- **Design Concepts**

Analysis Model -> Design Model

scenario-based elements
- use-cases - text
- use-case diagrams
- activity diagrams
- swim lane diagrams

flow-oriented elements
- data flow diagrams
- control-flow diagrams
- processing narratives

class-based elements
- class diagrams
- analysis packages
- CRC models
- collaboration diagrams

behavioral elements
- state diagrams
- sequence diagrams

Analysis Model

Component-Level Design

Interface Design

Architectural Design

Data/Class Design

Design Model

Design and Quality

- 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.
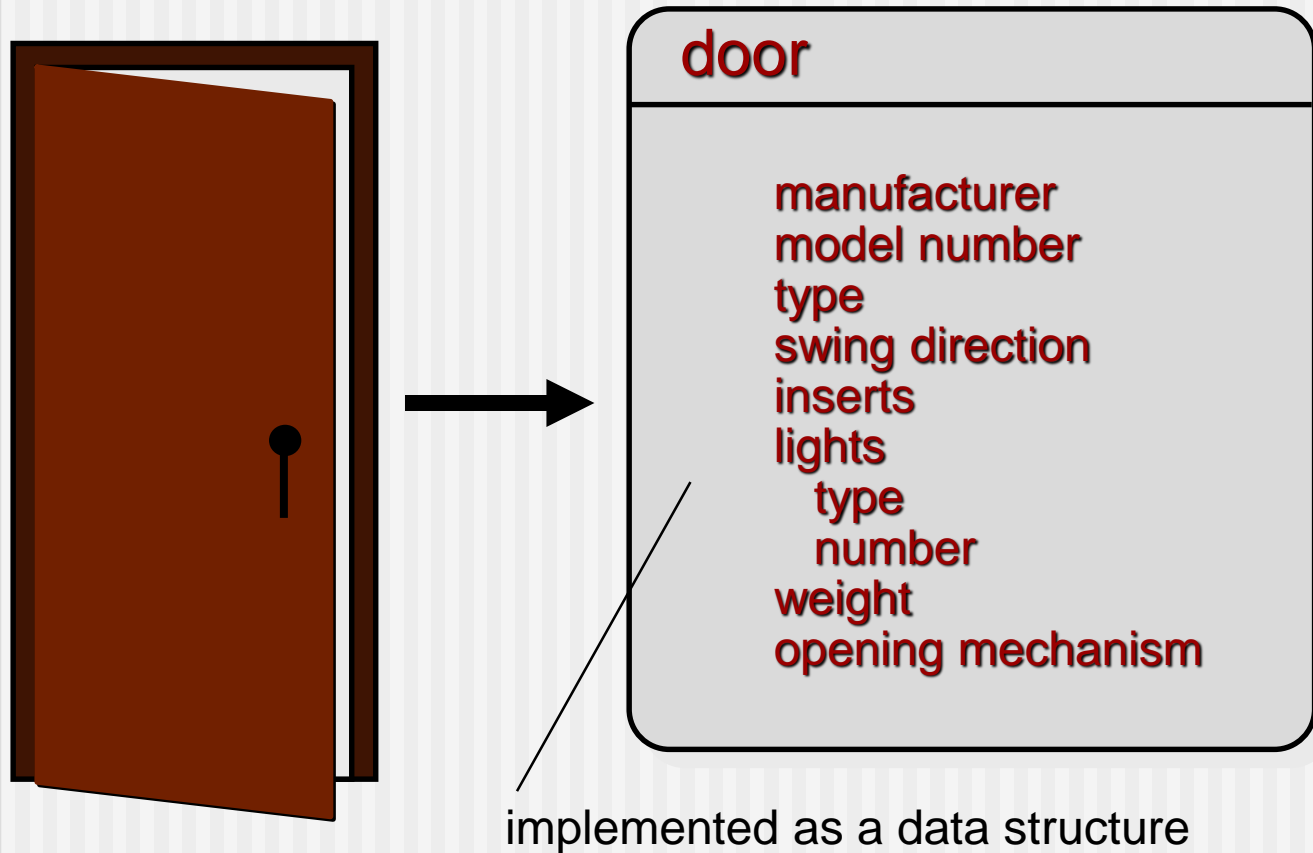
Design Principles

