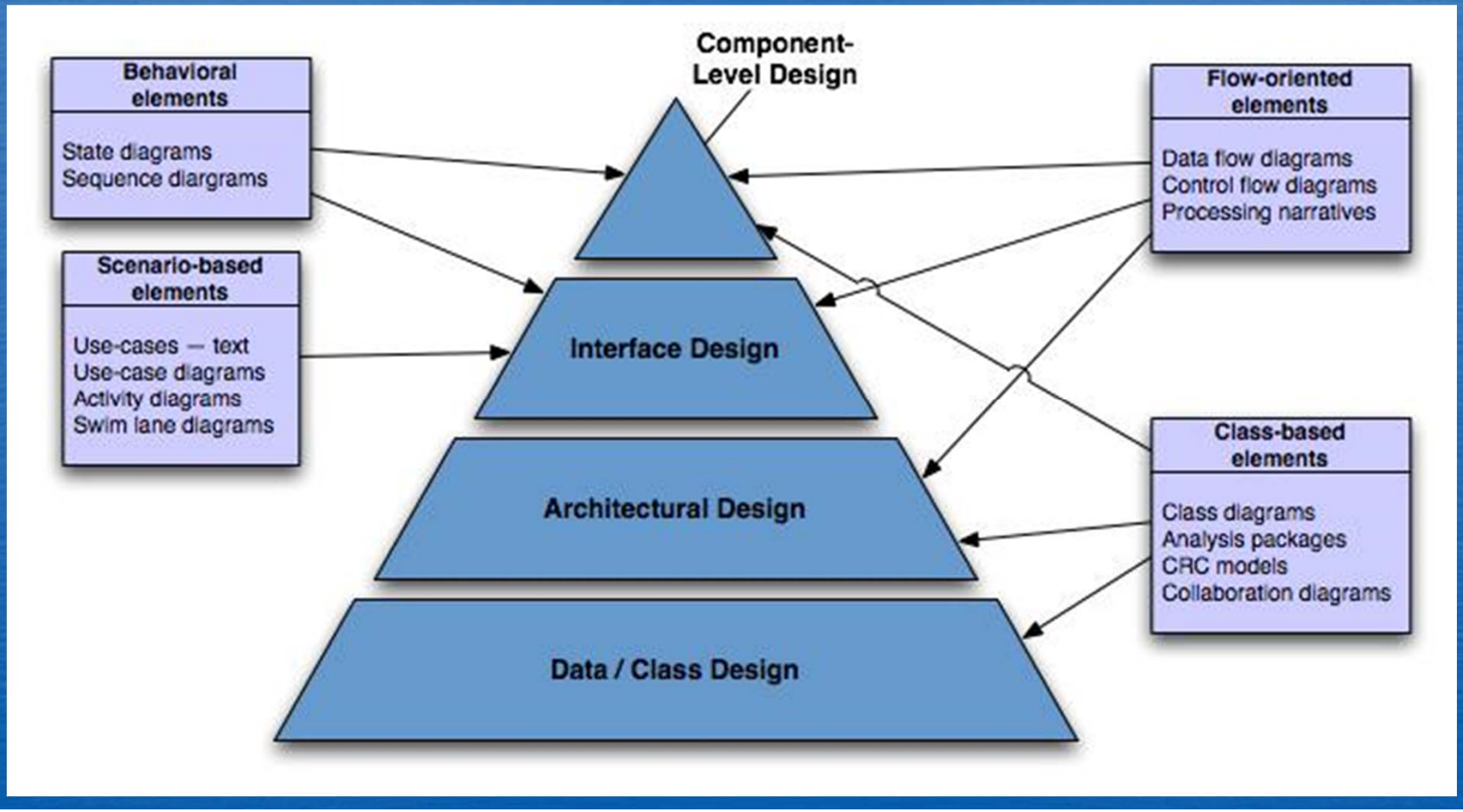
# Chapter 9
# Design Engineering

**Software Engineering: A Practitioner's Approach, 6th edition**
*by Roger S. Pressman*

# 1 概述

- 与分析模型不同：设计模型提供了软件的**数据结构**(data structure)、**体系结构**(architecture)、**接口**(interface)和**构件**(component)的细节，而这些都是实现系统所必须的。
- 设计可以采用很多不同的方法描绘软件。首先，设计必须体现系统或产品的体系结构；其次，为各类接口建模；最后，设计用于构建系统的软件构件。
- 软件设计是**建模活动**的最后一个软件工程活动，接着便要进入构造阶段(生成代码和测试)。

# Analysis → Design



# 2 设计过程和设计质量

- 软件设计是一个迭代的过程，通过设计过程，需求被变换为用于构建软件的“蓝图”【体系结构设计/概要设计、接口设计】。
- 随着设计迭代的推进，后续的精化导致更低抽象层次的设计表示【构件级设计/详细设计、接口设计】。
- 正式的技术评审（technical review）和设计走查（walkthrough）可用于评估设计演化的质量。

