Software Engineering

Chapter-8
Design Concepts

Software Engineering: A Practitioner's Approach, 7th edition
Originated by Roger S. Pressman

Software School, Fudan University
Spring Semester, 2016

Software Design

- Design: it is where you stand with a foot in two worlds
  - The world of people and human purposes (problem)
  - The world of technology (solution)
- Bring the two worlds together in design
- Why need design before implementation?
  - Produces a general picture of the system to be implemented
    - As a detailed plan for implementation that can meet the requirements
    - To enable early-stage estimation of cost, time, risk and quality
  - Complex system: decompose and integration
    - To (hierarchically) decompose the system into some parts that can be done one by one and allocated to different developers
    - Define a set of shared conventions and standards among different parts of the system and among different developers to ensure the integration

DECOMPOSE AND INTEGRATION

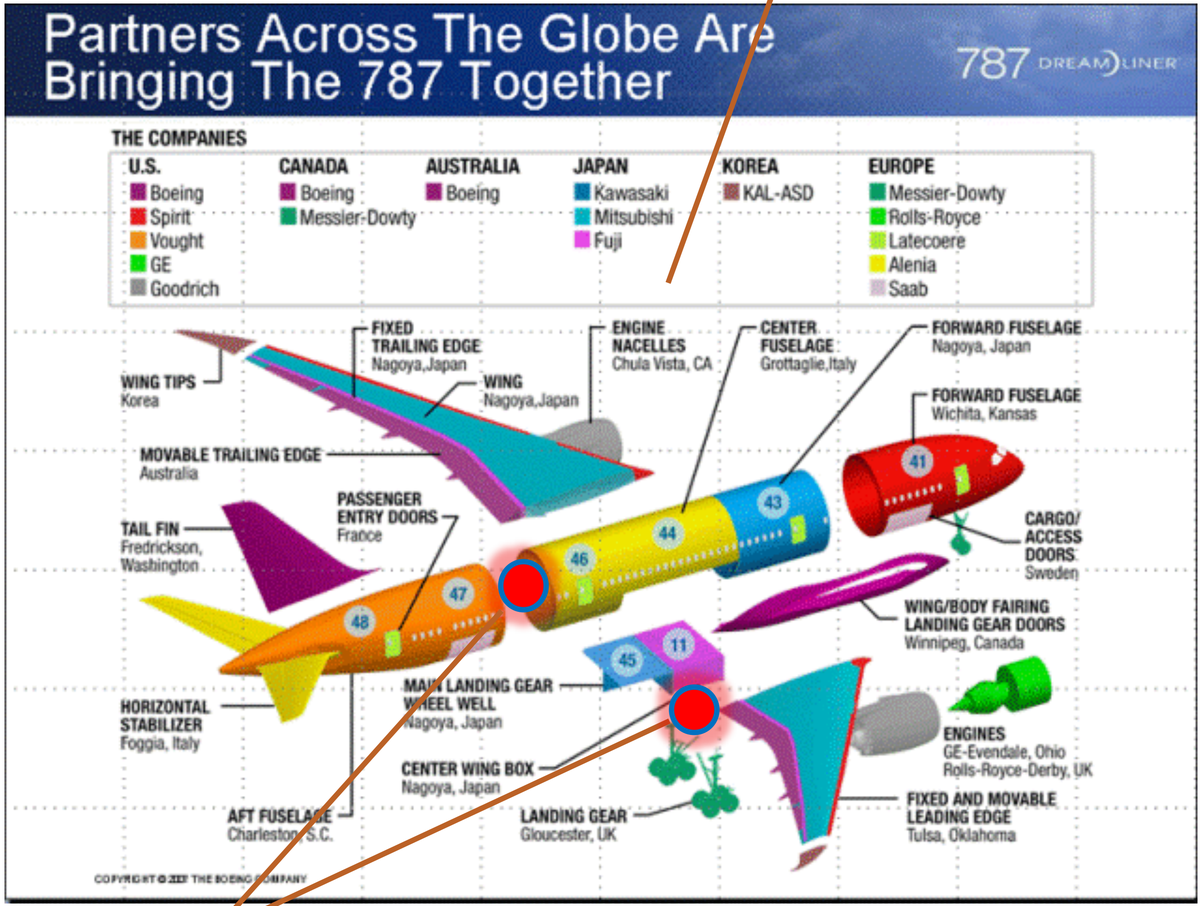
Decomposition

Convention

Constraint

Standard

……



Partners Across The Globe Are Bringing The 787 Together

787 DREAMLINER

THE COMPANIES

U.S.	CANADA	AUSTRALIA	JAPAN	KOREA	EUROPE
■ Boeing	■ Boeing	■ Boeing	■ Kawasaki	■ KAL-ASD	■ Messier-Dowty
■ Spirit	■ Messier-Dowty		■ Mitsubishi		■ Rolls-Royce
■ Vought			■ Fuji		■ Latecoere
■ GE					■ Alenia
■ Goodrich					■ Saab

Interface

Most of the overall quality is determined by the design

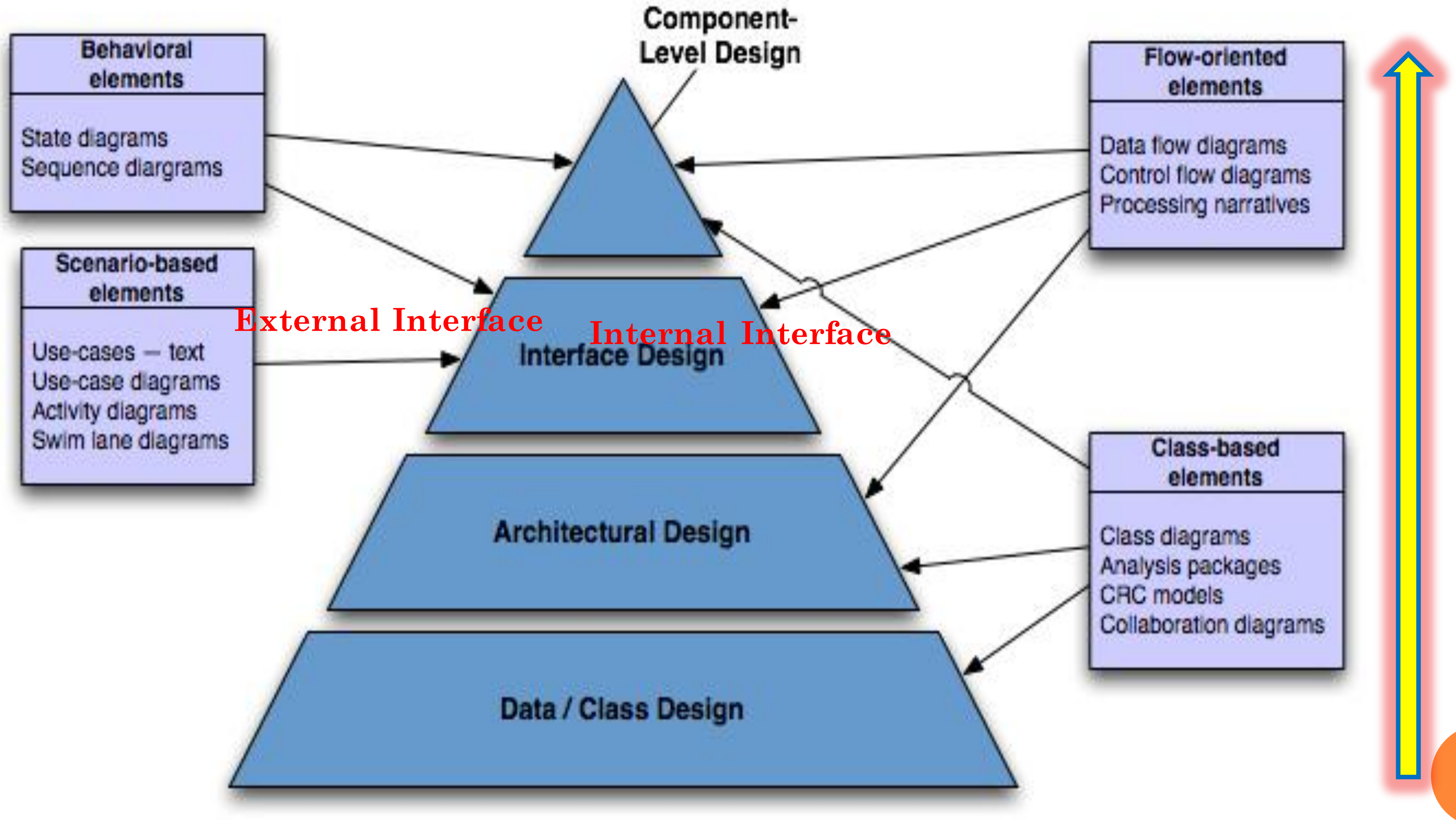
Goals of Design Process

- Meet the requirements
  - To implement all of the explicit requirements
  - To accommodate all of the implicit requirements

承上启下

- Guide for coding, testing and maintaining
  - To be readable and understandable for those who *use* the design
- A complete picture from an implementation perspective
  - To address data, functional and behavioral domains

From Analysis to Design



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 ATTRIBUTES (FURPS): EXTERNAL QUALITY

Quality: the main target that forces software design

- Functionality (功能性)
  - capability; generality of functions; overall security
- Usability (易用性)
  - considering human factors
- Reliability (可靠性)
  - failure; recoverability from failure; predictability
- Performance (性能)
  - processing speed, response time, resources needed
- Supportability (可支持性)
  - maintainability; testability; compatibility; configurability

DESIGN CONCEPTS-1

- abstraction — data, procedure, control
- architecture — the overall structure of the software
- patterns — “conveys the essence” of a proven design solution
- modularity — compartmentalization of data and function