- The design process should not suffer from 'tunnel vision.'
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should "minimize the intellectual distance" [DAV95] between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
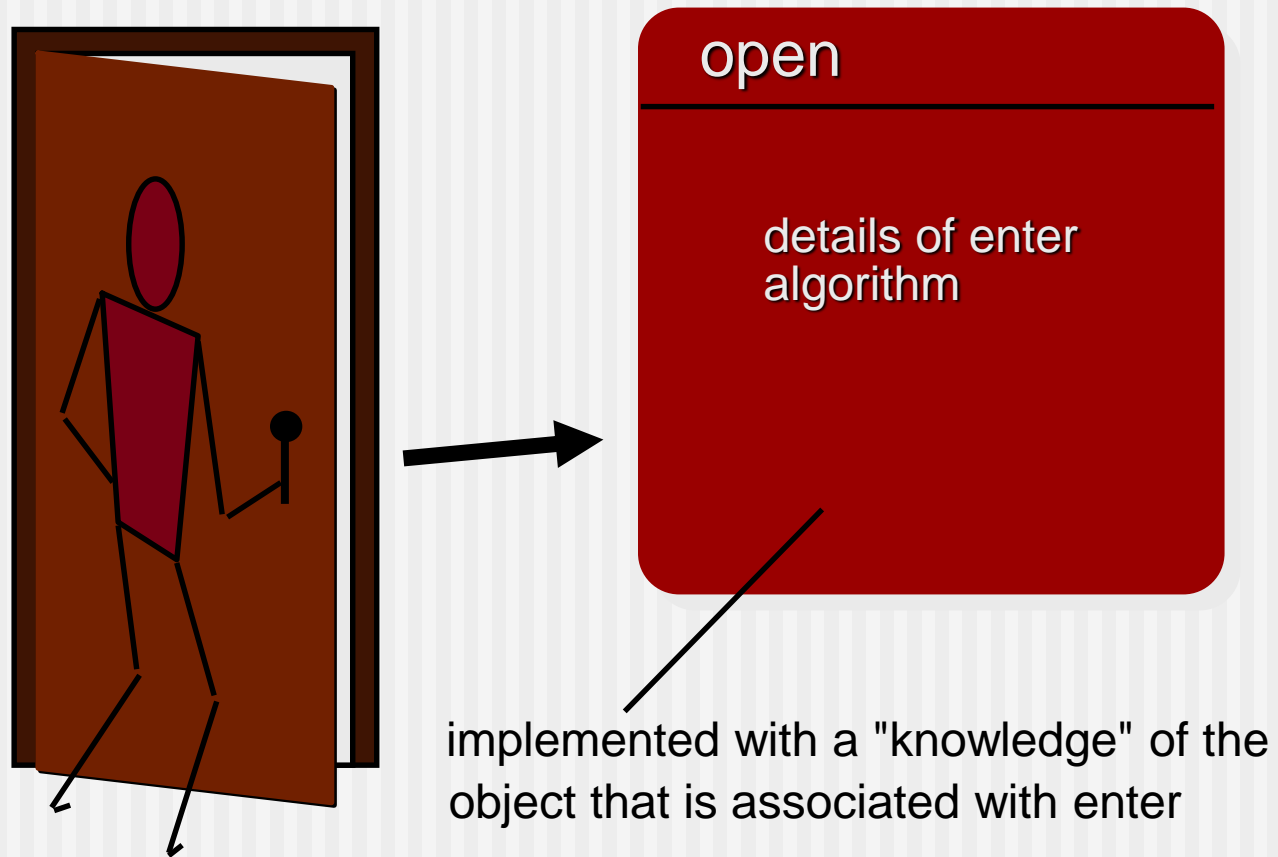- The design should be reviewed to minimize conceptual (semantic) errors.

