Chapter 14

Component-Level Design

Slide Set to accompany

Software Engineering: A Practitioner's Approach

by Roger S. Pressman

Table of Contents

14.1 What is a Component?

- OMG UML Specification [OMG01]

  "… a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

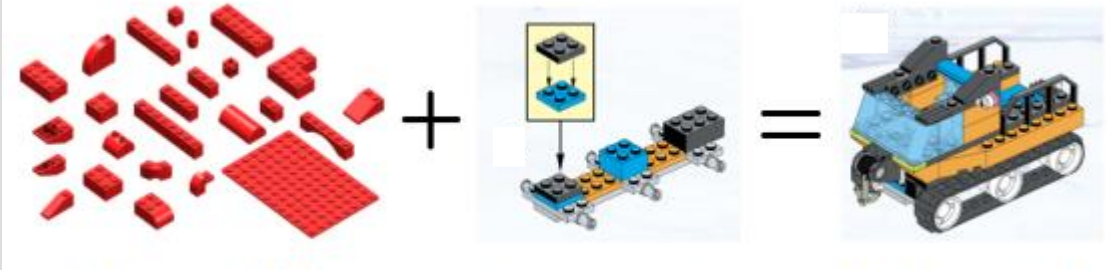
- OO view

  - *A set of collaborating classes*

- Conventional view:

  - The processing logic
  - The internal data structures
    - implement the processing logic
  - An interface that enables

    the component to be invoked and data to be passed to it.

