Design Fundamentals

Dr Awais Majeed

Department of Software Engineering
Bahria University, Islamabad

Outline

Introduction

Design and SDLC

Design Concepts

Introduction

- Design is what almost every engineer wants to do
- A place where creativity rules

Design creates a representation or model of the software which provides details about software architecture, data structures, interfaces, and components that are necessary to implement the system.

Introduction ...

- Many of the concepts in Software Design are influenced by works in Civil/Building architecture
- The Roman architecture critic Vitruvius identified that well-designed buildings exhibit firmness, commodity and delight
- In software:
  - Firmness: A program should not have any bugs that inhibit its function
  - Commodity: A program should be suitable for the purposes for which it was intended
  - Delight: The experience of using the program should be pleasurable one
- The goal of the design is to produce a model or representation that exhibits firmness, commodity and delight.

What a design represents?

Design represents:

1. Architecture of the system
2. Interface that connects the software to end user
3. Interface that connects software to other systems and devices and to its own constitutent components
4. Software components that are used to construct the system are also designed.

Design Levels

Software Architecture

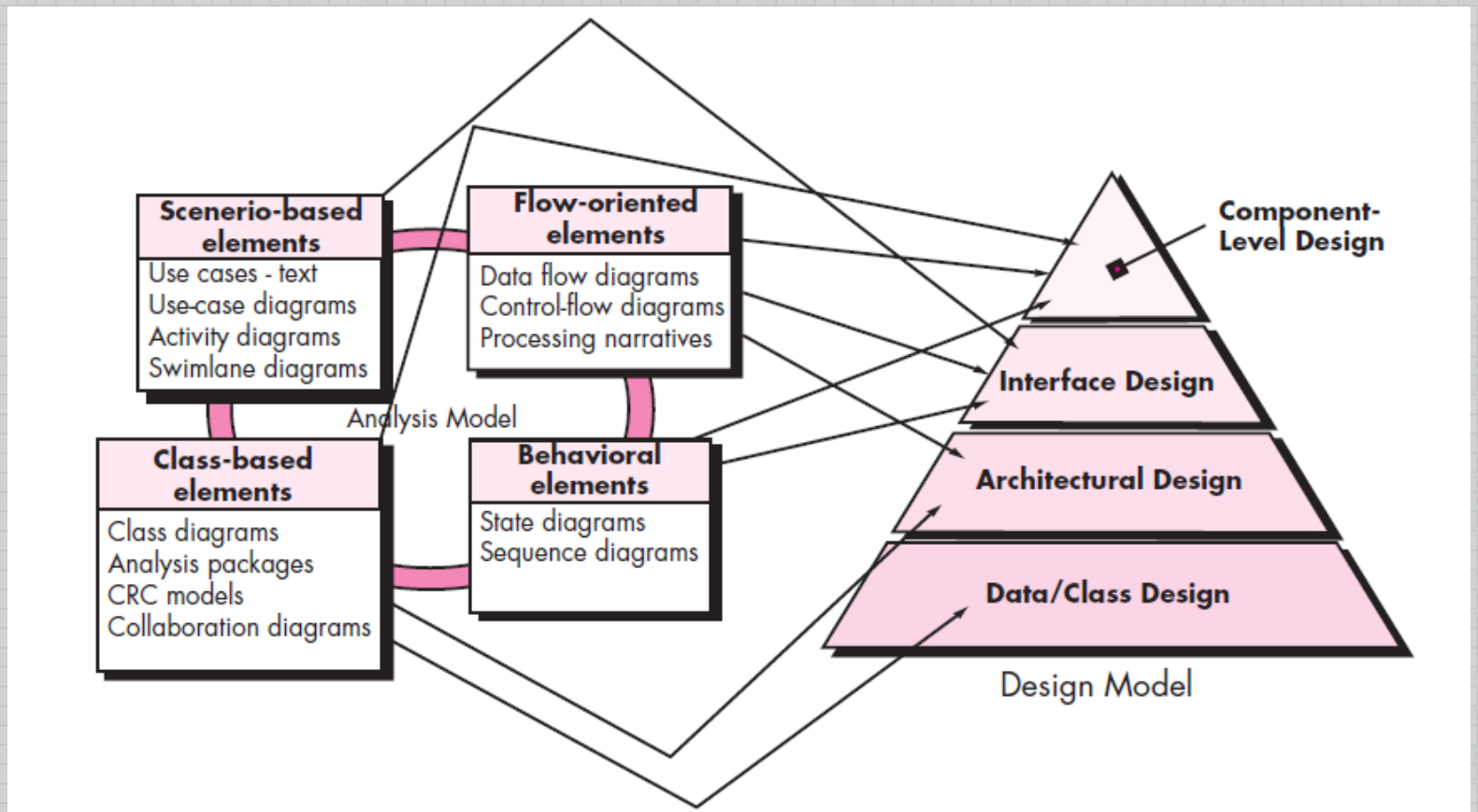
The overall structure of the software and the ways in which that structure provides conceptual integrity for a system

Detailed Design

Detailed Design describes the structure, properties and behaviour of the individual components in such a detail that these can be implemented by a developer easily.

