UNIT III: Software Design

CHAPTER 12 DESIGN CONCEPTS

CHAPTER 13 ARCHITECTURAL DESIGN

CHAPTER 14 COMPONENT-LEVEL DESIGN

CHAPTER 12 DESIGN CONCEPTS

- 12.1 Design within the Context of Software Engineering
- 12.2 The Design Process
- 12.3 Design Concepts
- 12.4 The Design Model

CHAPTER 13 ARCHITECTURAL DESIGN

- 13.1 Software Architecture
- 13.3 Architectural Styles

CHAPTER 14 COMPONENT-LEVEL DESIGN

- 14.1 What Is a Component? (Component Views)
- 14.2 Designing Class-Based Components
- 14.3 Conducting Component-Level Design

UNIT III: Software Design

CHAPTER 12 DESIGN CONCEPTS

CHAPTER 13 ARCHITECTURAL DESIGN

CHAPTER 14 COMPONENT-LEVEL DESIGN

CHAPTER 12 DESIGN CONCEPTS

Software Design

- Encompasses the **set of principles (heuristics), concepts, and practices** that lead to the development of a high quality system or product

- Design principles establish and overriding philosophy that guides the designer as the work is performed.
- Design concepts must be understood before the mechanics of design practice are applied.
- Software design practices change continuously as new methods, better analysis, and broader understanding evolve.

.

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

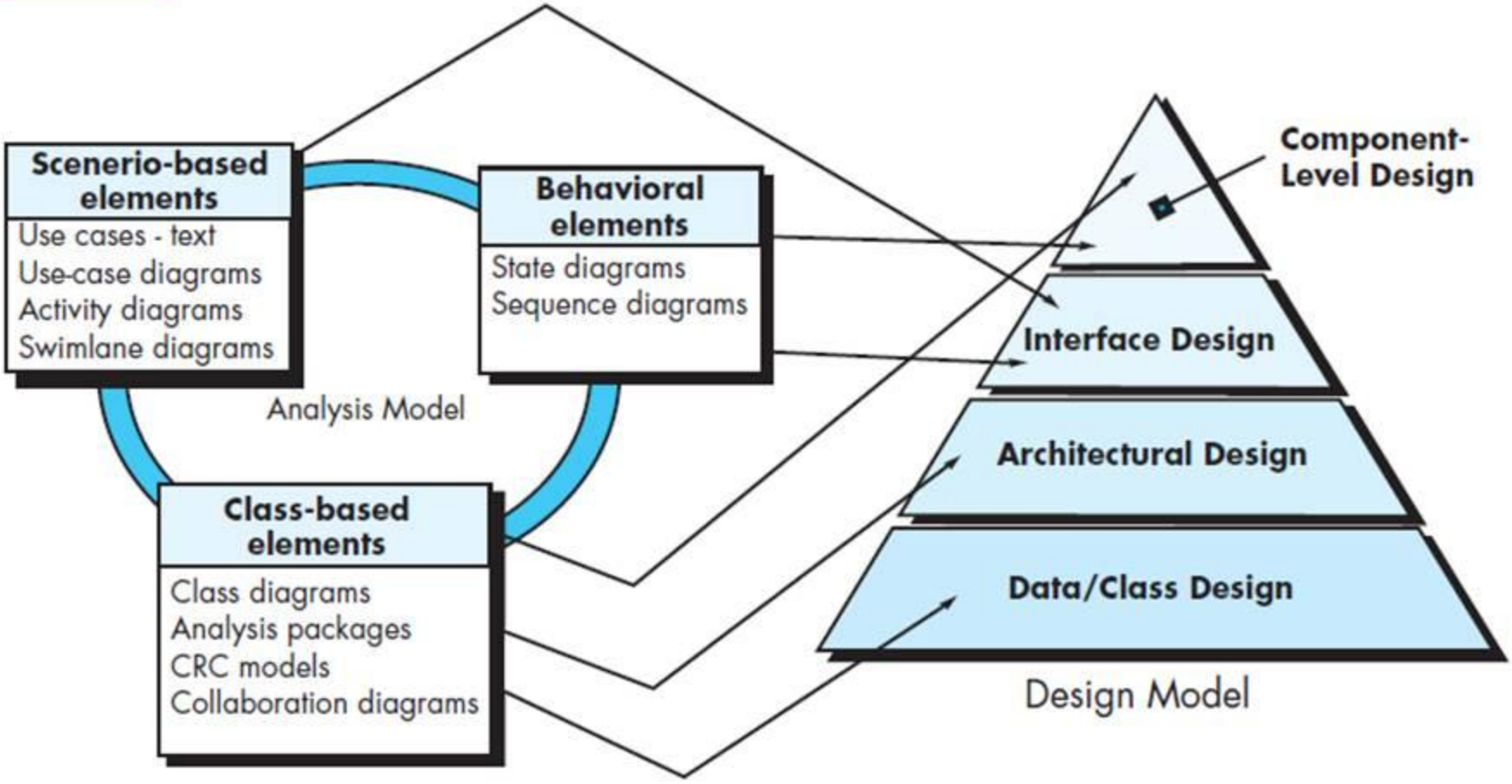
Good software design should exhibit:

- *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.

12.1 Design within the Context of Software Engineering

- Sits at the technical kernel of software engineering and applied regardless of process model.
- After requirement, it is last action within modeling activity and sets stage for construction.
- Elements of requirements model provides information that is necessary to create four design models required for complete specification of design.

FIGURE 12.1 Translating the requirements model into the design model



CRC: Class Responsibility Collaborator

12.1 Design within the Context of Software Engineering

1) Data/class design

- Transforms **class models into design class realizations** (data structures required) implement.
- **Objects and relationships** defined in CRC (Class Responsibility Collaborator) and detailed data content depicted by class attributes and other notation provide basis for data design action.
- **Part of class design** may occur with design of **software architecture.**

2) Architectural design

- Defines **relationship between major structural elements.**
- **Architectural styles and design patterns.**
- **Constraints** that affect the way in which architecture can be implemented.

