UNIT –III

PART-A DESIGN ENGINEERING

DESIGN ENGINEERING

- **What is Design ?** Design is an activity where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system.

- **Who does it?** Software engineers / Design engineers

- **Why is it important?** Design allows you to model the system or product that is to be built

DESIGN ENGINEERING

- **What are the steps?**
  - Architecture diagrams
  - Interfaces connecting modules
  - Software components

- **What is the work product?**

A design model that encompasses architectural, interface, component level, and deployment representations is the primary work product that is produced during software design.

DESIGN ENGINEERING

- The Design Process
- The Design Concepts
- The Design Model

The Design Process

- The software Design is an iterative process through which the requirements are translated into “a Blueprint” for constructing the software.
- The Design is represented at a high level of abstraction – A level that can be directly traced to the specific system objectives, and more detailed data, functional , and behavioral requirements.

Good Design Characteristics

- The design Must implement all of the **explicit requirements** contained in the requirement model and **implicit requirements** mentioned by the stakeholders.
- The design must be **readable, understandable guide** for all those who develop the code, and who test and subsequently who support the maintenance of the software.
- The design should provide the **complete picture of the software**, addressing data, functional and behavioral domain from an implementation perspective..

Quality Guidelines

There are 8 guidelines for a well-established design.

1. Design Should Exhibit an architecture

   - that has been created using recognizable architectural styles and patterns.
   - is composed of components exhibits good design characteristics
   - Can be implemented in an evolutionary fashion.

Quality Guidelines

2. Design should be modular.

3. Design should consists of distinct representations of data, architectures, interfaces and components.

4. Should lead to the data structures that are appropriate for the cl asses to be implemented and are drawn from the recognizable patterns.

5. The design should lead to the components that exhibits independent functional characteristics.

Quality Guidelines

6. The design should lead to the interfaces that reduce the complexity of connections between the components and with the external environment.
7. The design should be derived using a repeatable method that is driven by the information obtained during software requirements analysis.
8. Design should be represented using a notation that effectively communicate its meaning.

Quality Attributes

Hewlett-Packard developed the quality attributes as acronym **FURPS**

- Functionality – All functions are included.
- Usability - Aesthetics, consistency
- Reliability – Failure management.
- Performance – Non functional requirements
- Supportability – extensibility, adaptability, compatibility, configuration.

DESIGN CONCEPTS

- A set of fundamental software design concepts has evolved over the history of software engineering.
- “Fundamental software design concepts provide the necessary framework for “getting it right”

DESIGN CONCEPTS

1. Abstraction
2. Architecture
3. Patterns
4. Modularity
5. Separation of concerns
6. Information Hiding
7. Functional independence
8. Refinement
9. Refactoring
10. Design Classes

1. Abstraction

- While considering a modular solution to any problem, many levels of abstraction can be posed.
- 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
- There are two types of abstraction
    - Procedural Abstraction
    - Data Abstraction.

Procedural Abstraction

- A procedural abstraction refers to a sequence of instructions that have a specific and limited functions
- An example of a procedural abstraction would be open for a door. **Open** implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from the moving door, etc.,)

Data Abstraction

- A data abstraction is a named collection of data that describes a data object.
- In the context of the procedural abstraction open , can define a **data abstraction** called **door**.
- Eg., Door type, Swing Motion, Opening Mechanism, Weight , Dimension

2.Architecture

- Architecture is the **structure** or organization of program components(modules), the manner in which these components interact, and the structure of data that are used by the components.

- A set of architectural patterns enable a software engineer to use a design-level concepts.

- Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:
  - **Structural properties.**
  - **Extra-functional properties.**
  - **Families of related systems.**

Architecture : Different Models

The architectural design can be represented using one or more of a number of different models

1.Structural Models – Components Organization

2.Frame Work Models – Repeatable modules

3.Dynamic Models – Configuration changes

4.Process Models – Bussiness/technical process

5.Functional Models – hierarchy of components.

- **Structural Models** represent architecture as an organized collection of program components.
- **Frame work Models** increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

- **Dynamic Models** address the behavioral aspects of the program architecture , including how the structure or system configuration may change as a function of external events.
- **Process Models** focus on design of the business or technical process that the system must accommodate
- **Functional Model** can be used to represent the functional hierarchy of a system.

3.Patterns

- A design pattern describes a design structure that solves a particular design problem with in a specific context and **in spite of forces that may have an impact** on the manner in which the pattern is applied and used.

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

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

4. Separation of Concerns

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is **subdivided into pieces** that can each be solved independently.

If P1 and P2 are two separate processes where P1 is more complex to solve than P2. It is better to separate them logically and solve independently.

Combining two processes will increase the complexity.

5.Modularity

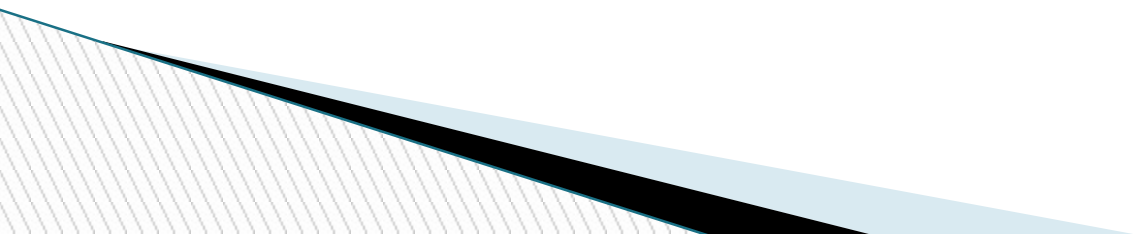
•Modularity means dividing software into smaller, **separately named** and addressable components (modules).

•It is a practical way to apply the separation of concerns principle, making programs easier to understand and manage.