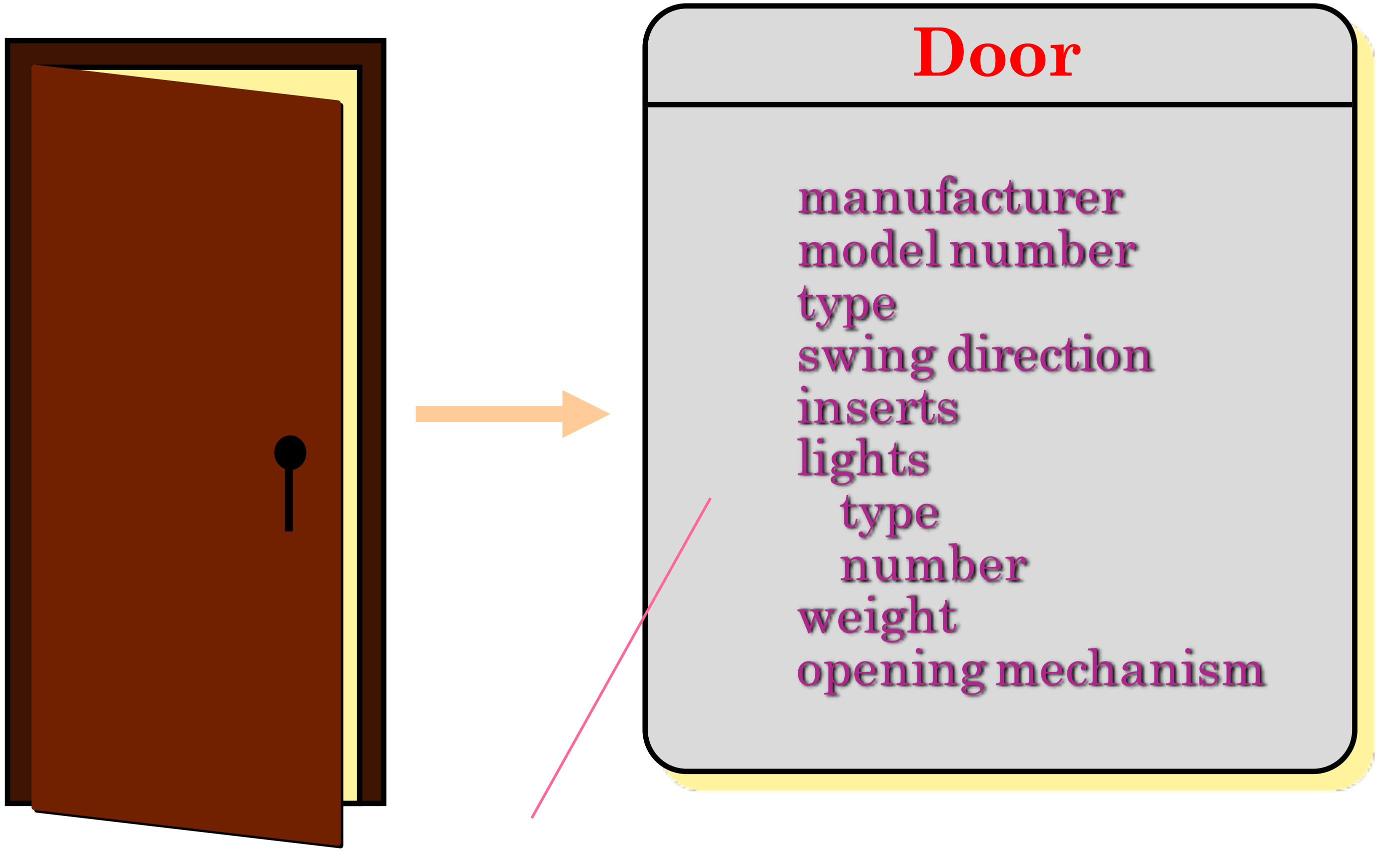
DESIGN CONCEPTS-2

- information hiding — controlled interfaces
- functional independence — modules with single-minded function (assessment: cohesion and coupling)
- refinement — elaboration of detail for all abstractions
- refactoring — improve design without effecting behavior

Design Principles

- 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

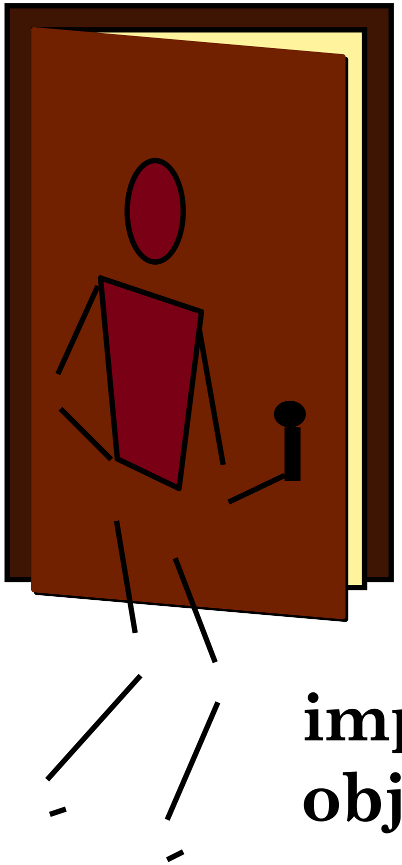


Door

- 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

- the components of a system (e.g., modules, objects, filters) and
- the manner in which those components are packaged and interact, e.g. objects are packaged to encapsulate both data and processing

Extra-functional properties

- how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems

- draw upon repeatable patterns that are commonly encountered in the design of families of similar systems.
- reuse architectural building blocks.

USING PATTERNS

- A particular recurring design problem
  - e.g. the requirement of "high-throughput processing","easy to deploy" and "easy to upgrade"
- In specific design contexts
  - e.g. a lot of PC clients connected by Internet or LAN
- A well-proven generic scheme for its solution
  - Constituent components, e.g. Browser and Server part
  - Their responsibilities and relationships, e.g. Browser for user interaction and Server for computation
  - The ways in which they collaborate, e.g. interaction by http

Using Patterns (cont.)

- The best designers in any field have an uncanny ability 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* 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.

Patterns at Different Levels

Architectural pattern

- expresses a fundamental structural organization schema for software systems
- predefined subsystems, their responsibilities, and rules and guidelines for organizing their relationships

Design pattern

- provides a scheme for refining the subsystems or components of a software system, or the relationships between them.
- describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.

Idiom

- a low-level pattern, specific to a programming language
- implement particular aspects of components or the relationships using the features of the given language.
- e.g. using conditional compilation or reflection to achieve configurability and flexibility

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

GOOD DESIGN?

- Easy to understand
  - Design structure well aligned with domain concepts
- Easy to develop and integrate
  - Each module focuses on limited things
  - Different modules interact as little as possible
- Easy to extend and modify
  - Extend without modifying existing modules
  - Localized influences with little global influences
- Easy to reuse
  - the whole design
  - individual modules

Separation of Concern (SOC)

- 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

SEPARATION OF CONCERN (CONT.)

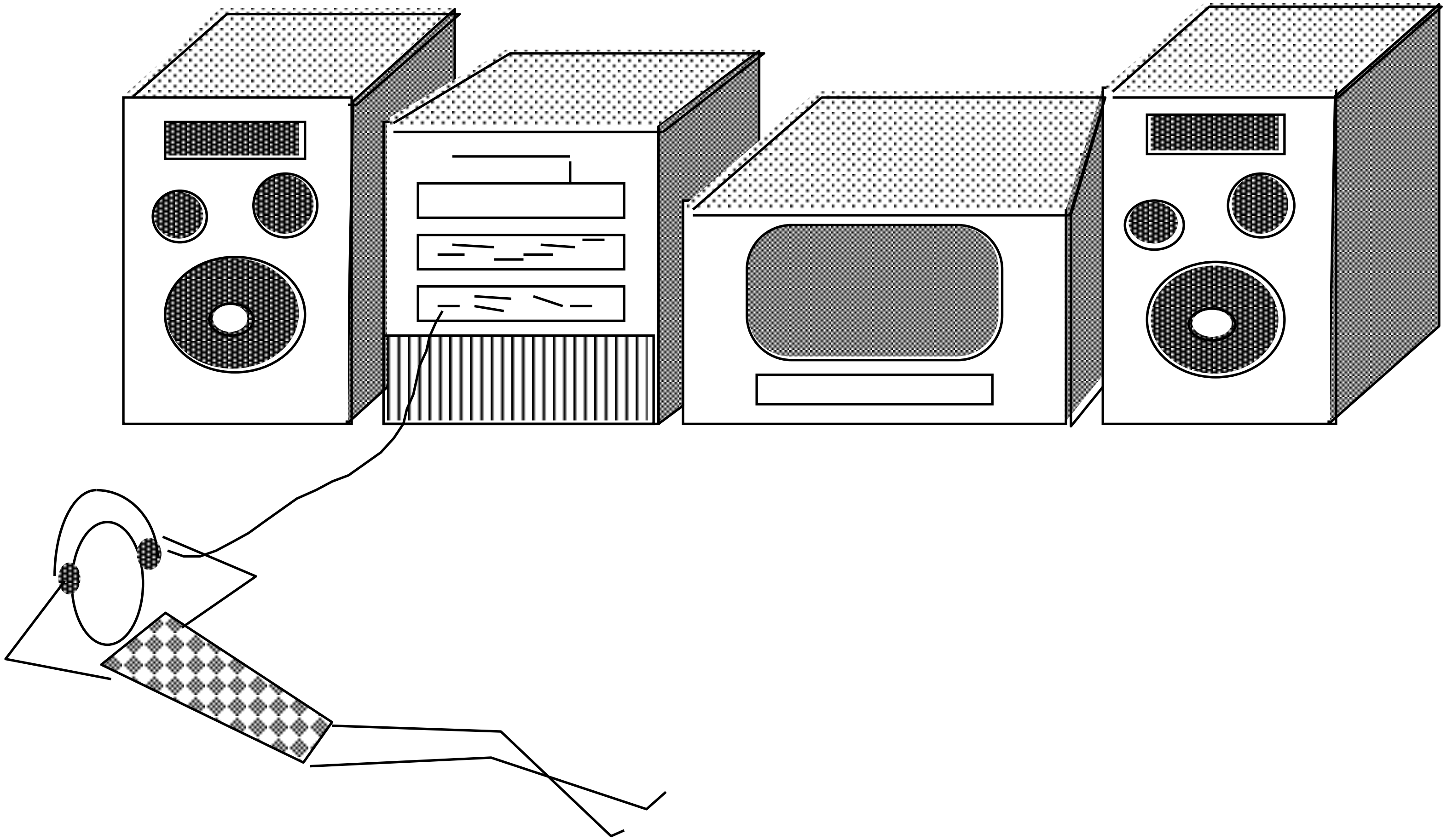
- separating a module into distinct features that overlap in functionality as little as possible
  - achieved through modularity and encapsulation (or “transparency” of operation)
  - with the help of information hiding
  - layered designs in information systems are also often based on separation of concerns
    - e.g., presentation layer, business logic layer, data access layer, database layer)

Separation of Concern (Cont.)

