UNIT II

Understanding Requirements: Requirements Engineering, Establishing the Ground work, Eliciting Requirements, Building the Requirement Model, Negotiating Requirements, and Validating Requirements.
Design Concepts: Design within the Context of Software Engineering, the Design Process, Design Concepts.
Architectural Design: Software Architecture, Architecture Genres, Architecture Styles, Architecture Design, Assessing Alternative Architecture Designs, Architecture Mapping Using Data Flow.

Understanding Requirements

REQUIREMENTS ENGINEERING

A BRIDGE TO DESIGN AND CONSTRUCTION:

Requirement engineering, like all other Software Engineering activities, must be adapted to the process, project, product and the people doing the work. Requirement Engineering begins during the communication activity and continues into the modeling activity. It is essential that the software team make a real effort to understand the requirements of a problem before the team atoms to solve the problem.

Requirement Engineering builds a bridge to design and construction. It allows a software team, to examine,

- About the context of the software work to be performed.
- The specific needs that design the construction must address.
- The priorities that guide the order in which work is to be completed.
- The information, functions and behaviors that will have a profound impact on the resultant design.

REQUIREMENTS ENGINEERING TASKS

Requirement Engineering Provides an appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification and managing requirements. Requirement engineering tasks are classified as:

- Inception
- Elicitation
- Elaboration
- Negotiation
- Specification
- Validation
- Requirements management

- **Inception:** In some cases, for casual conversation all that is needed is to precipitate in a major software engineering effort. At project Inception, Software Engineers ask a set of questions, which establish:
  - Basic understanding of the problem
  - People who want a solution.
  - Nature of solution that is desired.
  - Effectiveness of preliminary communication and collaboration between customer and developer.

- **Elicitation:** Elicitation is about asking the customers, uses and others "what they want", i.e., what the objectives of the product are, what is to be accomplished, how the product/ system fits into business needs, and finally how product is to be used on day-to-day basis. ***But elicitation is difficult because:***

- ✔ ***Problems of Scope:*** Boundary of the system is ill-defined, or customers/ users specify unnecessary technical detail that make confuse rather than clarify, overall system objectives.
- ✔ ***Problems of Understanding:*** Customers or users have a poor understanding of their computing environment, the problem domain etc., specify requirements that conflict with other users needs or specify requirements that are ambiguous or untestable.
- ✔ ***Problems of Volatility:*** Requirements change over time.

1) Software engineers overcome these three problems by gathering requirements in an organised manner.

- ❖ **Elaboration:** Information obtained from the customer during Inception and elicitation is expanded and refined during elaboration. It focuses on developing a refined technical model of software functions, features and constraints.

  Elaboration is driven by creation and reinforcement of user scenarios that describe how end-user will interact with the system. Each user scenario is parsed to extract analysis classes (business entities) that are visible to end user. The relationships and collaboration between classes are identified and UML diagrams are produced. End-result of elaboration is an analysis model that defines informational, functional and behavioural domain of the problem.

- ❖ **Negotiation:** In this task, customers, users and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Risks associated with each requirement are identified and analysed. Rough "guestimates" of development are made and used to assess impact of each requirement on project cost and delivery time. Measures are taken in negotiations (discussions) so that each party achieves some satisfaction.

- ❖ **Specification:** A specification can be written document, a set of graphical models, a formal mathematical model (algorithm), a collection of usage scenario or a prototype or a combination of these. It is necessary to remain flexible when specification is to be developed. For large systems, a written document is the best approach, whereas for smaller products, usage scenarios are best.
  1. Specification is final work product by requirements engineer.
  2. It serves as foundation for subsequent Software Engineering activities and describes function, performance and constraints in the product.

- ❖ **Validation:** Work products produced as a result of Requirement Engineering are assessed for quality during validation step. It examines,
  1. Specification to ensure all software requirements are stated unambiguously.
  2. That inconsistencies, omissions, errors have been detected and corrected.
  3. The work products conform to standards established for the process, project and product.

3. Primary requirement validation mechanism is the Formal Technical Review. The review team consists of Software Engineers, customers, users other stakeholders. Review team examines specification for errors, missing information, inconsistencies, conflicting requirements or unrealistic requirements.

- ❖ **Requirements Management**: "It is a set of activities that help the project team identify, control and track requirements and changes two requirements at any time as the project proceeds."
  1. It begins with identification; each requirement is assigned a unique identifier. Once requirement have been identified, traceability/ feasibility tables are developed. Each traceability table relates requirements to one or more aspects of the system or its environment.

  Possible traceability tables are:

  1) Features traceability table: shows how requirements relate to important customer observable system/

product features.

2) Source traceability table: Identifies source of each requirement.
3) Dependency traceability table: Indicates how requirements are related to one another.
4) Subsystem traceability table: Categorises requirements by the subsystem(s) that theygovern.
5) Interface traceability table: Shows how requirements relate to both internal and external systeminterfaces. These traceability tables are maintained in Requirements Database to understand how a change in requirements will affect different aspects of the system to be built.

INCEPTION (OR) INITIATING REQUIREMENT ENGINEERING PROCESS:

To get the project started and forward towards a successful solution, we need the following steps to initiate Requirement Engineering:

- **Identifying the Stakeholders:** stakeholder is defined as "anyone who benefits in a direct or indirect way from the system which is being developed." Each stakeholder has a different view of the system, achieves different benefits when system is successfully developed and is open to different risks.

  At inception, requirement engineer should create list of people who will input as requirements are elicited. The initial list will grow as stakeholders are contacted further.

- **Recognizing Multiple Viewpoints:** As different stakeholders exist; requirements of the system will be explored from many different points of the view. Each of various constituencies such as marketing groups, business managers, end users, Software Engineers will contribute information to the Requirements Engineering process. For ex. support engineer may focus on the software maintainability.

  The job of requirements engineer is to categorize all stakeholder information including inconsistent and conflicting requirements in a way that will allow decision makers to choose a consistent set of requirements for the system.

- **Working towards Collaboration:** Customers should collaborate among themselves and with Software Engineers to result a successful system. The job of requirement engineer is to identify areas of commonality and areas of conflict or inconsistency.

  In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" (Ex: business manager) may make the final decision about which requirements make the cut.

- **Asking the first question:** The questions asked at the Inception of the project should be "context free". The first set of questions focus on customer and other stakeholders, overall goals, and benefits.
  - For ex: requirements engineering might ask:
    - Who is behind the request for this work?
    - Who will use the solution?
    - What will be the economical benefit of the successful solution?
    - Is there another source for the solution that you need?

  These questions help to identify all stakeholders who will have interest in software to be built. These also identify measurable benefits of successful implementation and alternatives for software development.
  - Next set of questions include:
    - What problems will this solution address?
    - How would you characterize "good output"?
    - Can you show me the environment where solution will be used?
    - Will special performance issues or constraints affect the way the solution is approached?

  These questions enable software team to gain better understanding of the problems and allows customer to say his/her perceptions.
  - Final set of questions are:

- Are you the right person and are your answers "official"?
- Are my questions relevant to your problem?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

These questions focus on effectiveness of communication. These are also called as meta- questions. All these questions will help to "break the ice" and initiate the communication that is essential for successful elicitation.

ELICITING REQUIREMENTS

The Q&A session should be used for the first encounter only and then replaced by requirements elicitation format.

Collaborative Requirements Gathering: Many different approaches to collaborative requirements gathering have been proposed and each follows the basic guidelines:

1) Meetings are conducted and attended by both Software Engineers, customers along with stakeholders.
2) Rules for preparation and participation are established.
3) An agenda is suggested that is formal enough to cover all important points but informal enough to encourage free flow of ideas.
4) A "facilitator" (customer/ developer/ outsider) controls the meeting.
5) A "definition mechanism" (worksheets etc) can be used.
6) The goal is to
    a. Identify the problem.
    b. Propose elements of the solution
    c. Negotiate different approaches
    d. Specify preliminary set of solutionrequirements.

During Inception the stakeholders write a "one or two page request". A meeting place, time, date and a facilitator are selected. Then the product request is distributed to all attendees before the meeting date, and ask to go through the product request and make suggestions in the meeting.

Quality Function Deployment: QFD is a technique that translates the needs of customer into the technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the Software Engineering process". QFD identifies three types of requirements:

1) **Normal Requirements:** These reflect objectives and goals stated for a product during meetings with the customer. If these requirements are present then the customer is satisfied.
2) **Expected Requirements:** These are implicit to the product and customer does not explicitly state them. Their absence will cause significance dissatisfaction.
3) **Exciting Requirements:** These reflect features that go beyond customer's expectations and prove to be very satisfying when present.

- In meetings with the customer, Function Deployment determines value of each function that is required for the system.
- Information Deployment identifies both data objects and events that system must consumer and produce.
- Task Deployment examines behaviour of the system within context of its environment.
- Value Analysis is conducted to determine relative priority of the requirements determined during each of three deployments.

1. QFD uses customer interviews, observations, surveys and examination of historical data as raw data for

requirements gathering activity. These data are then translated into a table of requirements, called as the Customer voice table that is reviewed with customer.

User Scenarios: Developers and users create a set of scenarios that identify a trend of usage for the system to be constructed. These scenarios often called use cases provide a description of how the system will be used.

Elicitation Work Products: These depend on the size of the system/product to be built. These include:

- A statement of need or feasibility.
- A bounded statement of scope for system or product.
- A list of customers, users and stakeholders.
- A description of system's technical environment.
- A list of requirements and constraints that apply.
- A set of usage scenarios that provide insight into use of the system.
- Any prototypes developed to better define requirements.

ELABORATION

Developing Use Cases: "Use case is defined as set of sequence of actions performed by an actor to achieve a specific result". An actor refers to various people that use system or product within context of the function.

