UNIT -3

DESIGN CONCEPTS

Ms. Varsha
CSE Dept

Design Concepts?

“Idea behind design”

..............????

- Design?
- What is it?
- Who does it?
- Why is it important?
- What is design in software engineering?

Answers !!!

- “A PLAN OR DRAWING”.
- Software Design -- An iterative process transforming requirements into a “blueprint” for constructing the software
- Design allows you to model the system or product that is to be built.
- Software design is the process of implementing software solutions to one or more sets of problems

Translating the analysis model into the design model

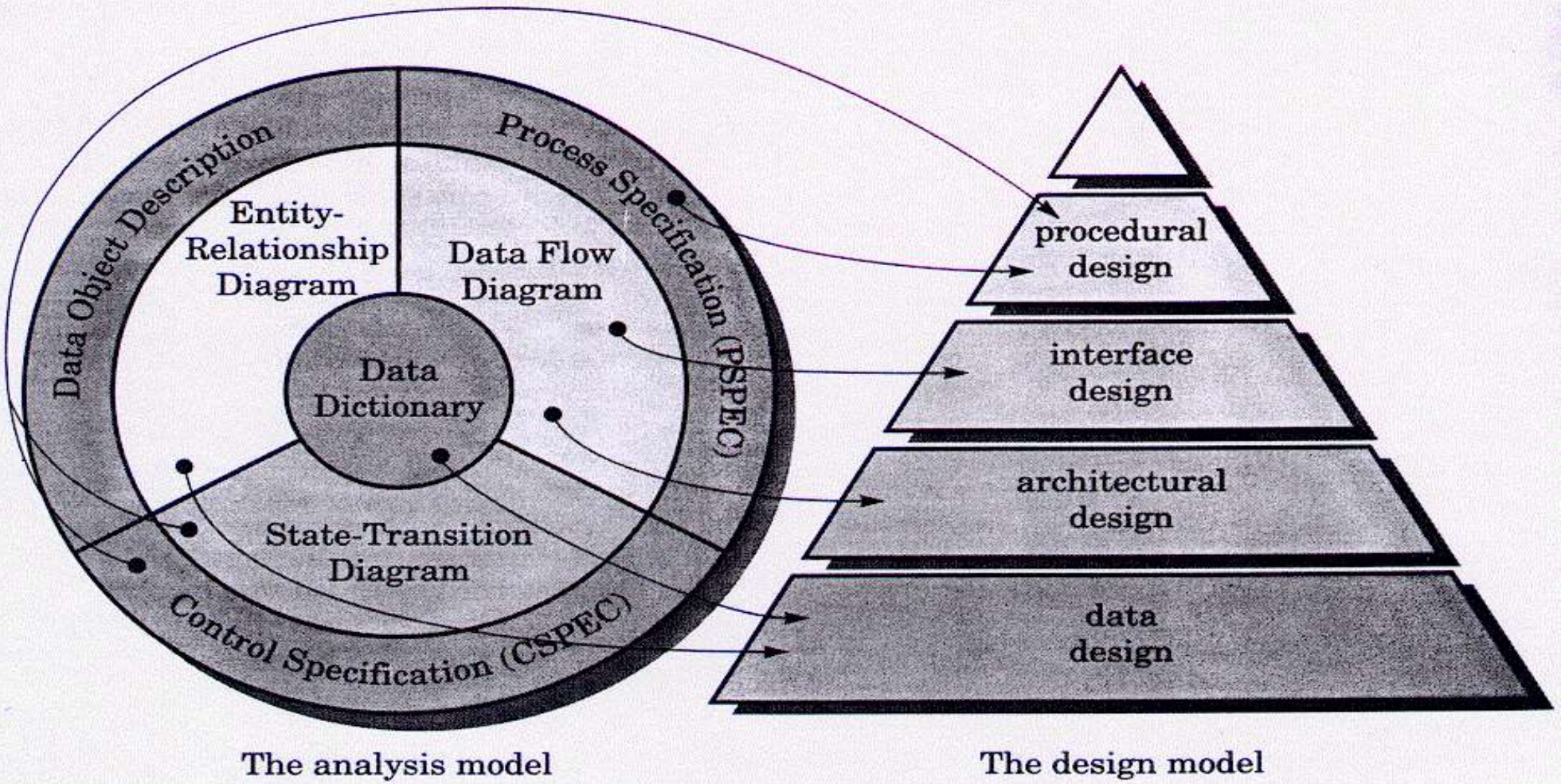


FIGURE 13.1. Translating the analysis model into a software design

Design Process:

- Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software

- During the design process the software requirements model is transformed into design models that describe the details of the data structures, system architecture, interface, and components.

- Design is considered to be high level of abstraction.

Quality Attributes of a good design

Good software design should exhibit:

- **Firmness:** A program should not have any bugs that inhibit its function.
- **Commodity:** A program should be suitable for the purposes for which it was intended
- **Delight:** The experience of using the program should be pleasurable one

Quality Guidelines

- A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
- A design should contain distinct representations of data, architecture, interfaces, and components
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

Fundamental Design concepts

- Abstraction
- Architecture
- Patterns
- Modularity
- Information hiding
- Functional independence
- Refinement
- Refactoring
- Design classes

Abstraction

Permits one to concentrate on a problem at some level of abstraction without regard to low level detail

Abstraction

Abstraction allows designers to focus on solving a problem without being concerned about irrelevant lower level details.

There are two types of abstraction available :

- **Procedural abstraction** – a sequence of instructions that have a specific and limited function. Eg: Word OPEN for a door.
- **Data abstraction** – a named collection of data that describes a data object. Data abstraction for door would be a set of attributes that describes the door (e.g. door type, swing direction, weight, dimension).
- **Control abstraction** – a program control mechanism without specifying internal detail. It is used to coordinate all activities in operating system

Architecture

Overall structure of the system

Architecture

The overall structure of the software and the ways in which the structure provides *conceptual integrity* for a system :

- Consists of components, connectors, and the relationship between them.
- Some of the Architecture models are described below,
- **Structural models** – architecture as organized collection of components
- **Framework models** – attempt to identify repeatable architectural patterns
- **Dynamic models** – indicate how program structure changes as a function of external events
- **Process models** – focus on the design of the business or technical process that system must accommodate
- **Functional models** – used to represent system functional hierarchy

Patterns

General reusable solution which can be used again and again.

Patterns

Software engineer can use the design pattern during the entire software design process.

Each pattern is to provide an insight to a designer who can determine the following-:

- Whether the *pattern can be reused.*
- Whether the pattern is applicable to the current project.
- Whether the pattern can be used to develop a similar but functionally or structurally different design pattern.

Patterns

Types of Design Patterns

Architectural patterns: These patterns are high-level strategies that refer to the overall structure and organization of a software system. That is, they define the elements of a software system such as subsystems, components, classes, etc.

Design patterns: These patterns are medium-level strategies that are used to solve design problems. They provide a means for the refinement of the elements of a software system or the relationship among components or mechanisms that affect component-to-component.