- Also an important design principle in many other areas, e.g.
  - urban planning, architecture and information design
  - Examples:
    - using corridors to connect rooms rather than having rooms open directly into each other
    - keeping the stove on one circuit and the lights on another
- Benefit: make it easier to understand, design and manage complex interdependent systems
  - functions can be optimized independently of other functions
  - failure of one function does not cause other functions to fail

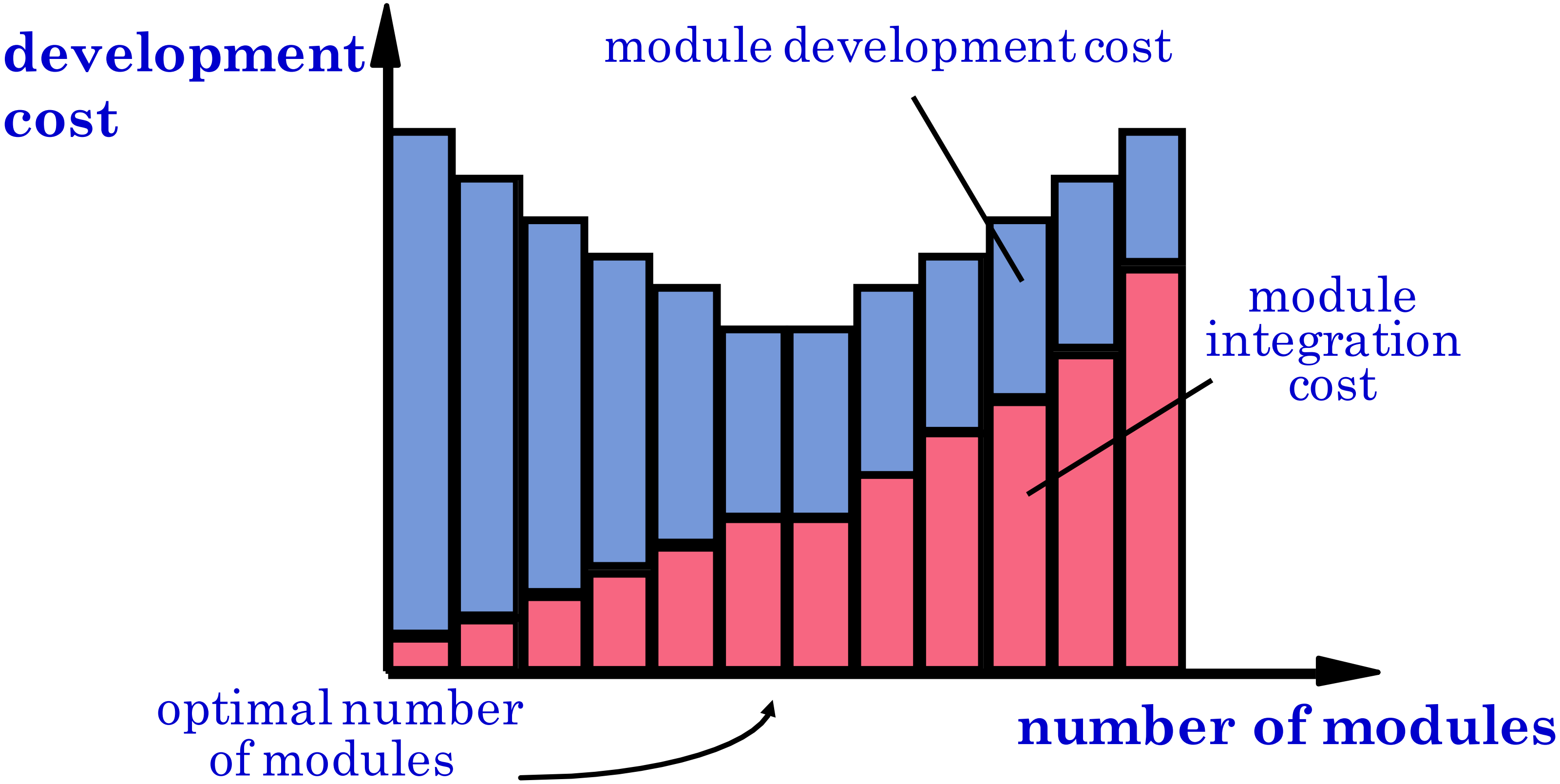
MODULAR DESIGN

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

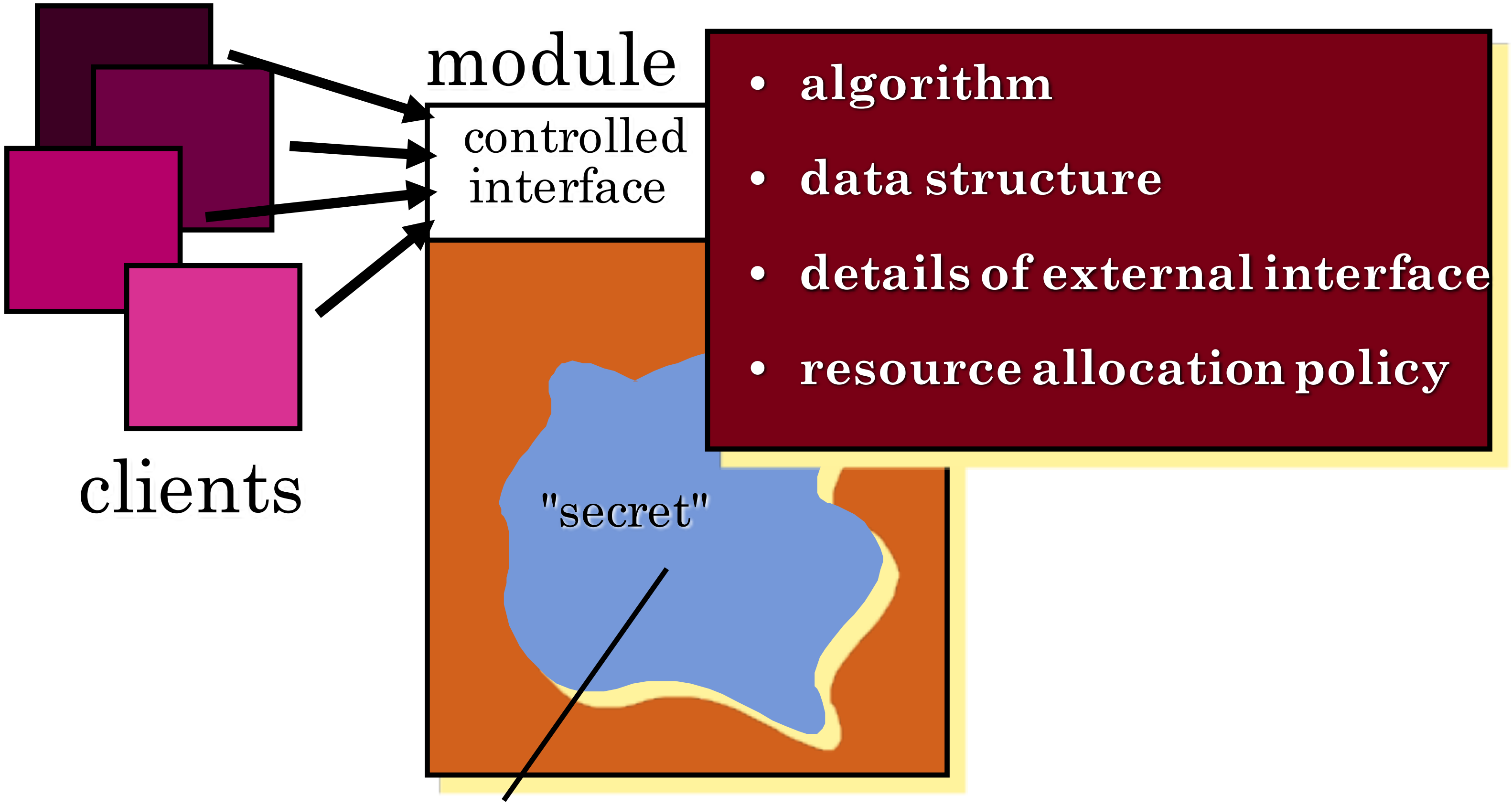


MODULARITY: TRADE-OFFS

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



Information Hiding



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

a specific design decision

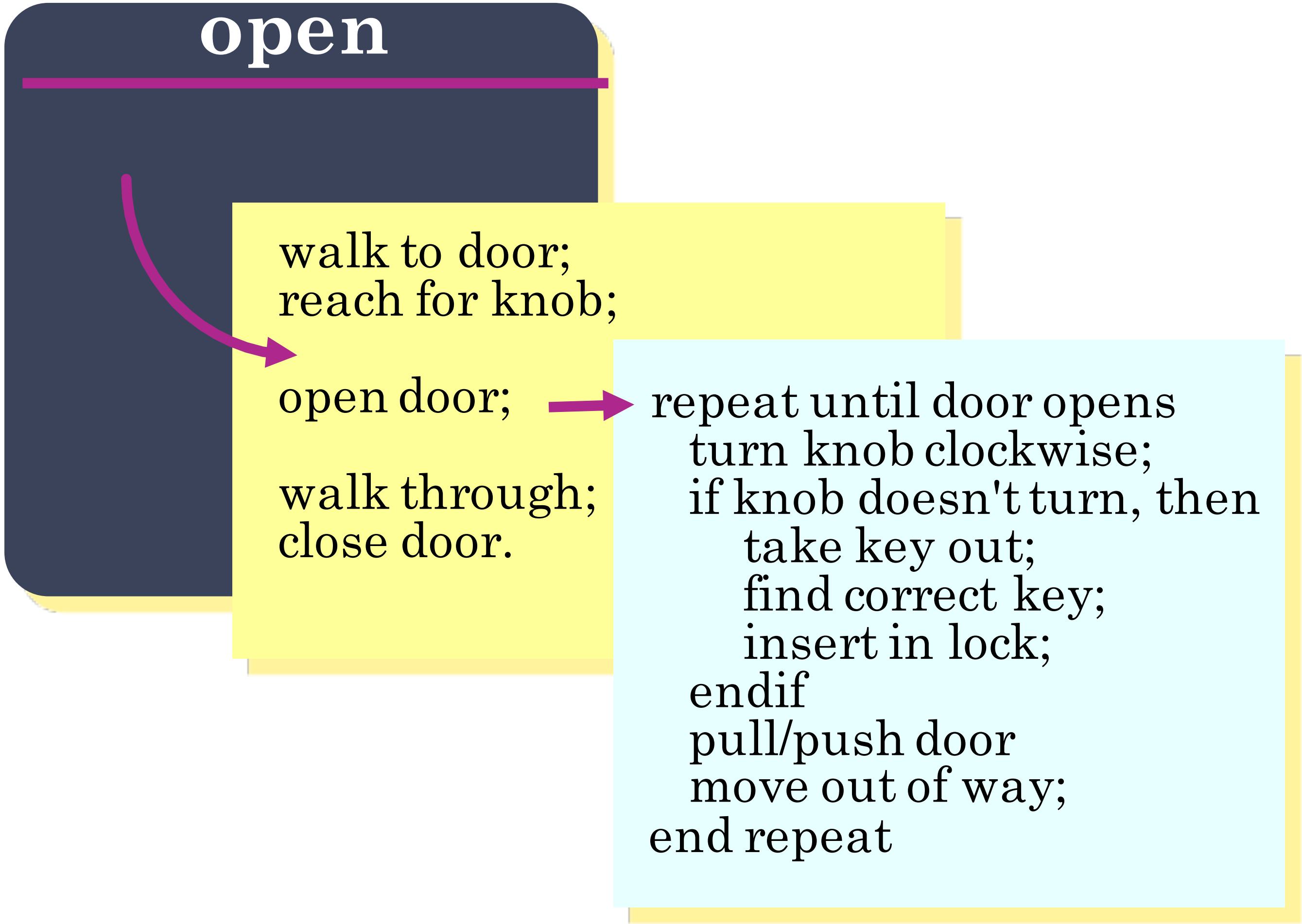
WHY INFORMATION HIDING?

- 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 good encapsulation and modulization, thus results in higher quality software

