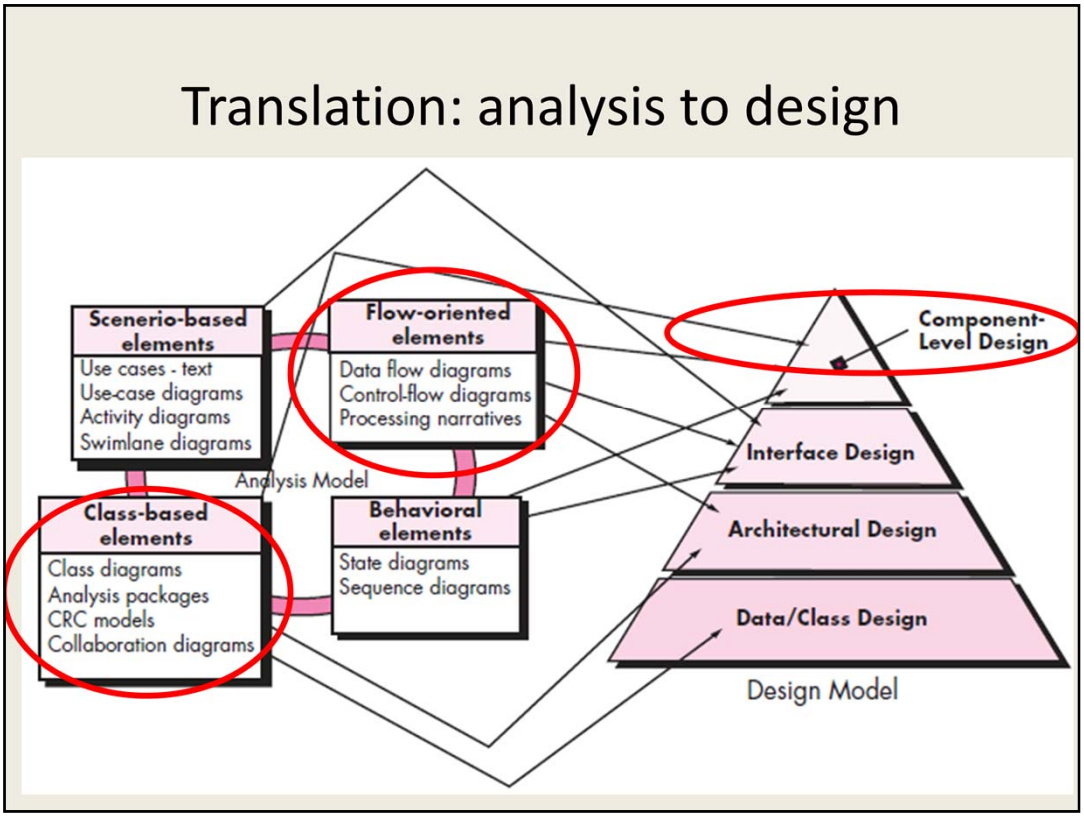
Component level design
(I)



Data/Class design: transforms analysis classes into implementation classes and data structures
Architectural design: defines relationships among the major structural elements of the software, the architectural styles and patterns
Interface design–defines how software elements, hardware elements, and end-users communicate. Usage scenarios and behavioral models are used
Component-level design–transforms structural elements into procedural descriptions of software components. Class-based models and behavioral models server as the basis

Only scenario-based elements serve just one design model

The view of component

- What is a component?
  - a modular building block for computer software
  - Depending on different views
    - Traditional view
    - Object-oriented view
    - Process-related view

component-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program

structure of the software has been established. The intent is to translate the design model into operational software.

We have talked about component quite a bit, without a clear definition.

