投票 最多可选1项

设置

Welcome back to Software Engineering!

How about your long holiday?

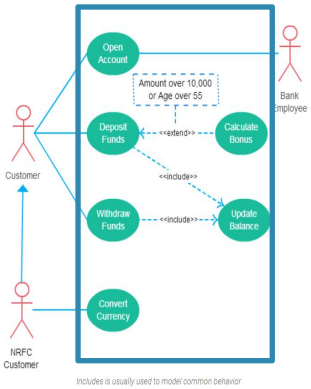
A great

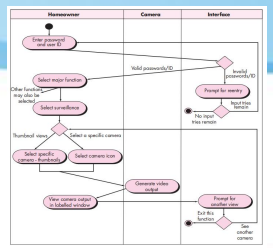
B not bad

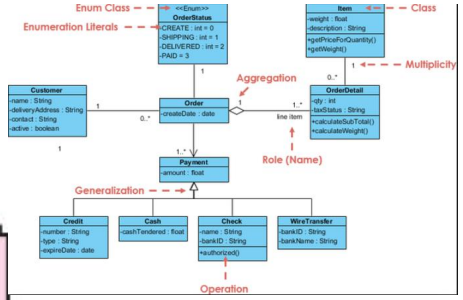
C other

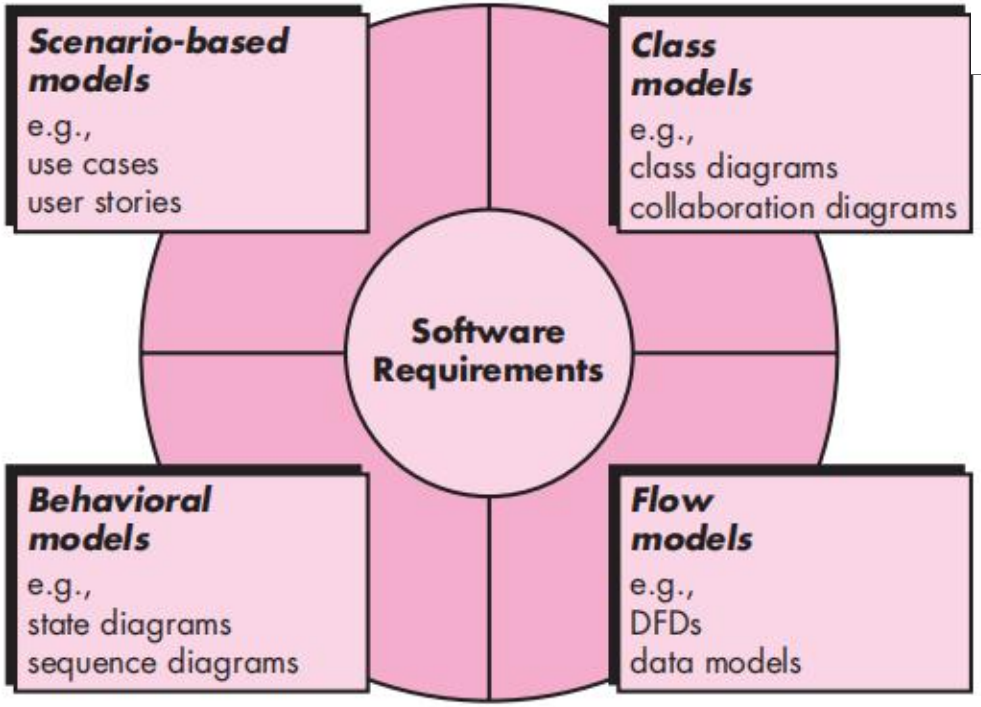
提交

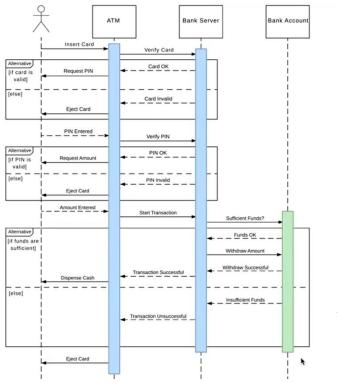
Review - Requirement Analysis

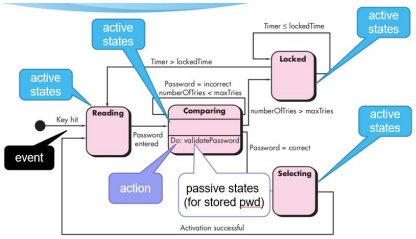


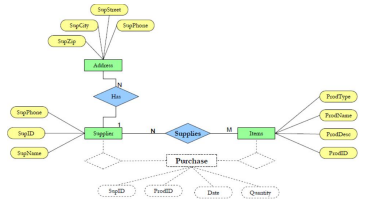


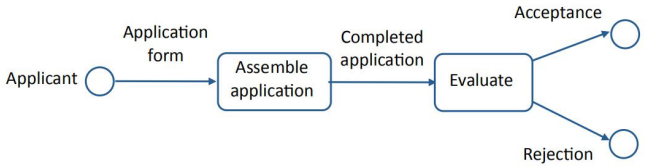




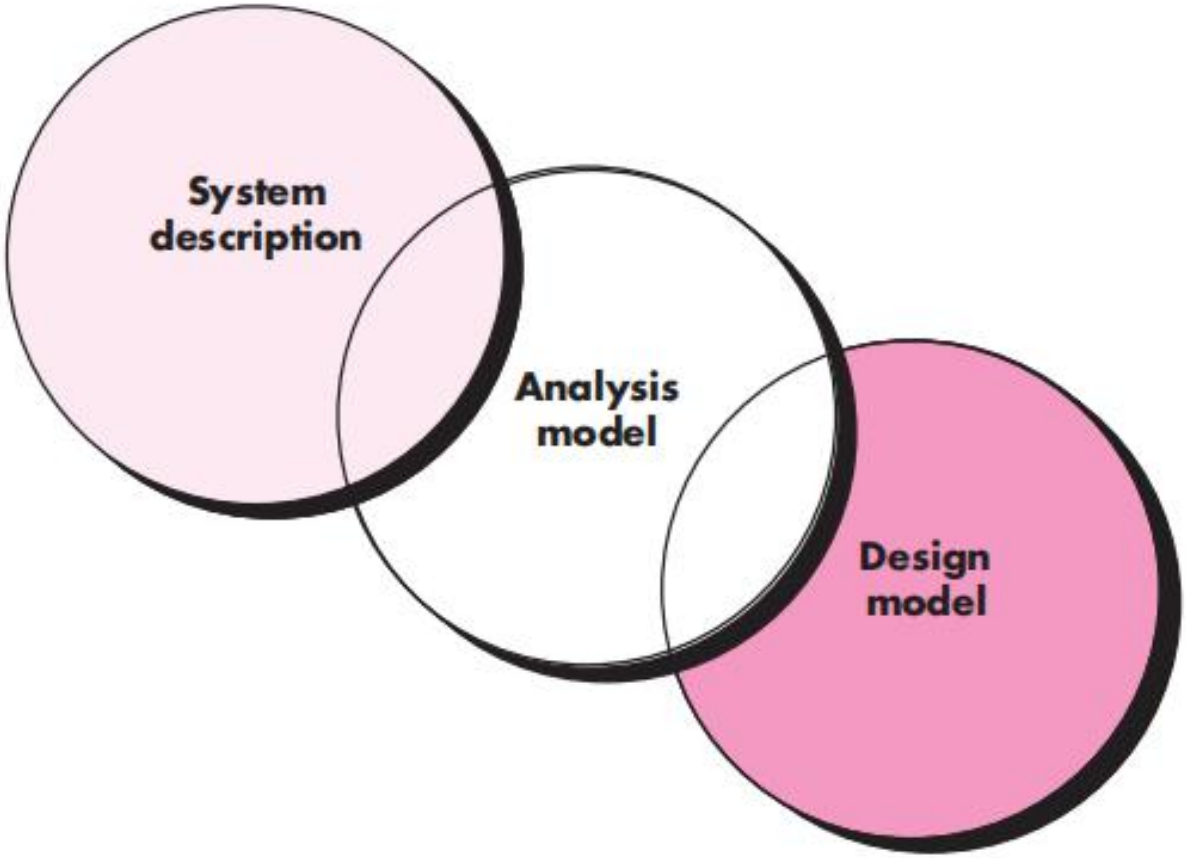








Review - Requirement Analysis



Software Engineering

Part 2
Modeling

Chapter 12
Design Concepts

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 8/e* (McGraw-Hill 2014). Slides copyright 2014 by Roger Pressman.