Stepwise Refinement

- Door: Open



open

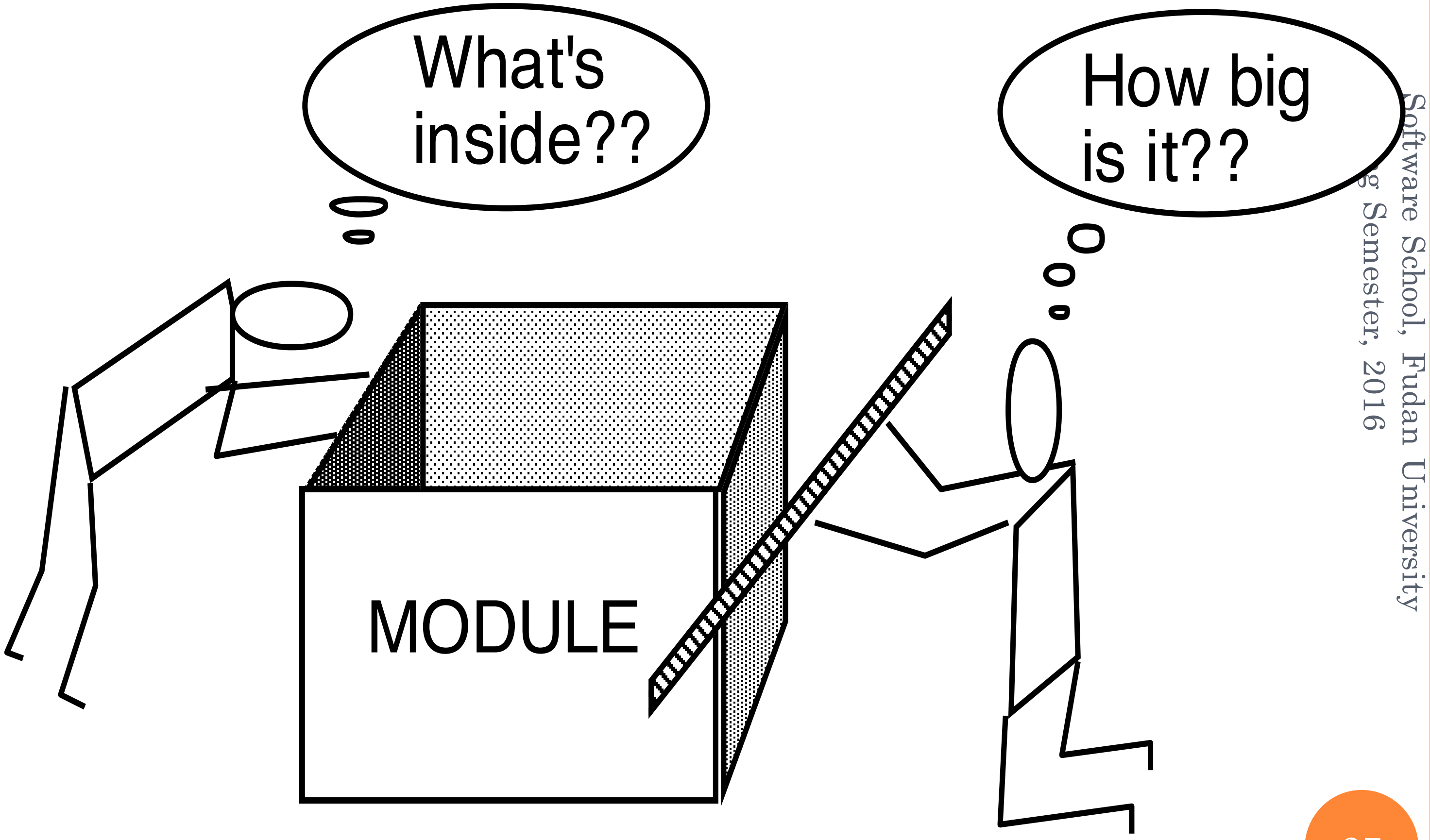
```
walk to door;
reach for knob;

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



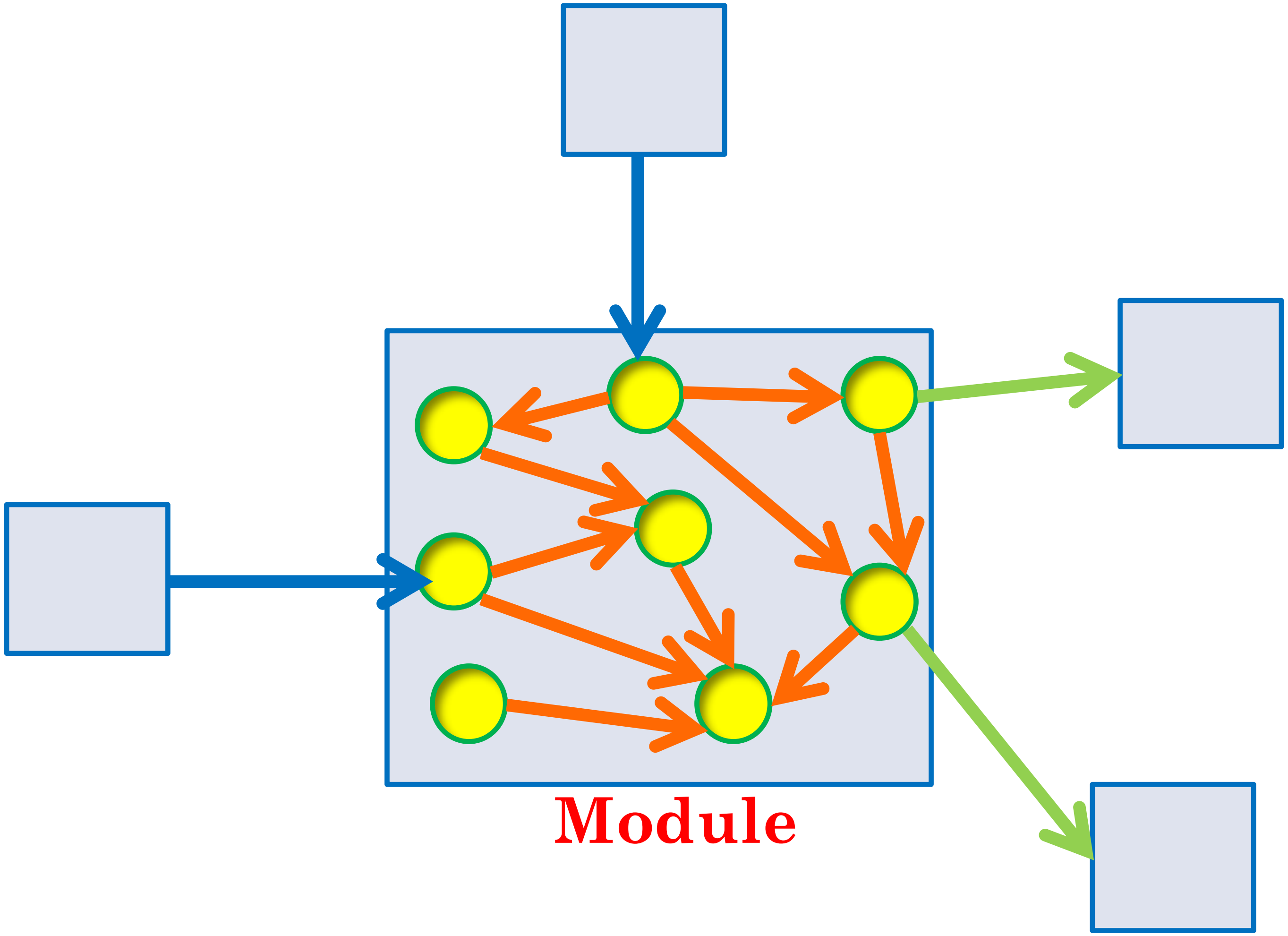
Functional Independence

- Developing modules with “single-minded” function and an “aversion” to excessive interaction with other modules
- Assessment criteria

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

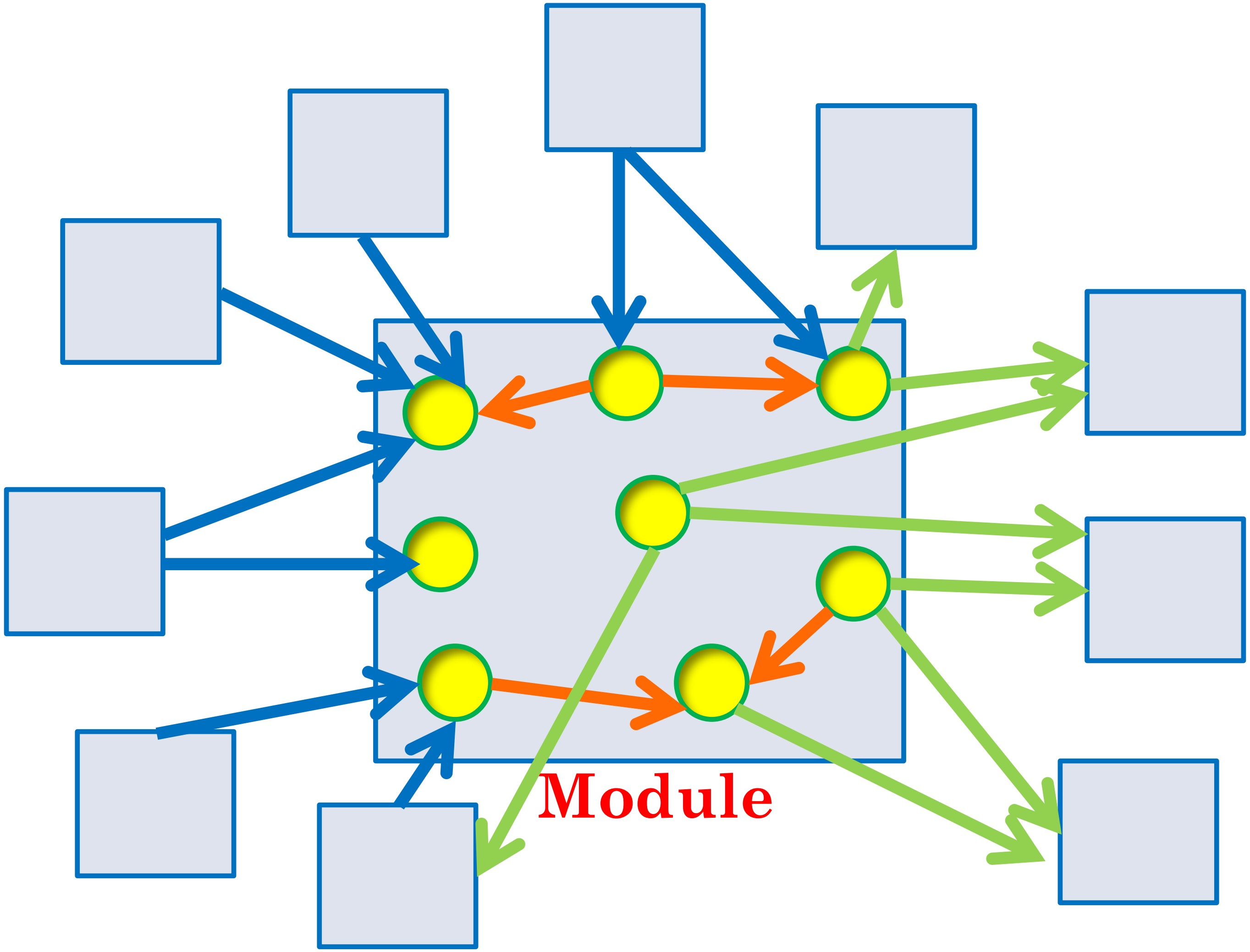
COUPLING - the degree to which a module is "connected" to other modules in the system.

Design with Good Functional Independence



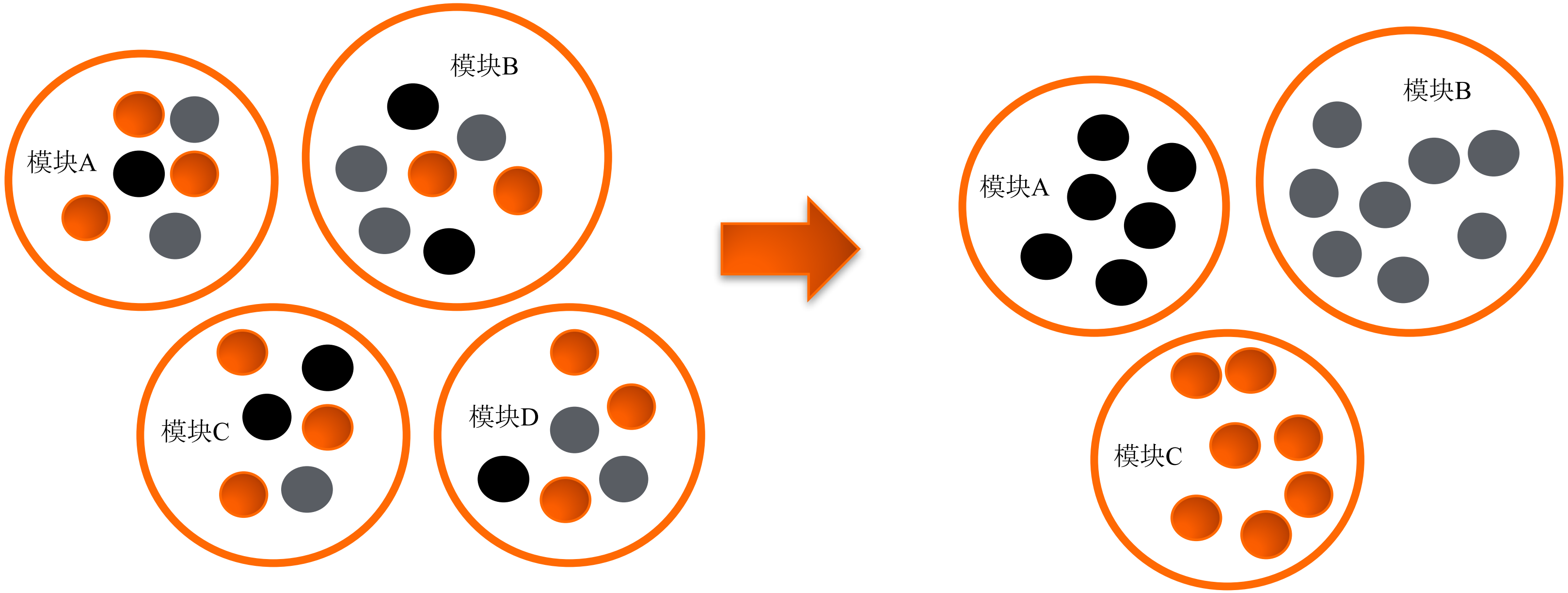
High Cohesion and Low Coupling

Design with Bad Functional Independence



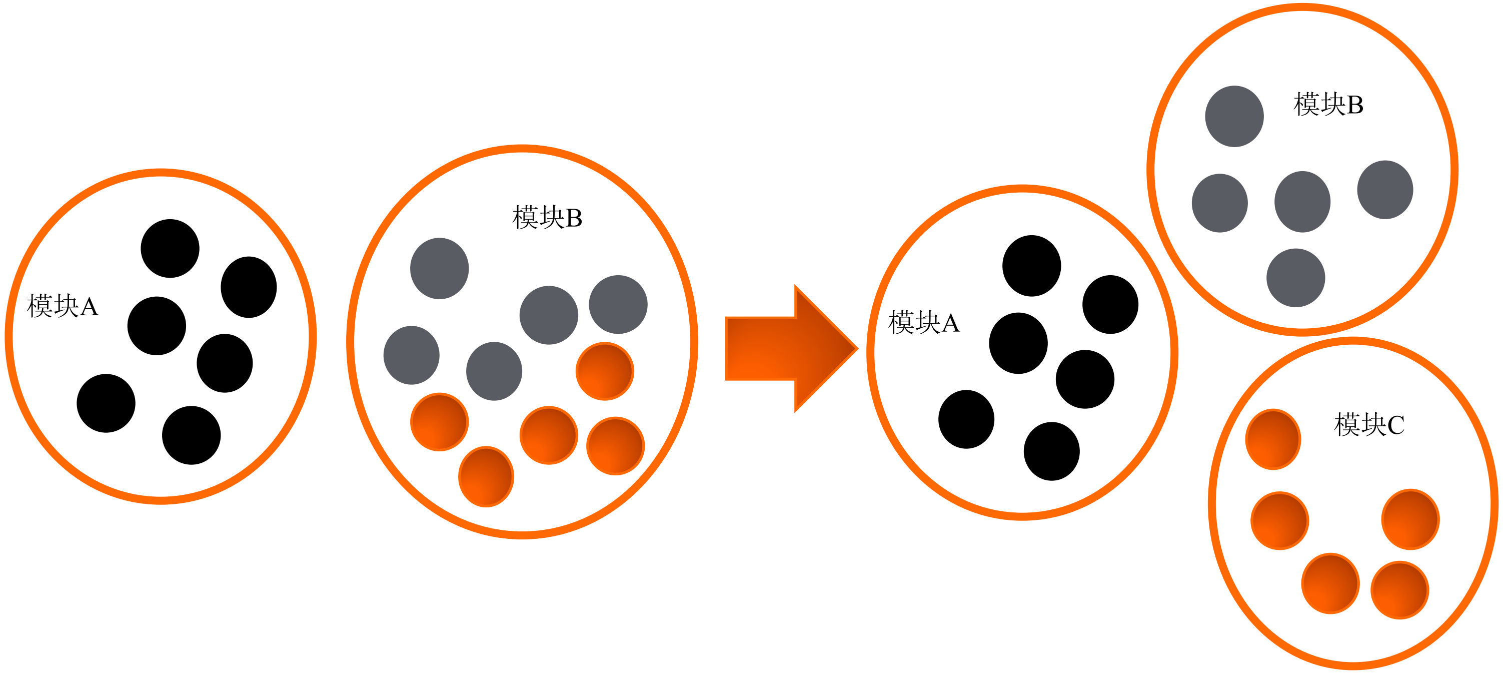
Low Cohesion and High Coupling

HIGH COHESION-1



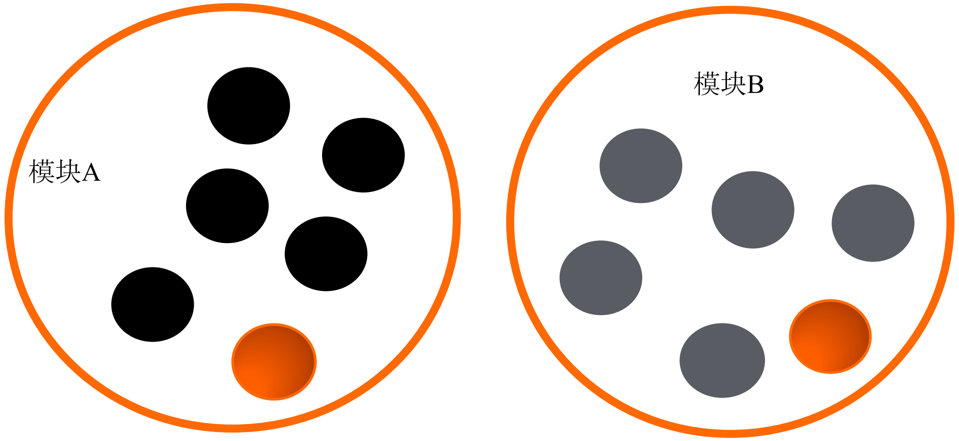
Put closely related things together

HIGH COHESION-2



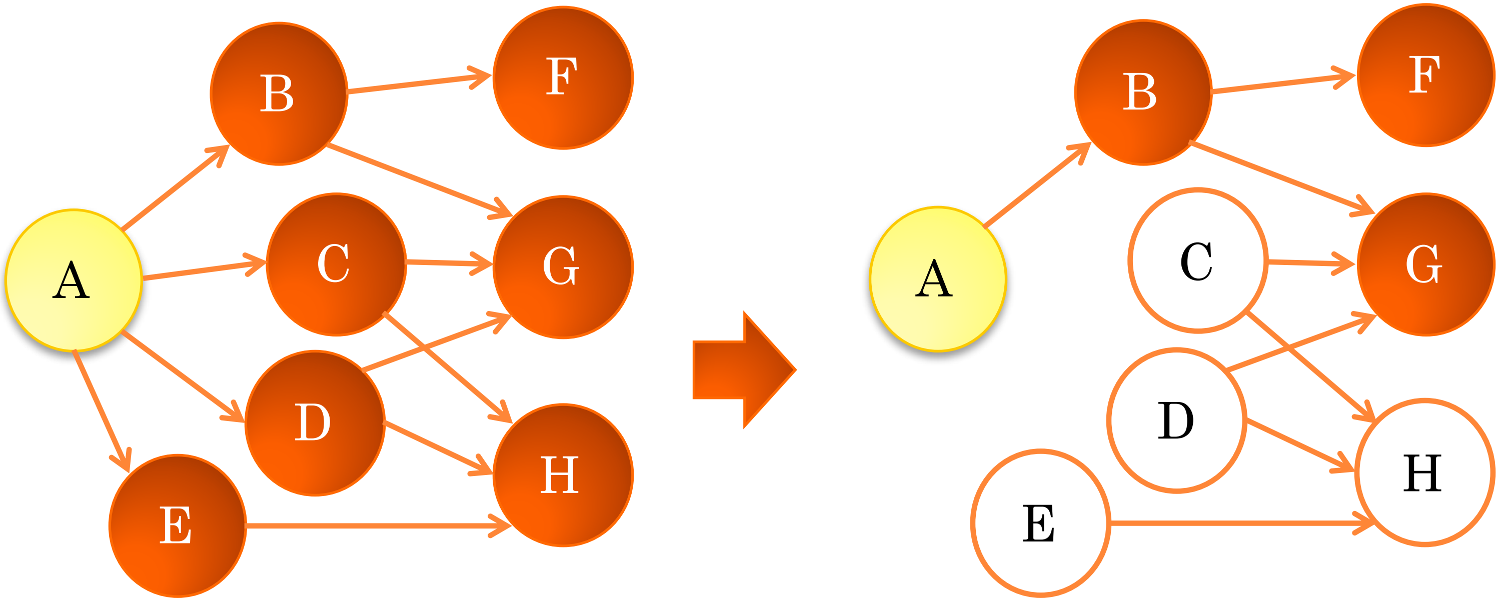
Only put closely related things together