Idioms: These patterns are low-level patterns, which are programming-language specific. They describe the implementation of a software component, the method used for interaction among software components.

Modularity

What is the "right" number of modules for a specific software design…???

Single attribute of software that allows a program to be intellectually manageable.

Modularity

- Software is divided into separately named and addressable components, often called modules, that are integrated to satisfy problem requirements.
- This leads to a "*divide and conquer*" conclusion—it's easier to solve a complex problem when you break it into manageable pieces.
- It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable“.

Modularity

- **Modular decomposability:**
  A design method provides a systematic mechanism for decomposing the problem into sub-problems
  -->reduce the complexity and achieve the modularity

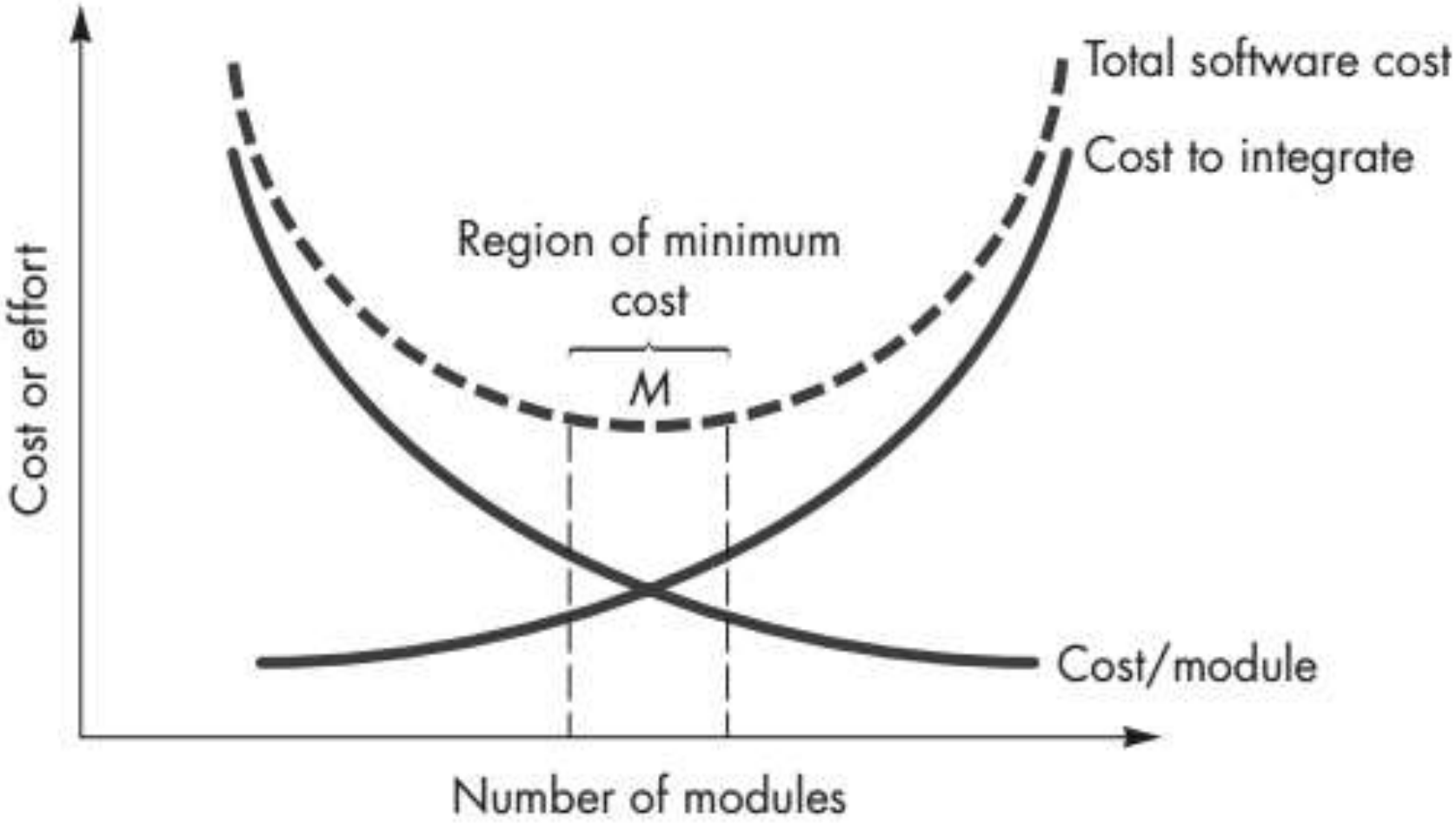
- **Modular composability:**
  A design method enables existing design components to be assembled into a new system.

- **Modular understandability:**
  A module can be understood as a standalone unit it will be easier to build and easier to change.

- **Modular continuity:**
  A small changes to the system requirements result in changes to individual modules, rather than system-wide changes.

- **Modular protection:**
  An aberrant condition occurs within a module and its effects are constrained within the module.