# 3 Design Concepts

abstraction — data, procedure

architecture — the overall structure of the software

patterns — “conveys the essence” of a proven design solution

modularity — compartmentalization of data and function[数据和功能划分]

information hiding — controlled interfaces

functional independence — high cohesion and low coupling

refinement — elaboration of detail for all abstractions

refactoring — improve design without effecting behavior

# 3.1 Abstraction et al.

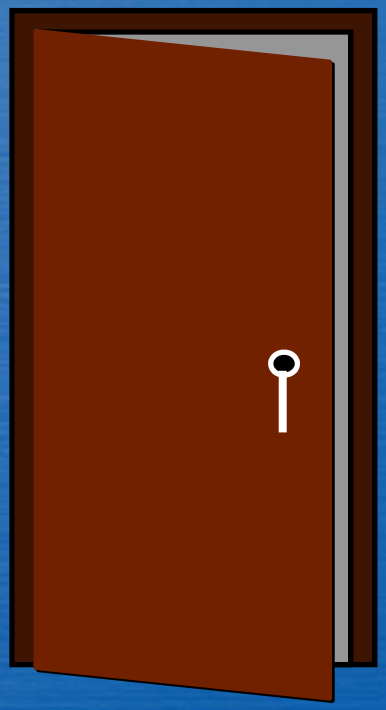
- Abstraction
  - process – extracting essential details
  - entity – a model or focused representation
- Information hiding
  - the suppression of inessential information
- Encapsulation
  - process – enclosing items in a container
  - entity – enclosure that holds the items

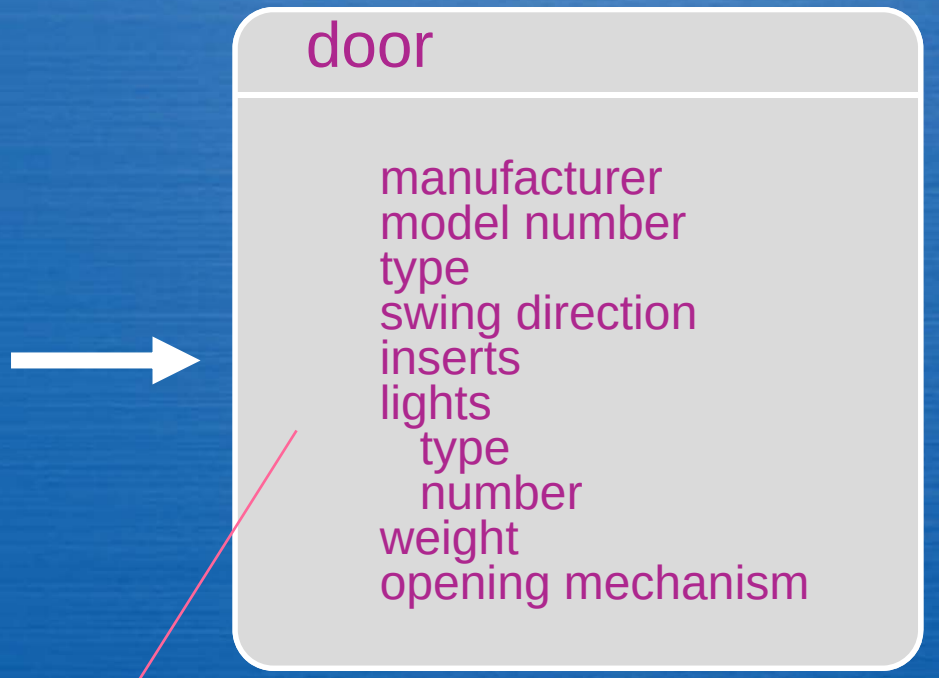
# Data Abstraction

实数
对象
类
构件
……

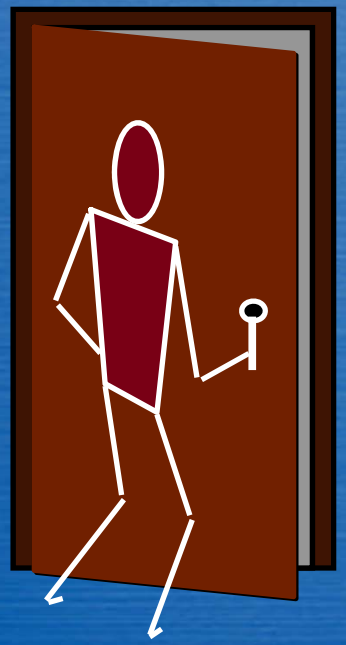
**数据抽象**是描述数据对象的冠名数据集合.

