Software Engineering

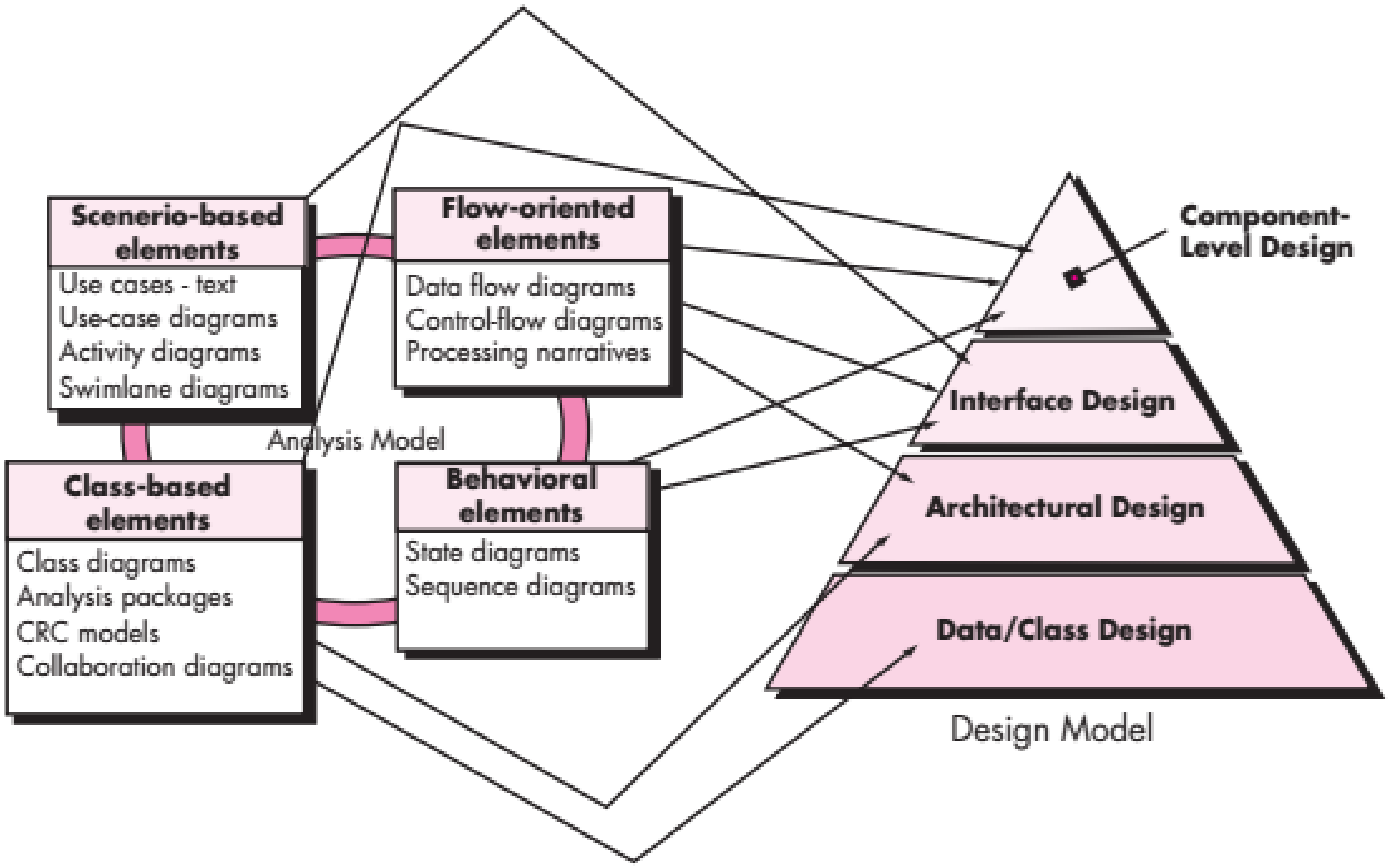
Presenting By:
B.Pranalini

Unit 3 Part 1: Design Engineering

Design:

- Mitch Kapor, the creator of Lotus 1-2-3, presented a "software design manifesto" in *Dr. Dobbs Journal*. He said:
  - 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.

REQUIREMENTS MODEL -> DESIGN MODEL



- The requirements model feed the design task.
- The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software.
- The architectural design representation—the framework of a computer-based system—is derived from the requirements model.
- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it.
- The component-level design transforms structural elements of the software architecture into a procedural description of software components.
- The importance of software design can be stated with a single word—*quality*

Design Process

Software Quality Guidelines and Attributes:

- the design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

- the design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

- the design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines

- A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion
  - For smaller systems, design can sometimes be developed linearly.
- 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.

Quality Attributes

- *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- *Usability* is assessed by considering human factors , overall aesthetics, consistency, and documentation.
- *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- *Performance* is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term,
- *maintainability*—and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration), the ease with which a system can be installed, and the ease with which problems can be localized.