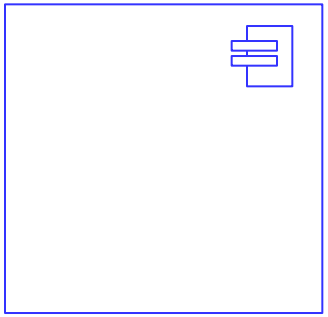


http://www.abse.info/facts/fact21-create-applications-building-blocks.html

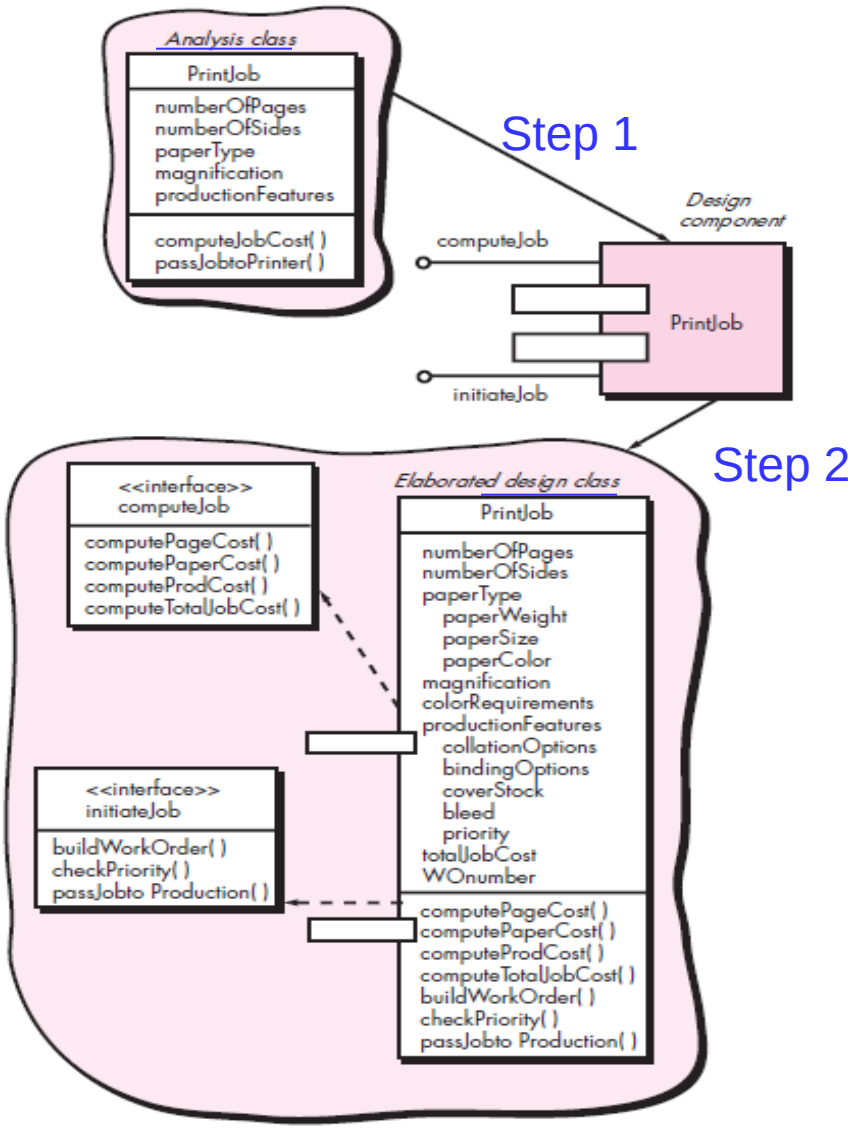
14.1.1 An Object-Oriented View

FIGURE 10.1

Elaboration of a design component



UML 2.0 notation for component



14.1.2 The Traditional View

FIGURE 10.2

Structure chart for a traditional system

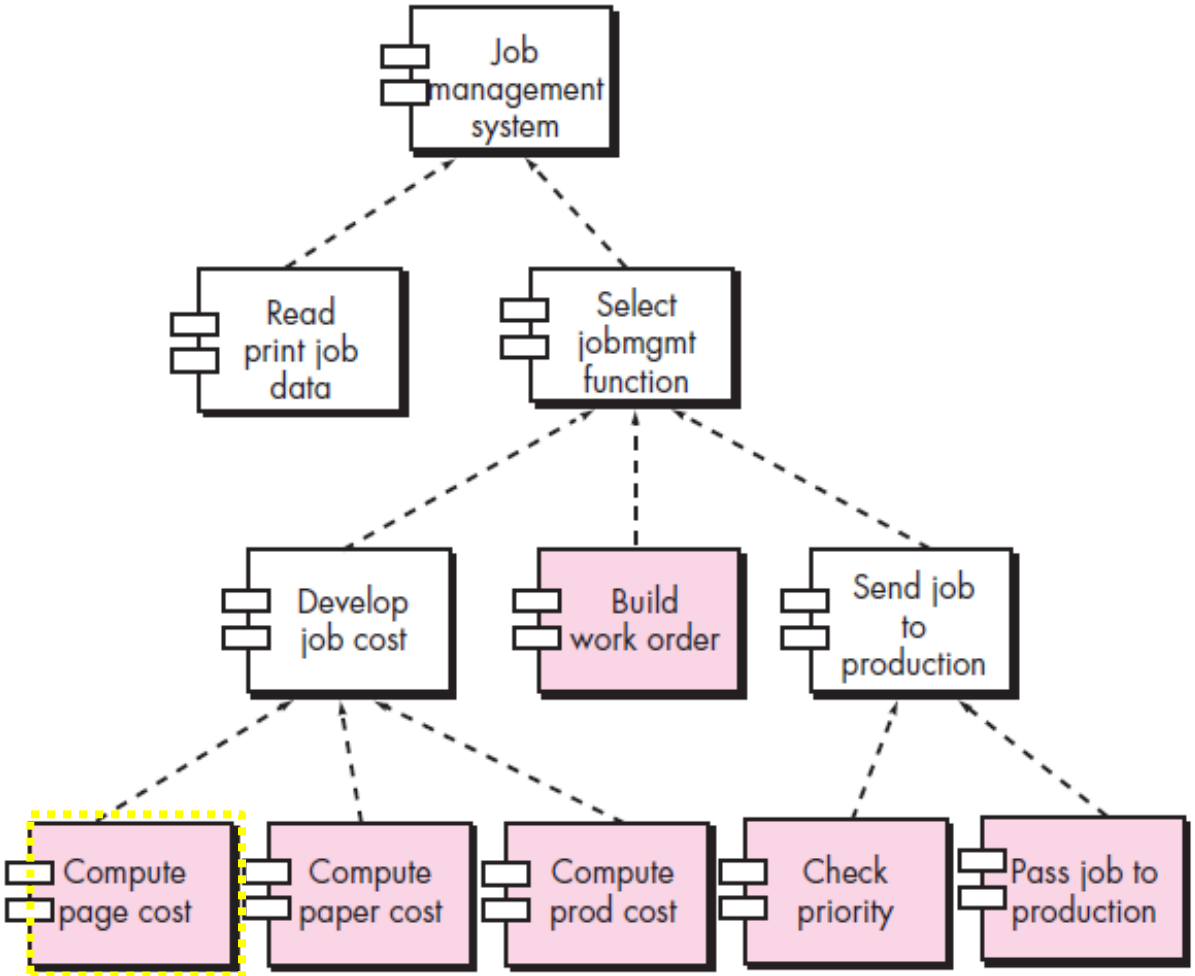
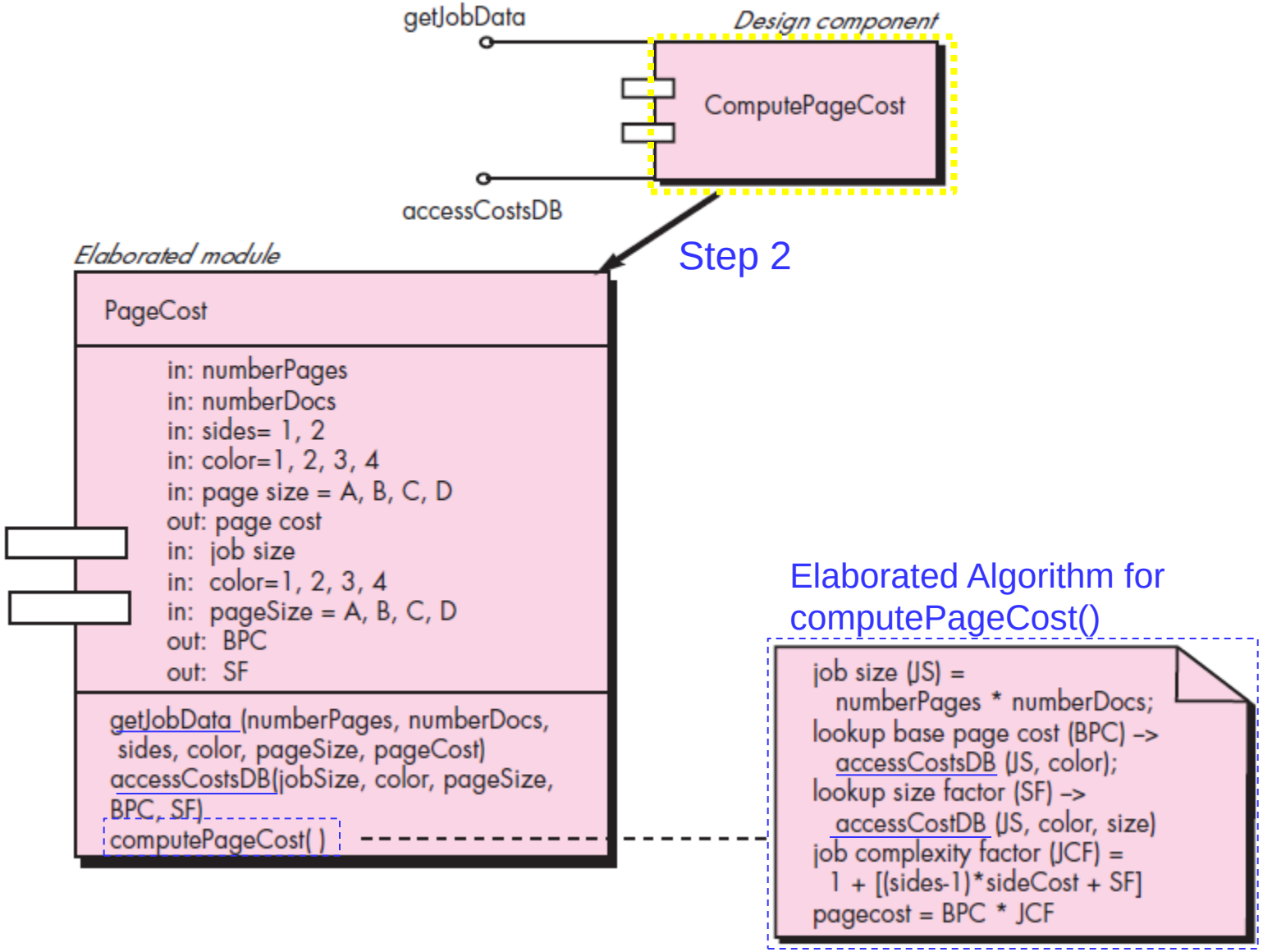


FIGURE 10.3 Component-level design for *ComputePageCost*



14.2 Designing Class-Based Components

- When an OO software engineering approach is chosen, component-level design focuses on:

  Step 1a. Elaboration of problem domain specific classes

  Step 1b. Definition and refinement of infrastructure classes in the requirements model.