例如,"门"的 数据抽象将包含一组描述门的属性: 门的类型/转动方向/开门机构,重量和尺寸等





# Procedural Abstraction



open

details of enter algorithm

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

过程抽象是指具有明确和有限功能的指令序列.

例如,"开"隐含了一长串的过程性步骤: 走到门前、伸手并抓住把手、转动把手并拉门、离开打开的门等

# 3.2 Architecture

- “The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.” [软件的整体结构和这种结构为系统提供概念上完整性的方式][SHA95a]
- 简而言之, **体系结构**是程序构件（模块）的结构或组织、这些构件交互的形式、以及构件所用的数据的结构。

# Architecture…

◆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.

# 3.3 Patterns

- A pattern for software architecture describes a particular recurring design problem［某个再现的特定设计问题］that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

# Some Kinds of Patterns

- An architectural pattern expresses **a fundamental structural organization schema for software systems**. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- A design pattern provides **a scheme for refining the subsystems** or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.
- An idiom is a low-level pattern **specific to a programming language**. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

# Design 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**

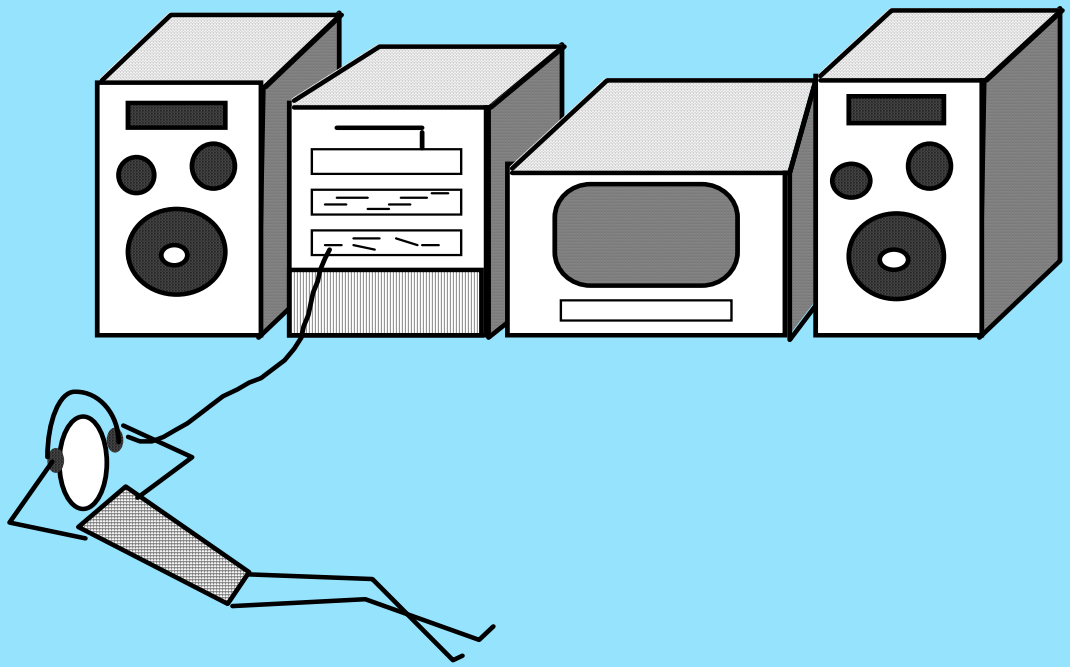
**Related patterns—cross-references related design patterns**

# Design Patterns

- The **best designers** in any field have an uncanny ability [离奇的能力] to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution
- A description of a design pattern may also consider a set of design forces.
  - Design forces [设计规范] describe non-functional requirements (e.g., ease of maintainability, portability) associated the software for which the pattern is to be applied.
- The pattern characteristics (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a variety of problems.