Modularity and software cost



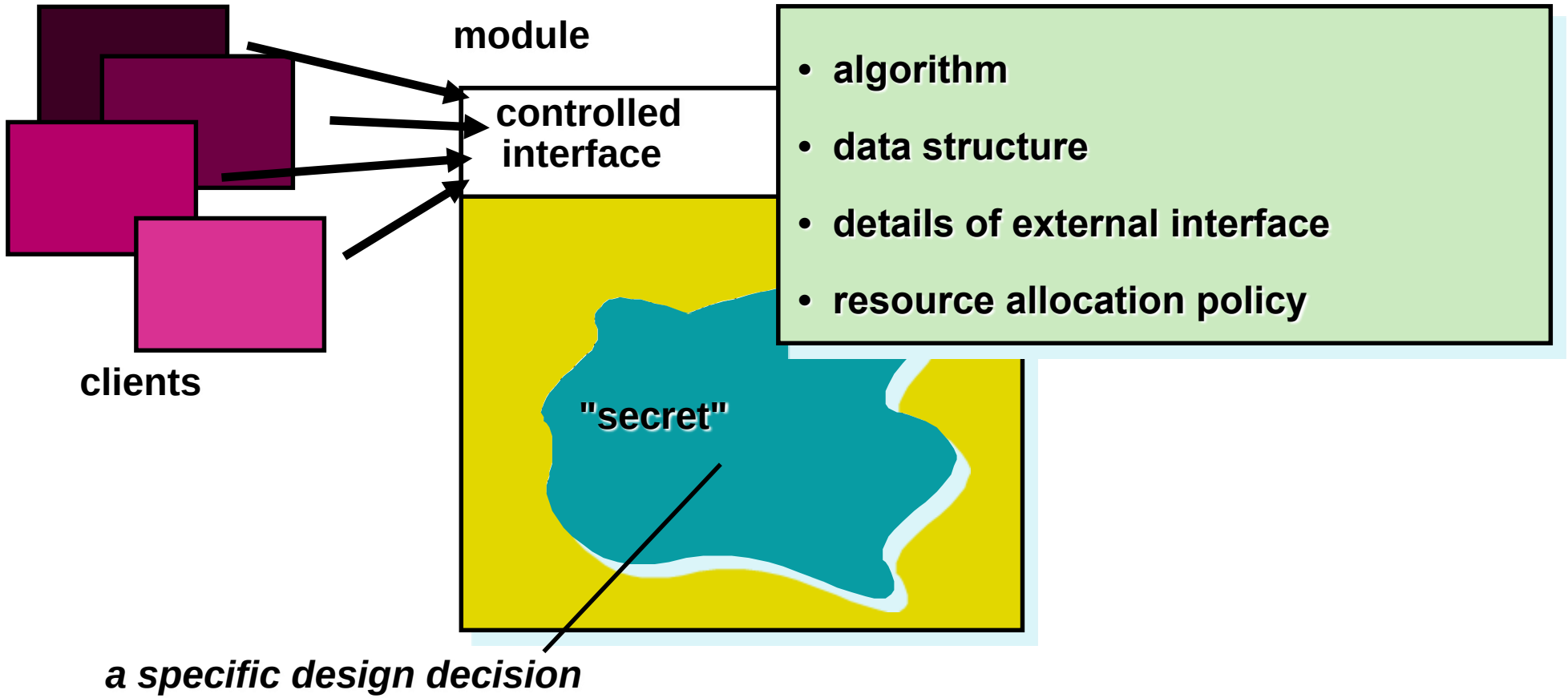
Information Hiding

The way of hiding the unnecessary details

Information Hiding

- Information (data and procedure) contained within a module should be 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
- Modules should be specified and designed so that information (procedure and data) contained within a module is *inaccessible* to other modules that have no need for such information.
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.
- This enforces *access constraints* to both procedural (i.e., implementation) detail and local data structures.

Information Hiding



Functional Independence

Achieved by developing modules with "single minded" function and low coupling.

Functional Independence

Functional independence is achieved by developing a module to perform given set of functions without interacting with other parts of the system.

It can be measured using two criteria:

-Cohesion

-Coupling

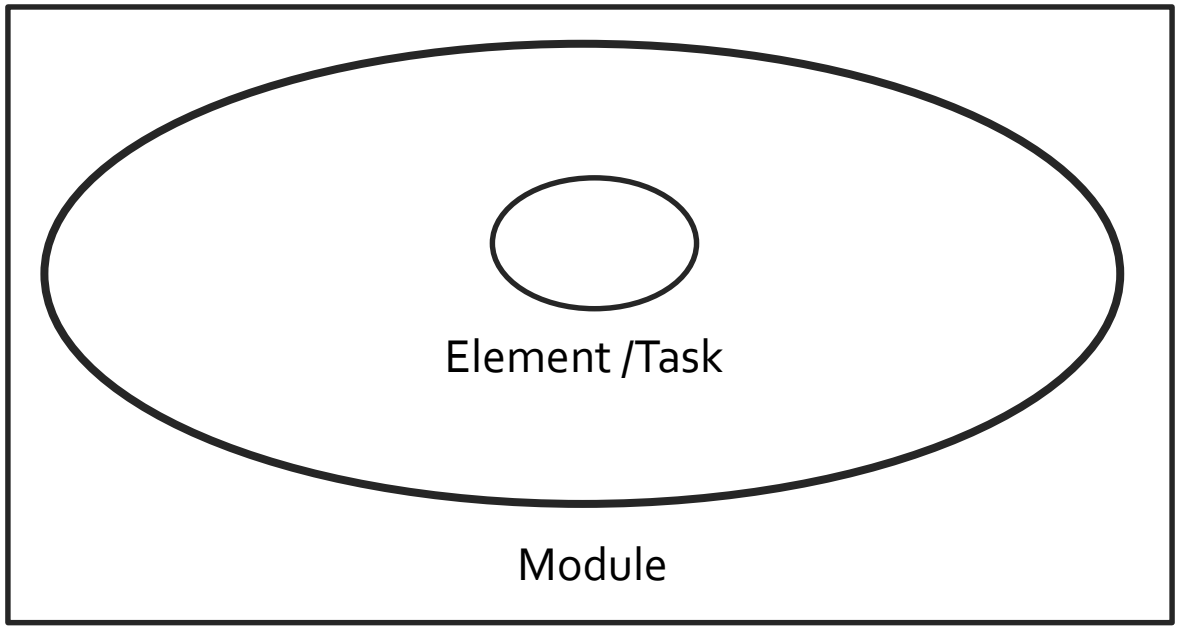
Design modules based on independent functional features

Functional Independence

Cohesion: A natural extension of the information hiding concept a module may perform a number of tasks. A cohesive module performs a single task in a procedure with little interactions with others.

Types of cohesion:

- ✓ Functional cohesion
- ✓ Sequential cohesion
- ✓ Communication cohesion
- ✓ Procedural cohesion
- ✓ Temporal cohesion
- ✓ Logical cohesion
- ✓ Coincidental cohesion

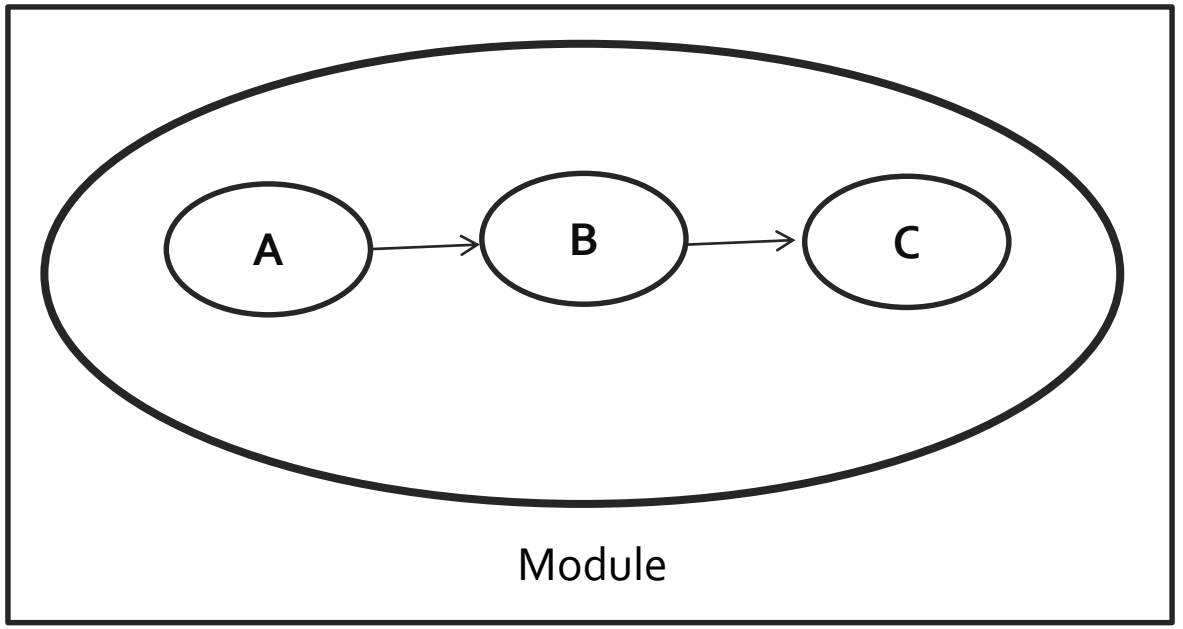


Functional Independence

Cohesion: A natural extension of the information hiding concept a module may perform a number of tasks. A cohesive module performs a single task in a procedure with little interactions with others.