  (as we have seen in Section 14.1.1.)

14.2.1 Basic Design Principles

- SOLID OO Design Principles
  - **S**ingle Responsibility Principle(SRP)
  - **O**pen-Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (ISP)
  - **D**ependency Inversion Principle (DIP)

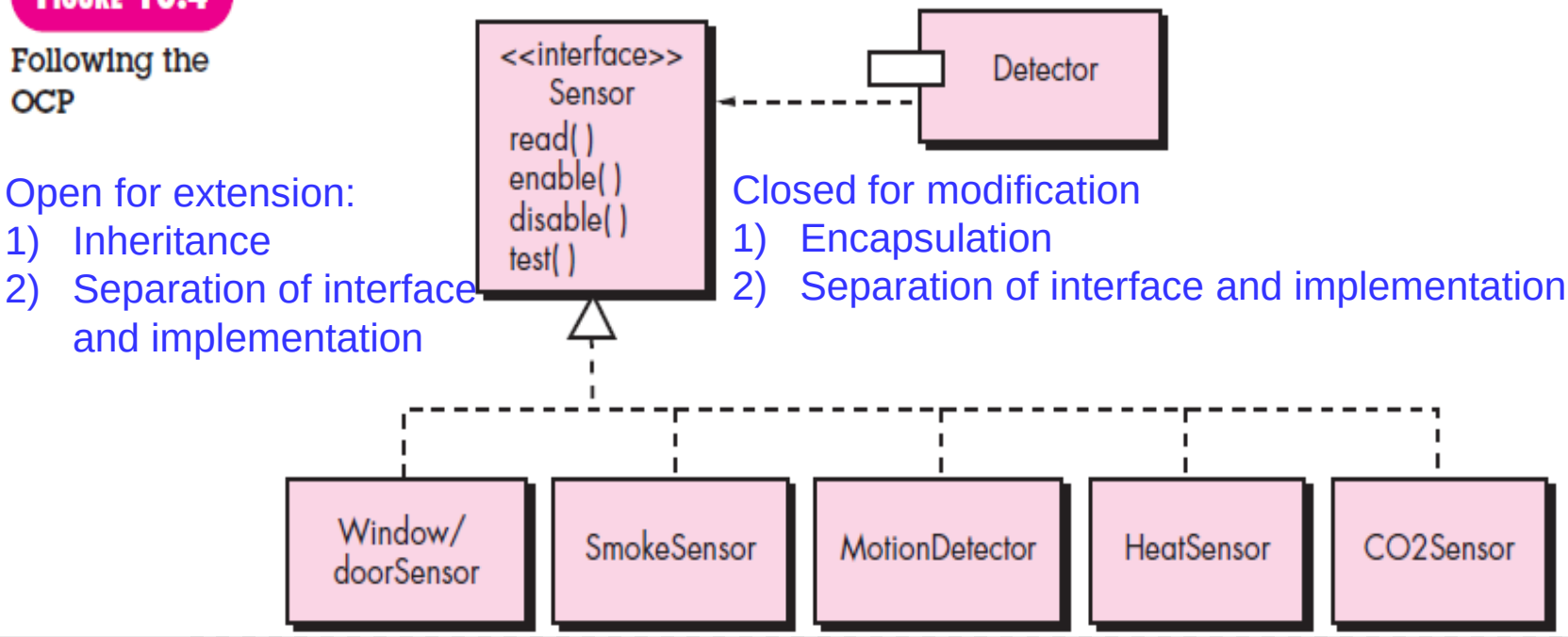
Source: Martin, R., "Design Principles and Design Patterns," downloaded from http:www.objectmentor.com, 2000.

SOLID Principles (1/2)

- Single Responsibility Principle(SRP). A class should have only one job.
- Open-Closed Principle (OCP). "A module [component] should be open for extension but closed for modification.

FIGURE 10.4

Following the OCP



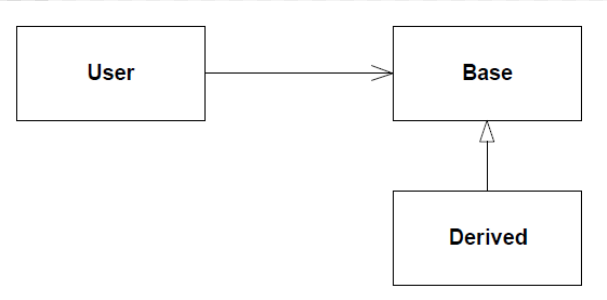
Open for extension:
1) Inheritance
2) Separation of interface and implementation

Closed for modification
1) Encapsulation
2) Separation of interface and implementation

SOLID Principles (2/2)

- Liskov Substitution Principle (LSP):

"Subclasses should be substitutable for their base classes.



A user of a base class should continue to function properly if a derivative of that base class is passed to it.

- Interface Segregation Principle (ISP):

"Many client-specific interfaces are better than one general purpose interface."

- Dependency Inversion Principle (DIP):

"Depend on abstractions. Do not depend on concretions."