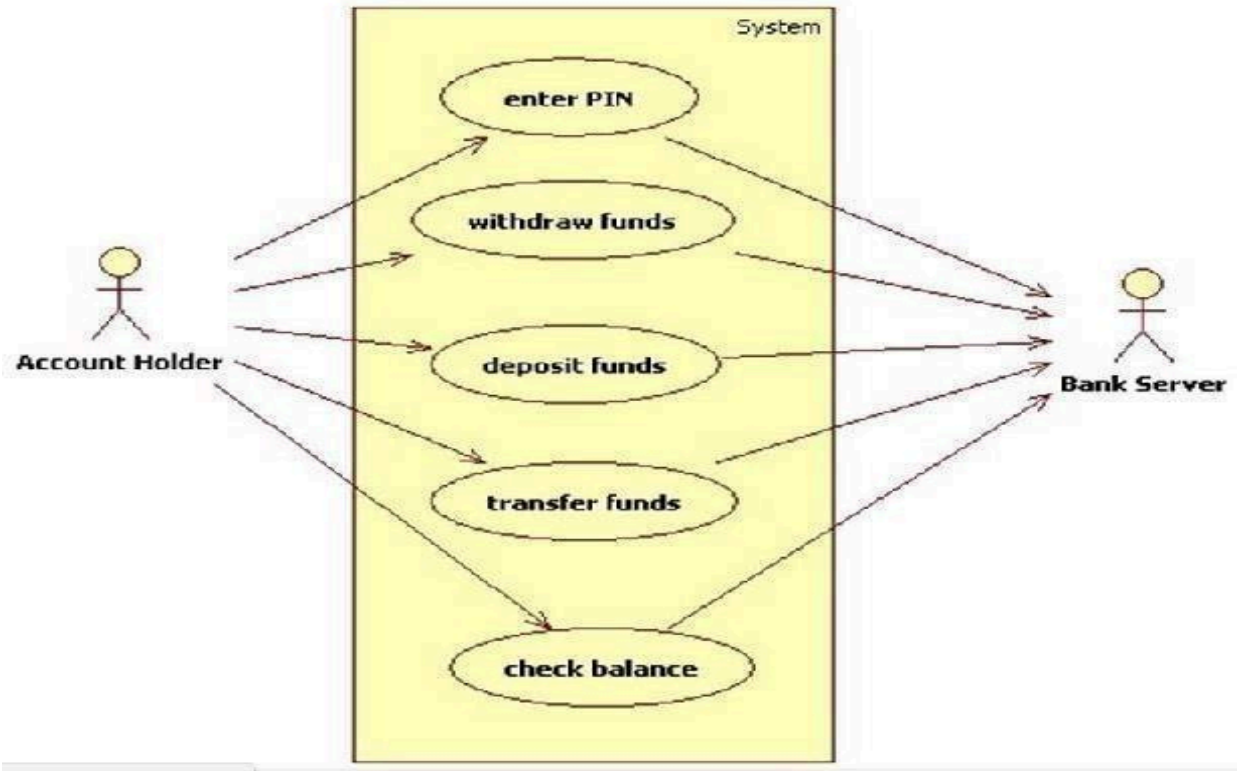


Fig: Use Case Diagram for ATM

Use Case is a collection of user scenarios that describe the thread of usage of a system. Each scenario is described from the point-of-view of an "actor"—a person or device that interacts with the software in some way. Each scenario answers the following questions:

- Who is the primary actor, the secondary actor (s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What extensions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?

- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Building the Analysis Model: The intent of the analysis model is to provide a description of the functional, informational and behavioural requirements for a computer-based system.

- Analysis model is a snapshot of requirements at any given time. We expect it to change. As the analysis model evolves, certain elements will become relatively stable and other elements may be more volatile, indicating customer does not yet understand requirements for system.

Elements of Analysis Model: The specific elements of the analysis model are dictated by analysis modeling method. These elements include:

i. **Scenario-based elements:** These are often the first part of analysis model that is developed. They serve as input (or) informational requirements for creation of other modeling elements.

1. A variation in scenario based modeling depicts activities (functions/operations) that have been defined as a part of requirement elicitation task, i.e., sequence of activities is defined as part of analysis model. Activities can be represented iteratively at different levels of abstraction by using activity diagrams (or) use case diagrams.

As an example, consider UML diagrams for eliciting requirements.

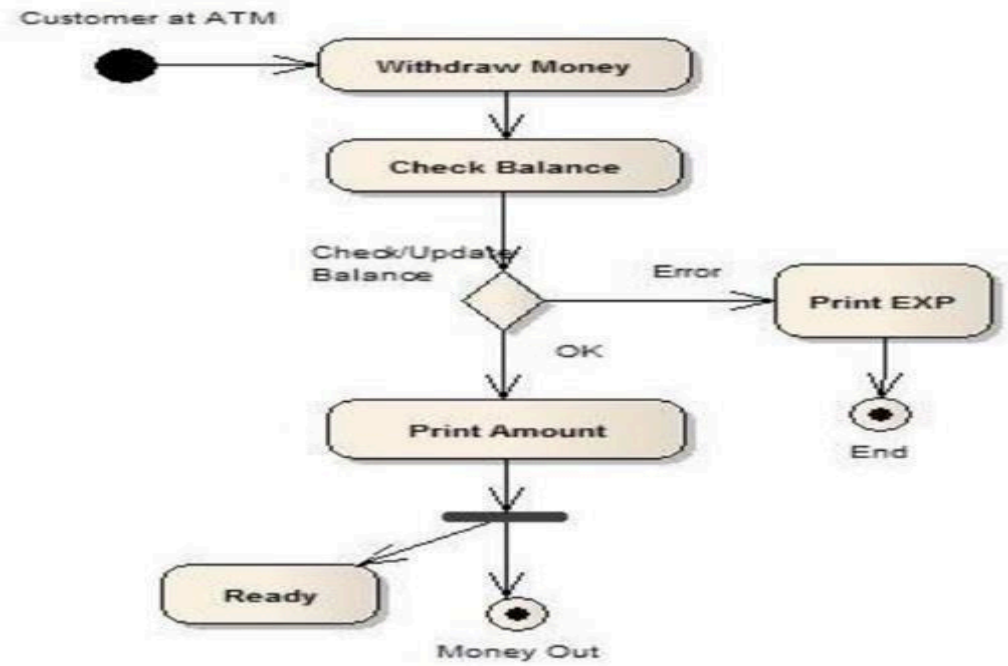
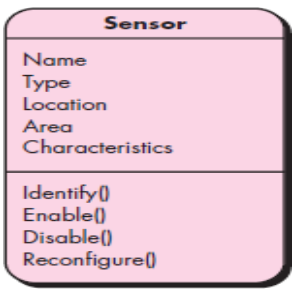


Fig: Activity Diagram

ii. **Class-based elements:** Each usage scenario implies a set of "Objects" that are manipulated as an actor interacts with the system. These objects are categorised into "Classes"- a collection of things that have similar attributes and common behaviour.

2. A class diagram usually represents the functional requirements in analysis model. Analysis model may also depict the manner in which classes collaborate with one another and relationships between them. An example for class diagram is,



iii. **Behavioral Elements:** The behaviour of a product can have a profound effect on design and implementation approach. [i.e., static or dynamic].

3. State diagram is used to represent behaviour of a system buy depicting its states and events that cause the system to change state. *A state is any observable mode of behaviour.*

4. A state diagram indicates what actions are taken as a consequence of a particular event. A state diagram is given as,

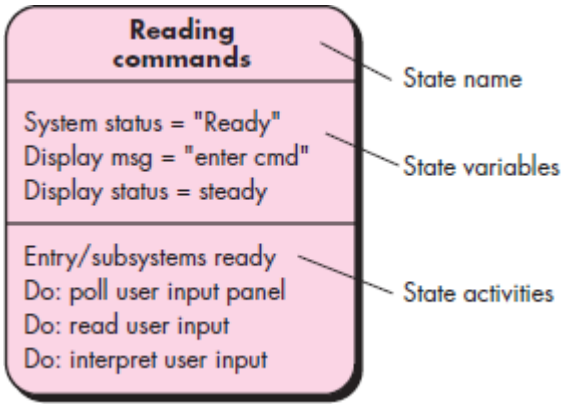


Fig: State Diagram Notation

iv. **Flow oriented elements:** Information/Data is transformed as it flows through a computer-based system. System accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. This data flow is depicted using DFDs (Data Flow Diagrams).

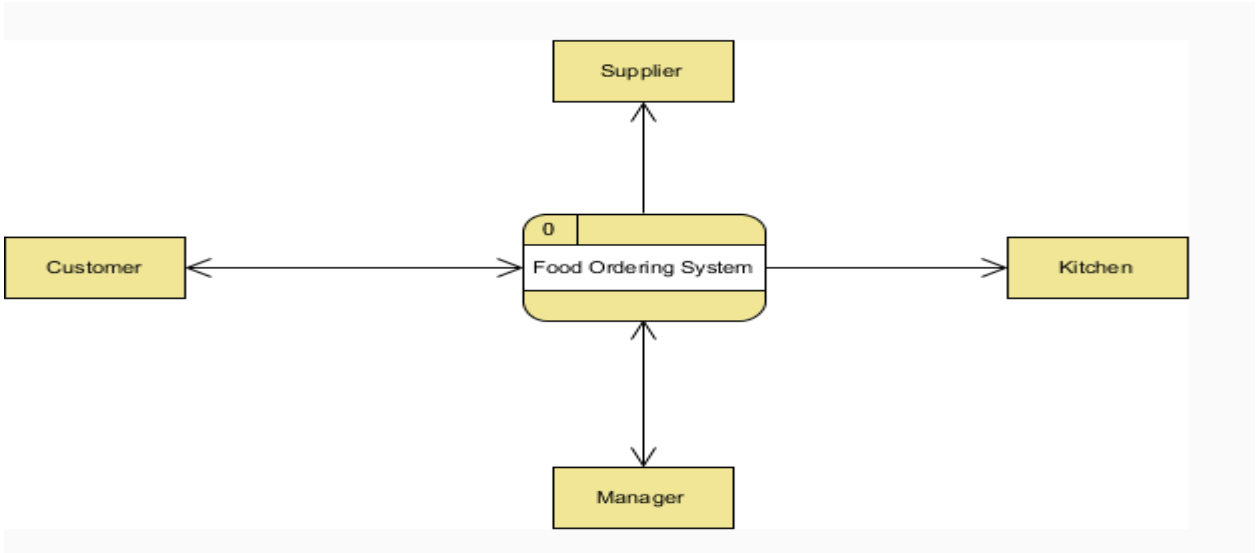


Fig: Data Flow Diagram

Analysis Patterns: Analysis patterns represent the things (class, function or behaviour) that can be reused when modeling many applications. Analysis patterns are integrated into analysis model by reference to the pattern name. They are also stored in a repository so that Requirement Engineers can reuse them.

Analysis pattern template includes:

- Pattern name: A descriptive that captures essence of pattern.
- Intent: Describes what pattern accomplishes or represents and/or what problem is addressed.
- Motivation: A scenario that illustrates how pattern can be used to address the problem.
- Forces and context: Description of external issues that can affect how pattern is used and how they will be resolved.
- Solution: Description of how pattern is applied to solve problem.

- Consequences: Address what happens when pattern is applied.
- Design: Discusses how analysis pattern can be achieved through use of known designpatterns.
- Known uses: Examples of uses within actual systems.
- Related patterns: One or more analysis patterns that are related to named pattern because the analysis pattern,
    - is commonly used with named pattern
    - is structurally similar to the named pattern
    - is a variation of named pattern.

NEGOTIATING REQUIREMENTS

Customer and developer enter into a process of negotiation, where they will have a discussion about balancing functionality, performance, and other product or system characteristics against cost and time to market.

a) The best negotiations strive for a **"win-win"** result. i.e., customer wins by getting system/ product that satisfies the needs, and software team wins by working to realistic and achievable budget and deadlines.