Types of cohesion:

- ✓ Functional cohesion
- ✓ Sequential cohesion
- ✓ Communication cohesion
- ✓ Procedural cohesion
- ✓ Temporal cohesion
- ✓ Logical cohesion
- ✓ Coincidental cohesion

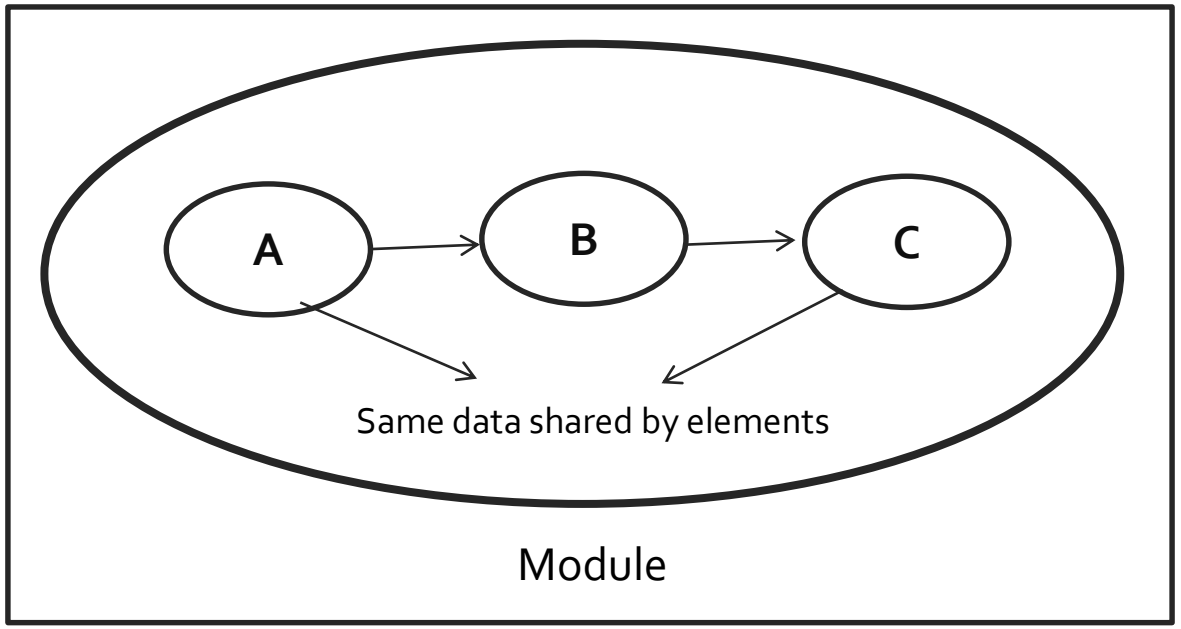


Functional Independence

Cohesion: A natural extension of the information hiding concept a module may perform a number of tasks. A cohesive module performs a single task in a procedure with little interactions with others.

Types of cohesion:

- ✓ Functional cohesion
- ✓ Sequential cohesion
- ➡ ✓ Communication cohesion
- ✓ Procedural cohesion
- ✓ Temporal cohesion
- ✓ Logical cohesion
- ✓ Coincidental cohesion

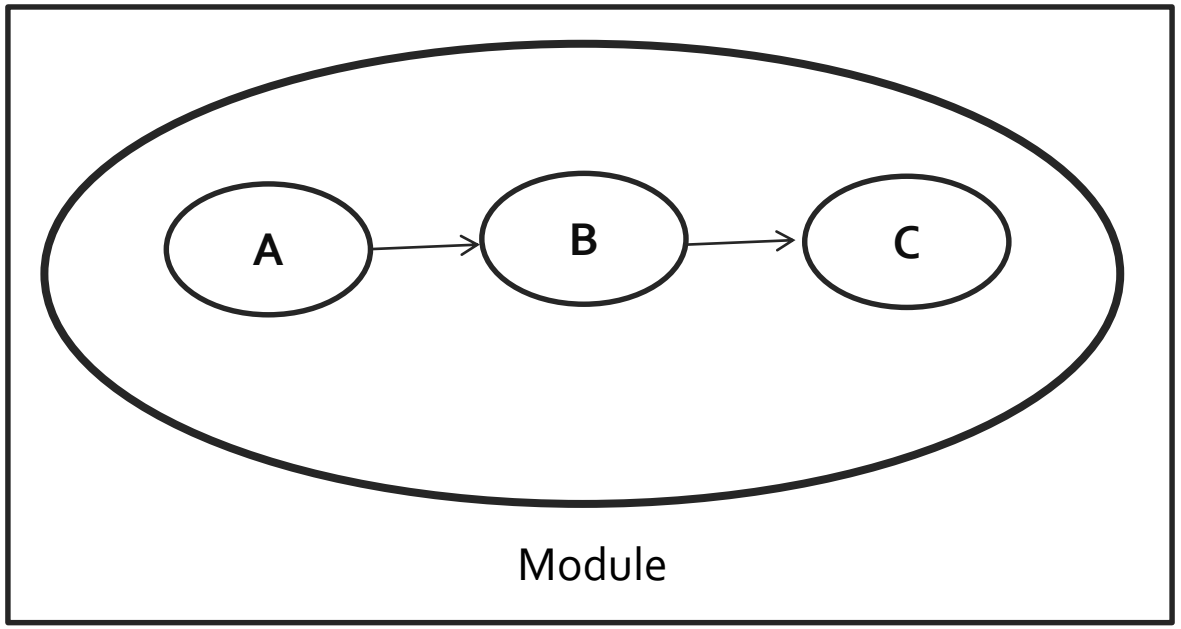


Functional Independence

Cohesion: A natural extension of the information hiding concept a module may perform a number of tasks. A cohesive module performs a single task in a procedure with little interactions with others.