The Evolution of Software Design

- The evolution of software design is a continuing process that has now spanned almost six decades.
- Procedural aspects of design definition evolved into a philosophy called *structured programming*
- Design work proposed methods for the translation of data into a design definition.
- Design methods: Analysis method, Software design method
- These methods have a number of common characteristics:

  (1) a mechanism for the translation of the requirements model into a design representation,

  (2) a notation for representing functional components and their interfaces,

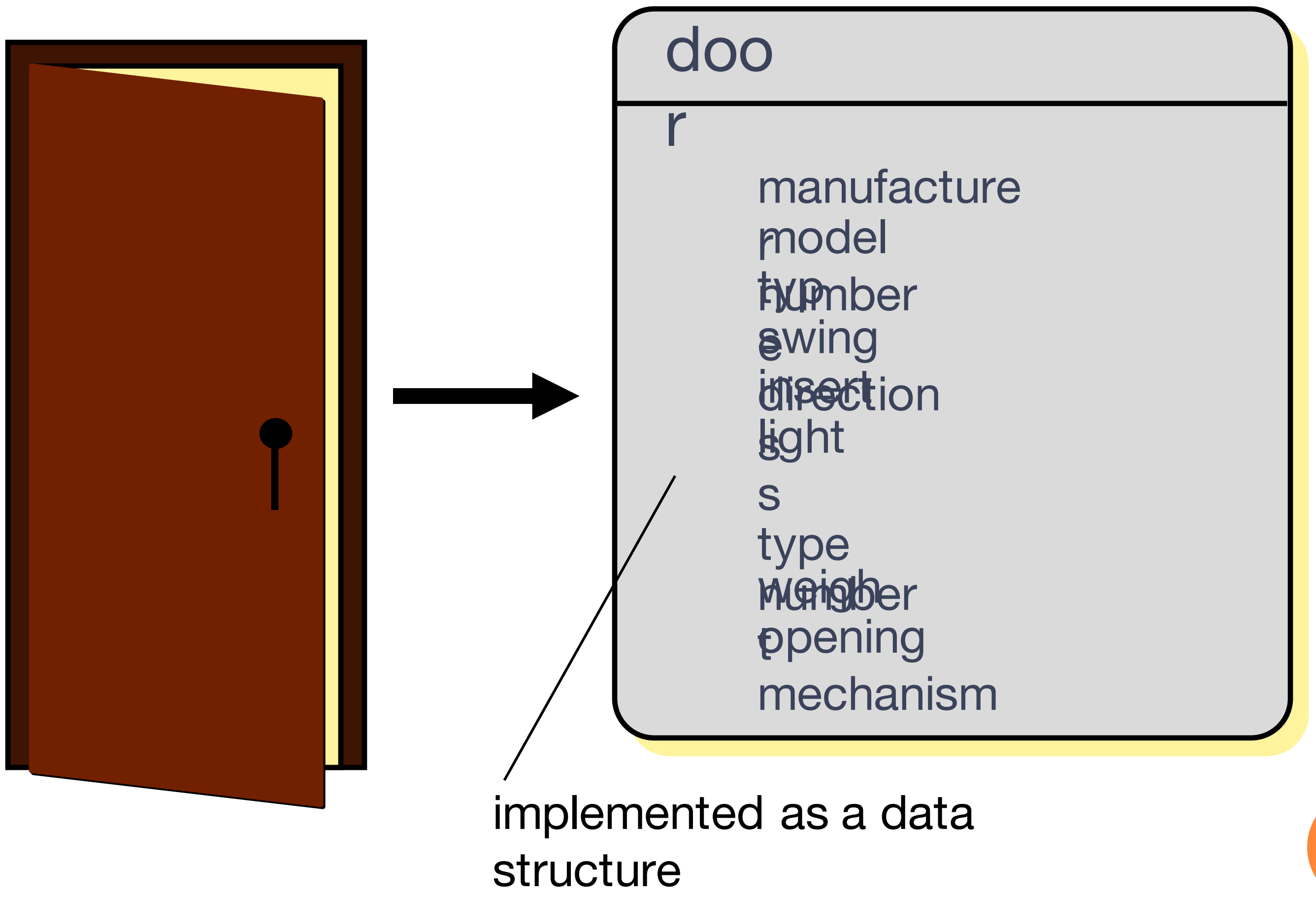
  (3) heuristics for refinement and partitioning, and

