

# UNIT 4

# Fundamental Software Design Concepts

## Abstraction

- A solution is stated in large terms using the language of the problem environment at the highest level abstraction.
- The lower level of abstraction provides a more detail description of the solution.
- A sequence of instruction that contain a specific and limited function refers in a procedural abstraction.
- A collection of data that describes a data object is a data abstraction.

# Architecture

- The complete structure of the software is known as software architecture.
- Structure provides conceptual integrity for a system in a number of ways.
- The architecture is the structure of program modules where they interact with each other in a specialized way.
- The components use the structure of data.
- The aim of the software design is to obtain an architectural framework of a system.
- The more detailed design activities are conducted from the framework

## Patterns

- A design pattern describes a design structure and that structure solves a particular design problem in a specified content.

## Modularity

- A software is separately divided into name and addressable components. Sometime they are called as modules which integrate to satisfy the problem requirements.
- Modularity is the single attribute of a software that permits a program to be managed easily.

## Information hiding

- Modules must be specified and designed so that the information like algorithm and data presented in a module is not accessible for other modules not requiring that information.

## Functional independence

The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.

The functional independence is accessed using two criteria i.e Cohesion and coupling.

## Cohesion

Cohesion is an extension of the information hiding concept.

A cohesive module performs a single task and it requires a small interaction with the other components in other parts of the program.

## Coupling

Coupling is an indication of interconnection between modules in a structure of software.

## Refinement

- Refinement is a top-down design approach.
- It is a process of elaboration.
- A program is established for refining levels of procedural details.
- A hierarchy is established by decomposing a statement of function in a stepwise manner till the programming language statement are reached.

## Refactoring

- It is a reorganization technique which simplifies the design of components without changing its function behaviour.
- Refactoring is the process of changing the software system in a way that it does not change the external behaviour of the code still improves its internal structure.

## Design Classes

- The model of software is defined as a set of design classes.
- Every class describes the elements of problem domain and that focus on features of the problem which are user visible.

## User interface classes

- Define all abstractions that are necessary for human computer interaction.
- Business domain classes are often refinements of the analysis classes.
- The classes identify the attributes and services that are required to implement some element of the business domain.
- Process classes implement lower-level business abstractions required to fully manage the business domain classes.

- Persistent classes represent data stores that will persist 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.

They define four characteristics of a well formed design class

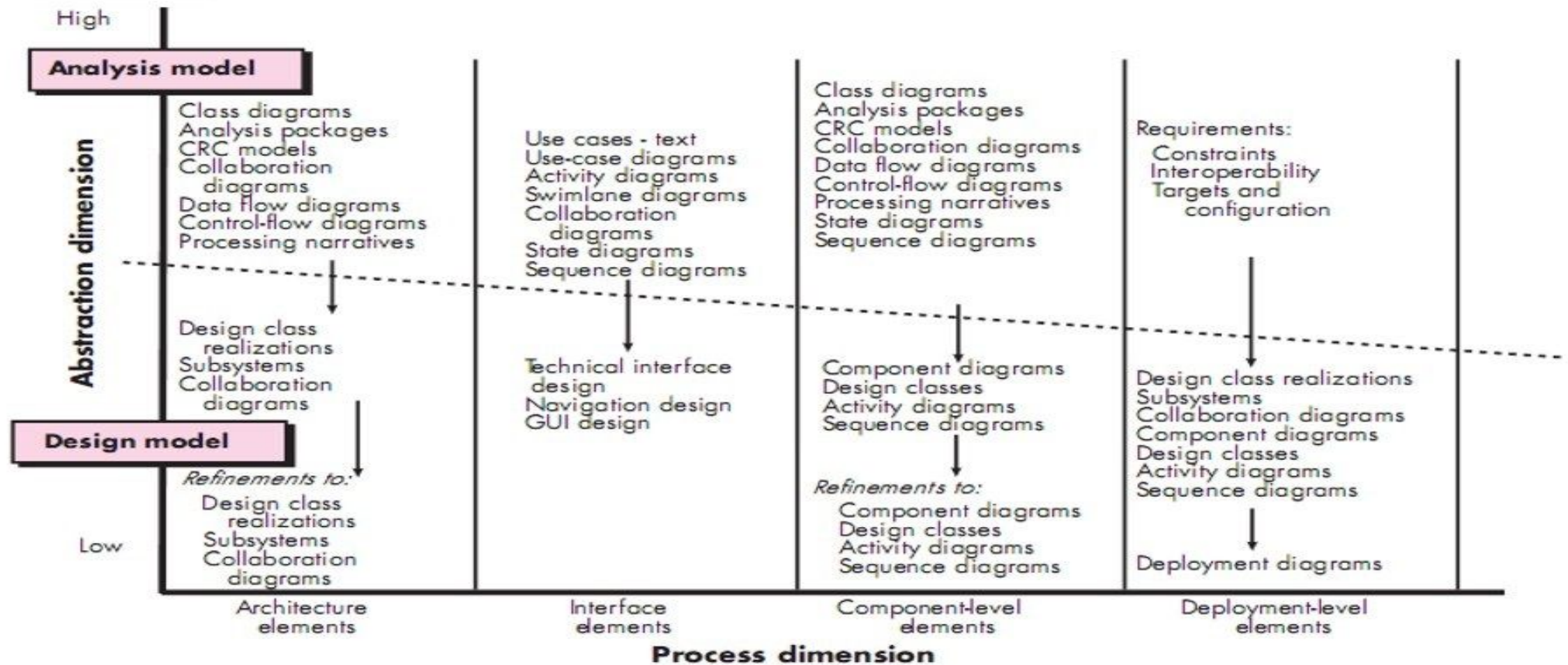
- Complete and sufficient
- Primitiveness
- High cohesion
- Low coupling



# Design Model

- Design modeling in software engineering represents the features of the software that helps engineer to develop it effectively, the architecture, the user interface, and the component level detail.
- Design modeling provides a variety of different views of the system like architecture plan for home or building.
- Different methods like data-driven, pattern-driven, or object-oriented methods are used for constructing the design model. All these methods use set of design principles for designing a model.

## Dimensions of the design model



# Design Model Elements

- Data elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural elements
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and “styles”
- Interface elements
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- Component elements
- Deployment elements

# Data Modeling

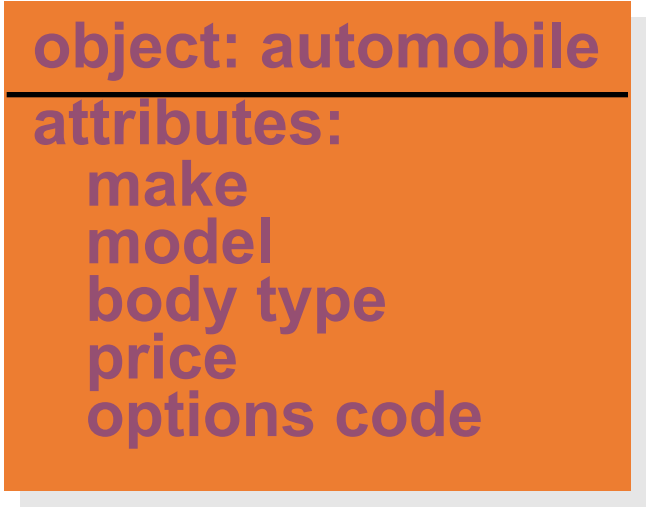
- examines data objects independently of processing
- focuses attention on the data domain
- creates a model at the customer's level of abstraction
- indicates how data objects relate to one another

# What is a Data Object?

- a representation of almost any composite information that must be understood by software.
  - *composite information*—something that has a number of different properties or attributes
- can be an **external entity** (e.g., anything that produces or consumes information), a **thing** (e.g., a report or a display), an **occurrence** (e.g., a telephone call) or **event** (e.g., an alarm), a **role** (e.g., salesperson), an **organizational unit** (e.g., accounting department), a **place** (e.g., a warehouse), or a **structure** (e.g., a file).
- The description of the data object incorporates the data object and all of its attributes.
- A data object encapsulates data only—there is no reference within a data object to operations that act on the data.

# Data Objects and Attributes

A data object contains a set of attributes that act as an aspect, quality, characteristic, or descriptor of the object



```
object: automobile
attributes:
  make
  model
  body type
  price
  options code
```

# What is a Relationship?

- Data objects are connected to one another in different ways.
  - A connection is established between **person** and **car** because the two objects are related.
    - A person *owns* a car
    - A person *is insured to drive* a car
- The relationships *owns* and *is insured to drive* define the relevant connections between **person** and **car**.
- Several instances of a relationship can exist
- Objects can be related in many different ways

# Architectural Elements

- The architectural model [Sha96] is derived 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, and
  - the availability of architectural patterns (Chapter 16) and styles (Chapter 13).



