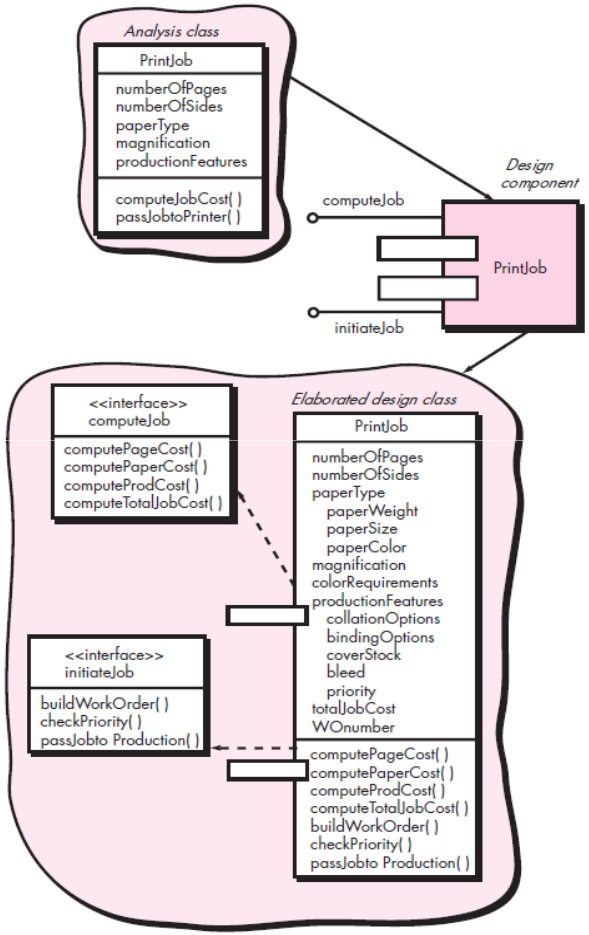
Component Level Design

Agenda

- Designing class based components
  - Basic design principles(OCP, LSP, DIP,ISP, REP, CCP,CRP)
  - Component level design guidelines(components, interfaces, dependencies and inheritance)
  - Cohesion (Functional, layer, communicational)
  - Coupling (content, common, control, stamp, data, routine call, type use, Inclusion or import, external)
- Traditional components

Component

- A component is a modular building block for computer software. UML defines a component as “a modular, deployable and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”
- Play a role to develop a system.
- Reside in software architecture and communicate and collaborate with other components and with entities that exist outside the boundaries of the software.
- Component contains a set of collaborating classes in OO view.

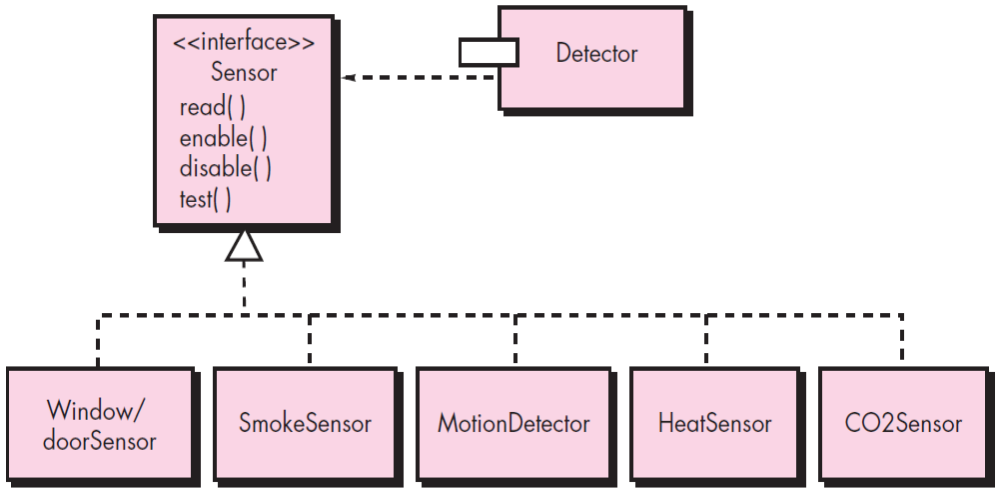


Designing Class based Components

- 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.
- Basic Design Principles :
  - The Open-Closed Principle (OCP),
  - The Liskov Substitution Principle (LSP),
  - Dependency Inversion Principle (DIP),
  - The Interface Segregation Principle (ISP),
  - The Release Reuse Equivalency Principle (REP),
  - The Common Closure Principle (CCP),
  - The Common Reuse Principle (CRP).

Designing Class based Components (Contd..)

- Open-Closed Principle (**OCP)**- *"A module [component] should be open for extension but closed for modification".* 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.



Designing Class based Components (Contd..)

- **Liskov Substitution Principle** (**LSP)-** *"Subclasses should be substitutable for their base classes".* A "contract" is a *precondition that must be true before the component uses a base class* and a *post-condition that should be true after the component uses a base class. When* you create derived classes, be sure they conform to the pre- and post-conditions.
- **Dependency Inversion Principle** (**DIP)-** *"Depend on abstractions. Do not depend on concretions".* 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.
- **Interface Segregation Principle (ISP)-** *"Many client-specific interfaces are better than one general purpose interface"* . ISP create a specialized interface to serve each major category of clients. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

Designing Class based Components (Contd..)

- **Release Reuse Equivalency Principle** (**REP) -***"The granule of reuse is the granule of release".* It is advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.
- **Common Closure Principle (CCP) -** *"Classes that change together belong together."* Classes should be packaged cohesively for more effective change control and release management.
- **Common Reuse Principle (CRP)** - *"Classes that aren't reused together should not be grouped together". O*nly classes that are reused together should be included within a package. When one or more classes within a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operates without incident.