Types of cohesion:

- ✓ Functional cohesion
- ✓ Sequential cohesion
- ✓ Communication cohesion
- ➡ ✓ Procedural cohesion
- ✓ Temporal cohesion
- ✓ Logical cohesion
- ✓ Coincidental cohesion

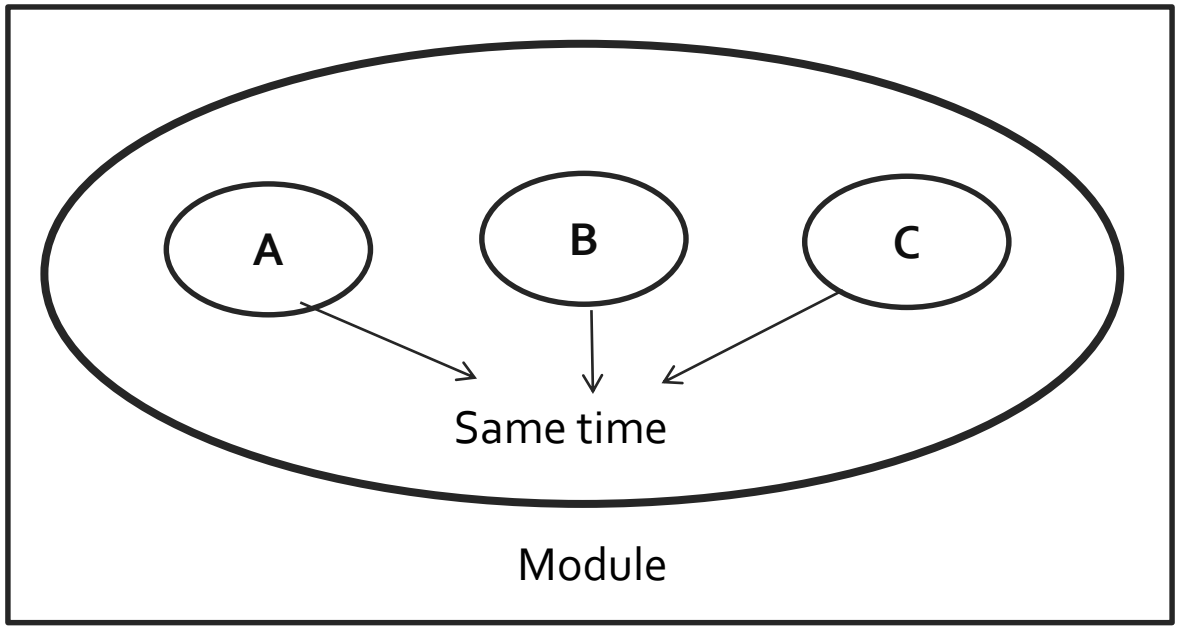


Functional Independence

Cohesion: A natural extension of the information hiding concept a module may perform a number of tasks. A cohesive module performs a single task in a procedure with little interactions with others.

Types of cohesion:

- ✓ Functional cohesion
- ✓ Sequential cohesion
- ✓ Communication cohesion
- ✓ Procedural cohesion
- ➡ ✓ Temporal cohesion
- ✓ Logical cohesion
- ✓ Coincidental cohesion

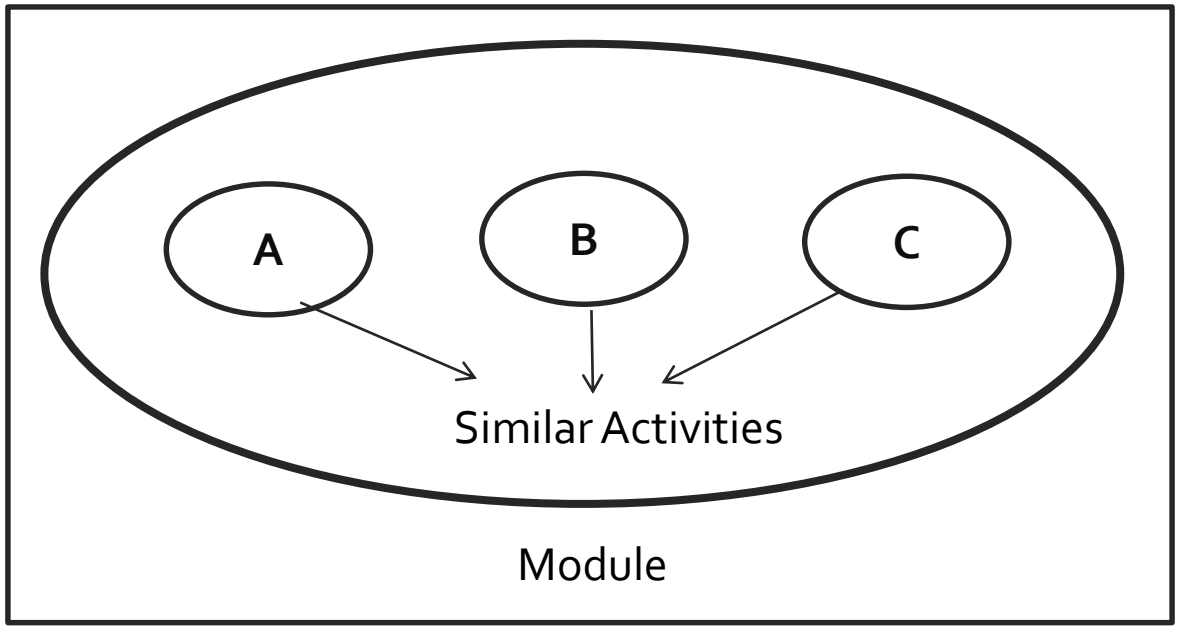


Functional Independence

Cohesion: A natural extension of the information hiding concept a module may perform a number of tasks. A cohesive module performs a single task in a procedure with little interactions with others.

Types of cohesion:

- ✓ Functional cohesion
- ✓ Sequential cohesion
- ✓ Communication cohesion
- ✓ Procedural cohesion
- ✓ Temporal cohesion
- ➡ ✓ Logical cohesion
- ✓ Coincidental cohesion

