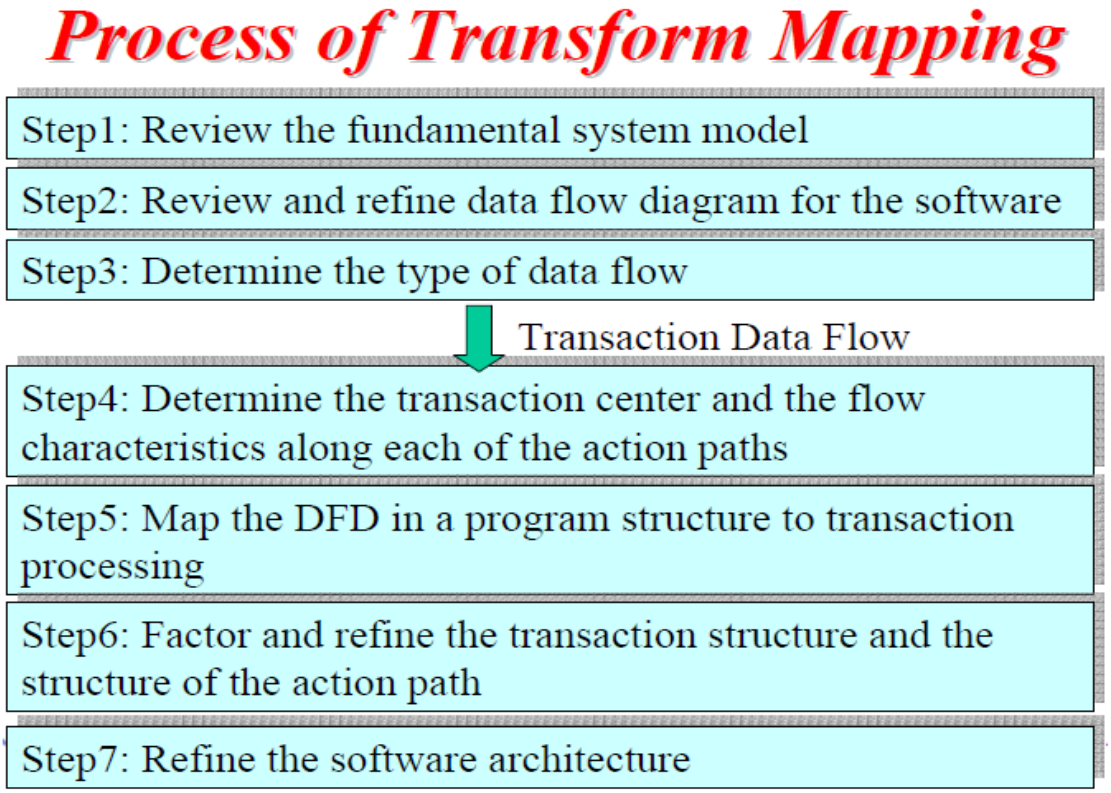
Refine the software Architecture

- Applying principles of “modular” .
- Components are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling.
- A good structure can be implemented without any difficulty, tested without confusion, and maintained without grief.