Dependency Inversion Principle

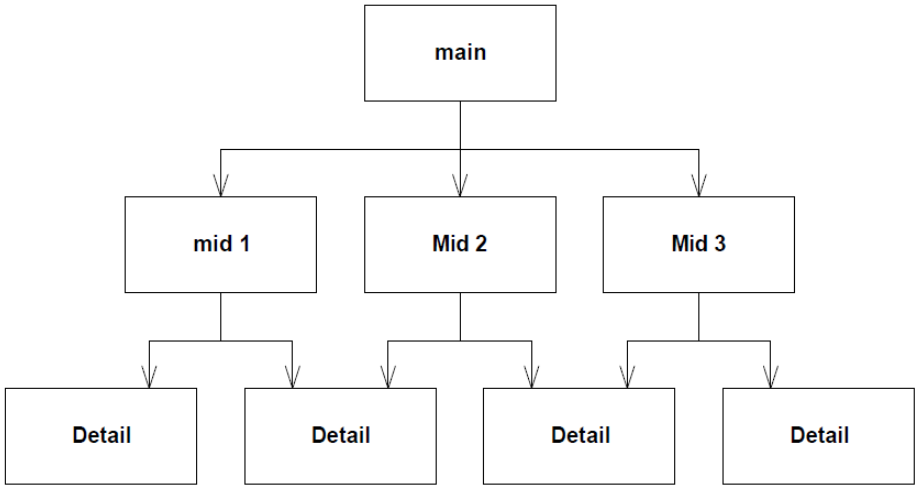


Figure 2-17
Dependency Structure of a Procedural Architecture

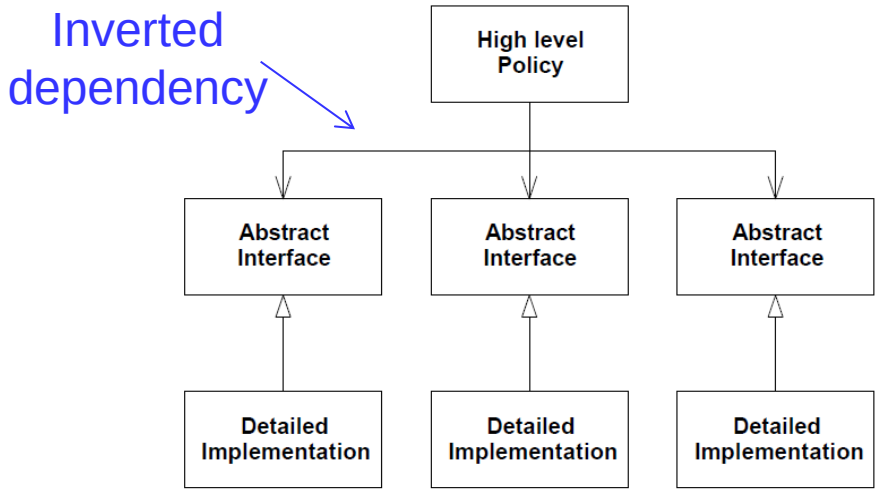


Figure 2-18
Dependency Structure of an Object Oriented Architecture

High level modules depend upon lower level modules.

Problem: Policies of high level modules care little about the details that implement them.

The modules that contain detailed implementation *depend on* abstractions.

➥ Concrete things change a lot whereas abstract things change much less frequently.

Additional Packaging 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."

14.2.2 Component-Level Design (Modeling) Guidelines

- Component Name
  - Naming conventions should be established as part of the architectural model and then refined and elaborated as part of the component-level model

  E.g. Use of stereotypes such as <<infrastructure>>, <<database>>, …

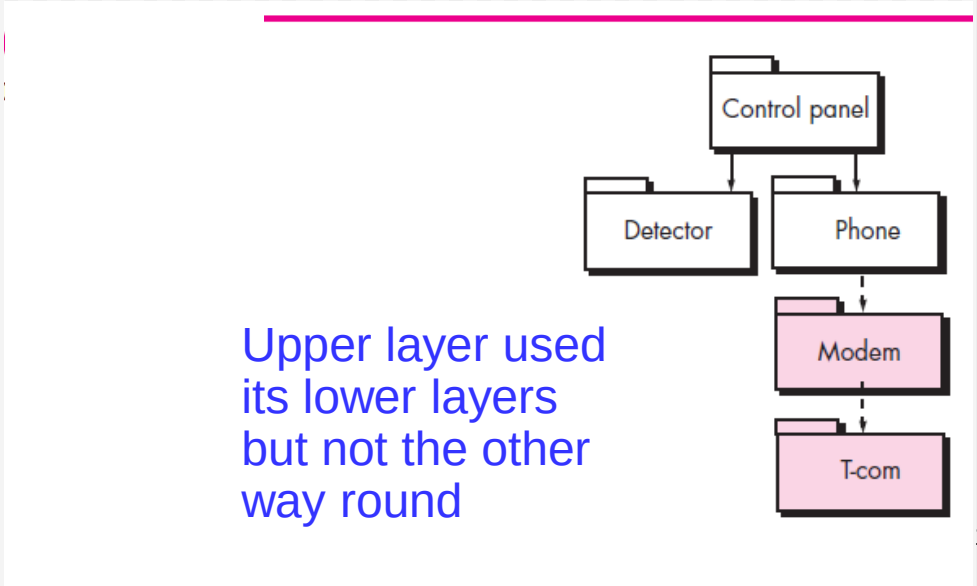
- Interfaces
  - E.g. Use lollipop representation when diagrams grow complex

- Dependencies and Inheritance
  - For readability, model
    - dependencies from left to right and
    - inheritance from bottom (derived classes) to top (base classes).

14.2.3 Cohesion

- Conventional view:
  The “single-mindedness” of a module
- 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
- Levels of cohesion
  - Functional
  - Layer (See Figure)

☛ High cohesion is preferred.



14.2.4 Coupling

- Conventional view

  The degree to which a component is connected to other components and to the external world

- OO view

  The degree to which classes are connected to one another

- Level of coupling

  Content, Control, External, etc.

☛ Low coupling is preferred.