Reproduced by Ning Li , 2022

Contents

Contents

12.1 Design within the Context of Software Engineering

- Good software design should exhibit:
  - ***Firmness***: Should not have any bugs that inhibit its function.
  - ***Commodity***: Suitable for the purposes for which it was intended.
  - ***Delight***: The experience of using the program should be pleasurable one.

12.1 Software Design

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.

.

12.1 Analysis Model -> Design Model

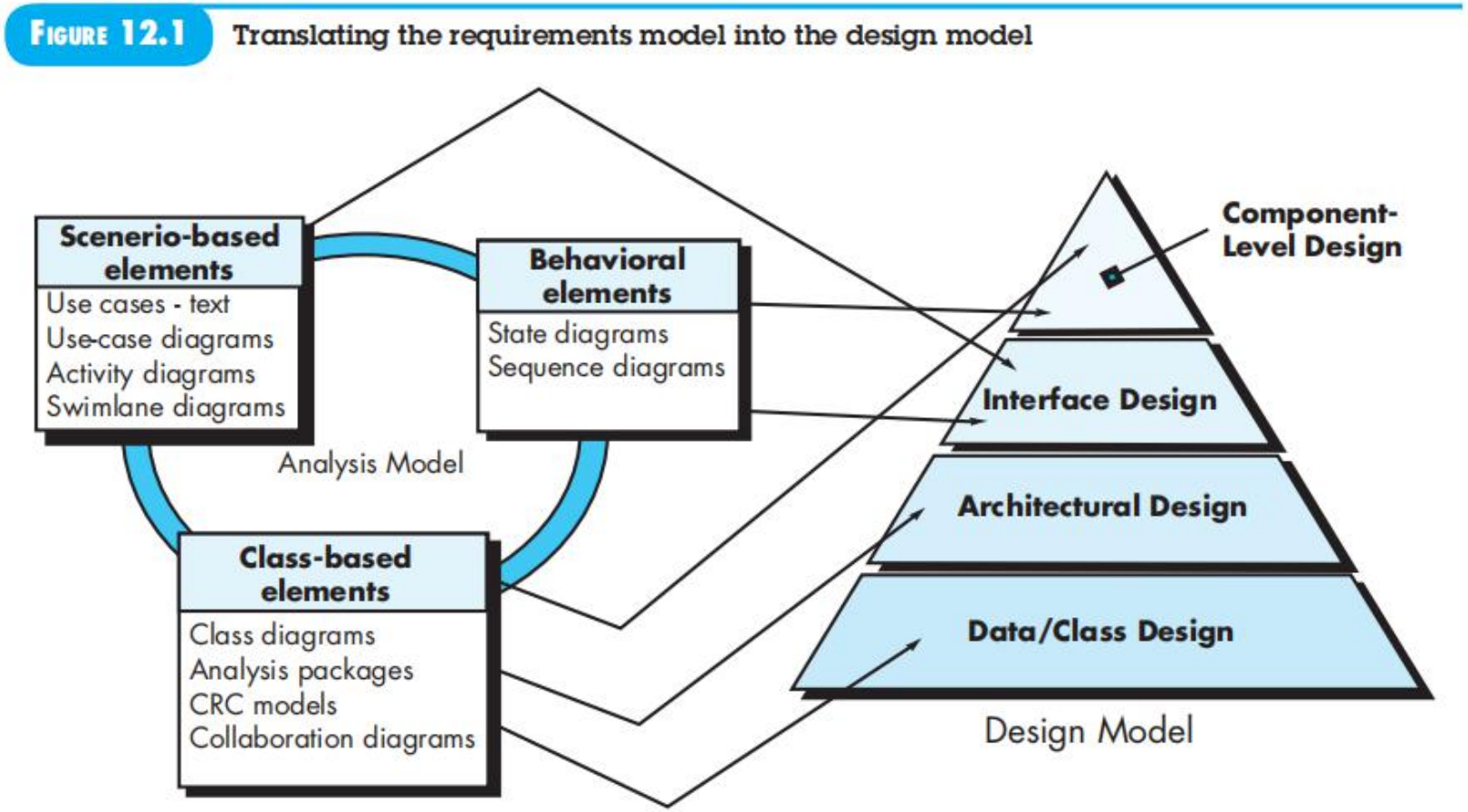


FIGURE 12.1 Translating the requirements model into the design model

12.1 Software Engineering Design

- Data/Class design – transforms analysis classes into implementation classes and data structures
- Architectural design – defines relationships among the major software structural elements
- Interface design – defines how software elements, hardware elements, and end-users communicate
- Component-level design – transforms structural elements into procedural descriptions of software components

12.2.1 Design and Quality

Three characteristics for a good design:

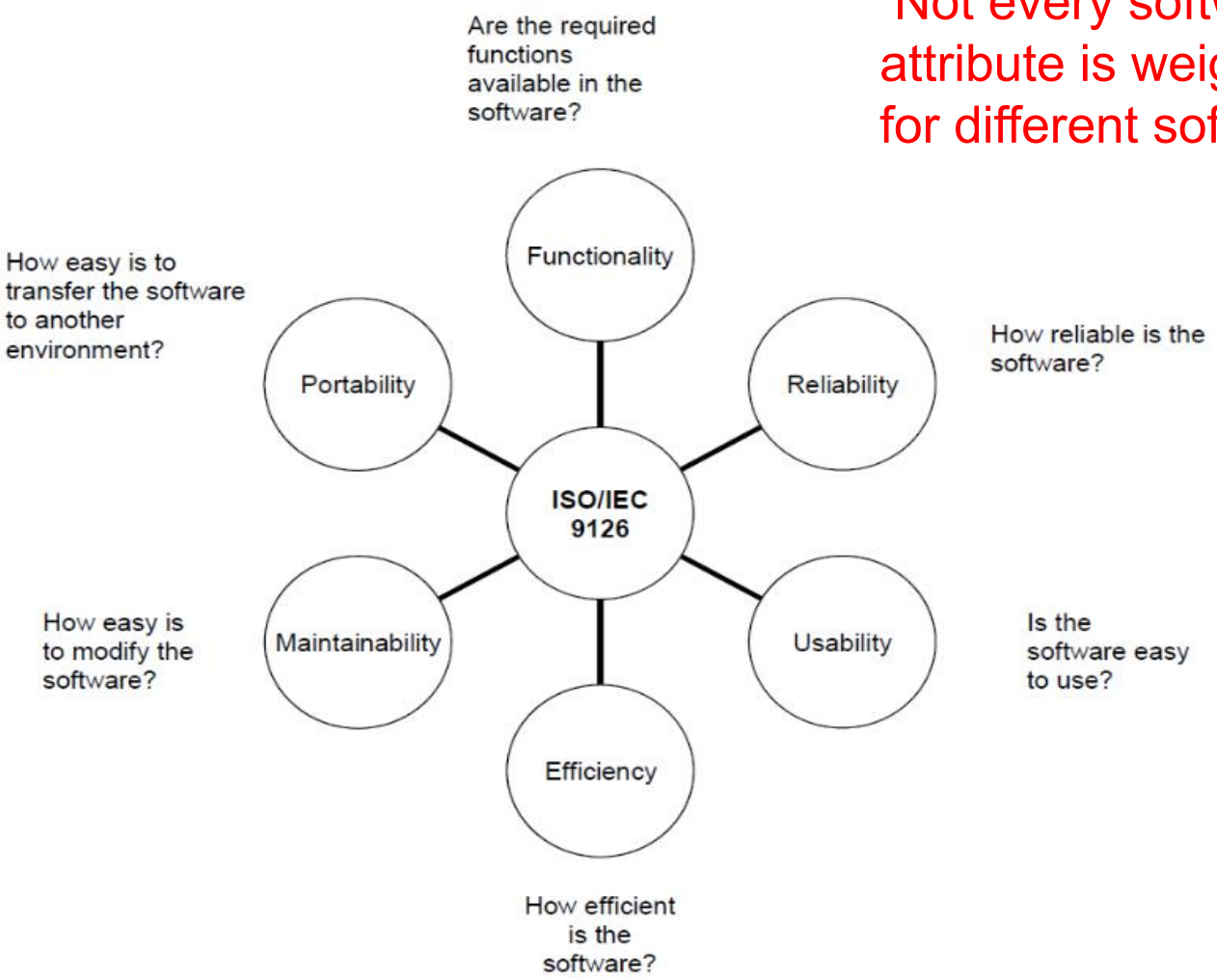
- the design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- the design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- the design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

