Chapter 12

- **Design Concepts**

Slide Set to accompany

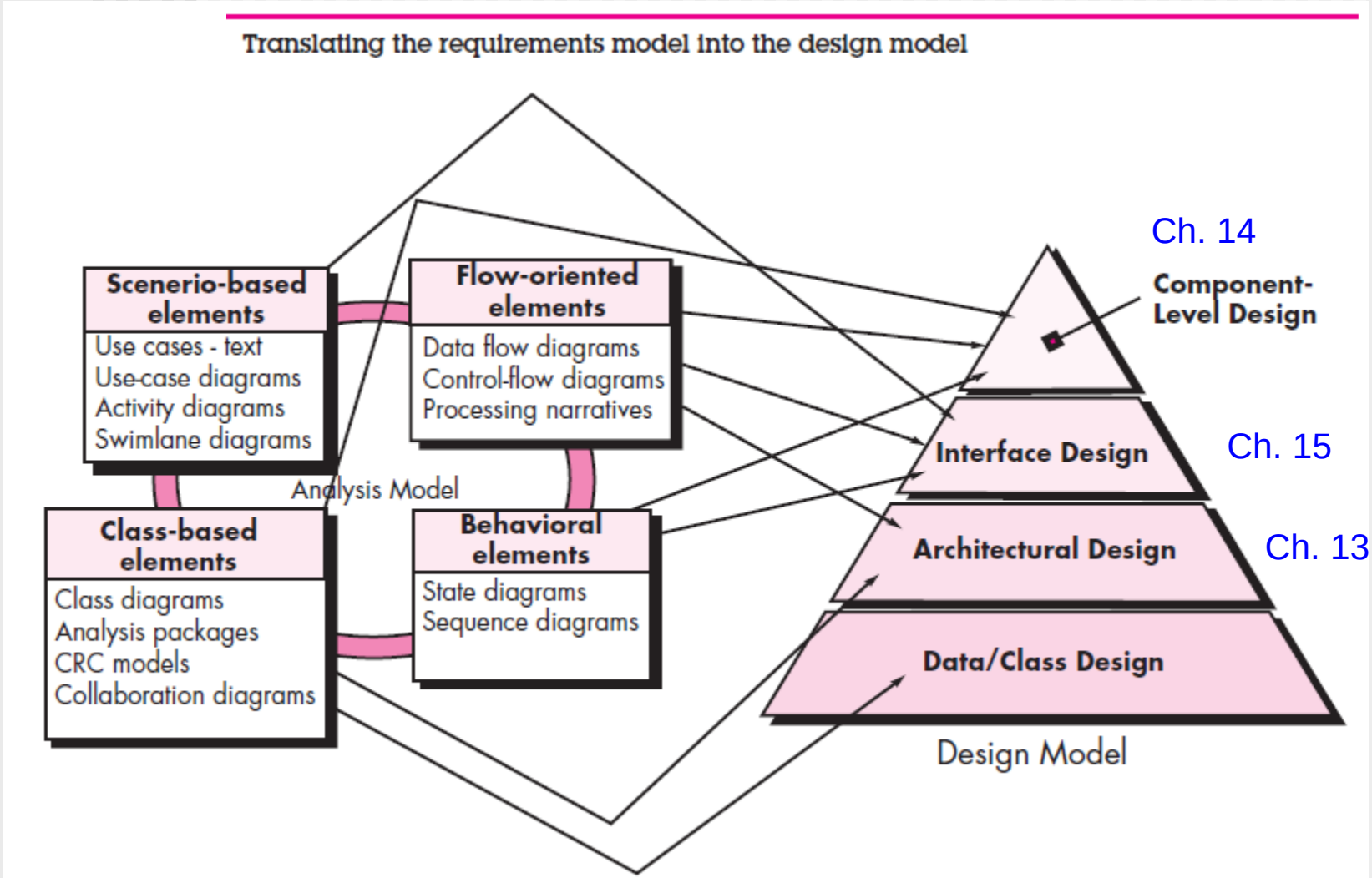
Software Engineering: A Practitioner's Approach

by Roger S. Pressman

Table of Contents

12.1 Design Within the Context of Software Engineering

From Analysis Model to Design Model



12.2 The Design Process

12.2.1 Software Quality Guidelines and Attributes

- The design must implement all of the explicit requirements contained in the analysis model, and accommodate all of the **implicit** requirements desired by the customer.
- The design must be a readable, understandable
- The design should provide a complete picture of the software regarding:
  - Data domain
  - Functional domain
  - Behavioral domain

Quality Guidelines

A design should

1. exhibit an architecture that
   (1) has been created using recognizable architectural styles or patterns,
   (2) is composed of components that exhibit good design characteristics
   (3) can be implemented in an evolutionary fashion
   - For smaller systems, design can sometimes be developed linearly.
2. be modular; software should be logically partitioned into elements or subsystems
3. contain distinct representations of data, architecture, interfaces, and components.
4. lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. lead to components that exhibit independent functional characteristics.
6. lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. be represented using a notation that effectively communicates its meaning.

