CSE4006 Software Engineering

09. Component-Level Design

Scott Uk-Jin Lee

Department of Computer Science and Engineering
Hanyang University ERICA Campus

1st Semester 2015

lab(se);

What is Component

- OMG UML Specification defines a component as

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

- OO view

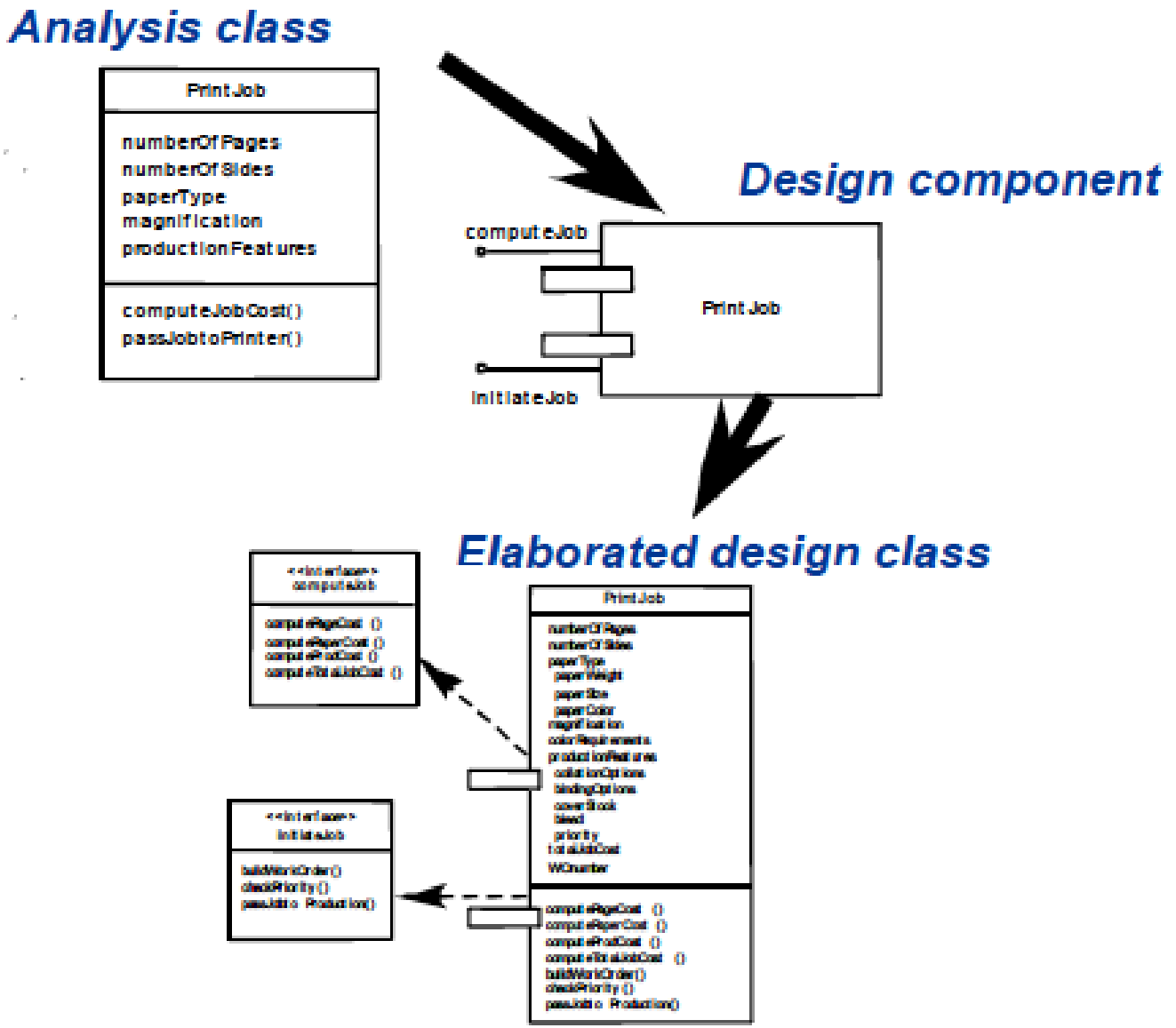
  a component contains **a set of collaborating classes**