Boehm defines a set of negotiation activities:

1) Identification of system/ subsystem key stakeholders.
2) Determination of stakeholders "Win conditions".
3) Negotiate stakeholders win conditions to reconcile them into a set of **win-win** conditions for all concerned (including software team).

VALIDATING REQUIREMENTS

Requirements are validated in this task by a review of customer requirements. A review of analysis model addresses the following questions:

1. Is each requirement consistent with overall objective for the system/ product?
2. Have all requirements been specified at proper level of abstraction?
3. Is the requirement really necessary (or) does it represent and add-on feature that may not be essential?
4. Is each requirement bounded and unambiguous?
5. Does each requirement have attribution? That is, is a source noted for each requirement?
6. Do any requirements conflict with other requirements?
7. Is each requirement testable?
8. Is each requirement achievable in technical environment?
9. Does requirements model properly reflect information, function and behaviour of system to be built?
10. Are all patterns consistent with customer requirements?
11. Have all patterns been properly validated?
12. Have requirements patterns been used to simplify the requirements model?
13. Has the requirements model been "partitioned" in a way that exposes progressively more detailed.

DESIGN CONCEPTS:

DESIGN ENGINEERING:

Design is a core engineering activity. Design creates a model of the software. Design engineering encompasses the set of principles, concepts and practices that lead to the development of a high quality system or product.

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity and delight. Design engineering for computer software changes continually as new methods, better analysis and broader understanding evolve.

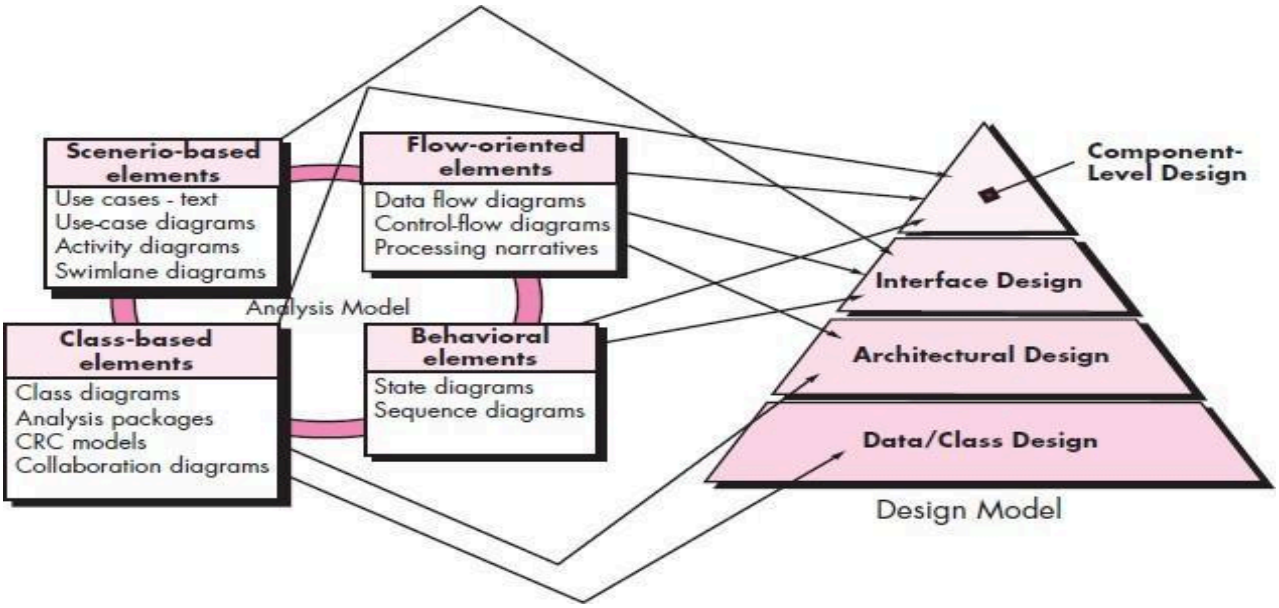
a) A product should be ***designed*** in a ***flexible*** manner to develop quality software.

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING:

Software design is the last software engineering action within modeling activity and sets the stage for development. The analysis model, manifested by scenario-based, class- based, flow-oriented and behavior elements, feed the design task.

Design model produces a data/class design, an architectural design, an interface design, a component design and a deployment design.

1. The data/class design transforms the analysis class models into design class realizations and data structures require implementing the software. Part of class design may occur as each software component is designed.
2. The architectural design defines the relationship between major structural elements of software, the architectural styles and design patterns and the constraints that affect the way in which architectural can be derived from system specifications, the analysis model and interaction of subsystems defined within



analysis model.

Fig: Translating the Requirements model to Design model

3. The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. Usage scenarios and behavioral models provide much of information required for the interface design.
4. The component-level design transforms structural element of the software architecture into a procedural description of software components. Information from class-based models, flow models, behavioral models serve as basis for component design.

DESIGN PROCESS AND DESIGN QUALITY:

Importance of software design can be stated with one word QUALITY. Software design serves as foundation for all software engineering and software support activities that follow.

Software design is an interactive process through which requirements are translated into a "blueprint" for constructing the software. Initially, design is represented at a high level of abstraction. As iteration occurs, subsequent refinement leads to design representations at lower levels of abstraction.

The following characteristics serve as a guide for evaluation of good design:

1. The design must implement all explicit requirements contained in analysis model, and accommodate all implicit requirements desired by customer.
2. Design must be a readable, understandable guide for those who generate code, who test and support the software.
3. Design should provide a complete picture of the software, addressing data, functional and behavioral domains.
5. Each of these characteristics is goal of the design process.

Quality Guidelines: Guidelines for quality design are:

1. A design should exhibit an architecture that
   a) has been created using recognizable architectural styles/patterns.
   b) composed of components that exhibit good design characteristics.
   c) can be implemented in an evolutionary fashion.
2. A design should be modular; software should be logically partitioned into elements or sub-systems.
3. It should contain distinct representations of data, architecture, interfaces and components.
4. It should lead to data structures that are appropriate for the classes to be implemented.
5. It should lead to components that exhibit independent functional characteristics.
6. It should lead to interfaces that reduce complexity of connections between components and external environment.
7. It should be represented using a repeatable (iterative) method.
8. It should be represented using a notation that effectively communicates its meaning.

b) Design engineering encourages good design through the application of fundamental design principles, systematic methodology and through review.

Quality Attributes: Hewlett-Packard (HP) developed a set software quality attributes, given by the acronym ***FURPS:***

1. ***Functionality:*** It is assessed by evaluating feature set and capabilities of the program, generality of functions and security of overall system.
2. ***Usability:*** It is assessed by considering human factors, overall aesthetics, consistency and documentation.
3. ***Reliability:*** It is evaluated by measuring frequency & severity of failure, ability to recover, accuracy of output results, Mean- Time-To-Failure (MTTF) and predictability of the program.
4. ***Performance:*** It is measured by processing speed, response time, resource consumption, throughout and efficiency.
5. ***Supportability:*** It combines the ability to extend the program (extensibility), adaptability, serviceability, which represent maintainability of the project.

These quality attributes must be considered as soon as design commences, but not after the design is complete and construction has begun.

DESIGN CONCEPTS

Fundamental software design concepts provide necessary framework for "getting it right".

1. **Abstraction:** At highest level of abstraction, a solution for design problem is stated in broad terms using language of the problem environment. At lower levels of abstraction, a more detailed description of solution is provided.

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

7. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function.

   Name of procedural abstraction implies these functions, but specific details are suppressed.

2. **Architecture:** It is the structure or organization of program components (modules), their interaction, and structure of data that are used by components. "Components can be generalized to represent major system elements & their interactions.

A set of architectural patterns enable a software engineer to reuse design level concepts. One goal of software design is to derive an architectural rendering of a system, which serves as a framework to conduct detailed design activities.

Architectural design can be represented using one or more of a number of different models:

1. Structural models: Represent architecture as collection of programcomponents.
2. Framework models: Increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.
3. Dynamic models: Address behavioral change aspects of programarchitecture.
4. Process models: Focus on design of business or technical process that the system must accommodate.
5. Functional models: Can be used to represent functional hierarchy of a system.

3. **Patterns:**"A design pattern describes a design structure that solves a particular design problem within a specific context amid "forces"(constraints) that may have an impact on the manner in which pattern is applied and used."

Intent of each design pattern is to provide a description that enables a designer to determine:

i. Whether pattern is applicable to the current work.
ii. Whether pattern can be reused (hence, saving design time)
iii. Whether pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

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

Modularity is the single attribute of software that allows a program to the intellectually manageable (by breaking big process into modules). Modularity leads to a "divide and conquer" strategy, it's easier to solve a complex problem when you break it into manageable pieces, hence effort required to develop becomes negligibly small.

We modularize a design, so that development can be more easily planned, software increments can be defined and delivered, changes can be more easily accommodated, testing and debugging can be conducted more efficiently and long- term maintenance can be conducted without serious side effects.

5. **Information Hiding:** It suggests that "modules should be specified and designed so that information (algorithms, data) contained within a module is inaccessible to other modules that have no need for such information."

Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module. As most data and procedure are hidden from other parts of the software, errors during modification are less likely to propagate to other locations within the software.

6. **Functional Independence:** It is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.

Functional independence is a key to good design and design is the key to software quality. Independence is assessed using two qualitative criteria:

- ***Cohesion:*** Cohesion is an indication of relative functional strength of a module. A cohesive module performs a single task, requiring little interaction with other components.
- ***Coupling:*** Coupling is an indication of interconnection among modules in software architecture. Coupling depends on the interface complexity between modules.

7. **Refinement:** Stepwise refinement is a top-down design strategy, which is actually a process of elaboration. A program is developed by successively refining levels of procedural detail.

It defines/begins with a statement of function that is defined at a high level of abstraction. Refinement helps the designer to reveal low-level details as design progresses, thus in creating a complete design model.

8. **Refactoring:** "Refactoring is the process of changing a software system in such a way that it does not alter external behavior of the code (design) yet improves its internal structure".

When software is refactored, the existing design is examined for redundancy, unused design elements, poorly constructed data structures, unnecessary algorithms etc for better design.

9. **Design Classes:** As the design model evolves, the software team must define a set of design classes that:
   - ❖ Refine analysis 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.