12.2.1 Quality Guidelines

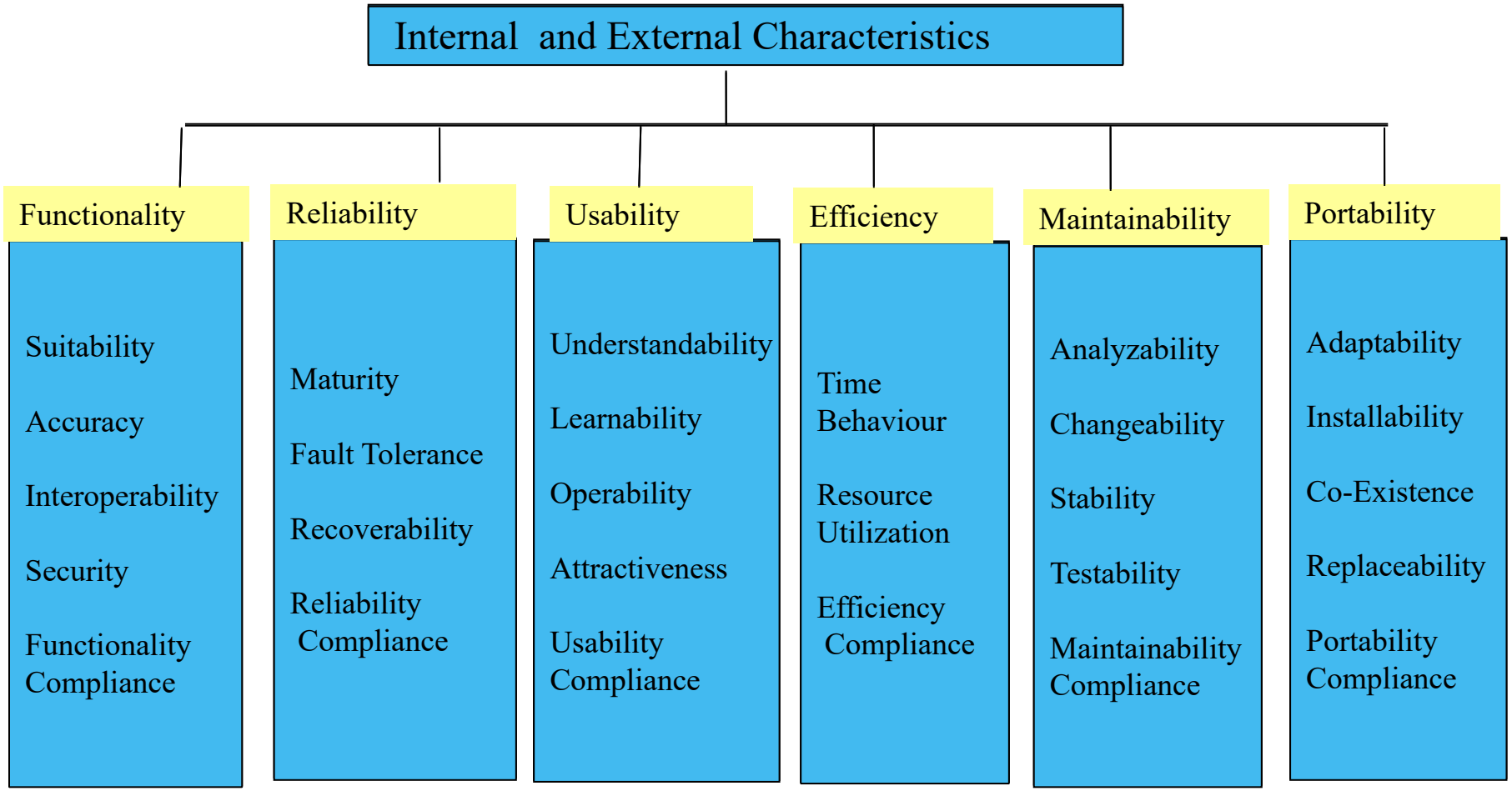
- A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion
- A design should be modular; that is, the software should 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.

12.2.1 Design and Quality

Not every software quality attribute is weighted equally for different software

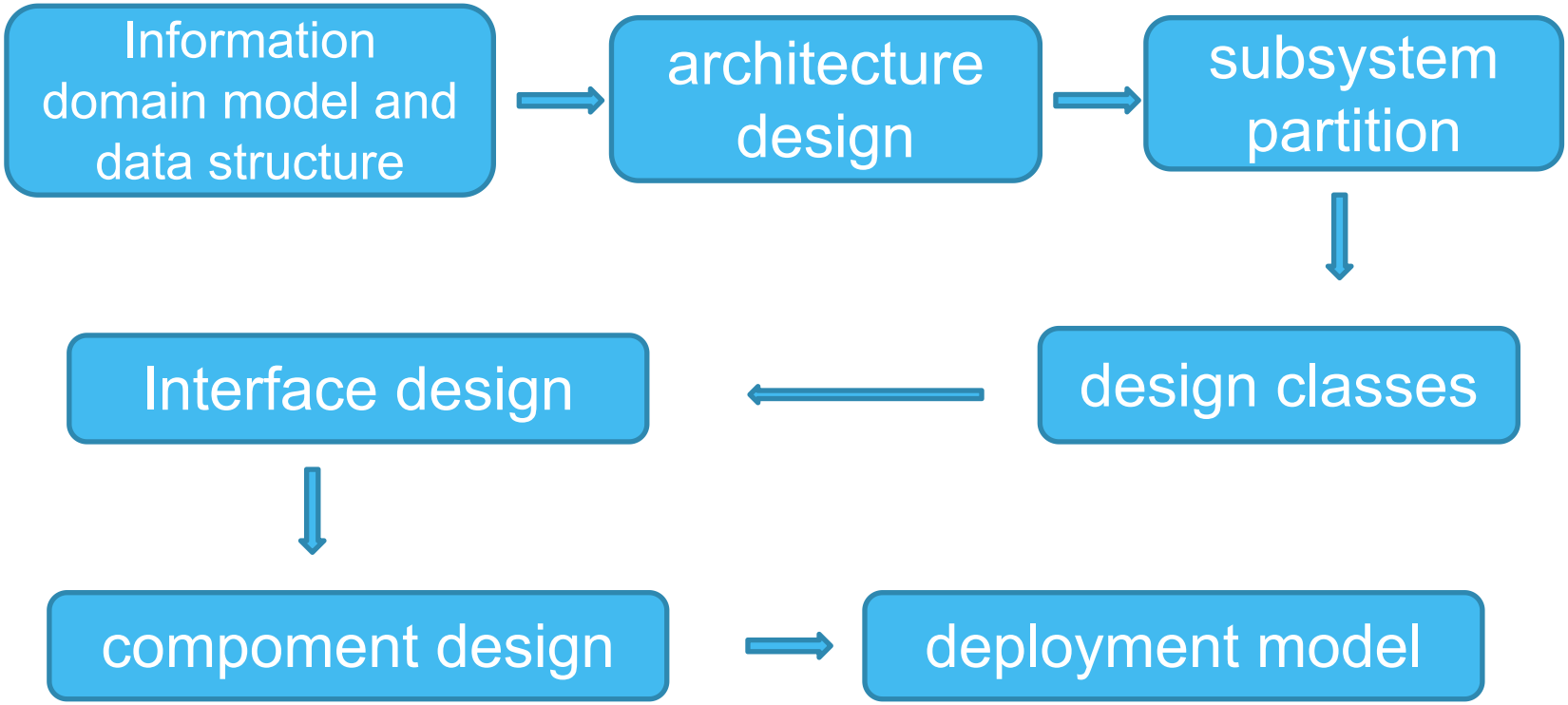


12.2.1 Design and Quality



12.2.2 Evolution and Design

Tasks for Design :



12.2.2 Evolution and Design

TASK SET

Generic Task Set for Design

1. Examine the information domain model and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style (pattern) that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture:
   - Be certain that each subsystem is functionally cohesive.
   - Design subsystem interfaces.
   - Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components:
   - Translate analysis class description into a design class.
   - Check each design class against design criteria; consider inheritance issues.
   - Define methods and messages associated with each design class.
   - Evaluate and select design patterns for a design class or a subsystem.
   - Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface:
   - Review results of task analysis.
   - Specify action sequence based on user scenarios.
   - Create behavioral model of the interface.
   - Define interface objects, control mechanisms.
   - Review the interface design and revise as required.
7. Conduct component-level design.
   - Specify all algorithms at a relatively low level of abstraction.
   - Refine the interface of each component.
   - Define component-level data structures.