Fundamental Concepts

- **Abstraction**—data, procedure, control
- **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

door

manufacturer
model number
type
swing direction
inserts
lights
  type
  number
weight
opening mechanism

implemented as a data structure

Procedural Abstraction

open

details of enter algorithm

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
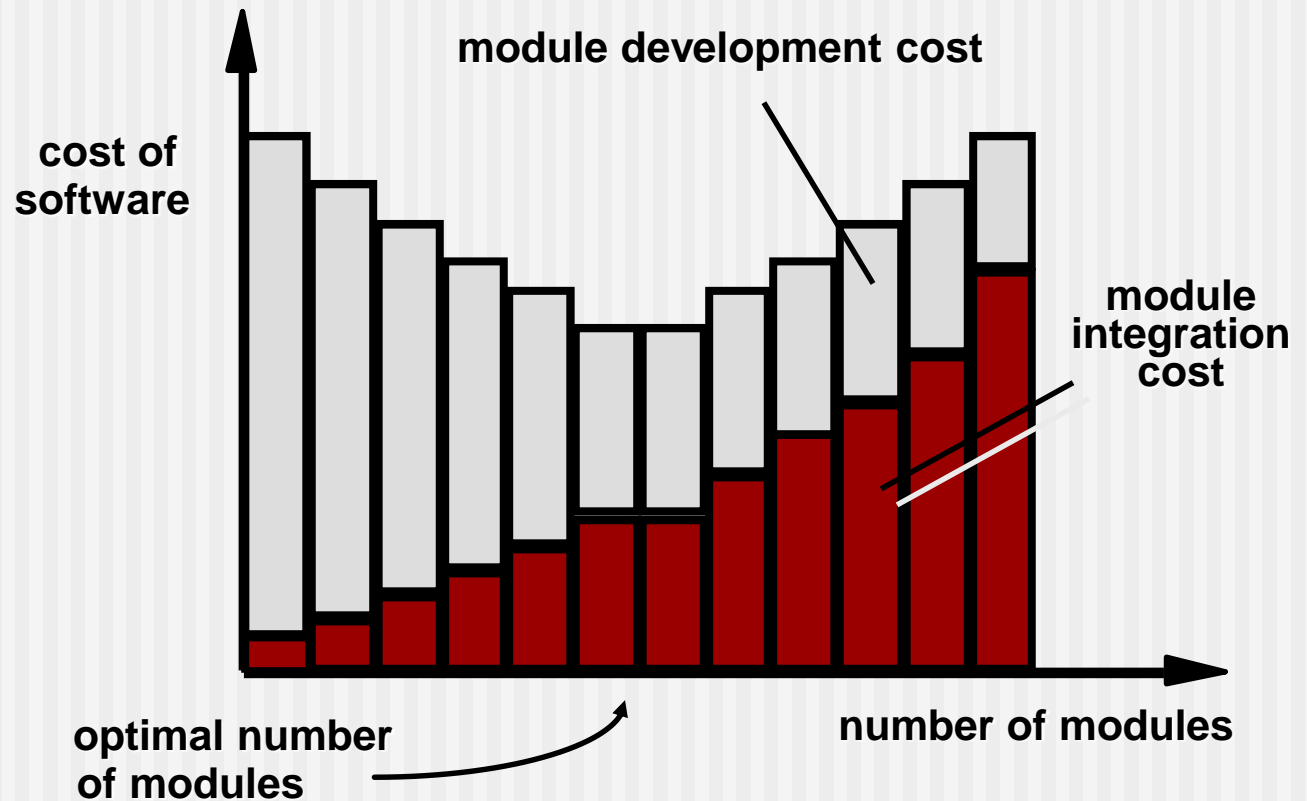
Separation of Concerns