# Interface Elements

- Interface is a set of operations that describes the externally observable behavior of a class and provides access to its public operations
- Important elements
  - User interface (UI)
  - External interfaces to other systems
  - Internal interfaces between various design components
- Modeled using UML communication diagrams (called collaboration diagrams in UML 1.x)

# Interface Elements

## UML Interface representation for control panel

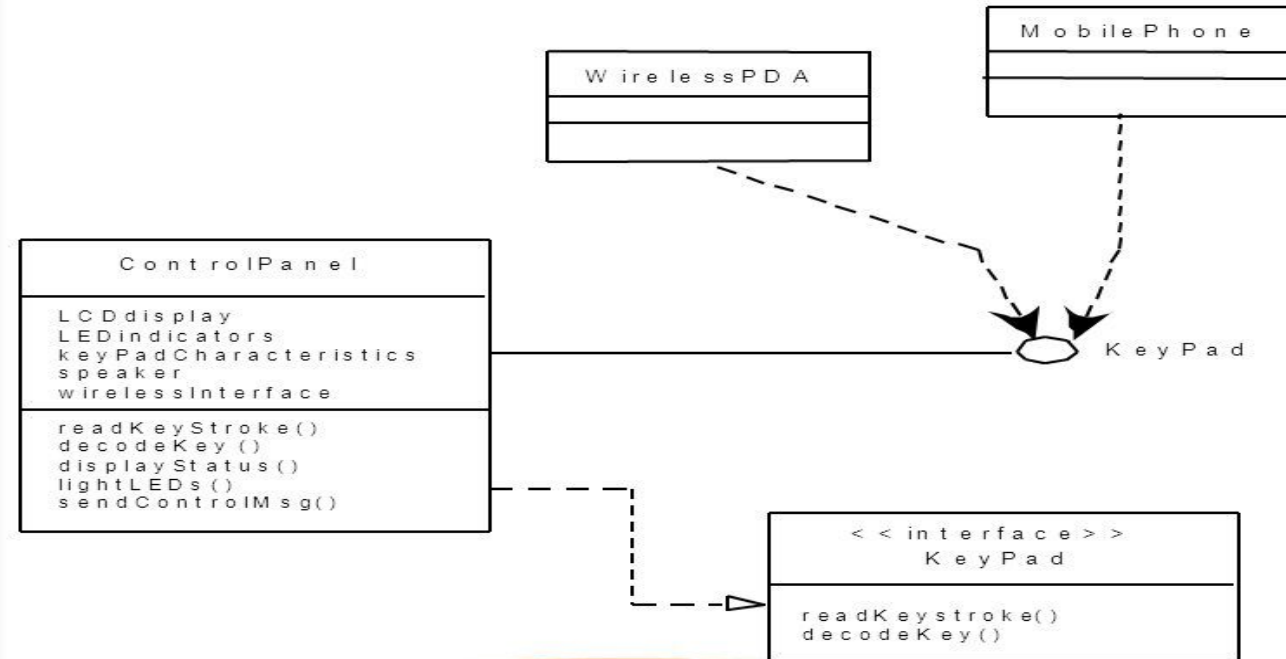
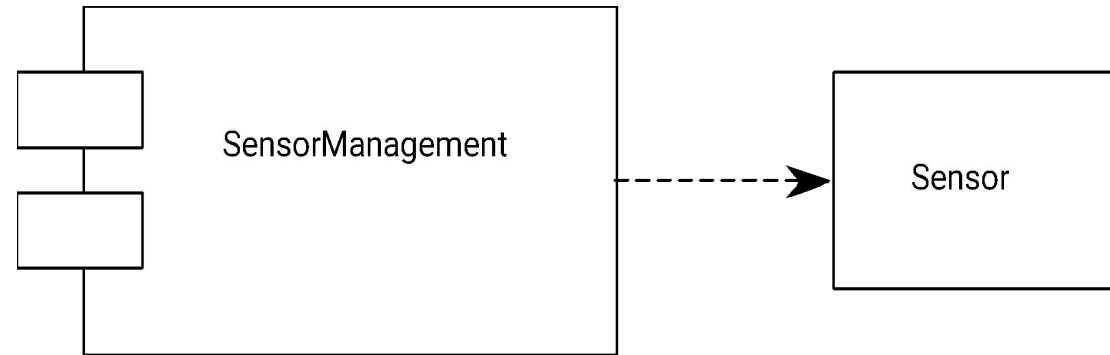


Figure 9.6 UML interface representation for ControlPanel

# Component Elements

- Describes the internal detail of each software component
- Defines
  - Data structures for all local data objects
  - Algorithmic detail for all component processing functions
  - Interface that allows access to all component operations
- Modeled using UML component diagrams, UML activity diagrams, pseudocode (PDL), and sometimes flowcharts

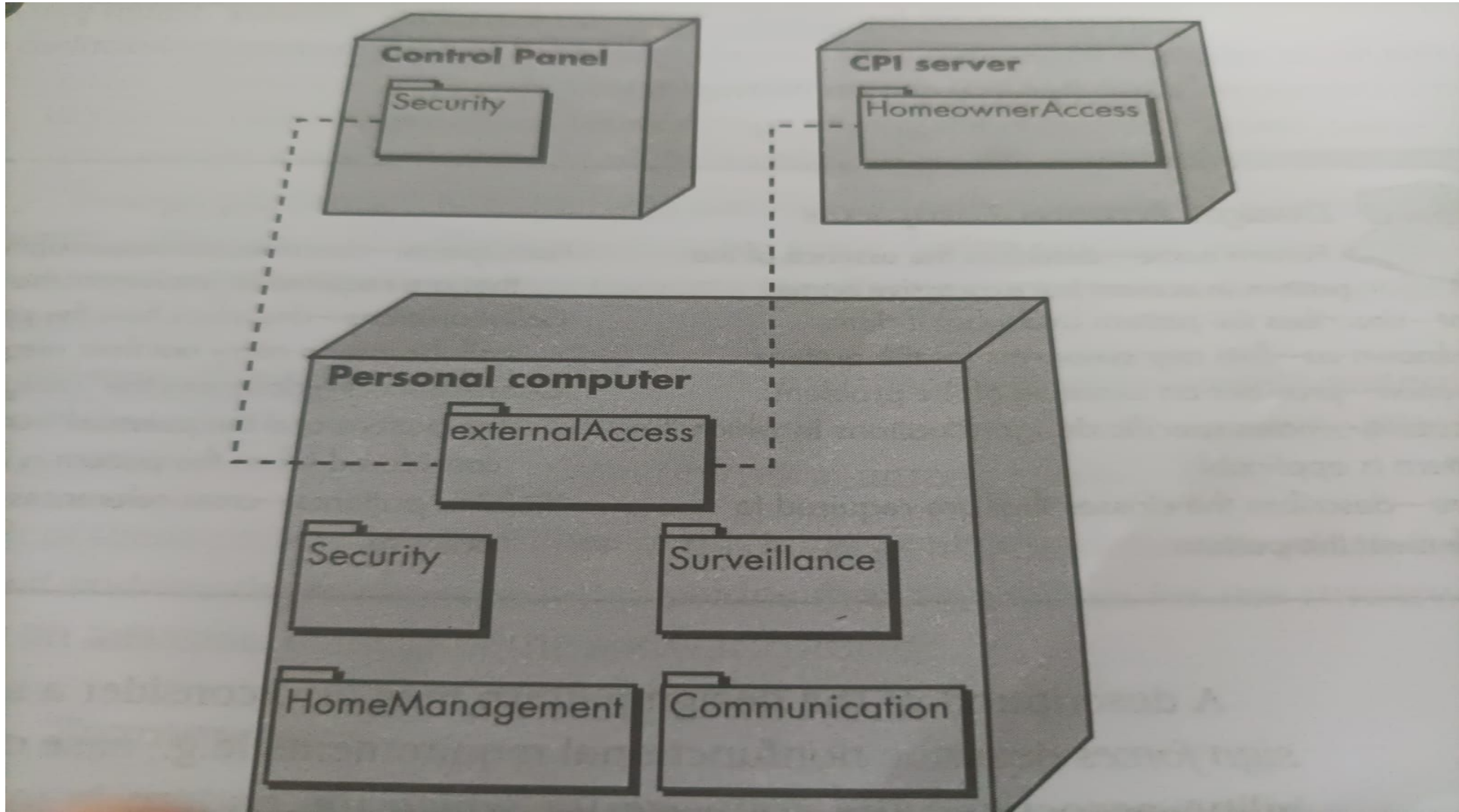
# Component Elements



# Deployment Elements

- Indicates how software functionality and subsystems will be allocated within the physical computing environment
- Modeled using UML deployment diagrams
- *Descriptor form* deployment diagrams show the computing environment but does not indicate configuration details
- *Instance form* deployment diagrams identifying specific named hardware configurations are developed during the latter stages of design

# Deployment Elements



# Why Architecture?

The architecture is not the operational software. Rather, it is a representation that enables a software engineer 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, and
- (3) **reduce the risks** associated with the construction of the software.