Design in SDLC

- Design is the last activity within the modelling activity in SDLC and sets the stage for construction (code generation and testing).
- It translates the models created during requirements analysis into design models



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 I

- *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.

Quality Guidelines II

- *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.*

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" 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.

From Davis [DAV95]

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

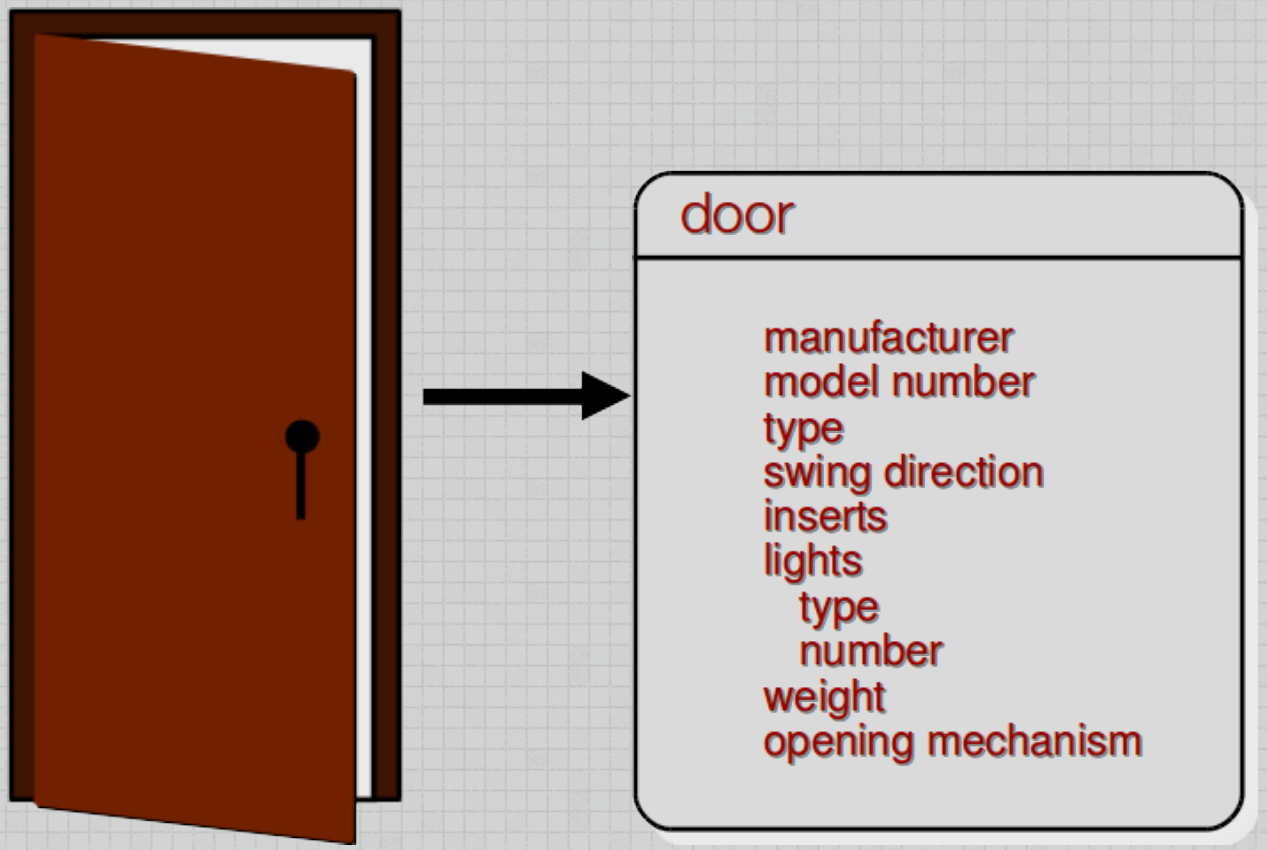
Aspects– a mechanism for understanding how global requirements affect design

Refactoring– a reorganization technique that simplifies the design

Design-Classes– provide design detail that will enable analysis classes to be implemented

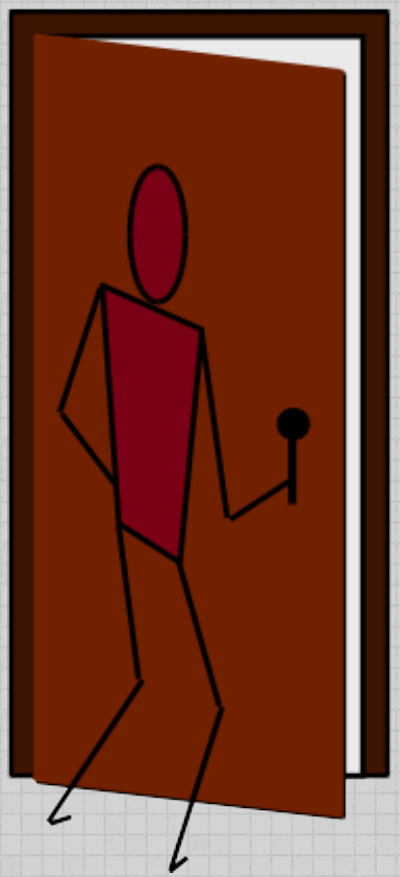
OO-DesignConcepts– Details in coming slides

Data Abstraction



Implemented as a data structure

Procedural Abstraction



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.

