mapped to a structure that corresponds to its specific flow characteristics.

(Refer fig no: 10.20 pg no: 318 from textbook)

Step 6 - Factor and refine the transaction structure and the structure of each action path.

→ Each action path of the data flow diagram has its own information flow characteristics.

→ The action-path related "substructure" is developed using the preceeding design steps.

(Refer fig no: 10.21 & 10.22 pg no: 319 from textbook)

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

→ Module Independence, efficacy of implementation and test and maintainability must be carefully considered as structural modifications are proposed.

* Refining the Architectural Design

→ The software designer should be concerned with developing a representation of software that will meet all functional and performance requirements and merit acceptance based on design measures and heuristics.

→ Refinement of software architecture during early stages of design is to be encouraged. Structural simplicity often reflects both elegance and efficiency.

→ Design refinement should strive for the smallest number of components that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

USER INTERFACE DESIGN — Interaction between the system & user.

INTERFACE ANALYSIS

"you better understand the problem before you attempt to design a solution."

→ Understanding the problem means,

* Understanding the people who will interact with the system through the interface.

* Understanding the tasks that end-users must perform to do their work.

* Understanding the content that is presented as part of the interface.

* Understanding the environment in which these tasks will be conducted.

* User Analysis

→ A designer can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system.

→ Flow dependencies represent dependence relationships between producers and consumers of resources.

→ Constrained dependencies represent constraints on the relative flow of control among a set of activities.

* Architectural Description Languages (ADL)

→ ADL provides a semantics and syntax for describing a software architecture.

→ It should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks and represent interfaces between components.

MAPPING DATA FLOW INTO A SOFTWARE ARCHITECTURE.

→ 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.

* Transform Flow

→ Information must enter and exit software in an "external world" form.

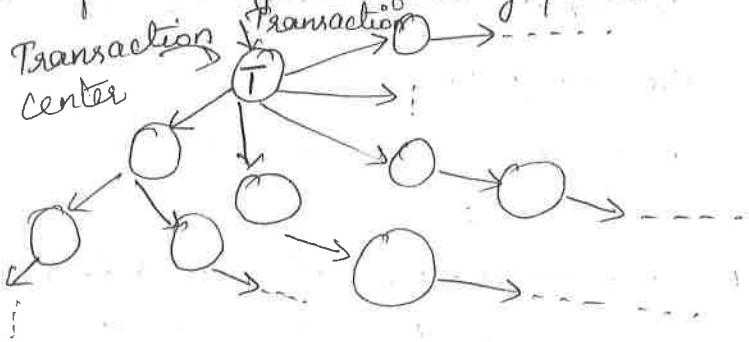
→ Informations enters the system along paths that transform external data into an internal form. These paths are identified as "incoming flow".

→ At the kernel of the software, a transition occurs. Incoming data are passed through a transform center and begin to move along paths that now lead "out" of the software. Data moving along these paths are called outgoing flow.

→ The overall flow of data occurs in a sequential manner and flow follows "straight line" paths. When a segment of a data flow diagram exhibits these characteristics, transform flow is present.

*Transaction flow

→Information flow is often characterized by a single data item, called a transaction, that triggers other data flow along one of many paths.



→The transaction is evaluated and, based on its value, flow along one of many action paths is ~~instead~~ initiated.

→The hub of information flow from which many action paths emanate is called a transaction ~~center~~ center.

* 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.

(Refer fig: 10.11 pg no: 309 from textbook)

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

→Information obtained from the analysis models is refined to produce greater detail.

* Are users experts in the subject matter that is addressed by the system?
* Do users want to know about the technology that sits behind the interface?
* Task Analysis and Modeling

→ The goal of task analysis is to answer the following questions:

* 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

→ The usecase is developed to show how an end-user performs some specific work-related task.

→ It is written in an informal style.

→ From usecases, software engineer can extract tasks, objects and the overall flow of the interaction.

→ Additional features of the system that would please the interior designer can also be conceived.

Task elaboration

→ It is a mechanism for refining the processing tasks that are required for software to accomplish some

desired function.

→ Task analysis can be applied in two ways.

→ 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 & set of tasks that are implemented in the context of the user interface.

⇒ Alternatively, the human engineer can study an existing specification for a computer-based solution and derive a set of user tasks that will accomodate the user model, the design model and the system perception.

→ The tasks are defined and classified and are finally refined.

Object elaboration

→ Rather than focusing on the tasks that a user must perform, the software engineer examines the use-case and other information obtained from the user and 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.

→ The user interface analysis model would

not provide a literal implementation for the Operations. However, as the design is elaborated, the details of each Operation are defined.

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 enable the interface designer to recognize three key interface characteristics:

(1) Each user implements different tasks via the interface.

(2) The interface design must accomodate access to and display of information from secondary information sources.

(3) Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration.

(Refer fig no:12.2 pg no 371 from textbook)

Hierarchical Representation

→Once workflow has been established, a task hierarchy can be defined for each user type.

→The hierarchy is derived by a stepwise elaboration of each task identified for the user.

Eg: Request that a prescription be refilled

* Provide identifying information

* Specify name

*Specify user id
*Specify PIN and password. } // subtasks
*Specify prescription number.
*Specify date refill is required.

*Analysis & Display Content

→ The user tasks identified in the preceding section lead to the presentation of a variety of different types of content.

→ For modern applications, display content can range from character-based reports, graphical displays or specialized information.

→ The analysis modeling techniques identify the output data objects that are produced by an application.

→ These data objects may be,

(1) Generated by components in other parts of the application.
(2) Acquired from data stored in a database that is accessible from the application.
(3) Transmitted from systems external to the application.