Design classes provide more technical detail as a guide for implementation. Five different types of design classes, each representing a different layer of design architecture are suggested. They are:

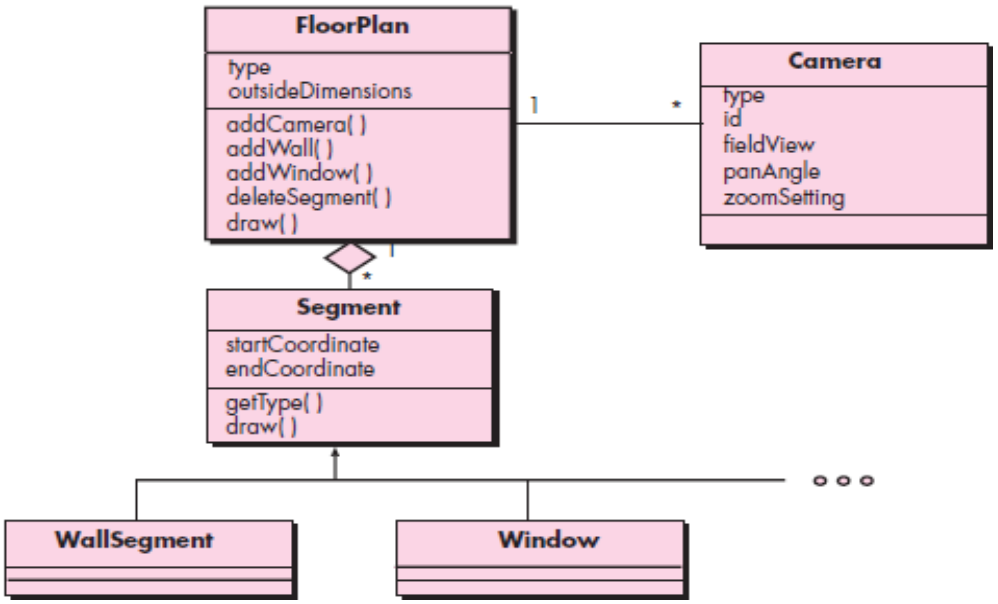
1. *User Interface Classes:* These define all abstractions that are necessary for Human Computer Interaction (HCI). HCI occurs within context of a metaphor (Ex: Order form, a checkbook) and design classes for interface may be visual representations of elements of metaphor.
2. *Business Domain Classes:* These are often refinements of the analysis classes defined earlier. The classes identify attributes and services (operations) that are required to implement some element of the business domain.
3. *Process Classes:* These implement lower-level business abstractions required to fully manage business domain classes.
4. *Persistent Classes:* These represent data stores (DBs) that will persist beyond execution of the software.
5. *System Classes:* These implement software management and control functions that enable system to operate and communicate. These are also known as supporting classes.

As design model evolves, software team must develop a complete set of attributes and operations for each design class.

Four characteristics of a well-formed design class:

1. ***Complete and Sufficient*:** A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class.(No more and No less).
2. ***Primitiveness*:** Methods associated with a design class should be focused on accomplishing one service the class. Once the service has been implemented, with a method, the class should not provide another way to accomplish same thing.
3. ***High Cohesion*:** A cohesion design class has a small, focused set of responsibilities and single- mindedly applies attributes and methods to implement those responsibilities.
4. ***Low Coupling*:** Collaboration between design classes should be kept to an acceptable minimum. If a design model is highly coupled, system is difficult to implement, test & maintain. So, design classes in a subsystem should have only limited knowledge of classes in other subsystems. It is also called as "*Law of Demeter*", suggests that a method should only send messages to methods in neighboring classes.

FIGURE 8.3
Design class for FloorPlan and composite aggregation for the class (see sidebar discussion)



THE DESIGN MODEL

The design model can be viewed in two different dimensions:

- ✔ The process dimension indicates evolution of design model as design tasks are executed as part of the software process.
- ✔ The abstraction dimension represents level of detail as each element of analysis model is transformed into a design equivalent and then refined iteratively.

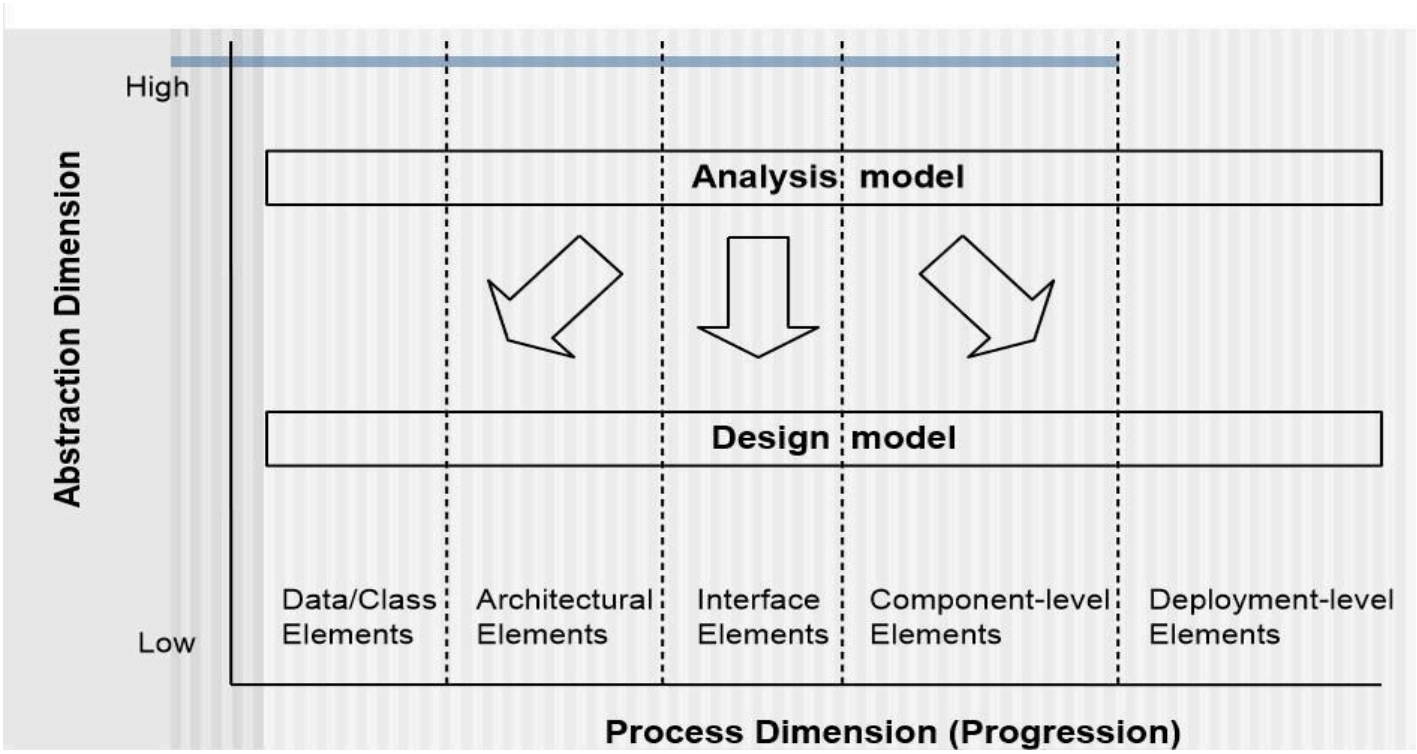


Fig: Dimensions of the Design Model

The elements of design model use many of same UML diagrams that were used in analysis model..The difference is that these diagrams are refined and elaborated as part of design, more implementation- specific detail is provided and emphasis is on architectural structure and style, components & interfaces.

Elements of design model:

A) **Data Design Elements:** Data design also sometimes referred as "Data Architecting". Data design creates a model of data and/or information that is represented at a high level of abstraction.

In many software applications, architecture of data will have a profound influence on architecture of software that must process it. Structure of data always plays important role in software design.

- At program component level, design of data structures and associated algorithms required to manipulate them is essential to the creation of high-quality applications.
- At application level, translation of data model into a DB is important to achieve business objectives.
- At business level, collection of information stored in DBs and reorganized into a "data warehouse" enables data mining or knowledge discovery.

B) **Architectural Design Elements:** These give us an overall view of the software. It is derived from 3 sources:

1) Information about application domain for software to be built.
2) Specific analysis model elements such as DFDs or analysis classes, their relationships and collaborations for the problem.
3) Availability of architectural patterns and styles.

C) **Interface Design Elements:** These tell how information flows into and out of the system and how it is communicated among components designed as part of the architecture. There are 3 important elements of interface design:

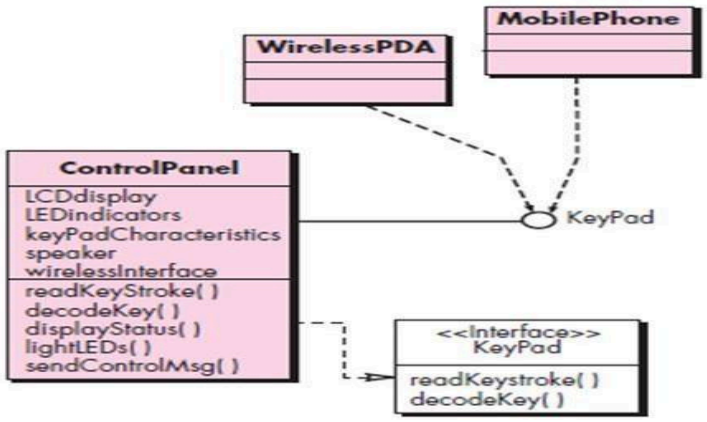
12 **User Interface (UI):** Design of a UI incorporates aesthetic elements (Ex: color, layout, graphics), ergonomic elements (information layout and placement, navigation), and technical elements (UI patterns, reusable components). In general, UI is a unique subsystem within overall application architecture.

13 **External interfaces to other systems, devices, networks, other producers/consumers of information:** The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during Requirement Engineering and verified. This design should incorporate error checking and appropriate security features.

14 **Internal interfaces between various design components:** It is closely aligned with component level design. Design realizations of analysis classes represent all operations and messaging schemes required to enable communication and collaboration between operations in various classes.

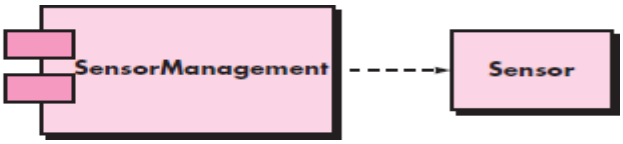
In some cases, an interface is modeled in same way as a class."An interface is a set of operations that describes some part of the behavior of a class and provides access to those operations."

Interface representation for Control-Panel



D) **Component-Level Design Elements:** Component level for software fully describes the internal detail of each software component. To accomplish this, component-level design defines detail for all processing that occurs within a component and an interface that allows access to all component operations.