   Review each component and correct all errors uncovered.
8. Develop a deployment model.

12.3 Fundamental Concepts of Design

1. Abstraction—data, procedure, control
2. Architecture—the overall structure of the software
3. Patterns—"conveys the essence" of a proven design solution
4. Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
5. Modularity—compartmentalization of data and function
6. Hiding—controlled interfaces
7. Functional independence—single-minded function and low coupling
8. Refinement—elaboration of detail for all abstractions
9. Aspects—a mechanism for understanding how global requirements affect design
10. Refactoring—a reorganization technique that simplifies the design
11. OO design concepts—Appendix II
12. Design Classes—provide design detail that will enable analysis classes to be implemented

12.3 Abstraction

■ Modular solution to any problem

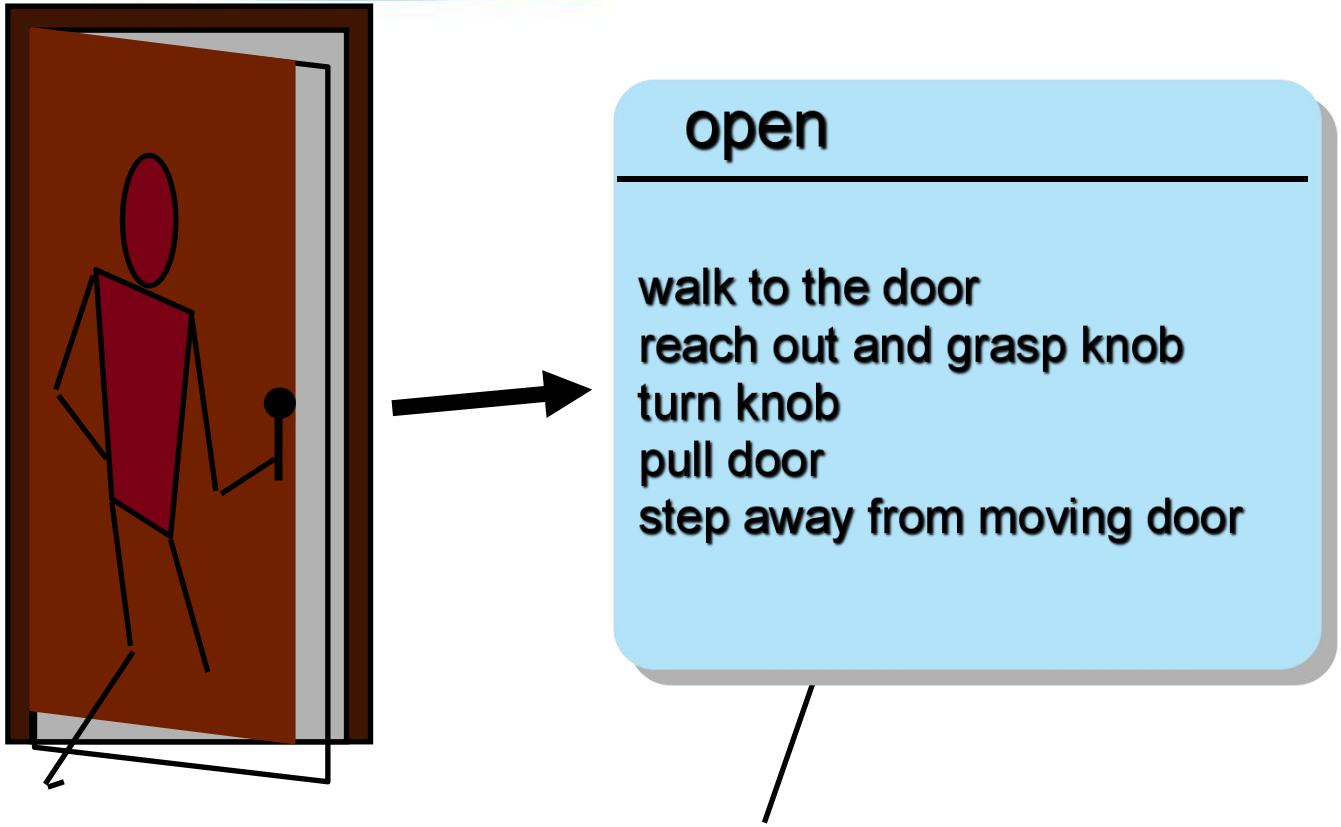
Key: Different level abstraction

- broad terms to describe
- more detailed description
- implementation-oriented
- solution can be directly implemented

Procedure

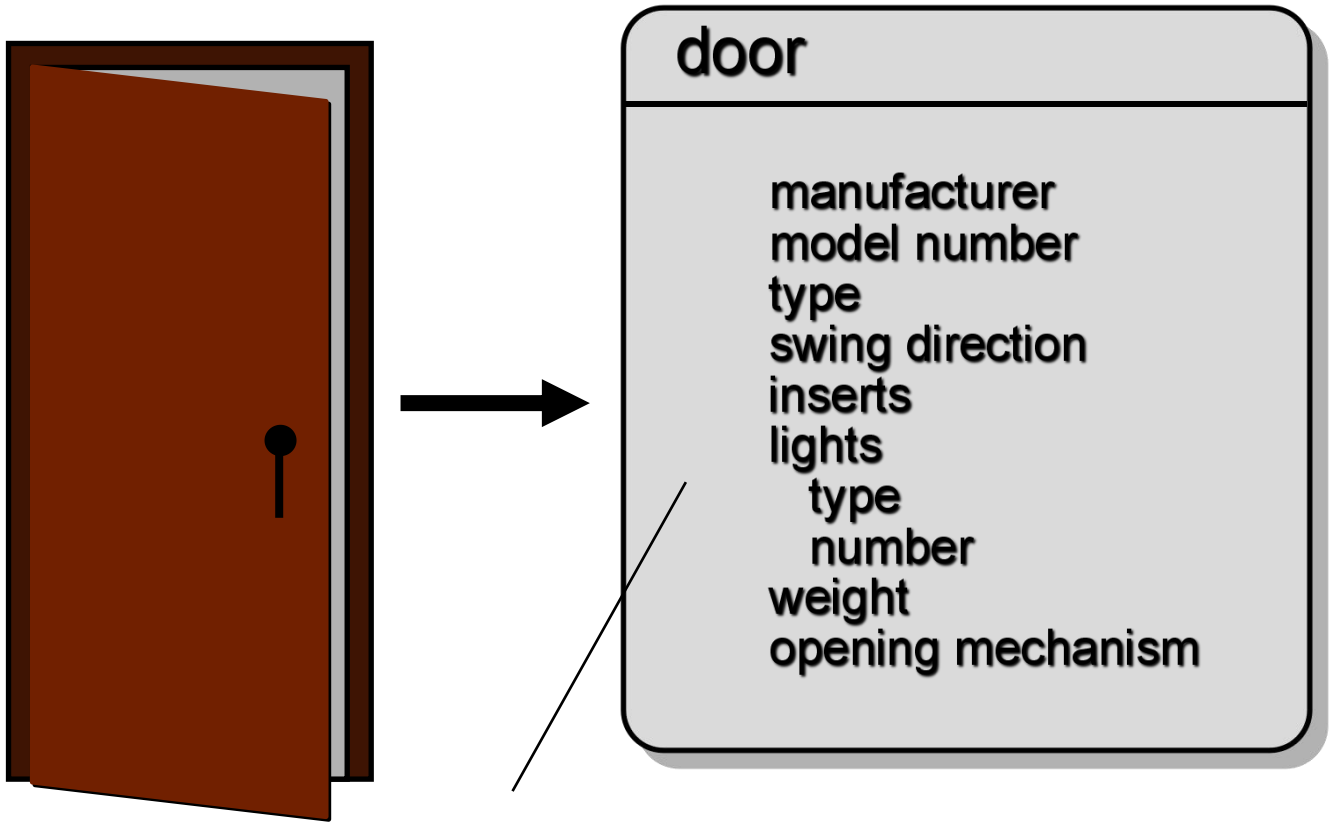
Data

12.3 Procedural Abstraction



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

12.3.1 Data Abstraction



12.3.2 Architecture

Architecture design specification should include:

- **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. For example, objects are packaged to encapsulate both data and the processing 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.

