# Design Concepts

- **Modeling: Chapter 12**

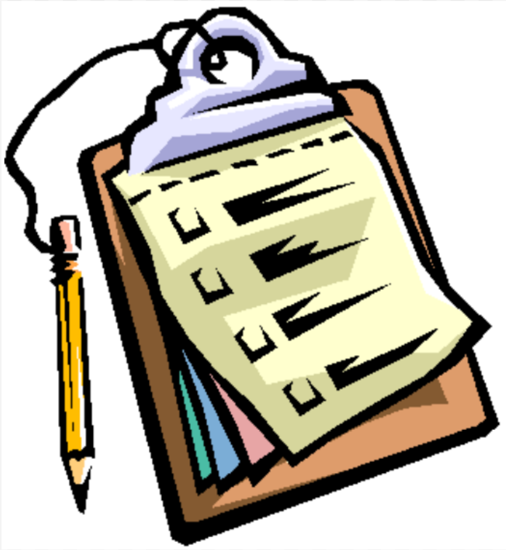
*Slide Set to accompany*

*Software Engineering: A Practitioner’s Approach, 8/e*

**by Roger S. Pressman and Bruce R. Maxim**

# Agenda

- Design Overview
- Design Concepts
- Design Models



# *Design Overview*

# Requirements, Design, and Implementation

- Requirements (Analysis) Model
  - Model of “**what** we need?”
  - Problem space
- Design Model
  - Model of “**how to** create or implement?”
  - Solution space
- Software Product (Implementations, Programs, …)
  - Actual creation (no model)

# Software Design

- Software design encompasses the set of **principles**, **concepts**, and **practices** that lead to the development of a high-quality product.
- Design principles establish an overriding philosophy that guides you in the design work
  - E.g., The design should exhibit uniformity and integration
- Design concepts must be understood before the mechanics of design practice apply
  - E.g., modularity, pattern, …
- Design practices lead to the creation of various software models that guide the construction activity

# Software Design Manifesto

- Mitch Kapor, presented a "software design manifesto" in *Dr. Dobbs Journal.* He said:
  - **Well-designed software programs should exhibit:**

1. *Firmness:* A program should not have any bugs that inhibit its function.
2. *Commodity:* A program should be suitable for the purposes for which it was intended.
3. *Delight:* The experience of using the program should be pleasurable one.

# Design and Quality

1. The design must cover **all the requirements**
   - Both functional and non-functional requirements
2. The design must be a **readable** and **understandable**
   - For those who generate code, test and subsequently support the software
3. The design should provide **a complete picture** of the software
   - Addressing the informationl, functional, and behavioral domains

# Design Principles

- The design process should not suffer from ‘tunnel vision.’
  - There are always alternative design solutions
  - The best designers consider all (or most) of them before settling on the final design model.
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- …

***From Davis [DAV95]***

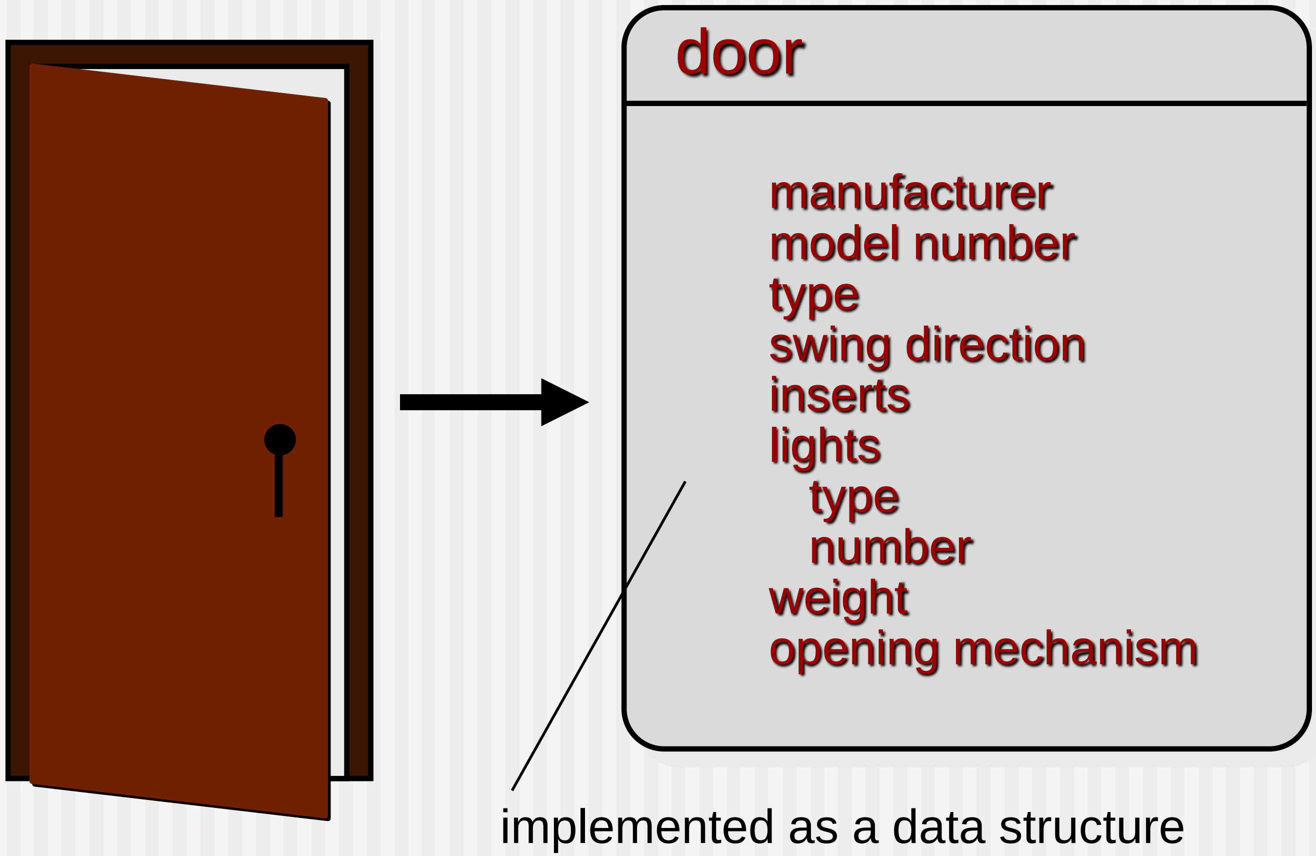
# *Design Concepts*

# Fundamental Concepts

- Abstraction—data and procedure
- 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
- OO design concepts—Appendix II
- Design Classes—provide design detail that will enable analysis classes to be implemented

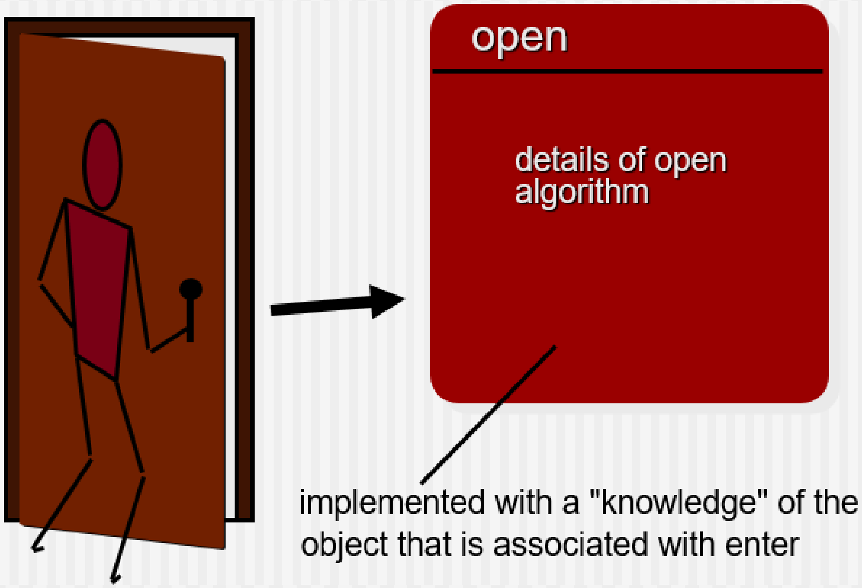
# 1) Data Abstraction

- A data abstraction is a named collection of data that describes a data object.



# 2) Procedural Abstraction

- A procedural abstraction refers to the ability of naming and later calling a set of instructions.
  - The name of a procedural abstraction implies its function.
  - It is called abstraction because the caller of the procedure only needs to know what the procedure does not how it does it.



# 3) Architecture

- Software architecture is:
  - the structure or organization of program components (modules),
  - the manner in which these components interact,
  - and the structure of data that are used by the components.