- 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.
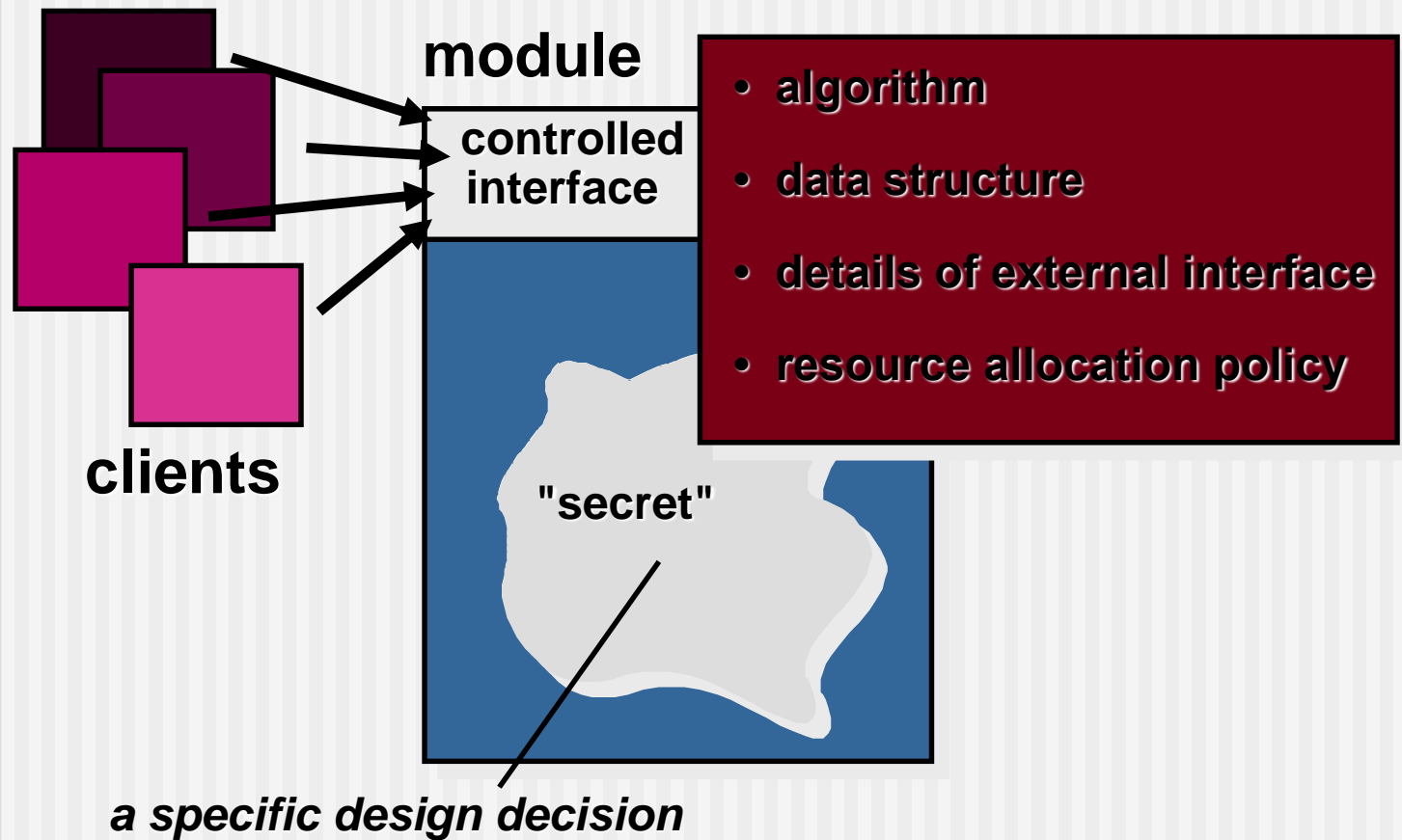
Modularity

