❖ Designing class based components

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

❖ The Interface Segregation Principle (ISP) – "Many client-specific interfaces are better than one general purpose interface". ISP suggests that you should create a specialized interface to serve each major category of clients.

❖ The Release Reuse Equivalency Principle (REP) – "The granule of reuse is the granule of release". When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it.

❖ The Common Closure Principle (CCP) – "Classes that change together belong together." when classes are packaged as part of a design, they should address the same functional or behavioral area.

❖ The Common Reuse Principle (CRP) – "Classes that aren't reused together should not be grouped together".

2. Component-Level Design Guidelines - Ambler suggests the following 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. You can choose to use stereotypes to help identify the nature of components at the detailed design level. Ex: <> might be used to identify an infrastructure component.

• Interfaces - Interfaces provide important information about communication and collaboration. Ambler recommends

1 lollipop representation of an interface should be used in UML when diagrams grow complex

2 interfaces should flow from the left-hand side of the component

3 only those interfaces that are relevant to the component under consideration should be shown

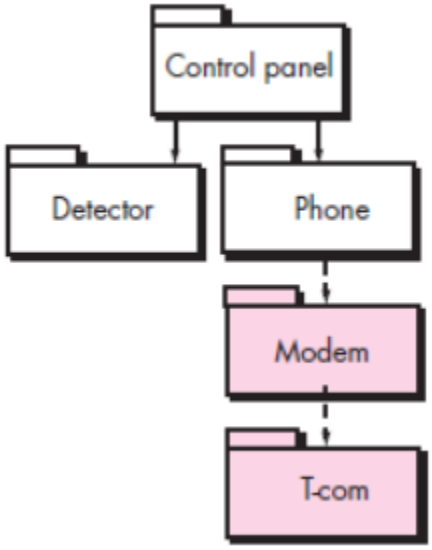
• Dependencies and Inheritance – dependencies from left to right and inheritance from bottom to top.

3. Cohesion – Described as the "single-mindedness" of a component. Different types of Cohesion.

• 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.

FIGURE 10.5
Layer cohesion



• Communicational - All operations that access the same data are defined within one class.

4. Coupling - 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. An important objective in component-level design is to keep coupling as low as is possible. Different types of coupling categories are

• Content coupling – Occurs when one component "surreptitiously modifies data that is internal to another component". This violates information hiding.

• Common coupling – Occurs when a number of components make use of a global variable. It leads to uncontrolled error propagation.

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

• Stamp coupling - Occurs when Class B is declared as a type for an argument of an operation of Class A . modification of a system leads to complex problems.

• Data coupling - Occurs when operations pass long strings of data arguments. Testing and maintenance are more difficult.

Routine call coupling - Occurs when one operation invokes another. It does not increase the connectedness of a system.

• Type use coupling – Occurs when component A uses a data type defined in component B. 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. It should be limited to a small number of components or classes within a system.

❖ Conducting component-level design

The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1 - Identify all design classes that correspond to the problem domain.

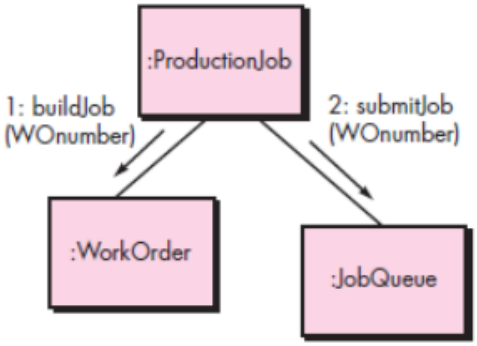
Step 2 - Identify all design classes that correspond to the infrastructure domain.

Step 3 - Elaborate all design classes that are not acquired as reusable components.

Step 3a - Specify message details when classes or components collaborate.