  (4) guidelines for quality assessment

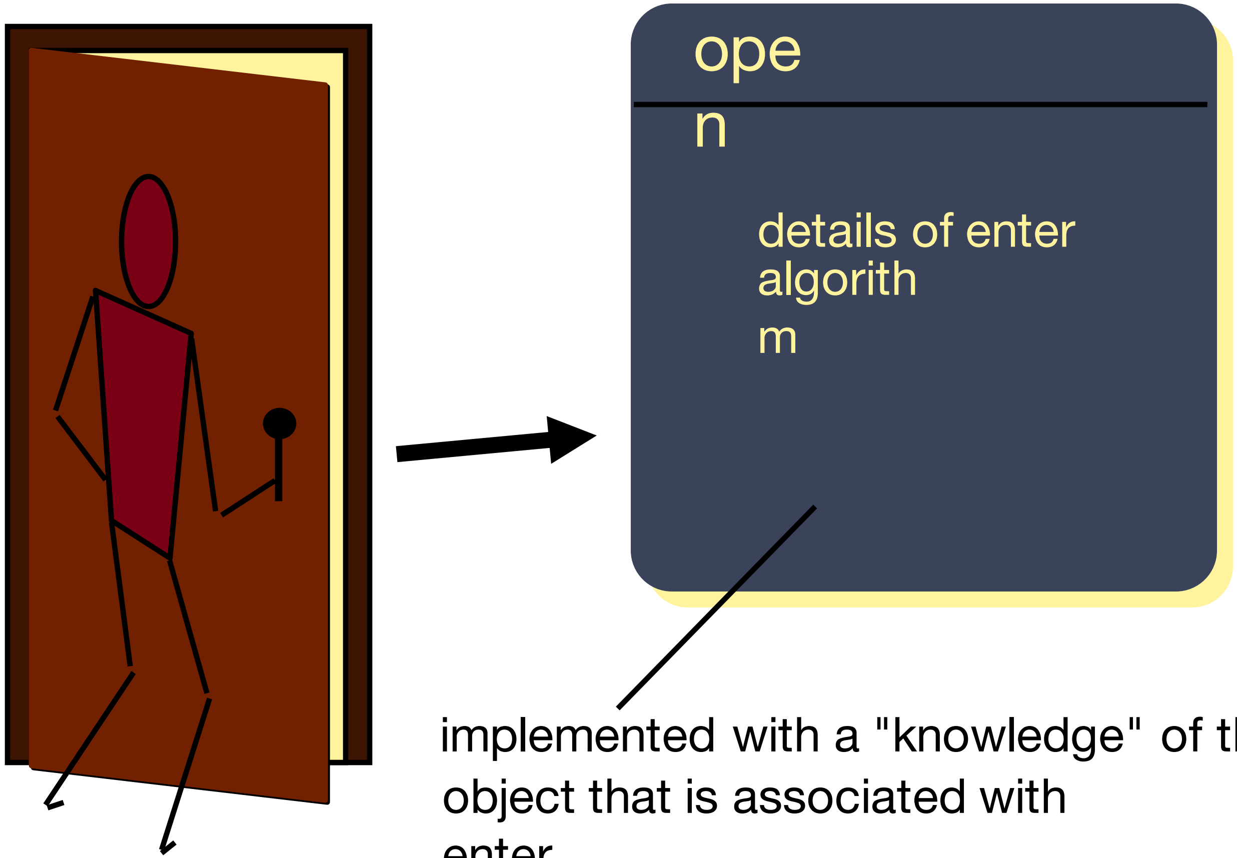
DESIGN CONCEPTS

- Abstraction—data, procedure
- Architecture—the overall structure of the software
- Patterns—”conveys the essence” of a proven design solution
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Aspects—a mechanism for understanding how global requirements affect design
- Refactoring—a reorganization technique that simplifies the design
- OO design concepts—Appendix II
- Design Classes—provide design detail that will enable analysis classes to be implemented

Data Abstraction



Procedural Abstraction



implemented with a "knowledge" of the object that is associated with enter

Architecture

“The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.” [SHA95a]

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Patterns

Design Pattern Template

Pattern name**—describes the essence of the pattern in a short but expressive name**

Intent**—describes the pattern and what it does**

Also-known-as**—lists any synonyms for the pattern**

Motivation**—provides an example of the problem**

Applicability**—notes specific design situations in which the pattern is applicable**

Structure**—describes the classes that are required to implement the pattern**

Participants**—describes the responsibilities of the classes that are required to implement the pattern**

Collaborations**—describes how the participants collaborate to carry out their responsibilities**