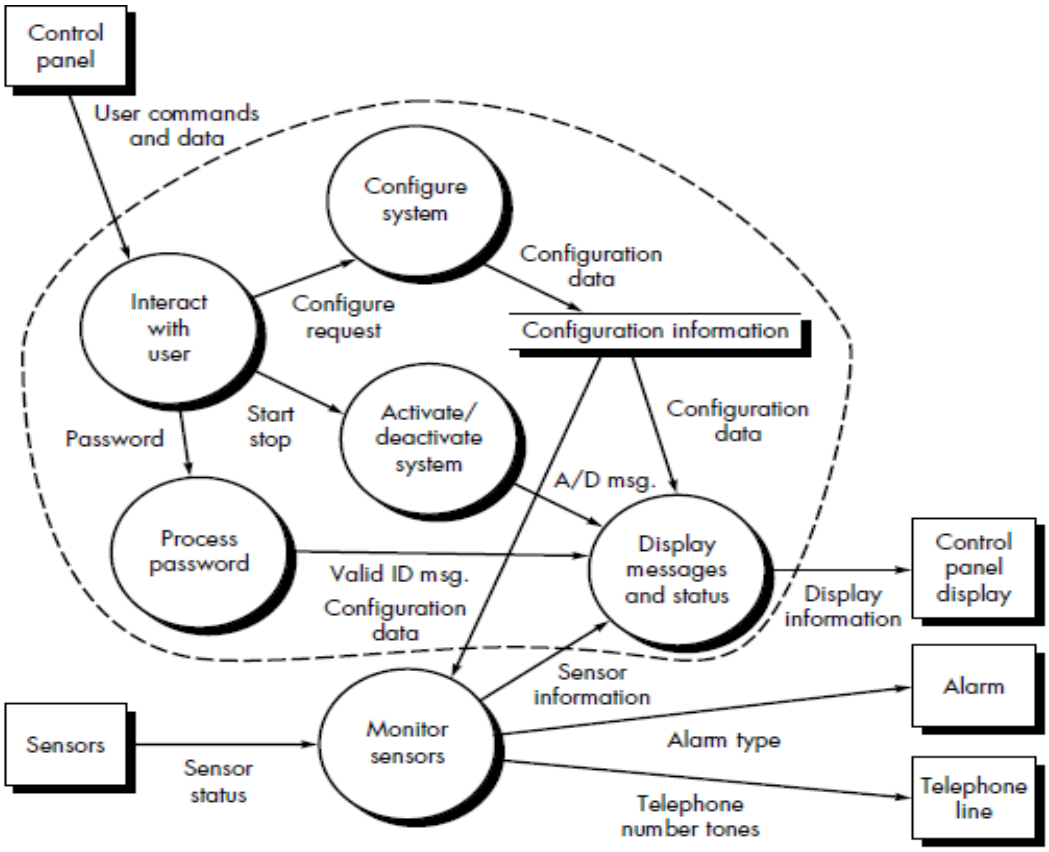
Refined program structure



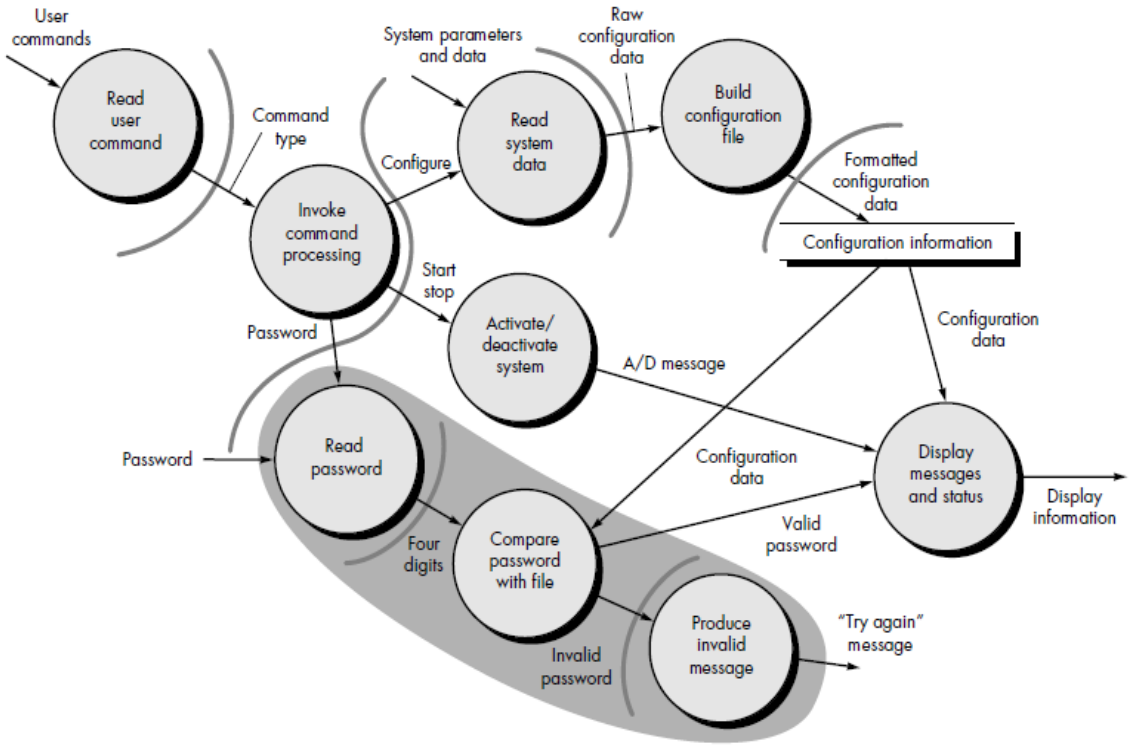
Transform mapping


Process of Transform Mapping
Step1: Review the fundamental system model
Step2: Review and refine data flow diagram for the software
Step3: Determine the type of data flow