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

# What is conceptual integrity?

- Conceptual integrity is the principle that anywhere you look in your system, you can tell that the design is part of the same overall design.
- Even if multiple people work on it, it would seem cohesive and consistent as if **only one mind** was guiding the work.
  - The system reflects one set of design ideas, instead of containing many good but independent, conflicting and uncoordinated ideas.
  - Development teams must have in their **collective minds** the same vision for the software

# Architectural Design

**There are a set of properties for an architectural design:**

1. **Structural properties.**
   - components of a system (e.g., modules, objects)
   - and the manner in which those components are packaged and interact with one another
2. **Extra-functional properties.** The architectural design should address how to achieve **non-functional requirements**
   - e.g., performance, reliability, security, etc
3. **Families of related systems.** The architectural design should draw upon **repeatable patterns** that are **commonly encountered** in the design of **similar systems**.
   - The design should reuse architectural building blocks

# 4) Patterns

- A software design pattern is a **general**, **reusable solution** to a **commonly occurring problem** within a **certain context** in software design

***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**

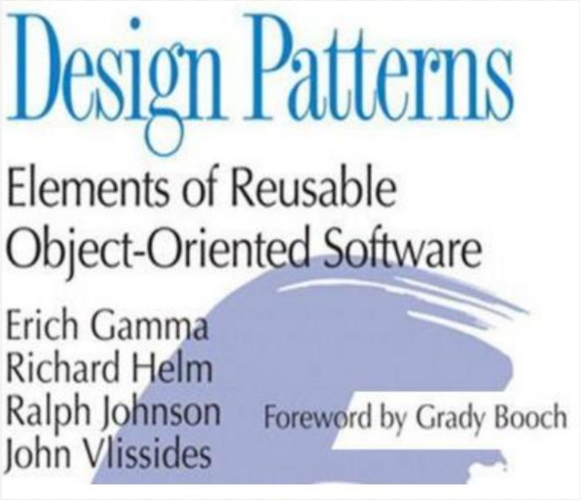
***Structure*****—describes the classes that are required to implement the pattern**

**….**

For an example, see Slide 62 (**Mediator** design pattern)

# Design Pattern Examples

- Singleton
  - Ensure a class has only one instance
- Object pool
  - Avoid expensive acquisition and release of resources
- Adapter
  - Convert the interface of a class into another interface clients expect
- …



# 5) Separation of Concerns

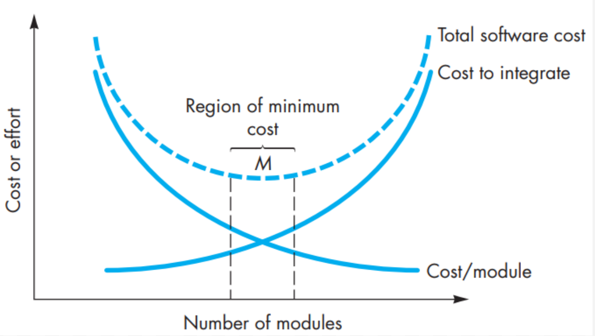
- A *concern* is a feature or behavior that is specified as part of the requirements
- Any complex problem can be more easily handled if it is subdivided into pieces
  - pieces that can be solved and/or optimized independently
- A problem takes less effort and time to solve:
  - By separating concerns into smaller, and therefore more manageable pieces
    - Divide-and-conquer strategy

# 6) Modularity

- Modularity is the most common manifestation of separation of concerns.
- **Monolithic** software

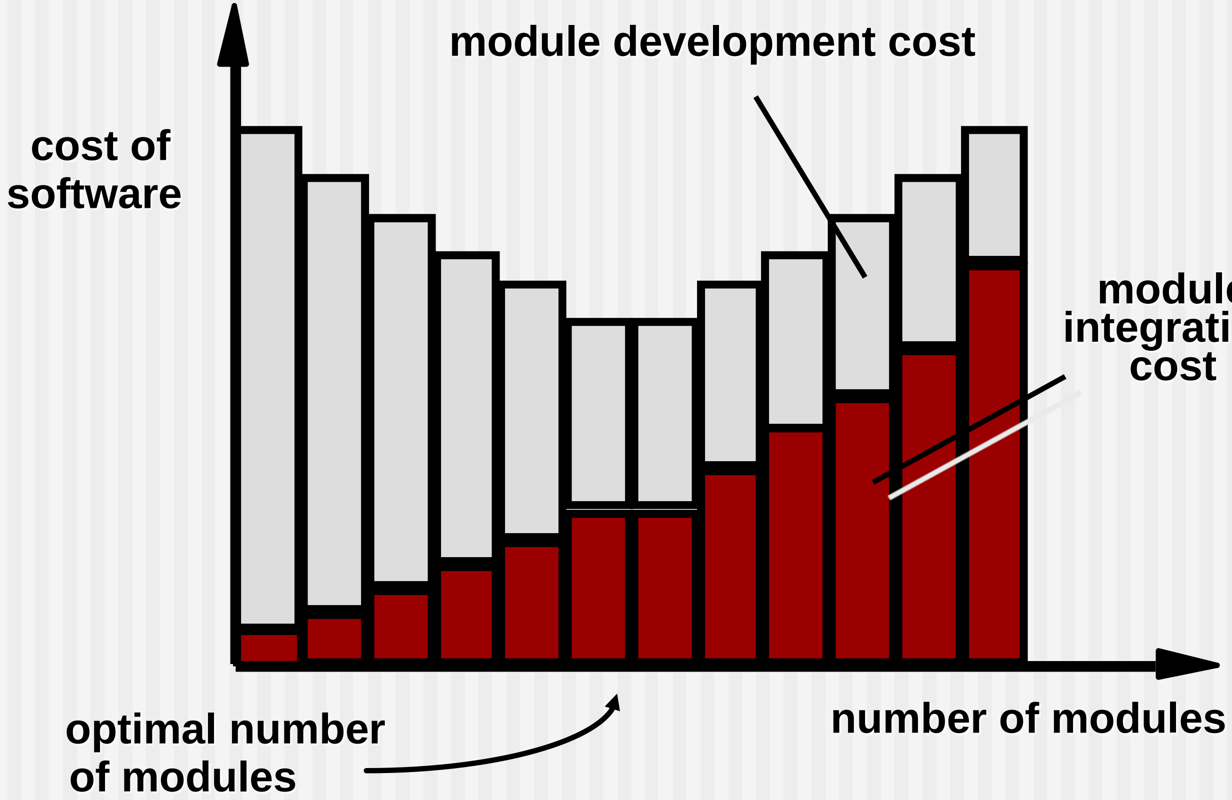
  (i.e., a large program composed of a single module)

  cannot be easily grasped by a software engineer.
  - The overall complexity would make understanding impossible.
- You should break the design into many modules
  - hoping to make understanding easier
  - and reduce the cost required to build [and maintain] the software
- What is the risk of “many modules”?
  - **Integration**



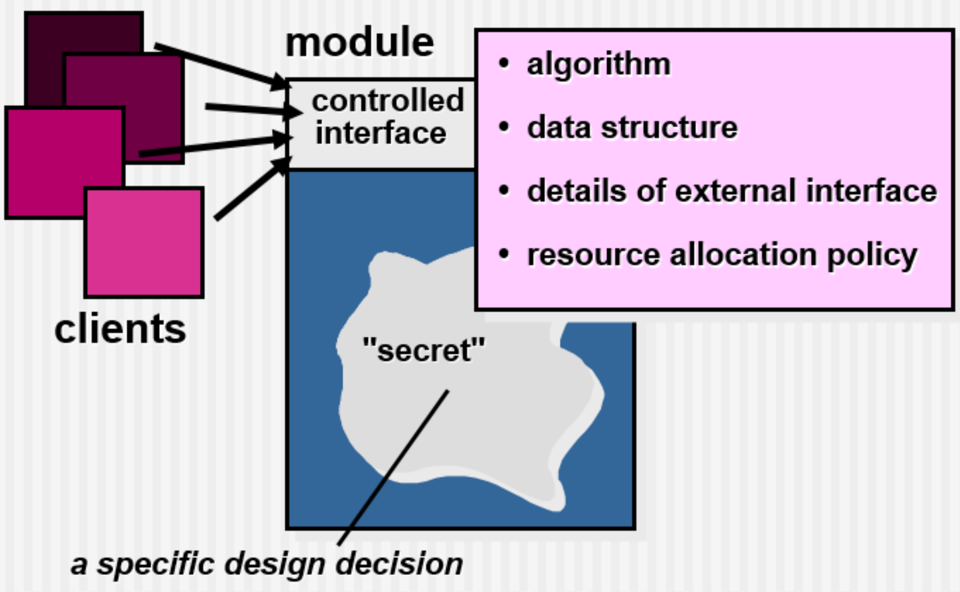

# Modularity: Trade-offs

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



# 7) Information Hiding

- Modules should be specified and designed so that:
  - information (algorithms and data) contained within a module is **<u>inaccessible</u>** <u>to other modules that have</u> **<u>no need</u>** <u>for such information</u>.