Consequences**—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented**

Related patterns**—cross-references related design patterns**

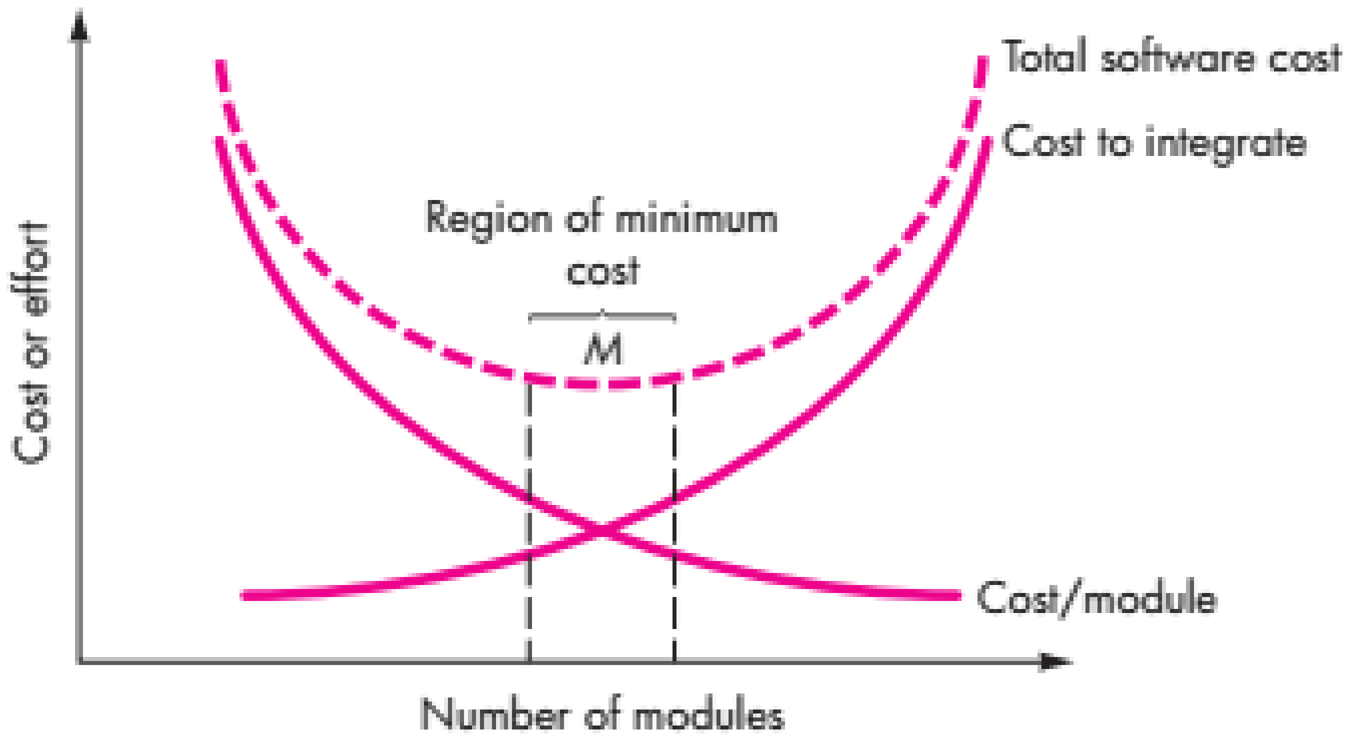
Separation of Concerns

- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

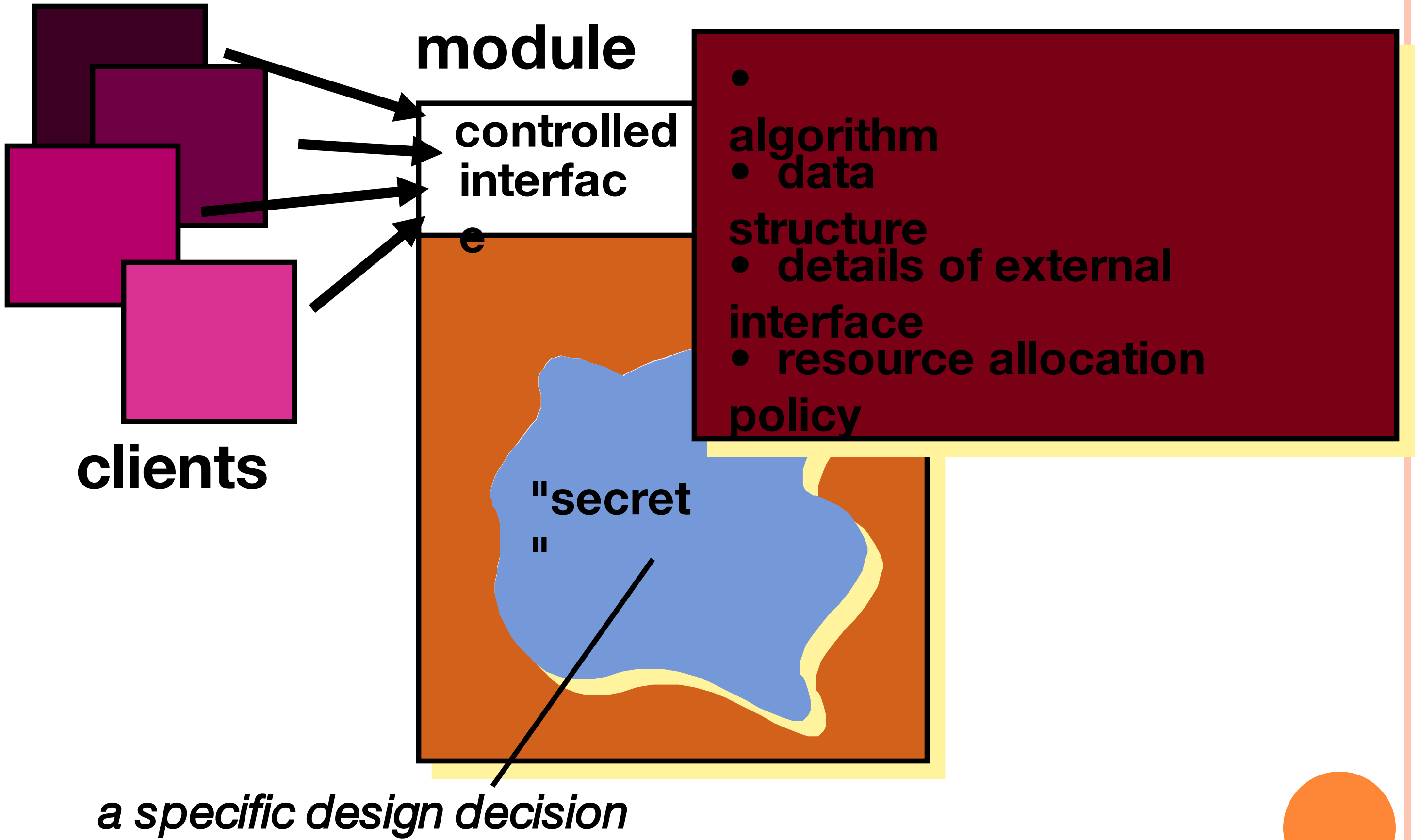
Modularity

- "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

Modularity and software cost



Information Hiding


module
controlled interfac e
clients
"secret "
a specific design decision
• algorithm
• data structure
• details of external interface
• resource allocation policy

Why Information Hiding?

- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

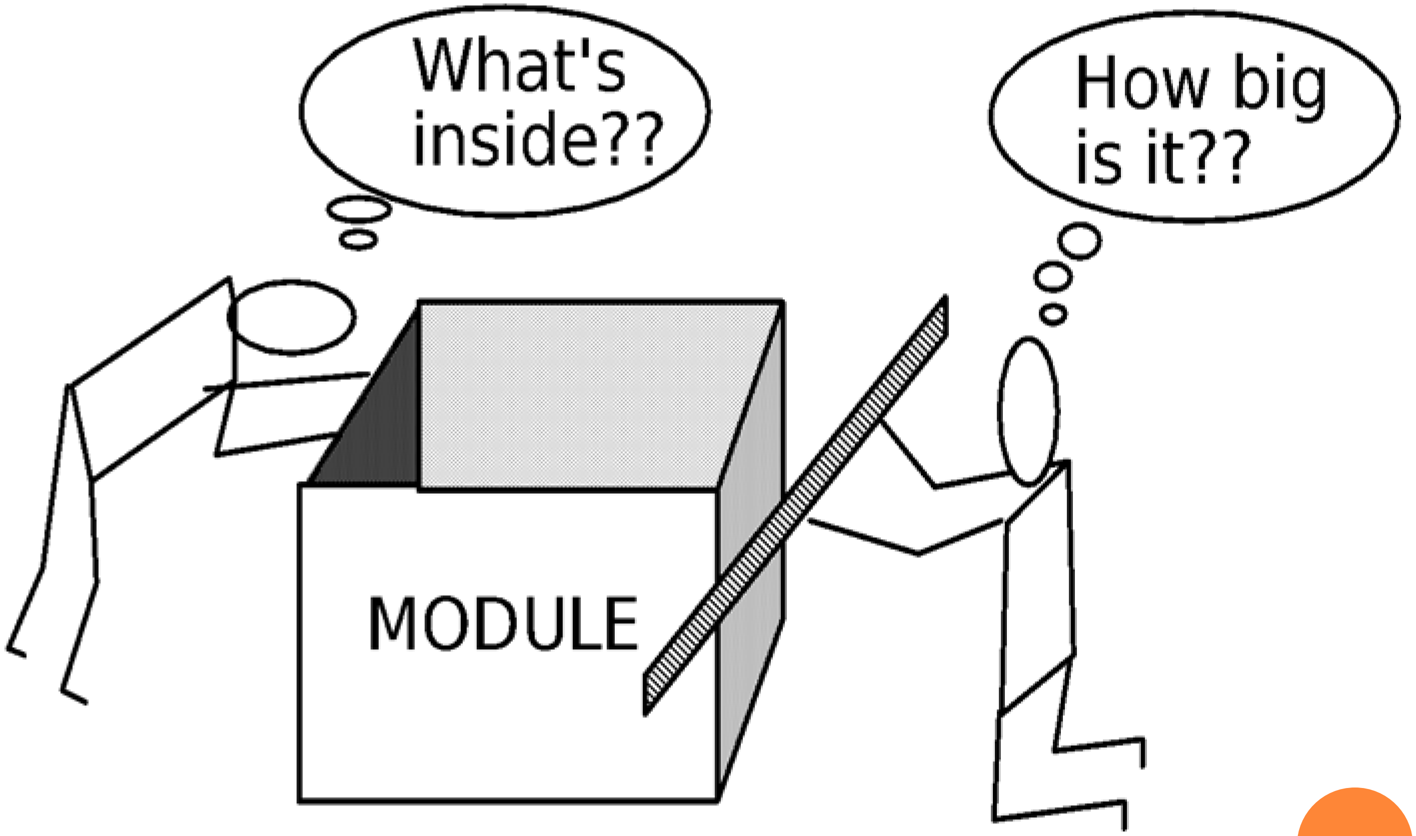
Stepwise Refinement

open

```
walk to door;
reach for knob;
open door;
walk through;
close door.
```

```
repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
    take key out;
    find correct key;
    insert in lock;
end if;
pull/push door
move out of way;
end repeat
```

Sizing Modules: Two Views



FUNCTIONAL INDEPENDENCE

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- *Cohesion* is an indication of the relative functional strength of a module.
  - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- *Coupling* is an indication of the relative interdependence among modules.
  - 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.

Aspects

- Consider two requirements, *A* and *B*. Requirement *A* *crosscuts* requirement *B* “if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account. [Ros04]
- An *aspect* is a representation of a cross-cutting concern.

Aspects—An Example

- Consider two requirements for the **SafeHomeAssured.com** WebApp. Requirement *A* is described via the use-case **Access camera surveillance via the Internet.** A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using* **SafeHomeAssured.com.** This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B*. Therefore, *A** and *B** are representations of concerns, and B* *cross-cuts* A*.
- An *aspect* is a representation of a cross-cutting concern. Therefore, the design representation, *B**, of the requirement, *a registered user must be validated prior to using* **SafeHomeAssured.com,** is an aspect of the *SafeHome* WebApp.