→ The requirements for content presentation are established by answering to the following queries:

(Refer the questions (8 points) from pg no: 372)

*Analysis of the Work Environment

→ The interface designer may be constrained by factors that mitigate against ease of use.

→ In addition to physical environmental factors, the work place & culture plays a vital role.

* Will System interaction be measured in some manner,
* Will two or more people have to share information before an input can be provided,
* How will support be provided to users of the System,

→ These related questions should be answered before the interface design commences.

INTERFACE DESIGN STEPS

→ Interface design is an iterative process.

→ It occurs a number of times, each elaborating and refining information developed in the preceding step.

→ User interface design models suggests the following steps:

* Using information developed during interface analysis, define interface objects and actions.
* Define events that will cause the state of the user interface to change.
* Depict each interface state as it will

* Indicate how the user interprets the state of the system from information provided through the interface

→ The designer must,

* Always follow the golden Rules

Place the user in control | Reduce the user's memory load. | Make the interface consistent.

* Model how the interface will be implemented.

* Consider the environment that will be used.

* Applying Interface Design steps

→ A description of a use-case is written. Nouns (objects) and Verbs (actions) are isolated to create a list of objects and actions.

→ Once the objects and actions have been defined and elaborated iteratively, they are categorized by type.

→ Target, Source and application objects are identified. A source object is dragged and dropped onto a target object.

→ An application object represents application-specific data that are not directly manipulated as part of screen interaction.

→ When the designer is satisfied that all important objects and actions have been defined, screen layout is performed. Screen layout is an

interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows and definition of major and minor menu items is conducted.

* User Interface design Patterns

→ A design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem.

→ Each patterns includes design classes, attributes, operations and interfaces.

* Design Issues

→ the design issues are:

* System response time
* User help facilities
* Error Information handling
* Command labeling.

→ It is far better to establish each as a design issue to be considered at the begining of software design, when changes are easy and costs are low.

Response Time

→ System response time is measured from the point at which the user performs some control action until the software responds with the desired output or action.

→ System response time has two important characteristics: length and variability.

→ If system response is too long, user frustration and stress is the inevitable result.

→ Variability refers to the deviation from average response time. Low variability enables the user to establish an interaction rhythm, even if the response time is relatively long.

#Help facilitees

→ Modern software provides on-line help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

→ The design issues to be addressed are:

* Will help be available for all system functions and at all times during system interaction?
* How will the user request help?
* How will help be represented?
* How will the user return to normal interaction?
* How will help information be structured?

#Error Handling

→ Error messages and warnings impart useless or misleading information and serve only to increase user frustration.

#Internationalization

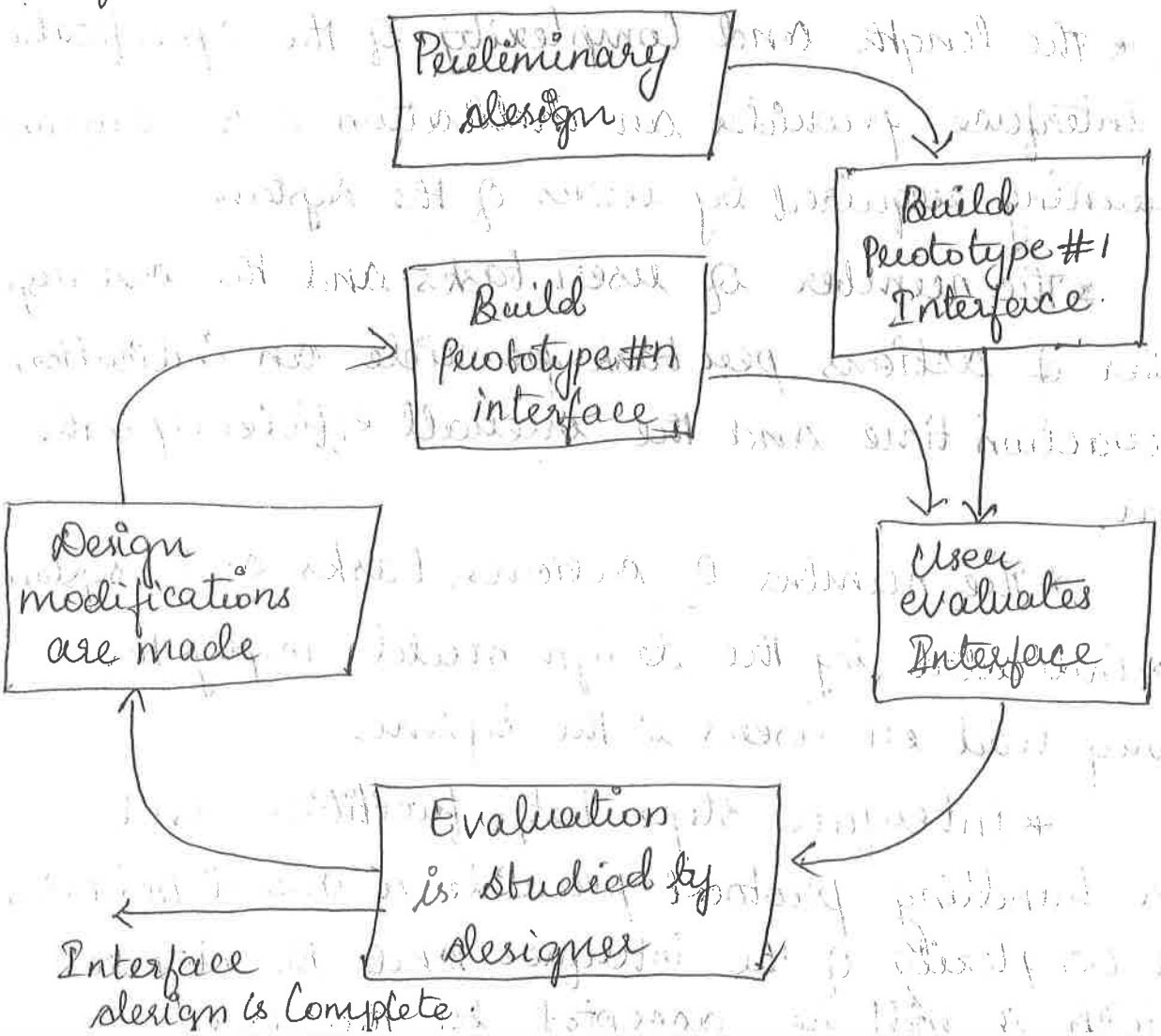
→ Use Interfaces should be designed to accomodate a generic core of functionality that can be delivered to all who use the software.