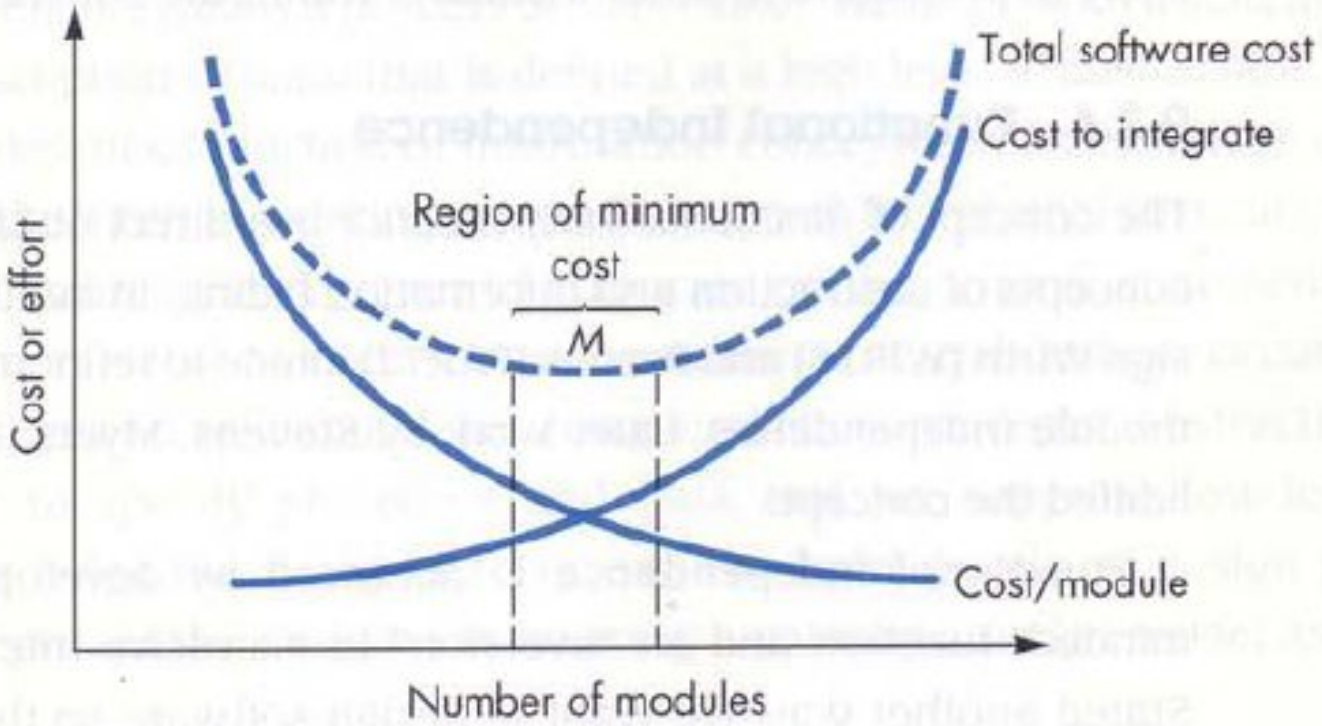
•Monolithic software (a single large program) is too complex to grasp due to many control paths, variables, and dependencies.

Proper modularity supports:

- Easier planning and development
- Incremental delivery
- Easier changes
- Efficient testing and debugging
- Simplified long-term maintenance



Modularity and software cost



•By modularizing, understanding becomes easier, and development costs are reduced.

•However, if software is divided into too many modules (overmodularity), integration costs increase.

•If divided into too few modules (undermodularity), complexity remains high.

•There is an optimal number of modules (M) that minimizes development cost, though it cannot be precisely predicted.

6. Information Hiding

•To achieve modularity we need to break down a software into optimum number of modules.

•Information hiding states that provide only necessary information to each module as per its functionality.

•According to this principle:

- Each module should hide its internal details (data and algorithms) from other modules.
- Only the necessary information should be exposed for interaction.
- This makes modules independent, easier to understand, modify, and maintain.

- Hiding supports modularity by ensuring modules are independent and only share the information needed for the software to function.
- Abstraction defines the essential procedures or information entities of the software.
- Hiding enforces access restrictions, so:
  - Internal procedural details of a module, and
  - Local data structures inside a module

  remain invisible to other modules.

- Using **information hiding** in modular design is especially beneficial during **testing and maintenance**.
- Since most data and internal details are hidden, changes can be made safely.
- This reduces the chance that errors introduced in one module will **spread to other parts** of the software.

7.Functional Indepedence

- Software with effective modularity, that is, independent modules, is easier to develop because function may be **compartmentalized** and interfaces are simplified.
- Independent modules are easier to maintain (and test) because secondary effects caused by design or code modifications are limited, **error propagation is reduced**, and reusable modules are possible.
- **Functional independence is a key to good design and design is the key to software quality**.

- Independence is assessed using two qualitative criteria:

a)Cohesion

b)Coupling

- **Cohesion** is an indication of the relative functional strength of a module.
- **Coupling** is an indication of the relative independence among modules.

Cohesion & Coupling

- A **Cohesion** module performs a single task, requiring little interaction with other components in other parts of a program.
- **Coupling** is indication of interconnection among modules in a software structure.
- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
- In software design, we should strive for the lowest possible coupling to reduce the '**ripple effect'.**

8.Refinement

- Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth.
- Refinement is actually a process of elaboration.
- Begin with a statement of function (or description or data) that is defined at a high level of abstraction.
- The function describes function or information conceptually but provides no information about the internal workings of the functions or the internal structure of data.

- Abstraction enables a designer to specify procedure and data and supress low-level details.
- Refinement helps the designer to reveal low-level details as design progresses.
- Both concepts aid the designer in creating a complete design model.

9.Refactoring

- Refactoring is a reorganization technique that simplifies the design(or code) of a component without changing its function or external behavior.
- 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.

- For example, a first design iteration might yield a component that exhibits low cohesion. The designer may decide that the component should be refactored into three separate components, each exhibiting high cohesion.

- **The result will be software that is easier to integrate , easier to test, and easier to maintain.**

10.Design Classes

- The software team must define a set of design classes that

1. Refine the analysis classes by providing design detail that will enable the classes to be implemented,
2. Create a new set of design classes that implement a software architecture to support the business solution

- Five different types of design classes :

1.User Interface Classes

2.Business domain Classes

3.Process Classes

4.Persistent Classes

5.System Classes

- **User Interface classes** define all abstractions that are necessary for human computer interaction (HCI). In many cases .

- **Business domain classes** are other refinement of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

- **Process classes** implement lower-level business abstractions required to fully manage the business the business domain classes.

- **Persistent classes** represent data stores(e.g., a database) that will persistent beyond the execution of the software.

- **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment .

Characteristics of Design class

- Complete and Sufficient
- Primitiveness
- High Cohesion
- Low Coupling

1. Complete and Sufficient

- A design class should have all the attributes and methods necessary to fulfill its responsibilities.
- It should not depend on outside elements to perform its main functions.
- Example: A Student class should contain data (name, roll number, grades) and methods (calculate GPA, display details) to manage student information completely.

2. Primitiveness

- Each operation (method) in a class should be defined at the lowest useful level of abstraction.
- This means methods should be simple, clear, and directly implementable without unnecessary complexity.
- Example: Instead of one huge method handling multiple tasks, have smaller methods like addMarks(), calculateGPA(), etc.

3. High Cohesion

- A class should focus on a **single, well-defined purpose**.
- All its attributes and methods should be closely related to that purpose.
- High cohesion makes a class **easier to understand, maintain, and reuse**.
- **Example:** A LibraryBook class should only deal with book-related data and actions, not unrelated tasks like user authentication.