12.3.3 Design Pattern

Enables a designer to determine:

1. Whether the pattern is applicable to the current work
2. Whether the pattern can be reused (saving time)
3. Whether the pattern can serve as a guide for developing a similar, but functionally or structually different pattern

(see Chapter16)

Take a break

Five minutes

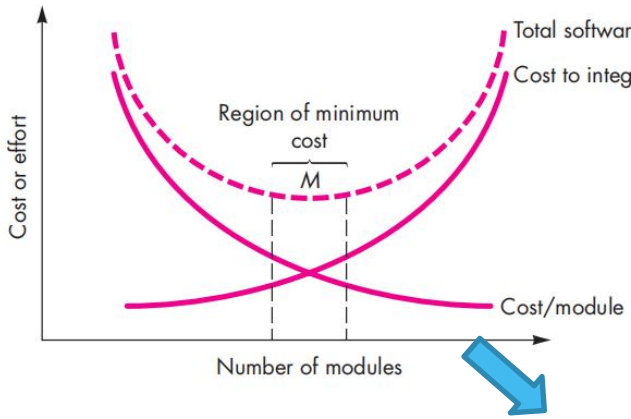
12.3.4 Separation of Concerns

- 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.

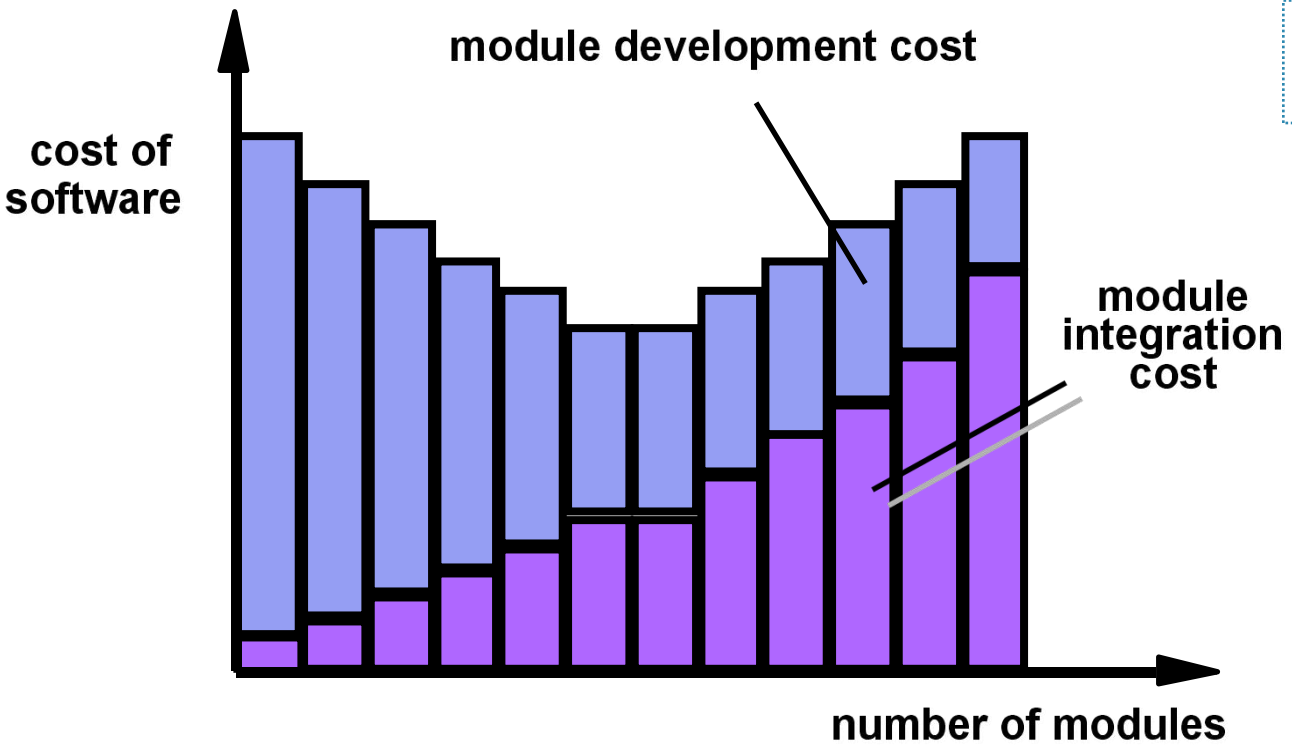
12.3.4 Modularity

- Modularity is the single attribute of software that allows a program to be intellectually manageable.
- 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.

12.3.5 Modularity: Trade-offs



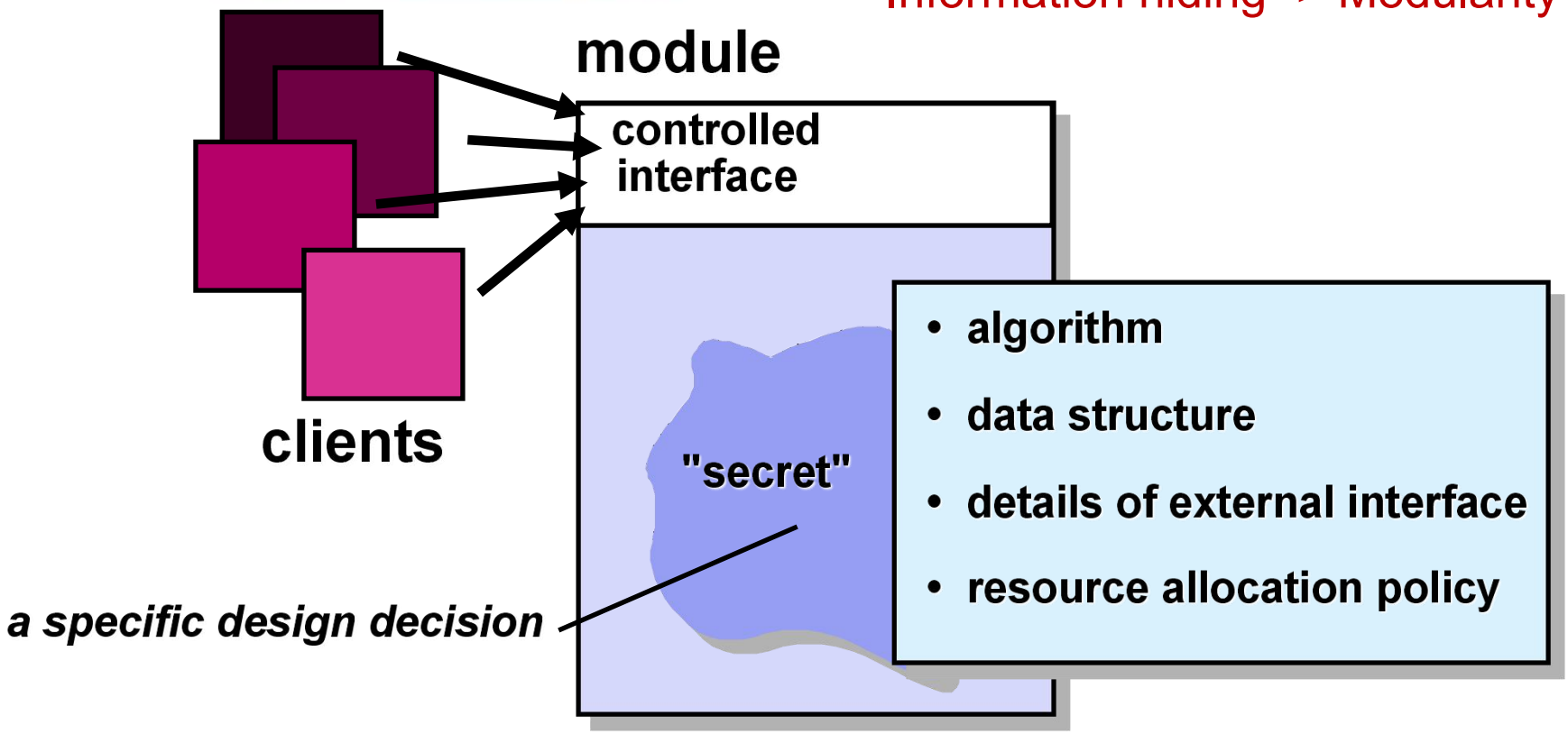
for a specific software design?