- **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

A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it.

- Pattern is "an idea that has been useful in one practical context and will probably be useful in others".
- Patterns are useful because these:
  - document existing, well-proven design experience.
  - provide common vocabulary and understanding for design principles
  - help in documenting software architectures.
  - support the construction of software with defined properties.
  - help you to build complex and heterogeneous software architectures.
  - help you to manage software complexity

Separation of Concerns

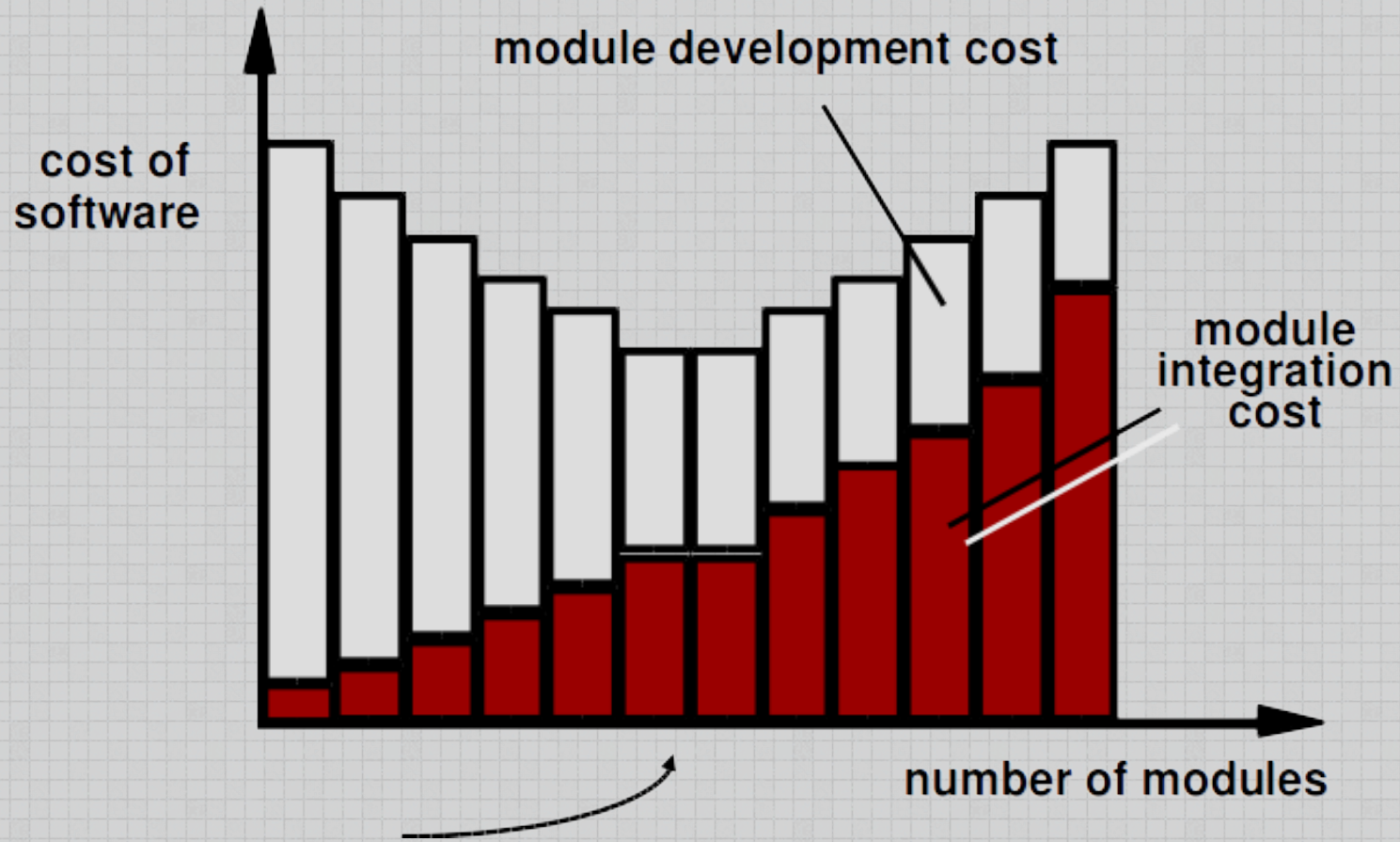
- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more 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?



Advantages of Modularity

- Easier to manage
- Easier to understand
- Reduces complexity
- Delegation / division of work
- Fault isolation
- Independent development
- Separation of concerns
- Reuse

Principles of Modular Design

How to decompose a software system into best set of modules?