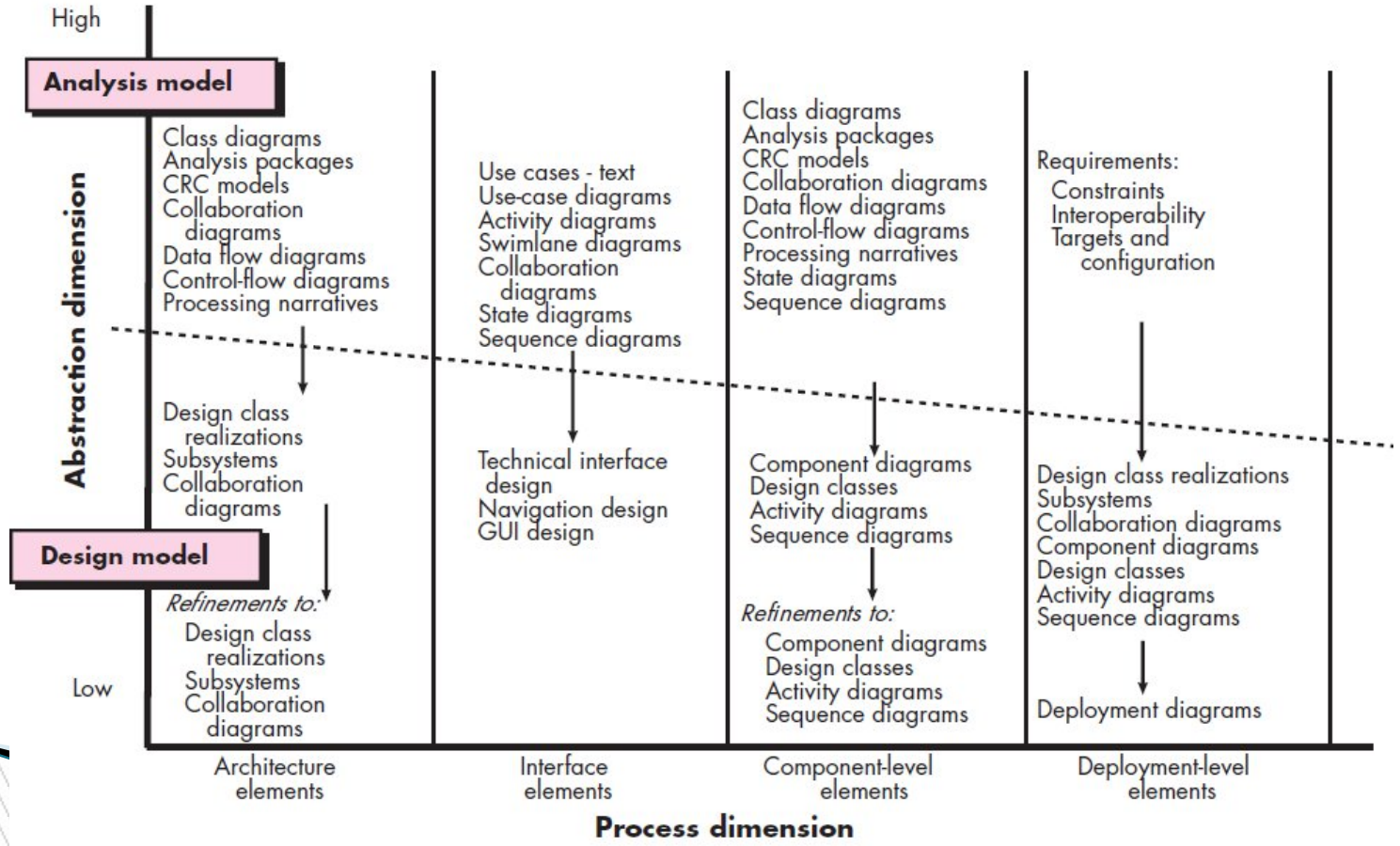
4. Low Coupling

- Classes should be **independent** and have minimal reliance on other classes.
- Low coupling ensures that a change in one class has **little or no effect** on others.
- This improves **modularity, flexibility, and maintainability**.
- **Example:** A Payment class should not directly depend on how a User class stores details; it should just use an interface or required data.

Design Model

The design model can be viewed in two different dimensions

- *process dimension*
- *abstraction dimension*



Elements in design model

- Data design elements
- Architectural design elements
- Interface design elements
- Component level design elements
- Deployment level design elements

Design Model

Data Design Elements

- High level abstraction data (cuctomer/ user's view)
- The structure of data is important part of software design.
- Design of data structures is based on this data model.
- The translation of a data model into a database is crucial.

Design Model

Architectural Design Elements

Architectural model is designed from three sources.

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

- The architectural design element is usually depicted as a set of interconnected subsystems.

Design Model

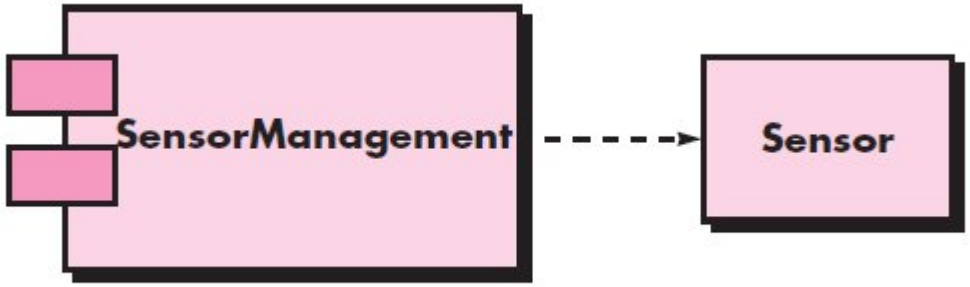
Interface Design Elements

- The interface design elements for software depict information flows into and out of the system and how it 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, devices, networks, or other producers or consumers of information
  3. internal interfaces between various design components.
- UI design is developed (layout, color, graphics, interaction mechanisms)

Design Model

Component Level Design Elements

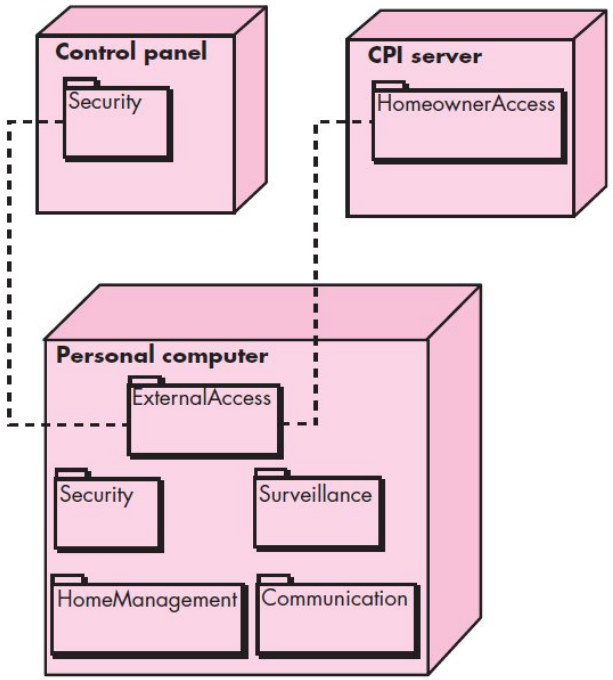
- The component-level design for software fully describes the internal detail of each software component.