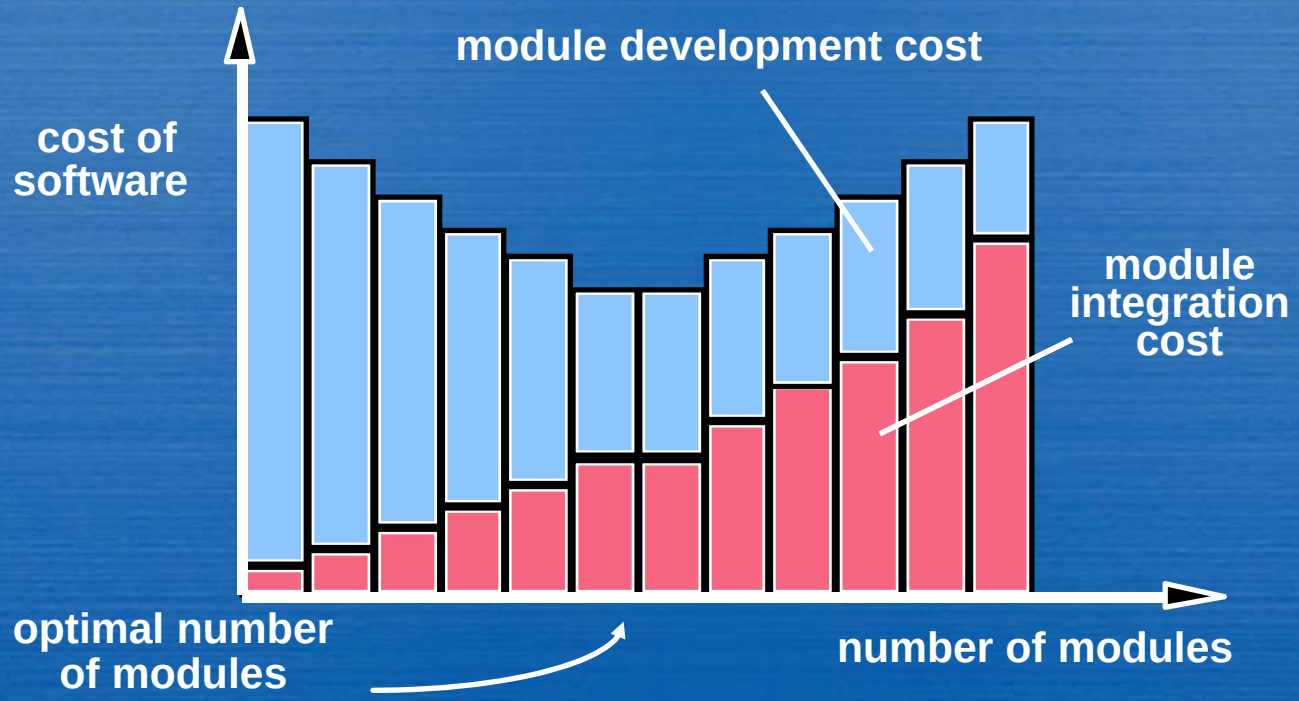
# 3.4 Modular Design

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

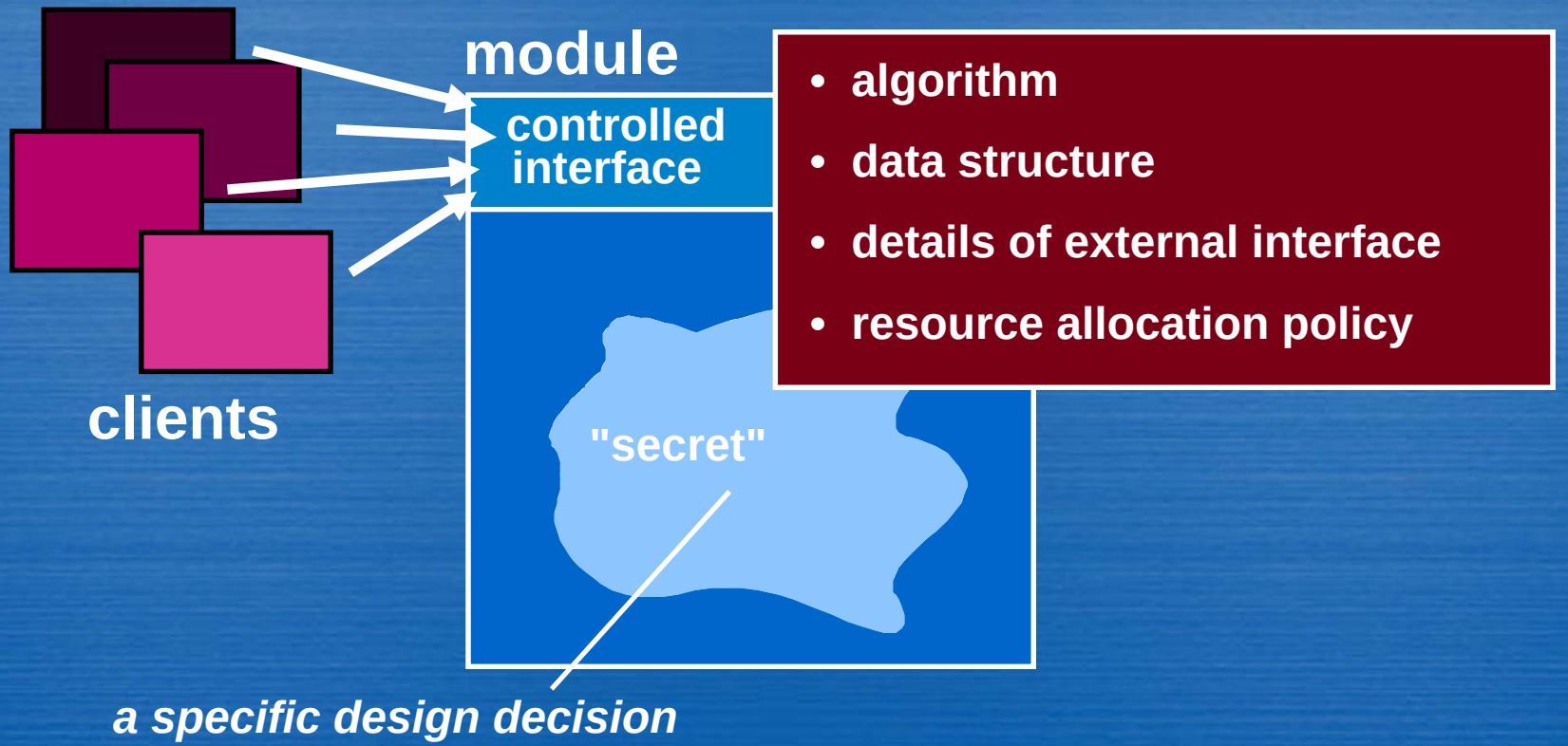


# Modularity: Trade-offs

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



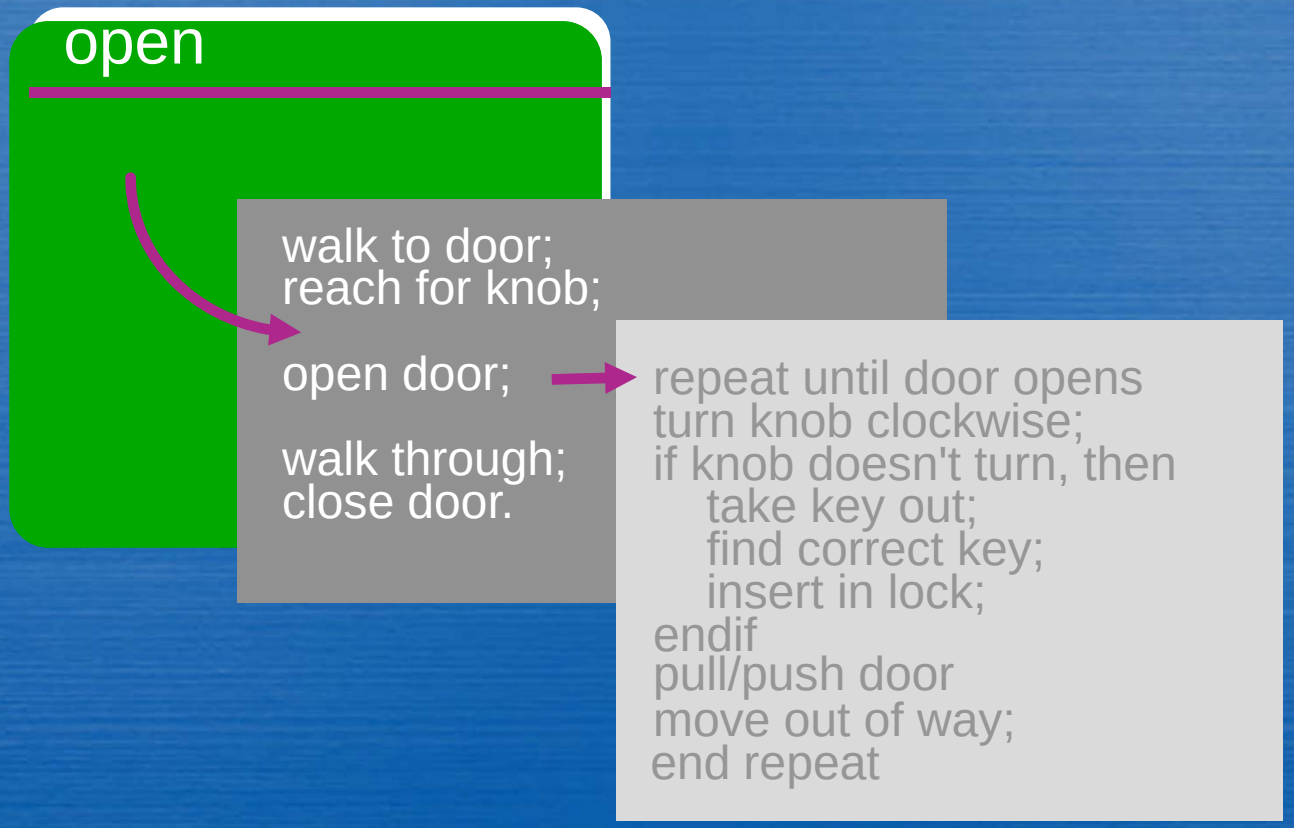
# 3.5 Information Hiding



# 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

# 3.6 Stepwise Refinement



# 3.7 Functional Independence

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