Design Principles

The design process should

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

From Davis [DAV95]

12.2.2 The Evolution of Software Design

- Modular programs [Den73]
- To down refining software structures [Wir71].
- Procedural aspects -> *structured programming* [Dah72], [Mil72].
- Translation of data flow [Ste74] or data structure ([Jac75],[War74])
- Object-oriented design [Jac92], [Gam95]
- Software architecture [Kru06]
- Design patterns [Hol06] [Sha05]
- Aspect-oriented methods [Cla05],[Jac04]
- Model-driven development [Sch06]
- Test-driven development [Ast04]

12.3 Design Concepts

12.3.1 Abstraction —data, procedure, control

12.3.2 Architecture —the overall structure of the software

12.3.3 Patterns —"conveys the essence" of a proven design solution

12.3.4 Separation of concerns —any complex problem can be more easily handled if it is subdivided into pieces

12.3.5 Modularity —compartmentalization of data and function

12.3.6 Information Hiding —controlled interfaces

12.3.7 Functional independence —single-minded function and low coupling

12.3.8 Refinement —elaboration of detail for all abstractions

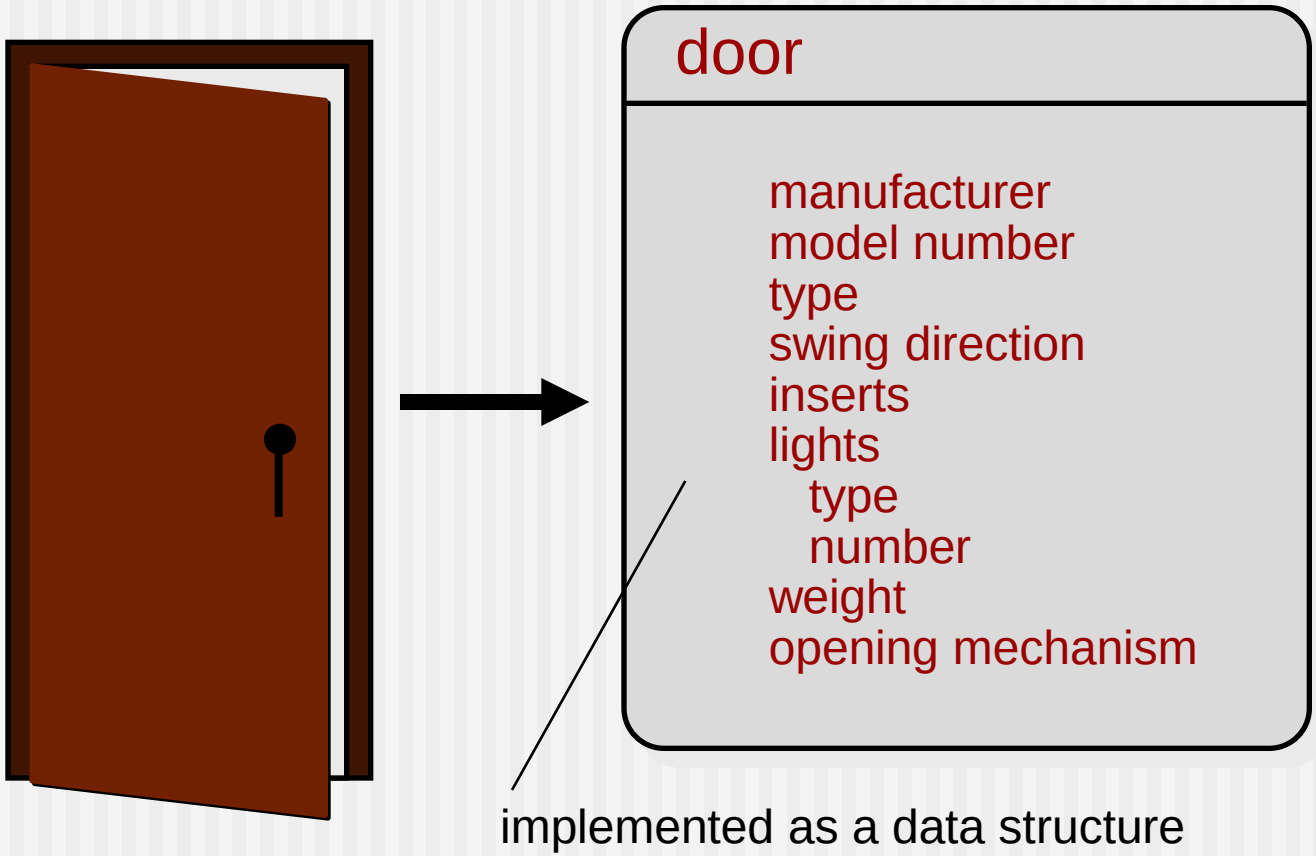
12.3.9 Aspects —a mechanism for understanding how global requirements affect design