# 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

# 8) Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and a “dislike" to excessive interaction with other modules.
- *Cohesion*: The relative functional strength of a module
- *Coupling*: The relative interdependence among modules
  - Collaboration between modules should be kept to an acceptable minimum (**loosely coupled**)
  - If a design model is tightly coupled (each design class collaborates with another), the system is difficult to implement, test and maintain

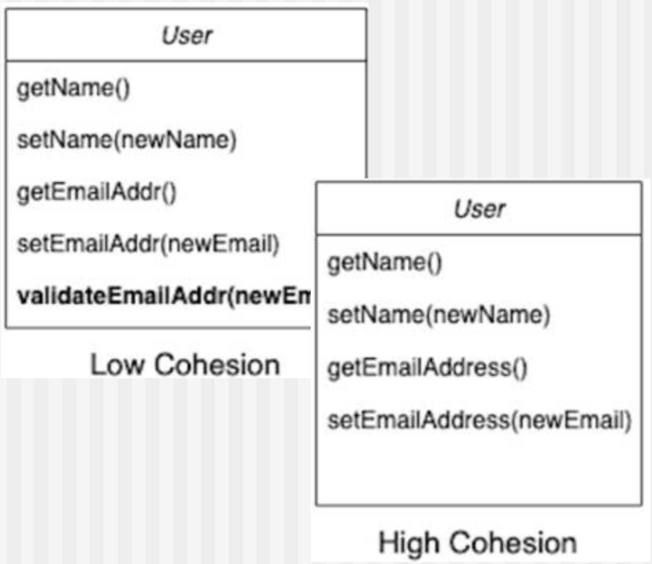
# Cohesion

- Cohesion is the degree to which elements inside a module (e.g., class, package, …) are functionally related to each other and united in their purpose
- A cohesive module
  - has a **small**, **focused** set of responsibilities
  - and **single-mindedly** applies attributes and methods to implement those responsibilities
  - performs a single task, requiring little interaction with other components in other parts of a program
- For example, all methods within class **User** should represent the user behavior
- Modules that perform many unrelated functions must be avoided

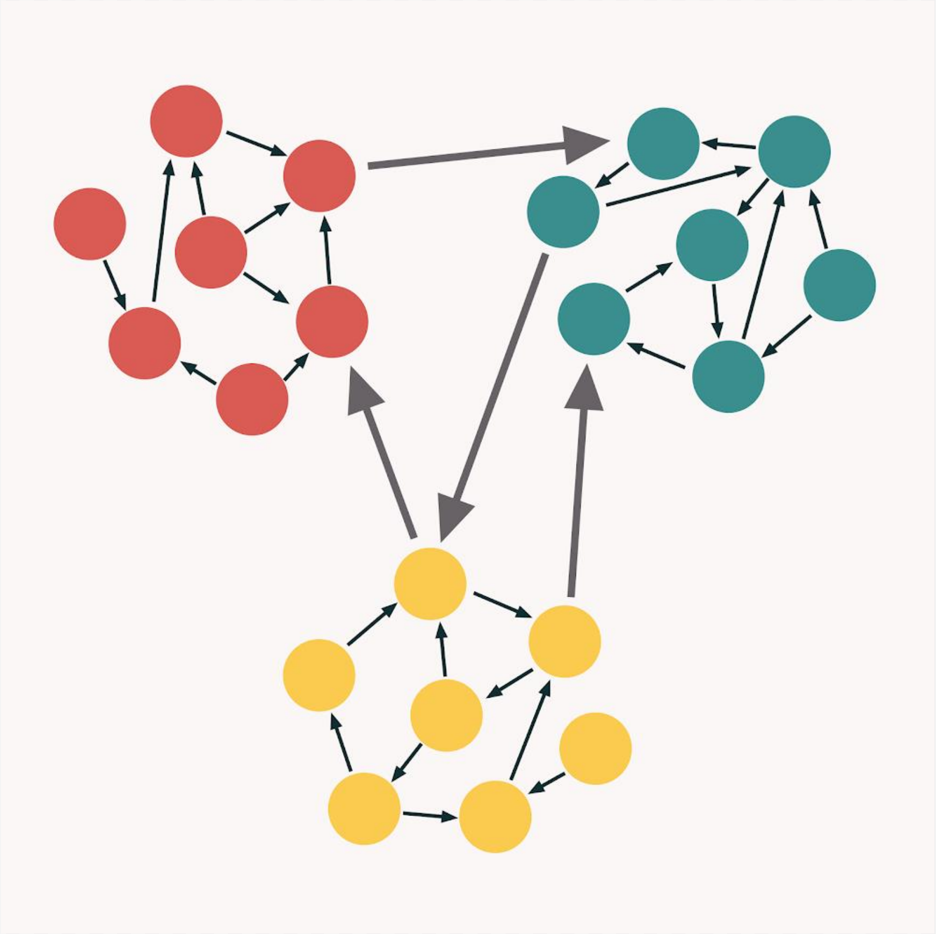
# Cohesion: Examples

- Example 1: class **VideoClip** might contain a set of methods for editing the video clip.
  - **VideoClip** is a cohesive class as long as:
    1. Each method focuses solely on attributes associated with the video clip
    2. All methods related to the video editing are incorporated in **VideoClip**

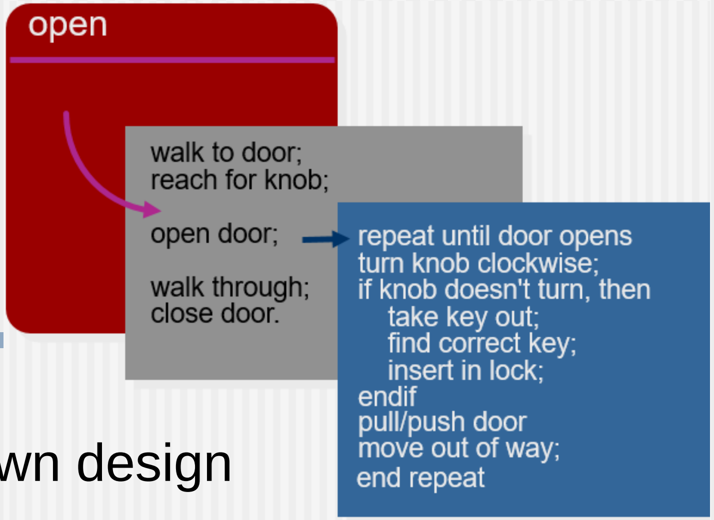
- Example 2: class **User** can be responsible for storing the email address of the user but not for validating it
  - That should belong to some other class like **Email**



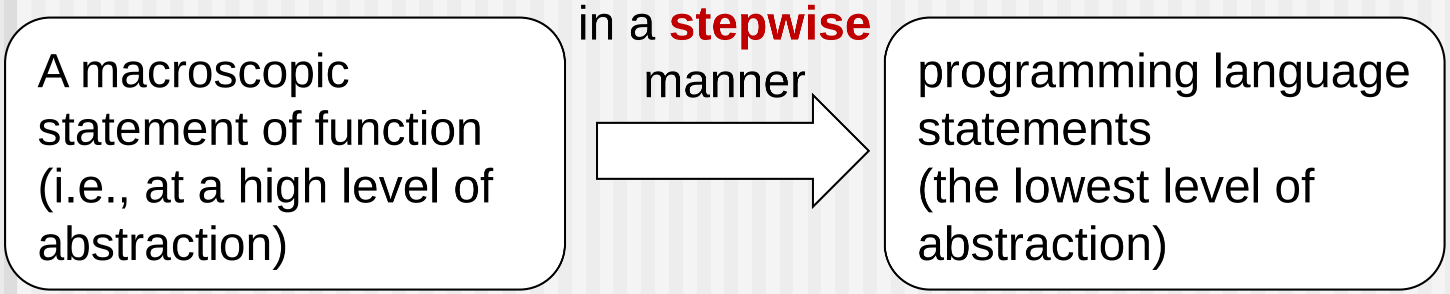
# A Loosely Coupled and Highly Cohesive System



# 9) Refinement

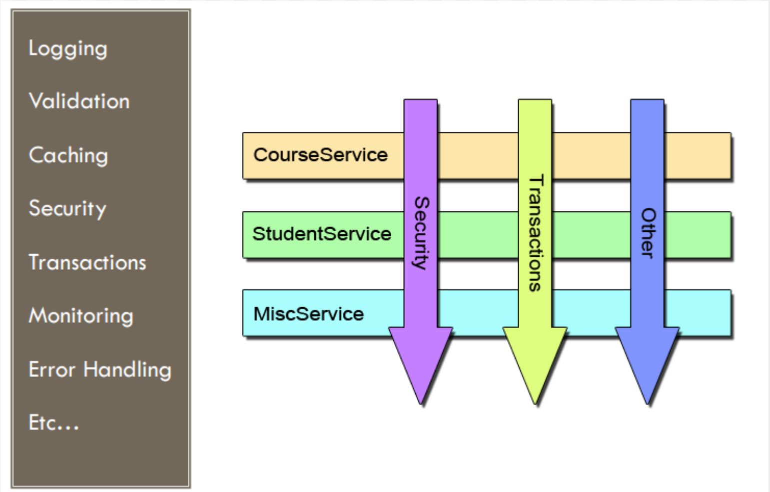



- Stepwise refinement is a top-down design strategy.
- An application is developed by successively refining levels of procedural detail.
  - Decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached.