# 3.8 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.

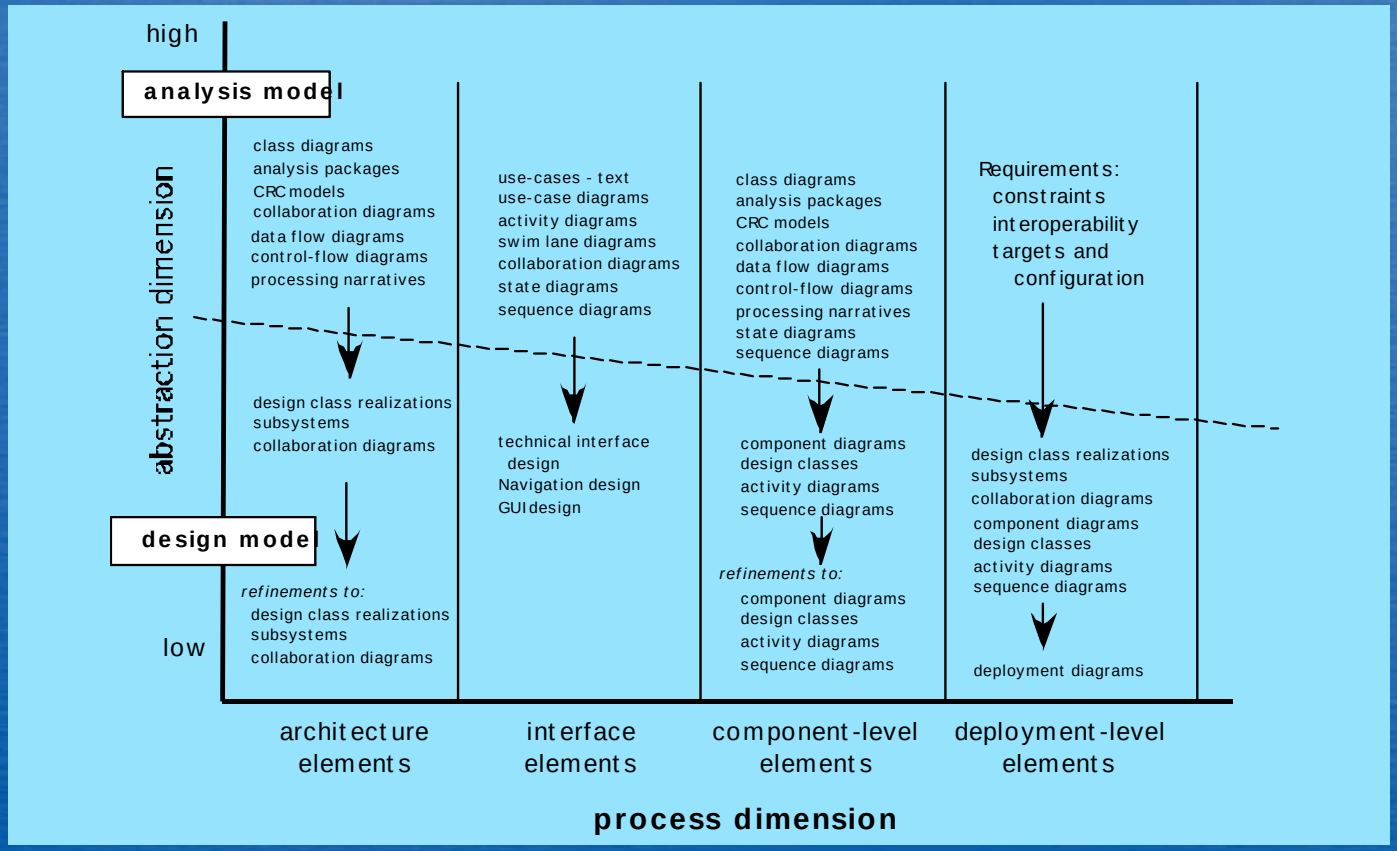
# 3.9 Design Classes

- User interface classes – define abstractions necessary for HCI.
- Business domain classes – refinements of analysis classes.
- Process classes – lower-level business abstractions that manage business domain classes.
- Persistent classes – data stores (databases) that persist beyond execution of the software.
- System classes – management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

# Well-formed Design Class

- Complete and sufficient – class should be a complete and sufficient encapsulation of reasonable attributes and methods.
- Primitiveness[原始性] – each method should be focused on one thing.
- High cohesion – class should be focused on one kind of thing.
- Low coupling – collaboration should be kept to an acceptable minimum.