3) Interface design

- Describes how **software communicates with systems that interoperate with it**, and with **humans who use it**.
- Implies a **flow of information (data/control) and a specific type of behavior.**
- **Usage scenarios and behavioral models** provide information.

4) Component-level design

- Transforms **structural elements of software architecture into a procedural description** of software components.
- Information obtained from **class-based models, flow models, and behavioral models** serve as the basis.

Design and Quality (Importance of Design)

- Design
  - helps to promote or foster quality in SE.
  - Only way to accurately translate stakeholder's requirements into a finished software product.
  - serves as foundation for all SE and support activities that follow.

Without design, we risk building an unstable system

- one that will fail when small changes are made;
- one that may be difficult to test;
- one whose quality cannot be assessed until late in software process (time is short and many dollars spent).

12.2 Design Process: 12.2.1 Software Quality Guidelines and Attributes

- **Iterative** process with **high level to low level** abstraction.
- **Software Design Characteristics (Goals)**
  - design must implement all explicit requirements contained in analysis model, and it must accommodate all of implicit requirements.
  - design must be a readable, understandable guide for those who generate code, test and support.
  - design should provide a complete picture of software, addressing data, functional, and behavioral domains from an implementation perspective.

12.2 Design Process: 12.2.1 Software Quality Guidelines and Attributes

Technical Criteria for Good Design (Guidelines to reach goals)

1. **Exhibit an architecture** that (1) is created using **recognizable architectural styles or patterns,** (2) composed of components that exhibit **good design characteristics** and (3) can be implemented in an **evolutionary fashion.**

2. **Modular**; logically partitioned into elements or subsystems.

3. **Distinct representations** of **data, architecture, interfaces, and components.**

4. **Should lead to data structures that are appropriate** for classes to be implemented and are drawn from recognizable data patterns.

12.2 Design Process: 12.2.1 Software Quality Guidelines and Attributes

Technical Criteria for Good Design (Guidelines to reach goals)

5. **Should lead to components that exhibit independent functional characteristics.**

6. **Should lead to interfaces that reduce the complexity** of connections between components and with external environment.

7. **Should be derived using a repeatable method** that is driven by information obtained during software requirements analysis.

8. **Should be represented using a notation that effectively communicates its meaning.**

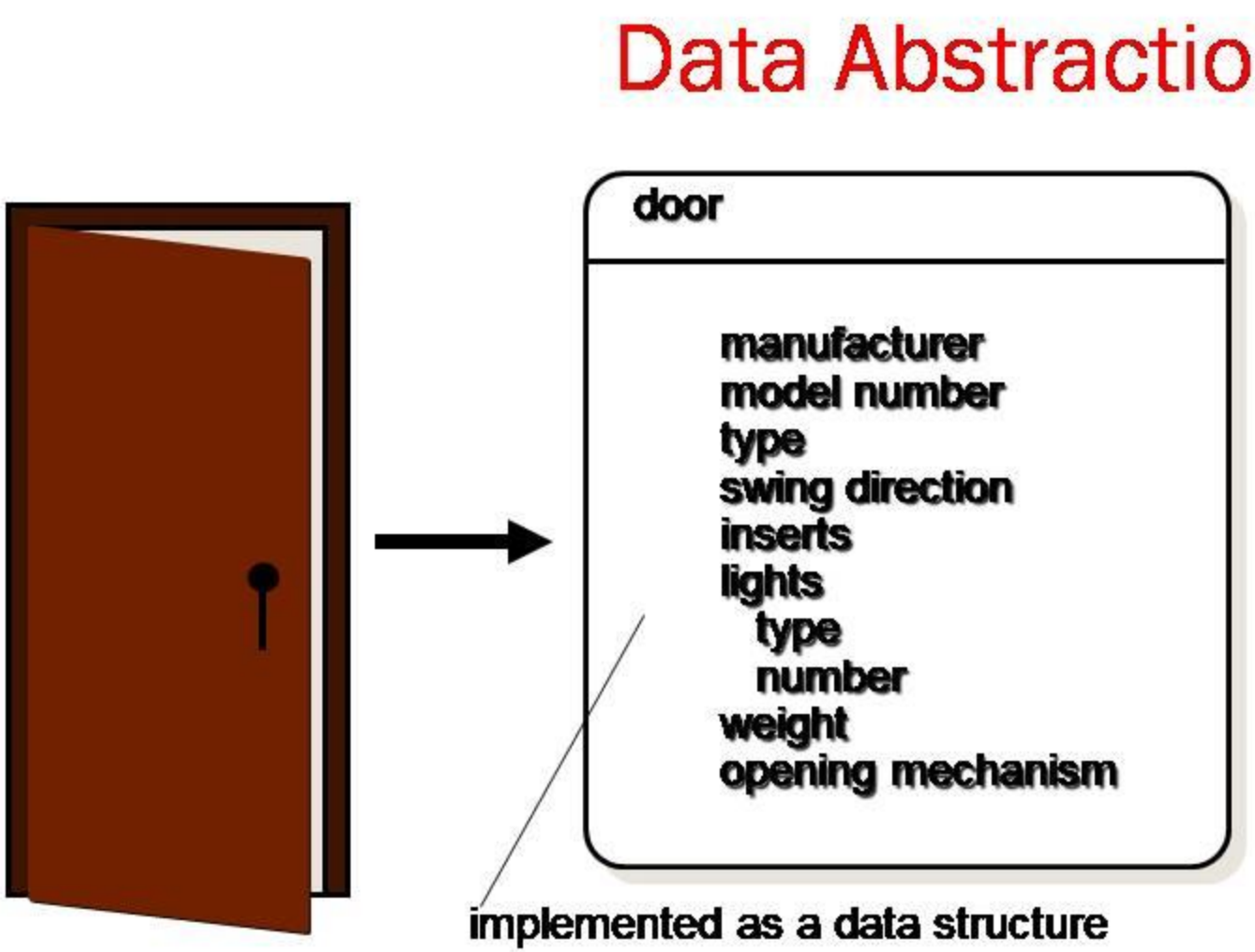
- **Design Quality Attributes (FURPS)** **(measures of software quality)**
  - **Functionality:** feature set and capabilities of program, generality of functions, and security of the system.
  - **Usability:** considering human factors, aesthetics, consistency, and documentation.
  - **Reliability:** frequency and severity of failure, accuracy of output results, MTTF, MTTR, and predictability of program.
  - **Performance** processing speed, response time, resource consumption, throughput, and efficiency.
  - **Supportability** ability to extend program, adaptability, serviceability, maintainability, testability, compatibility, configurability.