The true meaning of the term *component* will differ depending on the point of view

of the software engineer who uses it

The view of component: OO

- Component: a set of collaborating classes
- Design:
  - Begin with requirements model, elaborate analysis class and infrastructure class
    - Recall: how did we do data/class design?
- Case study:

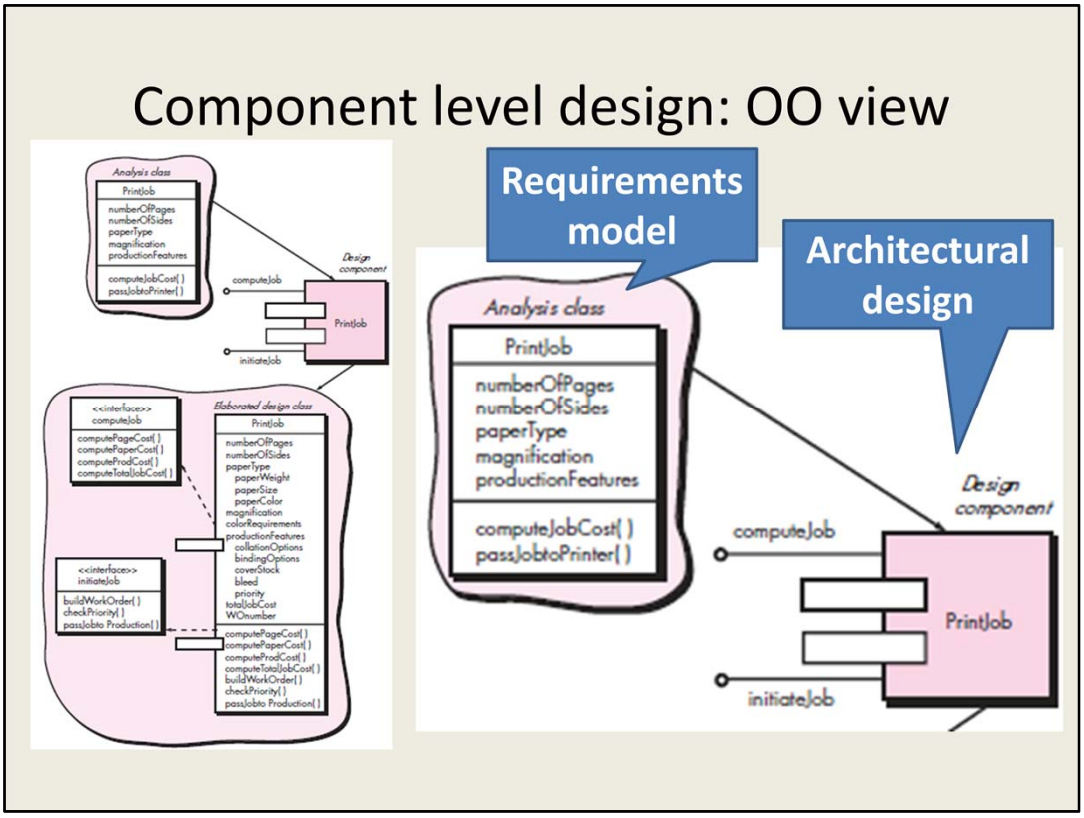
PrintShop software

- The overall intent is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility.

Each **class** within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.

All **interfaces** that enable the classes to communicate and collaborate with other design classes must also be defined.