Design Model

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



UNIT III – PART 2

SOFTWARE ARCHITECTURE

What is architecture ?

it is a representation that enables you to

(1) analyze the effectiveness of the design in meeting its stated requirements,

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

(3) reduce the risks associated with the construction of the software.

ARCHITECTURAL STYLES

- It is a template for creating a base design. Each architecture style has a set of components, connectors, constraints, semantic models.
- We have 5 different styles of software architecture.

1. Data-centered architecture – multiple clients one data repository
2. Data Flow architecture – Pipes and filters
3. Call and return architecture – Request /acknowledgement
4. Object oriented architecture – Encapsulation, message passing
5. Layered architecture – Core layer followed by subsequent layers.

FIGURE 9.1

Data-centered architecture

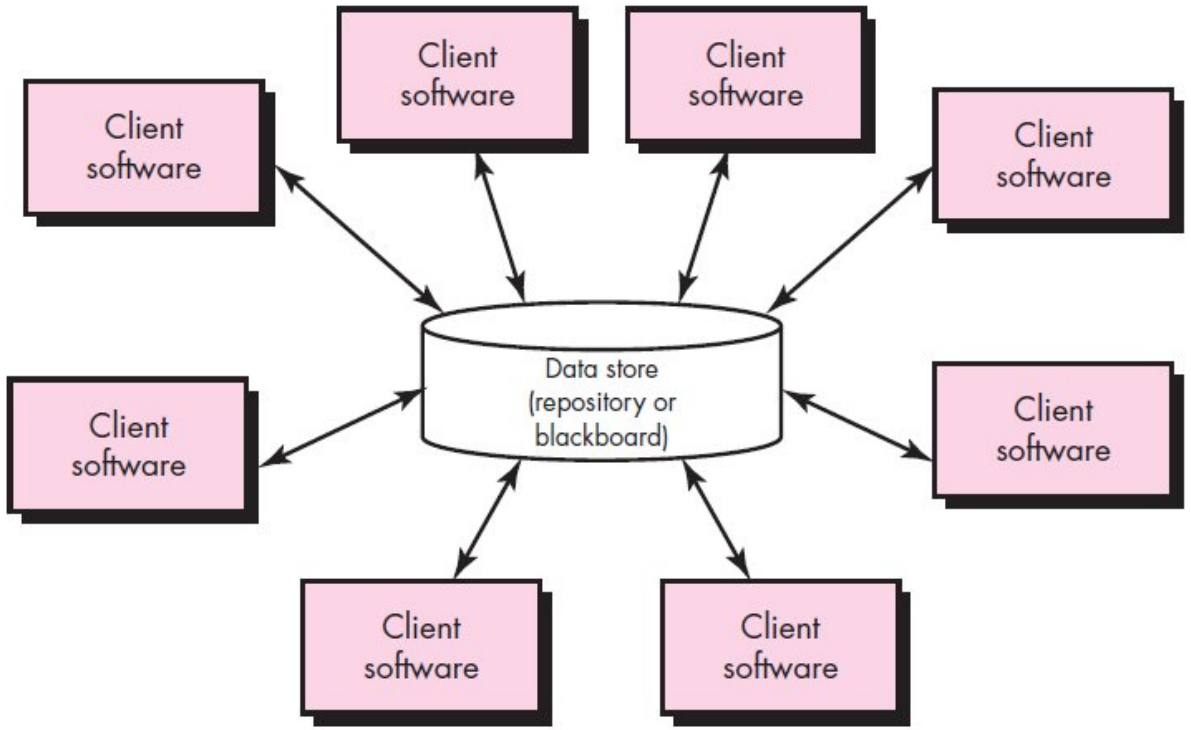
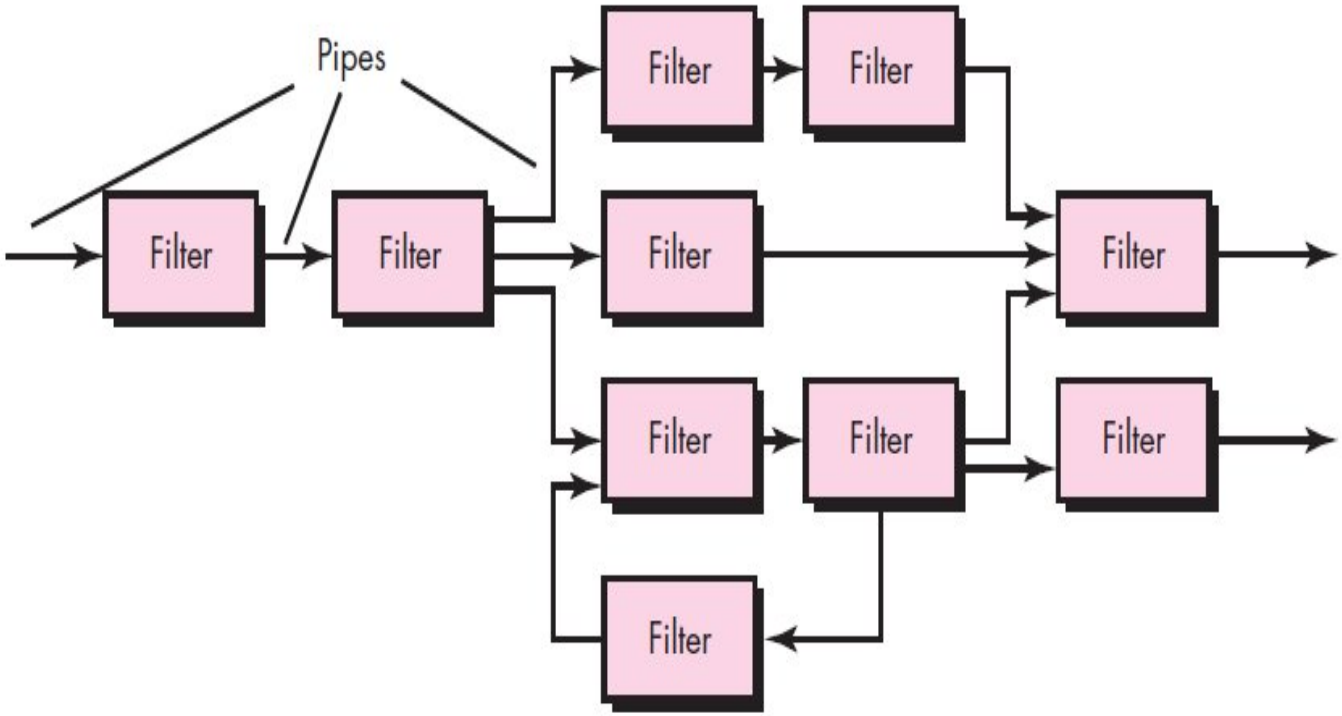