- **Traditional Approaches**
- **Object Oriented Approaches**
- **Design Patterns**
- **Aspect Oriented**

- **Note:** Regardless of the design method, apply a set of basic concepts to data, architectural, interface, and component-level design. They are discussed in next section.

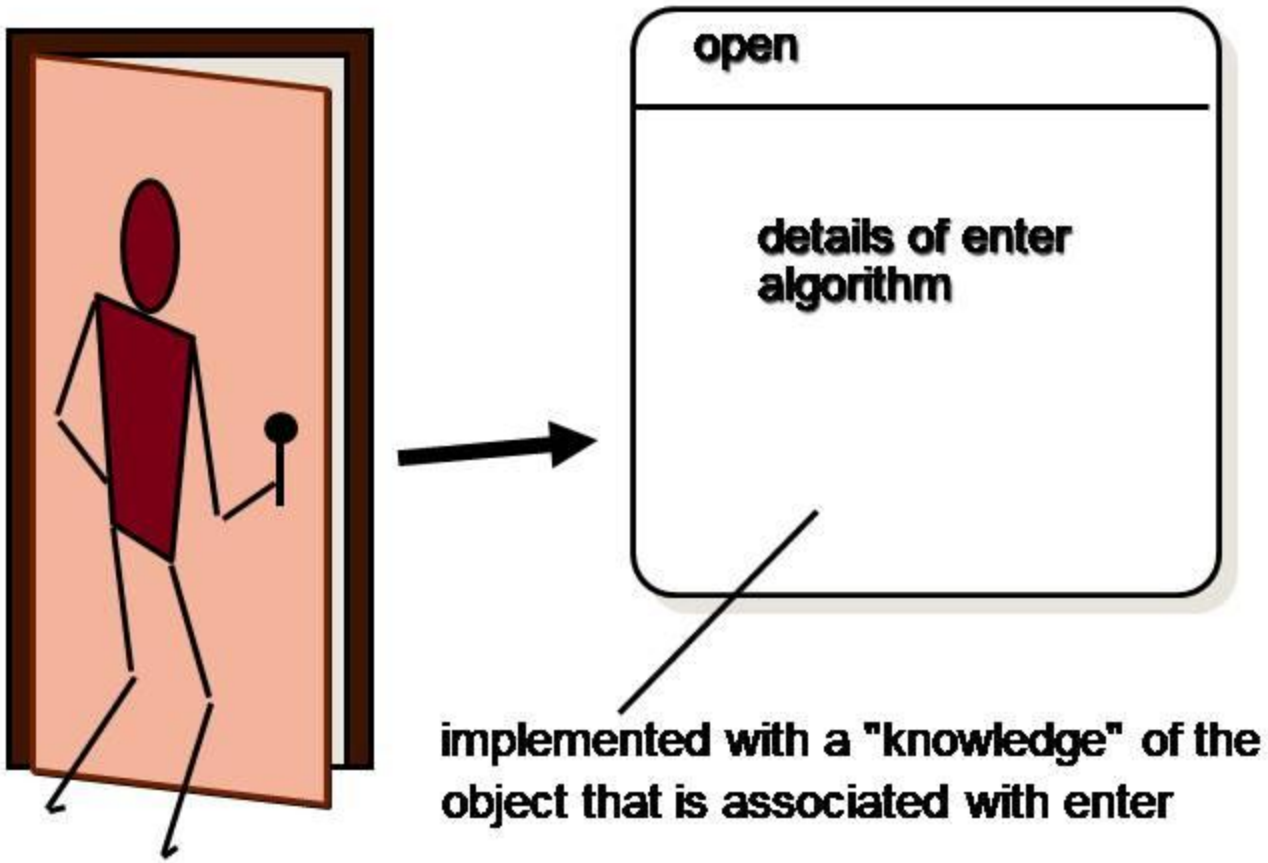
OO concepts, database concepts etc.

i) **Abstraction**—data, procedure, control



i) **Abstraction**—data, procedure, control

Procedural Abstraction

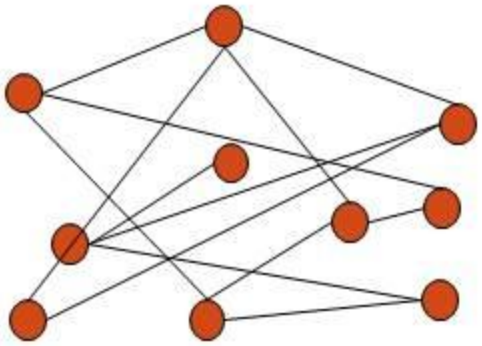


ii) Architecture—overall structure of and ways in which that structure provides conceptual integrity for system. [SHA95a]

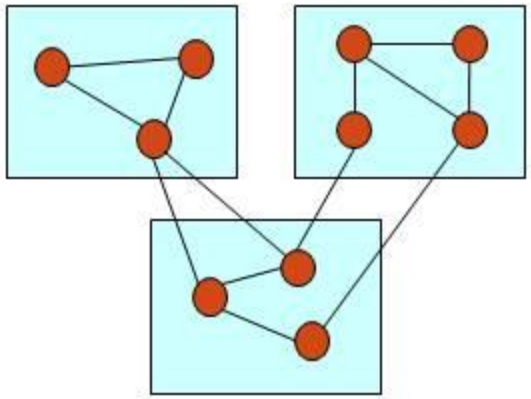
- **Structural properties.** defines components of a system (e.g., modules, objects, filters) and manner in which those components are packaged and interact with one another. **EX:** objects are packaged to encapsulate both data and processing.
- **Extra-functional properties.** design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- **Families of related systems.** architectural design should draw upon repeatable patterns that are commonly encountered in design of families of similar systems. Design should have the ability to reuse architectural building blocks.

iii) Modularity —compartmentalization of data and function (has relation with SoC)

- Allows a program to be intellectually manageable [Mye78].
- Monolithic software (large program with 1 module) cannot be easily grasped
  - Number of control paths, span of reference, number of variables, and overall complexity.
- Break design into modules, hoping to make understanding easier and reduce cost.