## Why is Architecture Important?

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture “constitutes a relatively small, intellectually graspable mode of how the system is structured and how its components work together”



# Architectural Descriptions

- The IEEE Computer Society has proposed IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive System*
  - to establish a conceptual framework and vocabulary for use during the design of software architecture,
  - to provide detailed guidelines for representing an architectural description and
  - to encourage sound architectural design practices.
- The IEEE Standard defines an *architectural description* (AD) as a “a collection of products to document an architecture.”
  - The description itself is represented using multiple views, where each *view* is “a representation of a whole system from the perspective of a related set of [stakeholder] concerns.”

# Data Design

The data design action translates data defined as part of the analysis model into data structures at the software component level and. When necessary into a database architecture at the application level.

## Data Design at the Architectural Level

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

To solve this challenge, the business IT community has developed data mining techniques, also called knowledge discovery in database (KDD) , that navigate through existing databases in an attempt to extract appropriate business-level information. An alternative solution, called a data warehouse, adds an additional layer to the data architecture.

A data warehouse is a separate data environment that is not directly integrated with day –to-day application but encompasses all data used by a business.

# Data Design at the Component Level

Data design at the component level focuses on the representation of the data structures that are directly accessed by one or more software components. We consider the following set of principles (adapted from for data specification):

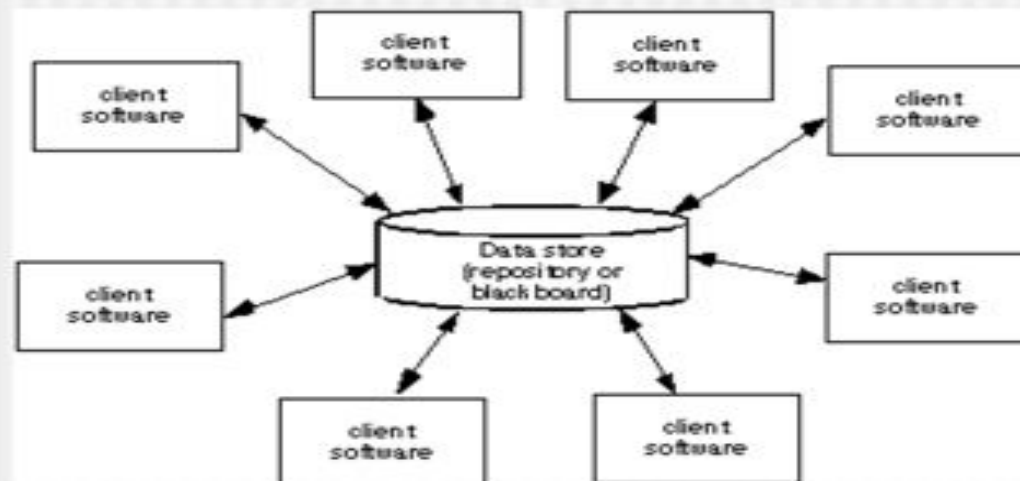
1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structure and the operations to be performed on each should be identified.
3. A mechanism for defining the content of each data object should be established and used to define both data and the operation applied it.
4. Low-level design decision should be known only to those modules that must make direct use of the data contained within the structure.
5. The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

# Architectural Styles

Each style describes a system category that encompasses: (1) a **set of components** (e.g., a 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 the system, and (4) **semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

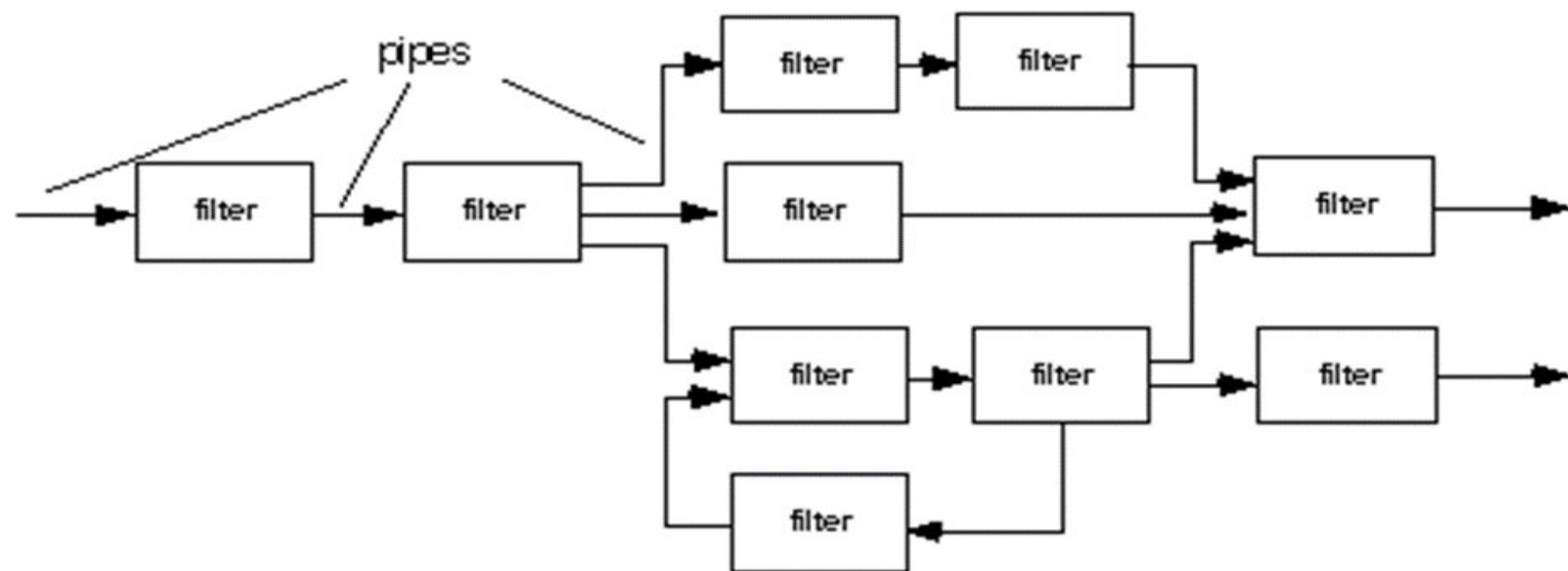
- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

## Data-Centered Architecture

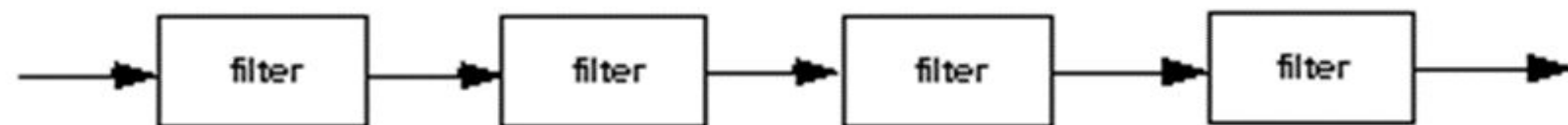


# Data-flow architecture

- The architecture is applied when input data are to be transformed through a series of computational or manipulate components into output data.
- A pipe and filter structure has a set of components, called filters, connected by pipes that transmit data from one component to accept data input of a certain form , and produces data output ( to the next filters) of a specified form.
- However, the filter does not require knowledge of the workings of its neighboring filters.