FIGURE 9.2

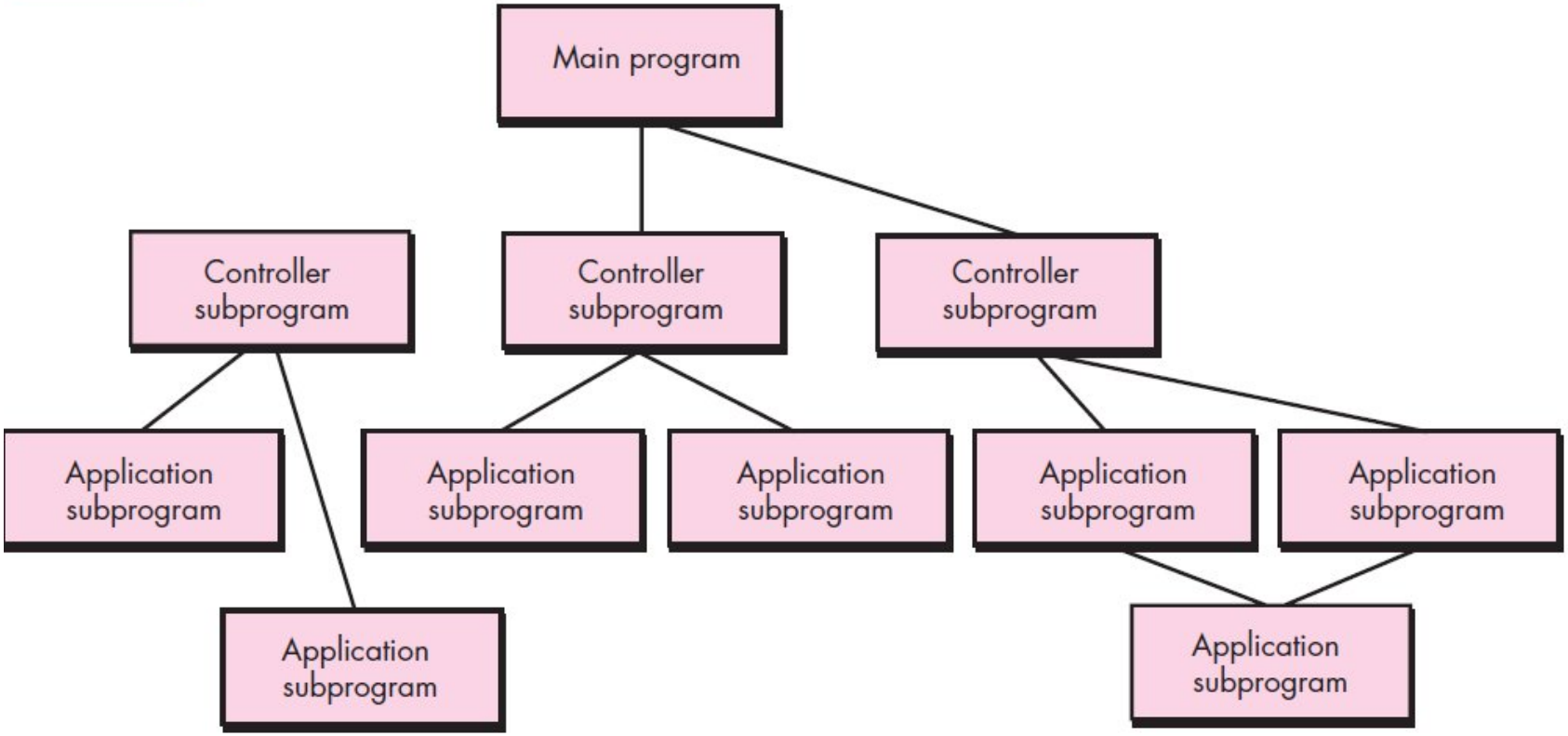
Data-flow architecture



Pipes and filters

Call and return

FIGURE 9.3 Main program/subprogram architecture



Object oriented architecture

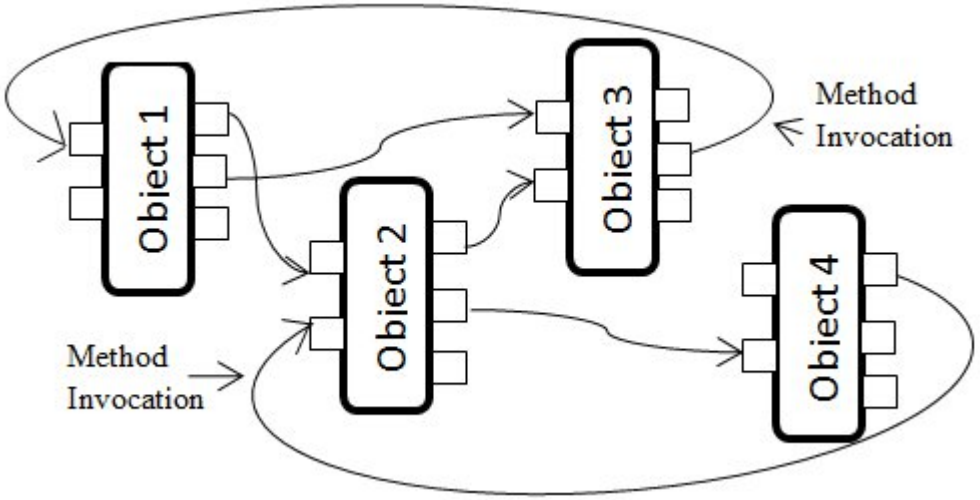
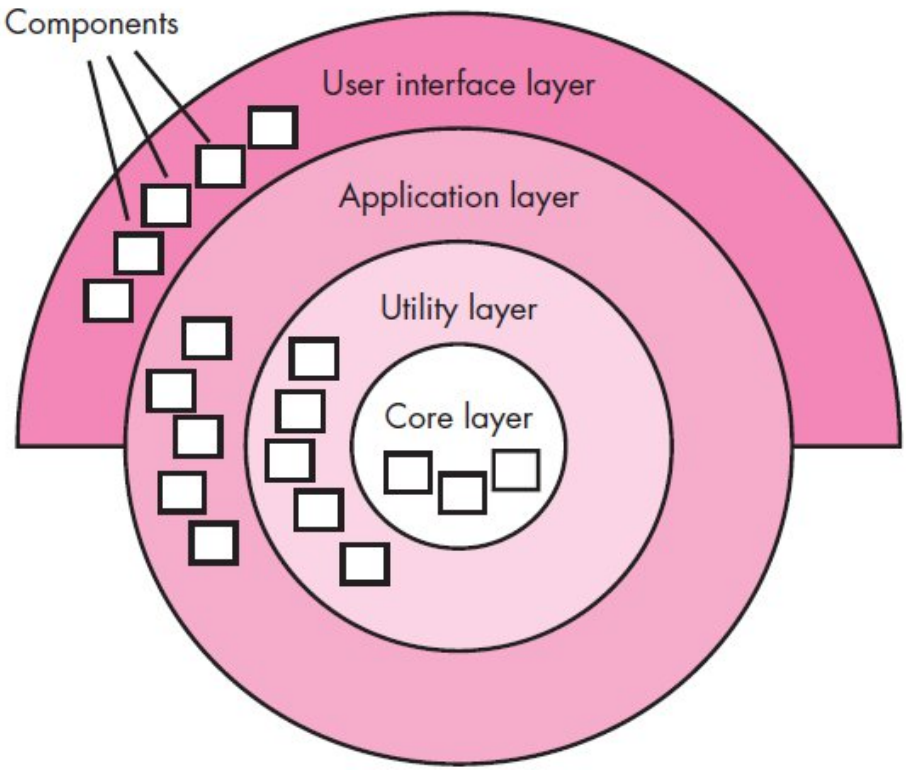


FIGURE 9.4

Layered architecture



Architectural Design

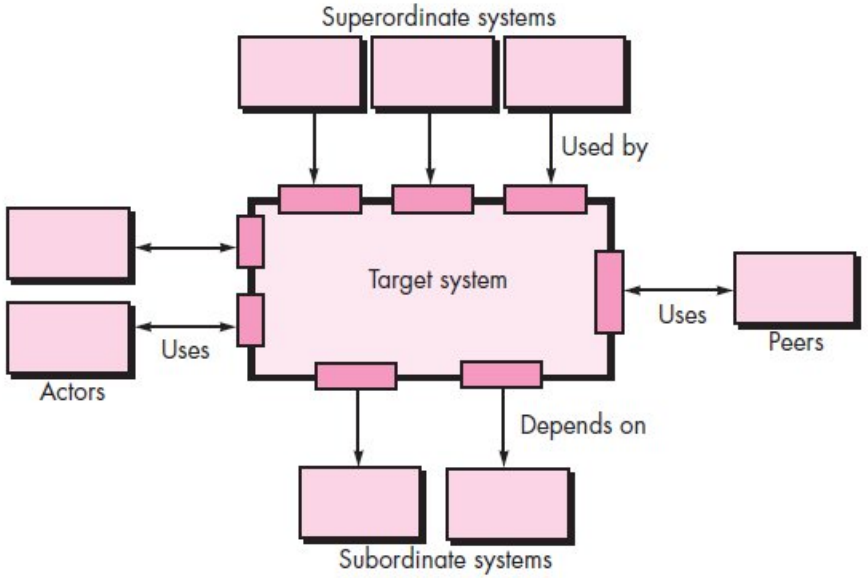
- Architectural design will involve all the **external entities** connect to our product along with the internal details.
- **Archetype** is an abstraction which describes a specific type of system behaviour. (only external behaviour not internal).
