UNIT- III

DESIGN ENGINEERING.

→ Software Design Process:

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

→ Characteristics of a Good design:

* The design must implement all of the explicit and implicit requirements.

* The design must be readable and understable.

* The design should provide a complete picture of the software addressing the data, functional and behavioral domains.

→ Quality Attributes: "FURPS" quality attributes represent a target for all software design.

* Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the delivered functions and the security of the system.

* Usability is assessed by overall aesthetics, consistency and documentation.

* Reliability is evaluated by measuring the frequency and severity of failure and MTTF (Mean-time-to-

*Performance is measured by processing speed, response time and efficiency.

*Supportability combines the ability to extend the program (extensibility), adaptability & serviceability which together is termed as "maintainability".

DESIGN CONCEPTS:

*Abstraction:

→ At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.

→ At lower levels of abstraction, a more detailed description of the solution is provided.

→ A procedural abstraction refers to a sequence of instructions that have a specific and limited function, but specific details are suppressed.

→ A data abstraction is a named collection of data that describes a data object.

*Architecture:

→ 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 by using:

*Structural models - Represents architecture as an organized collection of program components.

* Framework models - Increases the level of design abstraction.

* Dynamic models - Addresses the behavioral aspects of the program architecture.

* Process models - Focuses on the design of the business or technical process.

* Functional models - Represents the functional hierarchy of the system.

*Patterns:

(→ A design pattern describes a design structure that solves a particular design problem within a specific context)

→ The intent of the design pattern is to determine:

*whether the pattern is applicable to the current work.

*whether the pattern can be reused.

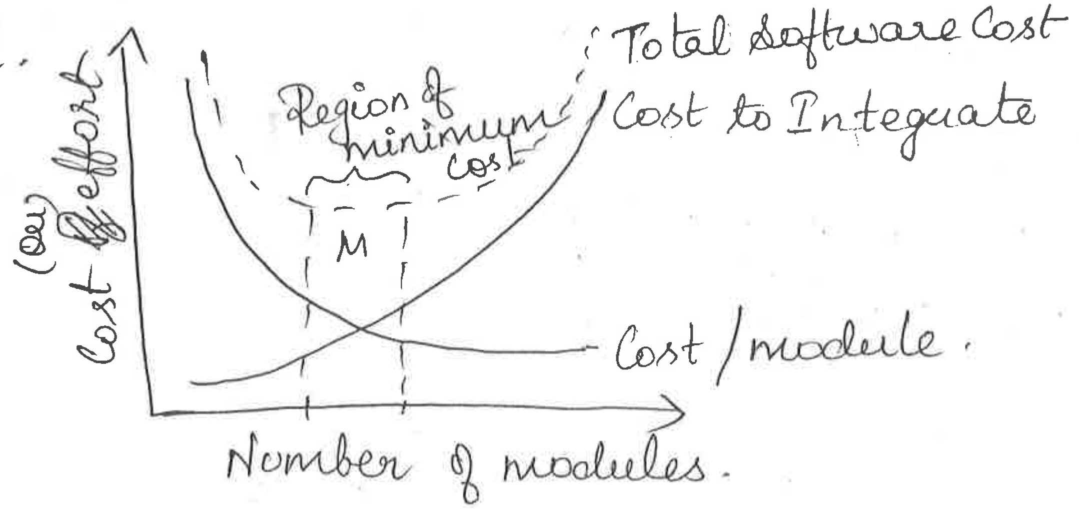
*whether the pattern can serve as a guide for developing a structurally different pattern.

*Modularity:

(→Software is divided into seperately named and addressable components, called modules that are integrated to satisfy problem requirements.

→ It leads to a "divide and conquer" strategy- it is easier to solve a complex problem when broken into manageable ~~pies~~ pieces.)

→ If the software is divided indefinitely, the effort required to develop it will become negligibley small.



→ Advantages:

* Development can be more easily planned.
* Software increments can be defined and delivere
* Changes can be more easily accomodated.
* Testing and debugging can be conducted more efficiently.
* Long-term maintenance can be conducted without serious side-effects.

* <u>Information Hiding</u>:

(→ Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information).

→ Hiding Implies the effective modularity that can be achieved by defining a set of independent modules that communicate with one another only that informati necessary to Achieve software function.

→ Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propoga to other locations within the software.

* Functional Independence:

→ It is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.

→ Independent modules is easier to develop ~~because~~ and maintain because error propogation is reduced and reusable modules are possible.

(→ Independence is assessed using two qualitative criteria:

* Coupling is an indication of the relative interdepende among modules.

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

(→ Low Coupling and High Cohesion is a good criteria for software design.)