(a) pipes and filters



(b) batch sequential

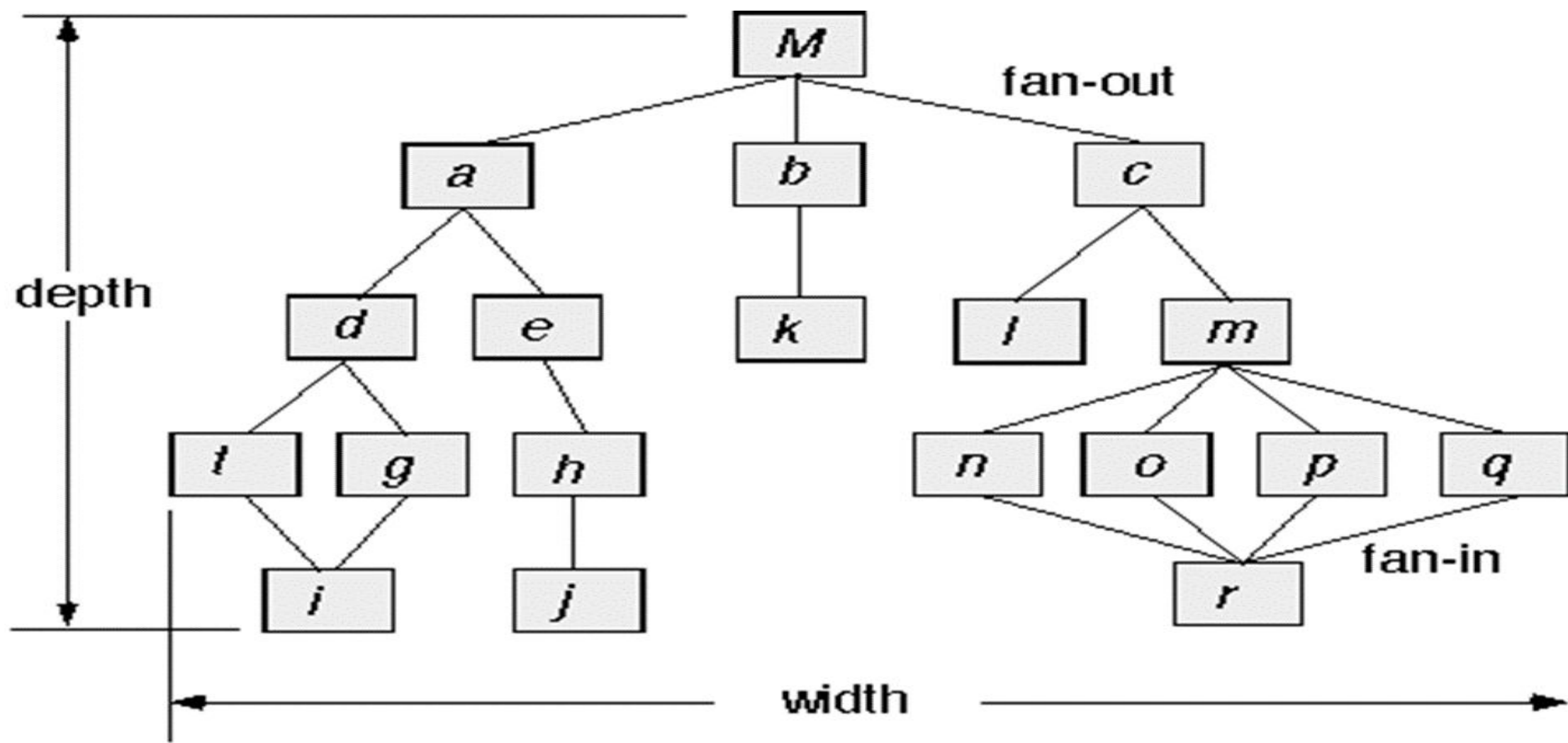
# Call and return architecture

- This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale.

## Two sub styles within category:

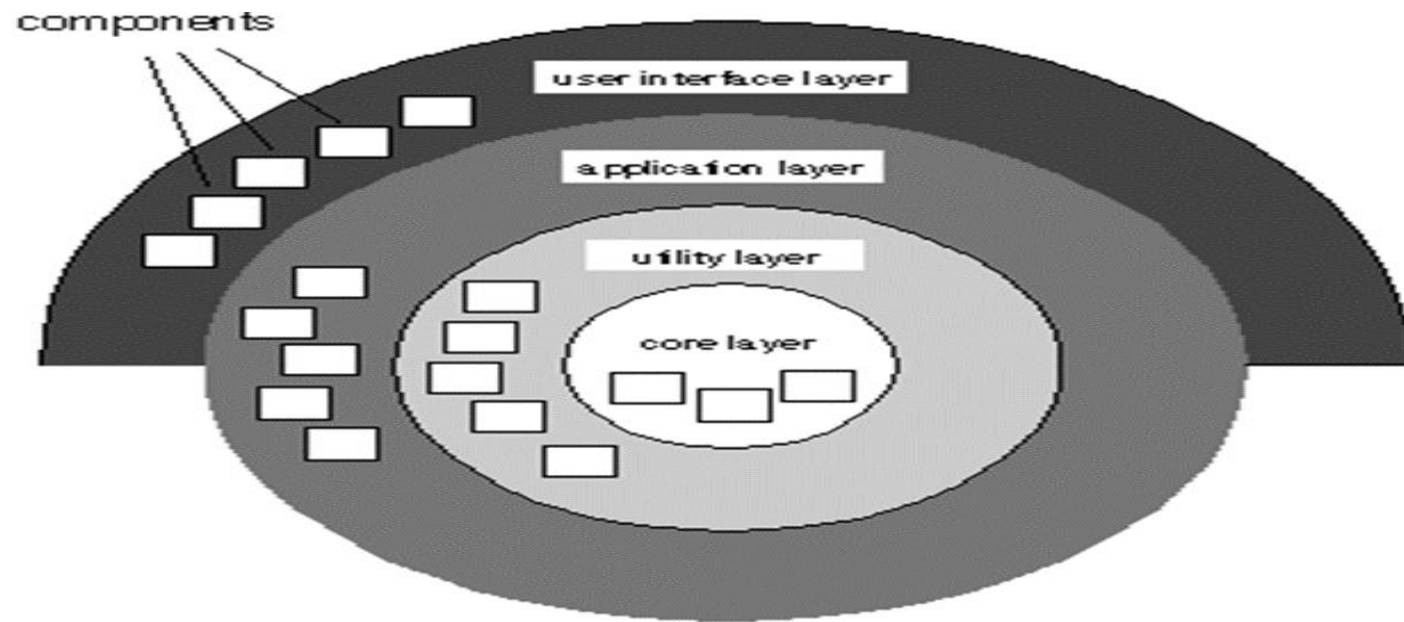
- Main programs/ subprograms architecture. This classic program structure decomposes functions into a control hierarchy where a “main” program invokes a number of program components, which in turn may invoke still other components.
- Remote procedure called architecture. The components of a main program/sub- program architecture are distributed across multiple computers on a network.





## Object-oriented architecture

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



# Architectural Patterns

A S/W architecture may have a number of architectural patterns that address issues such as concurrency, persistence, and distribution.

**Concurrency**—applications must handle multiple tasks in a manner that simulates parallelism

- operating system process management pattern
- task scheduler pattern

**Persistence**—Data persists if it survives past the execution of the process that created it. Persistent data are stored in a database or file and may be read and modified by other processes at a later time.

Two patterns are common:

- a database management system pattern that applies the storage and retrieval capability of a DBMS to the application architecture
- an application level persistence pattern that builds persistence features into the application architecture

**Distribution**— the manner in which systems or components within systems communicates with one another in a distributed environment, and the nature of the communication that occurs.

□ A broker acts as a 'middle-man' between the client component and a server component.



南京大學  
SOFTWARE INSTITUTE  
软件学院

Computing  
And

Software Engineering

## 4. Architecture Design —— Organization and Refinement

- Control
  - Procedure calls (local or remote)
  - Events
  - Messages and message buses
  - Client/server middleware
  - Delegate, Adapter
- Data
  - Shared Data
  - Pipes
  - Messages and message buses

# Architectural Design

---

- Establishing the overall structure of a software system
- Objectives
  - To introduce architectural design and to discuss its importance
  - To explain why multiple models are required to document a software architecture
  - To describe types of architectural model that may be used

## Contd...

- 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.