A UML component diagram



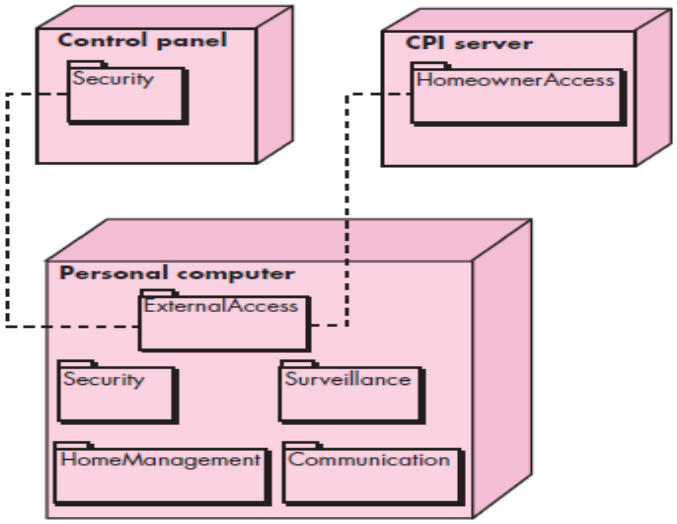
Design details of a component can be modeled at many different levels of abstraction. An activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudo code or some diagrammatic form.

E) **Deployment-Level Design Elements:** These indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

Deployment diagram shows the computing environment but does not explicitly indicate configuration details. Each instance of deployment is identified.

During design, a UML deployment diagram is first developed, and then refined. In a deployment diagram, each subsystem would be elaborated to indicate components that it implements.

A UML deployment diagram



Architectural Design:

SOFWARE ARCHITECTURE

What is Architecture? "The architecture of a system is a comprehensive framework that describes its form and structure - its components and how they fit together".

The software architecture of a computing system is the structure of the system, which comprise software components, externally visible properties of those components and the relationships among them. Architecture is representation that enables software engineer to,

b) Analyze effectiveness of design in meeting its stated requirements.
c) Consider architectural alternatives at a stage when making design changes is still relatively easy.
d) Reduce the risks associated with the construction of software.

In the context of architectural design, a software component can be as simple as a program module, a class or a database.

Why is Architecture Important? The three key reasons that software architecture is important are:

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in dev of a computer-based system.
- Architecture highlights early design decisions that will have a profound impact on all software engineering work and on ultimate success of system.
- Architecture "constitutes a relatively smalls intellectually graspable model of how system is structured and how its components work together.
- Architectural styles and patterns can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture inpredictable ways.

DATA DESIGN

Data Design actions translate data objects as part of the analysis model into DSS at software component level, and when necessary, a database architecture at application level.

Data Design at the Architectural Level: In today's software business environment, where a lot of data and databases are used, the challenge is to extract useful information, particularly when information desired is cross- functional.

1 To solve this challenge, business IT community has developed Data Mining techniques, also called Knowledge Discovery in Databases (KDD), that navigate through existing databases to extract appropriate business level in formation.

2 An alternative solution, called a Data Warehouse is a large, independent database that has access to data that are stored in databases that serve set of applications required by a business.

Data Design at the Component Level: Data design at the component level focuses on representation of data structures that are directly accessed by one or more software components.

Set of Principles for Data Specification are:

- ***The systematic analysis principles applied to functions and behavior should also be applied to***

***data*:** Representations of data flow and content should also be developed and reviewed, data objects should be identified, alternative data organizations should be considered, and impact of data modeling on software design should be evaluated.

- ***All Data Structure and operations to be performed on each should be identified*:** Design of an efficient data structure (DS) must take the operations to be performed into account. The **attributes** and **operations** encapsulated within a class satisfy this principle.
- ***A mechanism for defining content of each data object should be established and used to define both data and operations applied to it*:** Class diagrams define the attributes contained within a class and operations that are applied to attributes.
- ***Low-level data design decisions should be deferred until late in the design process*:** Overall data organization may be defined during requirements analysis, refined during data design work, and specified in detail during component- level design.
- ***The representation of a data structure should be known only to those modulesthat must make direct use of data contained within the structure*:** Concepts of information hiding and coupling provide important insight into the quality of a software design.
- ***A library of useful DS and operations that may be applied to them should be developed*:** A class library achieves this principle.
- ***A software design and programming language should support the specification and realization of abstract data types*:** The implementation of a sophisticated data structure can be made exceedingly difficult if no means for direct specification of structure exists in programming language chosen for implementation

These principles form a basis for a component level data design approach, that can be integrated in to both analysis and design activities.

ARCHITECTURAL STYLES AND PATTERNS

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system.

Each style describes a system category that encompasses:

1. A set of components (Ex: database, computational modules) that perform a function required by a system.
2. A set of connectors that enable "communication, coordination, and cooperation" among components.
3. Constraints that define how components can be integrated to form system.
4. Semantic models that enable a designer to understand overall properties of a system by analyzing know properties its constituent parts.

An architectural pattern, like an architectural style, imposes a transformation on the design of an entire system. A Pattern differs from a style in number of fundamental ways:

1. Scope of a pattern is less broad, focusing on one aspect of the architecture, rather than architecture entirely.
2. A pattern will describe how software handles some aspect of its functionality at the infrastructure level.
3. Architectural patterns bend to address specific behavioral issues within the context of architecture (Ex: Interrupts)

ARCHITECTURAL STYLES:

1. **Data-Centered Architecture:** A data store (Ex: Database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or modify data within the store.

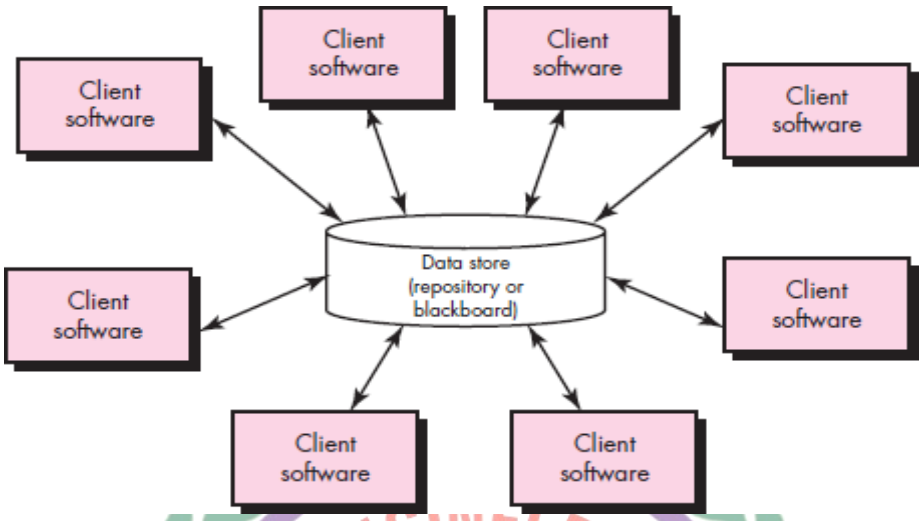


Fig: Data-Centered Architecture

Client software access a central repository. It accesses data independent of any changes to data/actions of other client software. A data-centered architecture promotes inerrability. i.e., existing components can be changed and new client components added to the architecture without concern about other clients. Client components independently execute processes.

2. **Data-Flow Architecture:** This architecture is applied when input data are to be transformed through a series of computational (or manipulative) components in to 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 and produces data output of a specified form.

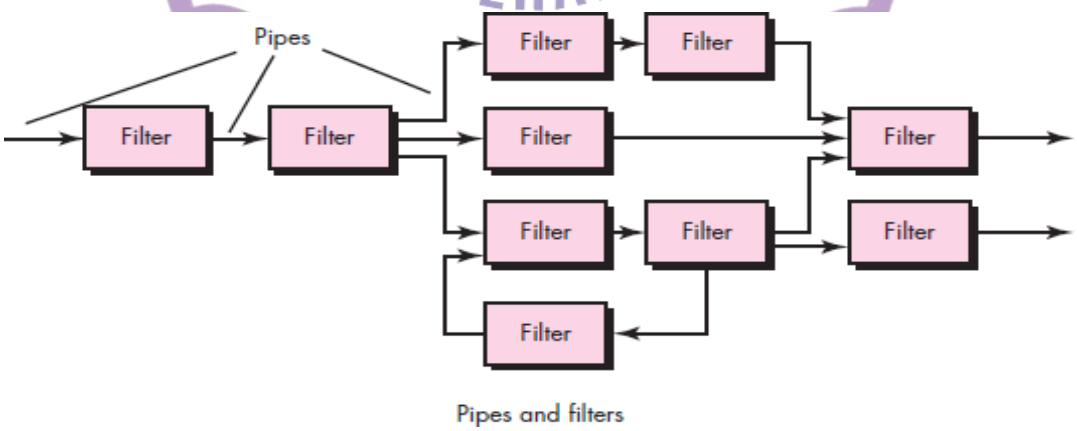
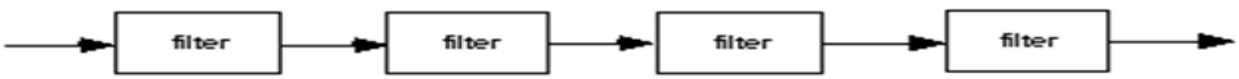


Fig: Data-Flow Architecture

If the data flow 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.



(b) batch sequential

3. **Call and Return Architecture:** It enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale.

Main program/subprogram Architecture: The classic program structure decomposes function into a control hierarchy, where a "main" program invokes no of program components, which in turn may invoke still other components.

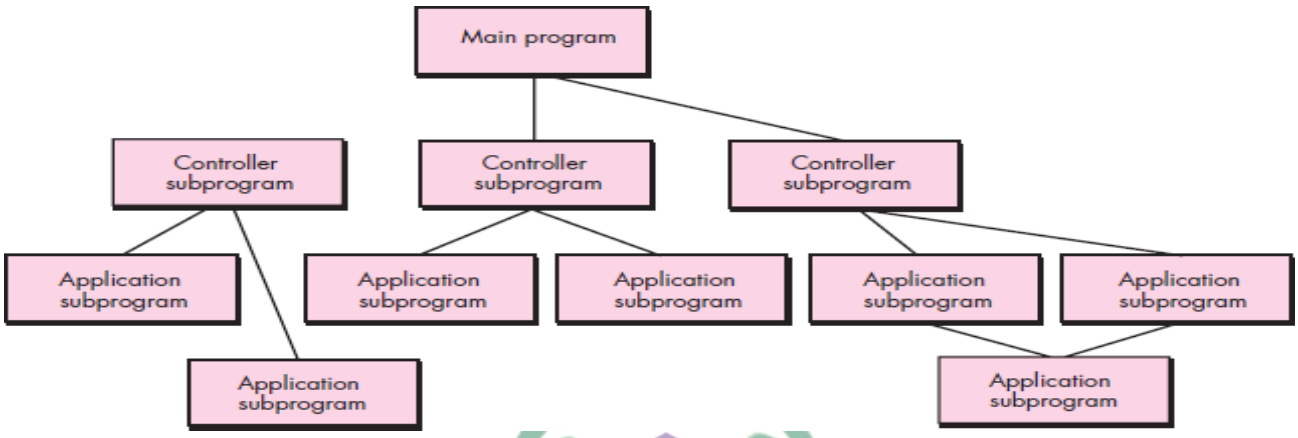


Fig: Main Program/Subprogram Architecture

Remote Procedure Call Architecture: The components of main program/subprogram architecture are distributed across multiple computers on a network.