- Information hiding
- Functional independence
- Cohesion
- Coupling

Information Hiding

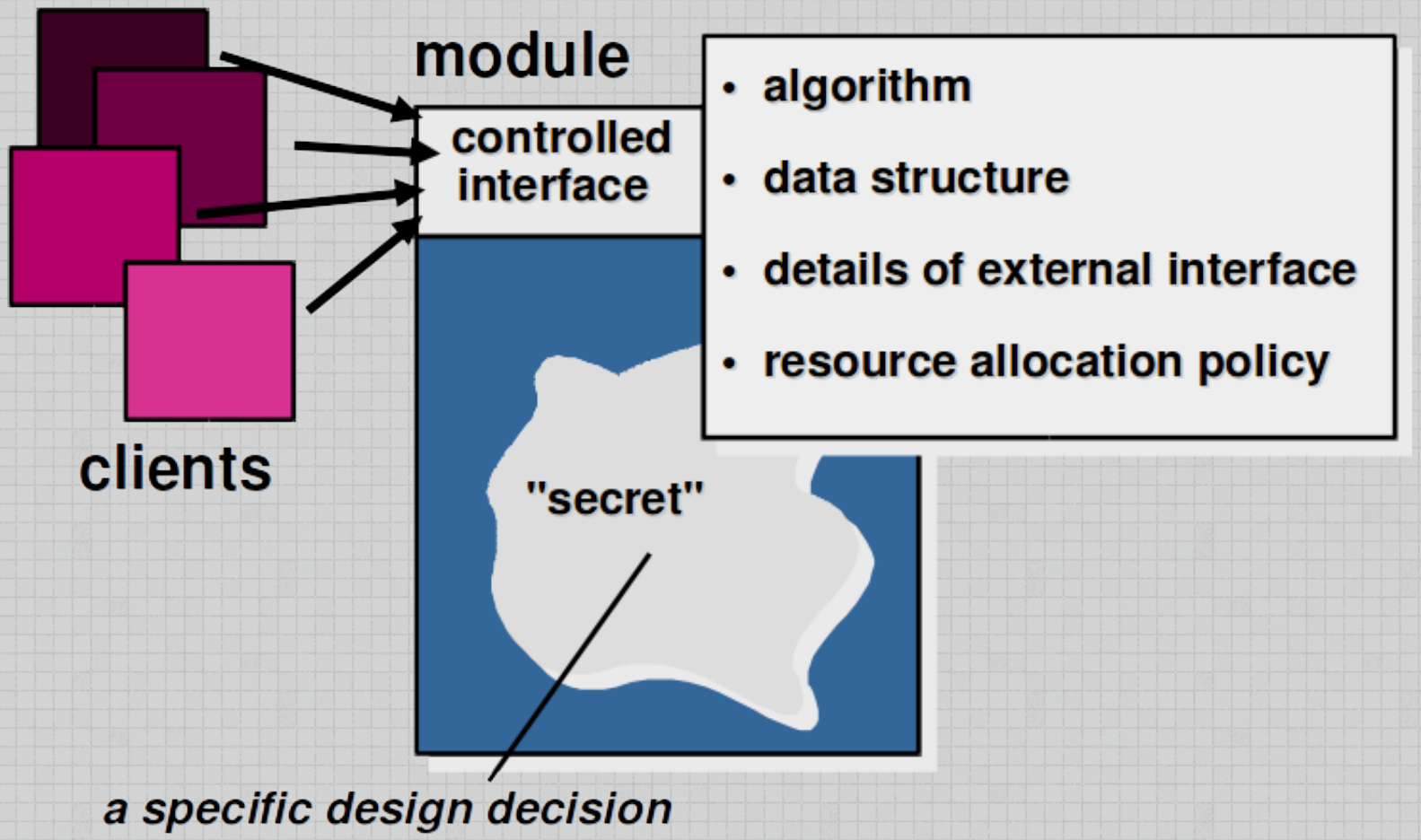
Design the modules in such a way that information (data & procedures) contained in one module is directly inaccessible to other modules.

- Modules to be characterized by *design decisions that are hidden* from all others.
- Independent modules.

Benefits:

- When modifications are required, it reduces the chances of propagating to other modules.