# 4 The Design Model




**过程维度**表示设计模型的演化,设计工作作为软件过程的一部分被执行.

**抽象维度**表示过程的详细级别,分析模型的每个元素转化为一个等价的设计,然后迭代精化.

# Design Model Elements

- Data elements
  - Architectural level → databases and files
  - Component level → data structures
- Architectural elements
  - An architectural model is derived from:
    - ① Application domain[应用领域信息]
    - ② Analysis model[特定分析模型元素]
    - ③ Available styles and patterns[体系结构风格和模式]
- Interface elements
  - There are three parts to the interface design element:
    - ① The user interface (UI)
    - ② Interfaces to external systems
    - ③ Interfaces to components within the application
- Component elements
- Deployment elements

# Interface Elements

软件的**接口设计**相当于一组房屋的门、窗和外部设施的详细绘图。

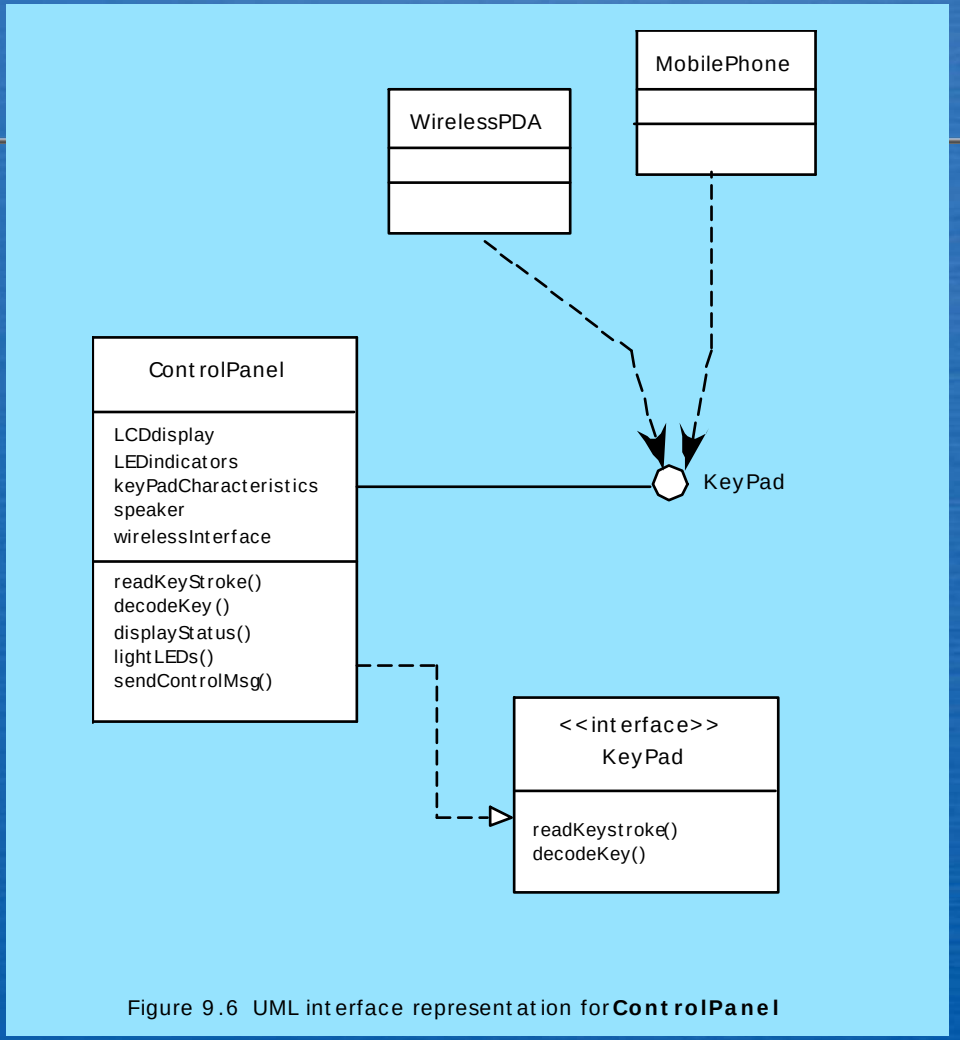
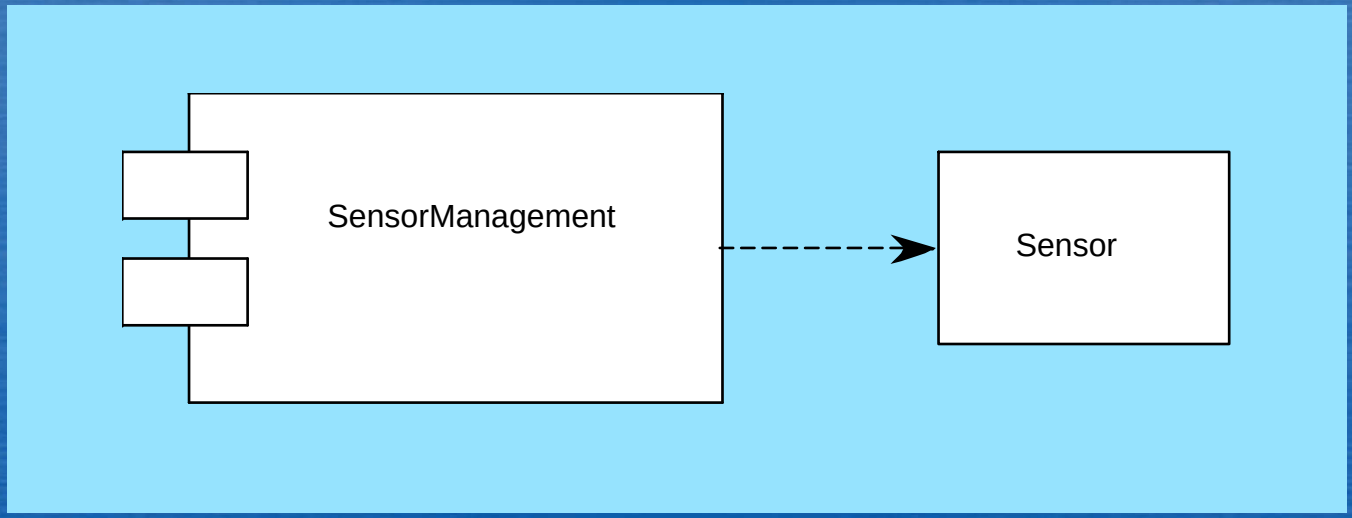