4. **Object-Oriented Architecture:** 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.

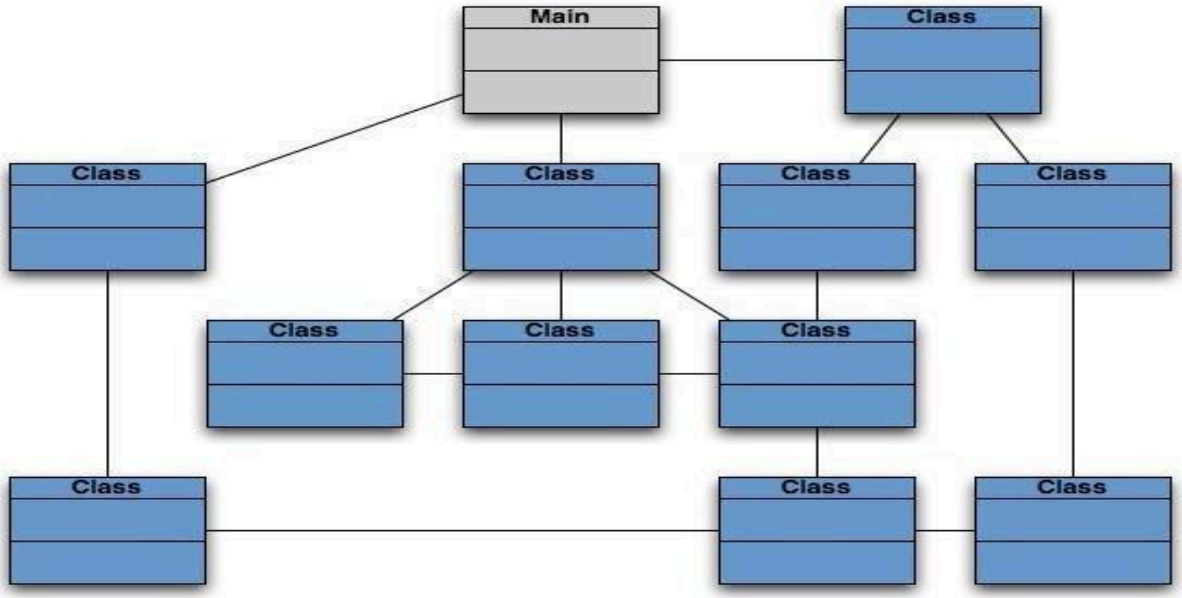


Fig: Object-Oriented Architecture

5. **Layered Architecture:** In this architecture, a number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.

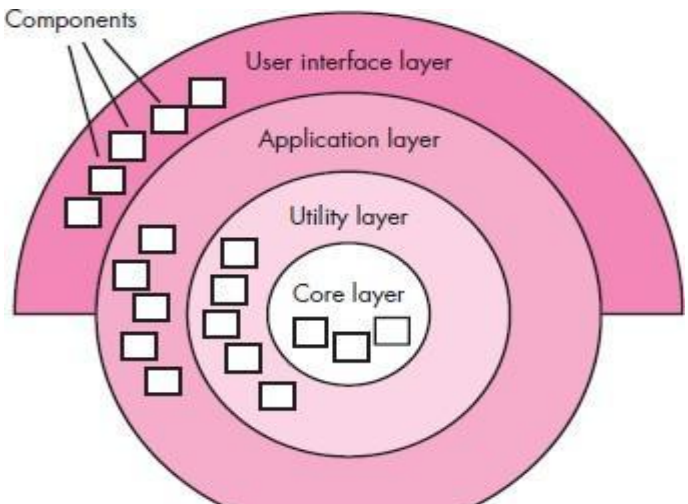


Fig: Layered Architecture

- At outer layer, components service under UI operations.
- At the inner layer, components perform OS interfacing.
- Intermediate layers provide utility services and application software functions.
- Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style or combination of styles that best fits can be chosen.

ARCHITECTURAL PATTERNS: "Architectural patterns define a specific approach for handling some behavioral characteristic of the system". Some domains can be:

1. ***Concurrency:*** Many applications must handle multiple tasks in a manner that simulates parallelism. There are a number of different approaches to handle concurrency.
   A) Operating System Process Management Patterns: It provides built in operating system features that allow components to execute concurrently. Pattern also incorporates operating system functionality that manages the communication between processes, scheduling and other capabilities required to achieve concurrency.
   B) Task Scheduler at application level: This pattern contains a set of active objects each contains a tick () operation, which performs its functions, before returning control back to scheduler.
2. ***Persistence:*** Persistent data are stored in a database or file and may be read or modified by other processes at a later time. TWO architectural patterns are used to achieve persistence.
   A) DBMS Pattern: Applies the storage and retrieval capability of a DBMS to the application architecture.
   B) Application Level Persistence Pattern: It builds persistence features into the application architecture.
3. ***Distribution***: Distribution problem addresses the manner in which systems or components within systems communicate with one another in a distributed environment. Broker pattern addresses this problem. CORBA is an example for broker pattern.

Broker Pattern: A broker acts as a "middle-man" between the client component and server component. The client sends a message to the broker, and broker completes connection.

Organization and Refinement: Following questions provide insight into architectural style that has been derived.

a. ***Control***: How is control managed within the architecture? Does a distinct control hierarchy exist? How do components transfer control within the system? How is control shared among components? What is the control topology? Is control synchronized (or) do components operate asynchronously?
b. **Data**: How are data communicated between components? Is the data flow continuous? What is the made of data transfer? Do data components exist? What is the made of data transfer? Do data components exist? If so, what is their roe? How do functional components interact with data components? Are data components passive/active?

ARCHITECTURAL DESIGN

As architectural design begins, the software to be developed must be put into context i.e. the design should define the external entities (other systems, people, and devices) that the software interacts with and the nature of interaction.

Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. An *archetype* is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled

architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

Representing the System in Context: At the architectural design level, a software architect uses an Architectural Context Diagram (ACD) to model the manner in which software interacts with external entities to its boundaries.

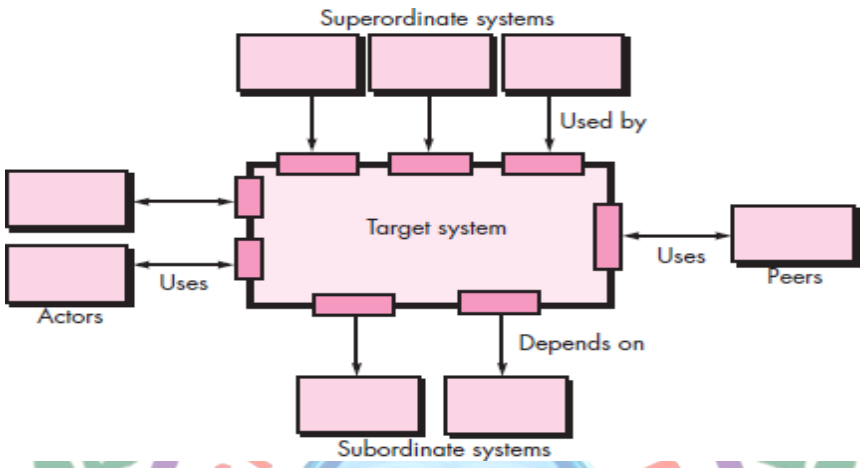


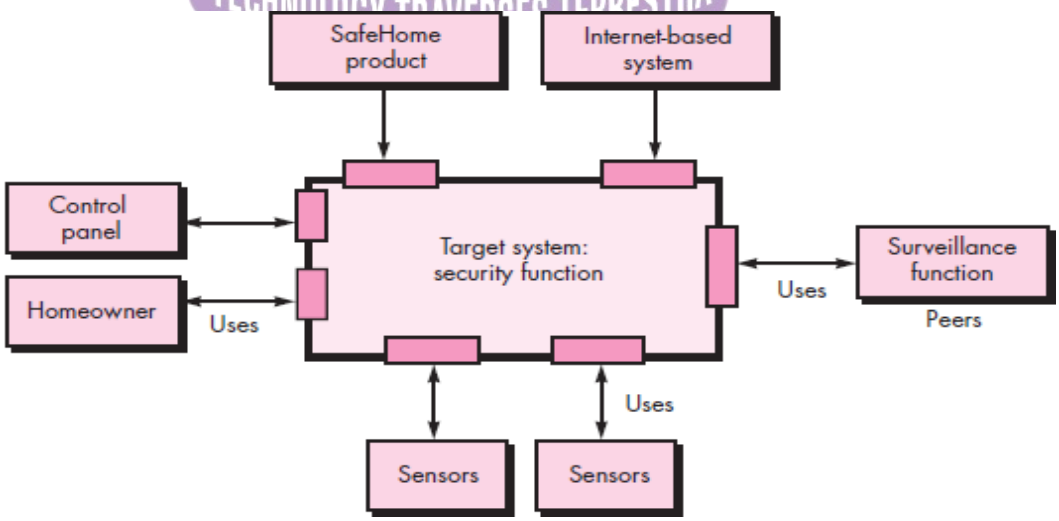
Fig: Architectural Context Diagram

Referring to the figure, systems that incorporate with the target system are represented as:

1. **Superordinate Systems**: Those systems that use the target system as part of some higher level processing scheme.
2. **Subordinate Systems**: Systems that are used by target system and provide data or processing that are necessary to complete target system functionality.
3. **Peer-Level Systems**: Systems that interact on a peer- to – peer basis (i.e. information is produced or consumed by the peers and the target system)
4. **Actors**: Entities that interact with the target system by producing/consuming information that is necessary for requisite processing.

7 Each of these external entities communicates with the target system through an interface (small shaded rectangles in fig). All data that flow into and out of the target system must be identified at this stage.

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 architecture for target system". Archetypes can be derived by examining the analysis classes.

