Design Engineering

Design Engineering

- Course Objective : To understand the design concepts and design engineering

- Course Outcomes : Able to demonstrate the concepts of design engineering

Contents

- Design Process, Design Concepts
- The Design Model: Data Design, Architectural, interface Design Elements.
- Agile Design: An Introduction to Agile Design, Phases and Life cycle of Agile,
- Agile design principles, Agile design methodology: benefits and conditions.
- Introduction to Scrum, Extreme Programming, JIRA.

Design and Quality

- McGlaughlin [McG91] suggests three characteristics that serve as a guide for the evaluation of a good design:

- **the design must implement all of the explicit requirements**

- **the design must be a readable, understandable guide**

- **the design should provide a complete picture of the software**

- *Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?*

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.**

Quality Attributes

- Hewlett-Packard [Gra87] developed a set of software quality attributes that has been given the acronym *FURPS—functionality, usability, reliability, performance, and supportability*. The FURPS quality attributes represent a target for all software design:

- ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

- ***Usability*** is assessed by considering human factors, consistency, and documentation.

Quality Attributes

3. ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

4. ***Performance*** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.

5. ***Supportability*** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, *maintainability*—and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, the ease with which a system can be installed, and the ease with which problems can be localized.)

Design Concepts

A set of fundamental software design concepts has evolved over the history of software engineering.

Each helps you answer the following questions:

1. What criteria can be used to partition software into individual components?
2. How is function or data structure detail separated from a conceptual representation of the software?
3. What uniform criteria define the technical quality of a software design?

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

Data abstraction

- In software engineering, data abstraction refers to the process of simplifying large data structures into smaller, more manageable chunks. This allows programmers to concentrate on the data's most salient features without being distracted by complexity.

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

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

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.