Predict the number of modules?

12.3.6 Information Hiding

Information hiding -> Modularity



Defining a set of independent modules that communicate with one another only that information necessary.

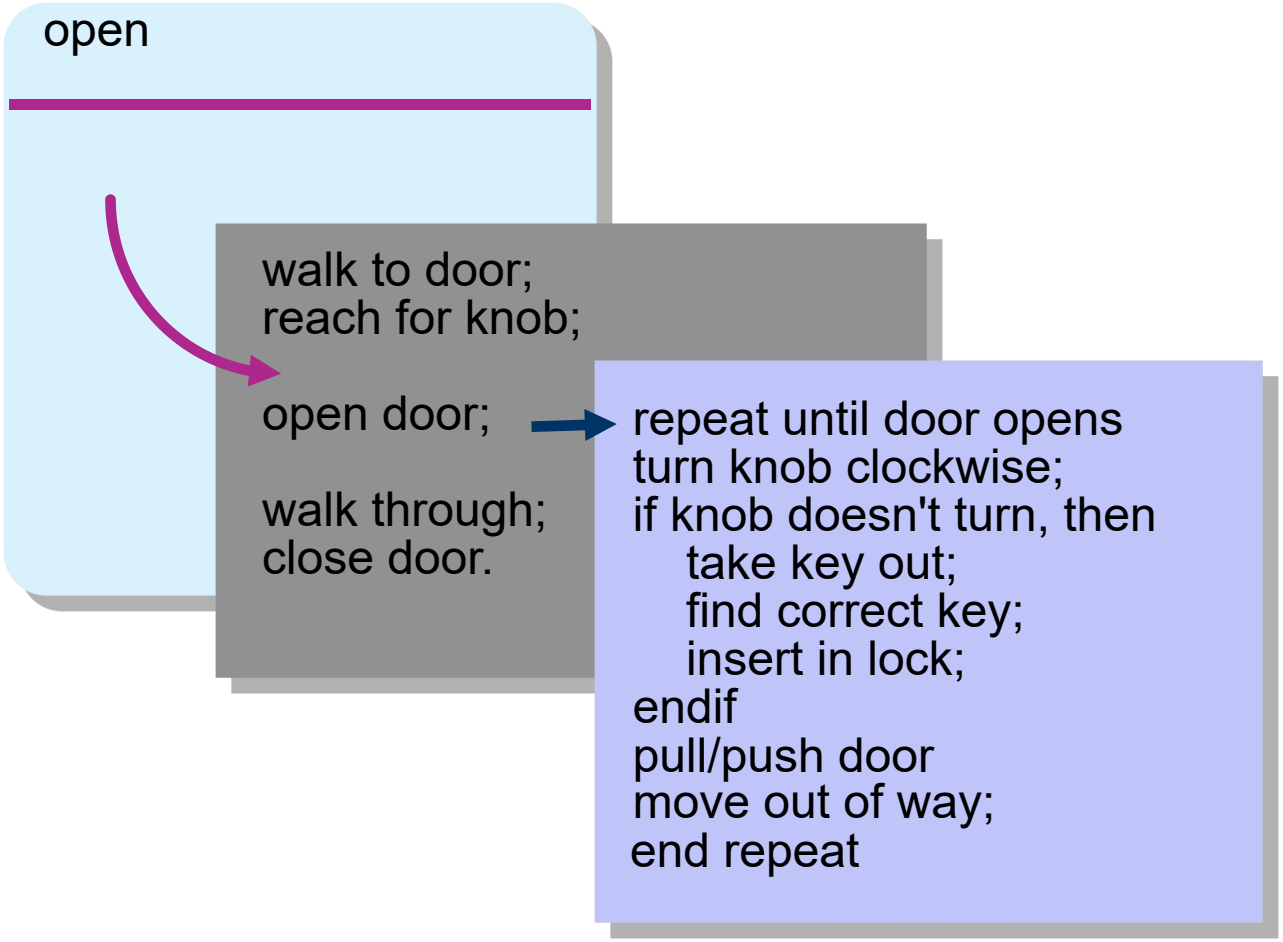
12.3.6 Why Information Hiding?

- 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

- 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 within a subsystem.
  - A cohesive module performs a single task, requiring little interaction with other components . Stated simply, a cohesive module should (ideally) do just one thing.
- ***Coupling*** is an indication of the relative interdependence between two subsystems.
  - 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.

12.3.8 Stepwise Refinement



Take a break

Five minutes

Review

- Abstraction (data and process)
- Architecture (structure)
- Design Pattern
- Modularity
- Independence (*Cohesion , Coupling*)
- Information Hiding
- Stepwise Refinement

A: Low Cohesion, Low Coupling
B: Low Cohesidon, High Coupling
C: High Cohesion, Low Coupling
D: High Cohesion, High Coupling

12.3.9 Aspects—An Example

An *aspect* is a representation of a cross-cutting concern.

For SafeHomeAssured.com WebApp.

- **Requirement *A*** : 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*****.** This requirement is applicable for all functions.
- Design refinement : B *cross-cuts* A. (B -> A)
- **Therefore, *B*, *a registered user must be validated prior to using* SafeHomeAssured.com, is an aspect of the *SafeHome* WebApp.**

12.3.9 Aspects - Concept

- Consider two requirements, *A* and *B*. Requirement *A* *crosscuts* requirement *B*: if a software decomposition has been chosen in which *B* cannot be satisfied without taking *A* into account.. (A -> B)
- An *aspect* is a representation of a cross-cutting concern.
- An aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components

12.3.10 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.

•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.

12.3.11 OO Design Concepts

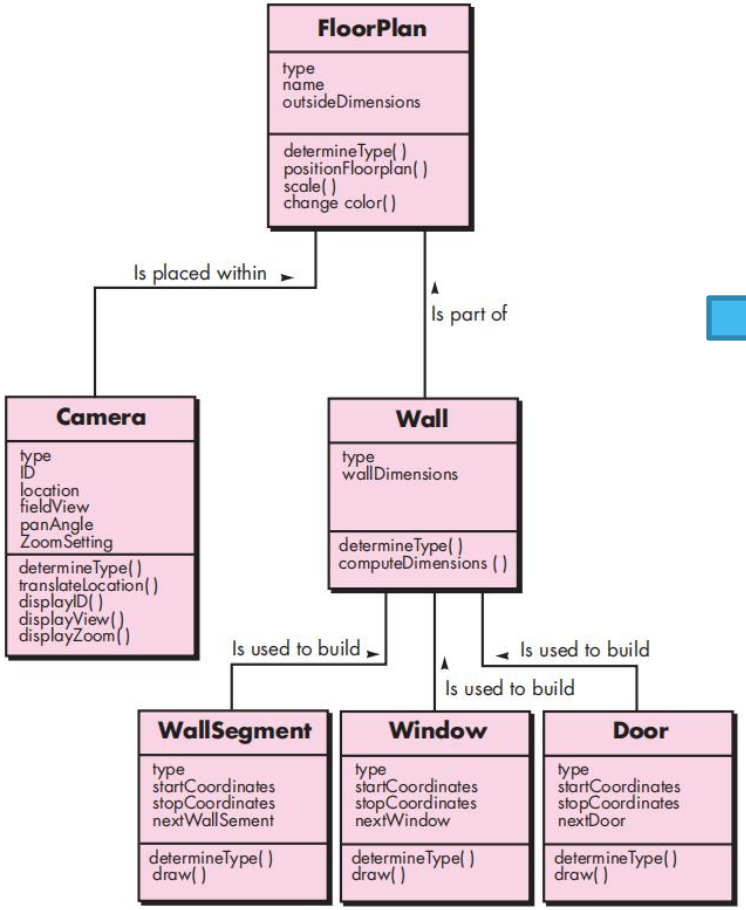
- Design classes
  - Entity classes (Object)
  - Boundary classes (Interface)
  - Controller classes (Control)
- Inheritance — all responsibilities of a superclass is immediately inherited by all subclasses
- Messages — stimulate some behavior to occur in the receiving object
- Polymorphism — a characteristic that greatly reduces the effort required to extend the design