- "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: Trade-offs

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

cost of software

module development cost

module integration cost

optimal number of modules

number of modules

Information Hiding

module

controlled interface

- **algorithm**
- **data structure**
- **details of external interface**
- **resource allocation policy**

clients

"secret"

a specific design decision
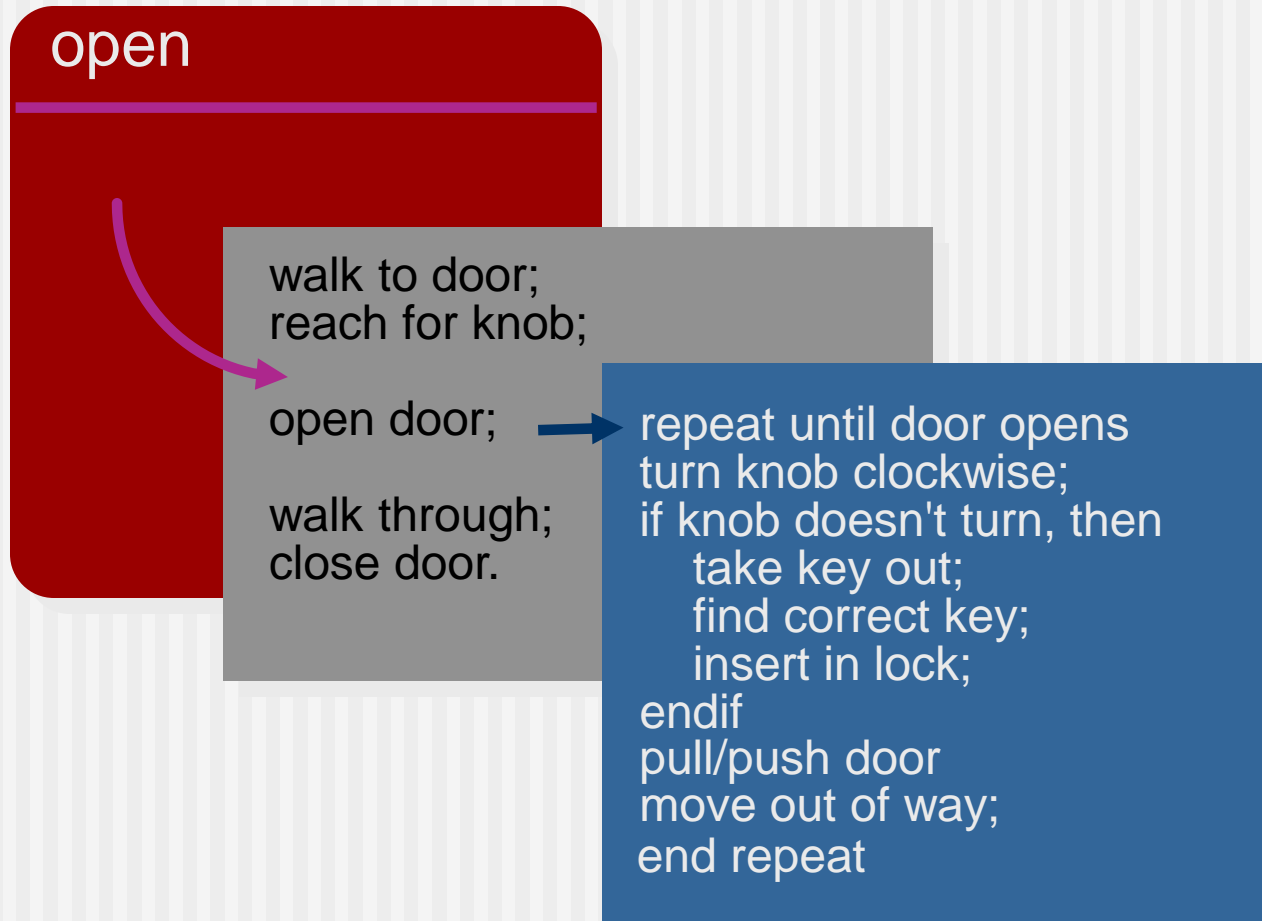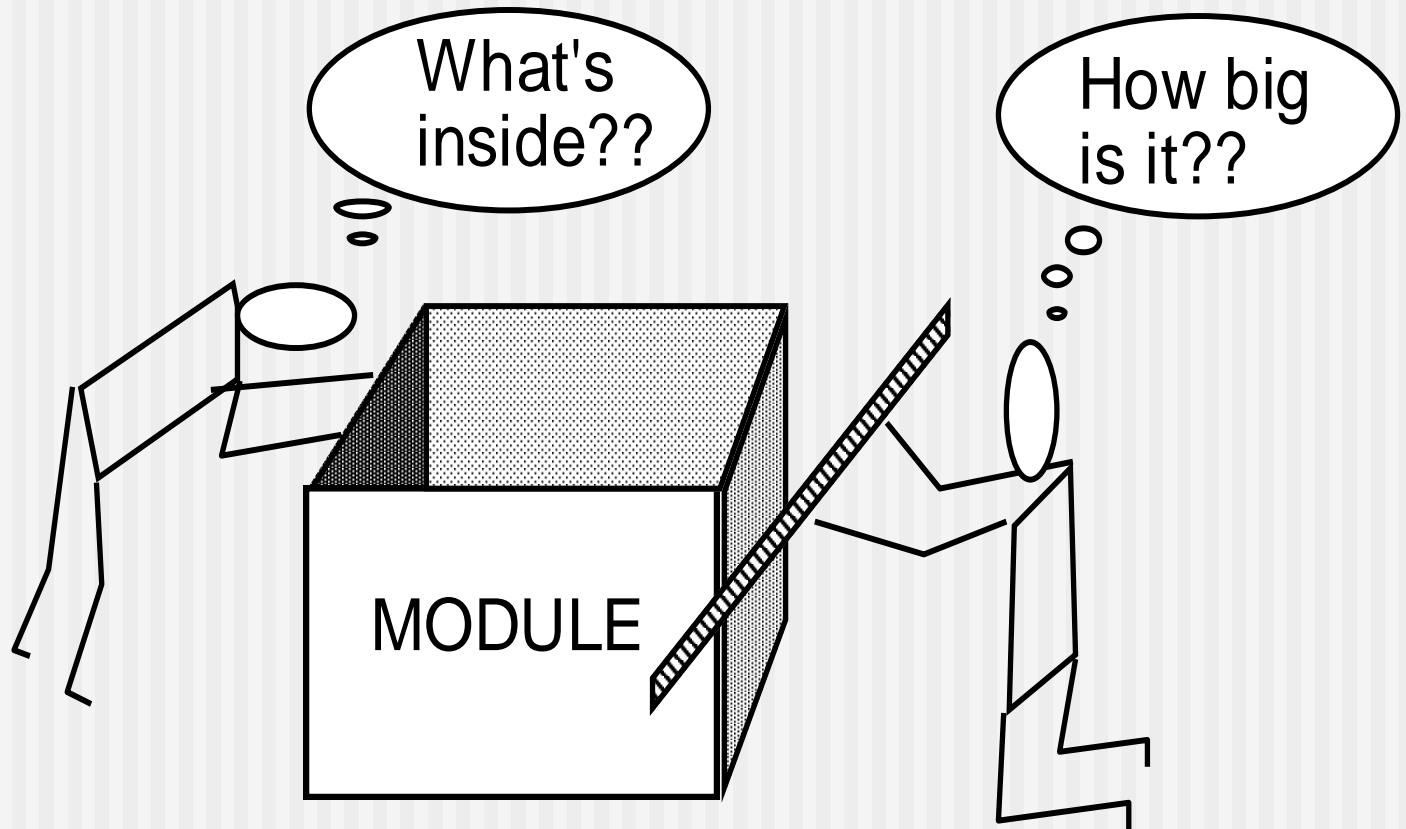
Why Information Hiding?