Ex:

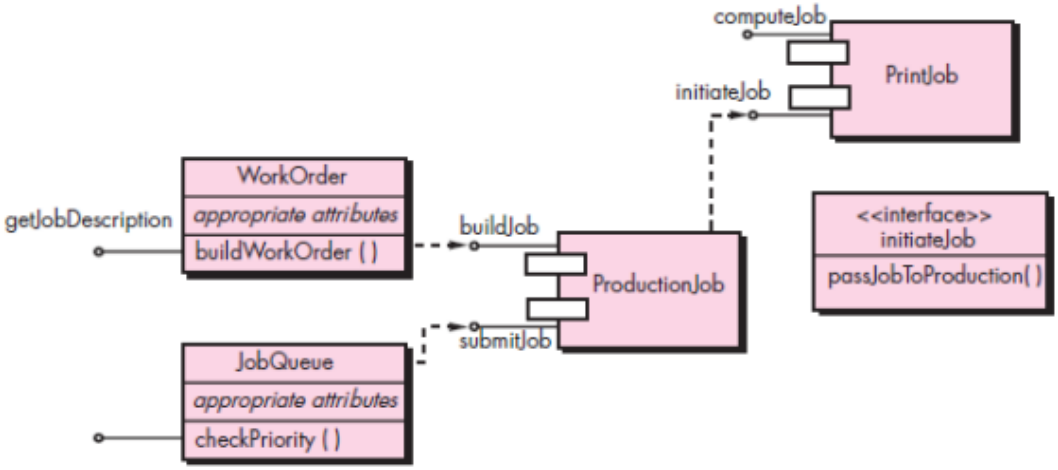
FIGURE 10.6
Collaboration diagram with messaging



Step 3b - Identify appropriate interfaces for each component.

Ex: The interface initiates Job

FIGURE 10.7 Refactoring interfaces and class definitions for PrintJob

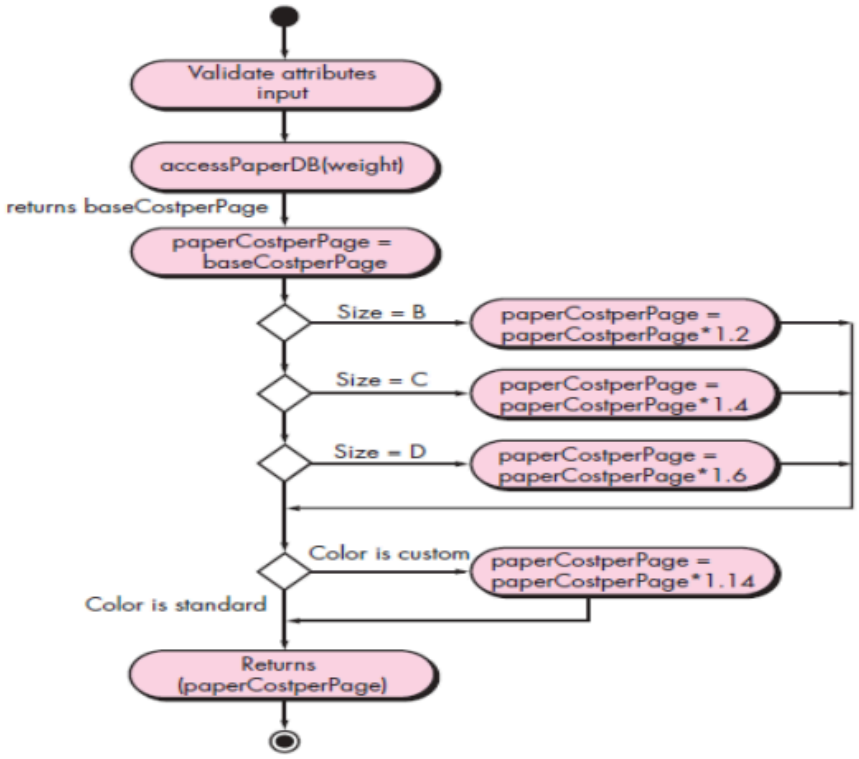


Step 3c - Elaborate attributes and define data types and data structures required to implement them.

Step 3d - Describe processing flow within each operation in detail.

Ex: UML activity diagram for compute Paper Cost ().

FIGURE 10.8
UML activity diagram for *compute-PaperCost()*



Step 4 - Describe persistent data sources (databases and files) and identify the classes required to manage them.

Step 5 - Develop and elaborate behavioral representations for a class or component

Step 6 -Elaborate deployment diagrams to provide additional implementation detail

Step 7 - Refactor every component-level design representation and always consider alternatives.