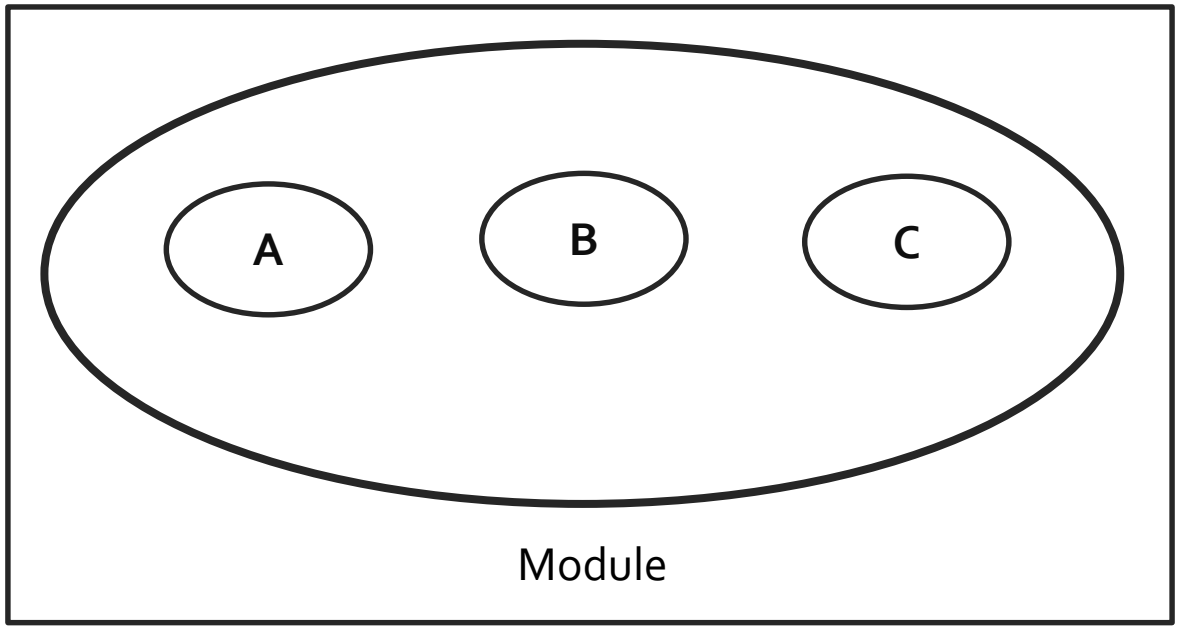


Functional Independence

Cohesion: A natural extension of the information hiding concept a module may perform a number of tasks. A cohesive module performs a single task in a procedure with little interactions with others.

Types of cohesion:

- ✓ **Functional cohesion**
- ✓ **Sequential cohesion**
- ✓ **Communication cohesion**
- ✓ **Procedural cohesion**
- ✓ **Temporal cohesion**
- ✓ **Logical cohesion**
- ➡ ✓ **Coincidental cohesion**



Functional Independence

Coupling: A measure of interconnection/interdependence among modules in a program structure. Coupling depends on the interface complexity between modules.

Types of coupling:

- **No direct coupling** : independent of each other.
- **Data coupling** : passing parameter or data interaction
- **Stamp coupling** : data structure id passed via argument list
- **Control coupling** : share related control logical (control flag)
- **Common coupling** : sharing common data areas.
- **content coupling** : module A use of data or control information maintained in another module.

Refinement

Stepwise refinement is a top-down design strategy

Refinement

- Refinement is actually a process of *elaboration.*
- It begin with a statement of function (or description of information) that is defined at a high level of abstraction and reach at the lower level of abstraction.
- Terminates when all instructions are expressed in terms of any underlying computer or programming language.
- Abstraction and refinement are complementary concepts

Refactoring

restructuring (rearranging) code…
“changing the structure of a program so that the functionality is not changed”

Refactoring:

- Refactoring is a reorganization technique that simplifies the design (or internal code structure) of a component without changing its function or external behaviour.
- It removes redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failures.

Object Oriented Design Concepts:

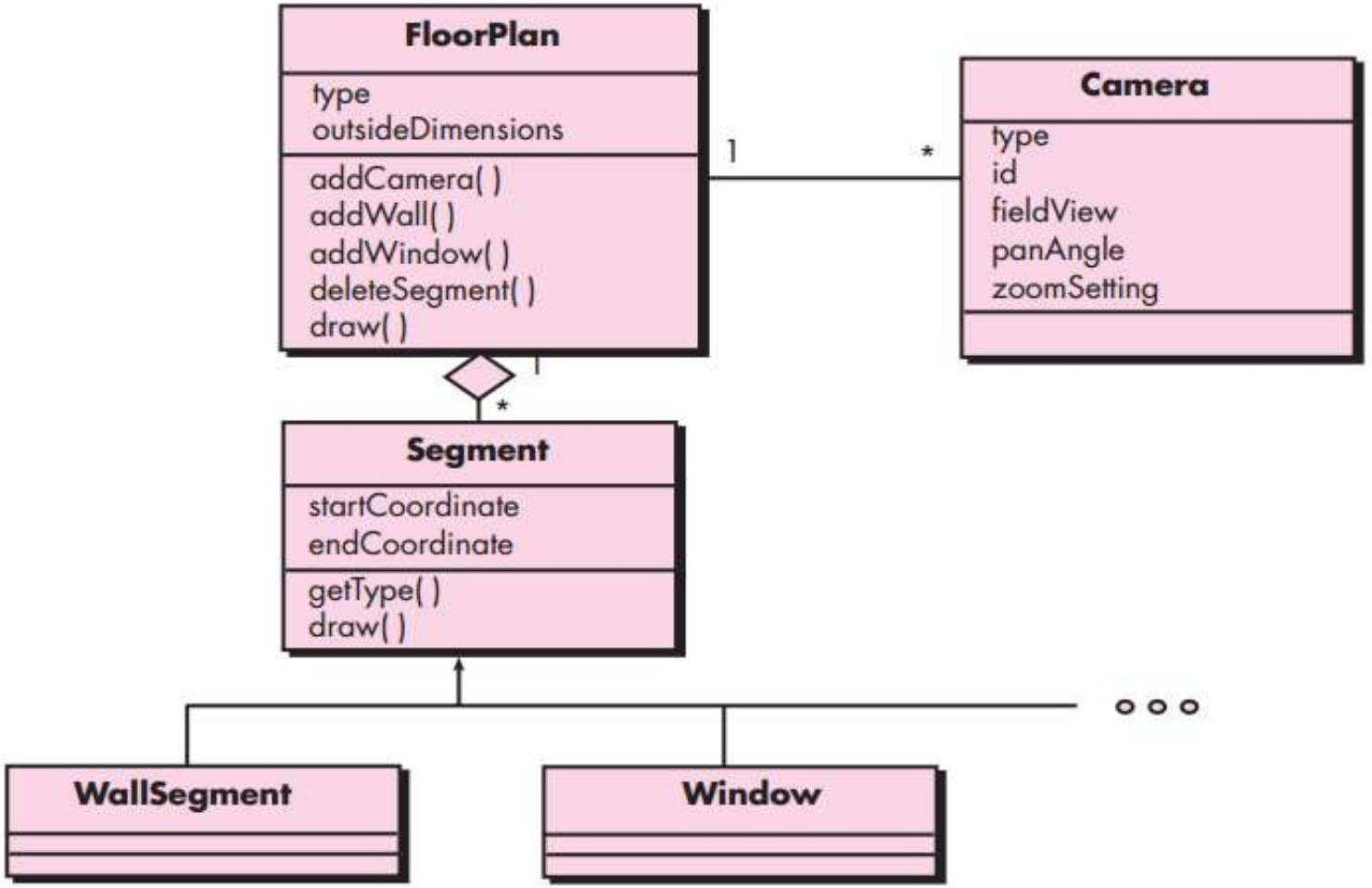
"Design system using self-contained objects and object classes"

- Objects are abstractions of real world or system entities and manage themselves
- Objects are independent and encapsulate state and represent information
- System functionality is expressed in terms of object services
- Shared data are eliminated. Objects communicate by message passing