- Design is done by collecting all the information about product from requirements phase and putting them into exact archetype and architectural style.
- There are 4 steps in architectural design.

Architectural Design

1. Representing the System in Context

- Arc
  beh

FIGURE 9.5
Architectural context diagram
Source: Adapted from [Bos00].

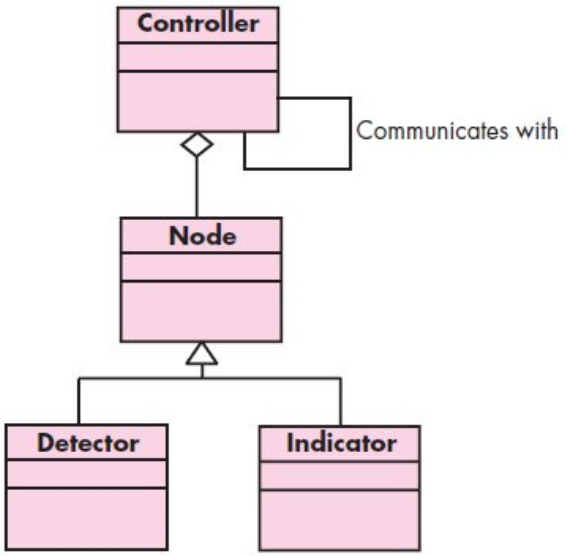


Architectural Design

2. Defining Archetypes

- Archetype is a class that gives high level abstraction for a particular function or feature.

FIGURE 9.7
UML relationships for *SafeHome* security function archetypes
Source: Adapted from [Bos00].