14.3 Conducting Component-Level Design

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

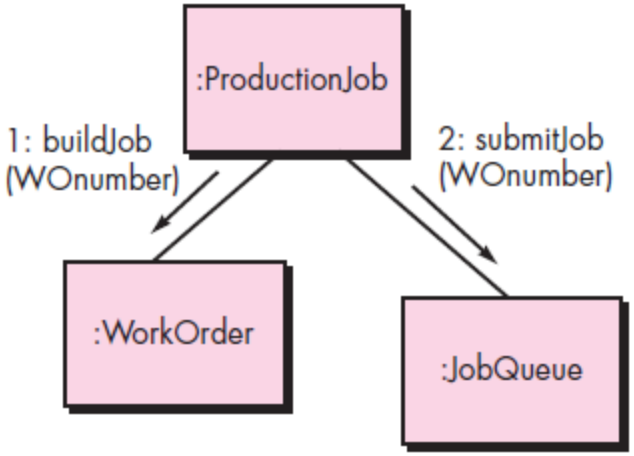
Step 2. Identify all design classes that correspond to the infrastructure domain.

Step 3. Elaborate design classes (that are not acquired as reusable components).

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

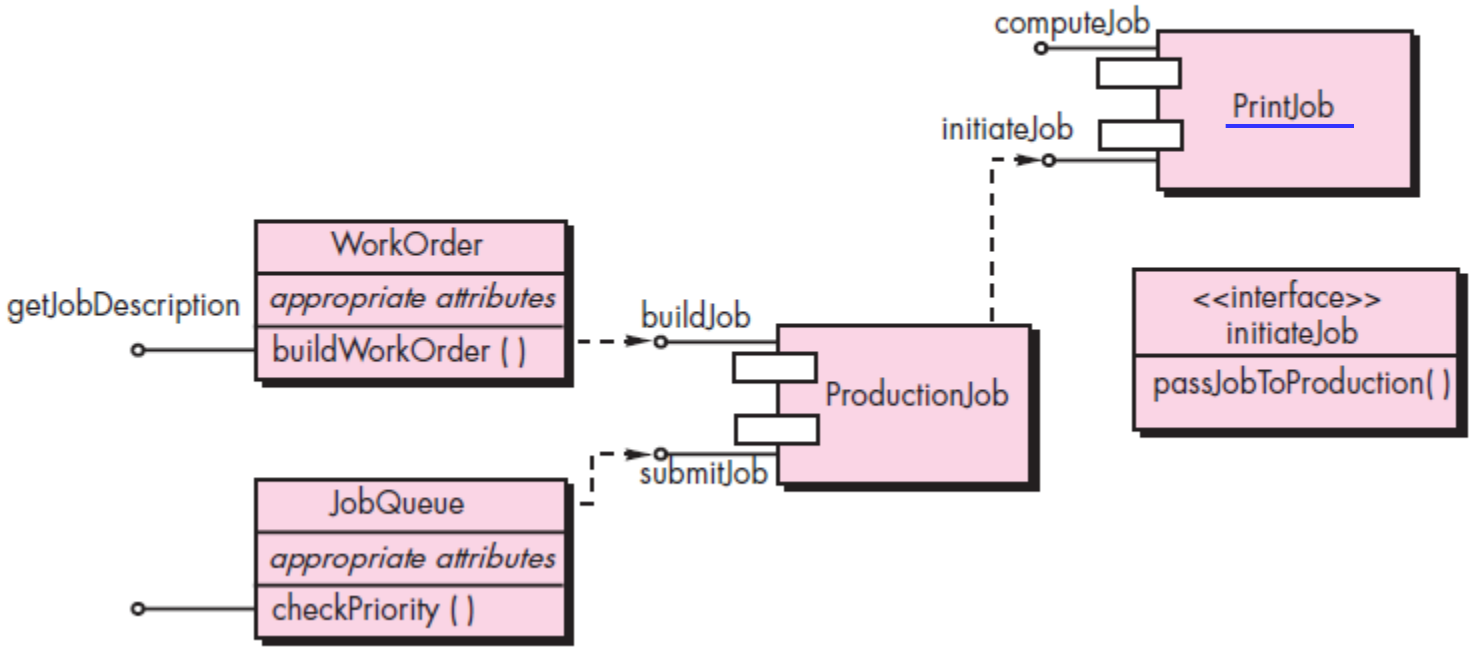
FIGURE 10.6

Collaboration diagram with messaging



Step 3b. Identify appropriate interfaces for each component.

FIGURE 10.7 Refactoring interfaces and class definitions for PrintJob



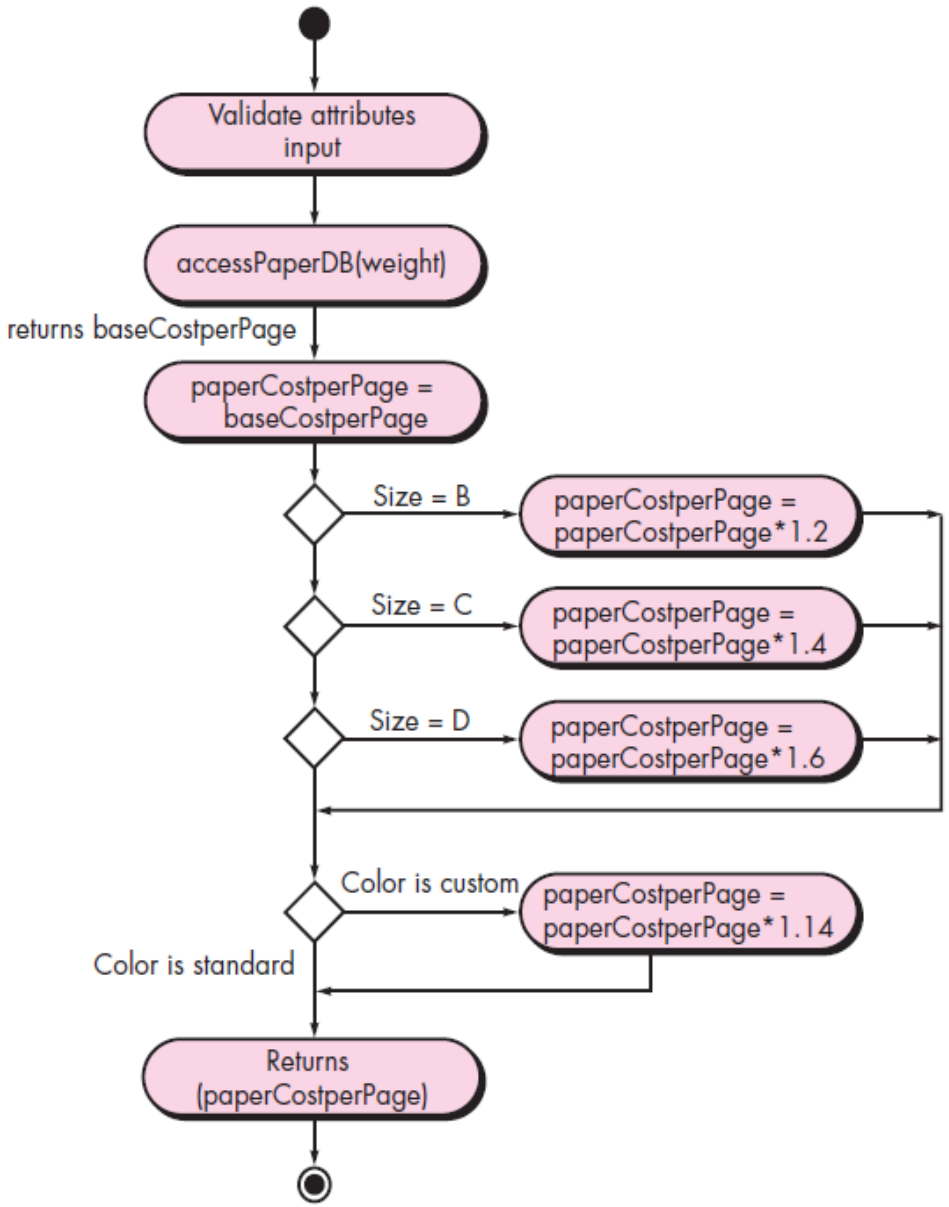
Step 3c. Elaborate attributes and define data types and data structures required to implement them.