* Refinement:

(→Refinement is a process of elaboration.

→It describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the data.)

→Refinement helps the designer to reveal low-level details as design progresses, whereas abstraction enables a designer to specify procedure and data and yet suppress low-level details.

→Hence, refinement and abstraction are complementary concepts.

*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 structures or any other design failure that can be corrected to yield a better design.

* <u>Design classes:</u>

(→ The software team must define a set of desig classes that,

* Refine the analyses classes by providing design detail that will enable the classes to be implemented.

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

→ <u>Different types</u> of design classes:

* <u>User Interface classes</u> - defines all abstractions that are necessary for human-computer interaction (HCI

* <u>Business domain classes</u> - Identifies the attributes and services that are required to implement some element of the business domain.

* <u>Process classes</u> - Implement lower-level business abstractions required to fully manage the business domai classes.

* <u>Persistent classes</u> - Represent data ~~sour~~ stores that will persist beyond the execution of the software.

* <u>System classes</u> - Implements software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

→ A software team must develop a complete set of attributes and operations for each design class.

→ Characteristics of a well-formed design class:

* Complete and sufficient

A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class.

* Primitiveness

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

* High Cohesion

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

* Low Coupling

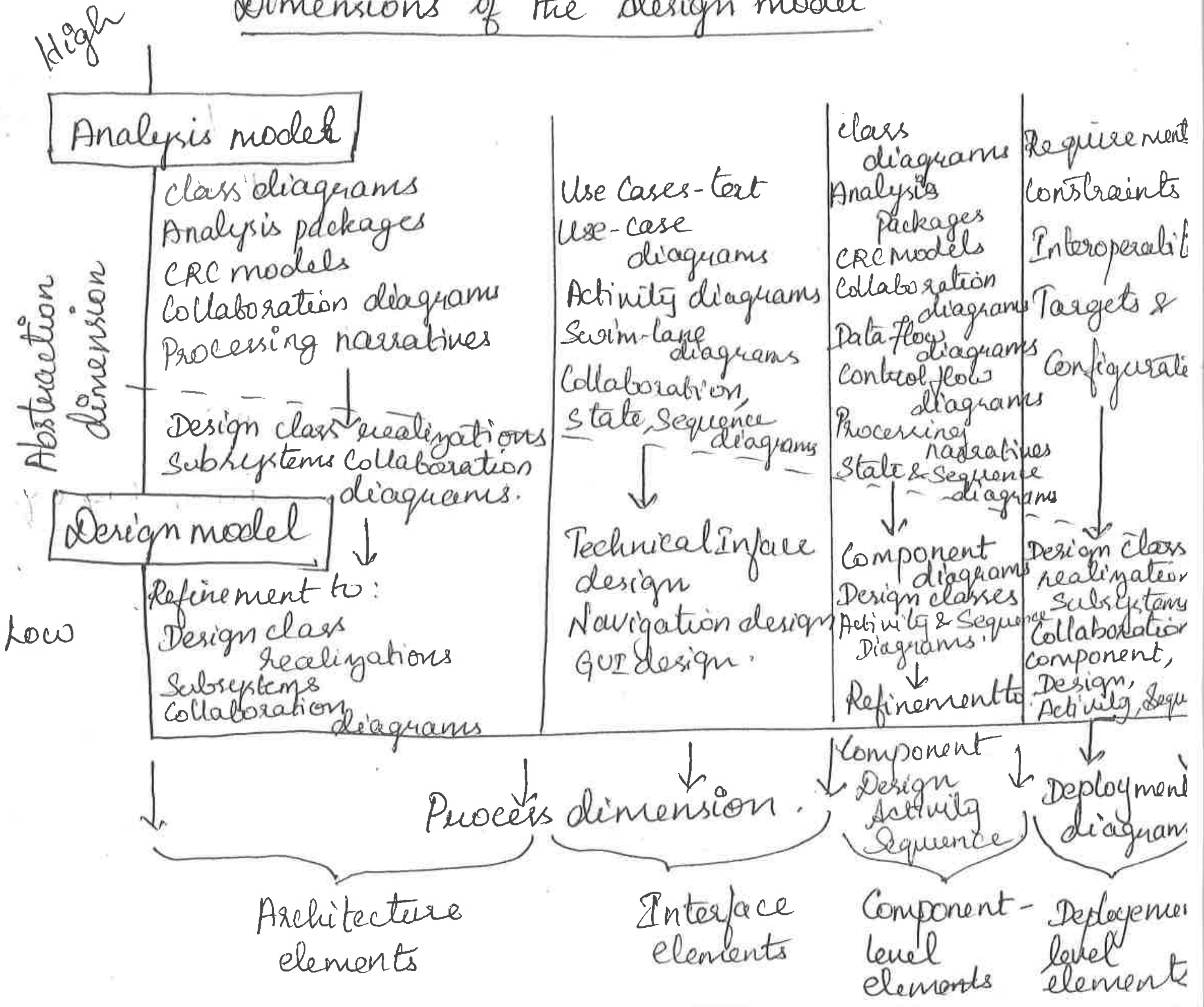
Design classes within a subsystem should have only limited knowledge of classes in other subsystems. This implies that a method should only send messages to methods in neighbouring classes.