Architectural Design

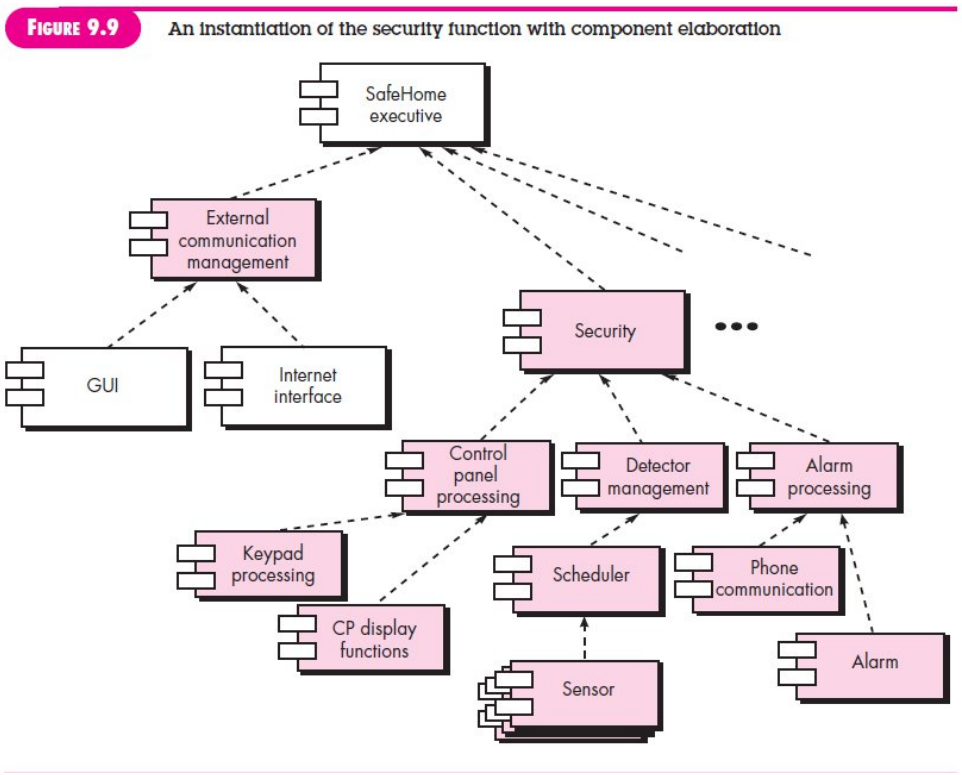
3. Refining the Architecture into Components

- The real architecture begins from component design.
- Components are chosen from two domains.
  - Application domain
  - Infrastructure domain
- **Example : Safe Home Detector**
  - *External communication management*—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
  - *Control panel processing*—manages all control panel functionality.
  - *Detector management*—coordinates access to all detectors attached to the
  - system.
  - *Alarm processing*—verifies and acts on all alarm conditions.

Architectural Design

4. Describing Instantiations of the System

All the chosen components and archetypes are assembled / incorporated into single diagram and final design is completed.



ARCHITECTURAL MAPPING USING DATA FLOW

- Requirement analysis will give us text based information of required product.
- Entire requirement cannot directly fit into the 5 types of architectural styles available.
- Many times there is a combination of styles related to a single product.
- To convert the requirements into design and fit them into a unique architectural style, there is no standard format or a generic mapping.
- There are a set of steps when followed properly, a quality design model output is produced which can be considered as "**Architectural Mapping**".

ARCHITECTURAL MAPPING USING DATA FLOW

Data Flow Diagram

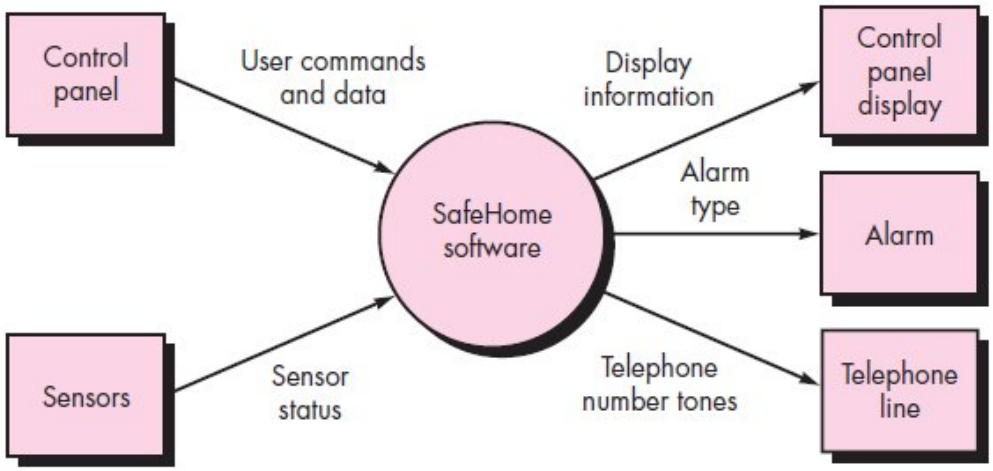
- Data Flow Diagram (DFD) is a graphical tool that visualizes how data moves through a system by depicting the flow of information between processes, data storage, and external entities.
- Data flow diagram is represented in levels, below is a level 0 DFD

ARCHITECTURAL MAPPING USING DATA FLOW

- There are 7 steps in our Transformational mapping, which converts a DFD into a design model and fits it into one of the architectural styles.
- Example : SafeHome security system

- **Step 1. Review the fundamental system model. –** Level 0 DFD of SafeHome security system