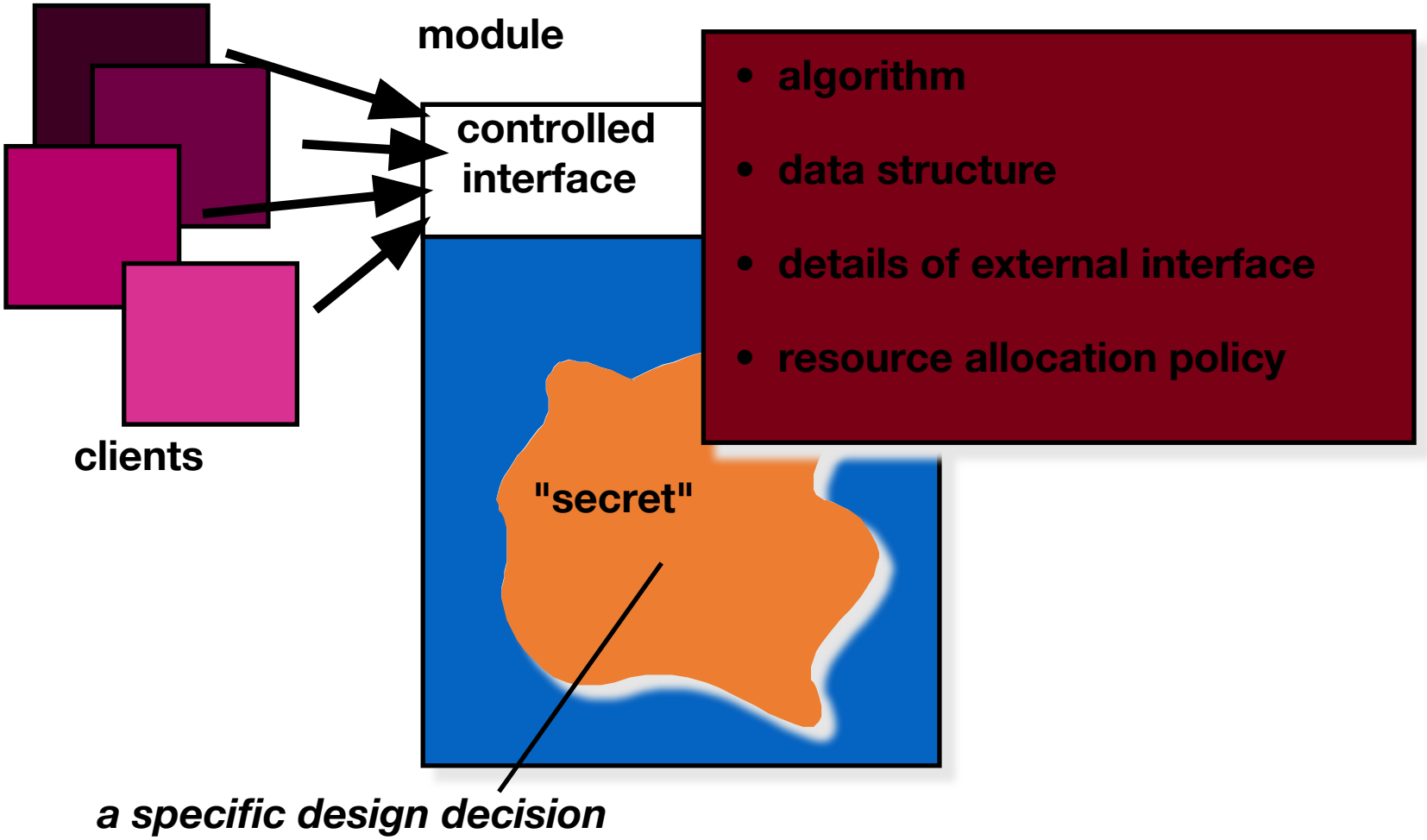
Modularity

- "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.

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

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 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*** is an indication of the relative interdependence among modules.
  - 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.