Transaction Data Flow
Step4: Determine the transaction center and the flow characteristics along each of the action paths
Step5: Map the DFD in a program structure to transaction processing
Step6: Factor and refine the transaction structure and the structure of the action path
Step7: Refine the software architecture

Level 1 DFD



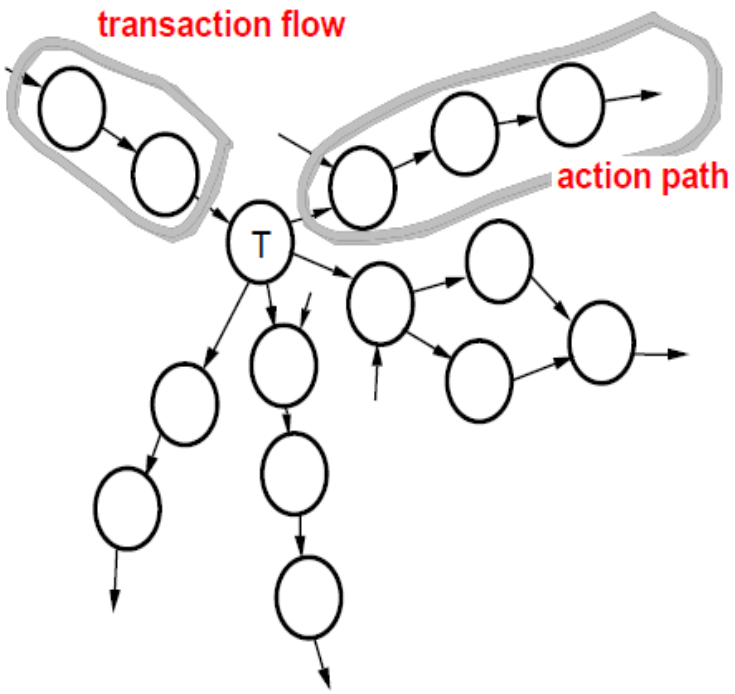
Level 2 DFD for user interaction sub system



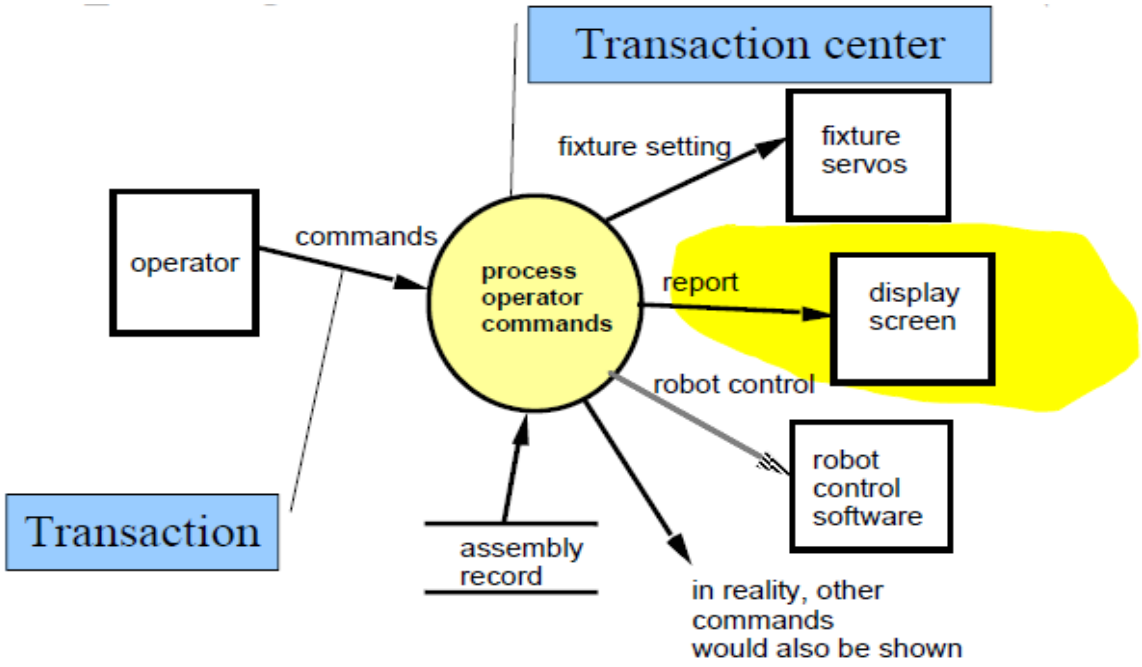
Step 3. Determine whether the DFD has transform or transaction flow characteristics.