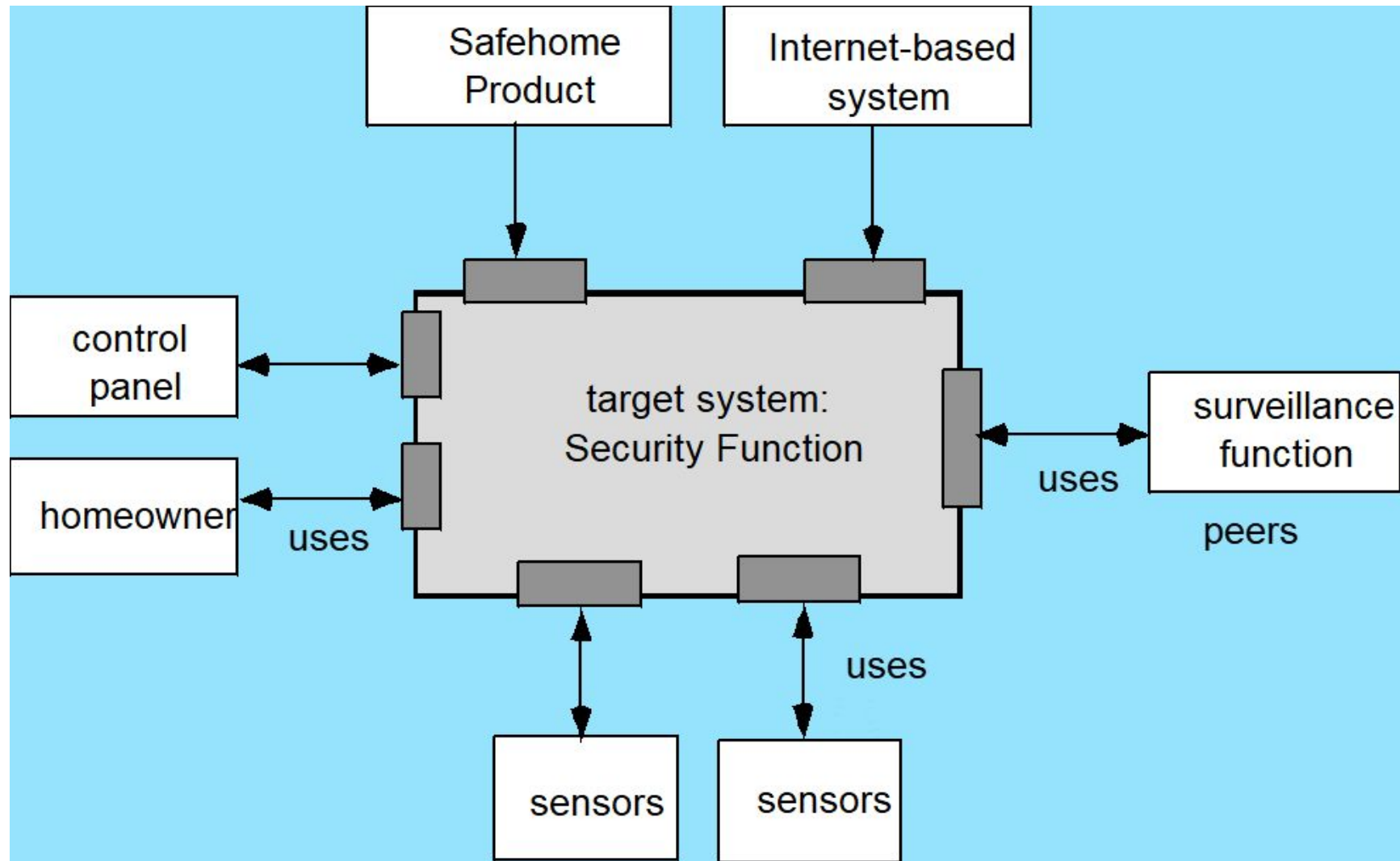


# Representing the System in Context

- At the architectural design level, a software architect uses an **architectural context diagram (ACD)** to model the manner in which software interacts with entities external to its boundaries.

The systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as

- **Super ordinate systems** : **those** systems that use the target system as part of some higher-level processing scheme.
- **Subordinate systems**—**those** systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- **Peer-level systems**—**those** systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system).
- **Actors—entities** (*people, devices*) that interact with the target system by producing or consuming information.





# Defining Archetypes

- An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system.
- In general, a relatively small set of archetypes is required to design even relatively complex systems.
- Archetypes are the abstract building blocks of an architectural design.
- In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model.
- An archetype is a generic, idealized model of a person, object, or concept from which similar instances are derived, copied, patterned, or emulated.
- For example, an archetype for a car: wheels, doors, seats, engine In software engineering

## Contd..

As per in previous figure : The SafeHome home security function, you might define the following archetypes :

**Node** : Represents a cohesive collection of input and output elements of the home security function.

For example a node might be included of (1) various sensors and (2) a variety of alarm (output) indicators.

**Detector** : An abstraction that covers all sensing equipment that feeds information into the target system.

**Indicator**. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

**Controller**. An abstraction that describes the mechanism that allows the arming (Supporting) or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

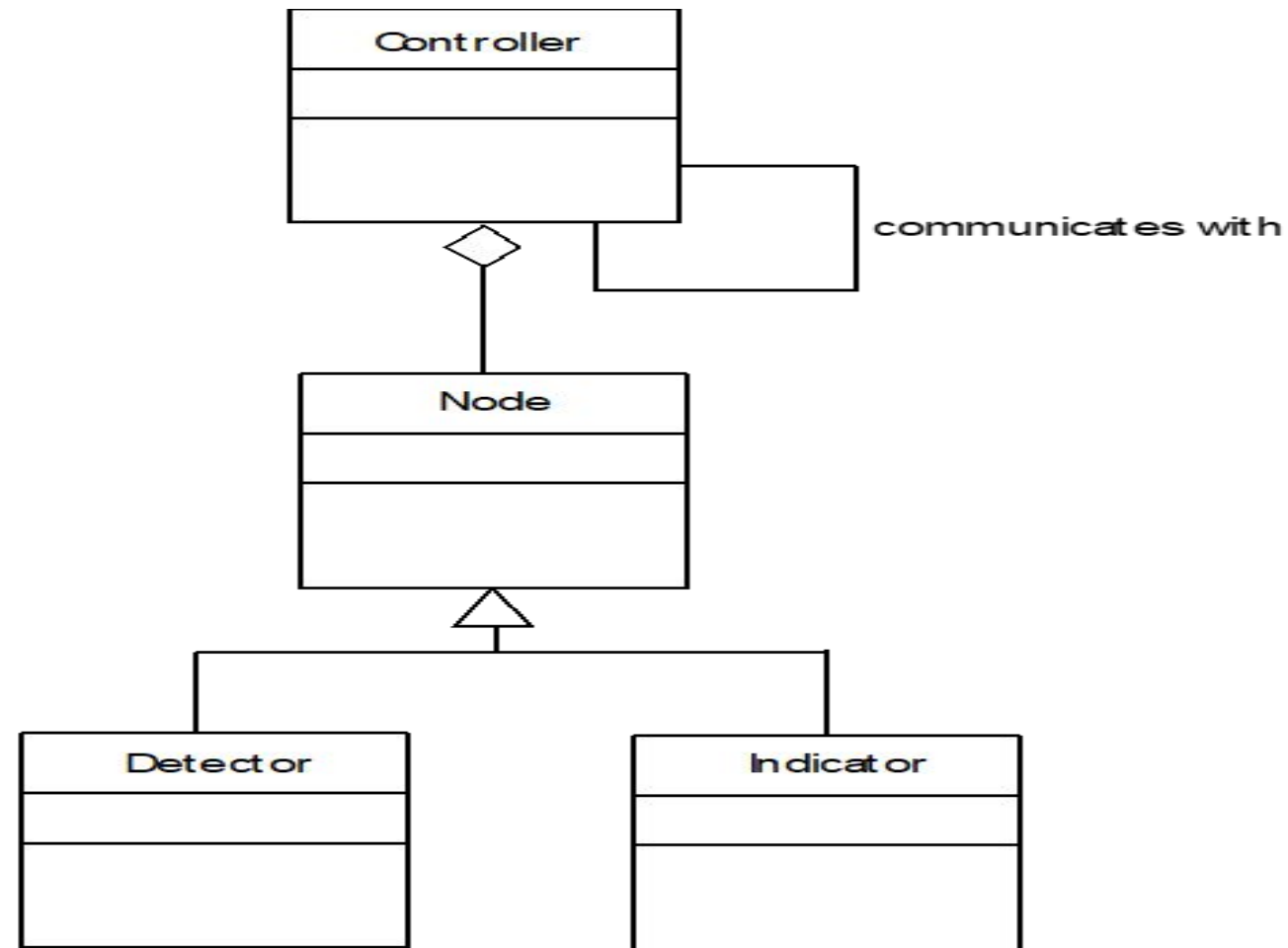
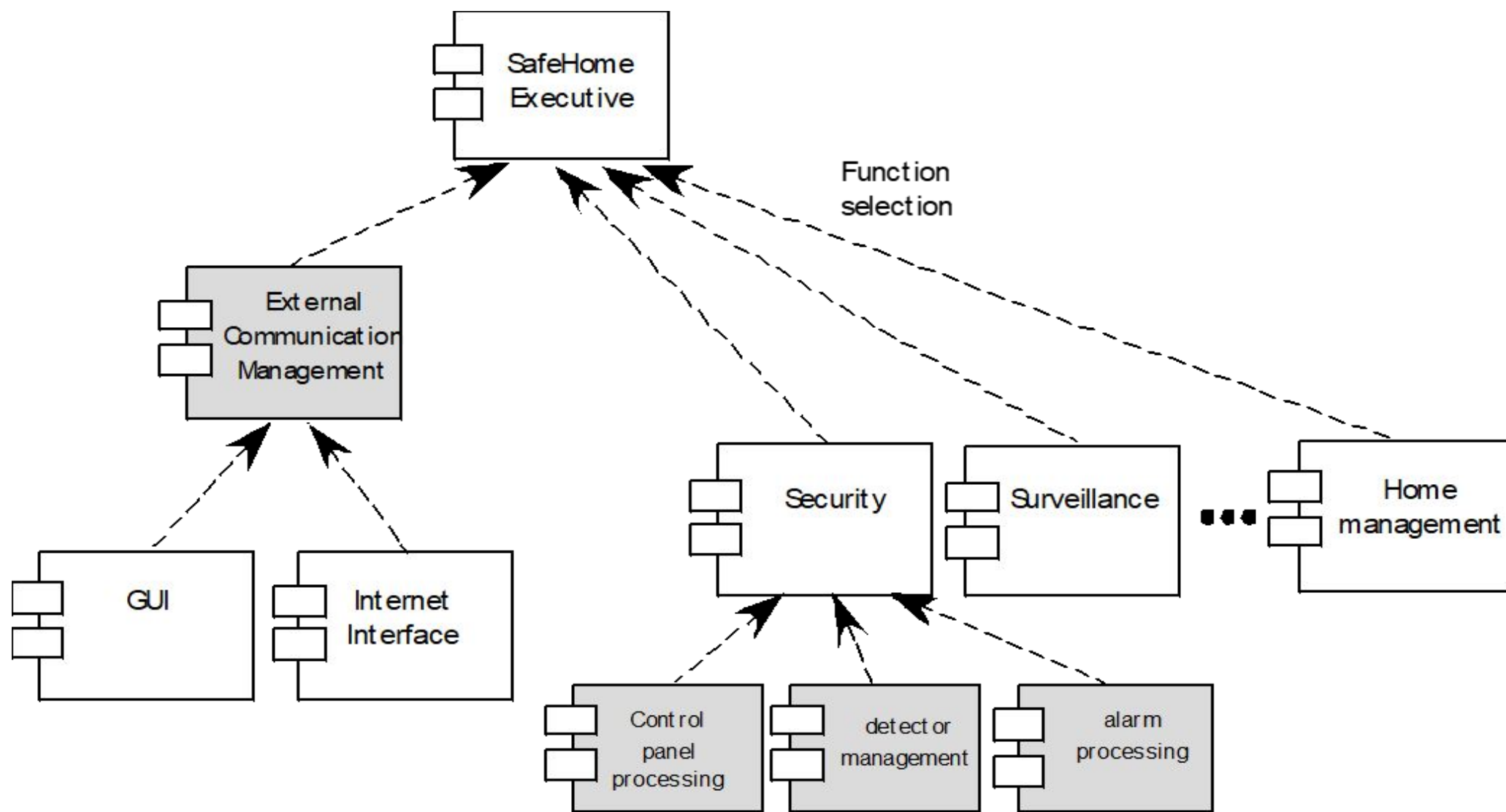


Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])

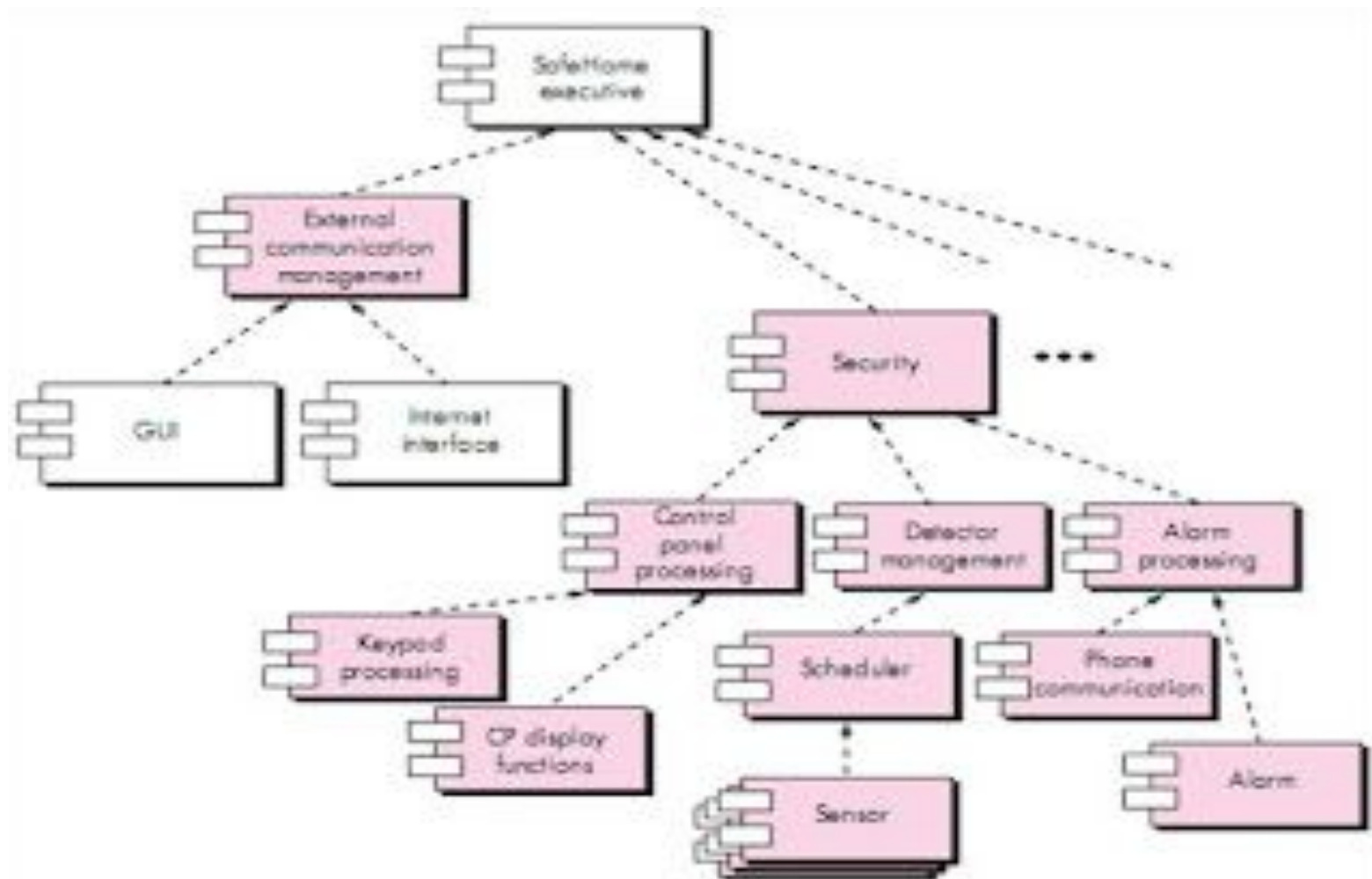
# Refining the Architecture into Components

- ❖ As the software architecture is refined into components.
- ❖ Analysis classes represent entities within the application (business) domain that must be addressed within the software architecture.
- ❖ In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.
- ❖ For Example : The SafeHome home security function example, you might define the set of top-level components that address the following functionality:
- ❖ 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



# Describing Instantiations of the System

- The architectural design that has been modeled to this point is still relatively high level.
- The context of the system has been represented
- Archetypes that indicate the important abstractions within the problem domain have been defined,
- The overall structure of the system is apparent, and the major software components have been identified.
- However, further refinement is still necessary.
- To accomplish this, an actual instantiation of the architecture is developed. It means, again it simplify by more details.



# **Designing Class-Based Components**

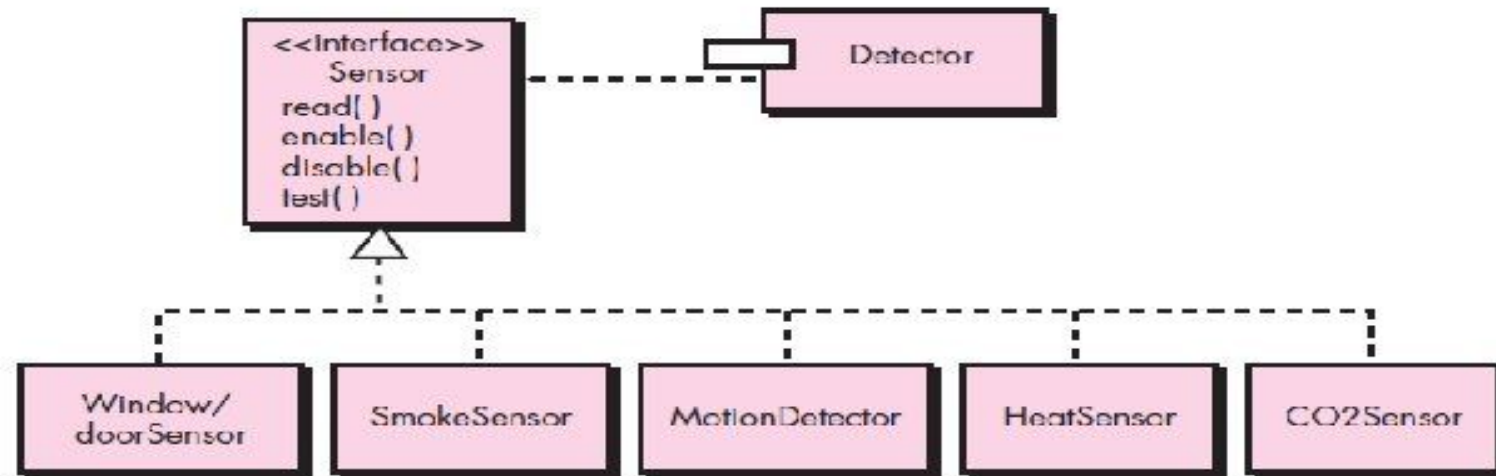
- **Basic Design Principles**
- **Component-Level Design Guidelines**
- **Cohesion**
- **Coupling**



# Designing Class-Based Components

## Basic Design Principles

1. **The Open-Closed Principle (OCP).** “A module [component] should be open for extension but closed for modification.