Step 3d.

Describe processing flow within each operation in detail.

FIGURE 10.8

UML activity diagram for *compute-PaperCost()*

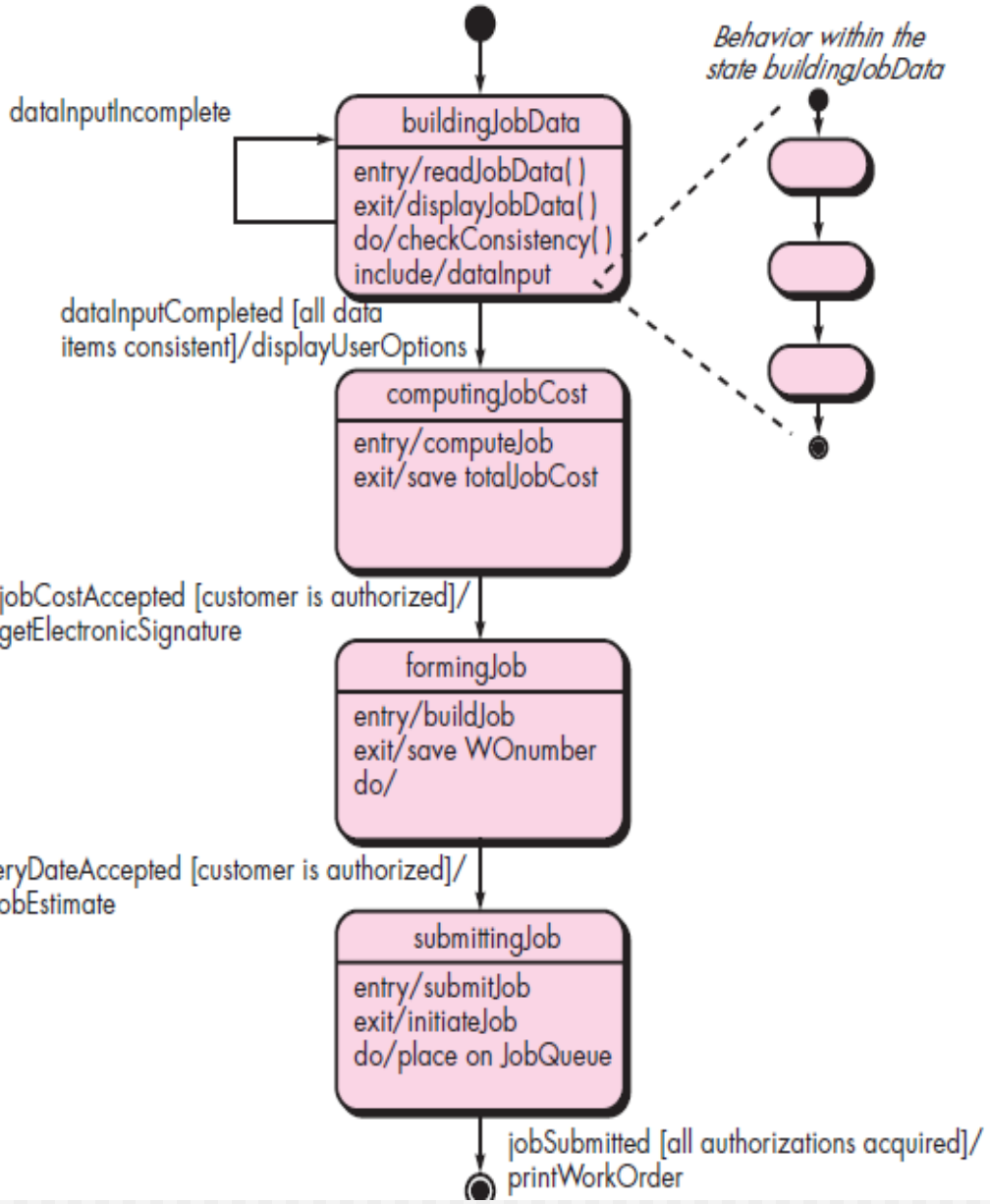


Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.