- Steps 1, 2, and 3 are identical to corresponding steps in transform mapping.
- The DFD shown in Figure has a classic transaction flow characteristic. However, flow along two of the action paths emanating from the invoke command processing bubble appears to have transform flow characteristics.
- Flow boundaries must be established for both flow types.

Sample transaction DFD



Sample transaction DFD

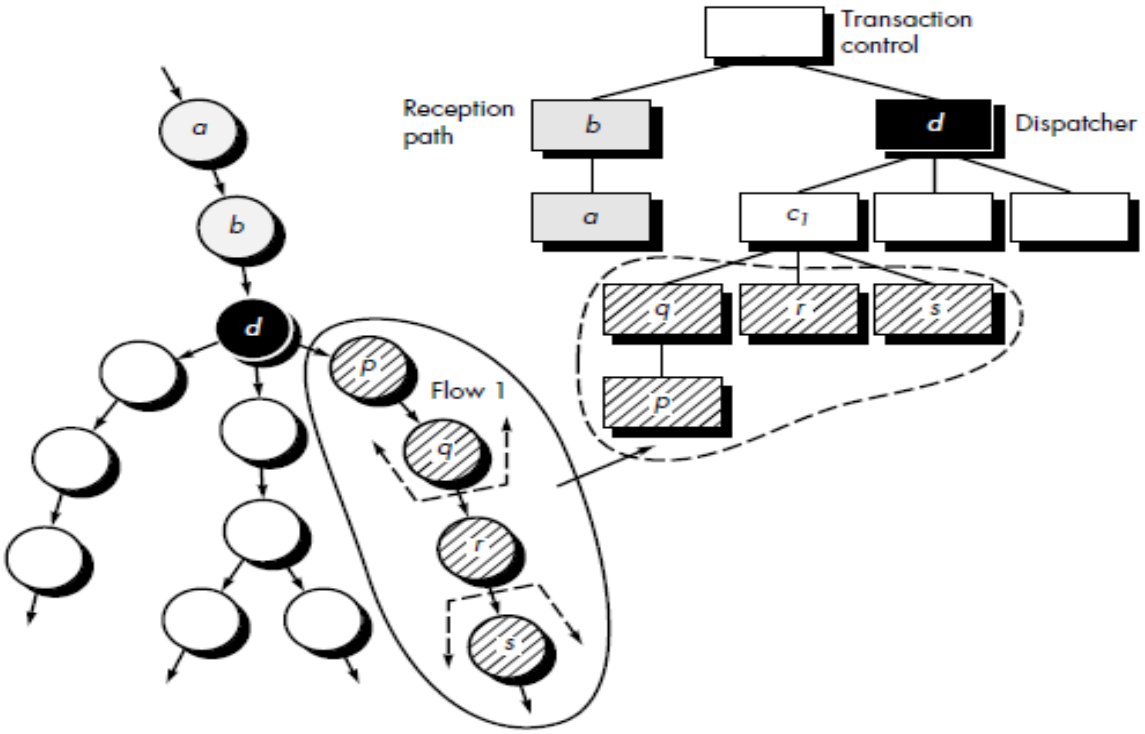


Step 4:Identify the transaction centre and action path

- Three parts of transaction DFD
  - Input flow
  - Transaction center

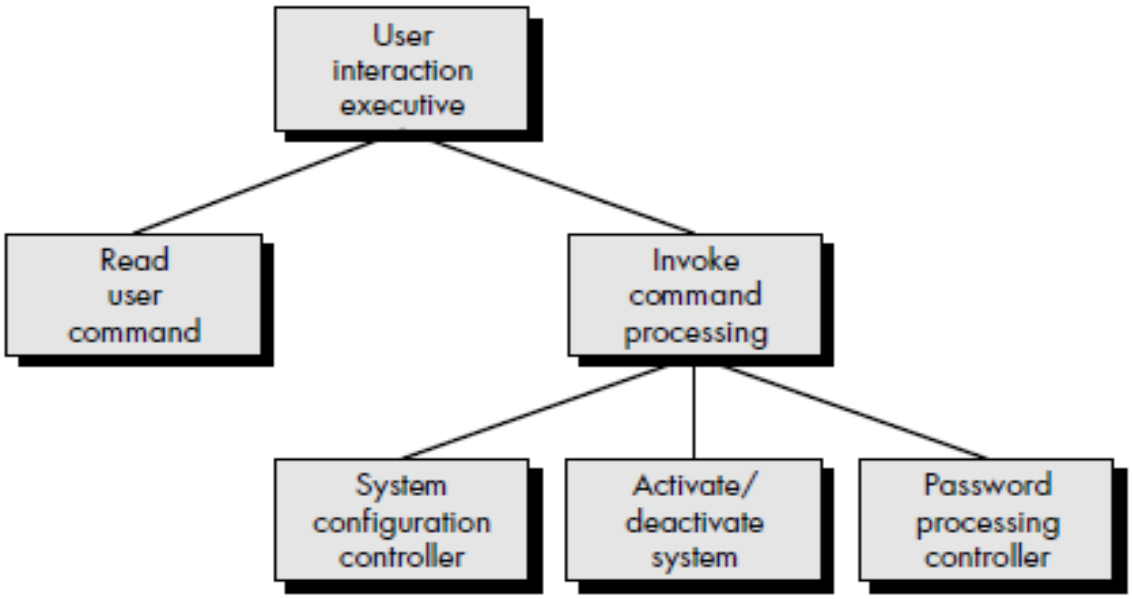
- Action path
- Evaluate the flow characteristics of each action path
  - Transaction flow or transform flow
- Each action path must be evaluated for its individual flow characteristic. For example, the "password" path has transform characteristics. Incoming, transform, and outgoing flow are indicated with boundaries