→ Localization features enable the interface to be customized for a specific market.

→ A variety of internationalization guidelines addresses broad design issues and discrete implementation issues.

* Design Evaluation

→ The user interface evaluation cycle takes the form as shown below:



→ 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 commands comments about the efficacy of the interface.

→ Design modifications are made based on the 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:

* The length and complexity of the specification and interface provides an indication of the amount of learning required by users of the system.

* The number of user tasks and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.

* The number of actions, tasks and system states indicated by the design model imply the memory load on users of the system.

* Interface style, 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.

→ The error message provides no real indication of what went wrong or where to look to get additional information.

→ 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 from the error.
* The message should indicate any negative consequences of the error so that the user can check to ensure that they have not occurred.
* The message should be accompanied by an audible or visual cue.
* The messages should be nonjudgemental. That is, the wording should never place blame on the user.

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

Menu and Command labeling

→ The typed command was once the most common mode of interaction between users and system software and was commonly used for applications

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

* Will every menu option have a corresponding command?

* What form will commands take?

* 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 who may be physically challenged is an imperative for moral, legal and business reasons.

→ A variety of accessibility guidelines provide detailed suggestions for designing interfaces that achieve varying levels of accessibility.

→ Others provide specific guidelines for "assistive technology" that addresses the needs of those with visual, hearing, mobility, speech and learning impairments.

ARCHITECTURAL PATTERNS

→ Architectural patterns for software define a specific approach for handling some behavioural characteristics of the system.

* Concurrency

→ Many applications must handle multiple tasks in a manner that simulates parallelism.

→ There are a number of different ways in which an application can handle concurrency, and each can be presented by a different architectural pattern.

Eg1: One approach is to use an operating system process management pattern - that provides built-in OS features that allow components to execute concurrently.

Eg2: Another approach might be to define a task scheduler pattern which contains a set of active objects that each contains a tick() operation.

The scheduler periodically invokes tick() for each object, which then performs the functions it must perform before returning control back to the scheduler, which then invokes the tick() operation for the next concurrent object.

* Persistence

→ Persistent data are stored in a database or file and may be read or modified by other processes at a later time.

→ In object-oriented environments, the values of all object's attributes, the general state of the object and other supplementary informations are stored for future

retrieval and use.

→ There are two architectural patterns to achieve persistence:

* Database Management System pattern that applies the storage and retrieval capability of a DBMS to the application architecture.

* Application level persistence pattern that builds persistence features into the application architecture.

* Distribution

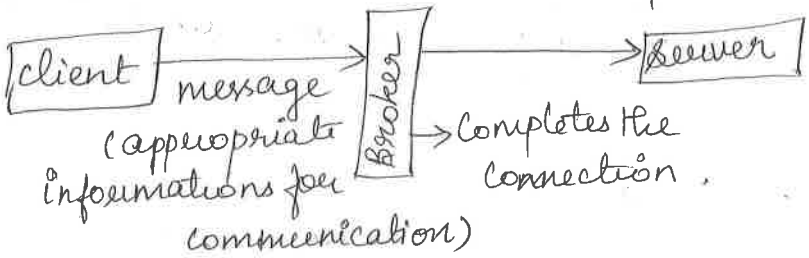
→ The distribution problem addresses the manner in which systems or components within systems communicate with one another in a distributed environment.

→ There are two elements to this problem:

(a) the way in which entities connect to one another.

(b) the nature of the communication that occurs.

→ The most common architectural pattern established to address the distribution problem is the "broker" pattern. A "broker" acts as a "middle man" between the client and server component.



Eg: CORBA.

Organization and Refinement

→ A set of design criteria is established that can be used to assess an architectural design.

(*) 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 the components?

→ What is the control topology?

→ Is control synchronized or do components operate asynchronously?

(*) Data

→ How are data communicated between components?

→ Is the flow of data continuous?

→ What is the mode of data transfer?

→ What is the role of data components?

→ How functional and data components communicate with each other?

→ How do data and control interact within the system?