10 parts, 13 connections



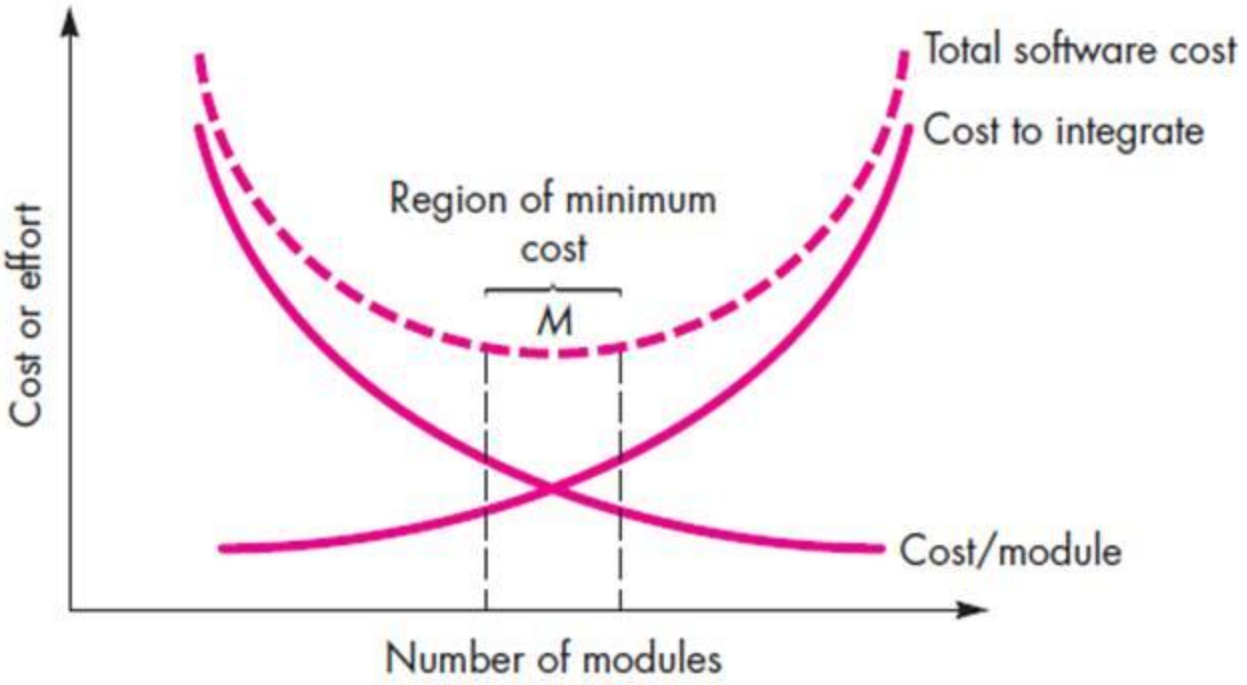
10 parts, 13 connections, 3 modules

iii) Modularity — Trade-Offs

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

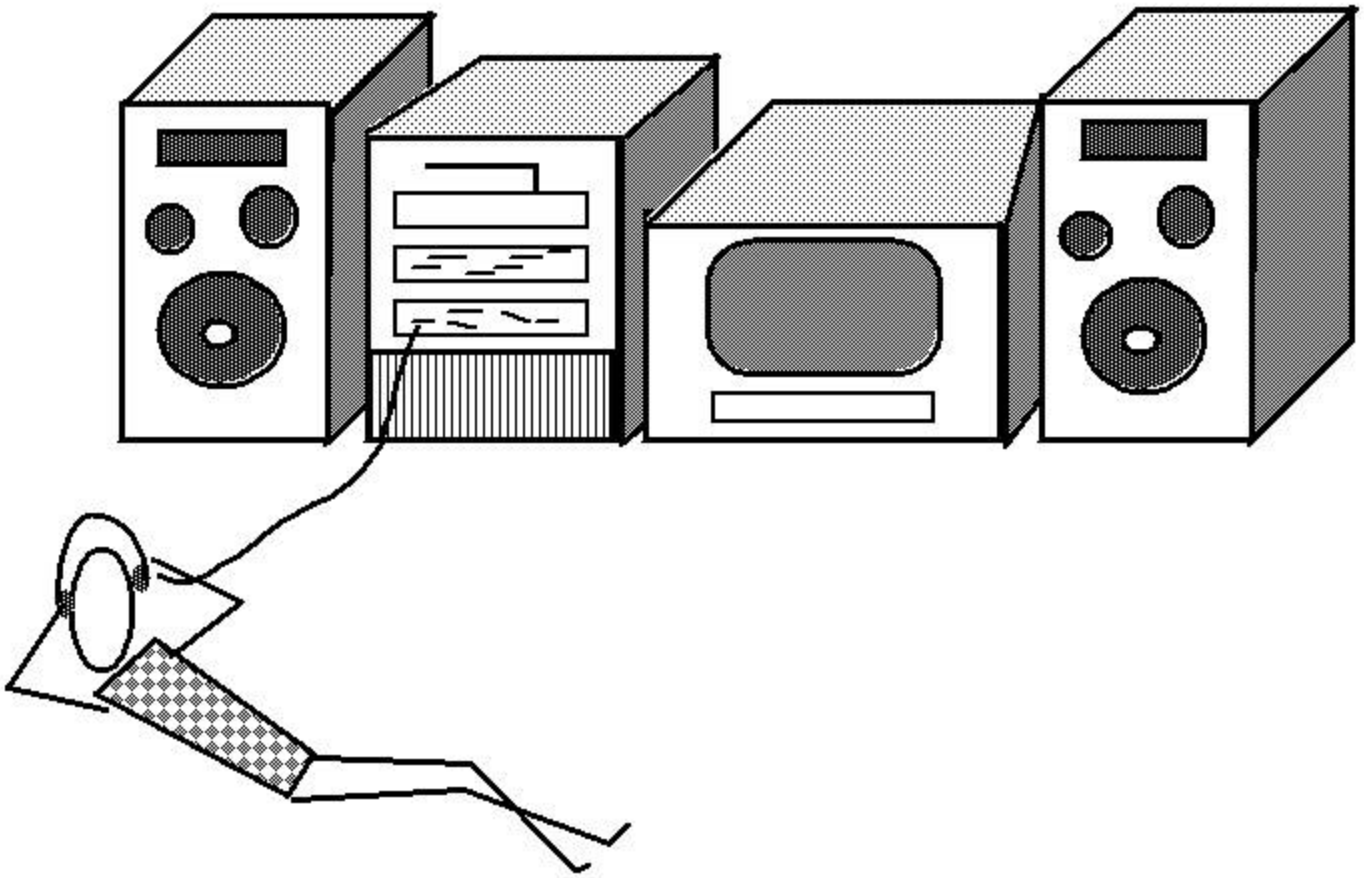