Step 5: Map DFD in program structure

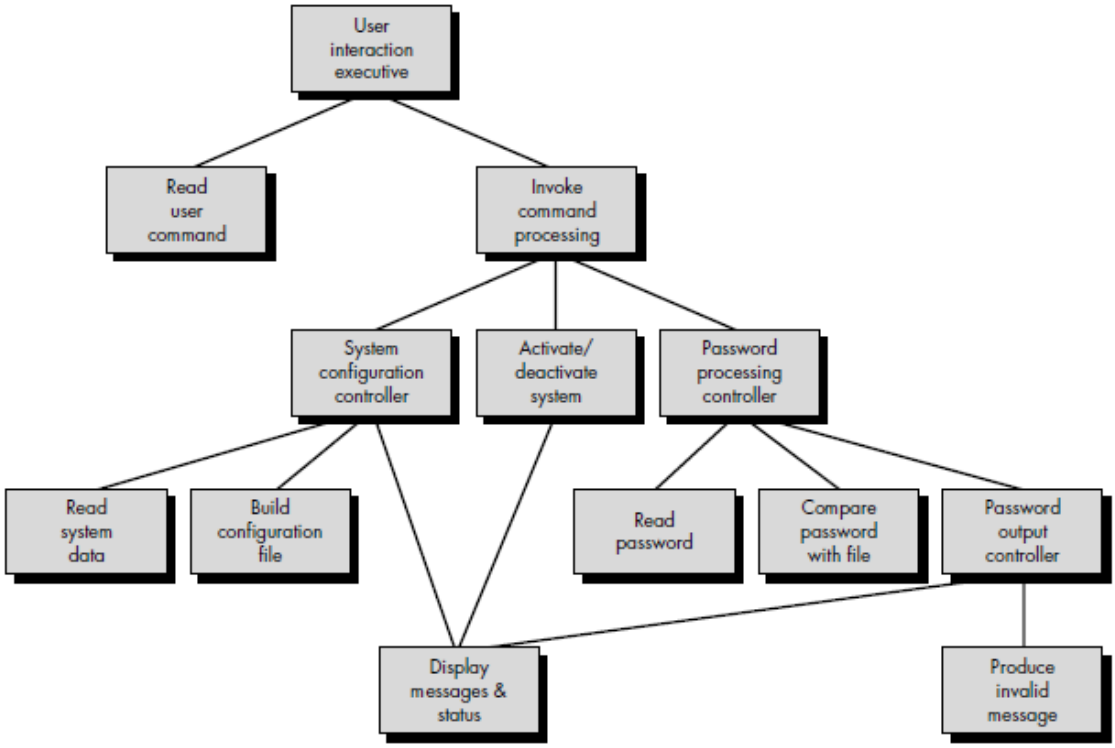


- Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch
- The structure of the incoming branch is developed in much the same way as transform mapping. Starting at the transaction center, bubbles along the incoming path are mapped into modules.
- The structure of the dispatch branch contains a dispatcher module that controls all subordinate action modules.
- Each action flow path of the DFD is mapped to a structure that corresponds to its specific flow characteristics.

First level factoring for user interaction sub system



Step 6:Factor and Refine the Transaction Structure and Each Action Path



Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.

3.7 User Interface Design

- The Golden Rules
  - Place the user in control.
  - Reduce the user's memory load.
  - Make the interface consistent

Typical Design Errors

- lack of consistency
- too much memorization
- no guidance / help
- no context sensitivity
- poor response
- Arcane/unfriendly

Place the User in Control

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

Reduce the User's Memory Load

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

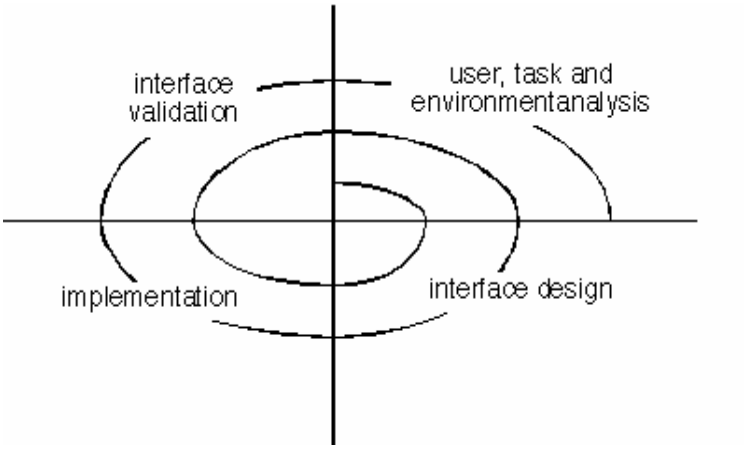
Make the Interface Consistent

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

User Interface Design Models

- **User model**—a profile of all end users of the system
- **Design model**—a design realization of the user model
- **Mental model** (system perception)—the user's mental image of what the interface is
- **Implementation model**—the interface "look and feel" coupled with supporting information that describe interface syntax and semantics

User Interface Design Process



Interface Analysis

Interface analysis means understanding

(1) the people (end-users) who will interact with the system through the interface;
(2) the tasks that end-users must perform to do their work,
(3) the content that is presented as part of the interface
(4) the environment in which these tasks will be conducted**.**

User Analysis

- Are users trained professionals, technician, 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 expressed a desire for 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 that is addressed by the system?
- Do users want to know about the technology the sits behind the interface