# 10) Aspects

- Consider two requirements, *A* and *B*:
  Requirement *A* ***crosscuts*** requirement *B*,
  if *B* cannot be satisfied without taking *A* into account.
- Cross-cutting concerns often **cannot be cleanly decomposed** from the rest of the system
- An **aspect** is a representation of a cross-cutting concern



# 11) Refactoring

- Martin Fowler 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 [bad smells] that can be corrected to yield a better design

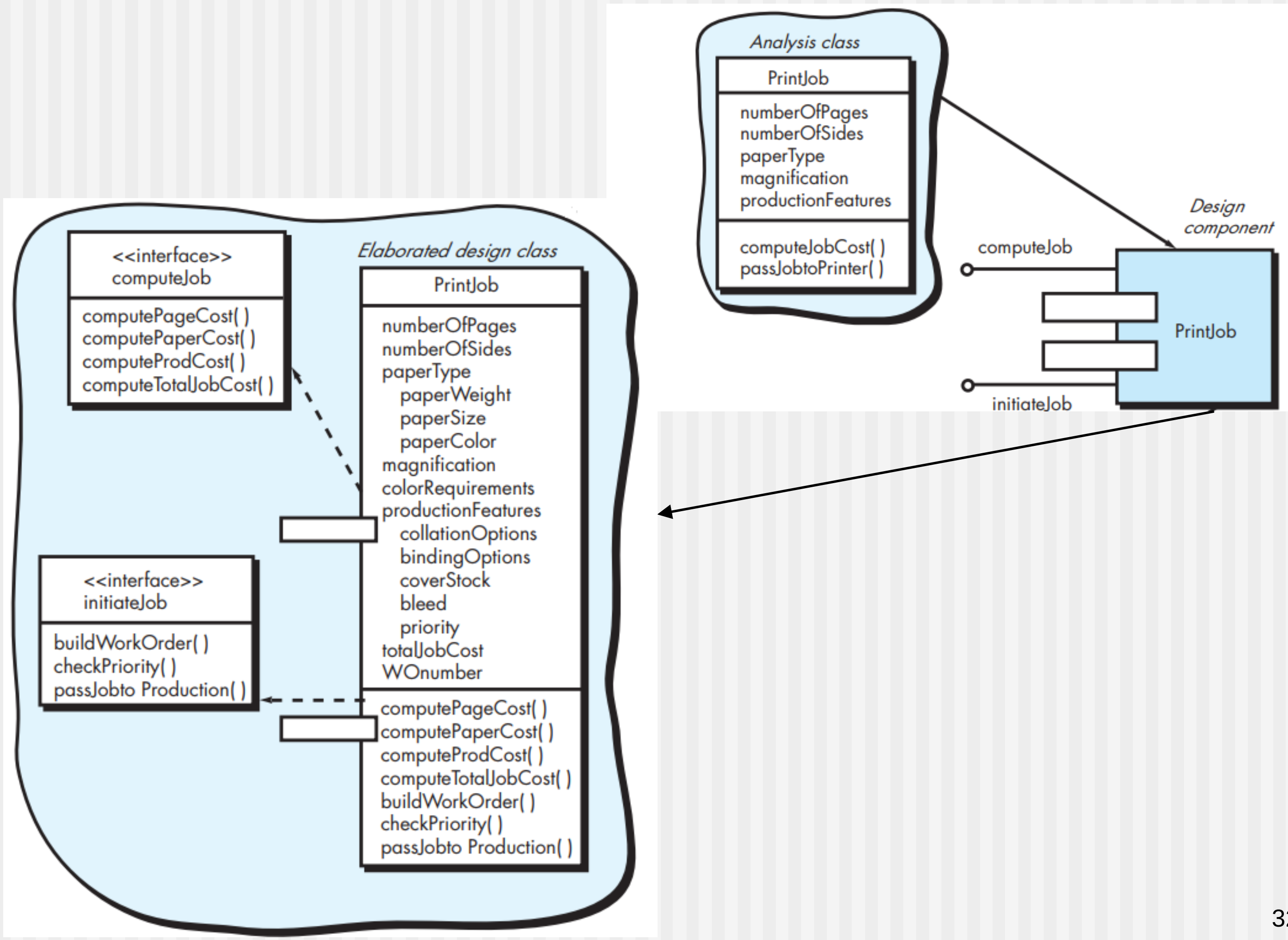
# 12) OO Design Concepts

- 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—a characteristic that greatly reduces the effort required to extend the design

# Design Classes: Entity

- Entity classes are extracted directly from the problem statement
  - Also called business classes
  - e.g., **FloorPlan** and **Sensor**
  - Things that are to be stored in a database and persist
  - Analysis classes are refined during design to become entity classes
    - The analysis model defines a set of analysis classes
    - The abstraction level of an analysis class is relatively high
    - As the design model evolves, design classes refine the analysis classes by providing more details that will enable the classes to be implemented

# Analysis Class vs Design Class



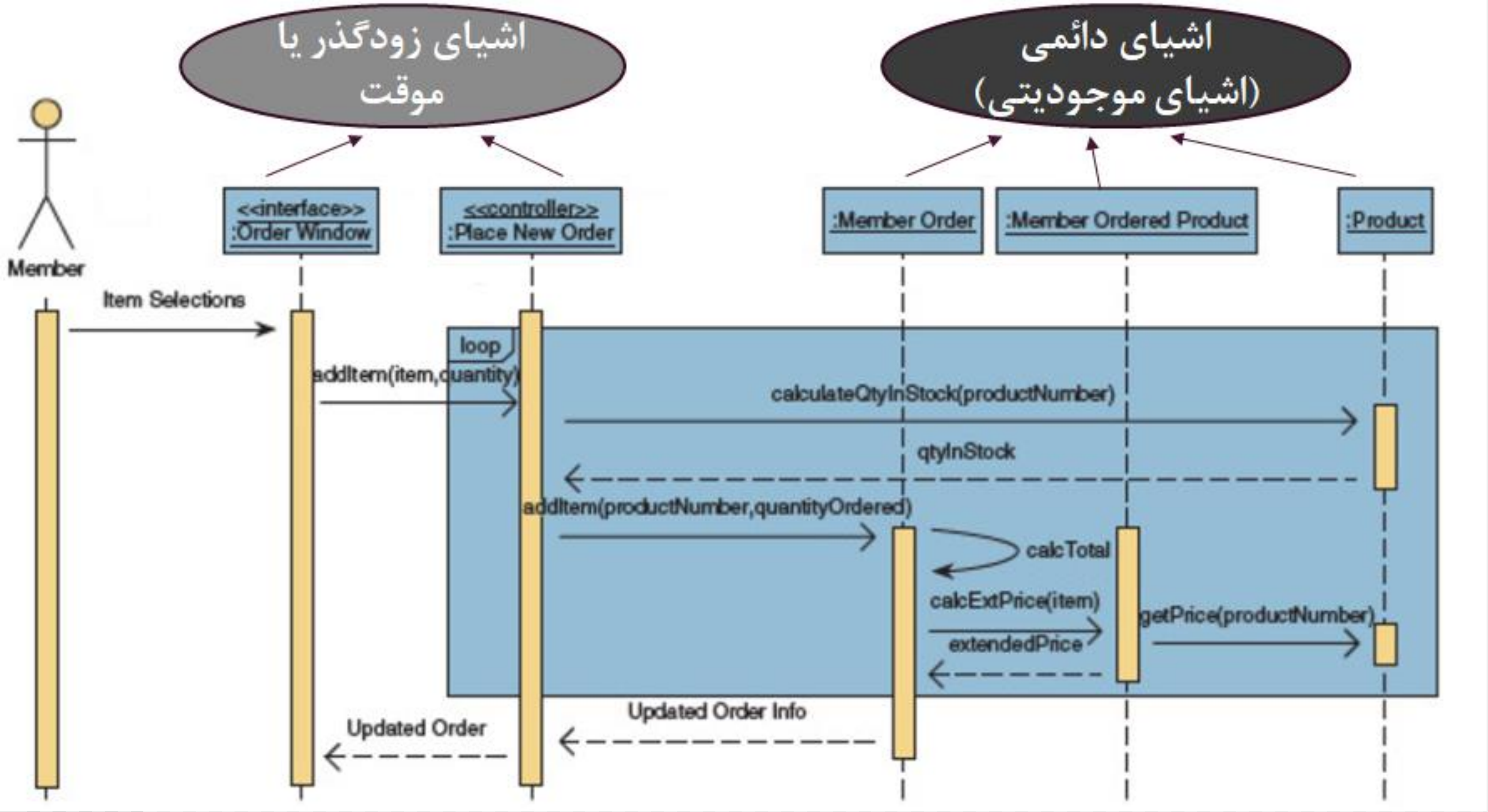
# Design Classes: Boundary

- Boundary classes are developed during design to create the user interface that the user sees and interacts
  - E.g., interactive screen or printed reports
  - Entity objects contain information that is important to users, but they do not display themselves
  - Boundary classes are designed with the responsibility of managing how entity objects are represented to users
  - For example: **CameraWindow**
    - This class has the responsibility of displaying surveillance camera output for the SafeHome homeowner