FIGURE 8.2

Modularity and software cost

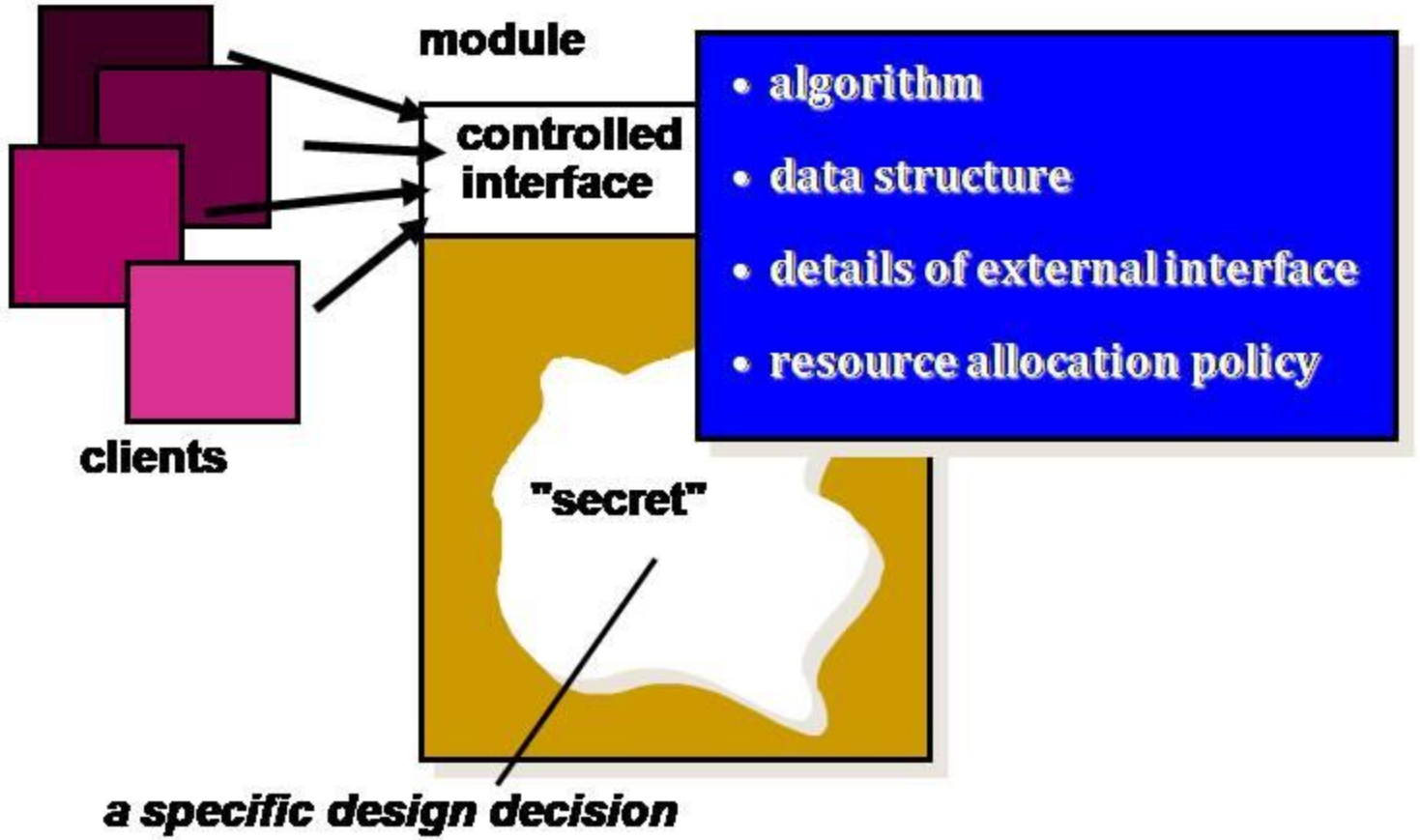


iii) Modularity

easier to build, easier to change, easier to fix ...



iv) **Information Hiding** — controlled interfaces (set of modules)