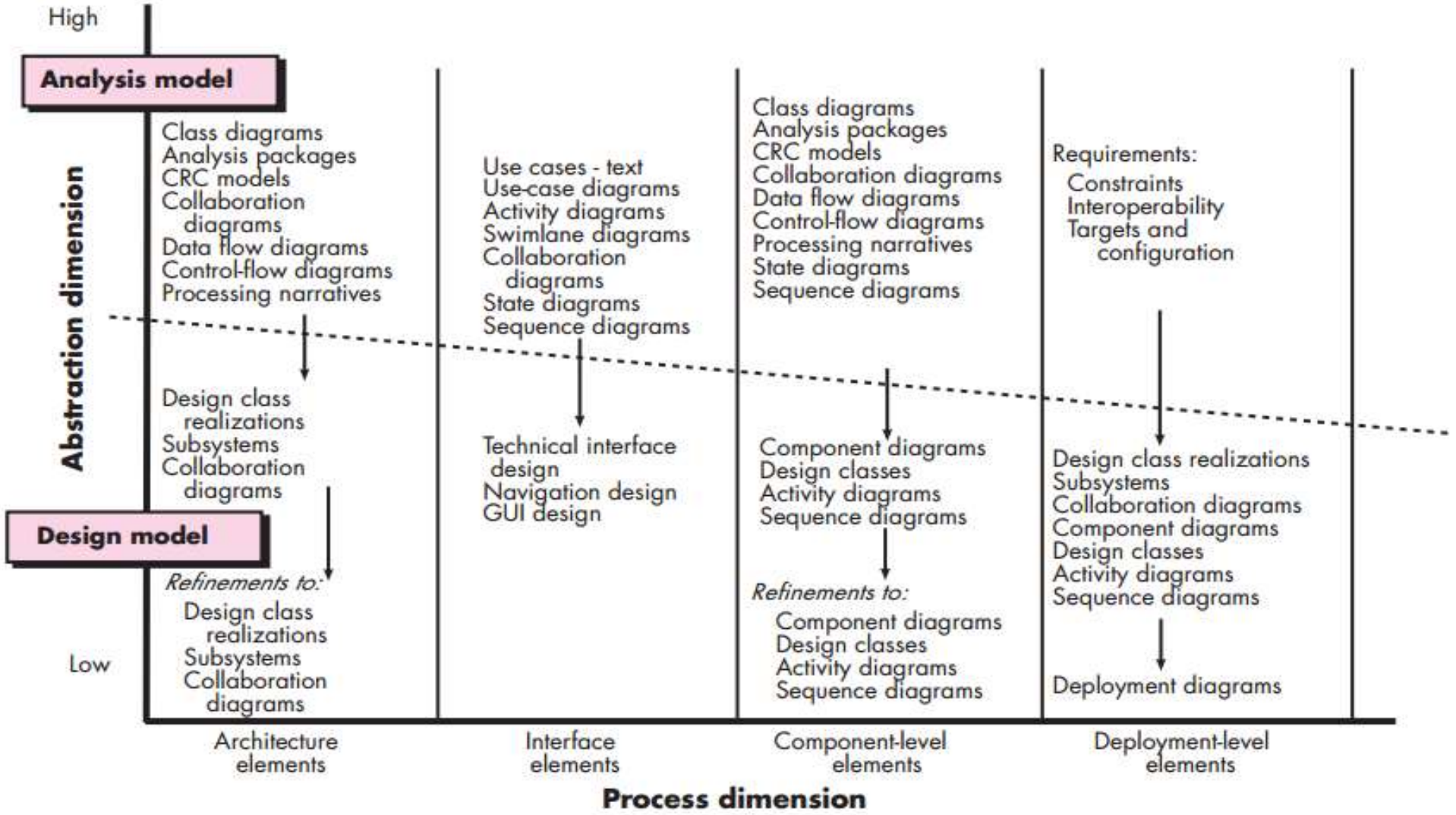
Design Classes

- The requirements model defines a set of analysis classes
- Set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
- Characteristics of Design Classes

Design class for FloorPlan and composite aggregation for the class



Dimensions of the design model



Design Model Elements

- Data Design Elements
- Architectural Design Elements
- Interface Design Elements
- Component-Level Design Elements
- Deployment-Level Design Elements

1.Data Design Elements

- Data-model → Data structures
- Data-model → Database architecture

2.Architectural Design Elements

The architectural model [sha96] is derived from three sources:

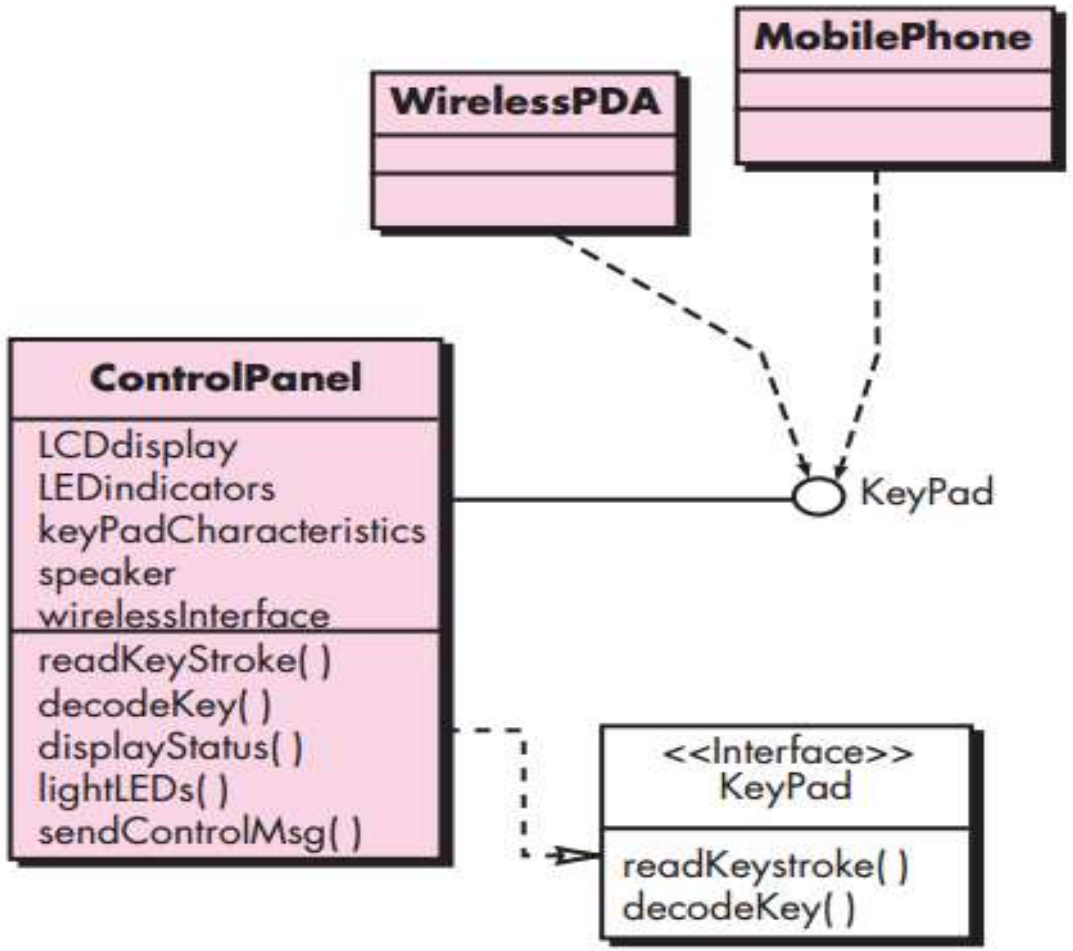
- Information about the application domain for the software to be built
- Specific requirements model elements such as data flow diagrams of analysis classes, their relationship and collaborations for the problems at hand
- The availability of architectural patterns and styles