Duplication → Low Cohesion



Eliminate Duplication (Clones)!

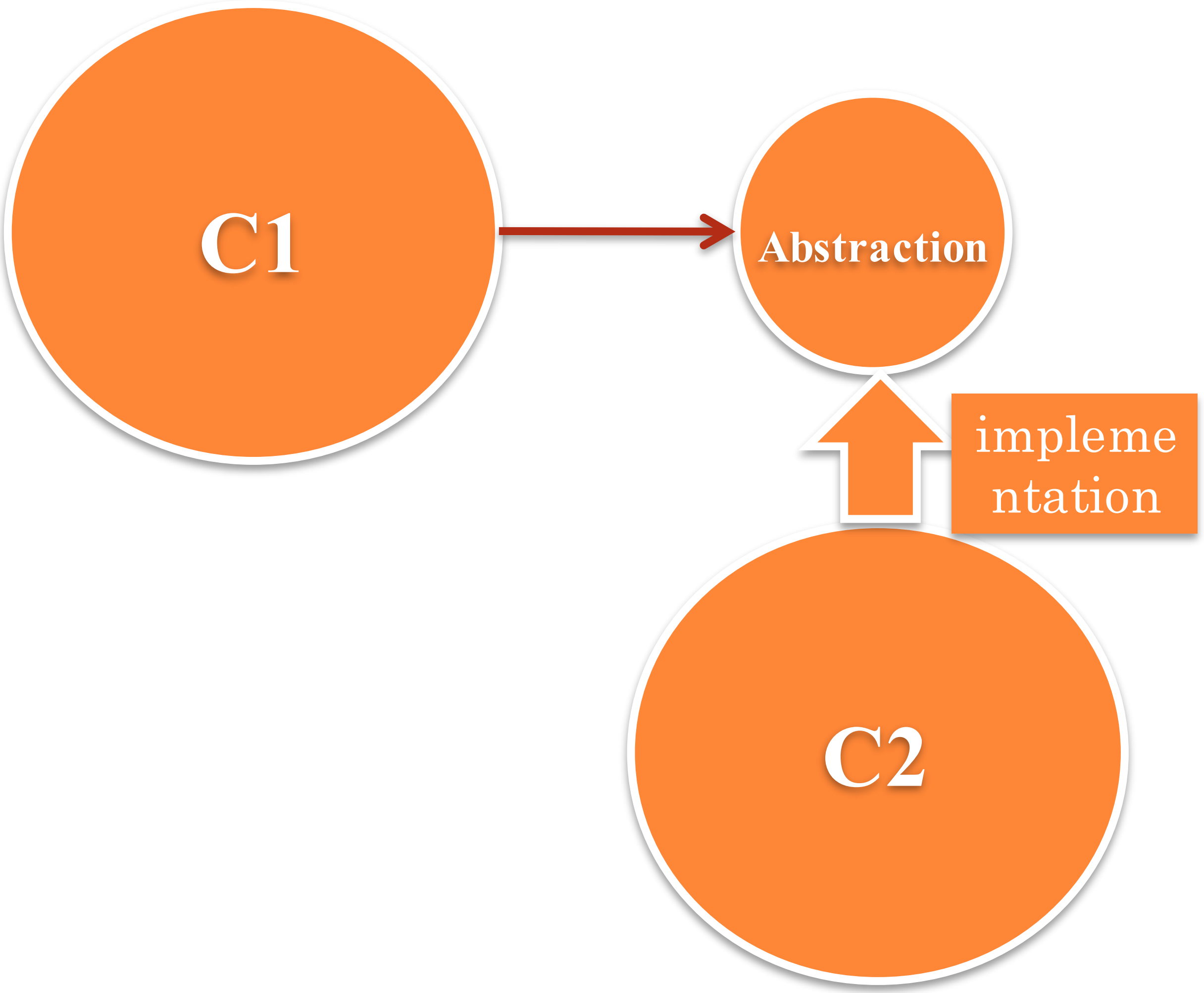
High Coupling → Hard to Reuse



LOWER COUPLING

- Less Knowledge Principle
- Depend on stable (abstract) things

Depend on Abstraction



C1

Abstraction

implementation

C2

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

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.

By Fowler [FOW99]

- Why refactoring
  - External quality and internal quality of a software may deviate greatly
  - Persistent changes and evolution will make the design quality degrade
  - Internal quality is essential for future maintenance

REFACTORING (CONT.)

- When software is refactored, the existing design is examined for …
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - any other design failure that can be corrected to yield a better design

REFACTORING (CONT.)

1. Duplicate Code
2. Long Method
3. Large class
4. Long Parameter List
5. Divergent Change
6. Shotgun Surgery