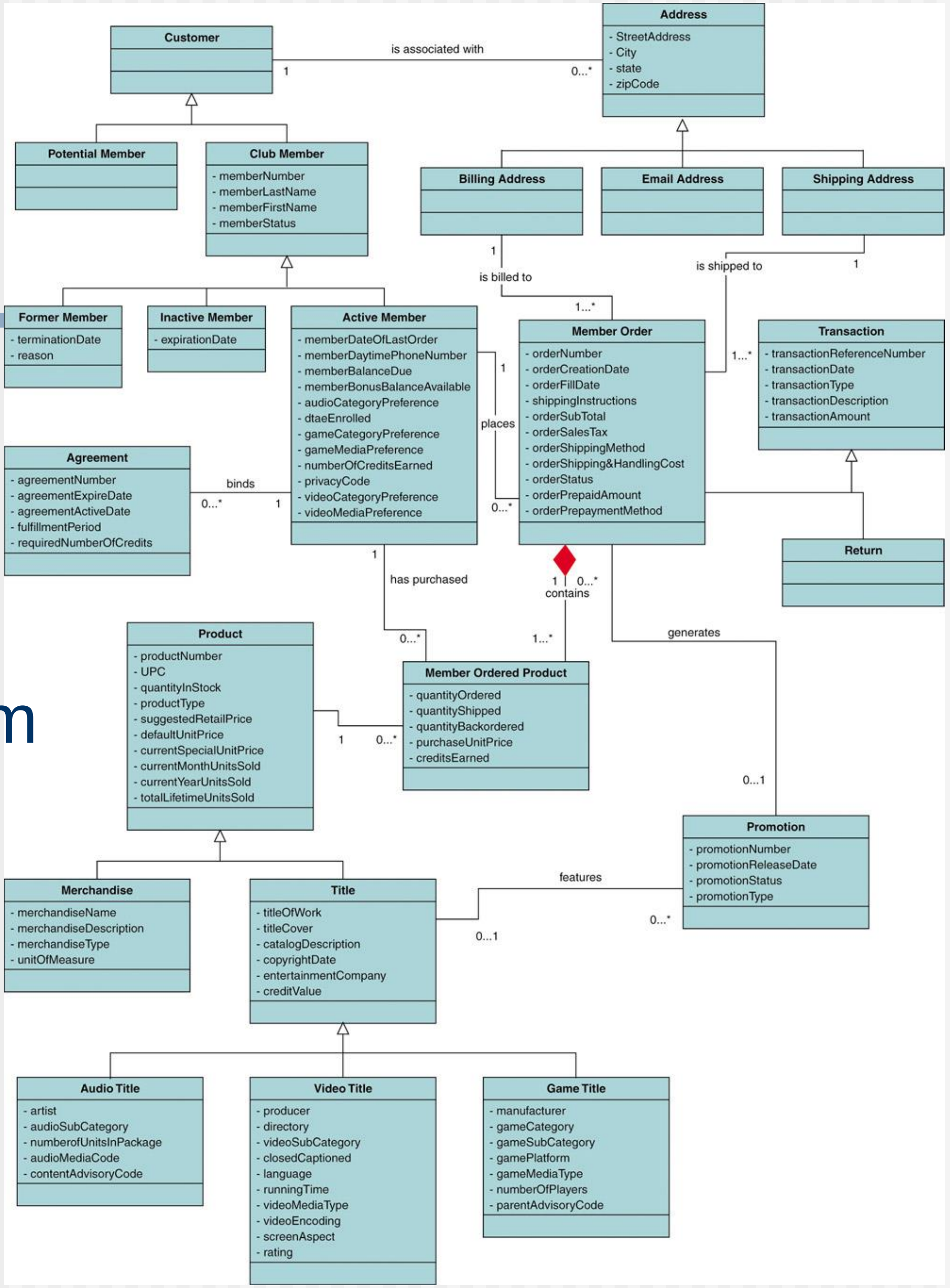
# Design Classes: Controller

- Controller classes are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they [controllers] obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.
  - In general, controller classes are not considered until the design activity has begun

# Entity, Boundary, Controller: An Example

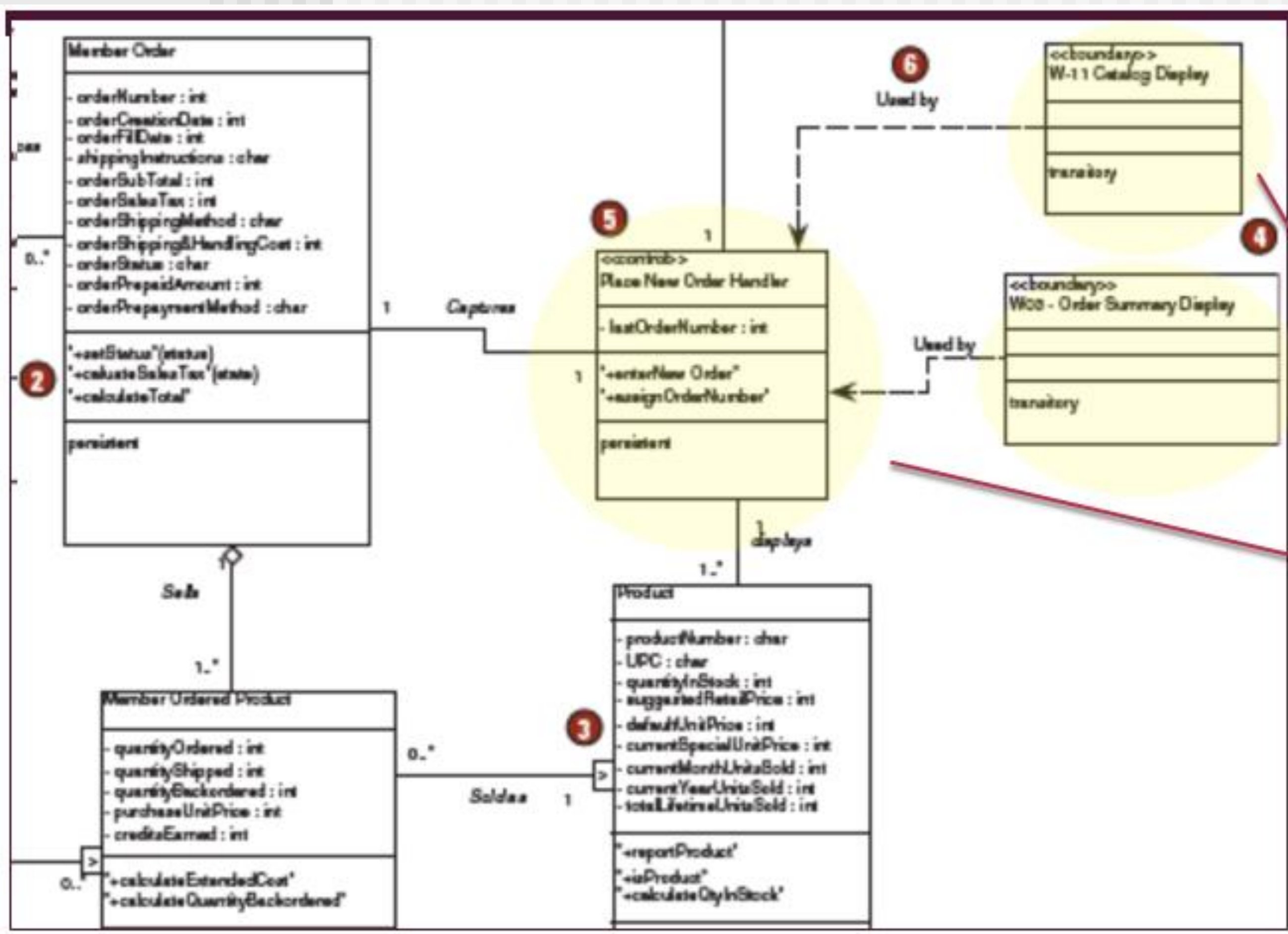


# An Example of Analysis Class Diagram vs Design Class Diagram




***Analysis Class Diagram***

# An Example of Analysis Class Diagram vs Design Class Diagram




***Design Class Diagram (partial)***

کلاس‌های جدید موقت
(مرزی و کنترلی) که در
مرحله طراحی اضافه شده‌اند

# Design Class Characteristics

- Each design class be reviewed to ensure that it is “well-formed”
- Four characteristics of a well-formed design class:

1. Complete and sufficient - includes **all necessary attributes and methods** and contains **only** those methods **needed** to achieve class intent (no more and no less)
2. Primitiveness – **each class method** focuses on providing **one service**
3. High cohesion – small, focused, single-minded classes
4. Low coupling – class collaboration kept to minimum

# Primitiveness: An Example

- Once the service has been implemented with a method
  - The class **should not provide another way** to accomplish the **same thing**
- For example, the class **VideoClip** for video editing software
- It has attributes **startPoint** and **endpoint** to indicate the start and end points of the clip
- The methods, *setStartPoint()* and *setEndPoint()*, provide the only means for establishing start and end points for the clip.

# *Design Model*

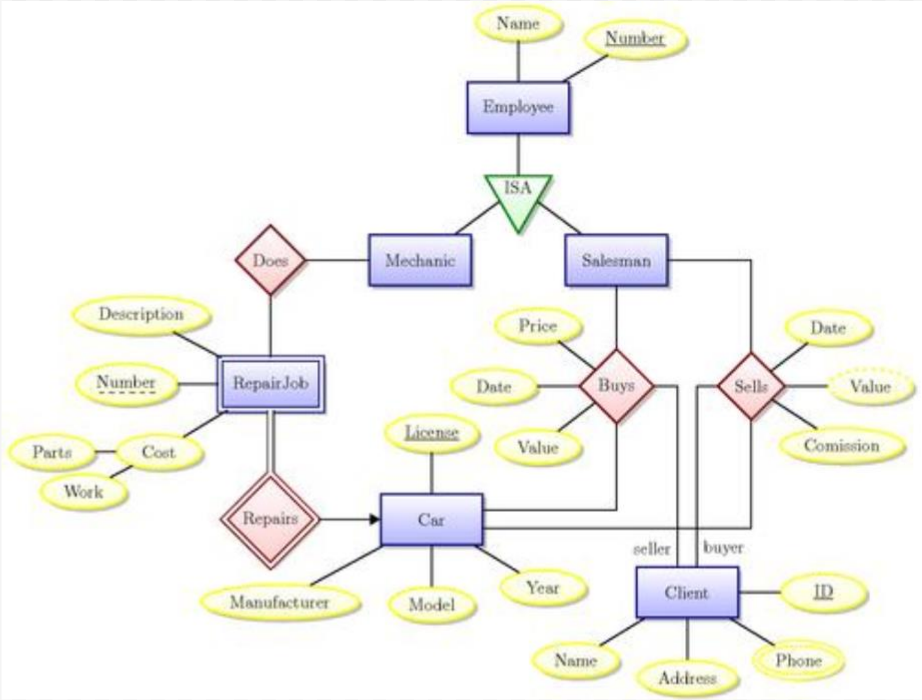
# Design Model Elements

- Data elements
  - Data model --> data structures (using ER for example)
  - Data model --> database architecture
- Architectural elements
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Architectural patterns and “styles” (Chapters 13 and 16)
- Interface elements
  - the user interface (UI) (Chapter 15)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- Component elements
- Deployment elements