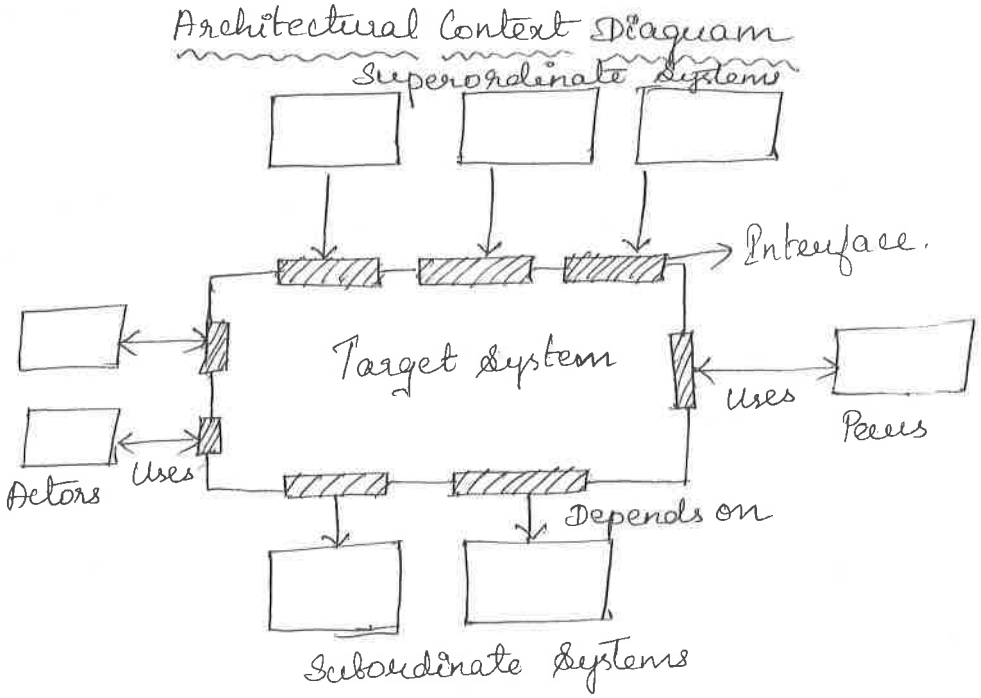
ARCHITECTURAL DESIGN:

*Representing the System in Context

→ A system context diagram accomplishes the requirements by representing the flow of information into and out of the system, the user interface and relevant support processing.

→ At the architectural design level, a software architect uses an architectural context diagram (ACD)

To model the manner in which software interacts with external entities.



* Superordinate Systems – These systems use the target system as part of some higher level processing scheme.

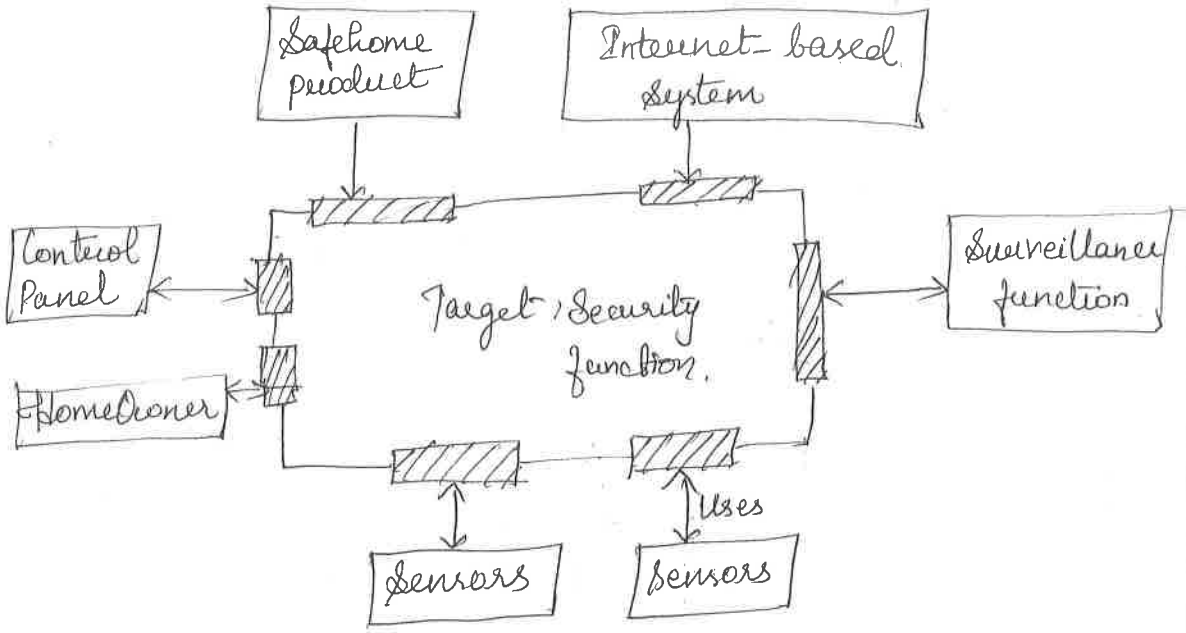
* Subordinate Systems – These systems are used by the target system and provide data or processing that are necessary to complete target system functionality.

* Peer-level Systems – These systems interact on a peer-to-peer basis. (Information is either produced or consumed by the peers and the target systems).

* Actors – These entities interact with the target system by producing or consuming information that is necessary for requisite processing.

→ Each of these external entities communicates with the target system by producing or consuming information through "Interface".

Eg: Architectural context diagram for the SafeHome security function.



* Defining Archetypes

→ An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system.

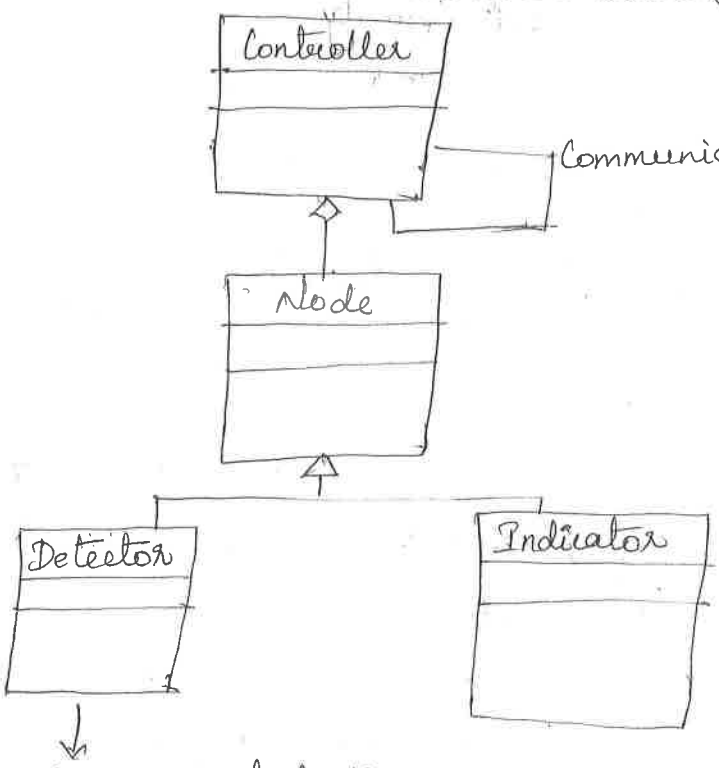
→ Archetypes can be derived by examining the analysis classes defined as part of the analysis model.