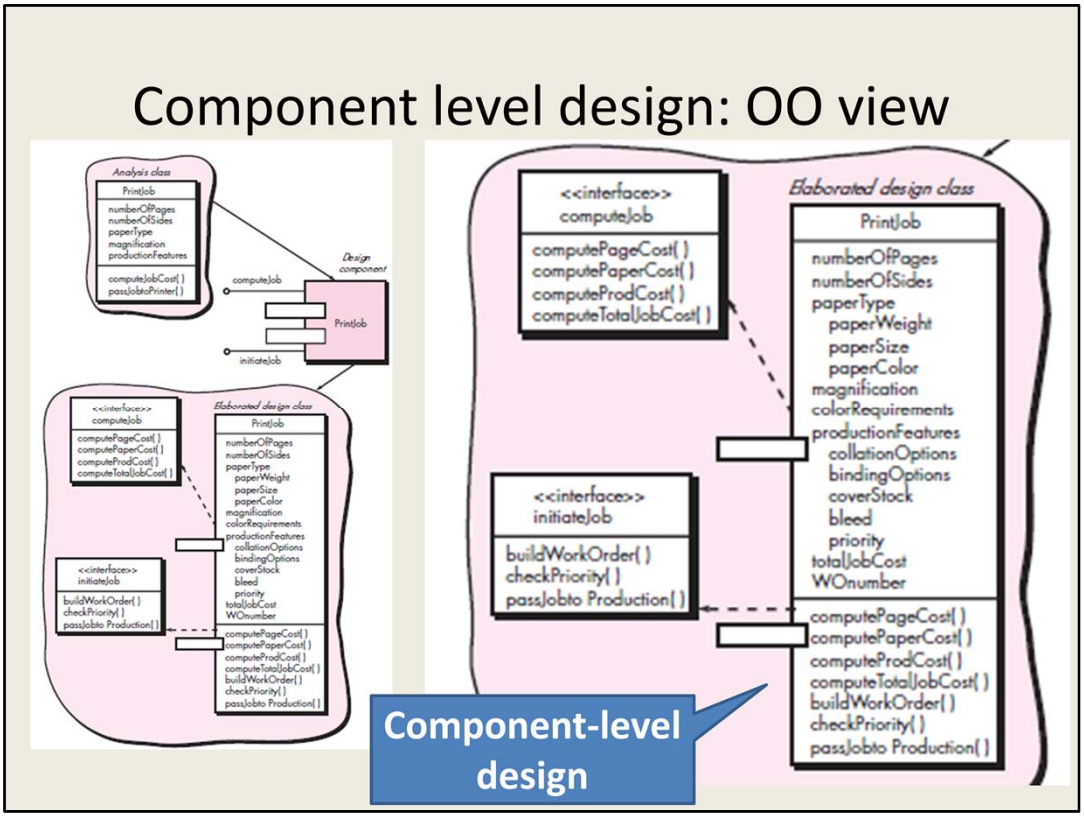
To define a component, we begin with the requirements model and elaborate analysis classes (for components that relate to the problem domain) and infrastructure classes (for components that provide support services for the problem domain).



During architectural design, **PrintJob** is defined as a component within the software architecture and is represented using the shorthand UML notation shown in the middle right of the figure.

PrintJob has two interfaces, *computeJob,* which provides job costing capability, and *initiateJob,* which passes the job along to the production facility.

Represented using the "***lollipop***" symbols shown to the left of the component box.

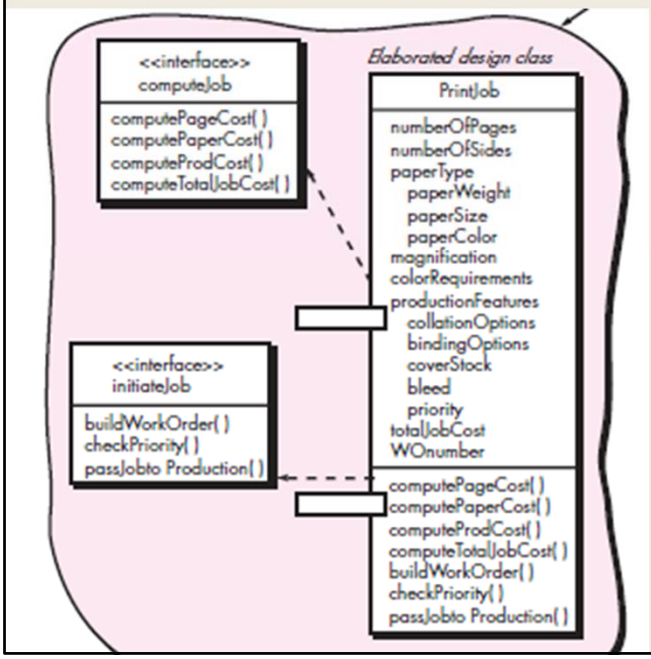


Component-level design begins at this point. The details of the component **PrintJob** must be elaborated to provide sufficient information to guide implementation. The original analysis class is elaborated to flesh out all attributes and operations required to implement the class as the component **PrintJob**

the elaborated design class **PrintJob** contains more detailed attribute information as well as an expanded description of operations required to implement the component. The interfaces *computeJob* and *initiateJob* imply communication and collaboration with other components (not shown here).