THE DESIGN MODEL

→ The design model can be viewed in two different dimensions.

- * Process dimension - Indicates the evolution of the design model as design tasks are executed as part of the software process.
- * Abstraction dimension - Represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively

Dimensions of the design model



→The model elements noted along the horiyontal axis are not always developed in a sequential fashion.

→Preliminary architectural design sets the stage and is followed by interface and component-level design.

→The deployment model is usually delayed until the design has been fully developed.

*Data Design elements

→Data design 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.

→At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-qual 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 level, the collection of informatio stored in seperate databases and reorganiyed into a "data warehouse" enables datamining or knowledge discovery that can have an impact on the success of the business itself.

* Architectural Design elements

→ Architectural design elements gives an overall view of the software.

→ The architectural model is derived from three sources:

(1) Information about the application domain for the software to be built.

(2) Specific analysis model elements (Eg: relationships & collaborations of analysis class).

(3) the availability of architectural patterns and styles.

* Interface Design elements

→ The interface design elements for software tell how information flows into and out of the system and how is communicated among the components defined as part of the architecture.

→ There are three important elements of Interface design:

(1) The User Interface (UI)

(2) External Interfaces to other systems/networks.

(3) Internal Interfaces between various design elements or components.

→ UI is a unique subsystem within the overall application architecture.

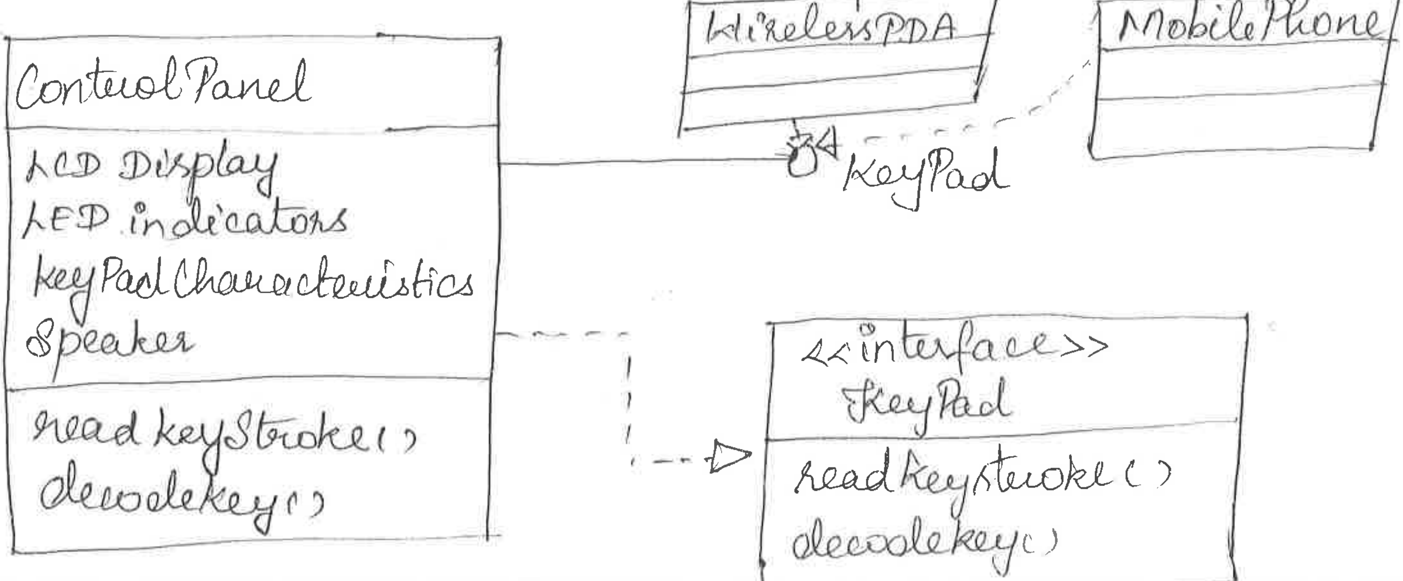
→ The design of external Interfaces requires definitive information about the entity to which information is sent or received. It should incorporate error checking and appropriate security features.