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

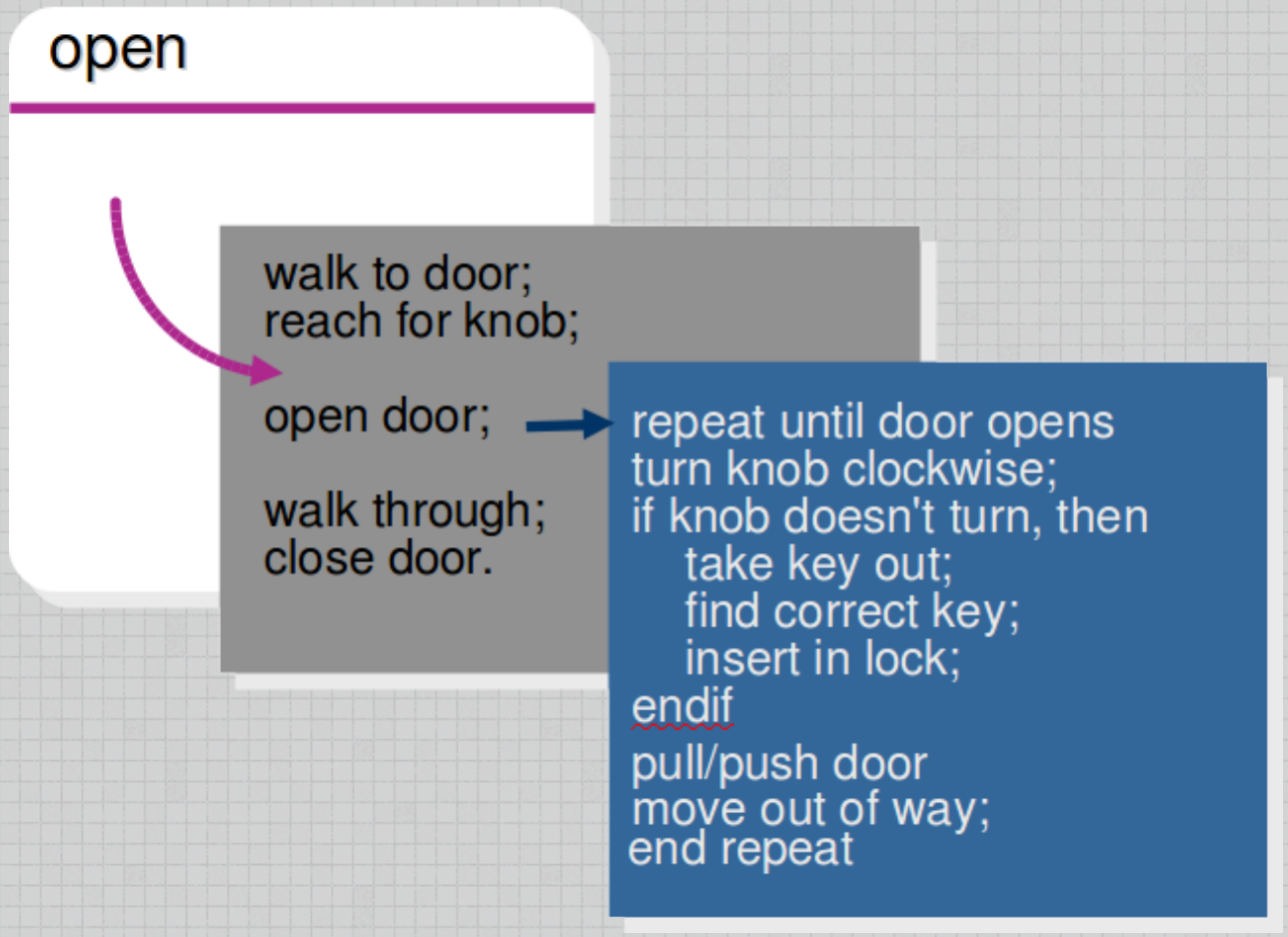
Information Hiding

In object-oriented programming, often it can be achieved with access specifiers.