Designing Class based Components (Contd..)

- Component-Level Design Guidelines :
- Design guidelines can be applied to components, their interfaces, and the dependencies and inheritance that have an impact on resultant design.
  - Component : Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.
  - Interfaces : Interfaces provide important information about communication and collaboration.
  - Dependencies and Inheritance : For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

Designing Class based Components (Contd..)

- Cohesion :
- 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.
- Different types of cohesion : Functional, Layer, Communicational.
  - Functional : this level of cohesion occurs when a component performs a targeted computation and then returns a result.
  - Layer : 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.
- Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.

Designing Class based Components (Contd..)

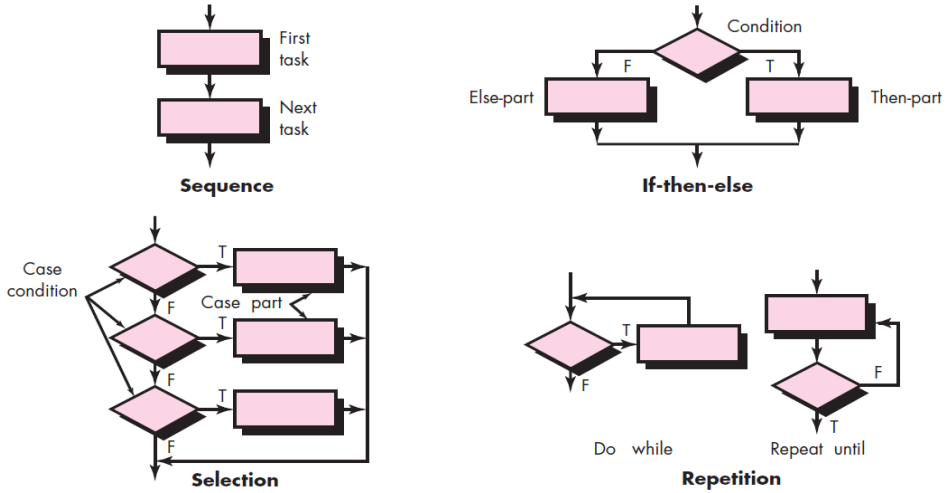
- *Coupling is a qualitative measure of the degree to which classes are connected to* one another. As classes (and components) become more interdependent, coupling increases. Coupling categories :
- **Content coupling. Occurs when one component "surreptitiously modifies** data that is internal to another component". This violates information hiding—a basic design concept.
- **Common coupling. Occurs when a number of components all make use of** a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.
- **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. If this is overlooked, an error will result.
- **Stamp coupling. Occurs when ClassB is declared as a type for an argument** of an operation of **ClassA. Because ClassB is now a part of the definition** of **ClassA, modifying the system becomes more complex.**
- **Data coupling. Occurs when operations pass long strings of data arguments.** The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.
- **Routine call coupling. Occurs when one operation invokes another. This** level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.
- **Type use coupling. Occurs when component A uses a data type defined in** component **B (e.g., this occurs whenever "a class declares an instance variable** or a local variable as having another class for its type". If the type definition changes, every component that uses the definition must also change.
- **Inclusion or import coupling. Occurs when component A imports or includes** a package or the content of component **B.**
- **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.

Traditional Components

- Initially a set of logical constructs, were proposed. The constructs are sequence, condition and repetition.
- Later, structured constructs were proposed to improve readability, testability, and maintainability.
- The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module. Categories : graphical design notation, tabular design notation, program design language.

Traditional Components

- Graphical design notation : UML, activity diagram allow to represent the element of structured constructs. A box indicate processing step. A diamond represent a logical condition. Arrow show the flow of control.
- Fig illustrates three structured constructs, sequence, selection and repetition.
- Structured constructs can introduce in efficiency when an escape from a set of nested loops or nested conditions is required.But can be resolved using nested flow.



Traditional Components

- Tabular design notation: decision table provide a notation that translates actions and conditions into a tabular form. The table is divided into four sections, each section contain conditions, actions, condition combinations, corresponding actions.

	Rules					
Conditions	**1**	**2**	**3**	**4**	**5**	**6**
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional *x* percent discount		✓		✓		✓

Traditional Components

- Program Design Language : PDL uses free-form natural language to express logical structure of a PL.
- PDL constructs are as follows:
  - Component definition
  - Interface definition
  - Data declaration
  - Block structuring
  - Condition constructs
  - Repetition constructs
  - Input-output constructs
  - Multitasking processing
  - Concurrent processing
  - Interrupt handling,
  - Interprocess synchronization etc.

```
component alarmManagement;
The intent of this component is to manage control panel switches and input from sensors by
type and to act on any alarm condition that is encountered.
    set default values for systemStatus (returned value), all data items
    initialize all system ports and reset all hardware
    check controlPanelSwitches (cps)
        if cps = "test" then invoke alarm set to "on"
        if cps = "alarmOff" then invoke alarm set to "off"
        if cps = "newBoundingValue" then invoke keyboardInput
        if cps = "burglarAlarmOff" invoke deactivateAlarm;
        •
        •
        •
        default for cps = none
    reset all signalValues and switches
    do for all sensors
        invoke checkSensor procedure returning signalValue
        if signalValue > bound [alarmType]
            then phoneMessage = message [alarmType]
                set alarmBell to "on" for alarmTimeSeconds
                set system status = "alarmCondition"
                parbegin
                    invoke alarm procedure with "on", alarmTimeSeconds;
                    invoke phone procedure set to alarmType, phoneNumber
                endpar
            else skip
        endif
    enddofor
end alarmManagement
```