12.3.10 Refactoring —a reorganization technique that simplifies the design

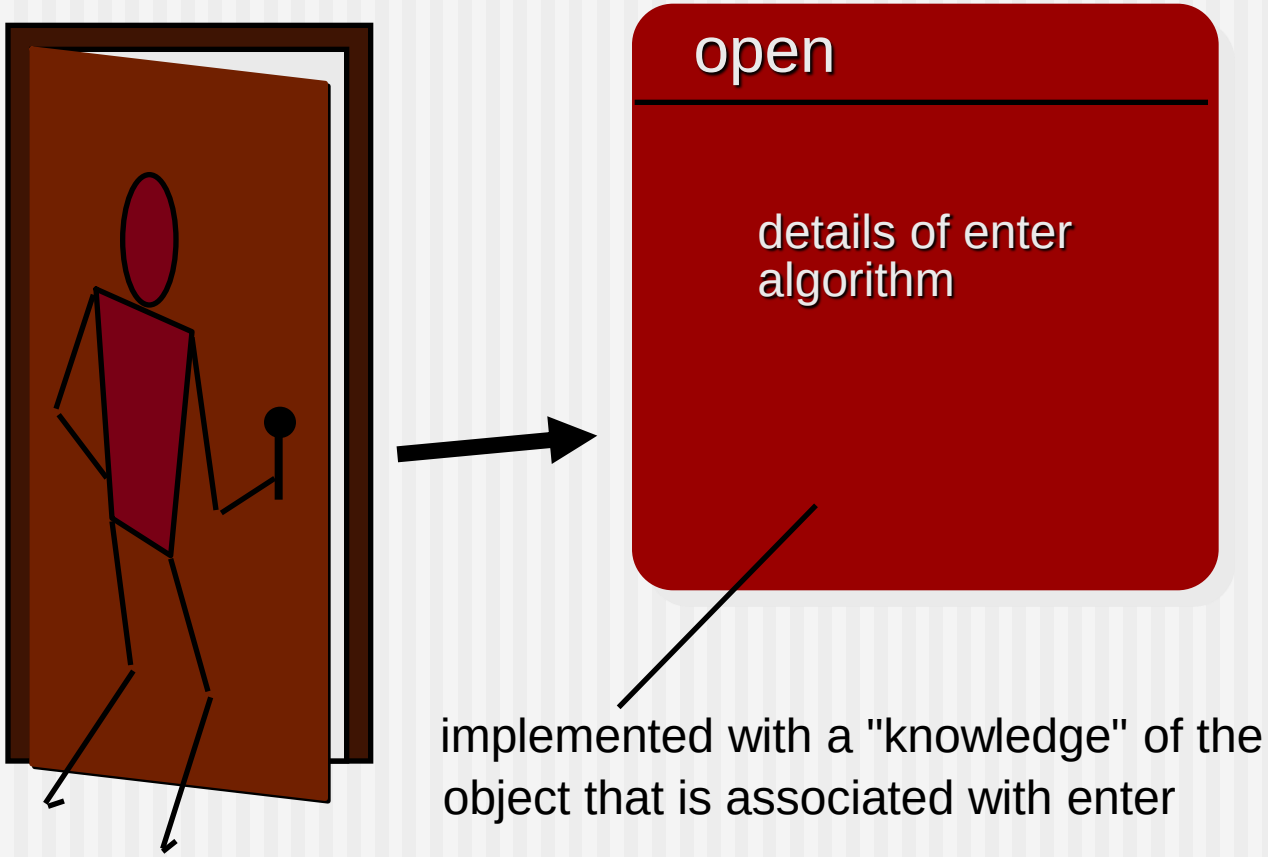
12.3. 11 OO design concepts —Appendix II

12.3.12 Design Classes —provide design detail that will enable analysis classes to be implemented

12.3.1 Abstraction
Data Abstraction



Procedural Abstraction



12.3.2 Architecture

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

Structural properties - 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.

Extra-functional (= Non-functional) properties. Design architecture should achieve requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. Should consider the design of families of similar systems. => Software Product Line

12.3.3 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

12.3.4 Separation of Concerns

- A 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.