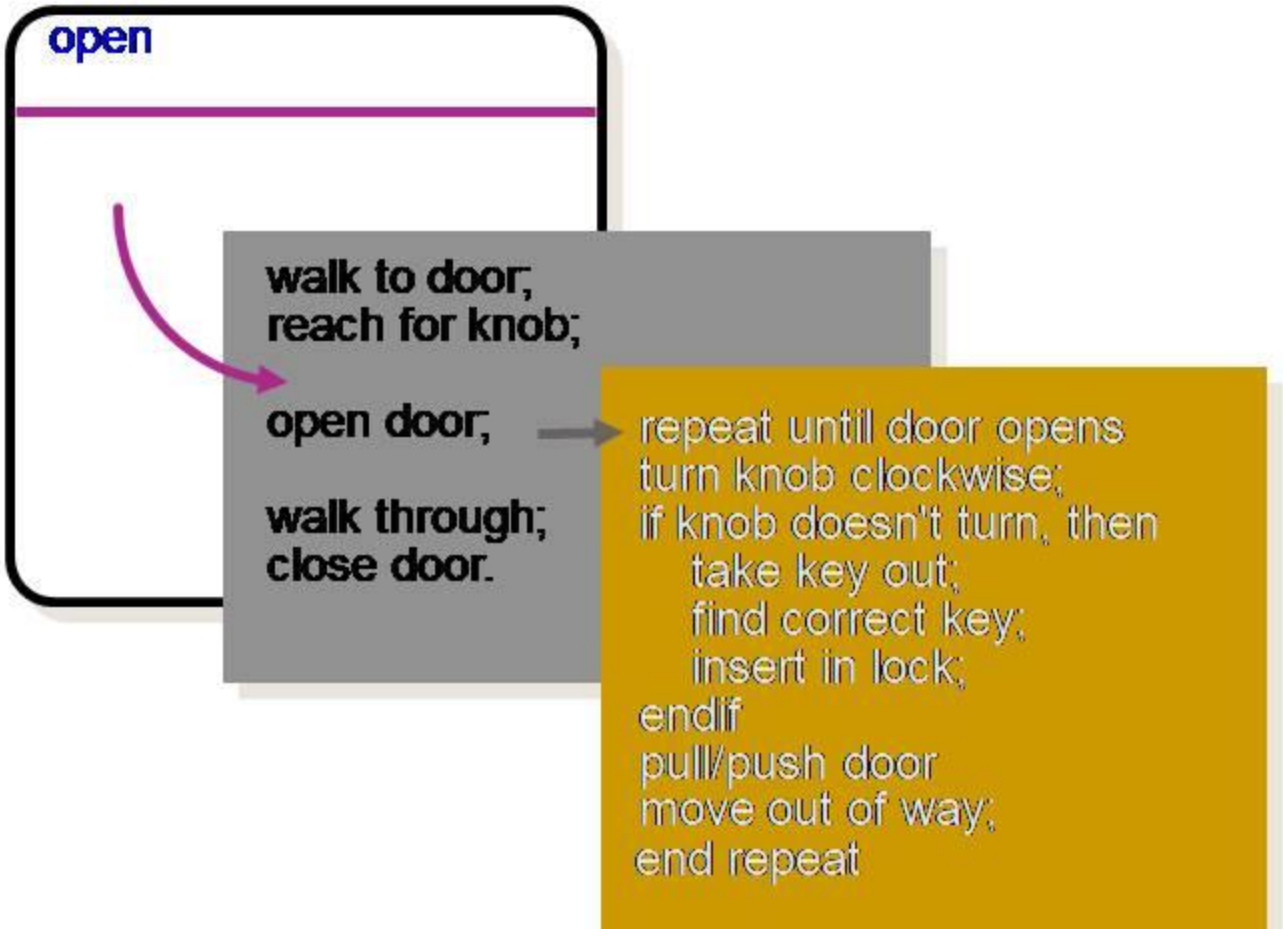


iv) Information Hiding — controlled interfaces

- Modules specified and designed such that information (algorithms and data) contained within is inaccessible to other modules that have no need for such information.
  - reduces the likelihood of "side effects"
  - Limits 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

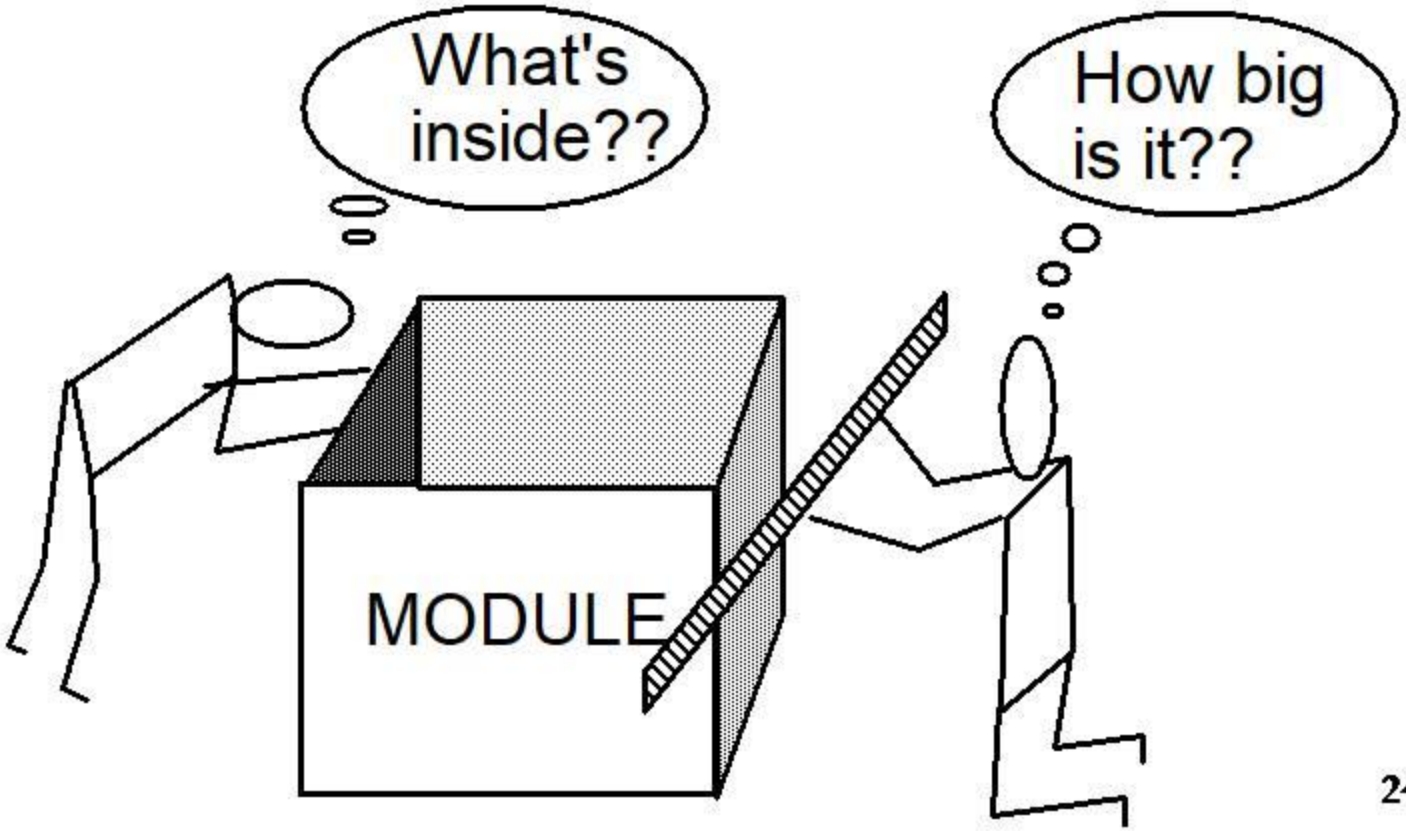
iv) Information Hiding — controlled interfaces