→ The design of internal interfaces is closely aligned with component-level design.

→ Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes.

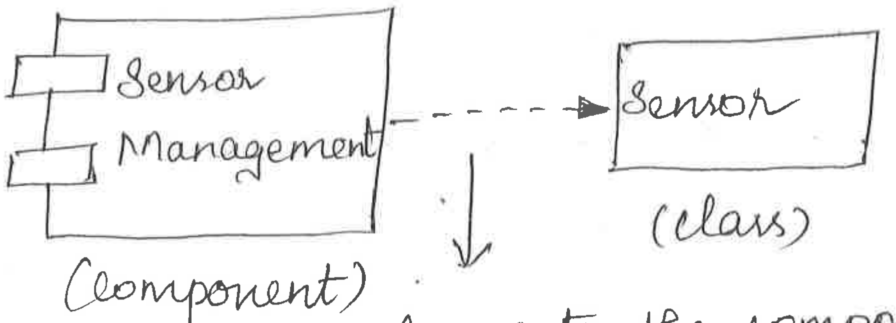
→ Each message must be designed to accomodate the requisite information transfer and the specific functiona requirements of the operation that has been requested.

"An Interface is a set of operations that describes some part of the behaviour of a class and provides access to those operations".



* Component-Level Design Elements

→The component_level design defines datastructures for all local dataobjects and algorithmic detail for all processing that occurs within a component and an interfa that allows access to all component operations.



Connects the component to the class named "sensor".

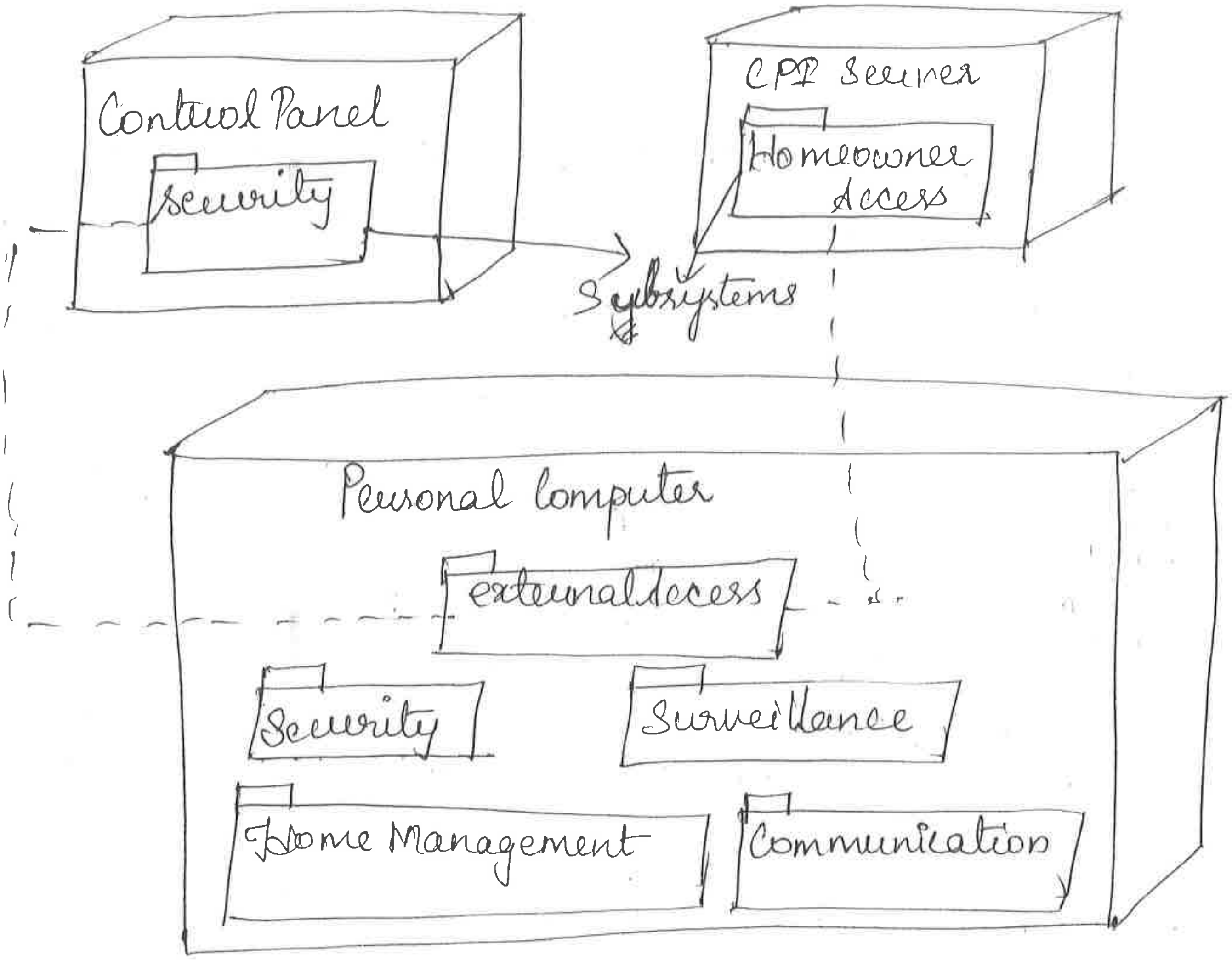
→Detailed procedural flow for a component can be represented using either pseudocode or some diagramatic ~~forma~~ form.