12.3.5 Modularity

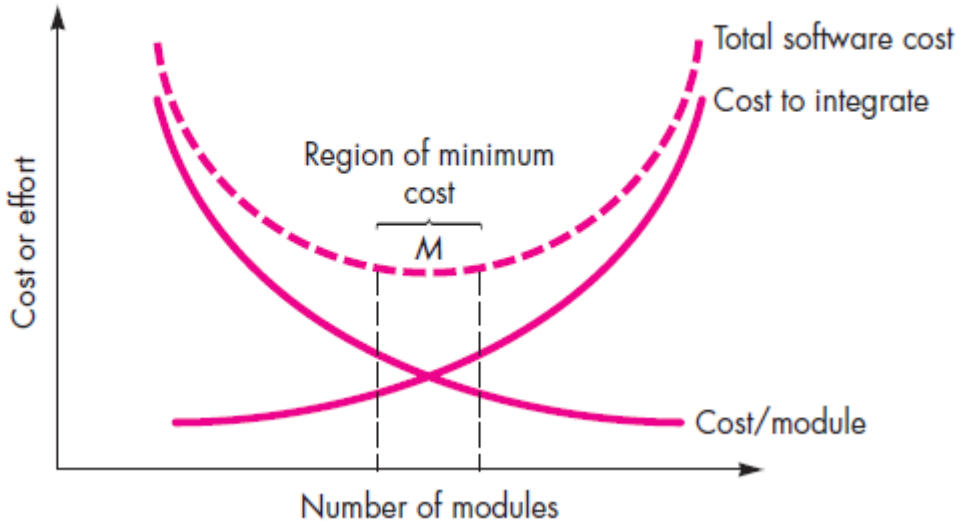
- “Modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - The number of control paths, span of reference, number of variables,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

Modularity: Trade-offs

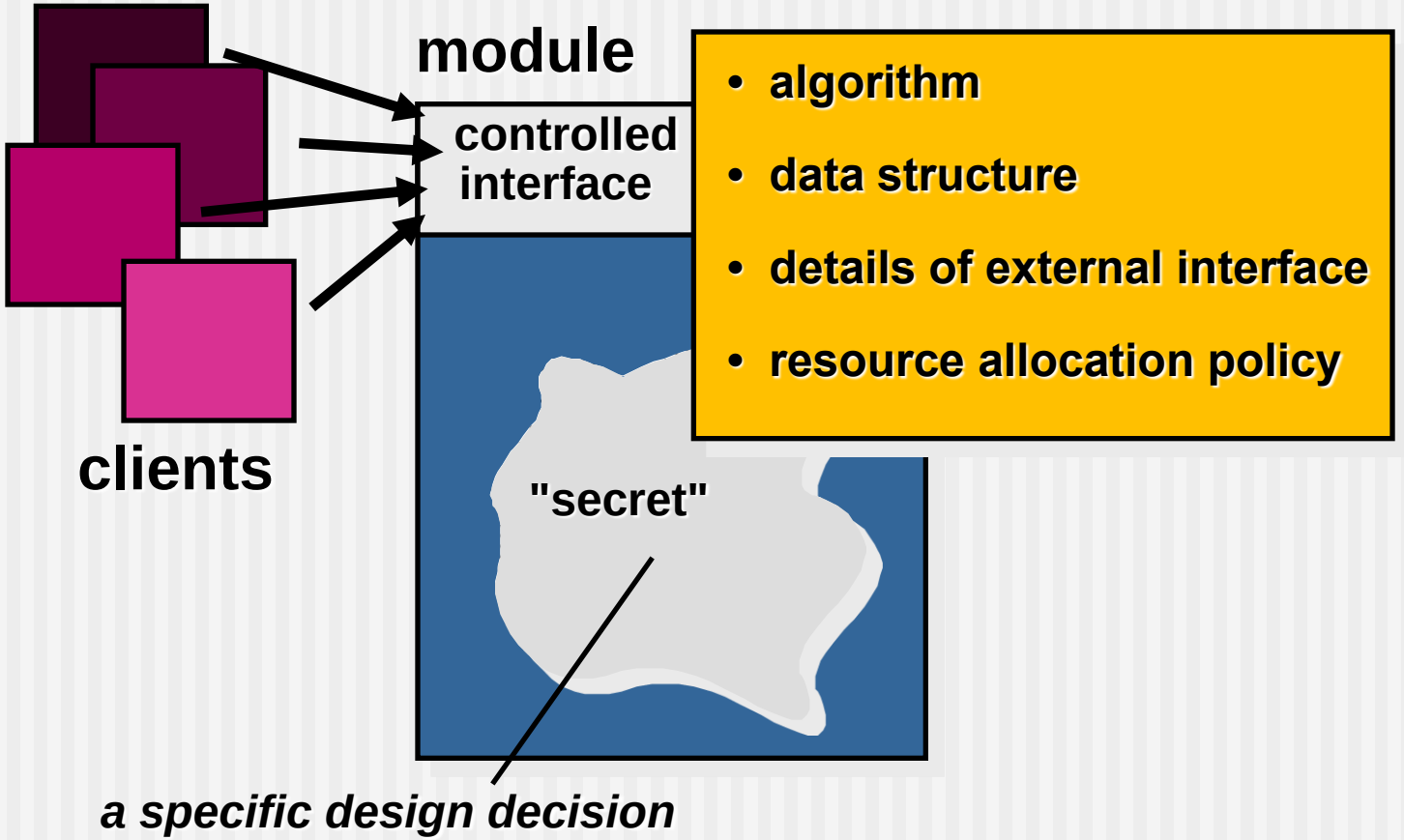
What is the "right" number of modules for a specific software design?