3. Interface Design Elements

There are three important elements of interface design:

- User interface (UI)
- External interfaces to other systems, devices, networks, or other producers or consumers of information
- Internal interfaces between various design components

Interface representation for Control Panel



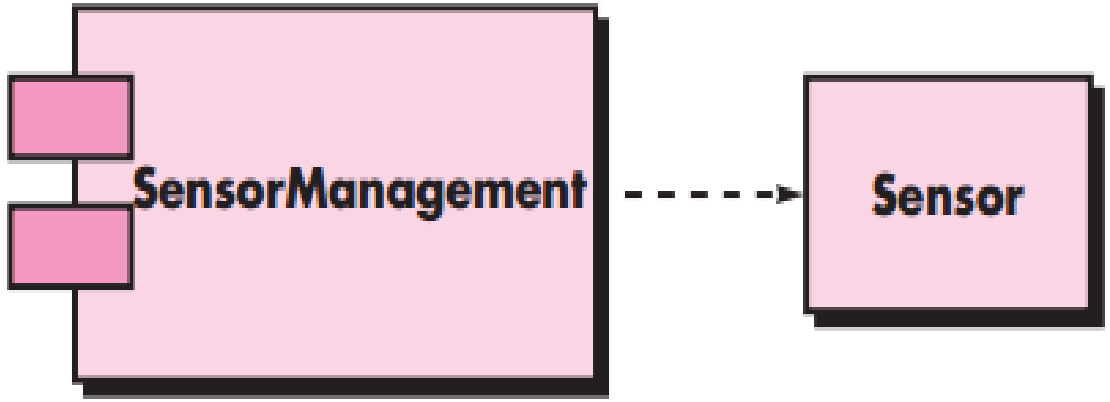
4.Component-level Design Elements

- The component-level design for software fully describes the internal detail of each software component
- The component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations

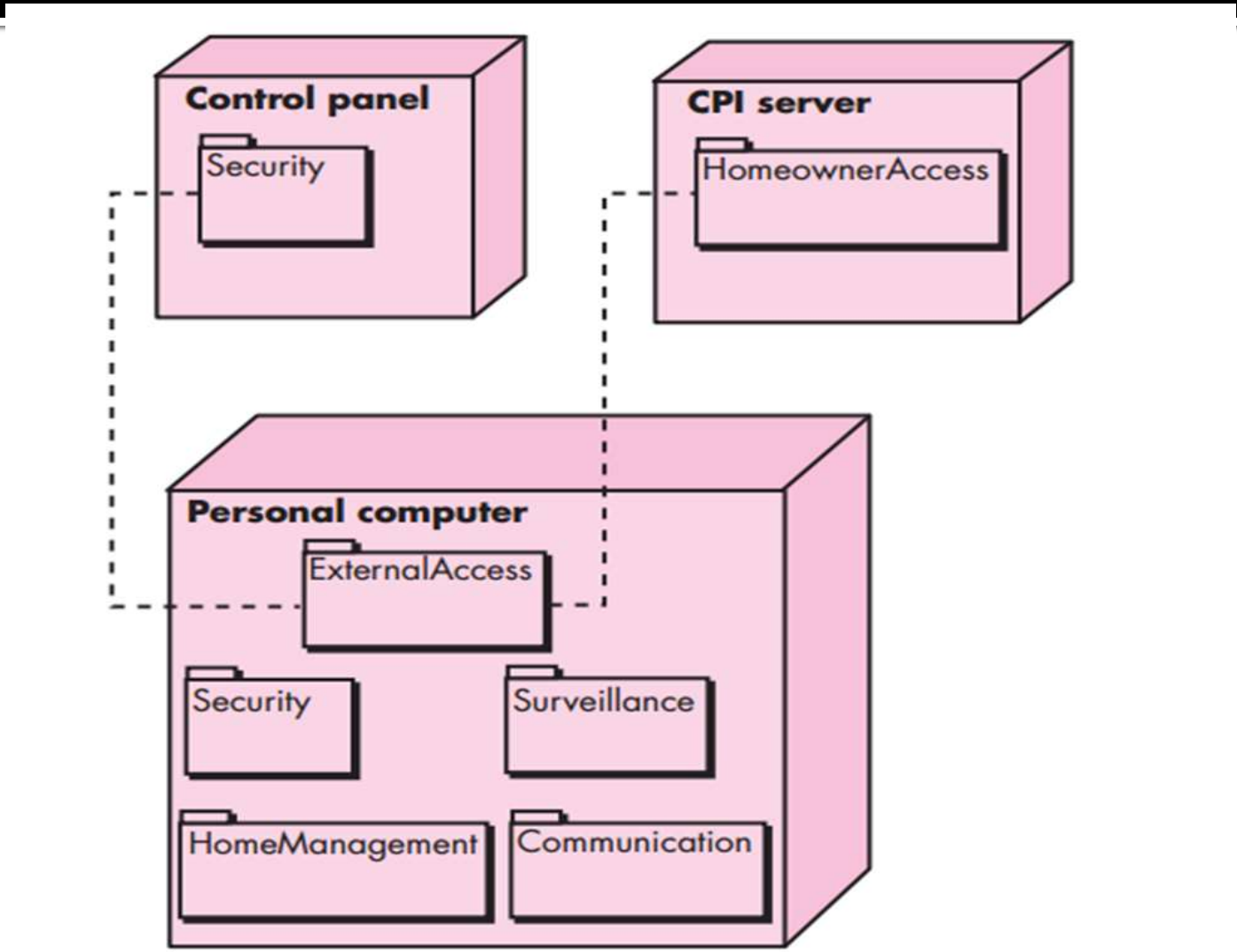
5.Deployment-level Design Elements

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

A UML component diagram



A UML deployment diagram



References:

1. "Software Engineering: A Practitioner's Approach" 5th Ed. by Roger S. Pressman, Mc-Graw-Hill, 2001
2. "Software Engineering" by Ian Sommerville, Addison-Wesley
3. "Software Engineering" 3rd edition by K.K.Aggrawal and Yogesh Singh, New age international publishers.

Thank You