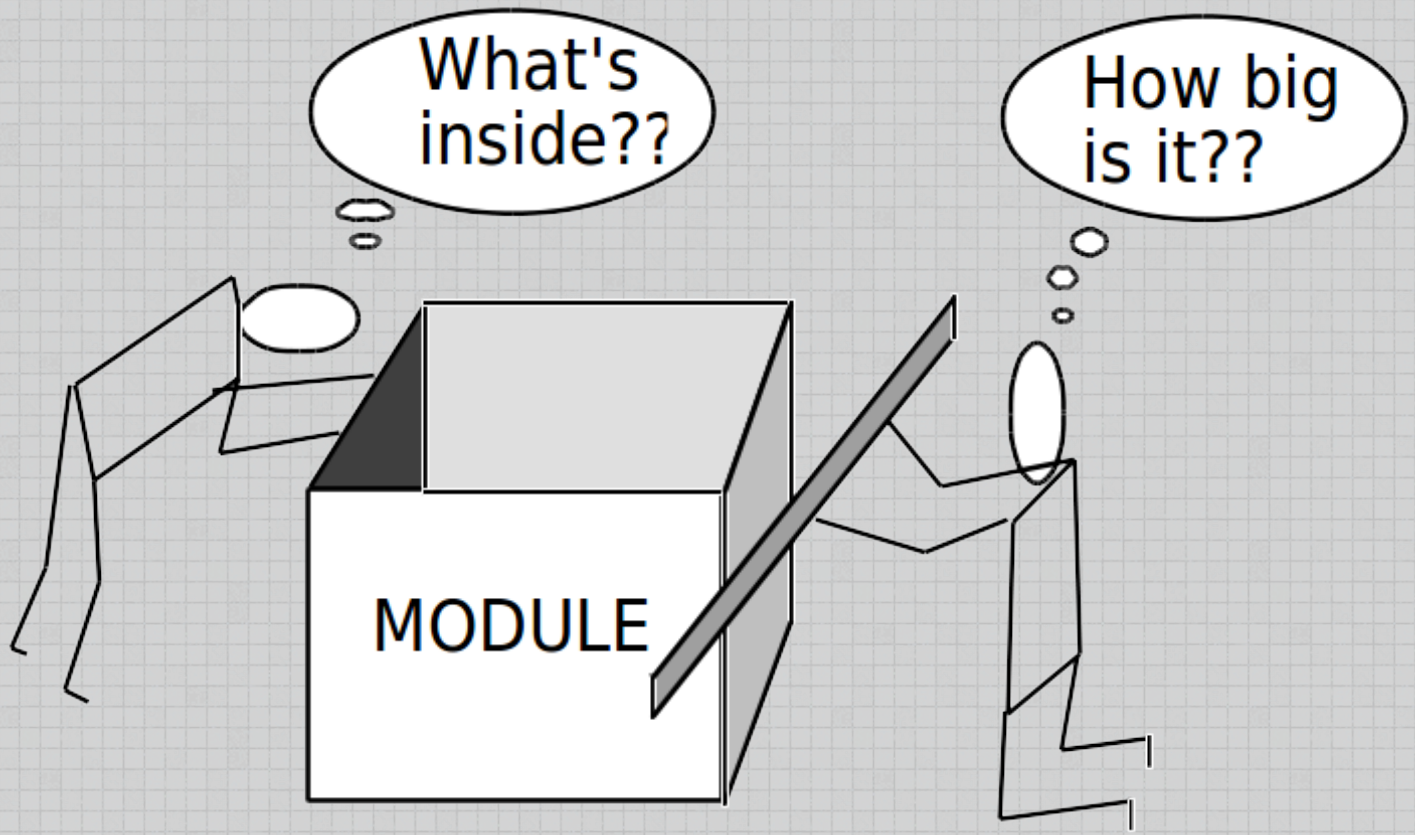
```
public class PersonalInfo{

        public String name;

        public String city;

        private int age;

        }
```

```
person1.age=25;
```

Stepwise Refinement



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.
- Each module addresses a specific sub function of requirements and has a simple interface when viewed from other parts of the system.
- Functions are compartmentalized and interfaces are simplified.
- **Benefits**-Error propagation is reduced and can lead to reusability of modules.
- Two indicators of functional independences
  - *Cohesion* is an indication of the relative functional strength of a module.
  - *Coupling* is an indication of the relative interdependence among modules.

Cohesion

Cohesion means that a module should(ideally) do one thing

- *It is an indication of the relative functional strength of the module.*
- A reflection of Information hiding and functional independence.
- A cohesive module performs a single task, requiring little interaction with other components in other parts of a program.
- High cohesion is good
  - Changes are likely to be local to a module
  - Easier to understand a module in isolation

Coupling

A measure of interconnection among modules in a software structure

- Coupling depends on the interface complexity between modules
- Depends on the interface complexity between the modules.
- Number of dependencies between modules.
- High coupling causes problems
  - Change propagation- ripple effect
  - Difficulty in understanding
  - Difficult reuse

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

- 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.