FIGURE 8.2

Modularity and software cost



12.3.6 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

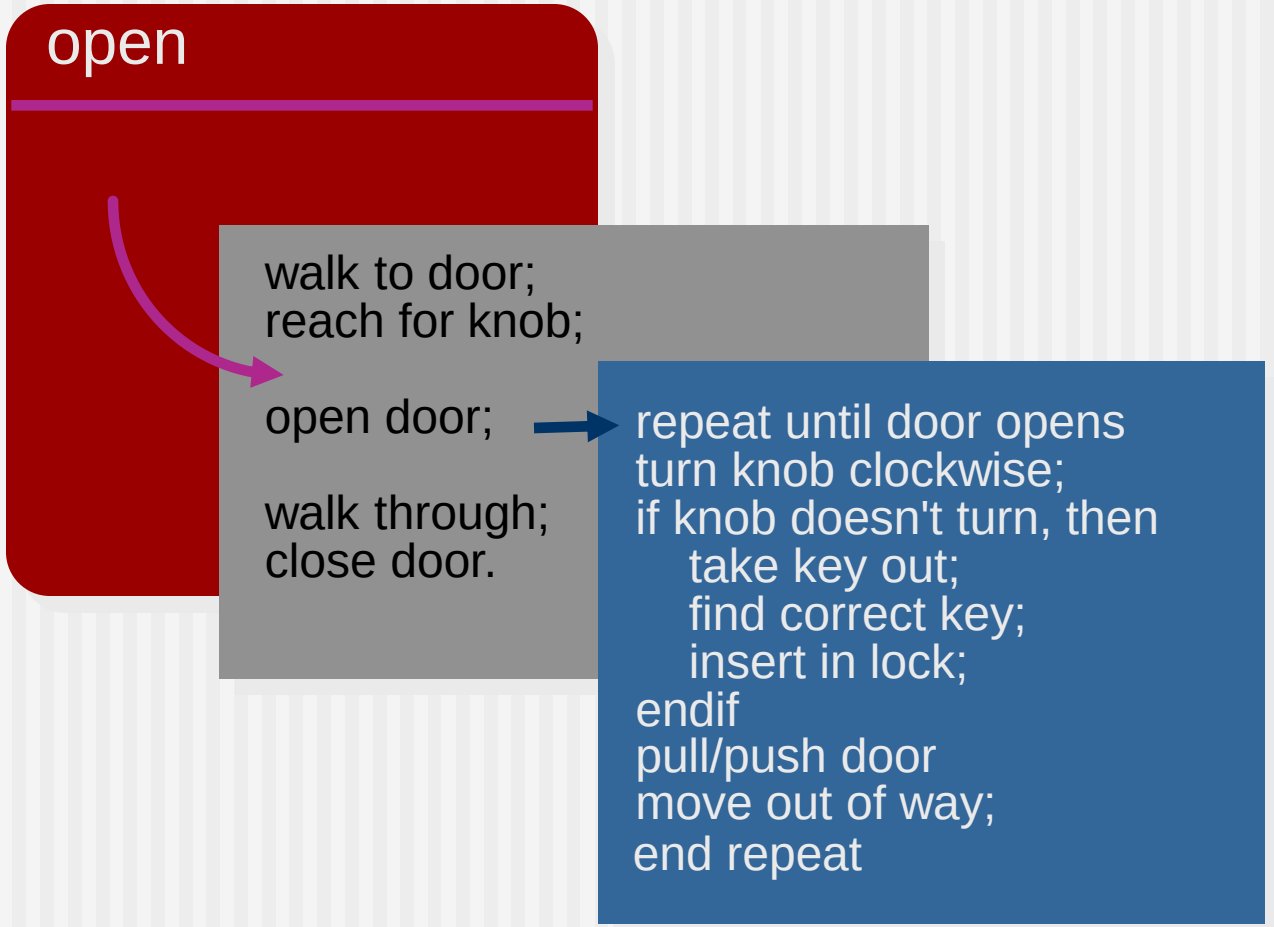
12.3.7 Functional Independence

- Achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- Cohesion: 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: relative interdependence among modules.
  - depends on
    - the interface complexity between modules
    - the point at which entry or reference is made to a module,
    - what data pass across the interface.