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.

- 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 (M)
  - Boundary classes (V)
  - Controller classes (C)

- **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

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.

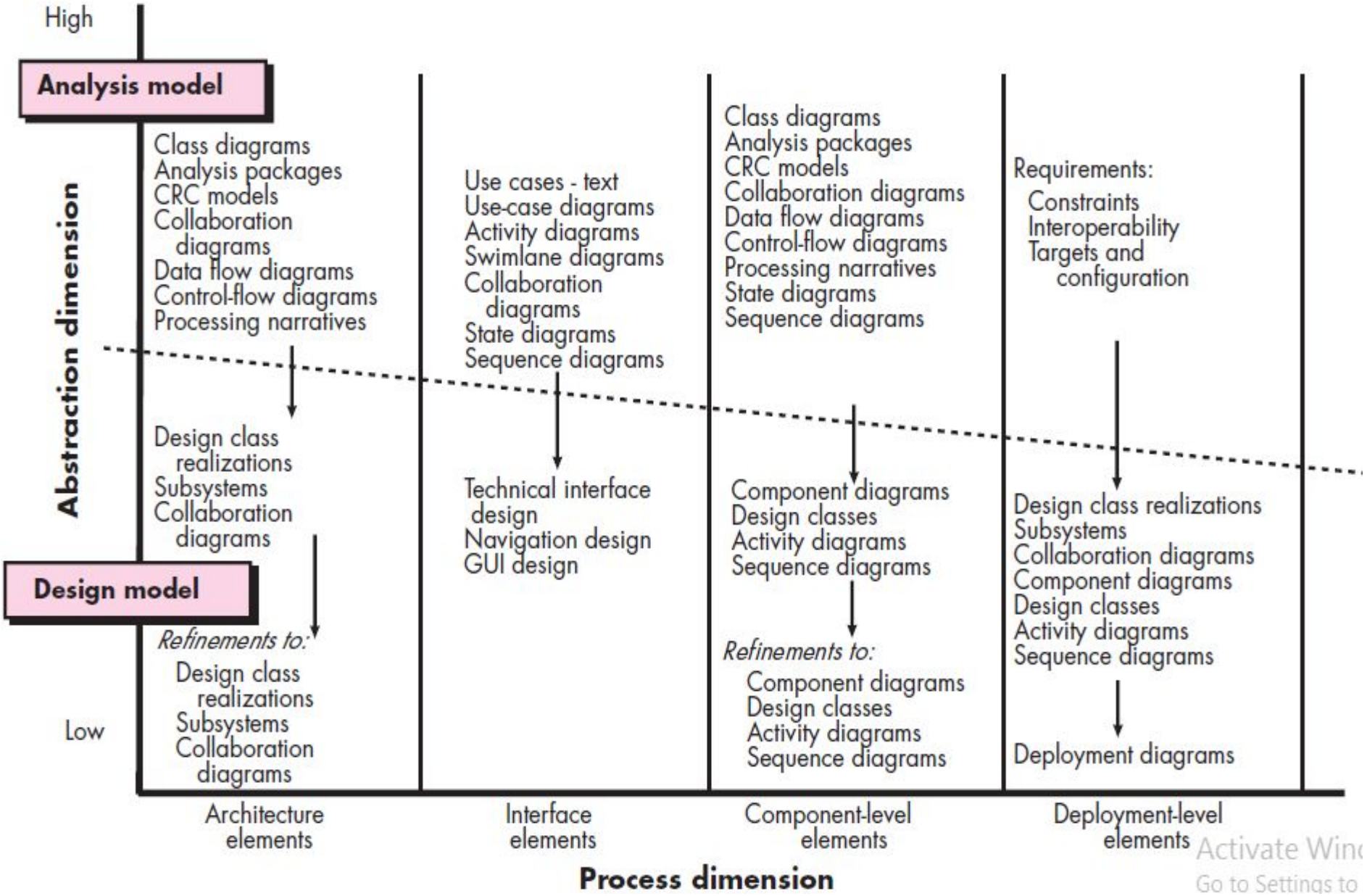
- **Controller classes** are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