3 Target system architecture is composed of these archetypes, which represent stable elements of the architecture.

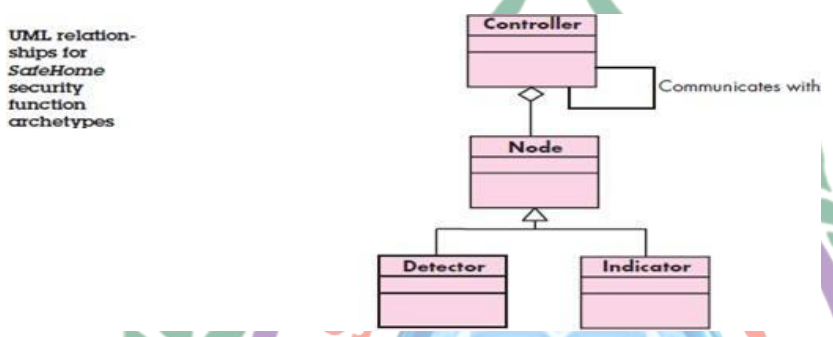
Following archetypes can be defined for an application:

8 **Node**: Represents a cohesive collection of input and output elements of the application.

9 **Detector**: An abstraction that encompasses all sensing equipment that feeds information into target system.

10 **Indicator**: An abstraction that represents all mechanisms for indicating an occurrence of a condition.

11 **Controller**: An abstraction that depicts the mechanism that allows the arming/disarming of a node. If controllers reside on a network, they have ability to communicate with one another.



Refining the Architecture into Components: As software architecture is refined into components, the structure of the system begins to emerge. Analysis classes represent entities within the application domain that must be addressed within the software architecture. So, application domain is one source for derivation and refinement of components.

Another source is Infrastructure Domain. The architecture must accommodate many infrastructure components that enable application components, but have no business connection to the application domain.

- Interfaces depicted in ACD imply one or more specialized components that process data that flow across interface. In some cases, complete subsystem architecture with many components must be designed.

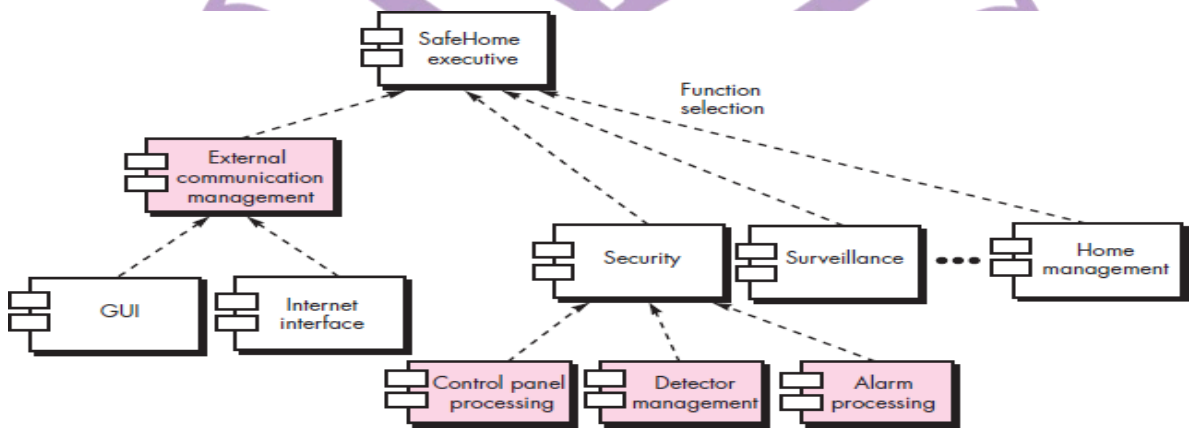


Fig: Refining the Architecture into Components

Describing Instantiations of the System: An actual instantiation of the system architecture is developed, when further refinement (as all design is iterative), is still necessary. By this, the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS

Design results in a no of architectural alternatives that are assessed to determine which is the most appropriate for the problem to be solved.

Architecture Trade-Off Analysis Method: The Software Engineering Institute (SEI) has developed an Architecture Trade –off Analysis Method (ATAM) that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1) ***Collect Scenarios:*** A set of use-cases is developed to represent the system from the user's point of view.
2) ***Elicit Requirements, constraints and Environments Description***: This info is required as part of RE and is used to be certain that all stakeholder concerns have been addressed.
3) ***Describe architectural styles/patterns that have been chosen to address the scenarios and requirements***
4) ***Evaluate quality Attributes b considering each attribute in isolation***: Quality attributes for architecture. Design assessment includes reliability, performance, security, maintainability, flexibility, testability, portability, reusability, andinteroperability.
5) ***Identify the sensitivity of Quality attributes to various architectural attributes for a specific architecture style:*** This can be accomplished by making small changes in architecture. Attributes that are significantly affected by variation in architecture are termed as sensitivity points.
6) ***Critique candidate Architectures (Developed in step 3) using sensitivity analysis conducted in stem 5:*** Once architecture sensitive points have been determined, finding trade-off points is simply the identification of architecture elements to which multiple attributes are sensitive.

These 6 steps represent first ATAM iteration. Based on results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail, and then ATAM steps are reapplied.

Architectural Complexity: A useful technique for assessing the overall complexity of a proposed architecture to consider dependencies between components within the architecture these dependencies are driven by information/control flow within the system.

Types of Dependencies:

12 Sharing Dependencies: These represent dependence relationships among consumers, who use the same resource or producers, who produce for the same consumers.

13 Flow Dependencies: These represent dependence relations between producers and consumers of resources.

14 Constrained Dependencies: These represent constraints on the relative flow of control among a set of activities.

The sharing and flow dependencies are similar to coupling. Coupling is an important design concept that is applicable at the architectural level and component level.

Architectural Description Languages (ADL): ADL provides a semantics and syntax for describing software architecture. ADL should provide the designer with the ability to decompose architecture components, compose individual components into larger architectural blocks, and represent interfaces between components. Once descriptive, language – based techniques for architecture design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

MAPPING DATA FLOW INTO A SOFTWARE ARCHITECTURE:

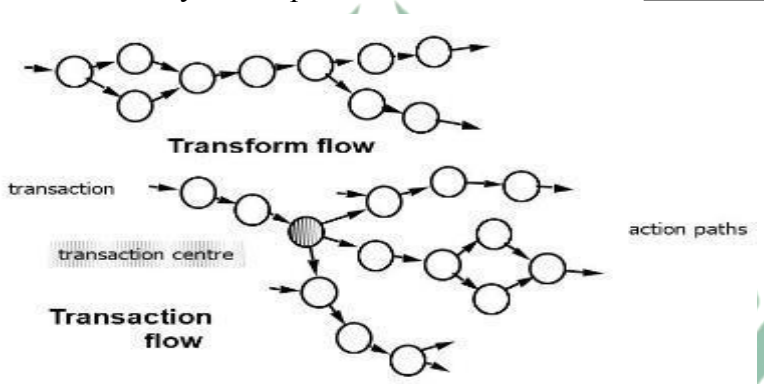
Call and return structure is the mostly commonly used to structure for many types of systems, for architectural mapping. This mapping technique enables a designer to derive reasonably complex call and return architectures from DFDS within analysis model. This technique, sometimes called "structured Design."

4 Structured Design is often characterized as a data flow oriented design method, as it provides a convenient transition from a DFD to software architecture. Type of info flow is the driver for mapping approach.

TRANSACTION FLOW: Information enters the system along paths that transform external data into an internal form. These paths are identified as incoming flow. At the kernel of software, a transition occurs. Incoming data are passed through a "transform center" and begin to move along paths that now lead out the software. Data moving along these paths are called outgoing flow.

4. Overall data flow occurs in a sequential manner and follows one, or only a few "straight line" paths. When a segment of a DFD exhibits these characteristics, Transform Flow is present.

TRANSACTION FLOW: Usually a transaction triggers data flow along one of many paths. Transaction Flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated and based on its value flow along one of many paths is initiated. The hub of information flow from which many action paths emanate is called as "transaction 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 architecture style. To map these DFDs into architecture, following design steps are initiated.

Step 1: Review the Fundamental System Model: Fundamental system model or context diagram (Level 0 DFD) depicts an operation (function) as a single transformation, representing the external producers and consumers ofdata that flow into and out of the function.

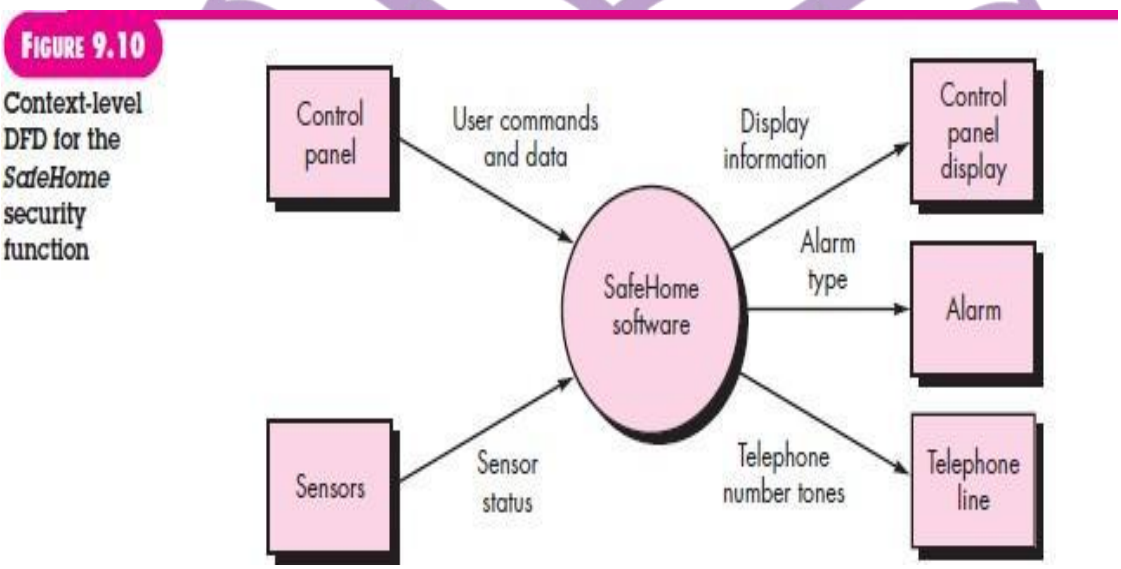


FIGURE 9.10
Context-level DFD for the *SafeHome* security function

Step 2: Review and Refine DFDs for the Software: Information obtained from analysis models is refined to produce greater detail. i.e., process implied by a transform performs a single, distinct function that can be implemented as a component in software.