# Data Design Elements

- Focuses on the data domain
  - Data objects independently of processing
  - The structure of data
  - Indicates how data objects relate to one another

*An example of a data model (using ER)*



# Architectural Design Elements

- The architectural model is usually depicted as a <u>set of interconnected subsystems</u>
  - often derived from analysis packages within the requirements model
- Each subsystem may have its own architecture
- The architectural model is derived from three sources:
  1. Information about the **application domain**
  2. Specific **requirements model** (the problem at hand)
     - Such as use cases or analysis classes
  3. The availability of **architectural patterns and styles**

(More details: next sessions)

# Interface Design Elements

- The interface design elements depict:
  - information flows into and out of a system
    - External communications
  - and how it is communicated among the components defined as part of the architecture
    - Internal communications
- Important elements
  - **User interface** (UI)
  - **External interfaces** to other systems
  - **Internal interfaces** between various design components

# Interface Elements—UI

- UI design incorporates:
  - **aesthetic** elements (e.g., layout, color, graphics)
  - **ergonomic** elements (e.g., information layout and placement, UI navigation)
  - **technical** elements (e.g., UI patterns, reusable components)
- Details in Chapter 15

# Interface Elements—External Interfaces

- The design of external interfaces requires information about the sender and receiver entities
- This information should be
  - collected during requirements engineering
  - and verified once the interface design commences
    - The designer should ensure that the specification for the interface is **accurate** and **complete**
- The design of external interfaces should incorporate error checking and appropriate security features.

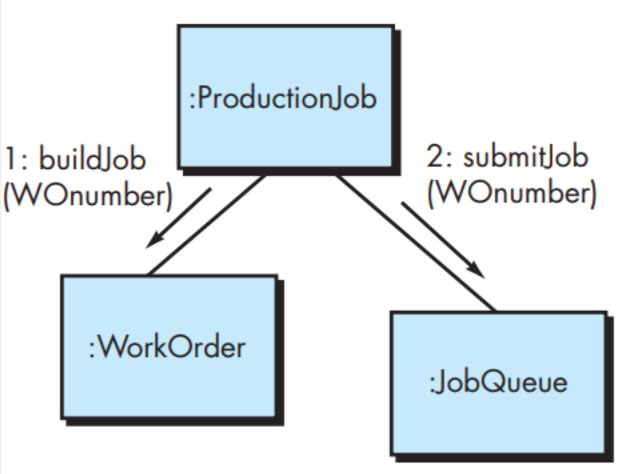
# Interface Elements—Internal Interfaces

- The design of internal interfaces is closely aligned with **component-level design**
  - represents all operations and the messaging schemes required to enable communication between various classes or components
- Two tasks are required:
  1. Specifying message details when classes or components collaborate
  2. Identify appropriate interfaces for each component
- Details in Chapter 14

# 1) Designing Messaging Schema

- The design model can show the details of collaborations by specifying the structure of messages that are passed between objects within a system
- During requirements modeling, collaboration diagrams show how analysis classes collaborate with one another
- Example:
  - Three objects collaborate to prepare a print job for submission to the production stream.
  - Messages are passed between objects as shown by arrows

***Collaboration diagram with messaging***



# 1) Designing Messaging Schema

- As design proceeds, each message is elaborated by expanding its syntax in the following manner:

[guard condition] sequence expression (return value) :=

message name (argument list)

  - [guard condition] specifies **conditions** that must be met before the message can be sent;
  - sequence expression is an integer value that indicates the sequential **order** in which a message is sent;
  - (return value) is the name of the **information** that is returned by the invoked operation
  - message name identifies the invoked **operation** (e.g., buildJob);
  - (argument list) is the list of **attributes** passed to the operation.

# 2) Designing the Interface of Objects

- A UML interface  is a classifier that declares a set of public operations
- The interface contains no internal structure, it has no attributes and no associations
- Two notations:

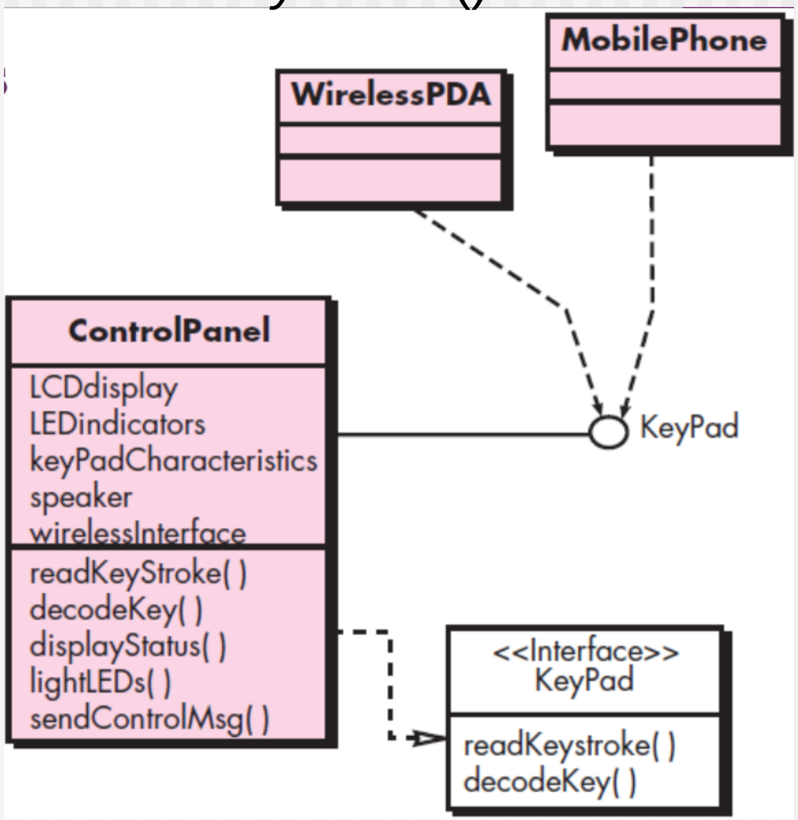

***Interface*** *SiteSearch is* ***realized*** *(implemented) by SearchService.*

*"**lollipop**" symbol is shorthand for a realization relationship of  an interface classifier*


***Interface*** *SiteSearch is* ***realized*** *(implemented) by SearchService.*

# An Example of Interface Representation

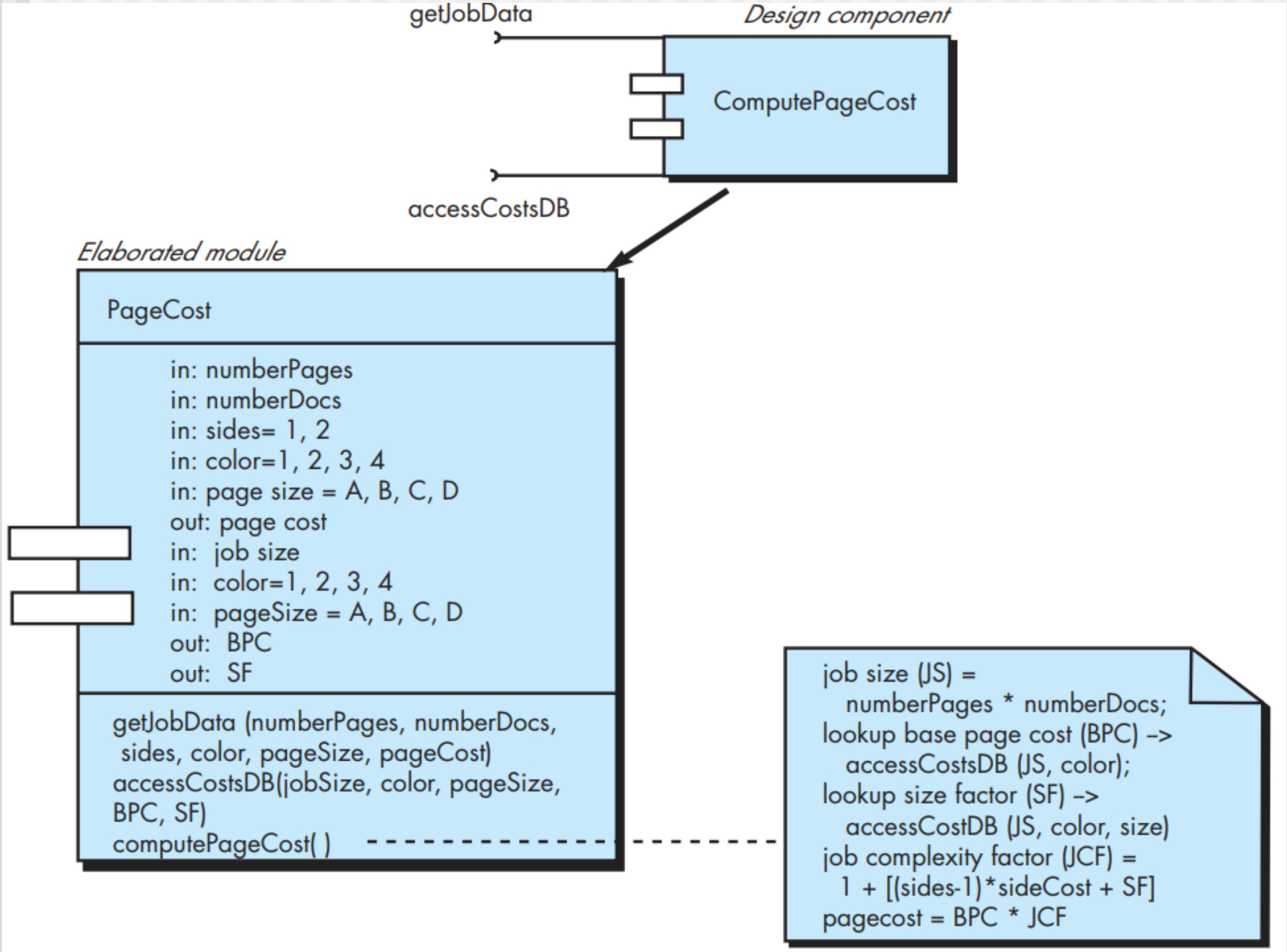
- **ControlPanel** provides the behavior associated with a keypad
  - It implements the operations *readKeyStroke*() and *decodeKey*()
  - ControlPanel provides theses two services to other classes (i.e., **WirelessPDA** and **Mobilephone**)
  - Interface **KeyPad** is realized (implemented) by **ControlPanel**