Stepwise Refinement



iv) Information Hiding — controlled interfaces

Sizing Modules: Two Views



(v) Functional Independence - achieved by developing modules with **single-minded** function and an **aversion to excessive interaction** with other modules. Measured using

Cohesion:

- indication of relative functional strength of a module.
- cohesive module performs single task, requiring less interaction with other components.
- cohesive module should do just one thing.

Coupling:

- indication of relative interdependence among modules.
- Coupling depends on interface complexity, point at which entry or reference is made to a module, and what data pass across interface.

vi) Refinement - elaboration of detail for all abstractions

- Stepwise refinement is a top-down design strategy.
- Refinement is a process of *elaboration*. *Begin with a statement of* function (or description of information) defined at high level of abstraction.
- Then elaborate on original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Abstraction and refinement are complementary.
- Abstraction enables to specify procedure and data internally but suppress need for "outsiders" to have knowledge of low-level details.
- Refinement helps to reveal low-level details as design progresses.
- Both concepts allow you to create a complete design model as the design evolves.

vii) Refactoring—a reorganization technique that simplifies the design

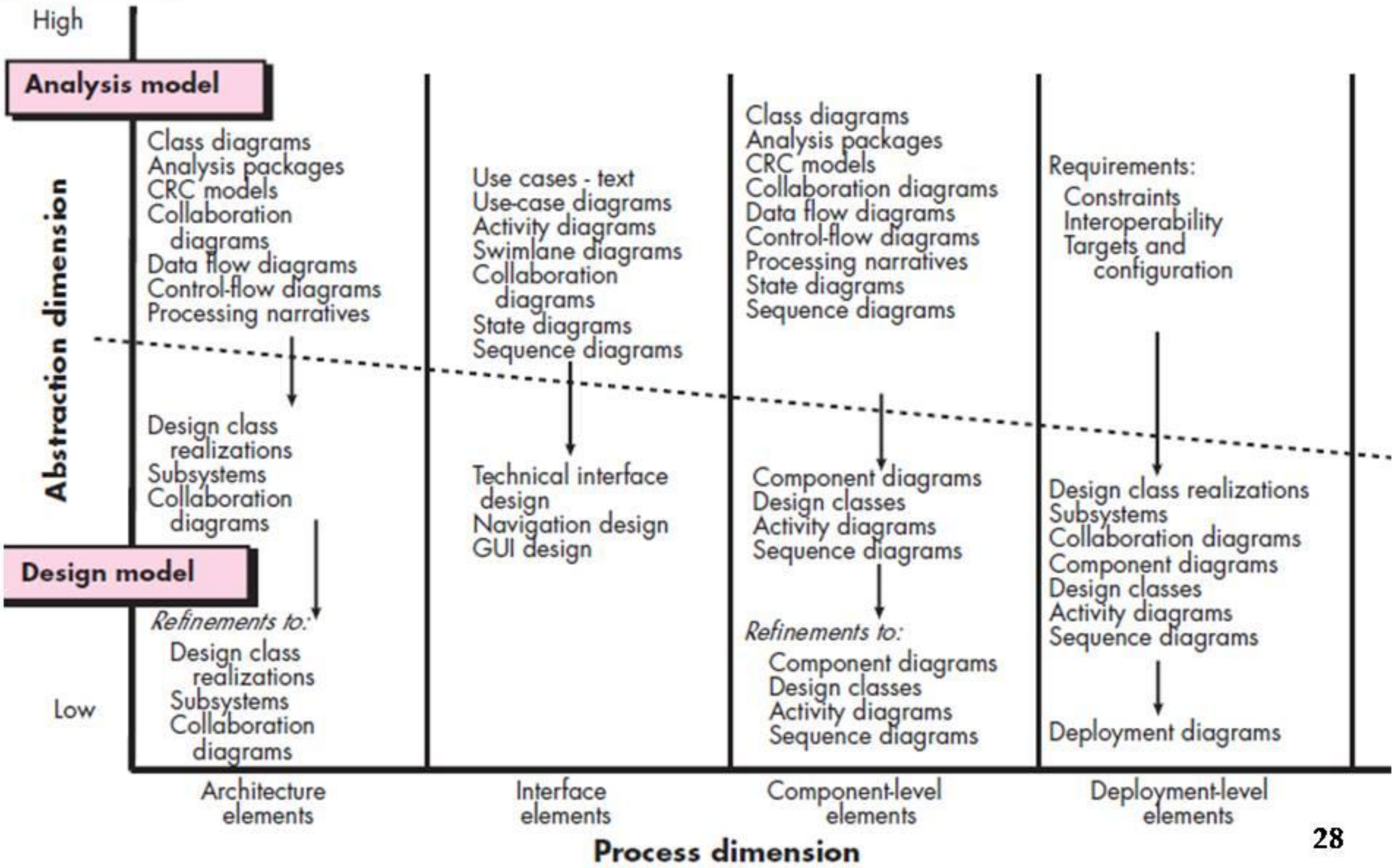
- Important design activity suggested for many agile methods, it is reorganization technique that simplifies design of a component.