- Conventional view

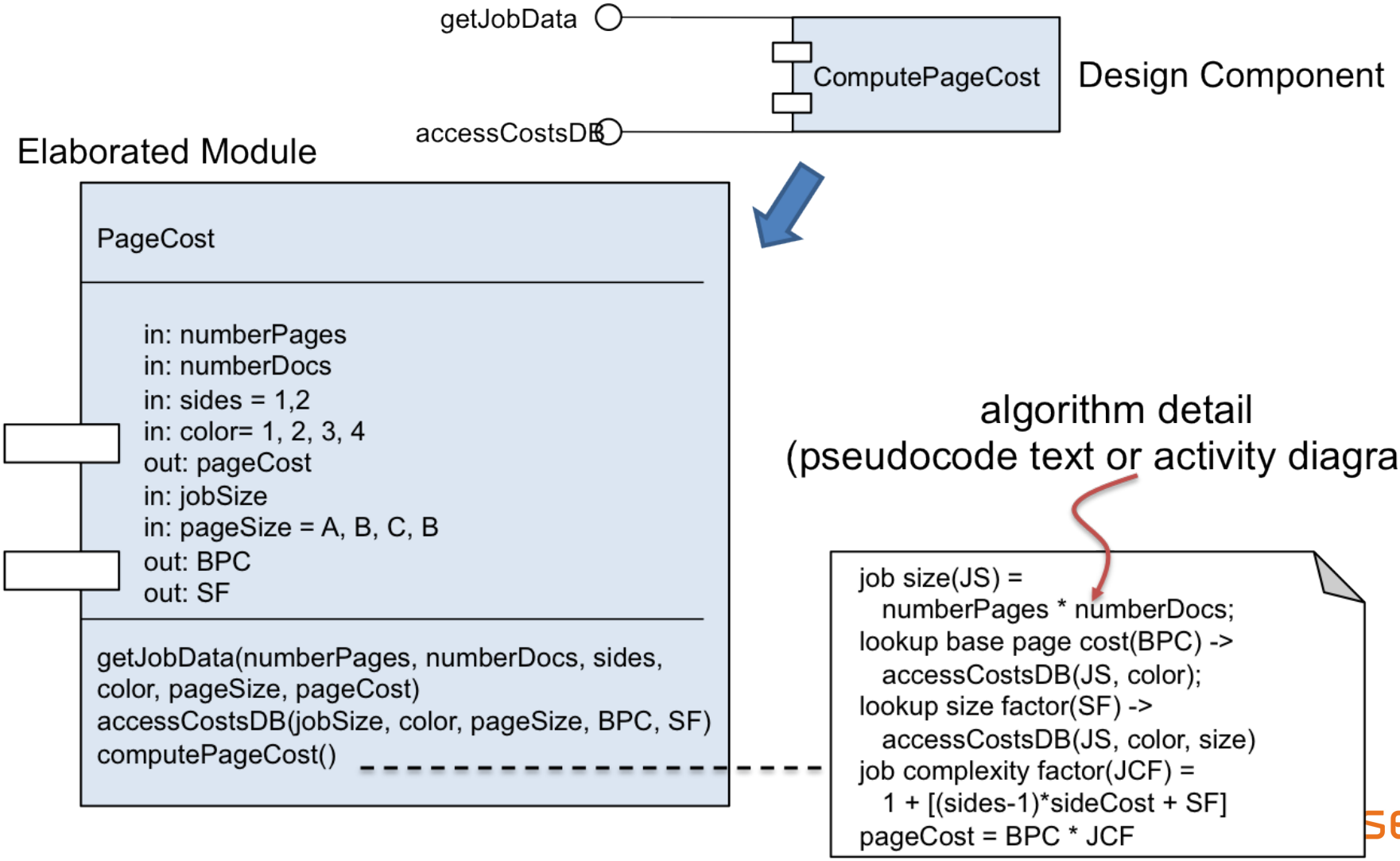
  logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

lab(se);

Object-Oriented Component



Component-Level Design



Basic Design Principles

- **Open-Closed Principle (OCP)**
  - A module should be open for extension but closed for modification
- Liskov Substitution Principle (LSP)
  - Subclasses should be substitutable for their base classes
- Dependency Inversion Principle (DIP)
  - Depend on abstractions. Do not depend on concretions
- Interface Segregation Principle (ISP)
  - Many client-specific interfaces are better than one general purpose interface
- Release Reuse Equivalency Principle (REP)
  - The granule of reuse is the granule of release
- Common Closure Principle (CCP)
  - Classes that change together belong together
- Common Reuse Principle (CRP)
  - Classes that aren't reused together should not be grouped together

lab(se);

Design 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
    - Architectural component names should be drawn from the problem domain and have meaning to all stakeholders
    - infrastructure components or elaborated component-level classes should be named to reflect implementation-specific meaning
    - stereotypes can be used to identify the nature of components at the detailed design level

  e.g., <<infrastructure>> <<database>> <<table>>

Design Guidelines

- Interface
  - provide important information about communication and collaboration
  - to achieve the OCP (Open-Closed Principle)
- Dependencies and Inheritance
  - for better readability, model :