REFACTORING

- Fowler [FOW99] defines refactoring in the following manner:
  - "Refactoring is the process of changing a software system in such a way that it does not alter the 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
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - or any other design failure that can be corrected to yield a better design.

OO Design Concepts

- Design classes
- Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

Design Classes

Five different types of design classes

- *User interface classes* define all abstractions that are necessary for **human computer interaction (HCI)**. In many cases, HCI occurs within the context of a *metaphor* (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.
- *Business domain classes* are often refinements 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 domain classes.
- *Persistent classes* represent data stores (e.g., a database) 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 and with the outside world.

FOUR CHARACTERISTICS OF DESIGN CLASS

- **Complete and sufficient.** A design class should be the complete **encapsulation of all attributes and methods** that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class
- **Primitiveness.** Methods associated with a design class should be focused on accomplishing **one service for the class**.
- **High cohesion.** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.
- **Low coupling.** Within the design model, it is necessary for **design classes to collaborate with one another.**

Aggregation:Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

Composition:Example: House (parent) and Room (child). Rooms don't exist separate to a House.

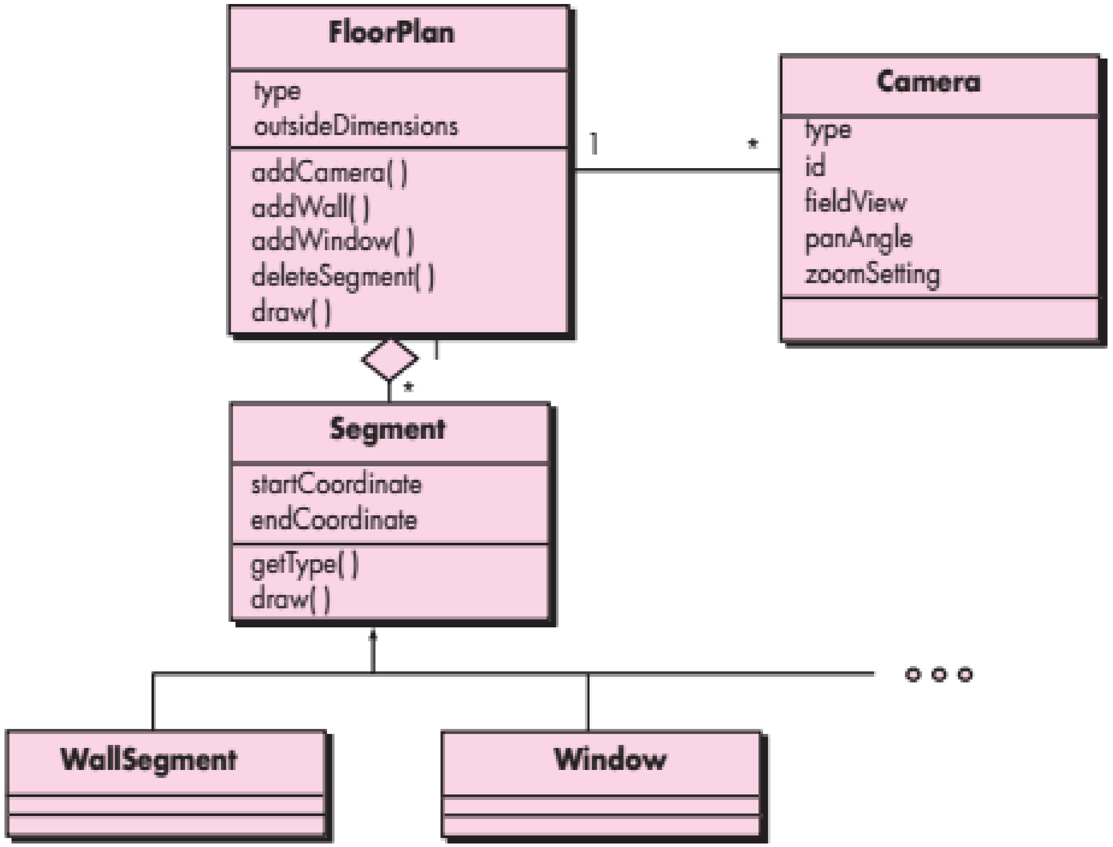
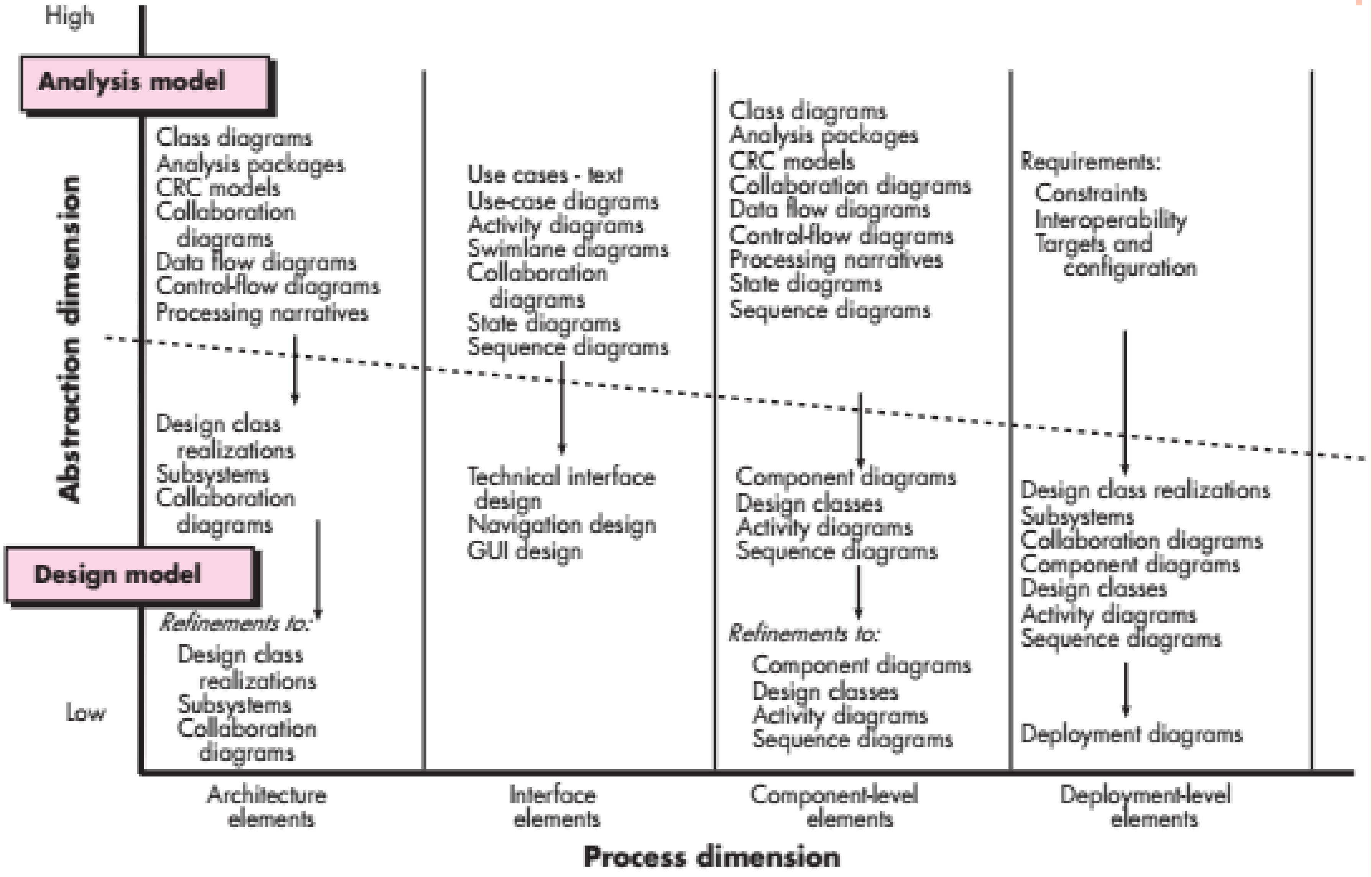