7. Feature Envy
8. Data Clumps
9. Primitive Obsession
10. Switch Statements
11. Parallel Inheritance Hierachies
12. Lazy Class
13. Speculative Generality
14. Temporary Field
15. Message Chains
16. Middle Man
17. Inappropriate Intimacy
18. Alternative Classes with Different Interfaces
19. Incomplete Library Class
20. Data Class
21. Refused Bequests
22. Comments

Eliminate bad smells in code!

Design Classes

- User interface classes
  - define abstractions necessary for HCI, e.g. an input form
- Business domain classes
  - refinements of analysis classes, e.g. Course, Student…
- Process classes
  - lower-level business abstractions that manage business domain classes, e.g. Registration Management
- Persistent classes
  - data stores (databases) that persist beyond execution of the software, e.g. entity classes with OR mapping
- System classes
  - management and control functions that enable the system to operate and communicate within its computing environment and with the outside world, e.g. Network Communication

Well-Formed Design Class

- Complete and sufficient – class should be a complete and sufficient encapsulation of reasonable attributes and methods
- High cohesion – class has a small, focused set of responsibilities
- Primitiveness – each method should be focused on one thing
- Low coupling – collaboration should be kept to an acceptable minimum
  - Don't talk to strangers!
    [*Law of Demeter*]
    [*Least Knowledge*]

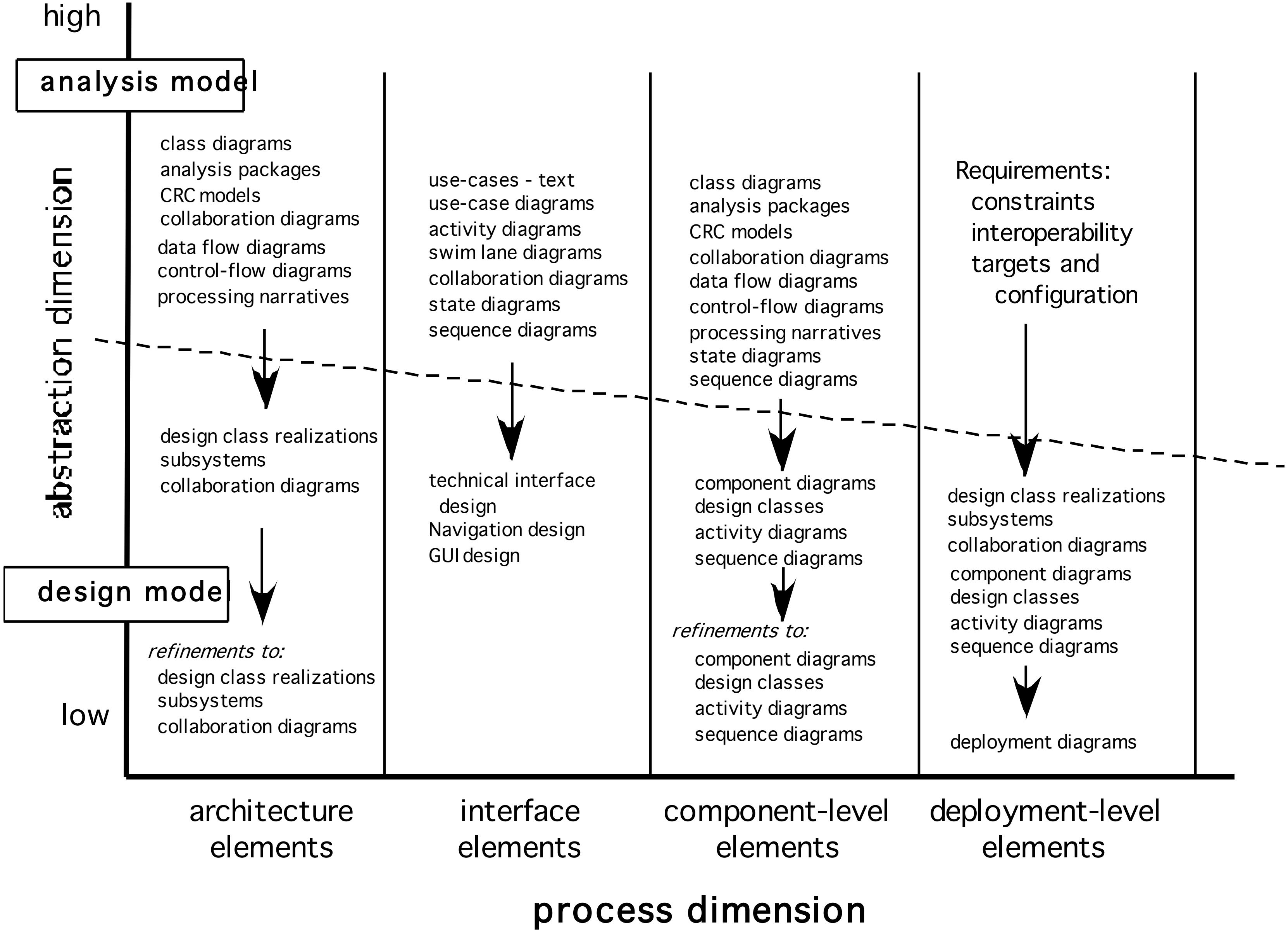
Law of Demeter

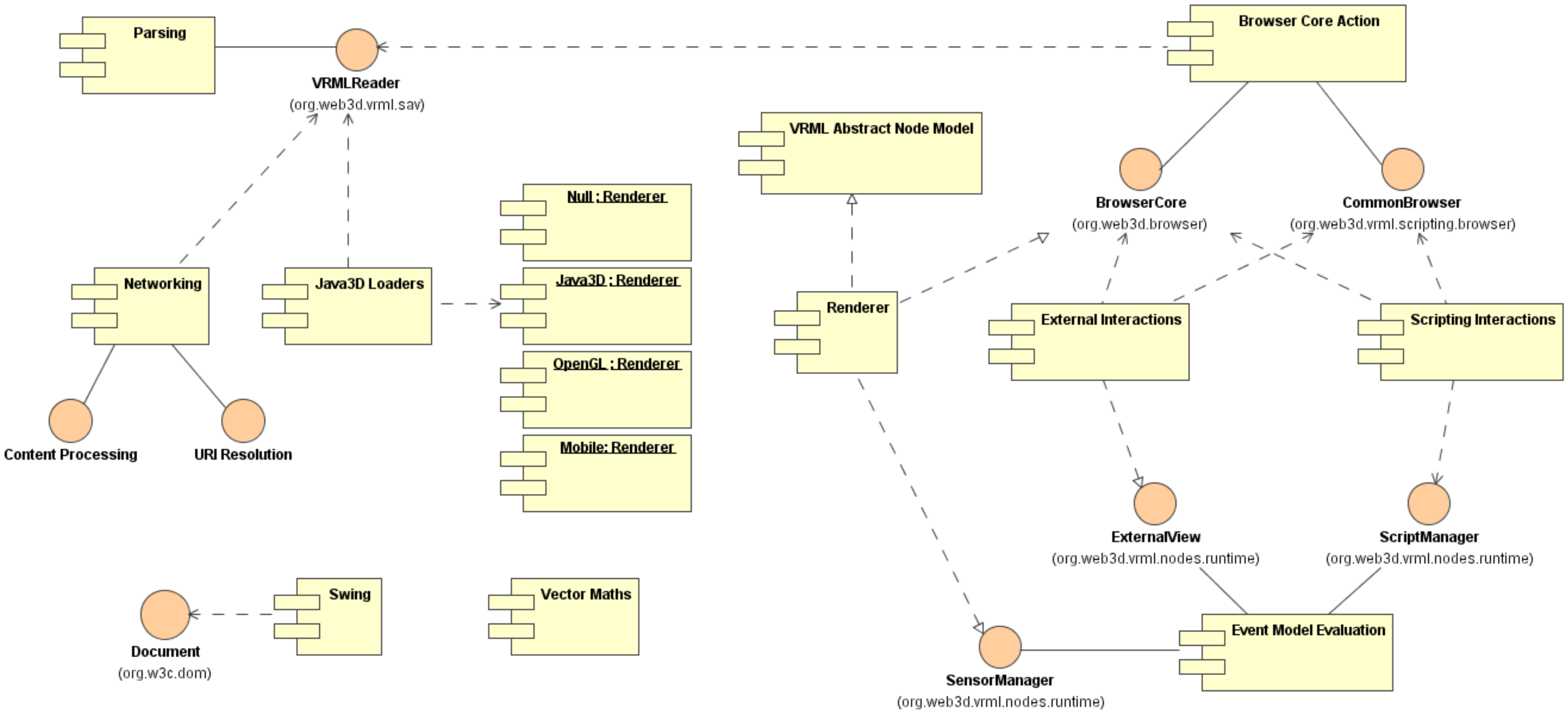
- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit
- Each unit should only talk to its friends; don't talk to strangers
- Only talk to your immediate friends

DESIGN MODEL ELEMENTS

- Data elements
  - Architectural level → databases and files
  - Component level → data structures
- Architectural elements
  - An architectural model is derived from:
    - Application domain
    - Analysis model
    - Available styles and patterns