→ The following archetypes are defined for SafeHome home security function:

* Node – Represents a cohesive collection of input and output elements of the home security function.
* Detector – An abstraction that encompasses all sensing equipment that feeds information into the target system.
* Indicator – An abstraction that represents all mechanisms for indicating 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.

UML relationships for SafeHome security function archetypes



* Refining the architecture into components

→ The analysis classes represent the entities with the application domain that must be addressed within the software architecture.

→ Hence, the application domain is one source for the derivation and refinement of components.

→ The other source is the infrastructure domain in which the architecture can accomodate many

infrastructure components that enable application components but have no business connection to the application domain.

→ The interfaces depicted in the architecture context diagram imply one or more specialized components that process the data that flow across the interface.

Eg: Top-level components of SafeHome home security function:

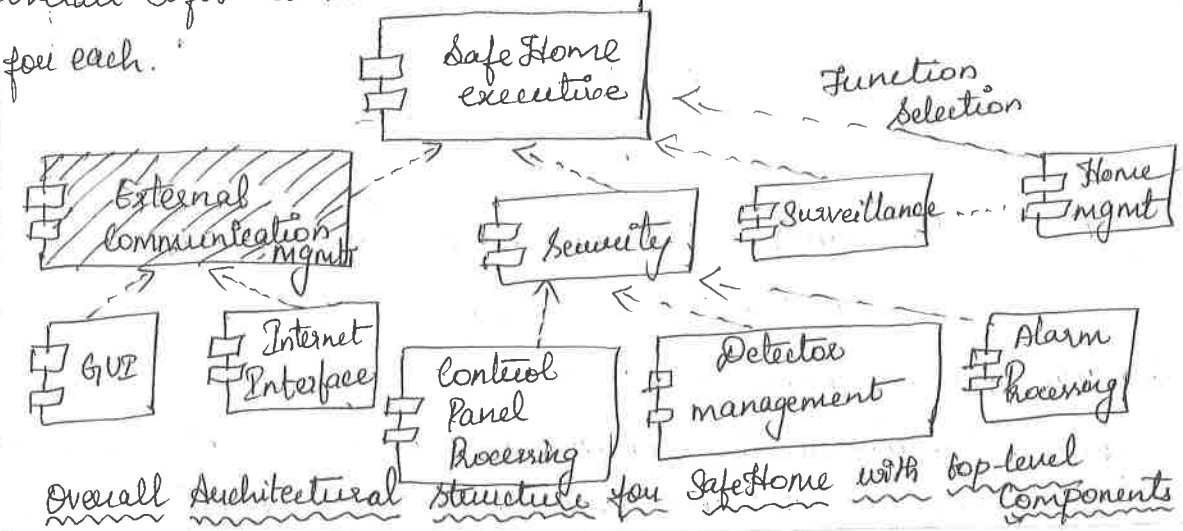
* External Communication Management - coordinates communication of the security function with external entities.

* 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.

→ Each of these top-level components would have to be elaborated iteratively and then positioned within the overall SafeHome architecture. Design classes would be defined for each.