For example, the operation *computePageCost()* (part of the *computeJob* interface) might collaborate with a **PricingTable** component that contains job pricing information. The *checkPriority()* operation (part of the *initiateJob* interface) might collaborate with a **JobQueue** component to determine the types and priorities of jobs currently awaiting production.

Component level design: OO view



- This elaboration activity is applied to every component as part of the architectural design.
- further elaboration is applied to each attribute, operation, and interface
- Data structures
- Algorithmic details

The data structures appropriate for each attribute must be specified.

The algorithmic detail required to implement the processing logic associated with each operation is designed.

Finally, the mechanisms required to implement the interface are designed

For object-oriented software, this may encompass the description of all messaging that is required to effect communication between objects within the system.

The view of component: traditional

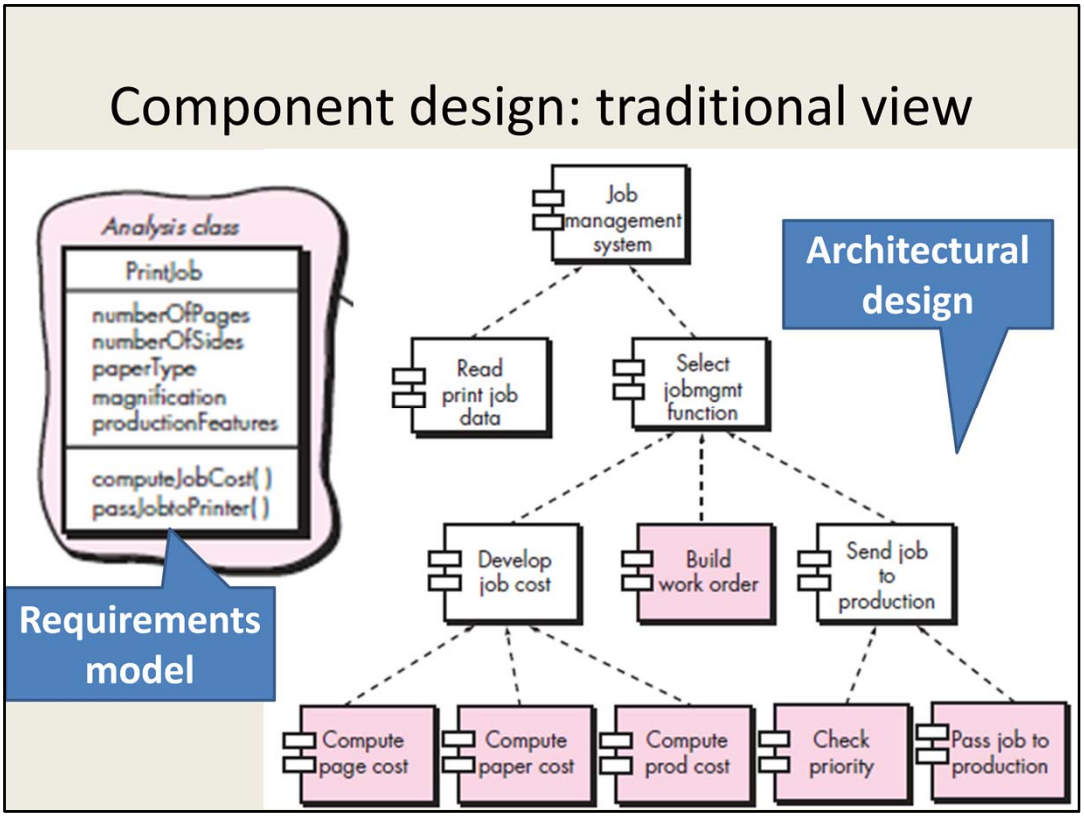
- **is a functional element of a program (module)** that incorporates a component contains
  - processing logic
  - the internal data structures that are required to implement the processing logic, and
  - an interface that enables the component to be invoked and data to be passed to it
- Component roles
  - Control / problem domain / infrastructure

A traditional component, also called a ***module,*** resides within the software architecture and serves one of three important roles:

(1) a *control component* that coordinates the invocation of all other problem domain components,