## 11.2 Designing Class-based Components

- **Basic design principles**

- **The Open-Closed Principle (OCP).** *“A module [component] should be open for extension but closed for modification.”*
- **The Liskov Substitution Principle (LSP).** *“Subclasses should be substitutable for their base classes.”*
- **Dependency Inversion Principle (DIP).** *“Depend on abstractions. Do not depend on concretions.”*
- **The Interface Segregation Principle (ISP).** *“Many client-specific interfaces are better than one general purpose interface.”*
- **The Release Reuse Equivalency Principle (REP).** *“The granule of reuse is the granule of release.”*
- **The Common Closure Principle (CCP).** *“Classes that change together belong together.”*
- **The Common Reuse Principle (CRP).** *“Classes that aren’t reused together should not be grouped together.”*



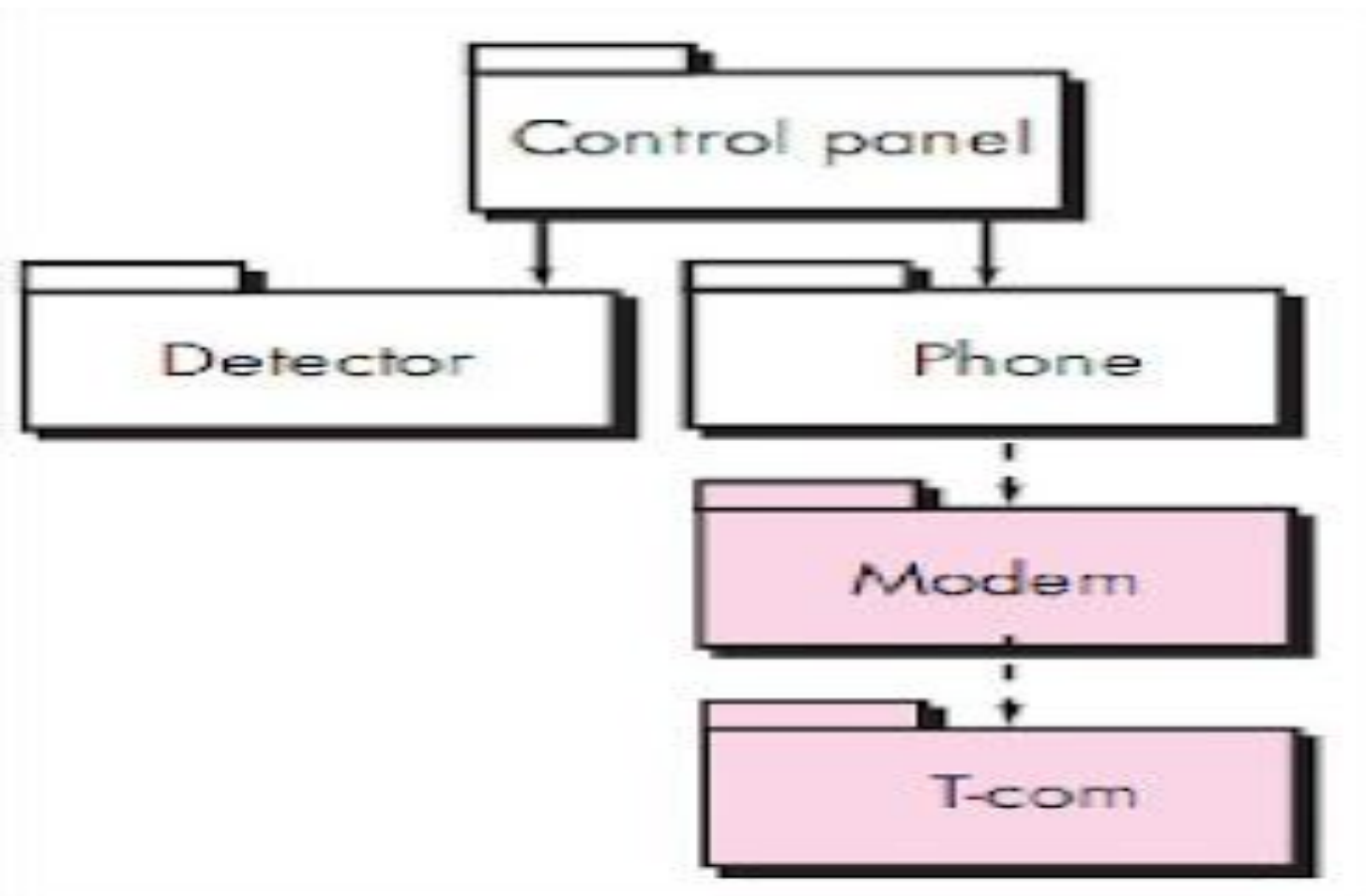
# Design Guidelines

---

- **Components**
  - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
- **Interfaces**
  - Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC)
- **Dependencies and Inheritance**
  - it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

# Cohesion

- Types of cohesion
  - Functional : Exhibited by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result
  - Layer : Exhibited by packages, components, and classes, this type of cohesion occurs when higher layer accesses the services of a lower layer but lower layer do not access higher level



# Contd..

## Communicational

- All operations that access the same data are defined within one class. In general, such classes focus only on the data for accessing and storing it.
- Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.
- It is important to note, however, that pragmatic (Practical) design and implementation issues sometimes force you to opt for lower levels of cohesion

- Sequential- Components or operations are grouped in a manner that allows the first to provide input to the next and so on.
- Procedural-Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked,even when there is no data passed between them.
- Temporal-Operations that are performed to reflect a specific behavior or state.
- Utility-Components,classes or operations that exist within the same category but are otherwise unrelated are grouped together.



# Coupling

Conventional view coupling is :

- The degree to which a component is connected to other components and to the external world.

## **Content Coupling**

- It occurs when one component “secretly modifies data that is internal to another component”
- This violates information hiding—a basic design concept.

## **Common coupling**

- It Occurs when a number of components all make use of a global variable.
- Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application)

## **Control coupling**

- It Occurs when operation A() invokes operation B() and passes a control flag to B. The control flag then “directs” logical flow within B.



## Stamp coupling

- It Occurs when ClassB is declared as a type for an argument of an operation of ClassA. Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex.

## Data coupling

- It Occurs when operations pass long strings of data arguments.
- The “bandwidth” of communication between classes and components grows and the complexity of the interface increases.
- Testing and maintenance are more difficult.

## Routine call coupling

- It Occurs when one operation invokes another.
- This level of coupling is common and is often quite necessary.
- However, it does increase the connectedness of a system.

## Type use coupling

- It Occurs, when component A uses a data type defined in component B (e.g., this occurs whenever “a class declares an instance variable or a local variable as having another class for its type”).
- If the type definition changes, every component that uses the definition must also change.

## Inclusion or import coupling

- It Occurs when component A imports or includes a package or the content of component B.

## External coupling

- It Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions).
- Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

# Component Level Design-I

- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain.
- Step 3. Elaborate all design classes that are not acquired as reusable components.
- Step 3a. Specify message details when classes or component collaborate.
- Step 3b. Identify appropriate interfaces for each component.

# Component-Level Design-II

- Step 3c. Elaborate attributes and define data types and data structures required to implement them.
- Step 3d. Describe processing flow within each operation in detail.
- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop and elaborate behavioral representations for a class or component.
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Factor every component-level design representation and always consider alternatives.

# User Interface Analysis and Design

The overall process for analyzing and designing a UI begins with the creation of models of system.

- User Interface Design Models

Four different models come into play when a user interface is to be analyzed and designed.  
“Prototyping”

1. User model: a profile of all end users of the system

Users can be categorized as:

- Novices: No syntactic and little semantic knowledge of the system.
- Knowledgeable, intermittent users: reasonable knowledge of the system.
- Knowledgeable, frequent users: good syntactic and semantic knowledge of the system.

2. Design model: a design realization of the user model that incorporates data, architectural, interface, and procedural representations of the software.

3. Mental model (system perception): the user’s mental image of what the interface is. The user’s mental model shapes how the user perceives the interface and whether the UI meets the user’s needs.