* Deployment-Level Design Elements:

→ Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

→ It shows the computing environment but does not explicitly indicate configuration details.

→ The instances of the deployment diagram can be identified at the latter stages of the design.



PATTERN-BASED SOFTWARE DESIGN:

→A software engineer should look for every opportunity to reuse existing design patterns rather than creating new ones.

(*) Describing a Design Pattern:

→The patterns has its own operations and attributes.

→A description of the design pattern may also consider a set of design forces.

→Design forces describe nonfunctional requiremen In addition forces define the constraints that may restrict the manner is which the design is to be implemented.

→The pattern characteristics indicate the attributes of the design that may be adjusted to enable the pattern to accomodate a variety of problems.

→These attributes represent characteristics of the design that can be searched so that an appropriate pattern can be found.

→Guidance associated with the use of a design pattern provides an indication of the remifications of design decisions.

Technical Problem : Software reuse is the inability to find existing reusable patterns when hundreds or thousands of candidate patterns exist.

→The search for the "right pattern" is aided immeasurably by a meaningful patternname.

(*) Using Patterns in Design:

→ types of design patterns

(1) Architectural Patterns:

*These patterns define the overall structure of the software, indicate the relationships among subsystems and software components, and defines the rules for specifying relationships among the elements of the architecture

(2) Design Patterns:

*These patterns address a specific element of the design such as an aggregation of components to solve some design problem, relationships among components to solve some design problems, ~~sclat~~, or the mechanisms for effecting component-to-component communication.

(3) Idioms:

* These are sometimes called coding patterns.
* These language-specific patterns generally implement an algorithmic element of a component, a specific-interface protocol, or a mechanism for communication among components.

(*) Frameworks:

→ A framework is not an architectural pattern, but rather a skeleton with a collection of "plug-points" that enable it to be adapted to a specific problem domain.

→ The "plug points" enable a designer to integrate problem-specific classes or functionality within the skeleton.

→ In an object-oriented context, a framework is a collection of cooperating classes.

→ Frameworks are applied with no changes. Additional design elements may be added, but only via the plugpoints that allow the designer to flesh out the framework skeleton.

SOFTWARE ARCHITECTURE:

→ The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.

→ The architecture is not the operational software.

→ It enables the software engineer to:

(*) Analyze the effectiveness of the design in meeting its stated requirements.

(*) consider architectural alternatives at a stage when making design changes is still relatively easy.

(*) Reduce the risks associated with the construction of the software.

→ Software architecture considers two levels of the design:

(1) Data design - Represents the data component of the architecture in conventional systems and class definitions in object-oriented systems.

(2) Architectural design - Representation of the structure of software components, their properties and interactions.

(*) Why is Architecture Important?

→ Enables communication between all parties interested in the development of a computer-based system.

→ Highlights early design decisions that will have a profound impact on all software engineering work that follows.

→ Constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together.

DATA DESIGN

→ The data design translates data objects into a data structures or a database architecture.

(*) Data design at the architectural level.

→ The challenge is to extract useful information from this data environment, particularly when the information desired is cross-functional.

→ To solve this challenge, data mining techniques, also called as KDD (Knowledge discovery in databases) is developed that provides an attempt to extract appropriate business-level information.

→ A data warehouse, adds an additional layer to the data architecture.

→ A data warehouse is a seperate data environment that is not directly integrated with day-to-day applications but encompasses all data used by a business.

→ A data warehouse is a large, independent database that has access to 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.

→ Principles:

* The systematic analysis principles applied to function and behavior ~~shob~~ should also be applied to data.