12.3.8 Refinement

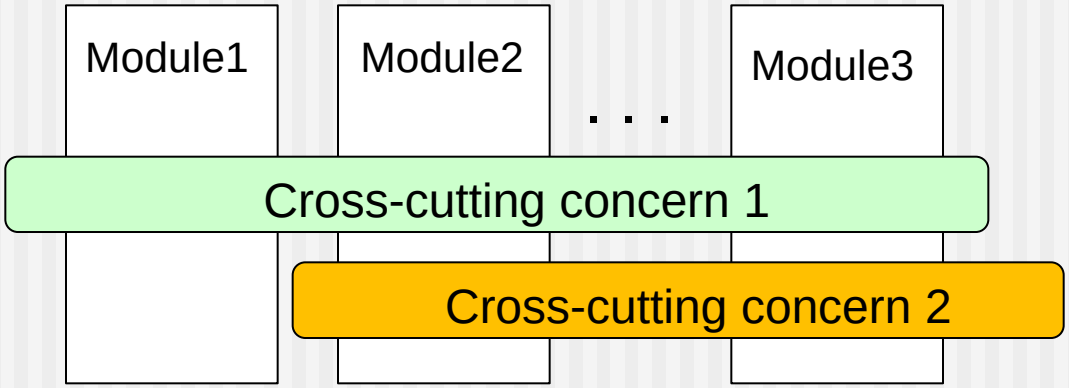
- Stepwise refinement
  - Top-down design strategy originally proposed by Niklaus Wirth [Wir71].
  - A program is developed by successively refining levels of procedural detail.
    - A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Example: Stepwise Refinement



12.3.9 Aspects

- A representation of a cross-cutting concern.
- 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]



Aspects—An Example

- Consider the requirements for the **SafeHomeAssured.com** WebApp.
  - Requirement *A:*
    - 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.
  - Requirement *B:*
    - States that *a registered user must be validated prior to using* **SafeHomeAssured.com.**
    - Applicable for all functions available to registered *SafeHome* users.
    - As design refinement occurs,
      - *A** is a design representation for requirement *A*
      - *B** is a design representation for requirement *B*.

      ☛ *A** and *B** are representations of concerns, and B* *cross-cuts* A*.
- *B** of the requirement is an aspect of the *SafeHome* WebApp.

12.3.10 Refactoring

- Refactoring [FOW99]
  - " 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."
  - 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.

12.3.11 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: characteristic that greatly reduces the effort required to extend the design

12.3.12 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.