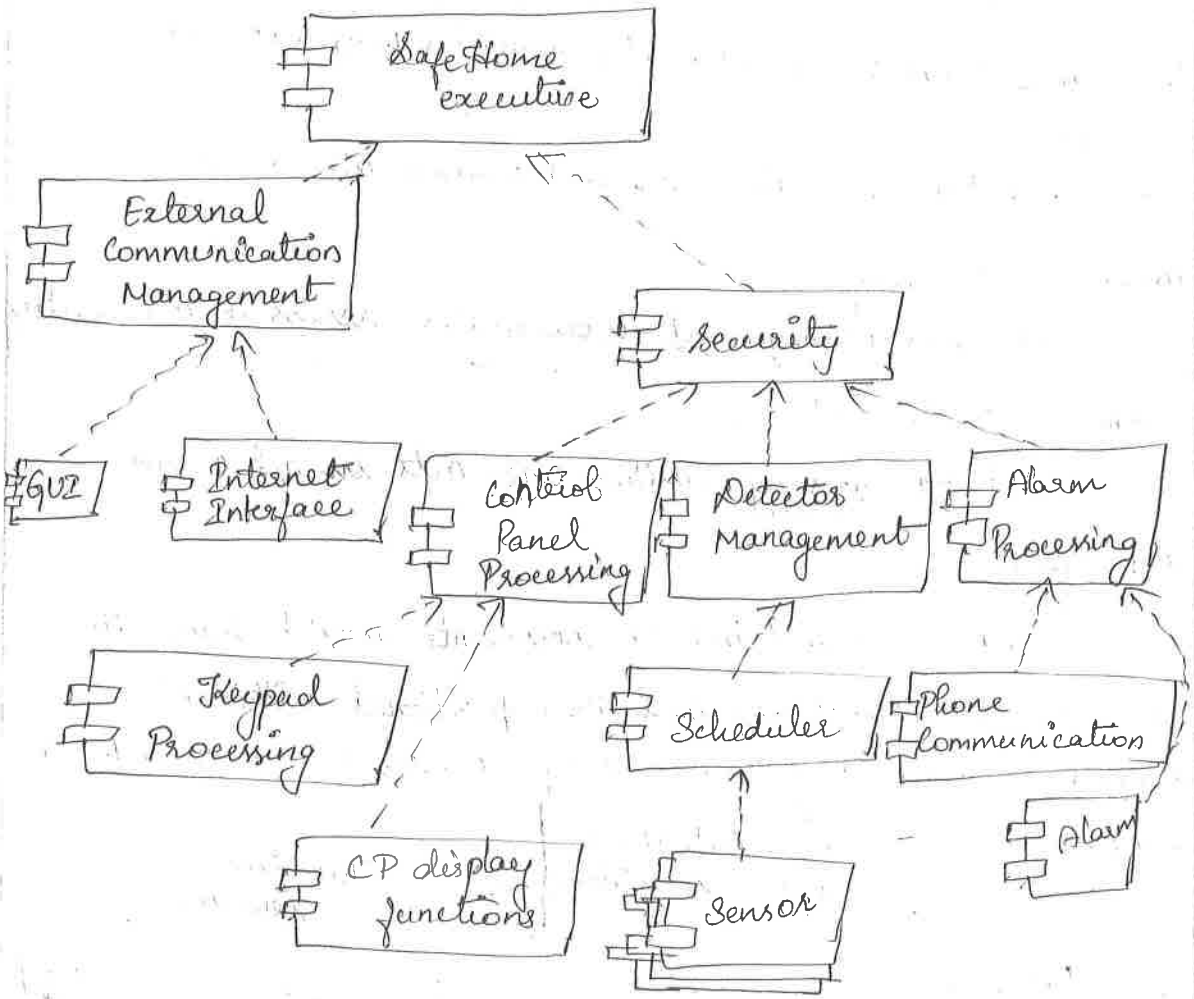
Overall Architectural structure for SafeHome with top-level components

*Describing Instantiations of the System

→ For further refinement, an actual instantiation of the architecture is developed.

→ Architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

→ Components are refined to show additional detail.



An Instantiation of the Security function with Component elaboration.

Eg: the detector management component interacts with a scheduler infrastructure component that implements "concurrent" polling of each sensor object used by the security system.

COMPONENT-LEVEL DESIGN:

* What is a component?

→ A component is defined as a modular, deployable and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.

→ Each component will communicate and collaborate with other components and entities of the software.

* DESIGNING CLASS-BASED COMPONENTS

→ In object oriented ~~software~~ engineering approach, component-level design focuses on the elaboration of analysis classes and the definition and refinement of infrastructure classes.

→ The detailed description of the attributes, operations and interfaces used by the classes is the design detail required as a precursor to the construction activity.

* Basic Design Principles

→ The motivation of these principles is to create designs that are more amenable to change and to reduce the propagation of side effects when changes do occur.

→ These principles are used to ~~design~~ guide the designer as each software component is developed.

→ Open-closed Principle (OCP)

*"A module 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 modifications to the component itself.

*To accomplish this, the designer creates abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

(Refer fig. 11.4 pg.no: 331 from the text book)

→ Liskov Substitution Principle (LSP):

*"Subclasses should be substitutable for their base classes".

*The design principle suggests that a component that uses a base class should continue to function properly if a class derived from a base class is passed to the component instead.

*A "contract" is a precondition that must be true before the component uses a base class and a post condition that should be true after the component uses a base class.

*When a designer creates derived classes, they must also conform to the pre and post-conditions.

→ Dependency Inversion Principle (DIP)

* "Depend on abstractions. Do not depend on concretions"

* Abstractions are the place where a design can be extended without great complication.

* The more a component depends on other concrete components, the more difficult it will be to extend.

→ Interface Segregation Principle (ISP).

* "Many client-specific Interfaces are better than one general purpose interface".

* It suggests that the designer should create a specialized interface to serve each major category of clients.

* Only those operations that are relevant to a particular category of clients should be specified in each of the specialized Interfaces.

→ Although component-level design principles provide useful guidance, components themselves do not exist in a vacuum.

→ Individual components or classes are organized into subsystems or packages.

Additional Packaging Principles that are applicable to component-level design:

→ Release Reuse Equivalency Principle (REP)

* "The granule of reuse is the granule of

* The developer commands to construct a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version.

* Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.

→ Common Closure Principle (CCP)

* "classes that change together belong together"

* classes should be packaged cohesively.

* When classes are packaged as part of a design, they should address the same functional or behavioral area.

* When some characteristics of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

→ Common Reuse Principle (CRP)

Cohesion - function, Layer, communicational, Sequential, Procedural, Temporal, Utility

* "classes that aren't reused together should not be grouped together".

Coupling - Content, Common, Control, Stamp, Data, Routine call, type use, Inclusion/Import, External

* When one or more classes with a package changes, the release number of the package changes.

* All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operate without incident.

→ If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing. Hence, only classes that are reused together should be included within a package.

* Component-level Design Guidelines

→ The guidelines are as follows:

Components

* Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model.

* Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.

* It is also worthwhile to use stereotypes to help identify the nature of components at the detailed design level.

Interfaces

* Interfaces provide important information about communication and collaboration.

* It is recommended that,

(1) lollipop representation of an interface should be used in the more formal UML box and dashed arrow approach, when diagrams grow complex.

(2) for consistency, interfaces should flow from the left-hand side of the component box.

(3) Only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available.

* The above recommendations are intended to simplify the visual nature of UML component diagrams.

Dependencies and Inheritance

* For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom to top.

* Component Interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency.

* Cohesion

→ Cohesion 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.