* All data structures and the operations to be performed on each should be identified.

* A mechanism for defining the content of each data object should be established and used to define both data and the operations applied to it.

* Low-level data design decisions should be deferred until late in the design process.

*The representation of a 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 datatypes.

ARCHITECTURAL STYLES AND PATTERNS:

→The Software that is built exhibits one of many architectural styles.

→Each style describes a system category that encompasses:

(*) a set of components that perform a function required by the system.

(*) a set of connectors that enable "communication, coordination and cooperation" among components.

(*) Constraints that define how components can be integrated to form the system.

(*) 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 style is a transformation that is imposed on the design of an entire System.

→An architectural pattern imposes a transformation on the design of an architecture.

→A pattern differs from a style in a number of fundamental ways:

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

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

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

→ Patterns can be used in conjunction with an architectural style to establish the overall structure of a system.

(*) Architectural Styles

(1) Data-centered architecture

→A data store resides at the center of this architecture and is accessed frequently by other components that update, add, delete or 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.

→ A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

→It promotes integrability. It emphasizes that existing components can be changed and new client components added to the architecture without concern about other clients. Client components independently execute processes.

→ A pattern differs from a style in a number of fundamental ways:

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

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

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

→ Patterns can be used in conjunction with an architectural style to establish the overall structure of a system.

(*) Architectural Styles

(1) Data-centered architecture

→ A data store resides at the center of this architecture and is accessed frequently by other components that update, add, delete or 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.

→ A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

→ It promotes integrability. It emphasizes that existing components can be changed and new client components added to the architecture without concern about other clients. Client components independently execute processes.

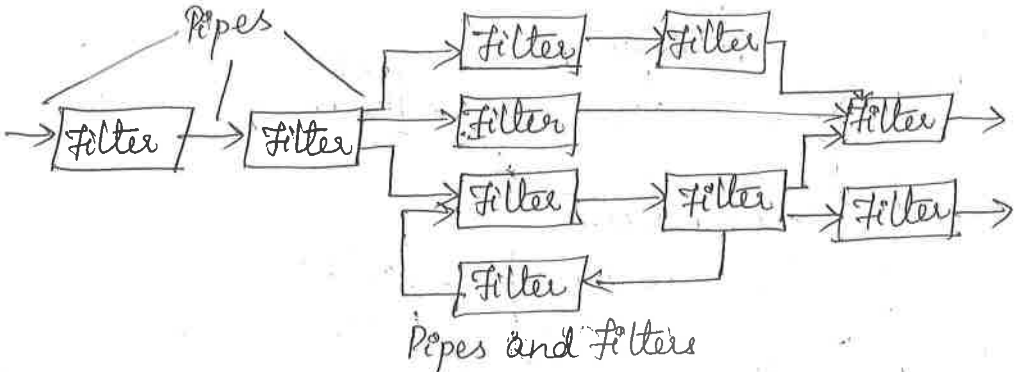


Fig: Data flow architecture

(3) Call and Return Architecture

* Main program/Subprogram architecture

→It 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 architecture

→The components of a main program/Subprogram architecture are distributed across multiple computers on a network.