Task Analysis and Modeling

Answers the following questions for task analysis

- What work will the user perform in specific circumstances?
- What tasks and subtasks will be performed as the user does the work?
- 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?
- Use-cases define basic interaction
- Task elaboration refines interactive tasks
- Object elaboration identifies interface objects (classes)
- Workflow analysis defines how a work process is completed
- when several people (and roles) are involved

Interface Design Activities

The first interface design steps can be accomplished using the following approach:

- Establish the goals and intentions for each task.
- Map each goal and intention to a sequence of specific actions.
- Specify the action sequence of tasks and subtasks, also called a user scenario, as it will be executed at the interface level.
- Indicate the state of the system; that is, what does the interface look like at the time that a user scenario is performed?
- Define control mechanisms; that is, the objects and actions available to the user to alter the system state.
- Show how control mechanisms affect the state of the system.
- Indicate how the user interprets the state of the system from information provided through the interface.

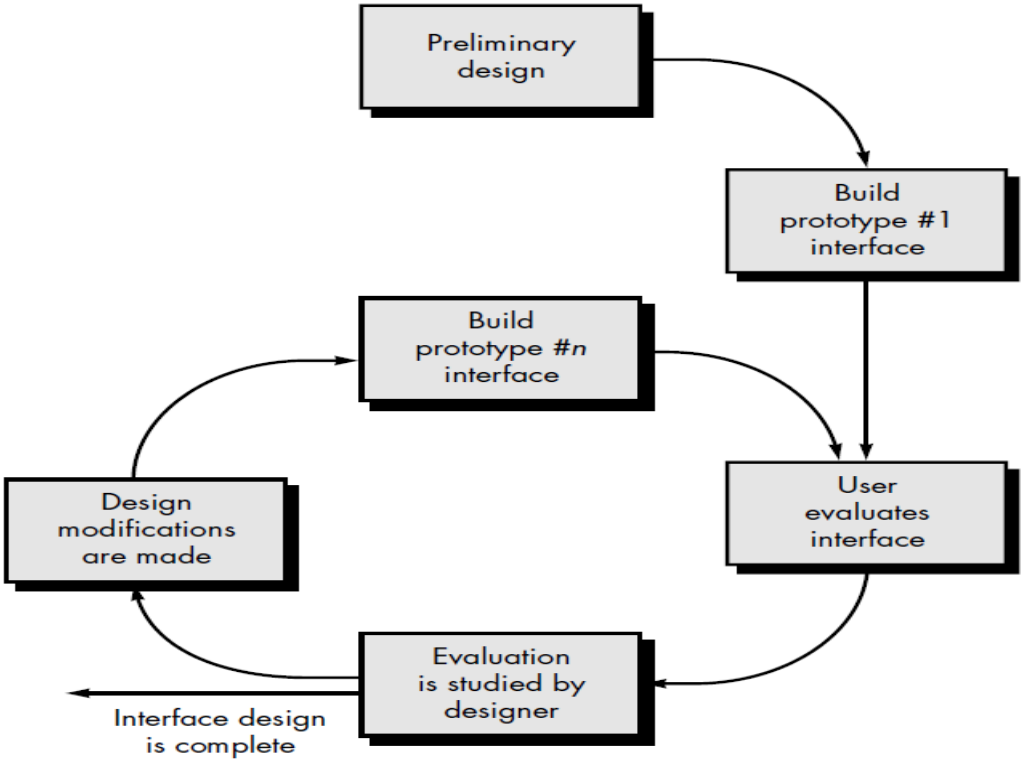
Implementation Tools

Implementation Tools are Called user- interface toolkits or user-interface development systems (UIDS), these tools provide components or objects that facilitate creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment.

A UIDS provides built-in mechanisms for

- managing input devices (such as a mouse or keyboard)
- validating user input
- handling errors and displaying error messages
- providing feedback (e.g., automatic input echo)
- providing help and prompts
- handling windows and fields, scrolling within windows
- establishing connections between application software and the interface
- insulating the application from interface management functions
- allowing the user to customize the interface

Design Evaluation



Reference Books:

1. Pressman, “Software Engineering and Application”, 6th Edition, Mcgraw International Edition, 2005.

2. Sommerville, ”Software Engineering”, 6th Edition, Pearson Education, 2000.