Functional Cohesion – It occurs when a module performs one and only one computation and then return a result.

#Layer Cohesion – It occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

#Communicational Cohesion – All operations that access the same data are defined within one class.

Lower levels of cohesion

#Sequential Cohesion – Components or operations are grouped in a manner that allows the first to provide input to the next and so on.

#Procedural Cohesion – Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when there is no data passed between them.

#Temporal Cohesion – Operations that are performed to reflect a specific behaviour or state.

#Utility Cohesion – Components, classes or operations that exist within the same category but are otherwise unrelated are grouped together.

*Coupling

→ Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes become more interdependent, coupling increases. An important objective is to have low coupling.

Content Coupling - Occurs when one component "surreptitiously modifies data that is internal to another coupling". This violates information hiding.

Common Coupling - Occurs when a number of components all make use of a global variable. It 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 is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes.

Stamp Coupling - Occurs when class B is declared as a type for an argument of an Operation of class A. Modification becomes complex.

Data Coupling - Occurs when Operations pass long strings of data arguments. Testing & maintenance are more difficult.

Routine Call Coupling - Occurs when one Operation invokes another.

Type Use Coupling - Occurs when component A uses a data type defined in component B.

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. (eg. database capabilities)

TRADITIONAL COMPONENTS

→ Component level design is also called as procedural design.

→ After data, architectural and interface design, the component level design occurs.

→ The goal of component level design is to translate design model into Operational Software.

→ Graphical, tabular or text based notations are used to create a set of Structured programming constructs. It translates design model into Operational Software. each component into procedural design model.

→ Hence the work product of component level design is procedural design model.

Structured Programming

→ There are three constructs of structured programming:

(1) Sequence – a linear processing of statements

(2) Condition – facilitates to test the logical conditions.

(3) Repetition – Denotes the looping.

→ Advantages of structured programming constructs:

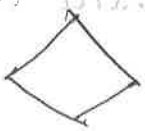
* Reduces program complexity.
* Enhances readability, testability and maintainability of the procedure.
* They are logical chunks that allow a reader to recognize procedural element from each programming module.

* Graphical Design Notations

→ The graphical notations are called flow chart.

→ Box → It is used for processing step.

↓ → Flow of Control → It is used to represent the flow of control from one construct to another.



→ Diamond → It is used to represent the ~~flow~~ conditions such as if-then-else or repeat until.

→ The programming constructs can be represented as follows:

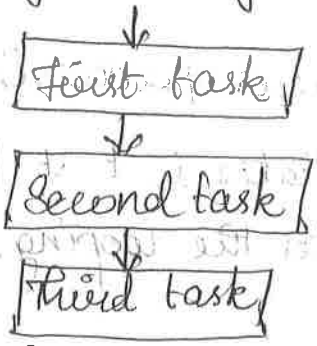


Fig. Sequence.

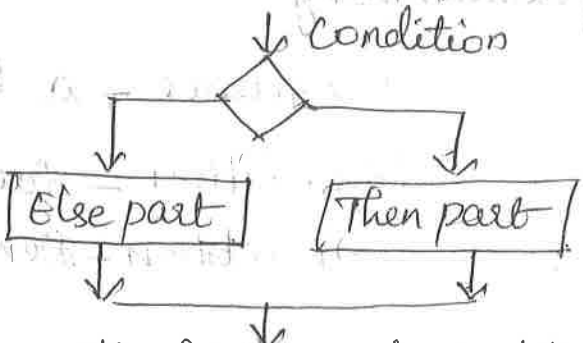
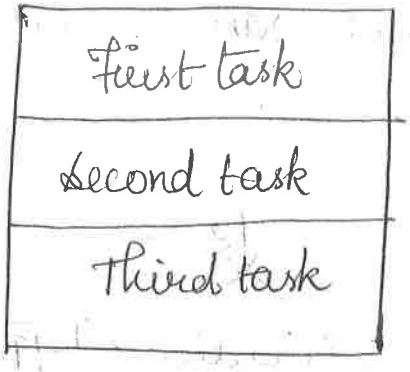
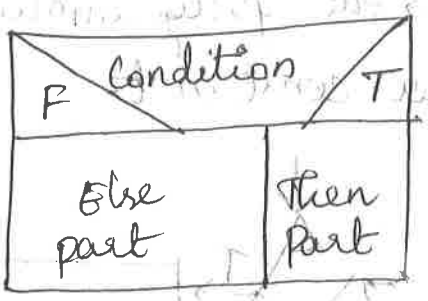


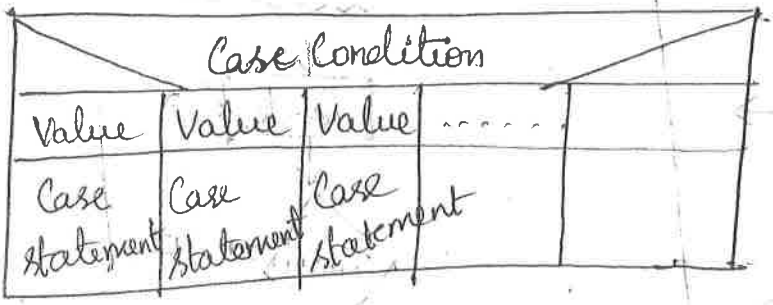
Fig. if-then-else condition



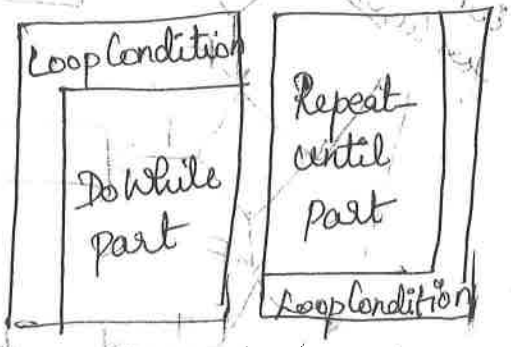
Sequence



If-then-else



Section



Repetition

*Tabular Design Notation

→ There are four sections:

* The first section is at the upper left corner. It consists of list of conditions.