(2) a *problem domain component* that implements a complete or partial function that is required by the customer, or

(3) an *infrastructure component* that is responsible for functions that support the processing required in the problem domain.



components are derived from the analysis model, like in OO view.

the component elaboration element of the analysis model serves as the basis for the derivation

Each component represented in the component hierarchy is mapped into a module hierarchy;

Control components(modules) reside near the top of the hierarchy (program architecture), and

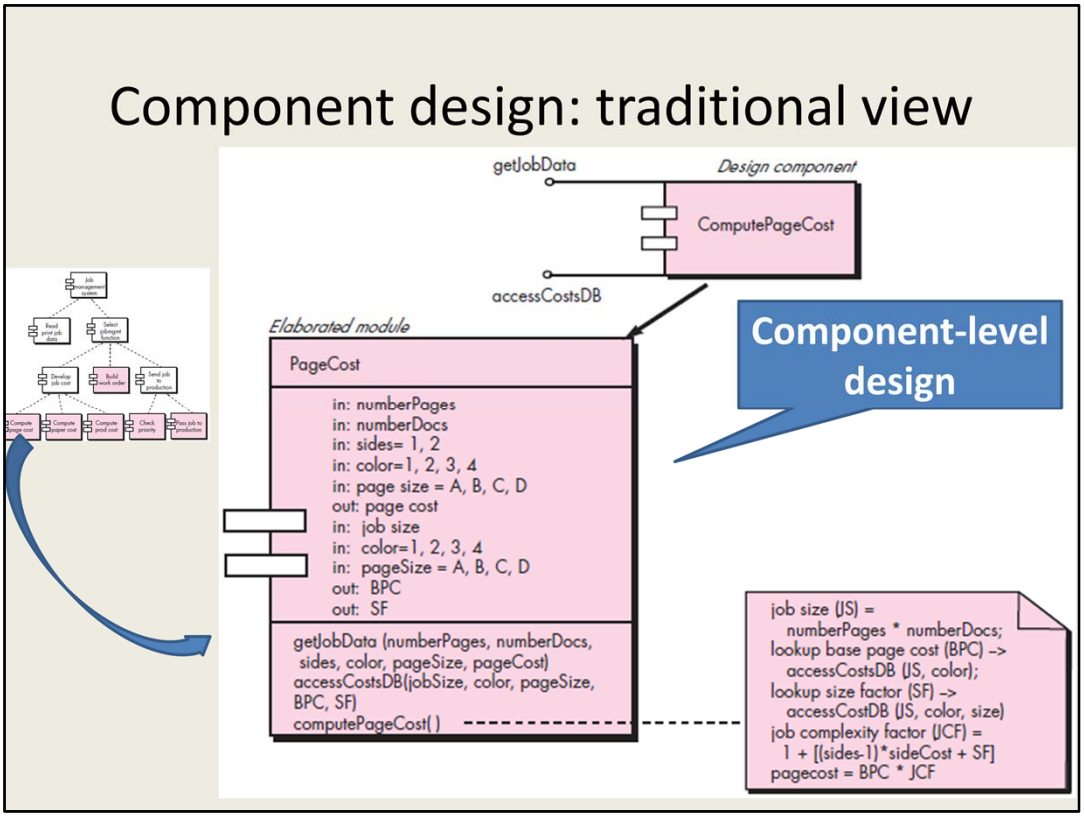
problem domain components tend to reside toward the bottom of the hierarchy.

Each box represents a software component.

Note that the shaded boxes are equivalent in function to the operations defined for the **PrintJob** class

In this case, however, each operation is represented as a separate module that is invoked.

Other modules are used to control processing and are therefore control components.



During component-level design, each module is elaborated

The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented.

The data structures that are used internal to the module are defined.

The algorithm that allows the module to accomplish its intended function is designed using the stepwise refinement approach

The behavior of the module is sometimes represented using a state diagram

consider the module *ComputePageCost.* The intent of this

module is to compute the printing cost per page based on specifications provided by

the customer. Data required to perform this function are: number of pages in the document,

total number of documents to be produced, one- or two-side printing, color requirements,

and size requirements. These data are passed to *ComputePageCost* via the module's interface.