Design Model

- The Design Model can be viewed in two different dimensions :

1. **Process Dimension** : It indicates the evolution of the design model as design task are executed as part of the software process
2. **Abstraction Dimension :** It Represents the level of detail as each element of analysis model is transformed into a design equivalent and then refined iteratively.

The Design Model



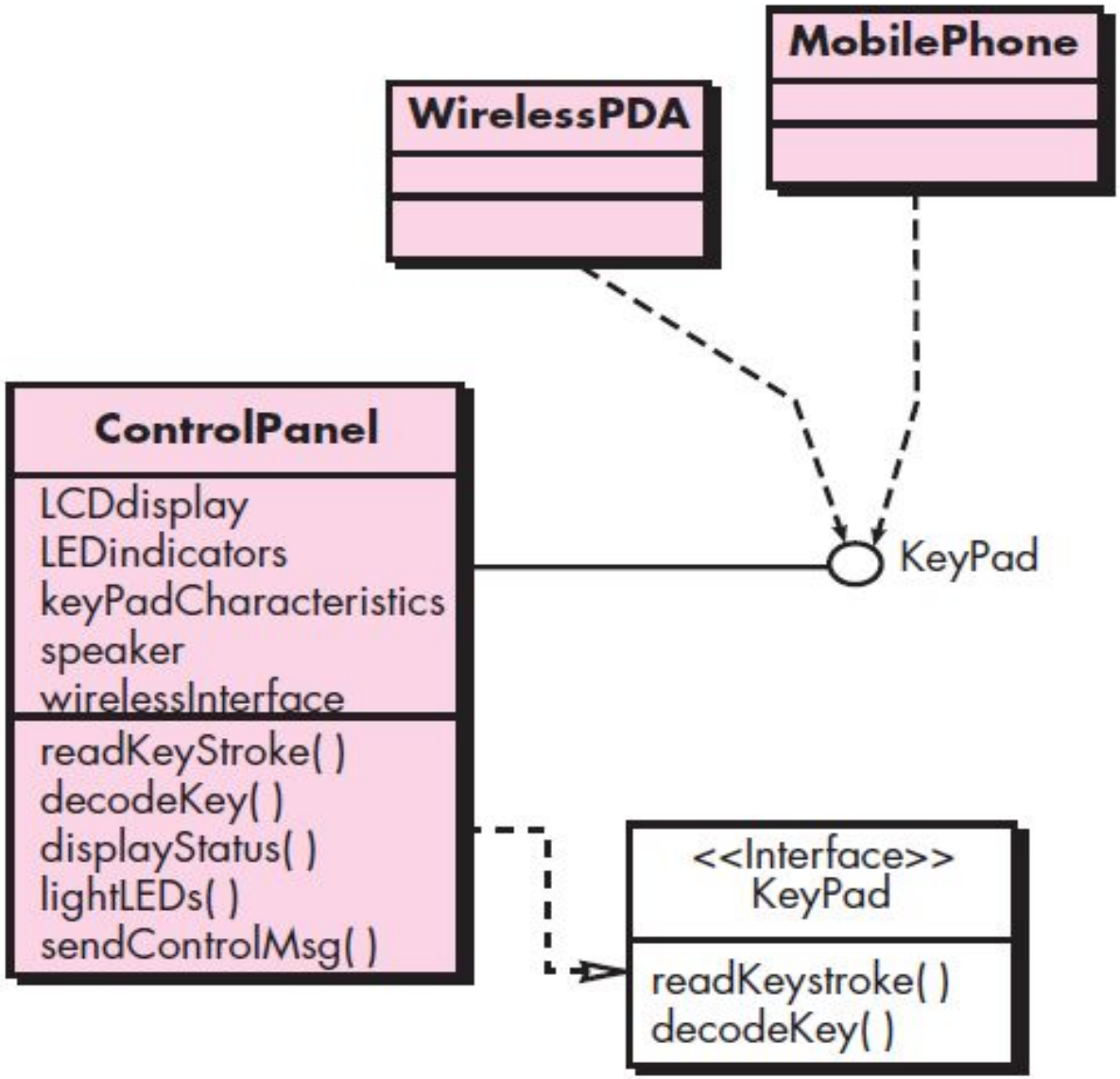
Design Model Elements

- **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”
- **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**