- 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;
endif
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.

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.
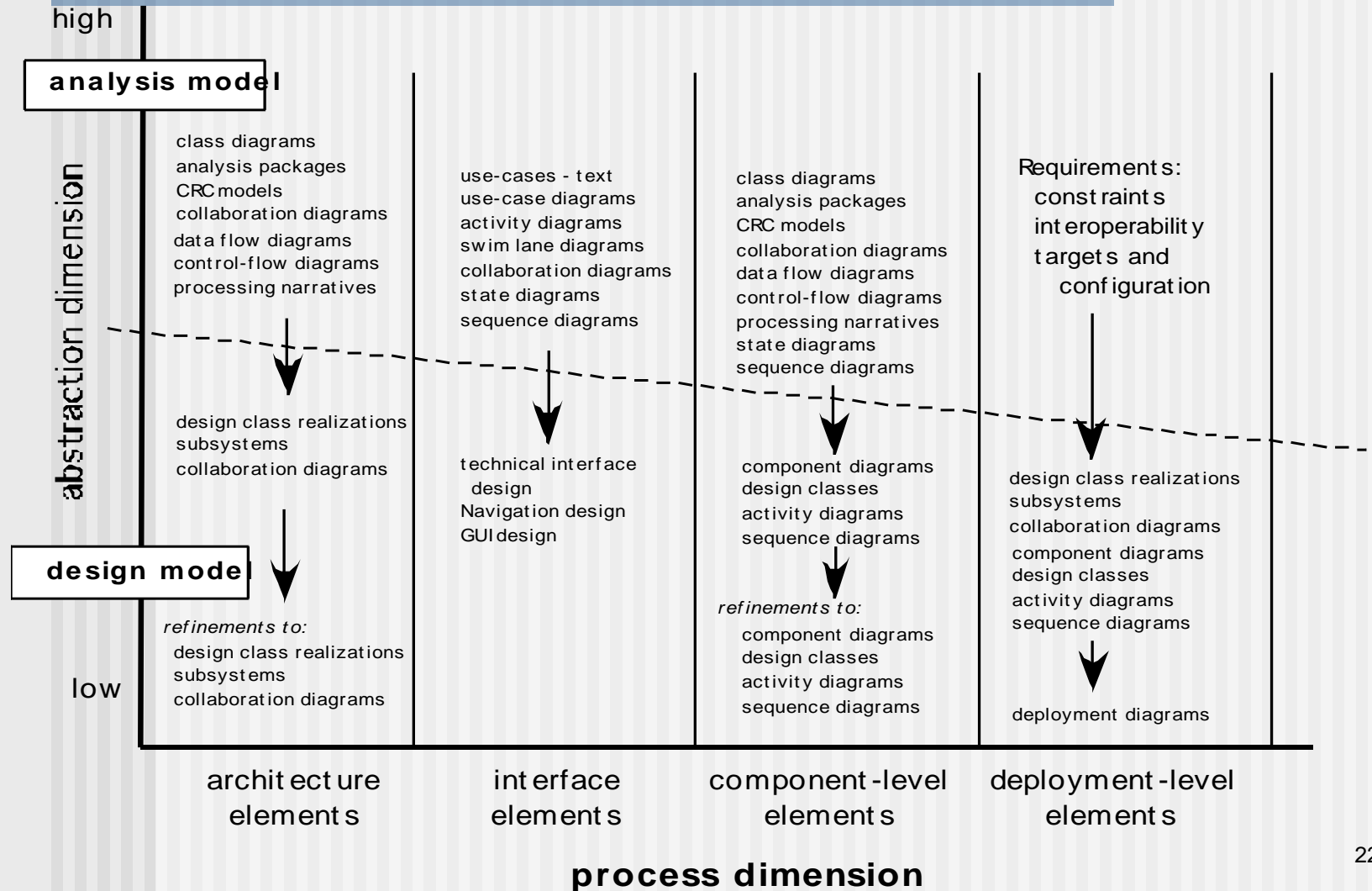
OO Design Concepts

- **Design classes**
  - Entity classes
  - Boundary classes
  - Controller 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

- Analysis classes are refined during design to become entity classes
- Boundary classes are developed during design to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
  - Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- Controller classes are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