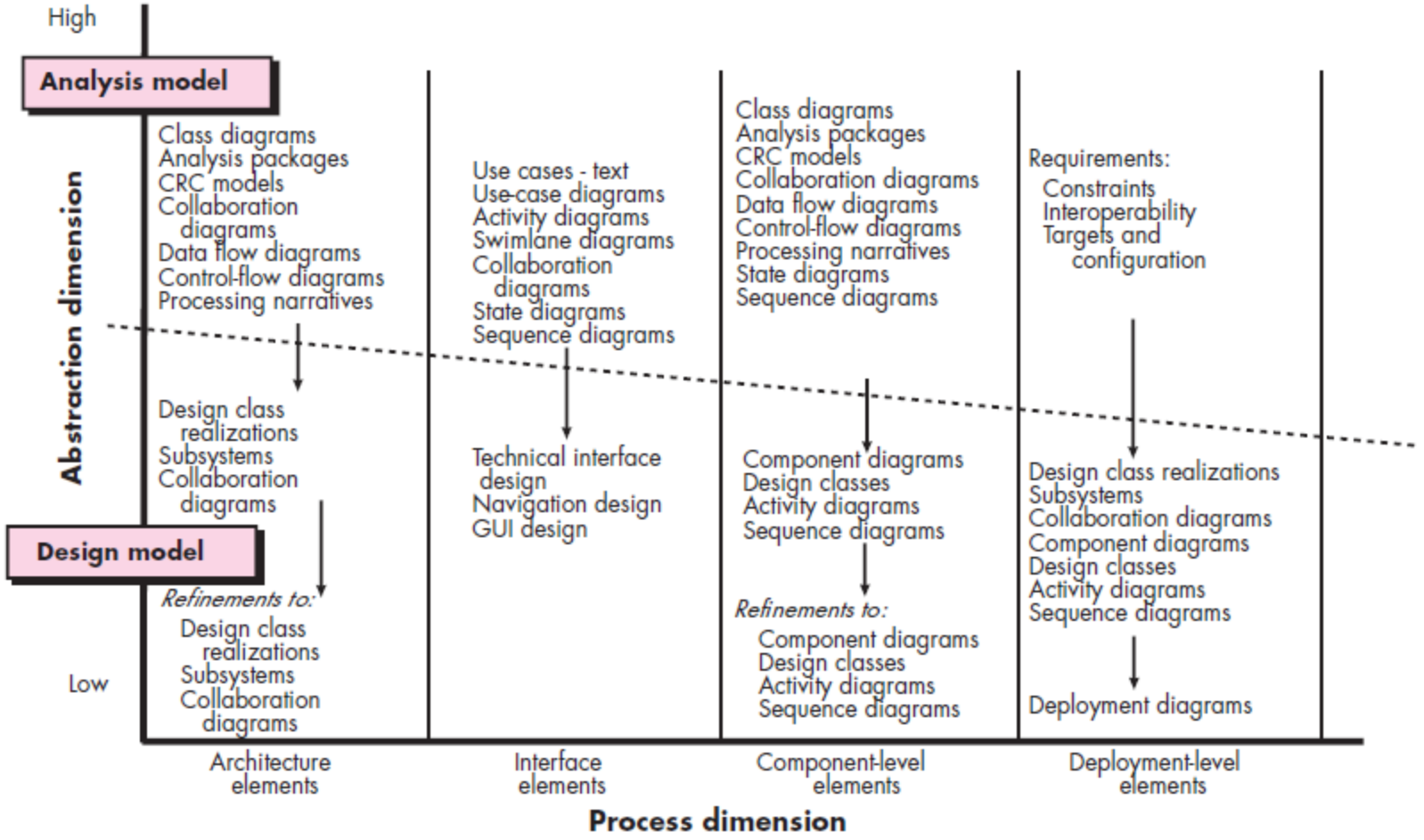
Design Class Characteristics

- Complete - includes all necessary attributes and methods) and sufficient (contains only those methods needed to achieve class intent)
- Primitiveness – each class method focuses on providing one service
- High cohesion – small, focused, single-minded classes
- Low coupling – class collaboration kept to minimum

12.4 The Design Model

- 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”
- 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

FIGURE 8.4 Dimensions of the design model



12.4.1 Data Design Elements

- Data Modeling
  - examines data objects independently of processing
  - focuses attention on the data domain
  - creates a model at the customer's level of abstraction
  - indicates how data objects relate to one another

What is a Data Object?

- a representation of almost any composite information that must be understood by software.
  - *composite information*—something that has a number of different properties or attributes
- can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).
- The description of the data object incorporates the data object and all of its attributes.
- A data object encapsulates data only—there is no reference within a data object to operations that act on the data.

Data Objects and Attributes

A data object contains a set of attributes that act as an aspect, quality, characteristic, or descriptor of the object

object: automobile

attributes:

- **make**
- **model**
- **body type**
- **price**
- **options code**

What is a Relationship?

- Data objects are connected to one another in different ways.
  - A connection is established between **person** and **car** because the two objects are related.
    - A person *owns* a car
    - A person *is insured to drive* a car
- The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car.**
- Several instances of a relationship can exist
- Objects can be related in many different ways