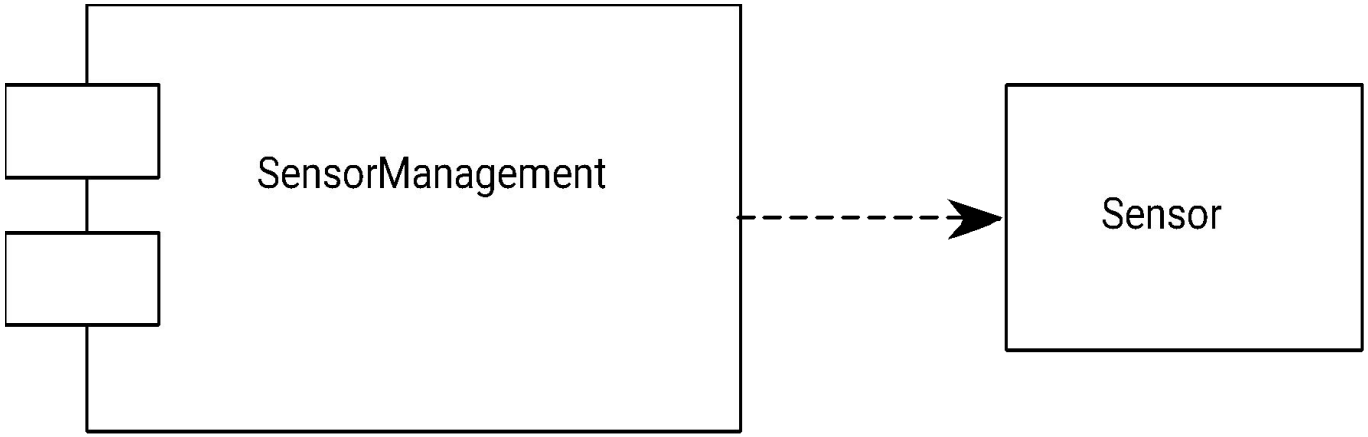
Architectural Elements

- 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.**

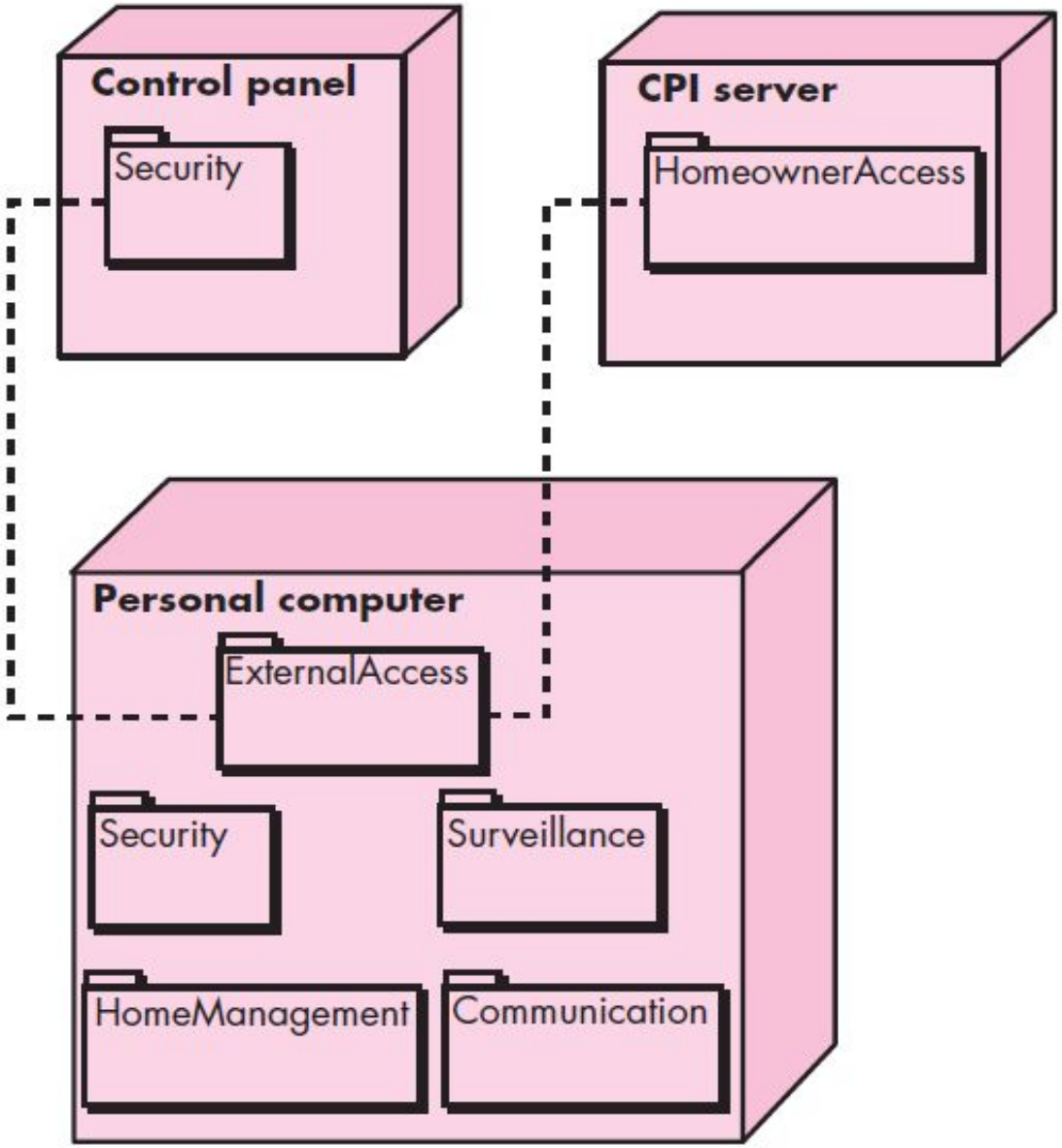
Interface Elements



Component Elements



Deployment Elements



Summary

- The intent of software design is to apply a set of principles, concepts, and practices that lead to the development of a high-quality system or product.
- The goal of design is to create a model of software that will implement all customer requirements correctly and bring delight to those who use it.
- The design process moves from a “big picture” view of software to a more narrow view that defines the detail required to implement a system.
- The process begins by focusing on architecture. Subsystems are defined; communication mechanisms among subsystems are established; components are identified, and a detailed description of each component is developed. In addition, external, internal, and user interfaces are designed.

Summary

- The design model encompasses four different elements.
- The ***architectural element*** uses information derived from the application domain, the requirements model, and available catalogs for patterns and styles to derive a complete structural representation of the software, its subsystems, and components.

- ***Component level elements*** define each of the modules (components) that populate the architecture.

- Finally, ***deployment-level design*** elements allocate the architecture, its components, and the interfaces to the physical configuration that will house the software.