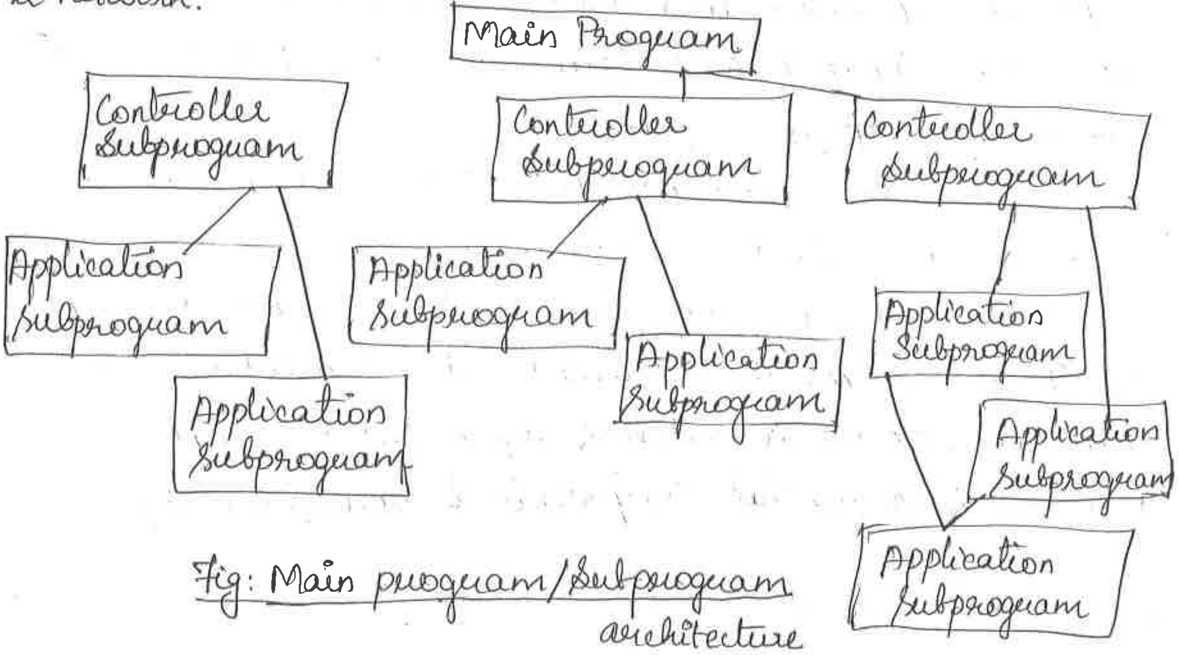


Fig: Main program/Subprogram architecture

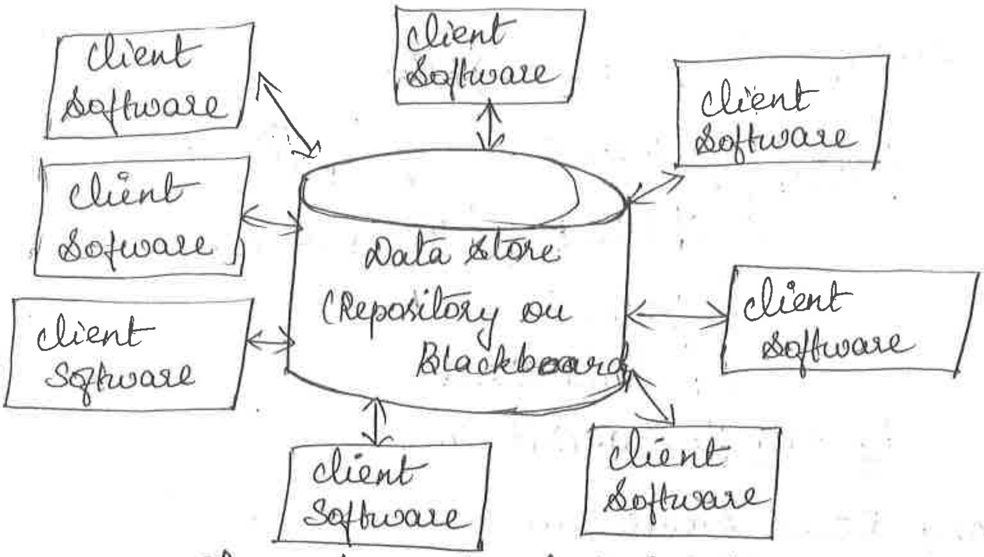


Fig: Data-centered Architecture

(2) Data-flow architecture

→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 structure 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.

→The filter does not require knowledge of the workings of its neighbouring filters..

→If the dataflow degenerates into a single line of transforms, it is termed batch sequential. This structure accepts a batch of data and then applies a series of sequential components to transform it.

* Object-Oriented Architecture

→ 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 Architecture

→ 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.

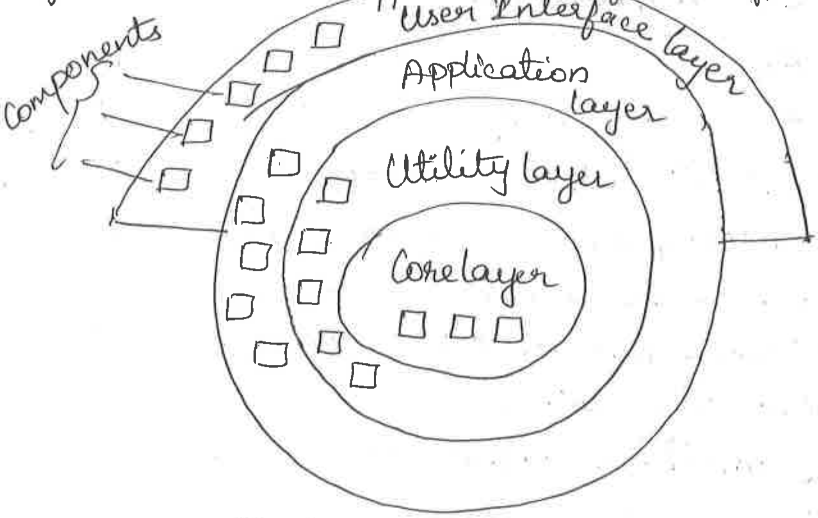


Fig: Layered Architecture

→ Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style or combination of styles that best fits those characteristics and constraints can be chosen.