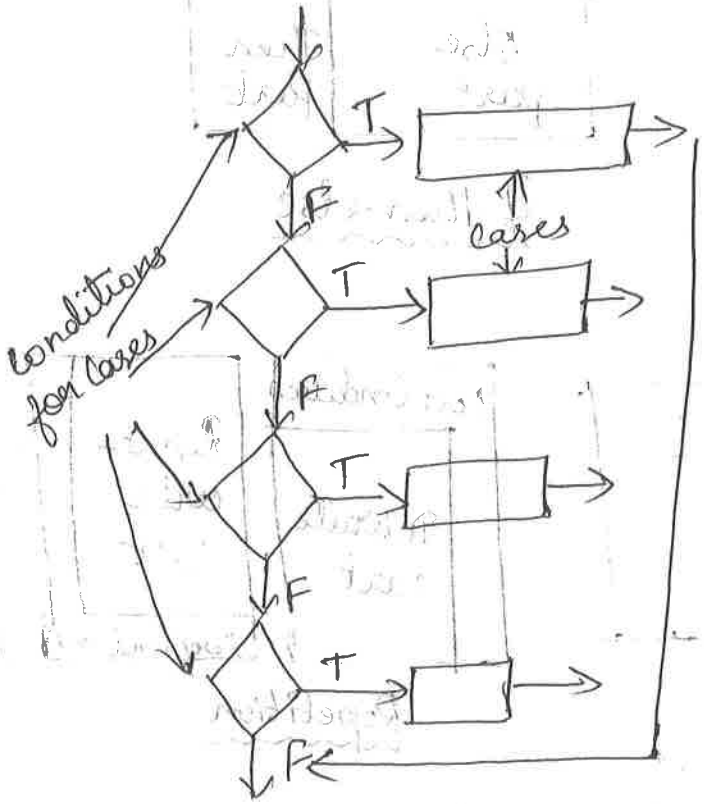
* The second section is the lower left hand corner. It consists of list of actions.

* The right hand portion is a matrix which represents the combination of conditions and actions. The combination of condition and actions together form processing rules for that particular procedure.

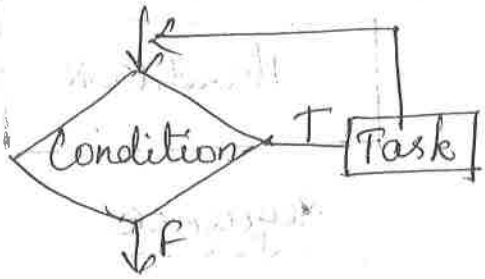
→ Steps to develop a decision table:

* List all the actions associated with particular procedure.

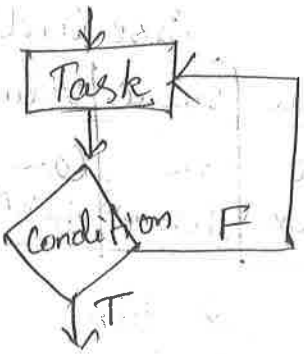
→ One programming construct can be within another construct:



Selection Condition



Do-While



Repetition

→ Another graphical notation is box diagram. It is known as NS chart or Nassi and Shneiderman chart.

→ characteristics:

* The scope of the programming constructs such as repetition, if-then-else is well defined.

* Arbitrary transfer of the control is not possible using this notation.

* Recursion can be represented conveniently.

* The scope of local and global data can be defined systhematically.

* List all the conditions associated with particular procedure.

* Associate each condition with each corresponding action. It must be represented in the table.

* Create the processing rule indicating that particular action exists on particular condition.

Eg:

Conditions	1	2	3		n
Condition 1	✓	✓			
Condition 2		✓	✓		
Actions					
Action 1	✓		✓		
Action 2	✓	✓			

* Program Design Language (PDL)

→ PDL is also called as pseudo code or structured English. It is used as a generic reference for design language.

→ PDL is not compiled. It is used to translate the design into the programming language.

→ A basic PDL syntax should posses the

procedure sequences, data declaration, condition constructs, repetition constructs and I/O constructs.

Eg: Searching for name "John" from the table.

```
Search (table, number of items)
Set count to zero
Read first item from table
DO FOR count is <= number of items
  IF table.name = "John"
    RETURN table.index
  ELSE
    count = count+1;
  Read Next item from table
  ENDIF
 END FOR
END;
```

DESIGN HEURISTIC

→ The program structure can be manipulated according to the design heuristics as shown below:

* Evaluate the first iteration of the program structure to reduce the coupling and Improve cohesion. The module independency can be achieved either by exploding (obtaining two or more modules in the final stage) or imploding (combining the result of different modules) the modules.

* Attempt to minimize the structures with high fan-out and strive for fan-in as depth increases - Fan-out means number of immediate subordinates to the module and fan-in means number of immediate ancestors the module have.

* Keep scope of the effect of a module within the scope of control of that module - The decisions made in particular module 'a' should not affect the module 'b' which lies outside the scope of module 'a'.

* Evaluate the module interfaces to reduce complexity and redundancy and improve consistency - The module interface should simply pass the information and should be consistent with the module.

* Define module whose function is predictable but avoid modules that are too restrictive - Modules should be designed with simplified internal processing so that expected data can be produced as a result.

* Strive for controlled entry modules by avoiding pathological connections - Software Interfaces should be constrained and controlled so that it will become manageable. Pathological connection means many references or branches into the middle of a module.