- 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.

8.4 The Design Model:

FIGURE 8.4 Dimensions of the design model



8.4 The Design Model:

- Data Design elements : Data architecture diagram: Front office
  - 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

Note: May not be followed in sequence

8.4 The Design Model:

A) Data Design elements : Data architecture diagram:

- Data model --> data structures
- Data model --> database architecture

- Data design creates model of data/information that is represented at a high level of abstraction (**customer/user's view).**

- At the program component level, the **design of data structures and the associated algorithms** required to manipulate them is essential to the creation of high-quality application.

  At application level, the **translation of data model into a database** is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and recognized into a "warehouse" enables data mining or knowledge discovery

8.4 The Design Model:

A) Data Design elements : Data architecture diagram:

- examines data objects independently of processing
- indicates how data objects relate to one another

B) 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 and styles.

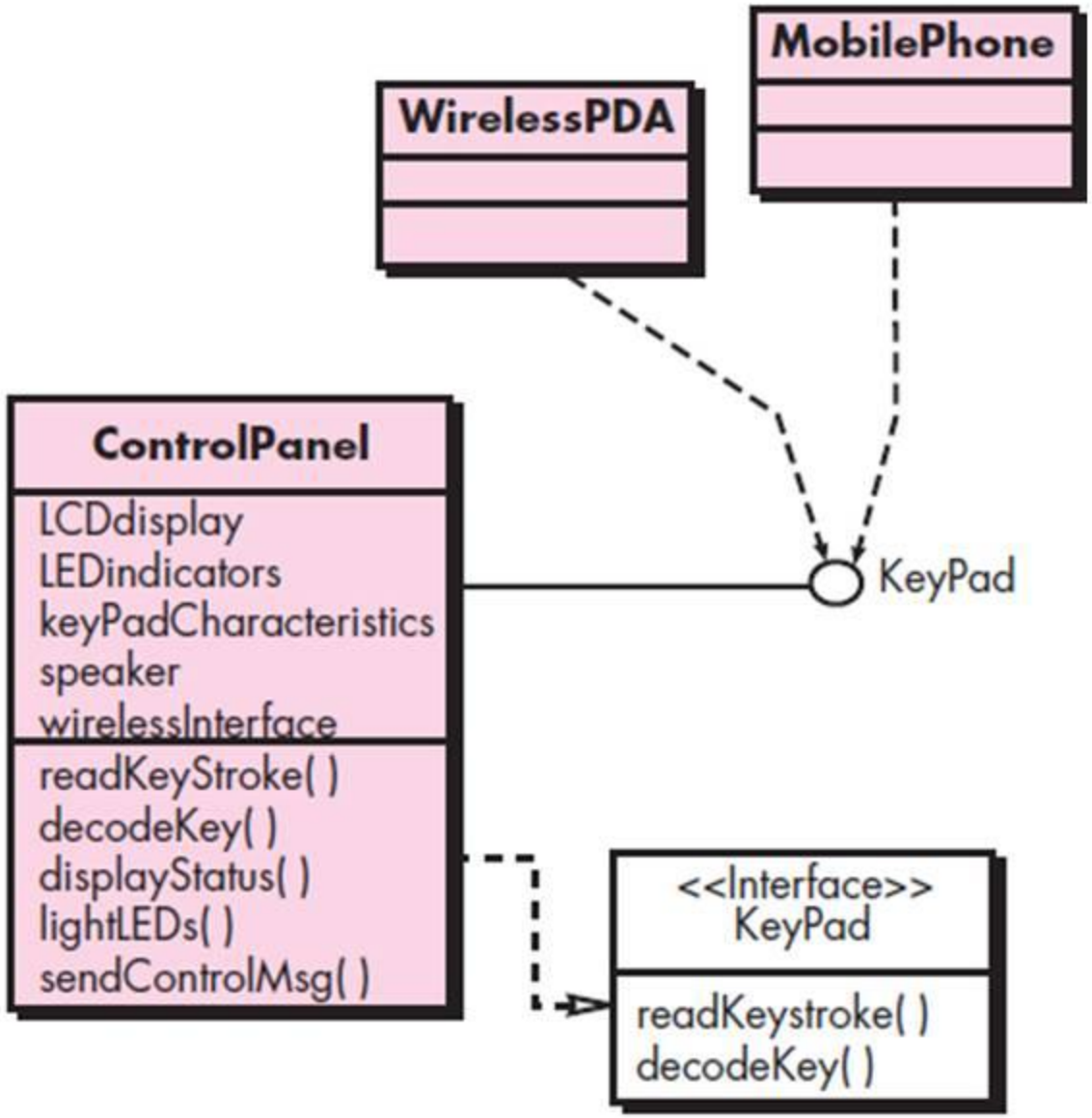
C) Interface Elements

- Interface is a **set of operations that describes** the externally observable behavior of a class and provides access to its public operations
- **Three** major elements:
  - User interface
  - External interface to other system, device, networks or producers or consumers of information
  - Internal interfaces between components.
- **Modeled** using UML communication diagrams (called collaboration diagrams in UML 1.x)

C) Interface Elements

FIGURE 8.5

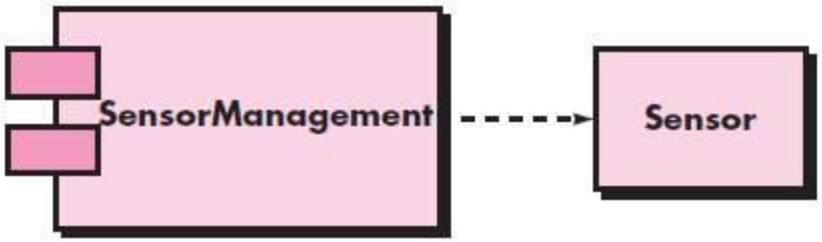
Interface representation for Control-Panel



D) 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

FIGURE 8.6
A UML component diagram



E) 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

E) Deployment Elements

FIGURE 8.7
A UML deployment diagram