ASSESSING ALTERNATIVE ARCHITECTURAL DESIGN.

* An architecture Trade-Off Analysis method.

→ The design ~~and~~ analysis activities that follow are performed iteratively:

(1) Collect scenarios - A set of use-cases is developed to represent the system from the user point of view.

(2) Elicit requirements, constraints and environment description - It is used to be certain that all stakeholders concerns have been addressed.

(3) Describe the architectural styles/patterns that have been choosen to address the scenarios and requirements.

(4) Evaluate quality attributes by considering each attribute in isolation - Reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.

(5) Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style. This can be accomplished by making small changes in the architecture and determining sensitivity of the quality attributes. Any attributes that are affected by ~~sensit~~ variation is known [illegible] as sensitivity points.

(6) Critique candidate architectures using the sensitivity analysis.

* Architectural Complexity:

→ A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture.

→ Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers.

(Refer fig 10.12, 10.13 from pg no:310 &
fig 10.14 from pg no:311 in textbook)

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

→ The designer selects global flow characteristics based on the prevailing nature of the DFD.

→ In addition, local regions of transform or transaction flow are isolated.

→ These sub-flows can be used to refine program architecture derived from a global characteristics described previously.

→ An overall transform characteristics will be assumed for information flow.

Step 4 - Isolate the transform center by specifying incoming and outgoing flow boundaries.

→ Incoming flow describes path that converts information from external to internal form.

→ Outgoing flow boundaries converts from internal to external form.

→ Incoming and outgoing flow boundaries are open to interpretation.

→ Different designers may select slightly different points in the flow as boundary locations.

→ Care should be taken when boundaries are selected because a variance of one bubble anlong a flow path will generally have little impact on the final program structure.

→ The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of boundaries.

Step 5 - Perform "first-level factoring".

→ Factoring results in a program structure in which top-level components perform decision making and low-level components perform most input, computation and output work.

→ Middle-level components perform some control and do moderate amounts of work.

→ When transform flow is encountered, a DFD is mapped to a specific structure that provides control for incoming, transform and outgoing information processing.

→ The number of modules at the first level should be limited to the minimum that can accomplish control functions and still maintain good functional independence characteristics.

(Refer fig no: 10.05 pg no: 312 from textbook)

Step 6 - Perform "second-level factoring"

→ Second-level factoring is accomplished by mapping individual transforms of a DFD into appropriate modules within the architecture.

→ Beginning at the transform-center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software architecture.

→ Two or even three bubbles can be combined and represented as one component, or a single bubble may be expanded to two or more components.

→ Practical considerations and measures of design quality dictate the outcome of ~~sec~~ second-level factoring.

→ Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side.

(Refer fig 10.16 & 10.17 from pg.no: ~~10~~ 314 in textbook)

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

→ A first-iteration architecture can always be refined by applying concepts of functional independence.

→ Components are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling and most importantly, a structure that ~~can~~ can be implemented without difficulty, tested without confusion and maintained without grief.

* Transaction Mapping

→ A single data item triggers one of a number of information flows that effect a function implied by the triggering data item.

→ The data item, called transaction, and its corresponding flow characteristics are discussed.

(Refer fig. no: 10.19 pg no: 317 from textbook)

Step 1 :- Review the fundamental system model.

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

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

Steps 1, 2 & 3 are identical as transform mapping.

Step 4 - Identify the transaction center and the flow characteristics along each of the action paths.

→ The location of the transaction center can be immediately discerned from the DFD.

→ The transaction center lies at the origin of a number of action paths that flow radially from it.

→ The incoming path and all action paths must be isolated. Each action path must be evaluated for its individual flow characteristics.

Step 5 - Map the DFD in a program structure amenable to transaction processing.

→ Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch.

→ 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

User Interviews

→Interviews involve representatives from the software team who meet with end-users to better understand their needs, motivations, work-culture and other issues.

→This can be accomplished in ~~one-one~~, one-on-one meetings or through focus groups.

Sales Input

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

Marketing Input

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

Support Input

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

→The following set of questions will help the interface designer better understand the users of a system:

* Are users trained professionals, technicians, clerical or manufacturing workers?

* What level of formal education does the average user have?

* Are the users capable of learning from written materials or have they 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?