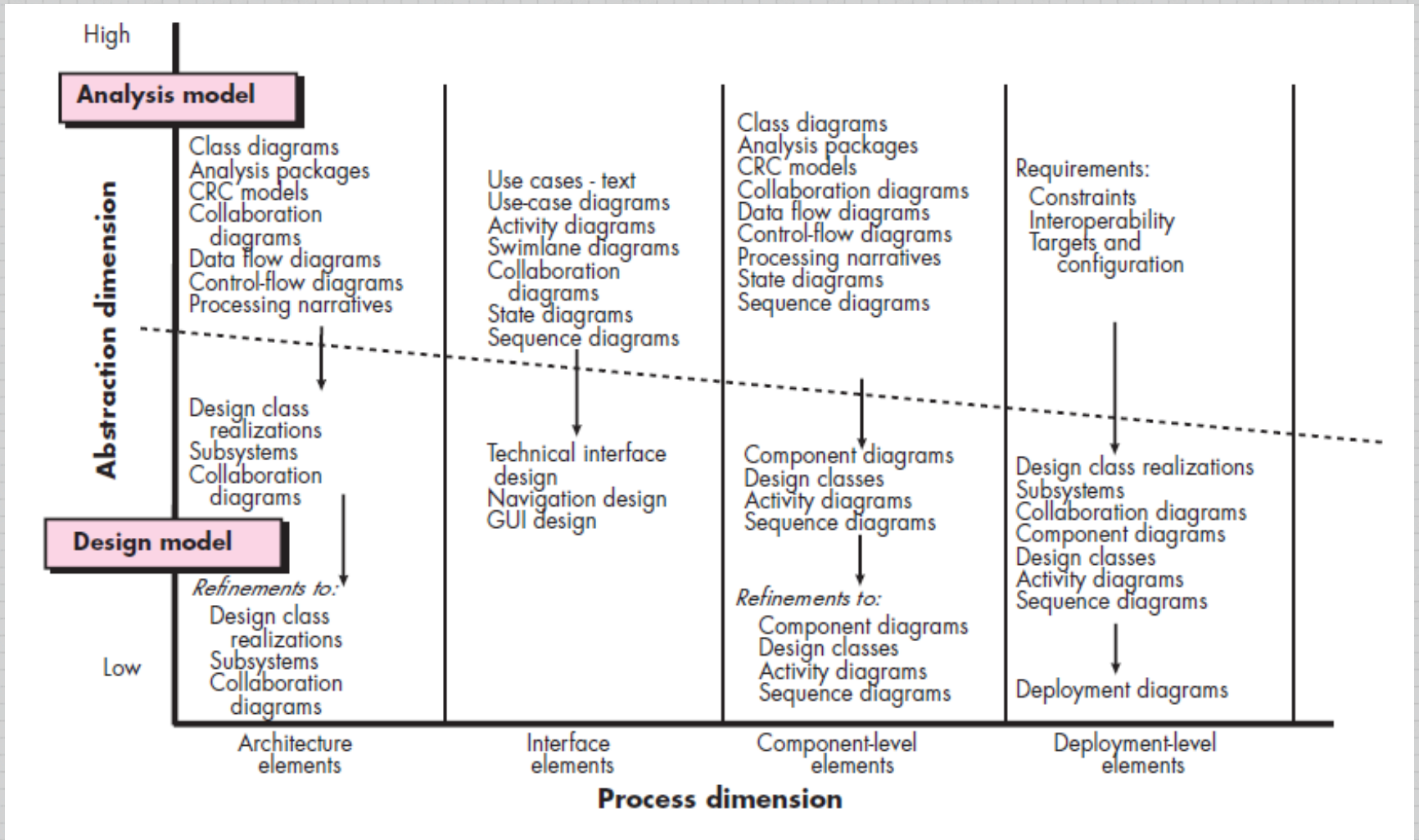
OO Design Concepts

- **Design classes**
  - Entity classes
  - Boundary classes
  - Controller classes
- **Inheritance**—all responsibilities of a superclass are 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 *design classes*
- Design classes refine the analysis classes by providing design details helpful in class implementation in the form of software infrastructure supporting the business solution.
- Five different types of design classes can be developed:
  1. User Interface classes: define all abstractions that are necessary for HCI and implement HCI using metaphors.
  2. Business Domain Classes: identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or more analysis classes.
  3. Process classes: implement lower-level business abstractions required to fully manage the business domain classes.
  4. Persistent classes: represent data stores(databases) that will persist beyond the execution of the software.
  5. 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.

The Design Model



Dimensions of the design model

Design Model Elements

- Data Design Elements
- Architectural Design Elements
- Interface elements
- Component-Level Design Elements
- Deployment-Level Desgin Elements

Data Design Elements

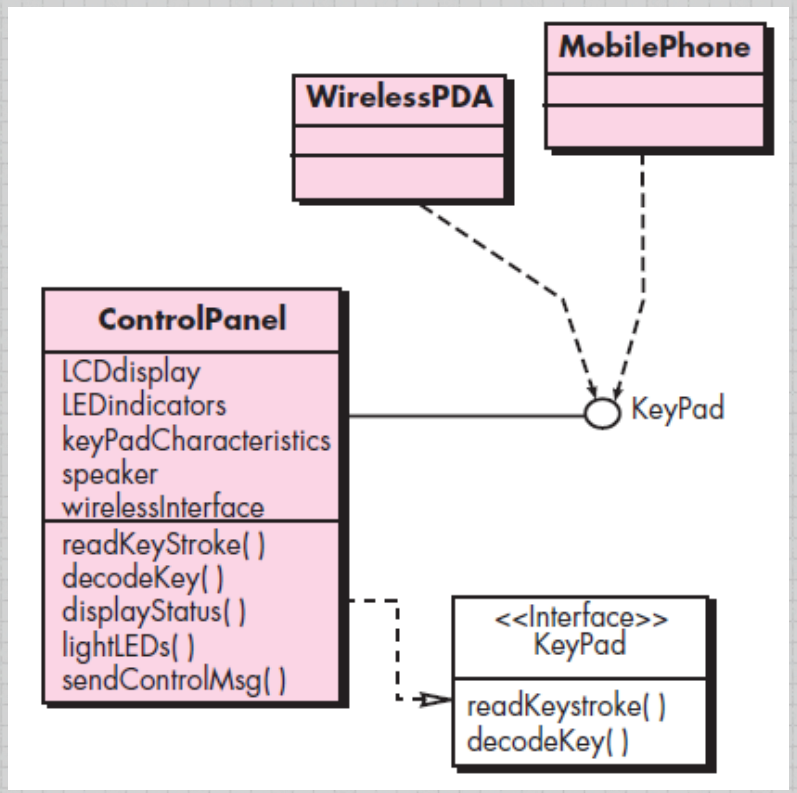
- **Data Design Elements**
  - Data design (data architecting) creates a model of the data and or information that is represented at a high level of abstraction (the customer/user view of data).
    - Data model –> database architecture, data warehouse
  - This data model is then refined into more implementation-specific representations.
    - Data model –> data structures

Architectural Design Elements

- Architectural design for software is equivelent to the floor plan of a house.
- Architectural design elements give us an overall view of the software.
- 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
  - The availability of architectural patterns and styles.