# Component-Level Design Elements

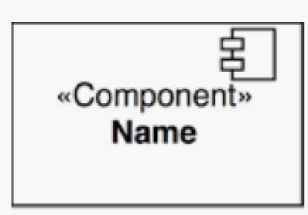
- Describes the **internal detail** of each software component
  - Data structures for all local data objects within a component
  - Algorithmic detail for all processing functions that occurs within a component
  - An interface that allows access to all operations (behaviors) of a component
- Details in Chapter 14

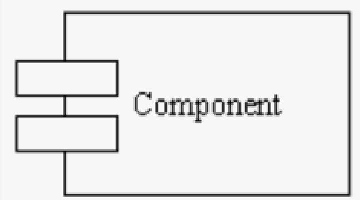
# An Example of Component-Level Design



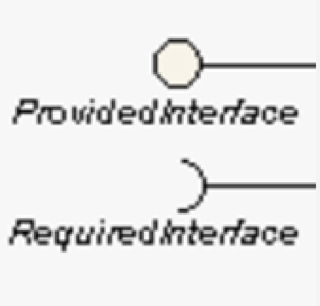
# Component Diagram

- In UML, a component is represented as follows:

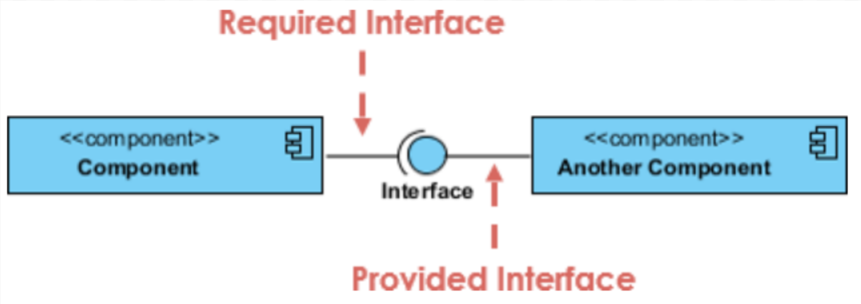


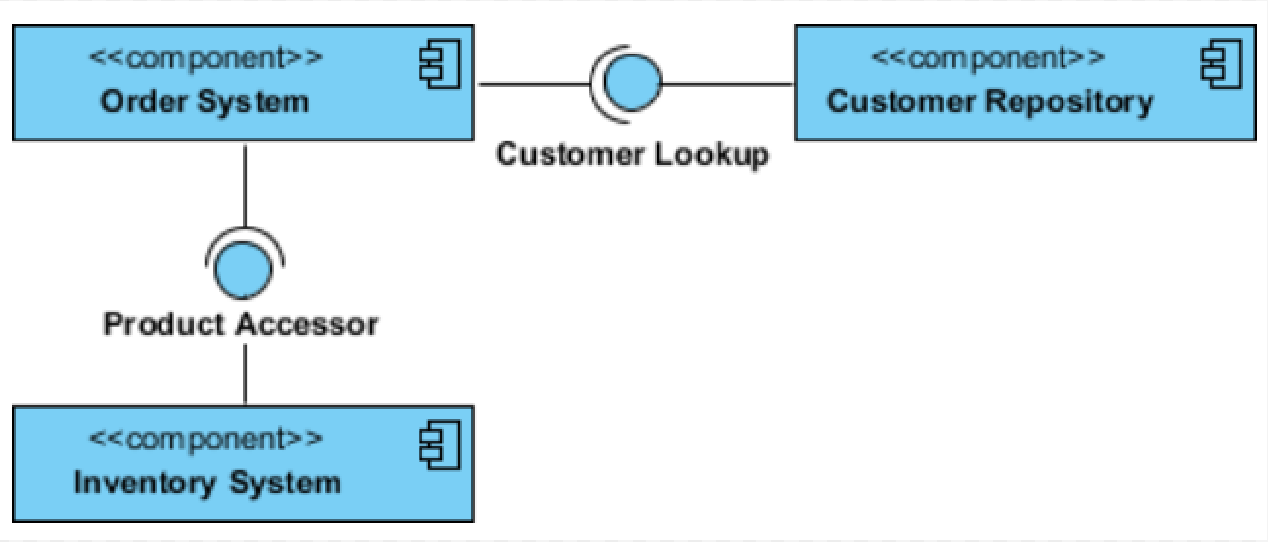

- Provided Interface and the Required Interface:



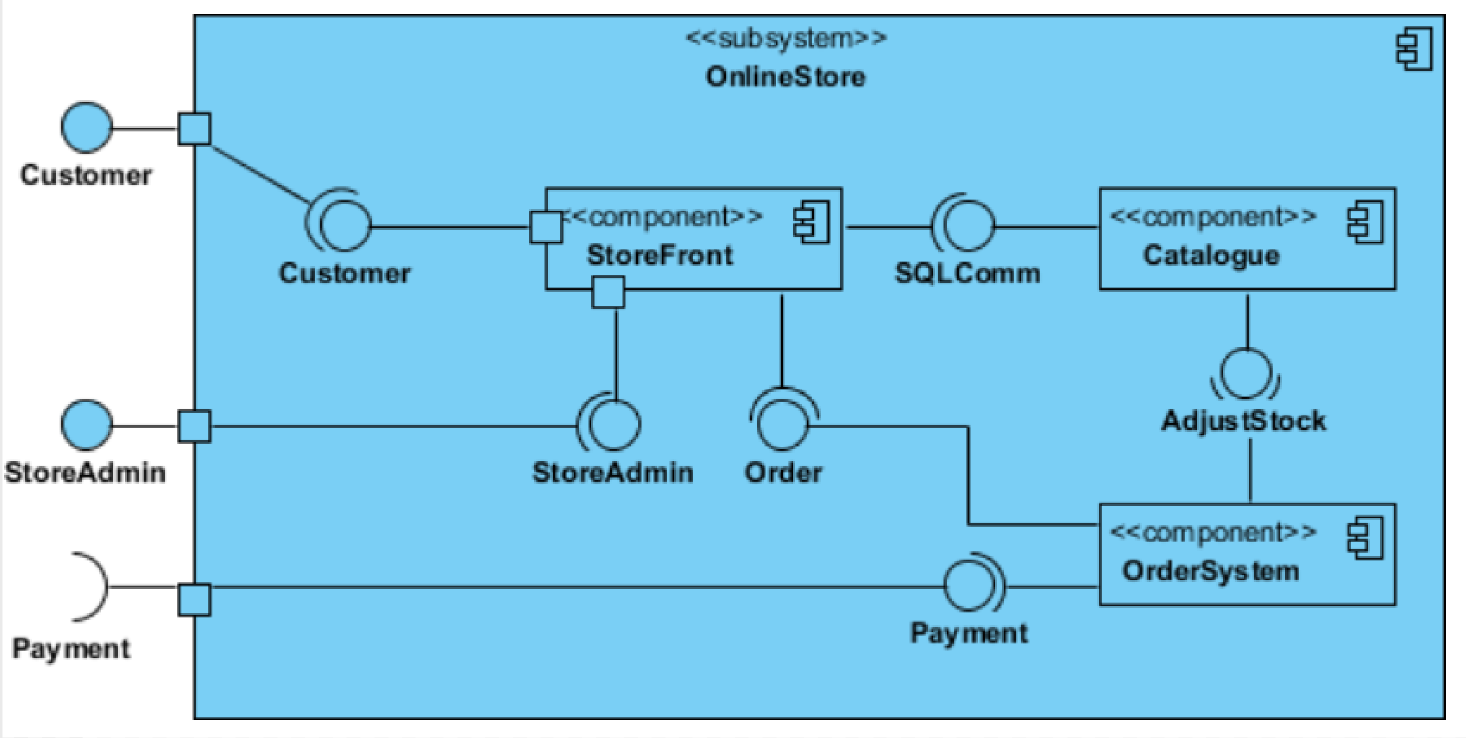

  - Provided interface: An interface (or services) that the component provides
  - Required interface: An interface (or services) that the component requires



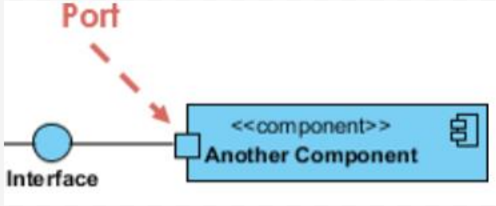
# Component Diagram: Example 1



# Component Diagram: Example 2




- What is **port**?