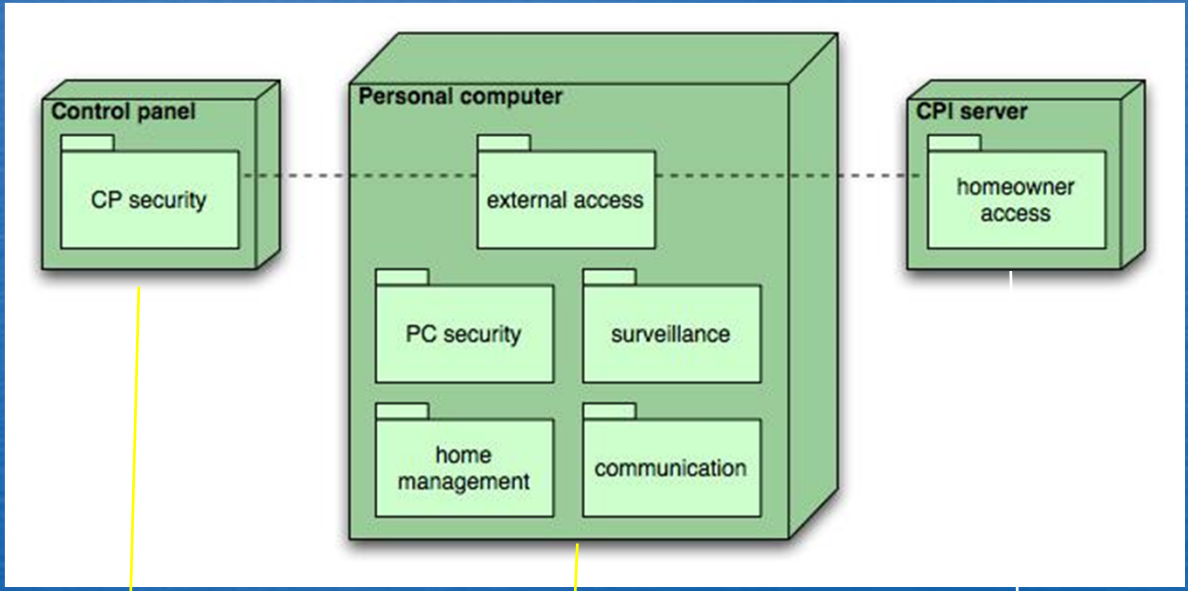

Figure 9.6 UML interface representation for **ControlPanel**

# Component Elements

软件的**构件级设计**完全类似于某个房屋中每个房间的一组详细绘图。



# Deployment Diagram




**部署级设计元素**指明软件功能和子系统将如何在支持软件的物理计算环境内分布。

SafeHome控制面板

基于住宅的PC

CPI公司的服务器

# 5 Design Principles

① The design process **should not suffer from** 'tunnel vision[视野狭隘].'
② The design **should be traceable to** the analysis model.
③ The design **should not reinvent** the wheel.
④ The design **should "minimize the intellectual distance**[理解差异]" between the software and the problem as it exists in the real world.
⑤ The design **should exhibit** uniformity and integration.
⑥ The design **should be structured to** accommodate change.
⑦ The design **should be structured to** degrade gently, even when aberrant data, events, or operating conditions are encountered.
⑧ **Design is not coding, coding is not design.**
⑨ The design **should be assessed** for quality as it is being created, not after the fact.
⑩ The design **should be reviewed to** minimize conceptual (semantic) errors.

# 6 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.

# 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.