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

FIGURE 10.9

Statechart fragment for PrintJob class



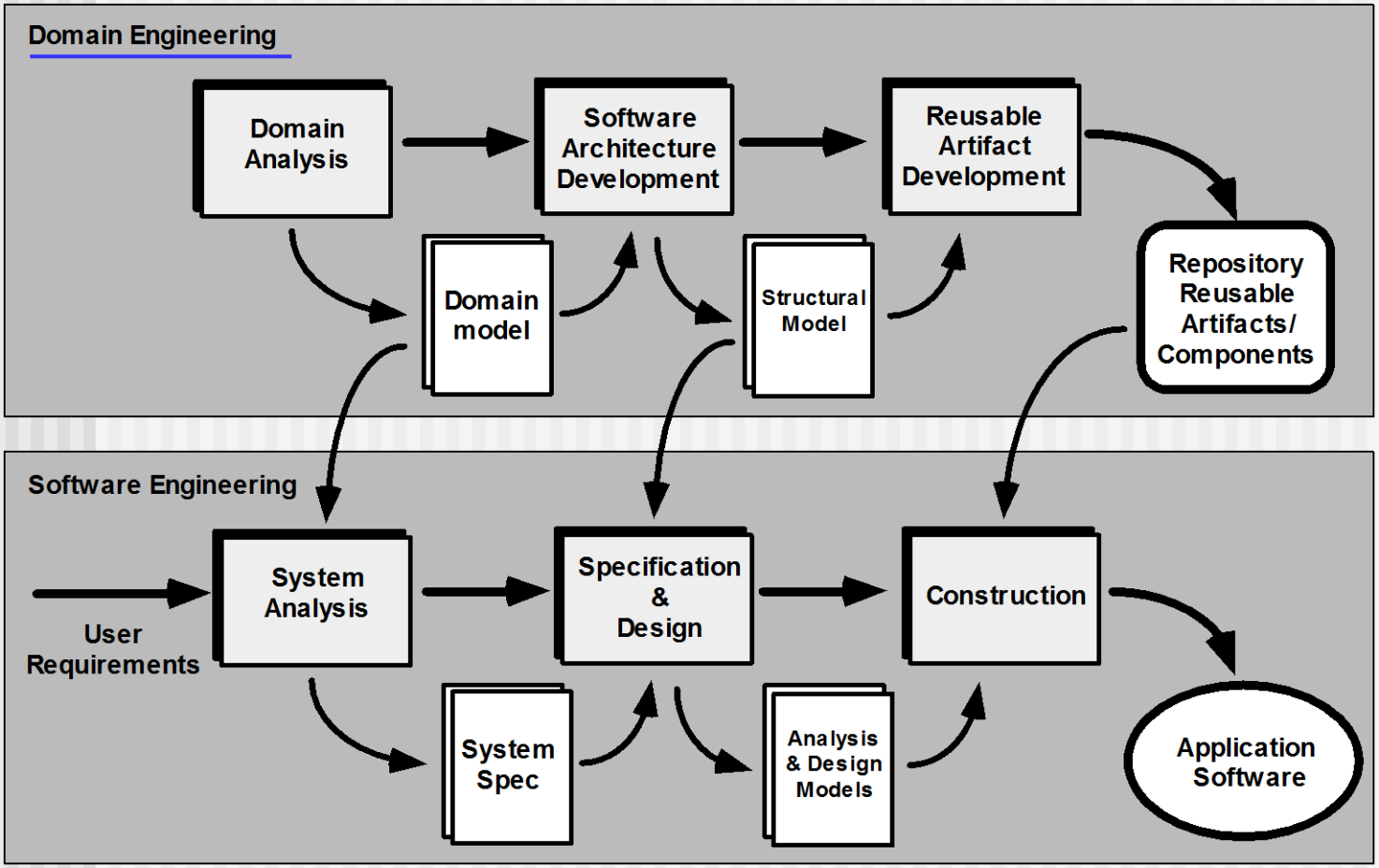
Step 6. Elaborate deployment diagrams to provide additional implementation detail.

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

14.7 Component-Based Development

- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirements?
  - Are internally-developed reusable components available to implement the requirements?
  - Are the interfaces for available components compatible within the architecture of the system to be built?

The CBSE Process



Component-Based SE

- A library of components must be available

- Components should have a consistent structure
- A standard should exist,

  Examples OMG/CORBA, Microsoft COM, Sun JavaBeans

14.7.1 Domain Engineering

- Identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain.

- The steps of *domain analysis*:
  1. Define the domain to be investigated.
  2. Categorize the items extracted from the domain.
  3. Collect a representative sample of applications in the domain.
  4. Analyze each application in the sample.
  5. Develop an analysis model for the objects.

Identifying Reusable Components

- Is component functionality required on future implementations?
- How common is the component's function within the domain?
- Is there duplication of the component's function within the domain?
- Is the component hardware-dependent?
- Does the hardware remain unchanged between implementations?
- Can the hardware specifics be removed to another component?
- Is the design optimized enough for the next implementation?
- Can we parameterize a non-reusable component so that it becomes reusable?
- Is the component reusable in many implementations with only minor changes?
- Is reuse through modification feasible?
- Can a non-reusable component be decomposed to yield reusable components?
- How valid is component decomposition for reuse?

14.7.2 Component Qualification, Adaptation and Composition

- The existence of reusable components does not guarantee that they can be integrated easily or effectively into a new application.

- CBSE Activities:
  - Component qualification
  - Component adaptation
  - Component composition

Component Qualification

To ensure that a candidate component is suitable, you must consider:

- application programming interface (API)
- development and integration tools required by the component
- run-time requirements including resource usage (e.g., memory or storage), timing or speed, and network protocol
- service requirements including operating system interfaces and support from other components
- security features including access controls and authentication protocol
- embedded design assumptions including the use of specific numerical or non-numerical algorithms
- exception handling

Component Adaptation

"Easy integration" is achieved if:

(1) consistent methods of resource management have been implemented for all components;

(2) common activities such as data management exist for all components, and

(3) interfaces within the architecture and with the external environment have been implemented in a consistent manner.