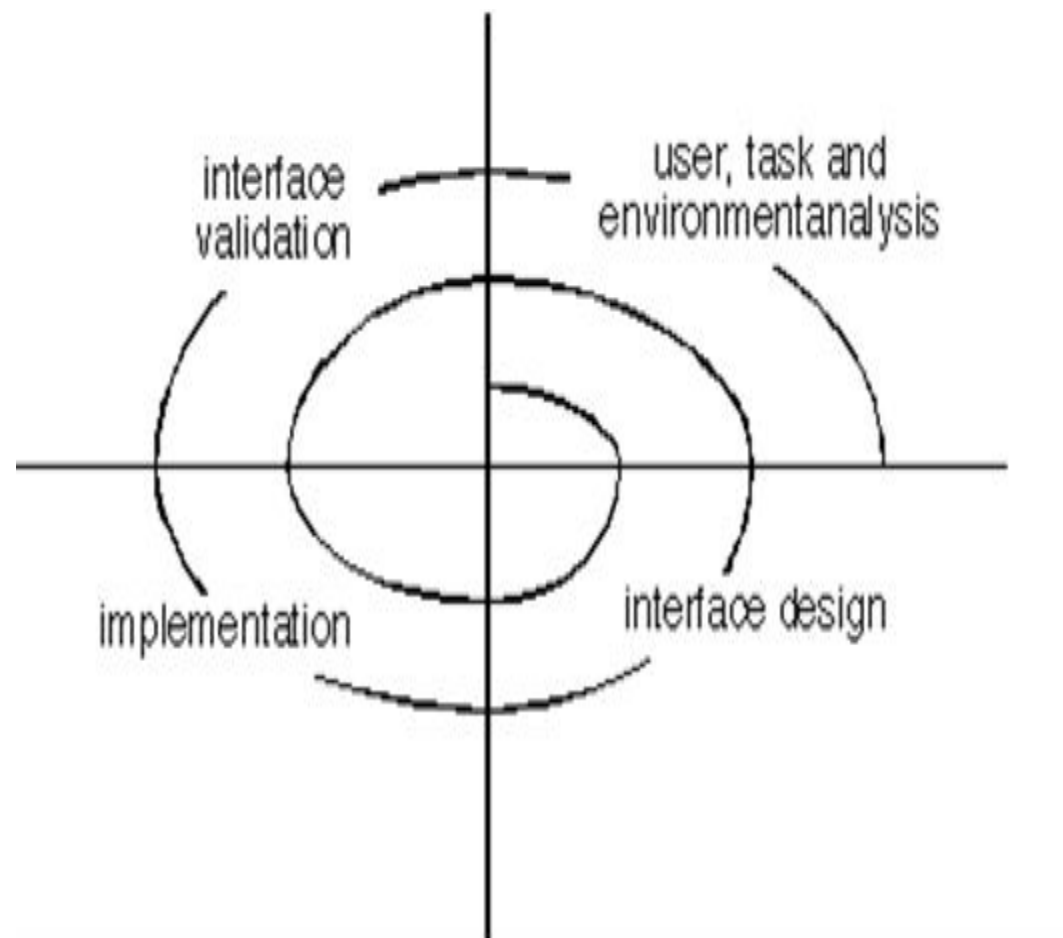
4. Implementation model: the interface “look and feel of the interface” coupled with all supporting information (documentation) that describes interface syntax and semantics.

# The Process

The analysis and design process for UIs is iterative and can be represented using a spiral model.

The user interface analysis and design process encompasses four distinct framework activities:

1. User, task and environment analysis and modeling.
2. Interface design
3. Interface construction (implementation)
4. Interface validation



# User Interface Design Models

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



The analysis of the user environment focuses on the physical work environment

- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environment factors?

The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences.

# Interface Analysis

A key tenet of all software engineering process models is this: you better understand the problem before you attempt to design a solution. Interface design analysis means understanding:

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

## User Analysis

- The only way that a designer can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. This can be accomplished by:
- User Interviews: The software team meets with the end-users to better understand their needs, motivations, work culture, and a myriad of other issues.
- Sales Input: Sales people meet with customers and users to help developers categorize users and better understand their requirements.
- Marketing Input: Market analysis can be invaluable in the definition of market segments while providing an understanding of how each segment might use the software in different ways.
- Support Input: Support staff talks with users on a daily basis, making them the most likely source of information on what works and what doesn't, and what they like and what they don't.

- Are users trained professionals, technician, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users' expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?

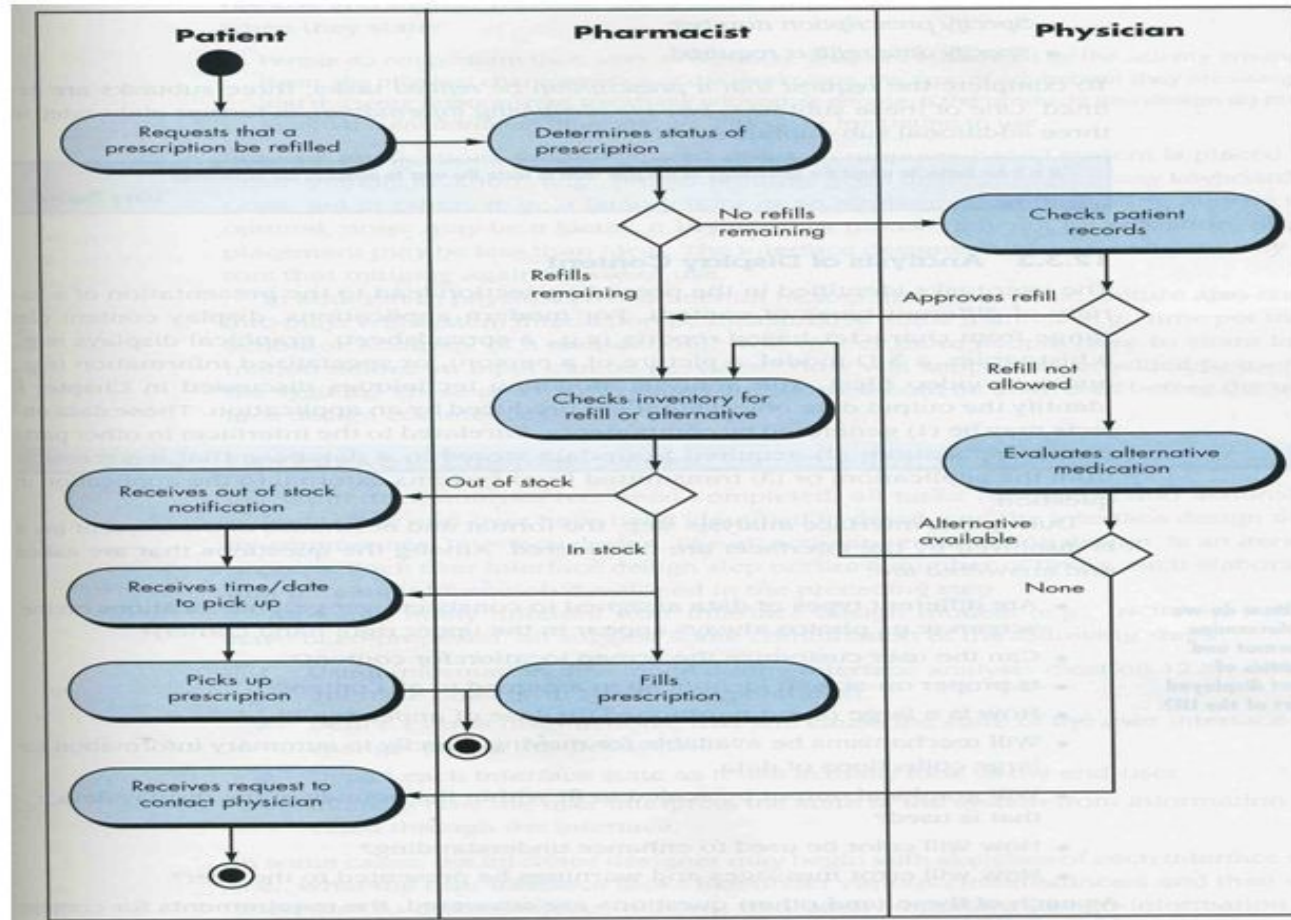
# Task Analysis and Modeling

- Task analysis strives to know and understand
  - The work the user performs in specific circumstances
  - The tasks and subtasks that will be performed as the user does the work
  - The specific problem domain objects that the user manipulates as work is performed
  - The sequence of work tasks (i.e., the workflow)
  - The hierarchy of tasks
- Use cases
  - Show how an end user performs some specific work-related task
  - Enable the software engineer to extract tasks, objects, and overall workflow of the interaction
  - Helps the software engineer to identify additional helpful features

# Elaboration Task

- During elaboration, the software engineer takes the information obtained during inception and elicitation and begins to expand and refine it
- Elaboration focuses on developing a refined technical model of software functions, features, and constraints
- It is an analysis modeling task
  - Use cases are developed
  - Domain classes are identified along with their attributes and relationships
  - State machine diagrams are used to capture the life on an object
- The end result is an analysis model that defines the functional, informational, and behavioral domains of the problem

# Swimlane Diagram



# Workflow Analysis

## Three key interface characteristics

- Each user implements different tasks via the interface; therefore the look and feel of the interface designed for the patient will be different from the one defined for pharmacists or physicians.
- The interface design for pharmacists or physicians must accomodate access to and display of information from secondary information sources
- Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and object elaboration



# Hierarchical Representation

As the interface is analyzed a process of elaboration occurs. Once workflow has been established a task hierarchy can be defined for each user type.

Request that a prescription be refilled

- ✓ Provide identifying information
- ✓ Specify name
- ✓ Specify userid
- ✓ Specify PIN and Password
- ✓ Specify Prescription number
- ✓ Specify date refill is required

# Analysis of Display Content

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data.
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color to be used to enhance understanding?
- How will error messages and warning be presented to the user?

# Interface Design Steps

- Using information developed during interface analysis, **define interface objects and actions** (operations).
- **Define events (user actions)** that will cause the state of the user interface to change. Model this behavior.
- **Depict each interface state** as it will actually look to the end-user.
- **Indicate how the user interprets the state of the system** from information provided through the interface.

# Design Issues

---

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization

# Interface Design Patterns

- Patterns are available for
  - The complete UI
  - Page layout
  - Forms and input
  - Tables
  - Direct data manipulation
  - Navigation
  - Searching
  - Page elements
  - e-Commerce