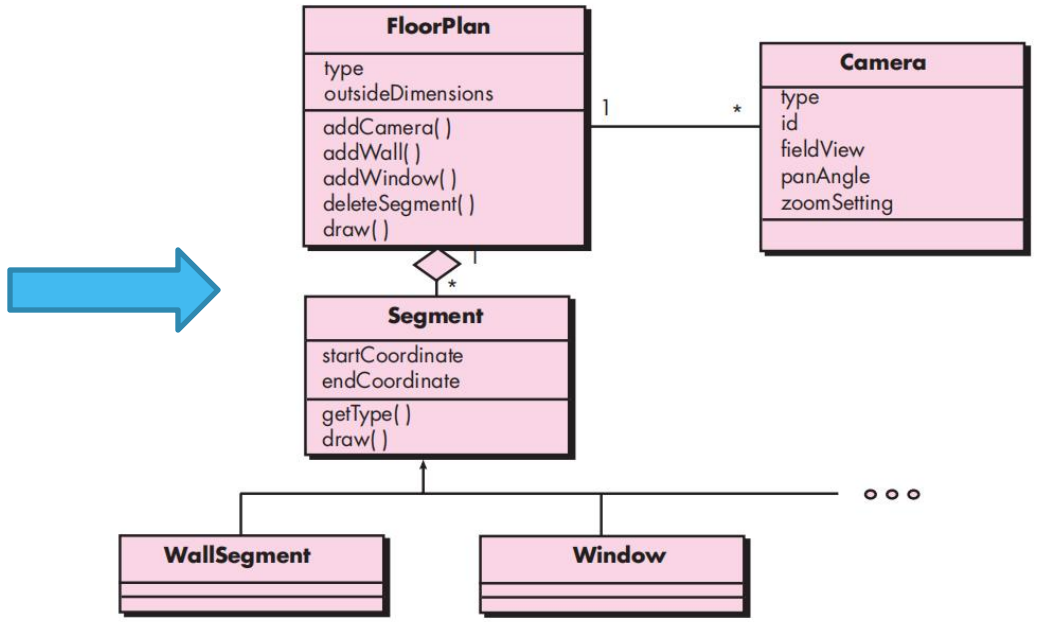
12.3.12 Design Classes

Analysis class => design class (5 types)

- **User interface classes** define all abstractions that are necessary for human computer interaction (HCI).
- **Business domain classes** are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- **Process classes** implement lower-level business abstractions required to fully manage the business domain classes.
- **Persistent classes** represent data stores (e.g., a database) that will persist beyond the execution of the software.
- **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