Fig: Design class for FloorPlan and composite aggregation for the class

The Design Model



DESIGN MODEL ELEMENTS

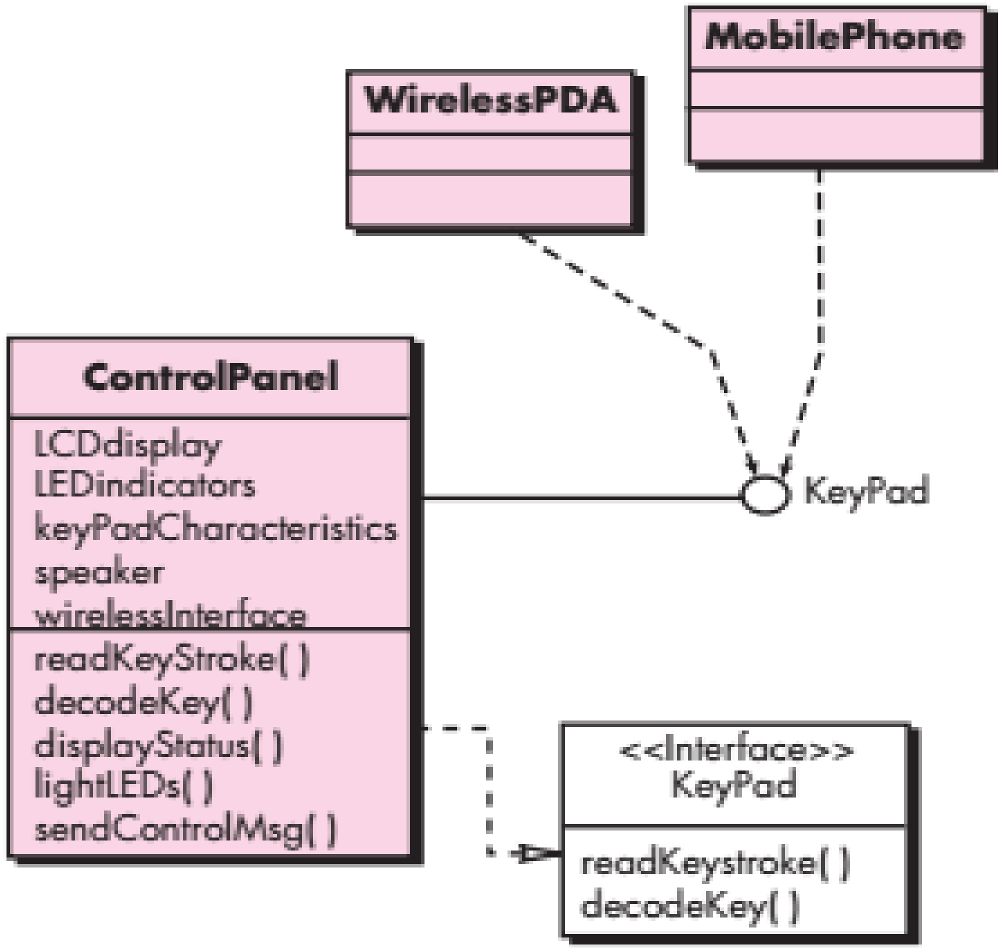
- Data design elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural design elements
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and "styles" (Chapters 9 and 12)
- Interface design 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 design elements
- Deployment design elements