ComputePageCost uses these data to determine a page cost that is based on

the size and complexity of the job—a function of all data passed to the module

via

the interface. Page cost is inversely proportional to the size of the job and directly proportional to the complexity of the job.

The *ComputePageCost* module accesses data by invoking the module *getJobData,* which allows all relevant data to be passed to the component, and a database interface, *accessCostsDB,* which enables the module to access a database that contains all printing costs.

As design continues, the *ComputePageCost* module is elaborated to provide algorithm detail and interface detail

Algorithm detail can be represented using the pseudocode text shown in the figure or with a UML activity diagram.

The interfaces are represented as a collection of input and output data objects or items.

Design elaboration continues until sufficient detail is provided to guide construction of the component.

The view of component: process-related view

- Make use of existing components or design patterns
- Choose them as needed from catalog to populate the architecture design

The object-oriented and traditional views of component-level design assume that the component is being designed from scratch.

That is, we have to create a new component based on specifications derived from the requirements model.

Over the past two decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. In essence, a catalog of proven design or code-level components is made available to us as design work proceeds.

We choose components or design patterns from the catalog and use them to populate the architecture.

Design class-based components

- Focus on:
  - the elaboration of problem domain specific classes
  - the definition and refinement of infrastructure classes contained in the requirements model

Component-level design draws on information developed as part of the requirements model and represented as part of the architectural model

For OO software engineering approach, component-level design focuses on the elaboration of problem domain specific classes and the definition and refinement of infrastructure classes contained in the requirements model.

The detailed description of the attributes, operations, and interfaces used by these classes is the design detail required as a precursor to the construction activity.

Design class-based components: principles

- The Open-Closed Principle (OCP).
  - *"A module [component] should be open for extension but closed for modification.*
- The Liskov Substitution Principle (LSP).
  - *"Subclasses should be substitutable for their base classes.*
- Dependency Inversion Principle (DIP).
  - *"Depend on abstractions. Do not depend on concretions."*
- The Interface Segregation Principle (ISP).
  - *"Many client-specific interfaces are better than one general purpose interface.*

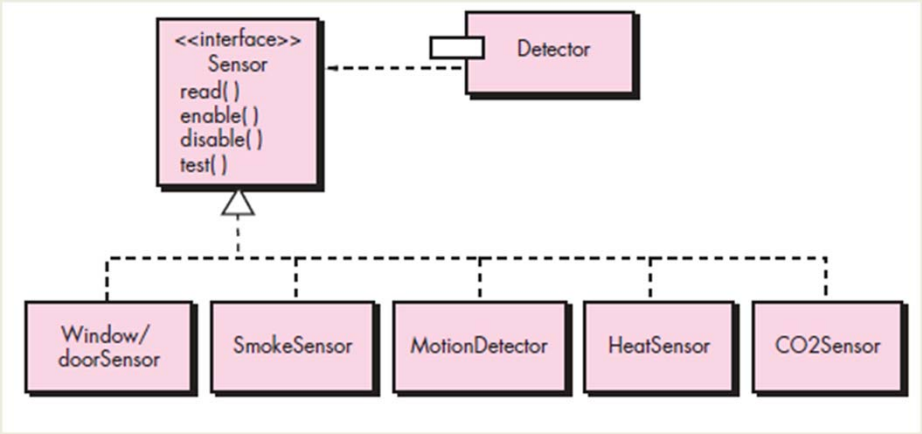
LSP: a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.

DIP: As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication. The more

a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

Design class-based components: principles

- The Open-Closed Principle (OCP).



Seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design.

Stated simply, you should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without

the need to make internal (code or logic-level) modifications to the component itself.

To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

For example, assume that the *SafeHome* security function makes use of a **Detector**

class that must check the status of each type of security sensor. It is likely that as time

passes, the number and types of security sensors will grow. If internal processing logic

is implemented as a sequence of if-then-else constructs, each addressing a different

sensor type, the addition of a new sensor type will require additional internal

processing

logic (still another if-then-else). This is a violation of OCP.

One way to accomplish OCP for the **Detector** class is illustrated in Figure 10.4.