The Design Model

high

analysis model

abstraction dimension

class diagrams
analysis packages
CRC models
collaboration diagrams
data flow diagrams
control-flow diagrams
processing narratives

use-cases - text
use-case diagrams
activity diagrams
swim lane diagrams
collaboration diagrams
state diagrams
sequence diagrams

class diagrams
analysis packages
CRC models
collaboration diagrams
data flow diagrams
control-flow diagrams
processing narratives
state diagrams
sequence diagrams

Requirements:
constraints
interoperability
targets and
configuration

design class realizations
subsystems
collaboration diagrams

technical interface
design
Navigation design
GUI design

component diagrams
design classes
activity diagrams
sequence diagrams

design class realizations
subsystems
collaboration diagrams
component diagrams
design classes
activity diagrams
sequence diagrams

design model

refinements to:
design class realizations
subsystems
collaboration diagrams

low

refinements to:
component diagrams
design classes
activity diagrams
sequence diagrams

deployment diagrams

architecture
elements

interface
elements

component-level
elements

deployment-level
elements
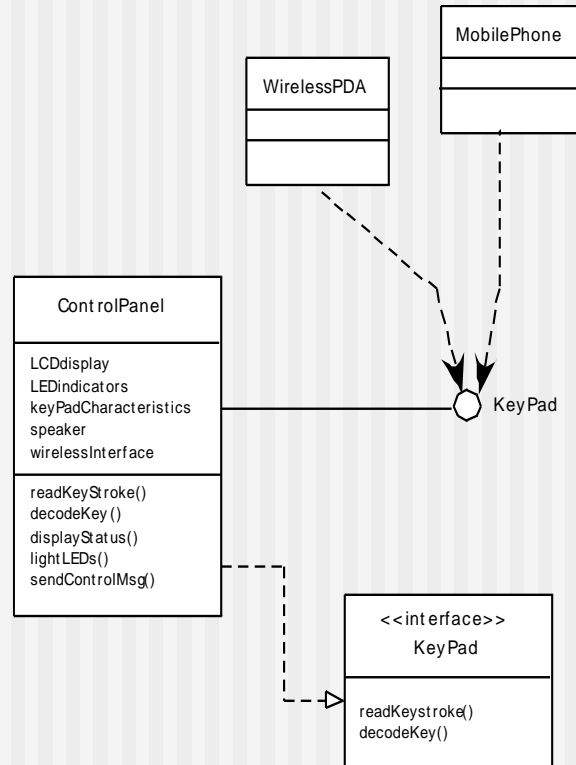
process dimension

22

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" (Chapters 9 and 12)
- **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.
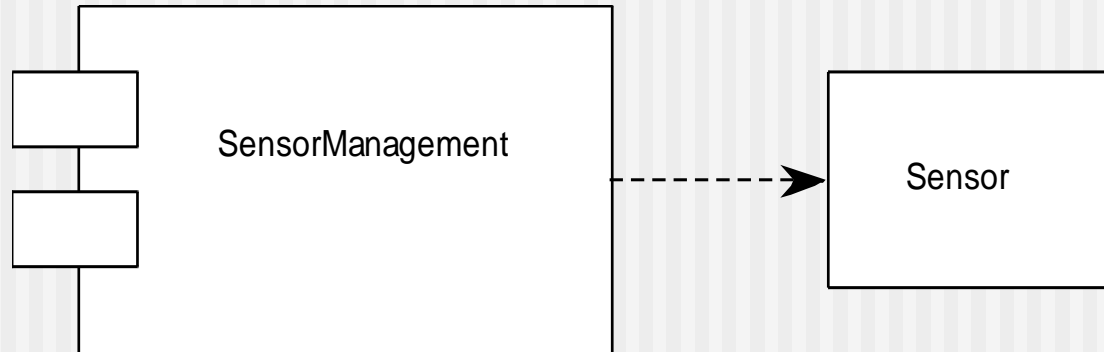- **Component elements**
- **Deployment elements**

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

Interface Elements

Component Elements

Deployment Elements