Architectural Design Elements

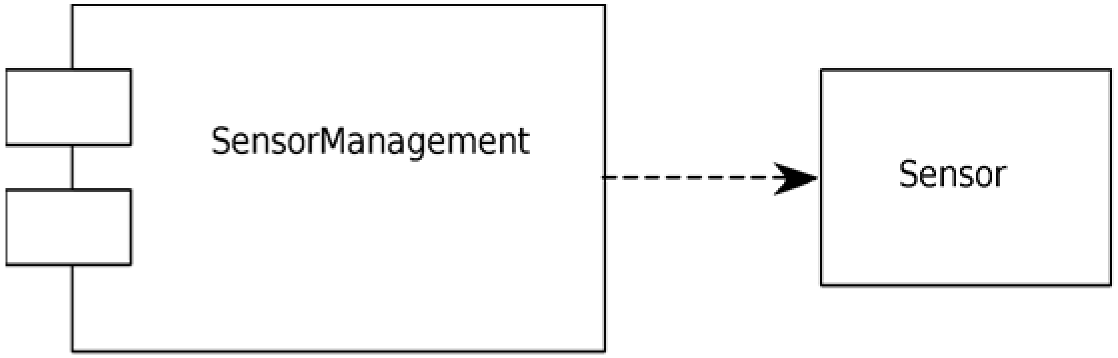
- The architectural model [Sha96] is derived from three sources:
  - information about the application domain for the software to be built;
  - specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
  - the availability of architectural patterns and styles.

Interface Design Elements

Fig: Interface representation for ControlPanel

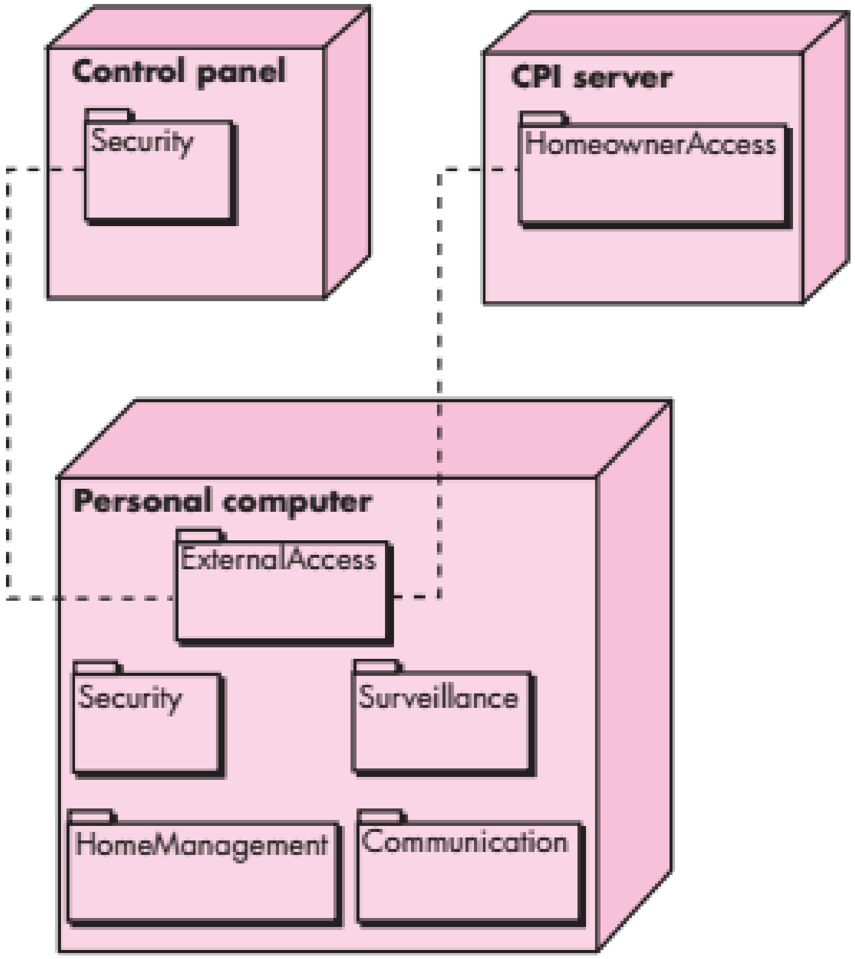


Component Design Elements



Deployment Design Elements

Fig: The UML deployment diagram



THANK YOU