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

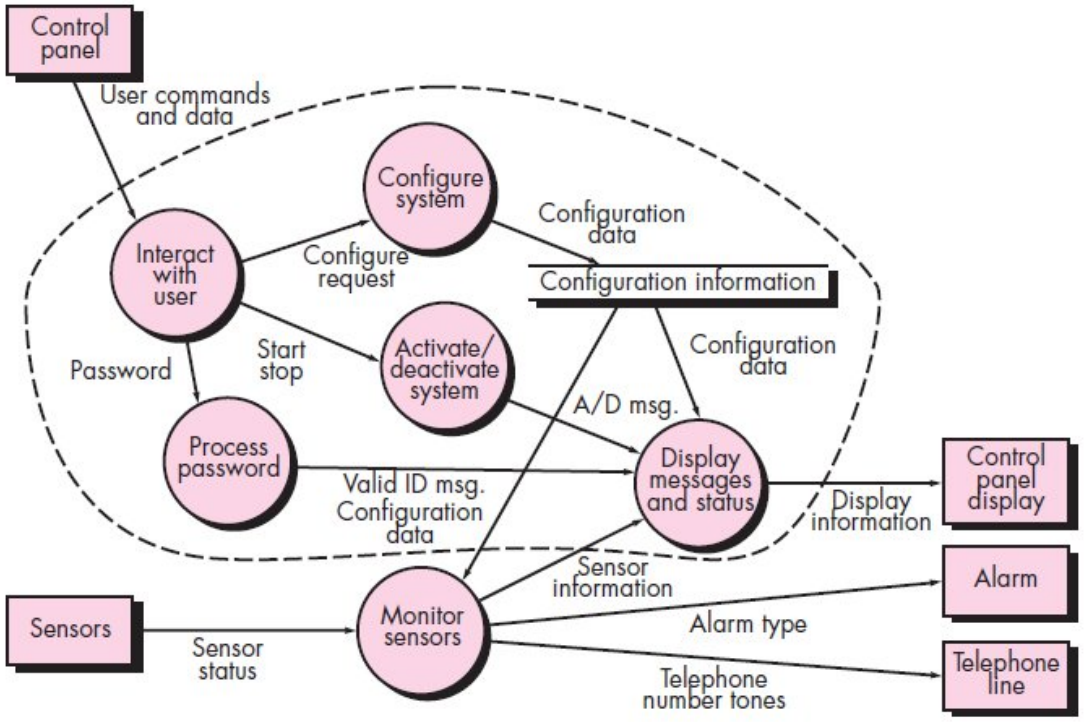


ARCHITECTURAL MAPPING USING DATA FLOW

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

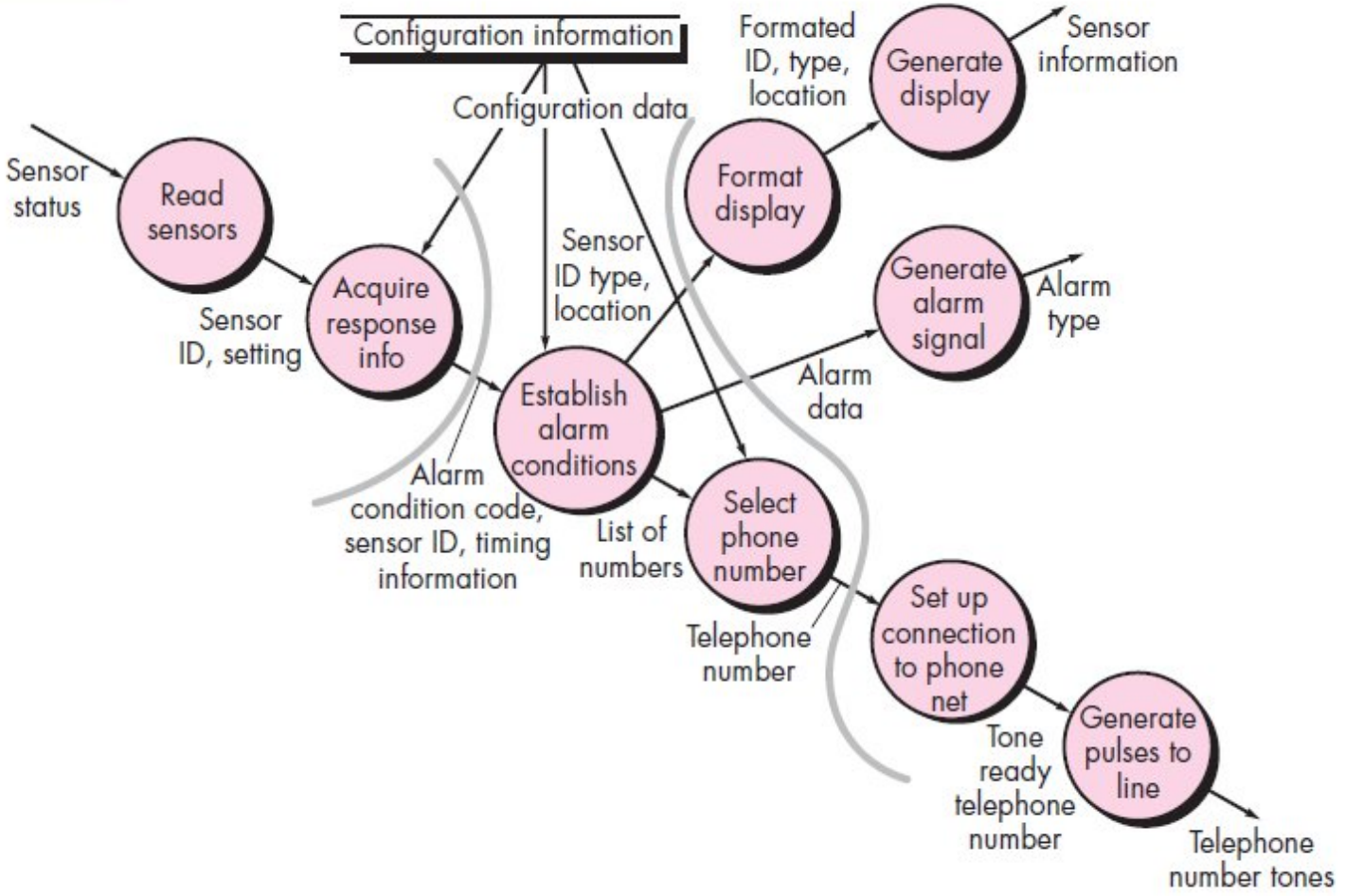
Info detail.
Lev

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



ARCHITECTURAL MAPPING USING DATA FLOW

FIGURE 9.13 Level 3 DFD for *monitor sensors* with flow boundaries



ARCHITECTURAL MAPPING USING DATA FLOW

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

  Check whether all data flows are clear and unambiguous.

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

  Set good boundaries for the data, no data must be lost in between

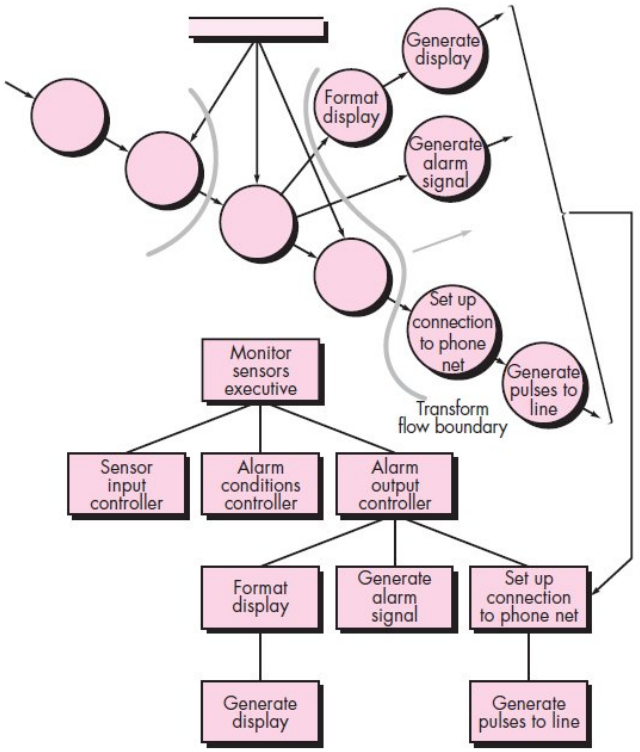
  Clearly specify the incoming and outgoing lines of data.

- **Step 5. Perform "first-level factoring."**
  - Organize the components into levels
  - top-level components perform decision making
  - Middle-level components perform some control and do moderate amounts of work.
  - Lowlevel components perform most input, computation, and output work

ARCHITECTURAL MAPPING USING DATA FLOW

- **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 the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordina

FIGURE 9.15
Second-level factoring for *monitor sensors*



ARCHITECTURAL MAPPING USING DATA FLOW

- **Step 7. Refine the 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 or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.