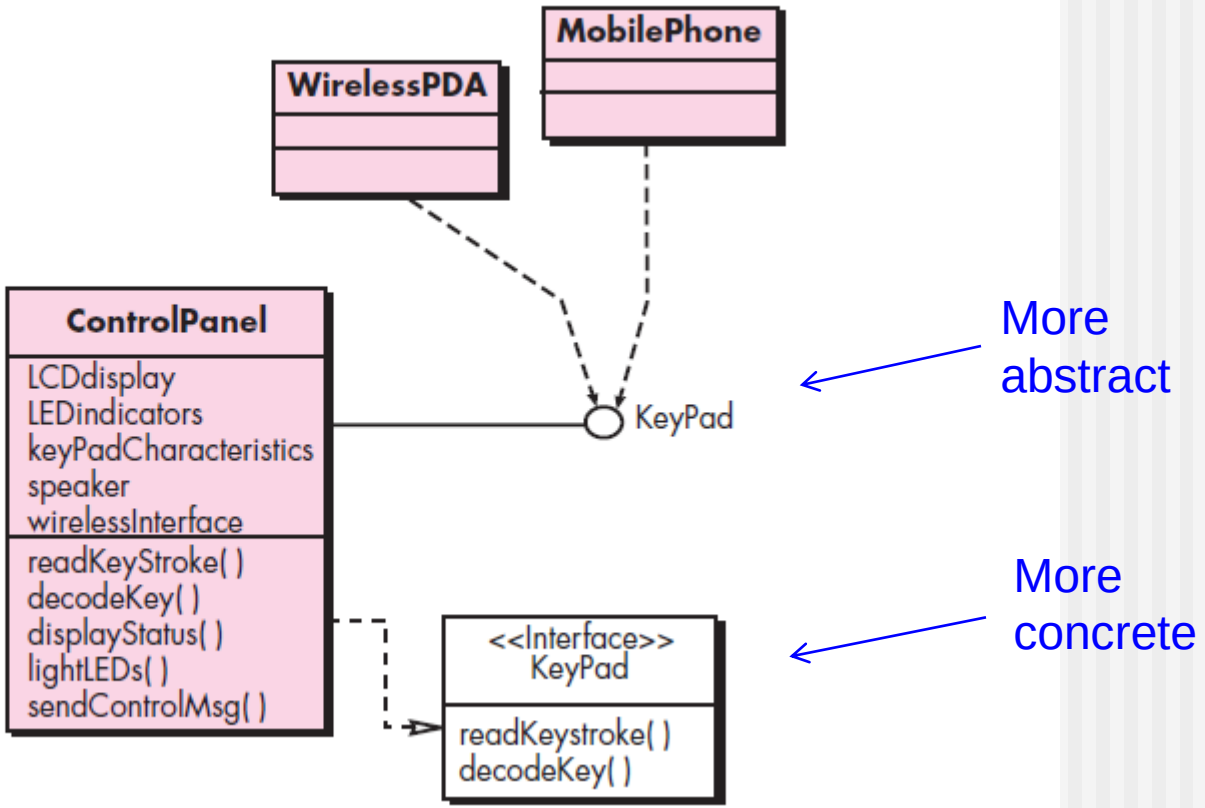
12.4.2 Architectural Design Elements

- The architectural model [Sha96] is derived from:
  1. Information about the application domain
  2. Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand
  3. The availability of architectural patterns and styles.

12.4.3 Interface Design Elements

FIGURE 8.5

Interface representation for Control-Panel



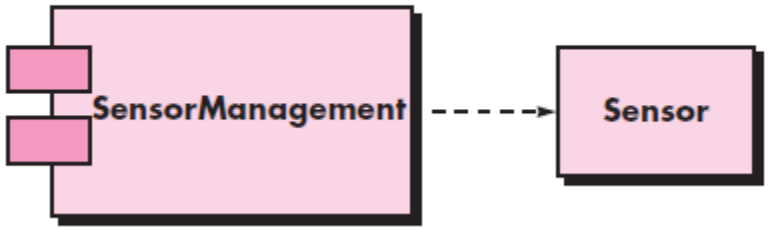
Interface Elements

- Interface is a set of operations that describes the externally observable behavior of a class and provides access to its public operations
- Important elements
  - User interface (UI)
  - External interfaces to other systems
  - Internal interfaces between various design components
- Modeled using UML communication diagrams (called collaboration diagrams in UML 1.x)

12.4.4 Component-level Design Elements

FIGURE 8.6

A UML component diagram



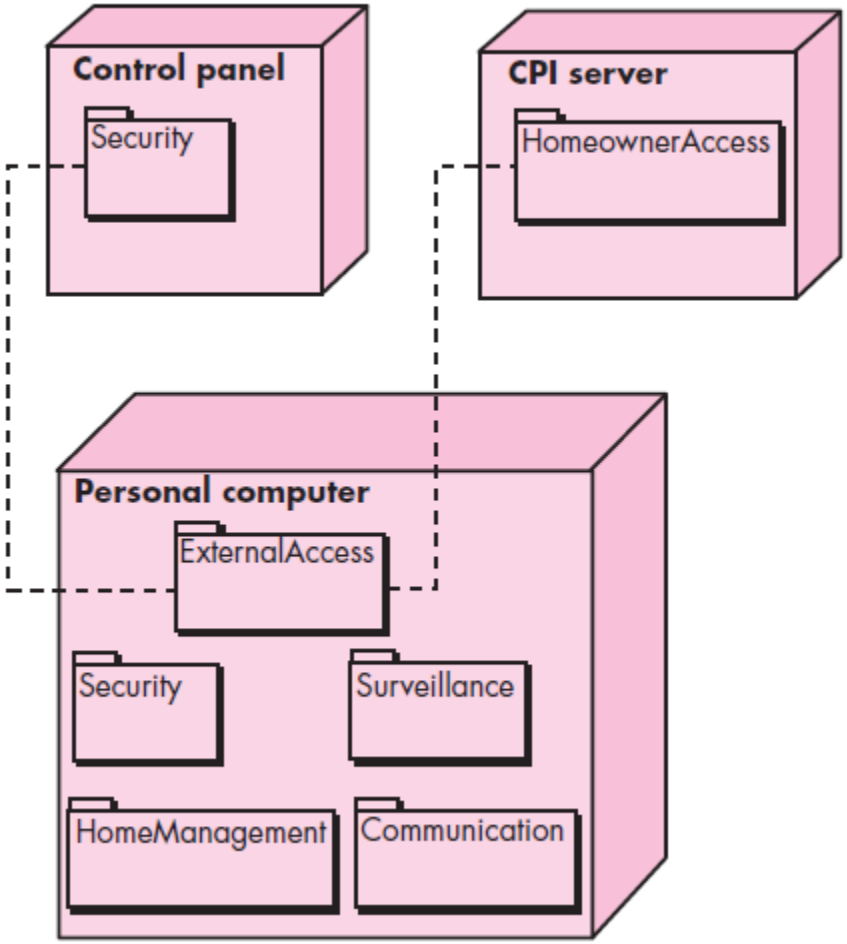
Component Elements

- 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

12.4.5 Deployment-Level Design Elements

FIGURE 8.7

A UML deployment diagram



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