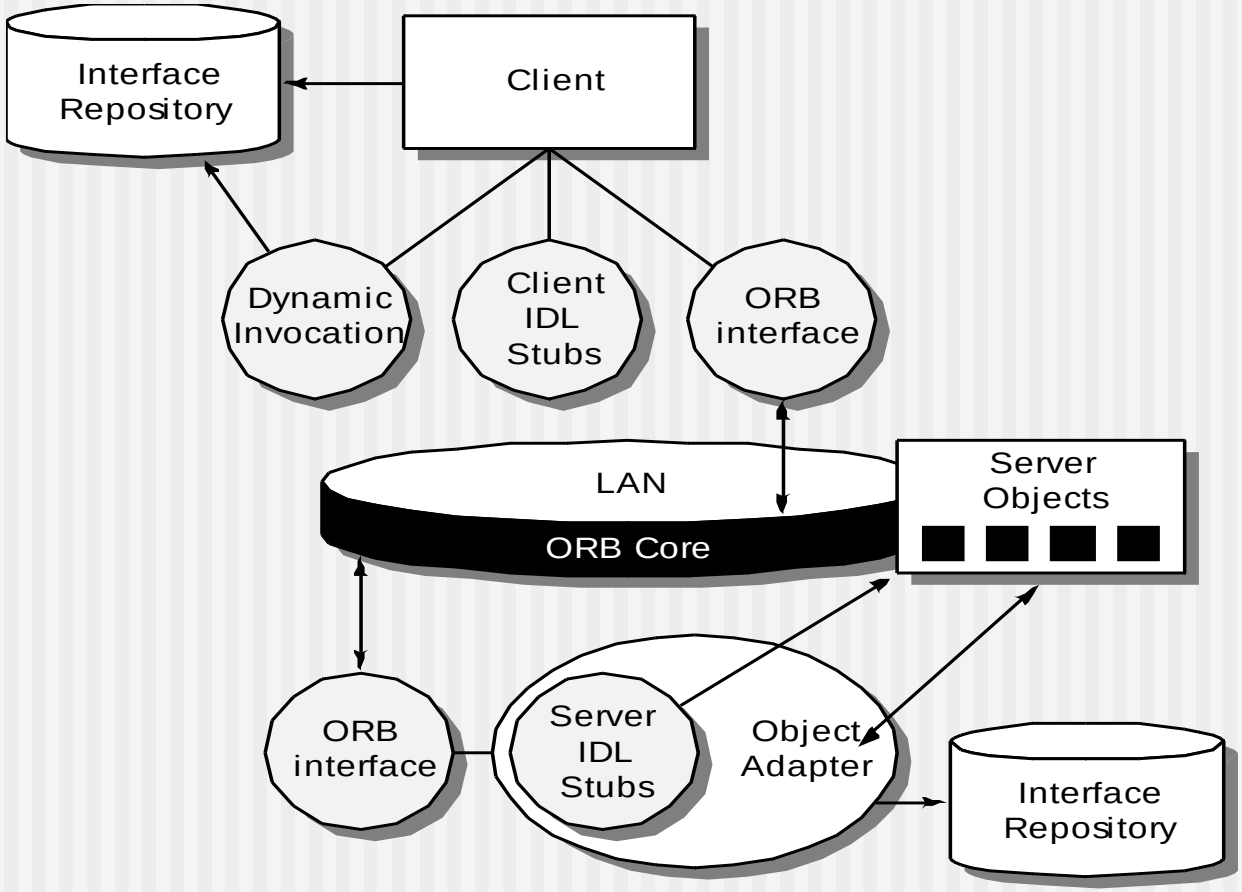
Component Composition

- An infrastructure must be established to bind components together
- Architectural ingredients for composition include:
  - Data exchange model
  - Automation
  - Structured storage
  - Underlying object model

Standards for component software

- OMG/ CORBA
  - The Object Management Group has published a *common object request broker architecture* (OMG/CORBA).
  - An object request broker (ORB) provides services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.
  - Integration of CORBA components (without modification) within a system is assured if an interface definition language (IDL) interface is created for every component.
  - Objects within the client application request one or more services from the ORB server. Requests are made via an IDL or dynamically at run time.
  - An interface repository contains all necessary information about the service's request and response formats.

ORB Architecture



- Microsoft COM(*component object model*)
  - Provides a specification for using components produced by various vendors within a single application running under the Windows operating system.
  - Encompasses:
    - COM interfaces (implemented as COM objects)
    - A set of mechanisms for registering and passing messages between COM interfaces.

- Sun JavaBeans component system
  - A portable, platform independent CBSE infrastructure developed using Java
  - Encompasses a set of tools, called the *Bean Development Kit* (BDK), that allows developers to
    - analyze how existing Beans (components) work
    - customize their behavior and appearance
    - establish mechanisms for coordination and communication
    - develop custom Beans for use in a specific application
    - test and evaluate Bean behavior.

14.7.3 Architectural Mismatch

- The designers of reusable components often make implicit assumptions about the environment. (E.g. the component control model, component interfaces, architectural infrastructures and so on.)
- If these assumptions are incorrect, architectural mismatch occurs.
- To prevent architectural mismatch, design concepts such as abstraction, hiding, functional independence, refinement and structured programming, etc. can be used.

14.7.4 Analysis and Design for Reuse

- Requirements are compared to descriptions of reusable components. (Specification matching)
- If specification matching points to an existing component that fits the needs of the current application, you can extract the component from a reuse library (repository).
- If there is no match, DFR(design for reuse) can be used to create a new component.

14.7.5 Classifying and Retrieving Components

In a large component repository, tens of thousands of reusable software components reside.

How do you find the one that your need?

- Enumerated classification—components are described by defining a hierarchical structure in which classes and varying levels of subclasses of software components are defined
- Faceted classification—a domain area is analyzed and a set of basic descriptive features are identified
- Attribute-value classification—a set of attributes are defined for all components in a domain area

An ideal description of a component consists of 3C

- **Concept**: what a component accomplishes
- **Content**: how this is achieved
- **Context**: where the component resides within its domain of applicability

☛ 3C should be translated into a concrete specification scheme.

The Reuse Environment

- A component database
  - capable of storing software components and the classification information necessary to retrieve them.
- A library management system
  - provides access to the database.
- A software component retrieval system (e.g., an object request broker)
  - enables a client application to retrieve components and services from the library server.
- CBSE tools

  - support the integration of reused components into a new design or implementation.

Impediments to Reuse

- Few companies have anything that even slightly resembles a comprehensive software reusability plan.
- Although an increasing number of software vendors currently sell tools or components that provide direct assistance for software reuse, the majority of software developers do not use tools or components.
- Relatively little training is available to help software engineers and managers understand and apply reuse.
- Many practitioners believe that reuse is “more trouble than it’s worth.”
- Many companies encourage of software development methodologies which do not facilitate reuse
- Few companies provide incentives to produce reusable program components.