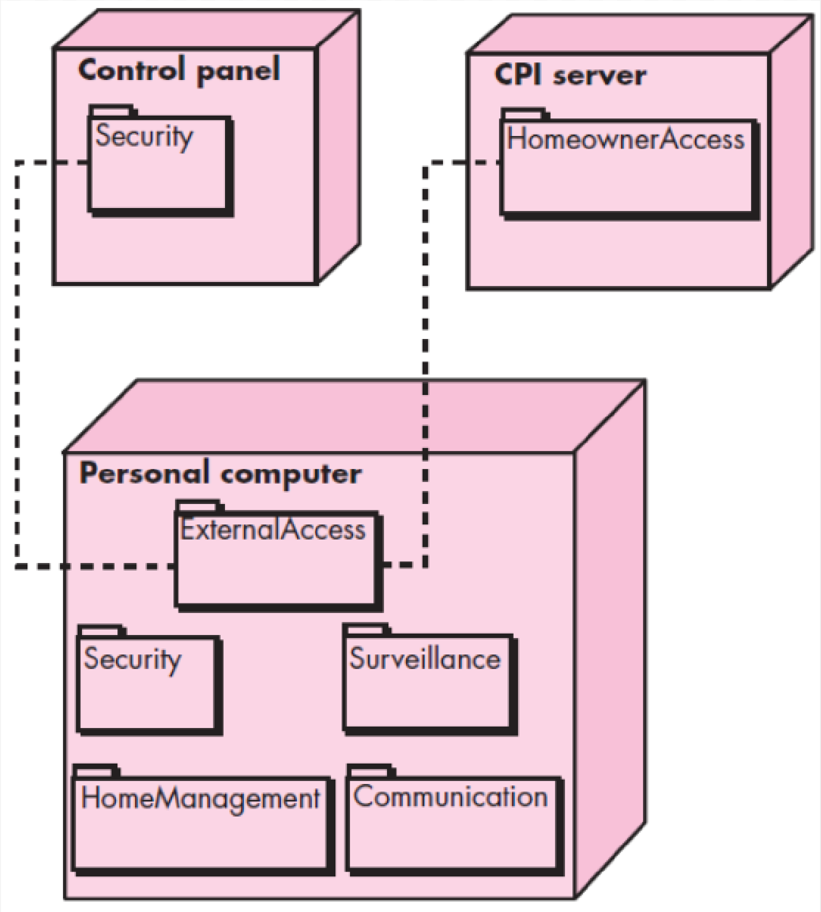
# Deployment-Level Design Elements

- Indicates how software functionality and subsystems will be allocated within the **physical computing environment**
- Modeled using UML deployment diagrams
  1. *Descriptor form* deployment diagrams show the computing environment <u>but does not indicate hardware configuration details</u>
  2. *Instance form* deployment diagrams <u>explicitly indicate hardware configuration details</u>
- Developed during the latter stages of design

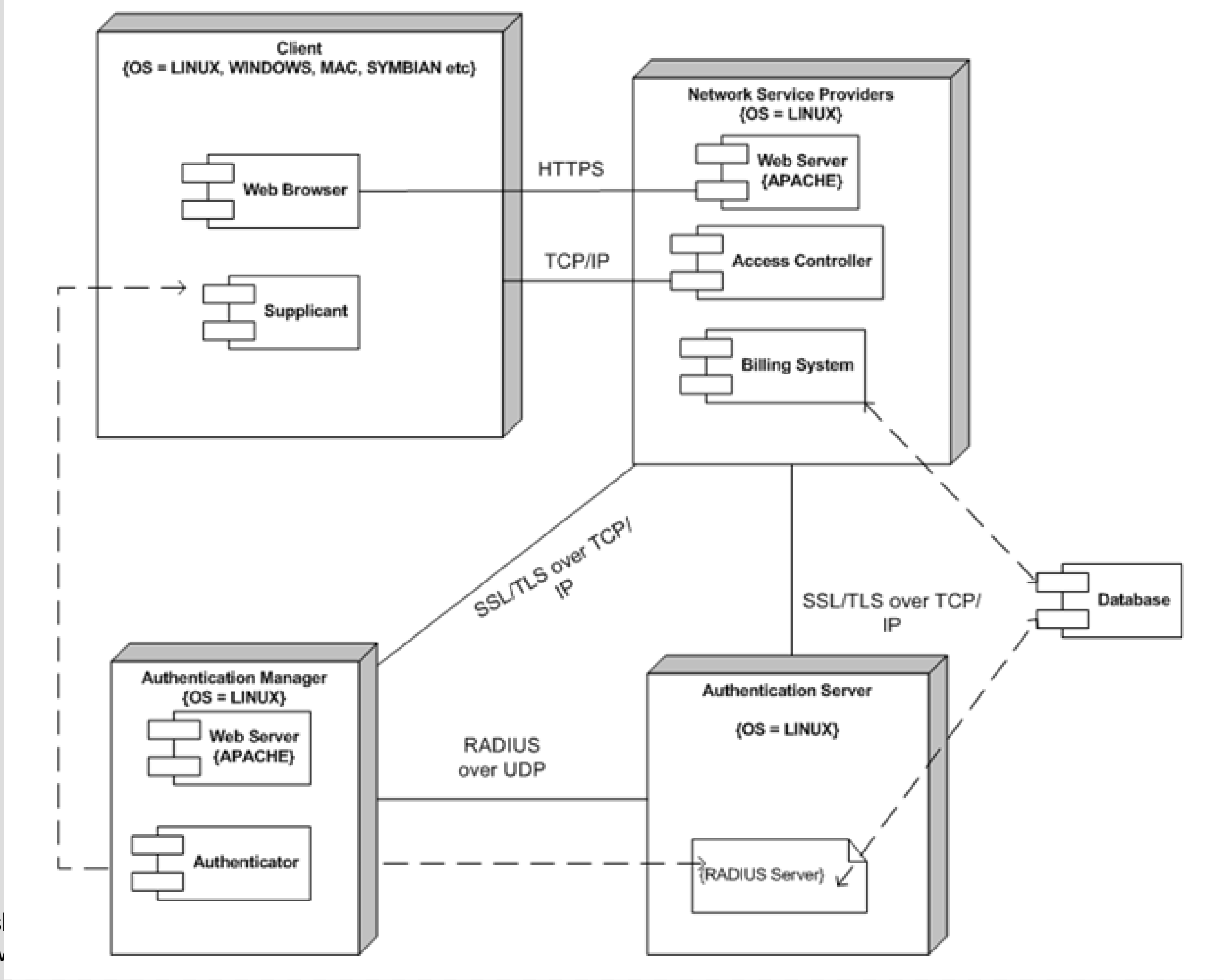
# Deployment Diagram: *Descriptor Form*

- The elements of the *SafeHome* product are configured to operate within three primary computing environments:

1. A homebased PC
2. The SafeHome control panel
3. A server housed at CPI Corp



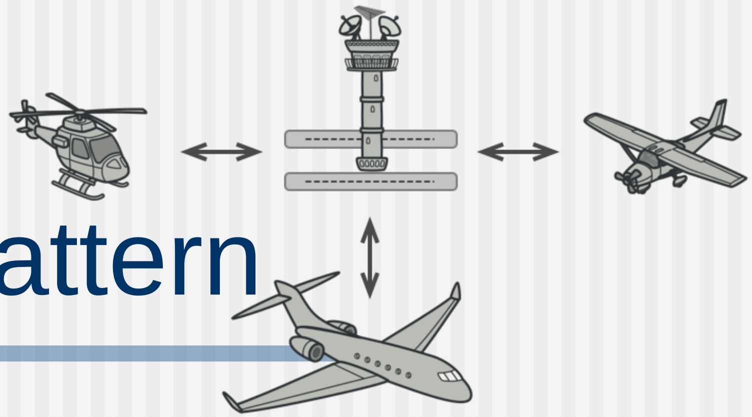
# Deployment Diagram: *Instance Form*



# Further Reading

# *The End*

# Mediator Design Pattern



- Intent
  - Mediator is a **behavioral design pattern** that lets you **reduce dependencies** between objects
  - Promotes **loose coupling**
- Solution
  - Restricts direct communications between objects and forces them to collaborate only via a mediator object
  - components collaborate indirectly by calling a special mediator object that redirects the calls to appropriate components
  - So, the components depend only on a single mediator class instead of being coupled to dozens of their colleagues

# Mediator Design Pattern

- Each component has a reference to a mediator
- The Mediator interface declares a method for communication between components (usually a single notification method).
- Concrete mediator often keep references to all components it manage and sometimes even manages their lifecycle
- Components must not be aware of other components
- When something happens with a component, it notifies the mediator
- Mediator can easily identify the sender and decide what should be done in return

(may do something on its own or pass the request to another component)