12.3.12 Design Classes





Design-oriented design class

Requirement-oriented analysis class

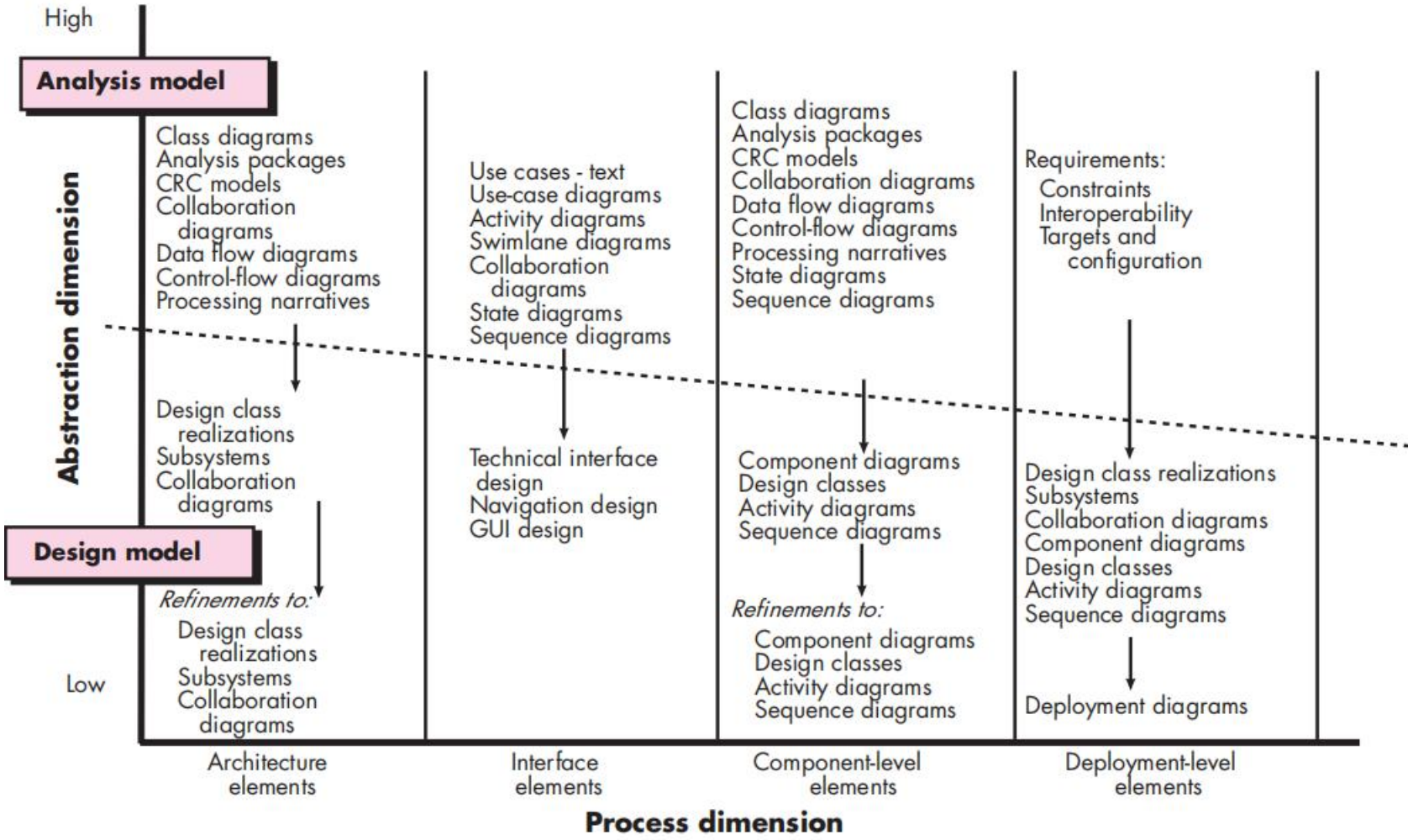
12.3.12 Design Class Characteristics

- 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

Take a break

30 seconds

12.4 The Design Model

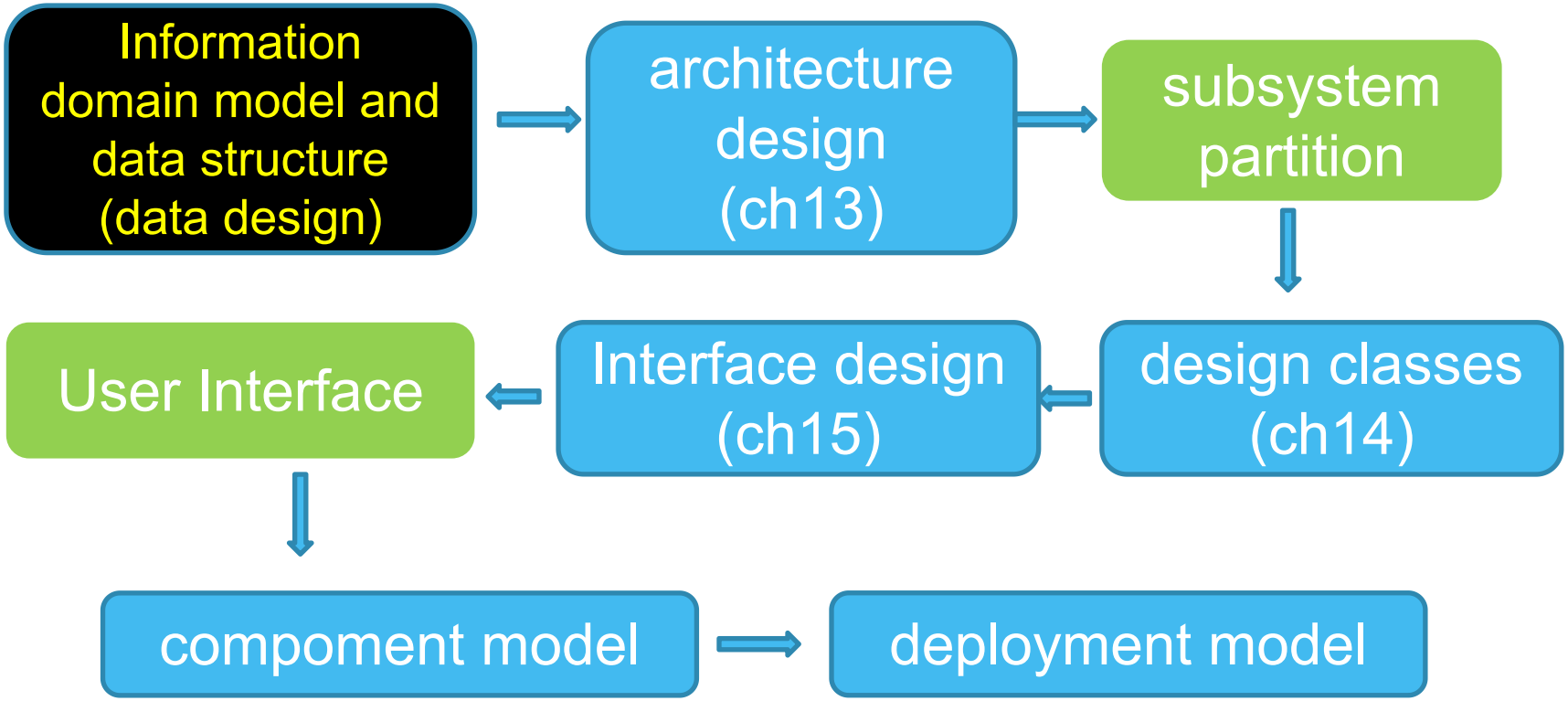


12.4 Design Model Elements

- 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

Evolution and Design

Tasks for Design :



12.4.1 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

12.4.1 What is a Data Object?

- 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.

12.4.1 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**

12.4.1 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 Elements

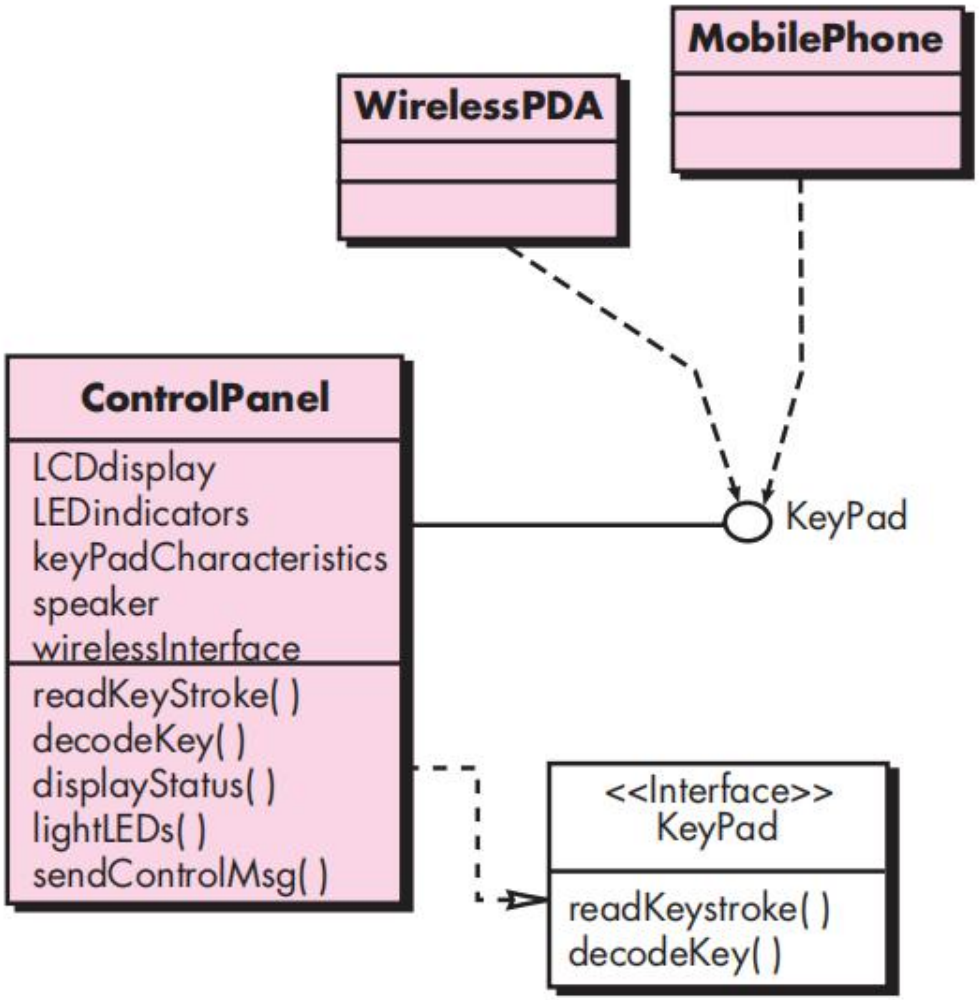
- The architectural model is derived from three sources:
  - information about the application domain for the software to be built;
  - specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
  - the availability of architectural patterns and styles .

Architectural == floor plan of a house

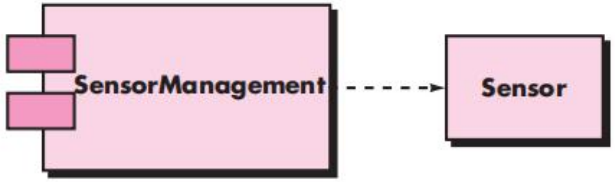
12.4.3 Interface Elements

- 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

12.4.3 Interface Elements



12.4.4 Component Elements

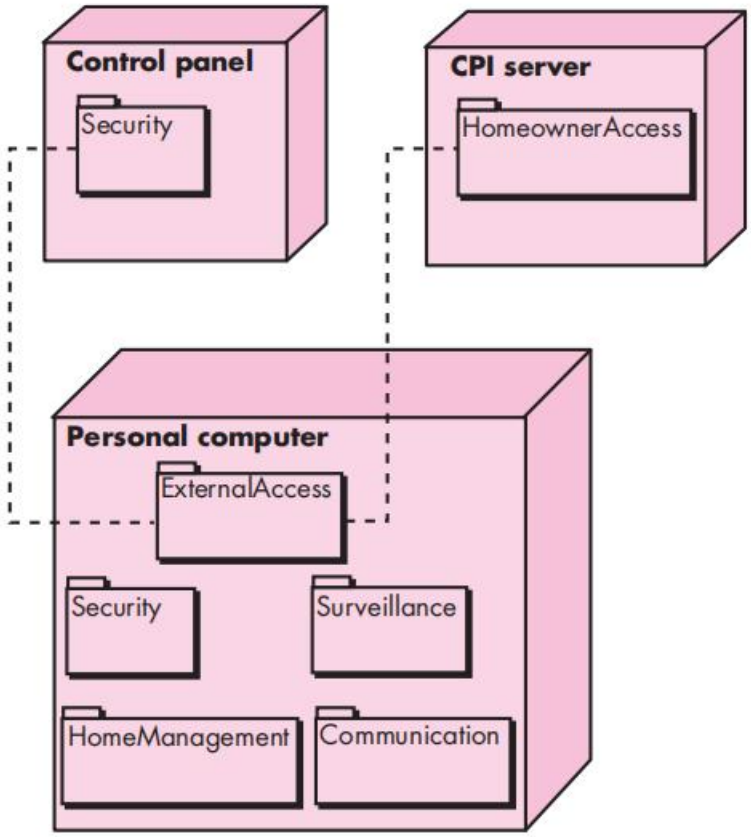


- Describes the internal detail of each component
- Defines
  - Data structures for all local data objects
  - Algorithmic detail for all component processing functions
  - Interface that allows access to all component operations
- Modeled using UML component diagrams, UML activity diagrams, pseudocode (PDL), and sometimes flowcharts

12.4.5 Deployment 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

12.4.5 Deployment Elements



Summary

- What is design? Taskset / Models?
  - principles, concepts, and practices
  - design quality
  - data/class design, architectural design, component-level design, interface design
- Fundamental Concepts of Design
  - abstraction, architecture, modularity, information hide, refinement, aspect, refactoring, OO design, etc
- UML
  - communication diagrams
  - Component diagram
  - Deployment diagram

THE END