The *sensor* interface presents a consistent view of sensors to the detector component.

If a new type of sensor is added no change is required for the **Detector** class (component). The OCP is preserved.

Design class-based components: principles

- The Interface Segregation Principle (ISP).
- SafeHome security
  - Needs Floorplan for
    - *placeDevice(), showDevice(), groupDevice(),* and *removeDevice().*
- SafeHome surveillance
  - Needs Floorplan for
    - *placeDevice(), showDevice(), groupDevice(),* and *removeDevice(),* along with *showFOV()* and *showDeviceID().*

ISP: you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client.

The *SafeHome* surveillance function uses the four operations noted for security, but also requires special operations to manage cameras: *showFOV()* and *showDeviceID().* The ISP suggests that client components from the two *SafeHome* functions have specialized interfaces defined for them.

Design class-based components: principles

- The Release Reuse Equivalency Principle (REP).
  - *"The granule of reuse is the granule of release."*
- The Common Closure Principle (CCP).
  - *"Classes that change together belong together."*
- The Common Reuse Principle (CRP).
  - *"Classes that aren't reused together should not be grouped together."*

In many cases, individual components or classes are organized into subsystgems or packages

REP: When classes or components are designed for reuse, Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.

CCP: Classes should be packaged cohesively; when classes are packaged as part of a design, they should address the same functional or behavioral

area. When some characteristic of that area must change, it is likely that only those classes within the package will require modification

CRP: only classes that are reused together should be included within a package.

Design class-based components: pragmatic guidelines

- Components
  - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
- Interfaces
  - Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OCP)
- Dependencies and Inheritance
  - it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

The names of architecture components including control component and problem domain component shall come from problem domain.

Infrastructure components can come for a technical background.

Cohesion in component level design

- What does it mean?
  - Conventional view: the "single-mindedness" of a component
  - OO view: a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.
- Level of cohesion
  - Functional
  - Layer
  - Communicational

In Chapter 8, I described cohesion as the "single-mindedness" of a component.

Within the context of component-level design for object-oriented systems, *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.

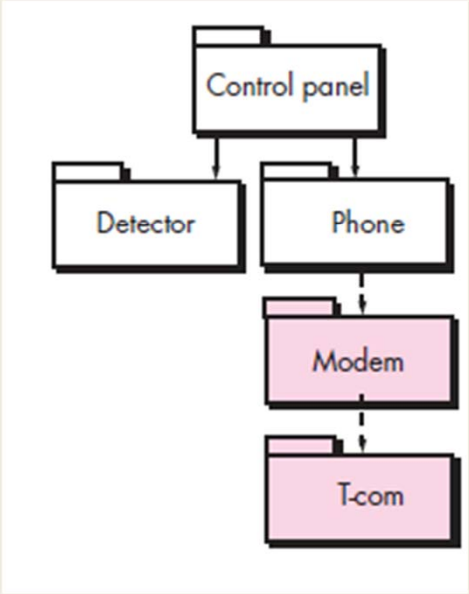
Functional: Exhibited primarily by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result.

Layer: Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

Communicational: All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

Cohesion in component level design

- Layer cohesion
  - Access is only downward



this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

The *SafeHome* security function requirement to make an outgoing phone call if an alarm is sensed.

The shaded packages contain infrastructure components.

Access is from the control panel package **downward**.

Coupling in component level design

- What is it?
  - Conventional view: The degree to which a component is connected to other components and to the external world
  - OO view: a qualitative measure of the degree to which classes are connected to one another
- Level of coupling
  - Content
  - Control
  - External

Content coupling: Occurs when one component "surreptitiously modifies data that is internal to another component"

This violates information hiding—a basic design concept.

Control coupling: Occurs when operation *A()* invokes operation *B()* and passes a control flag to *B.*

The control flag then "directs" logical flow within *B.*

The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes

External coupling: Occurs when a component communicates or collaborates with infrastructure components

(e.g., operating system functions, database capability, telecommunication functions).

Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system

Summary

- The concept of Component
  - Traditional view
  - OO view
  - Process-related view
- Component design
  - OO approach
  - Traditional approach
  - Process-related approach