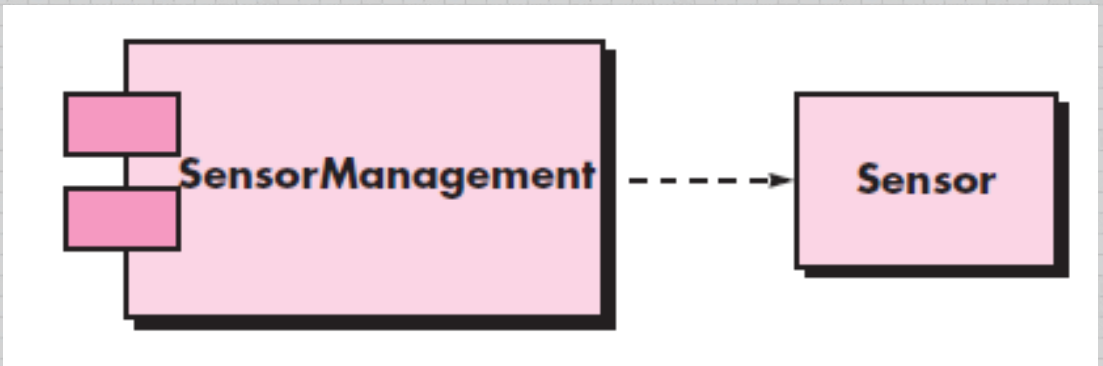
Interface Design Elements

- Different types of interfaces are considered:
  - 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-Level Design Elements

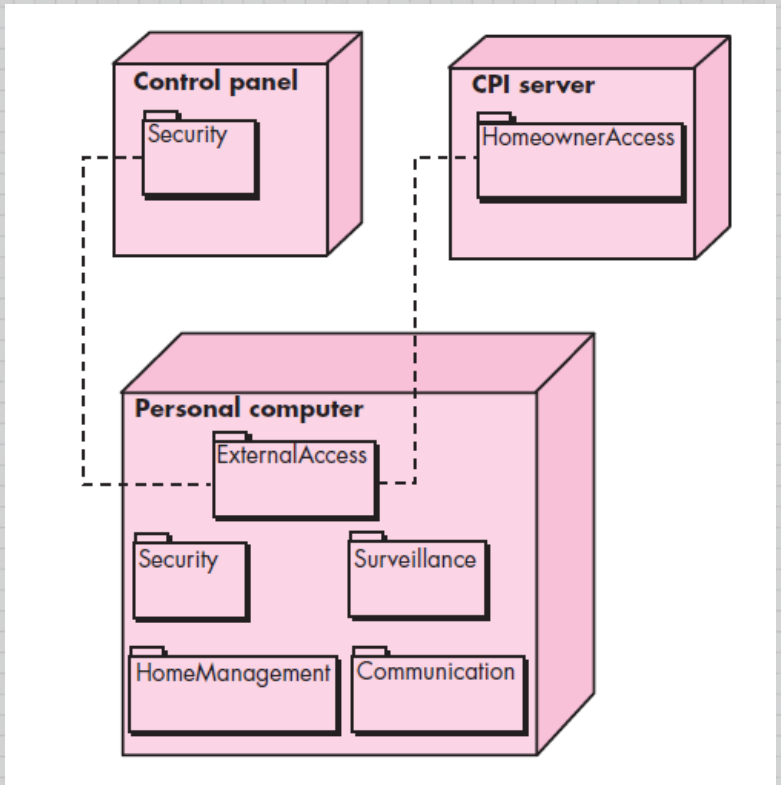
- The component-level design for software fully describes the internal detail of each software component.
- To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).
- Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure below:



- The design details of a component can be modeled at many different

Deployment-Level Design Elements

- Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.



Design Errors I

Incorrectness

- The design does not meet the users' requirements on its functionality and features.

Inconsistency

- Inconsistency is where a design does not work
  - Conflicting assumptions about functionality, data etc. by different design decisions/statements

Design Errors II

Ambiguity

- Different interpretations or ambiguous design specification
  - Ambiguity causes errors in the implementation of the design due to inconsistent interpretations made in the implementation process.

Inferiority

- The design does not address quality requirements adequately.
  - Typical inferior quality problems include inefficiency and inflexibility.

Conclusion

1. Design is an iterative process, it can't be correct or complete in 1st attempt.
2. You have to decide when to stop designing and move to implementation
3. Design is a creative process
4. Completeness of design upfront might not be possible in certain situations…

Self Study & References

1. Chapter 12 (Design Concepts) in “Software Engineering: A Practitioner’s Approach” 8th Ed by Pressman.