- Interface elements
  - User interface (UI)
  - Interfaces to external systems
  - Interfaces to components within the application
- Component elements
- Deployment elements

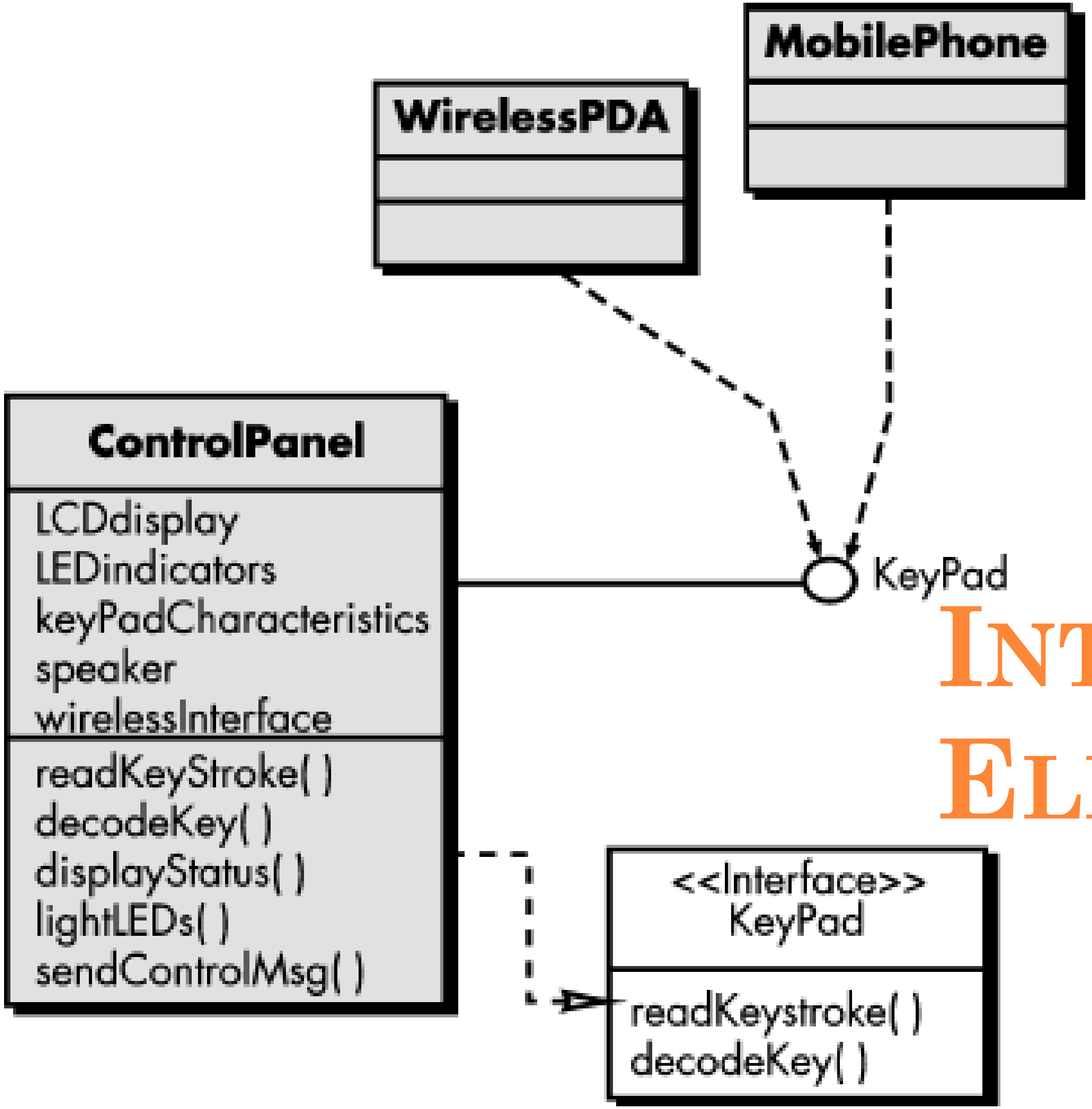
FROM ANALYSIS TO DESIGN



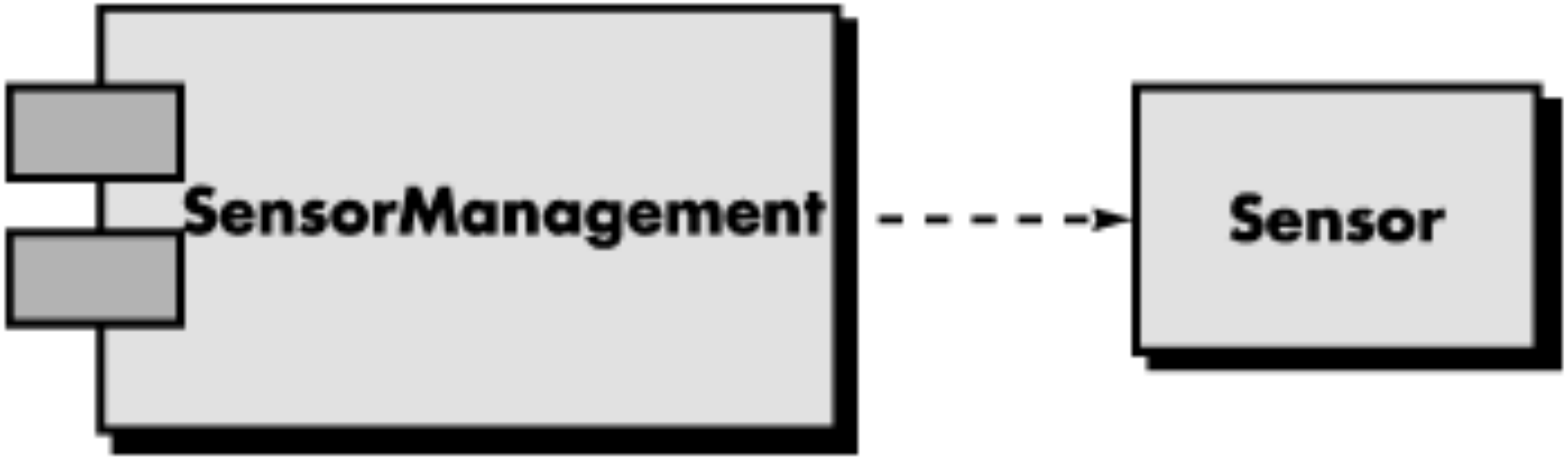


Architecture Elements

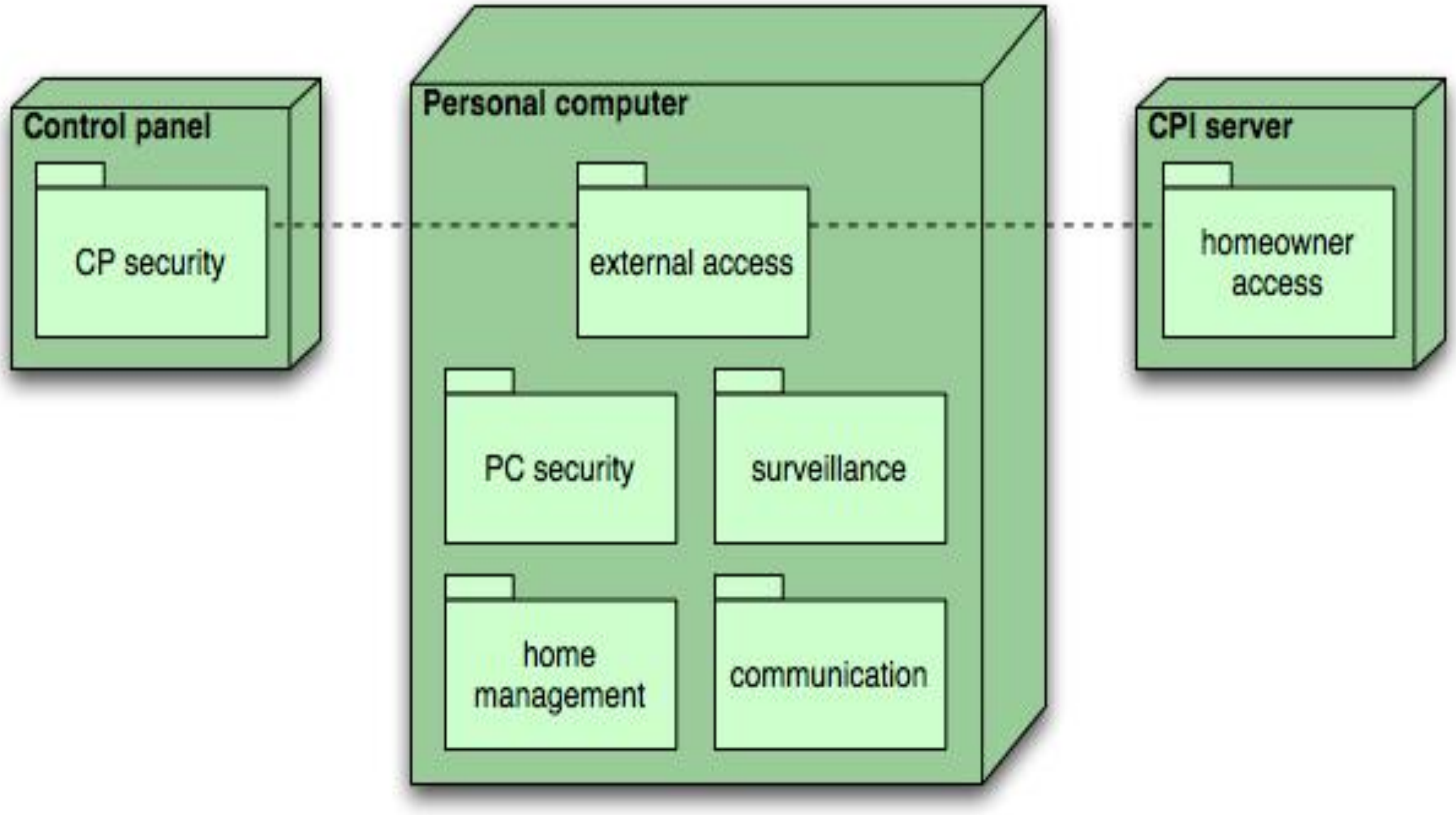
INTERFACE ELEMENTS



COMPONENT ELEMENTS



DEPLOYMENT DIAGRAM



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

DESIGN PRINCIPLES (CONT.)

- "***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***
- Structured to ***accommodate change***
- 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

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
  - the software should be logically partitioned into elements or subsystems
- A design should contain distinct representations of data, architecture, interfaces, and components.

QUALITY GUIDELINES (CONT.)

- 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

End of Chapter 8