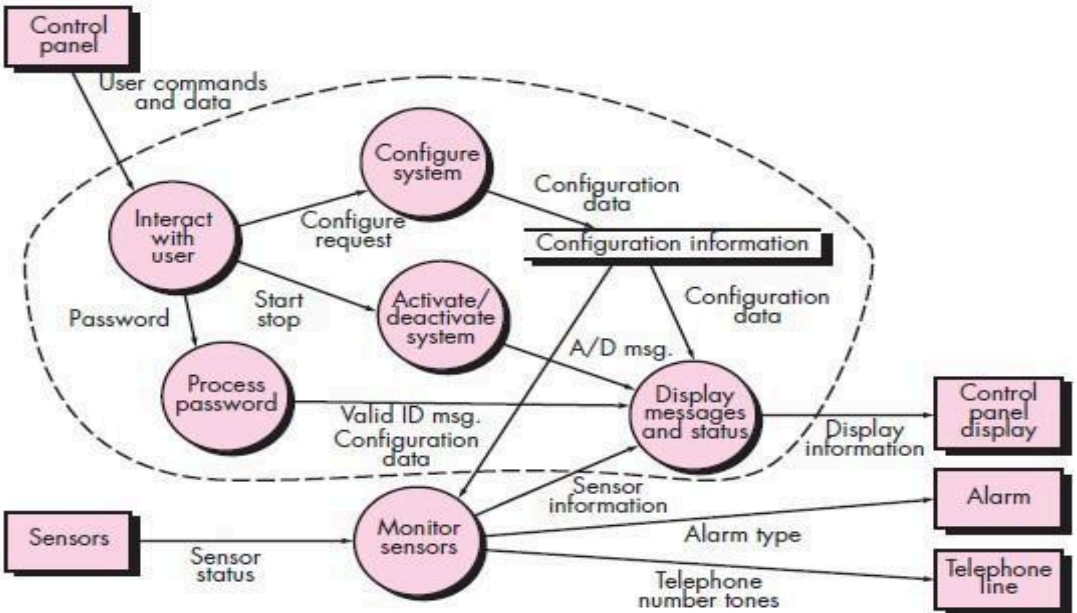


FIGURE 9.11
Level 1 DFD for the *SafeHome* security function

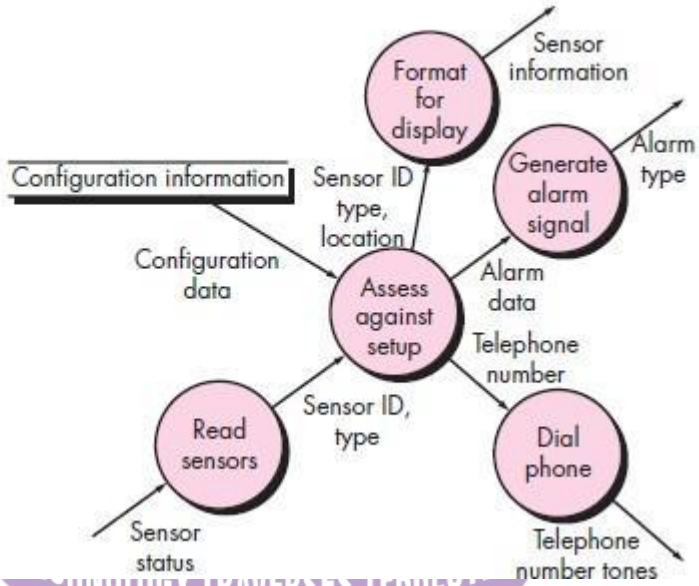


FIGURE 9.12
Level 2 DFD that refines the *monitor sensors* transform

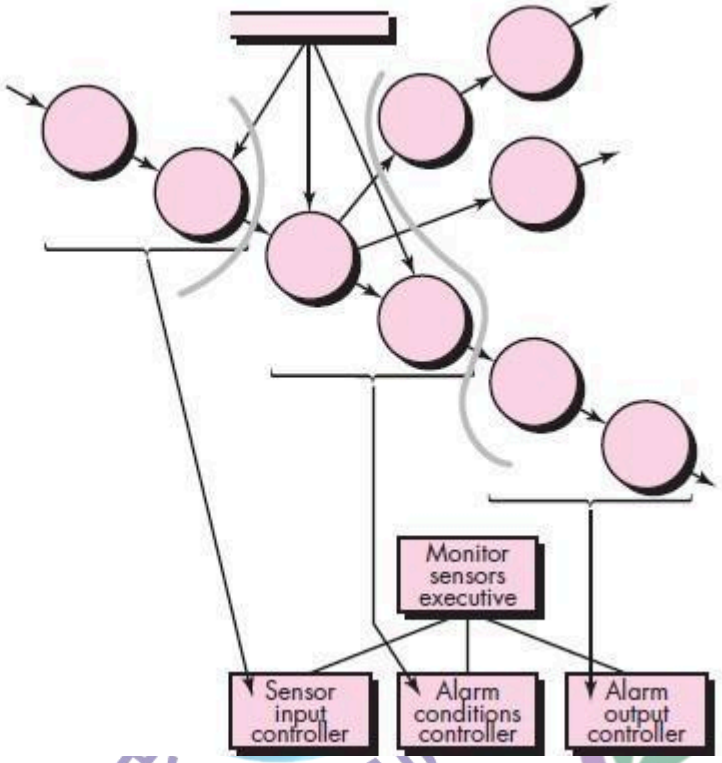
Step 3: Determine whether DFD has Transform (or) Transaction Flow Characteristics: In general, information flow within a system can always be rep as transform. However, in this step, designer selects global (software- wide) flow characteristics based on the prevailing nature of the DFD. Local regions of transform or transaction flow or isolated. These sub flows can be used to refine program architecture derived from a global characteristic described earlier. Hence, an overall transform characteristic will be assumed for information flow.

Step 4: Isolate Transform Center by Specifying Incoming and Outgoing Flow Boundaries: Usually, incoming and outgoing flow boundaries are open to interpretation. i.e. different designers may select slightly

different points in the flow as boundary locations. Proper care should be taken when boundaries are selected; a variance of one bubble along flow path will have little impact on final program structure.

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 input, computation and output work. Middle-level components perform some control and do moderate amounts of work. This type of mapping results in top-down distribution of control.

FIGURE 9.14
First-level factoring for *monitor sensors*



In the first-level factoring a main controller resides at the top of the program structure and coordinates the following subordinate control functions. When transform flow is encountered, a DFD is mapped to a specific structure (call & return architecture) that provides control for incoming, transform, outgoing information processing.

A) No of modules at first level should be limited to minimum.

Step 6: Perform "Second-Level Factoring": Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at transform center boundary and moving outward along incoming and the outgoing paths, transforms are mapped into sub-ordinate levels of the software structure.

Second-level factoring illustrates a one – to – one mapping between DFD transform and software modules, different mappings frequently occur. Practical considerations and measures of design quality dictate the outcome of second-level factoring. Review and refinement may lead to changes in this structure, but it can serve as a "first- iteration" design.

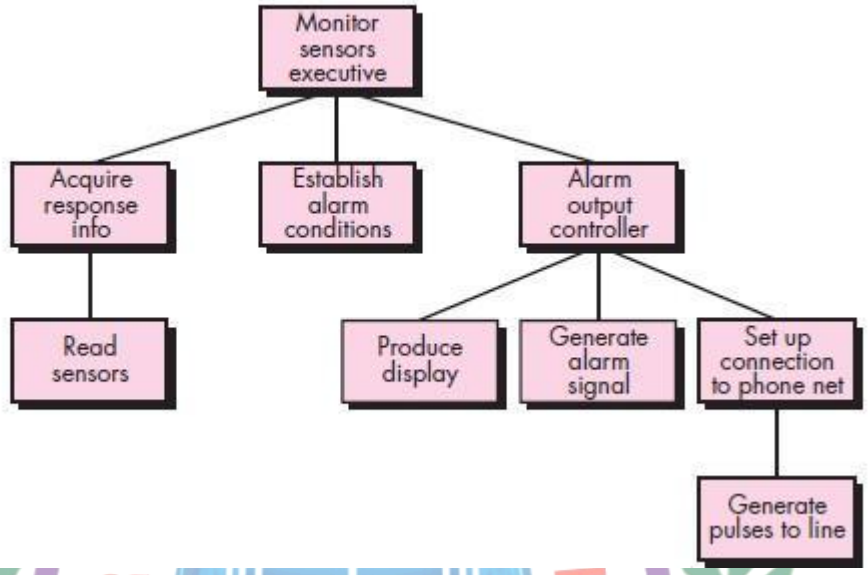
5 Factoring is accomplished by moving outward from the transform center boundary on incoming flow side.

6 Components mapped in preceding manner represent an initial design of software architecture.

Step 7: Refine First-Iteration Architecture using Design Heuristics for Improved Software Quality: A first iteration architecture can always be refined by applying concepts of functional independence. Components are exploded/imploded to produce sensible factoring, good cohesion, minimal coupling and most importantly, a structure that can be implemented without difficulty, tested without confusion and maintained without grief.

Refinements are dictated by analysis and assessment methods as well as practical considerations and common sense. Software requirements coupled with human judgment is the final arbiter.

FIGURE 9.17
Refined program structure for *monitor sensors*



7 The objective of preceding seven steps is to develop an architectural representation of software. i.e., once the structure is defined, we can evaluate and refine software architecture by viewing it as a whole.

5. Modifications at this time require little additional work, yet can have a profound impact on software quality.

TRANSACTION MAPPING: In many software applications, a single data item triggers one of a no of information flows. This data item is called as a transaction.

Design steps for transaction mapping are similar and identical to steps for transform mapping. A major difference lies in the mapping of DFD to software structure.

Step 1: Review the Fundamental System Model: This step is identical to corresponding step for transform mapping.

Step 2: Review and Refine DFDs for Software: This step is identical to corresponding step for transform mapping.

Step 3: Determine whether the DFD has Transform or Transaction Flow Characteristics: This step is identical to corresponding step in transform mapping. (Refer step 3 in transform mapping). However, flow boundaries must be established for both flow types.

Step 4: Identify the Transaction Center and 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 no of action paths that flow radially from it.

The incoming path and all action paths must also be isolated. Each action path must be evaluated for its individual flow characteristic. Incoming, transform and outgoing flow are indicated with boundaries.

Step 5: Map DFD in a Program Structure Amenable to Transaction Processing: The transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch. Structure of incoming branch is developed in much the same way as transform mapping. Starting at transaction center, bubbles along incoming path are mapped into modules.

Structure of dispatch branch contains a dispatcher module that controls all subordinate action modules.

Each action flow path of DFD is mapped to a structure that corresponds to its specific flow characteristics. **Step 6: Factor and Refine the Transaction Structure and Structure of Each Action Path:** Each action path of DFD has its own information flow characteristics, i.e., transform or transaction flow. The action path – related "substructure" is developed using the design steps.

Step 7: Refine First – Iteration Architecture Using Design Heuristics for Improved Software Quality:

This step is identical to corresponding step for transform mapping.

In both design approaches, criteria such as module independence, practicality and maintainability must be carefully considered as structural modifications are proposed.

Refining Architectural Design: Refinement of software architecture during early stages of design is to be encouraged. The approach for optimization is one of the true benefits derived by developing a representation of software architecture.