    - dependencies from left to right

    - inheritance from bottom(derived classes) to top(base classes)

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
  - Sequential: passing data from the first op to the following ops
  - Communicational: all operations that access the same data
  - Procedural: similar to Sequential, not without data passing
  - Temporal: ex. Error handling class, initialization class
  - Coincidental: not related operations in a module

lab(se);

Coupling

- 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
- Levels of coupling
  - Content: one component modifies data of another component
  - Common: when components make use of a global variable
  - Control: A invokes B and passes a control flag to B
  - Type use: class A uses a data type defined in class B
  - Inclusion or import: import or include components contents and packages

lab(se);

Component Level Design

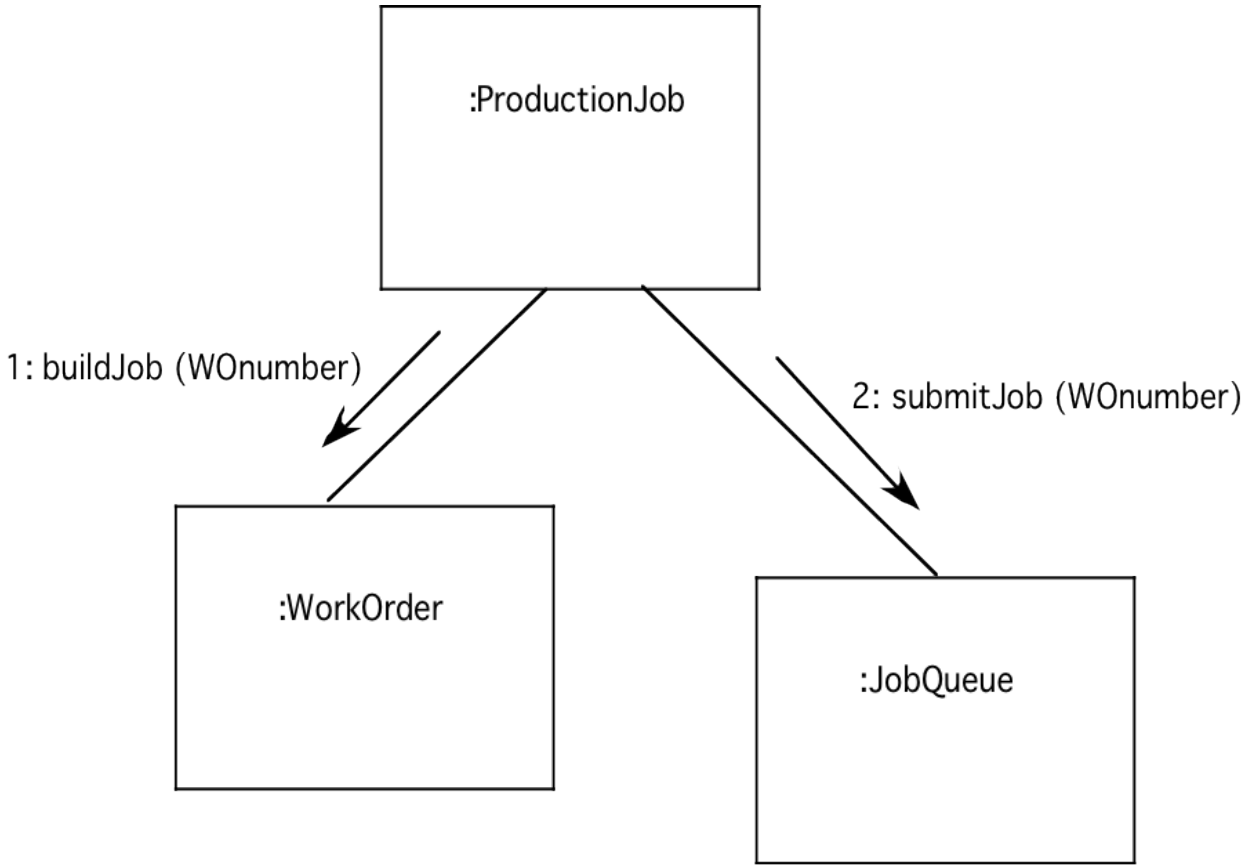
1. Identify all design classes that correspond to the problem domain
2. Identify all design classes that correspond to the infrastructure domain
   e.g. GUI components, OS components, object & data management components, etc
3. Elaborate all design classes that are not acquired as reusable components
   1. Specify **message details** when classes or component collaborate
   2. Identify appropriate **interfaces** for each component
   3. Elaborate **attributes** and define data types and data structures required to implement them
   4. Describe **processing flow (activity diagram)** within each operation in detail

lab(se);

Component Level Design

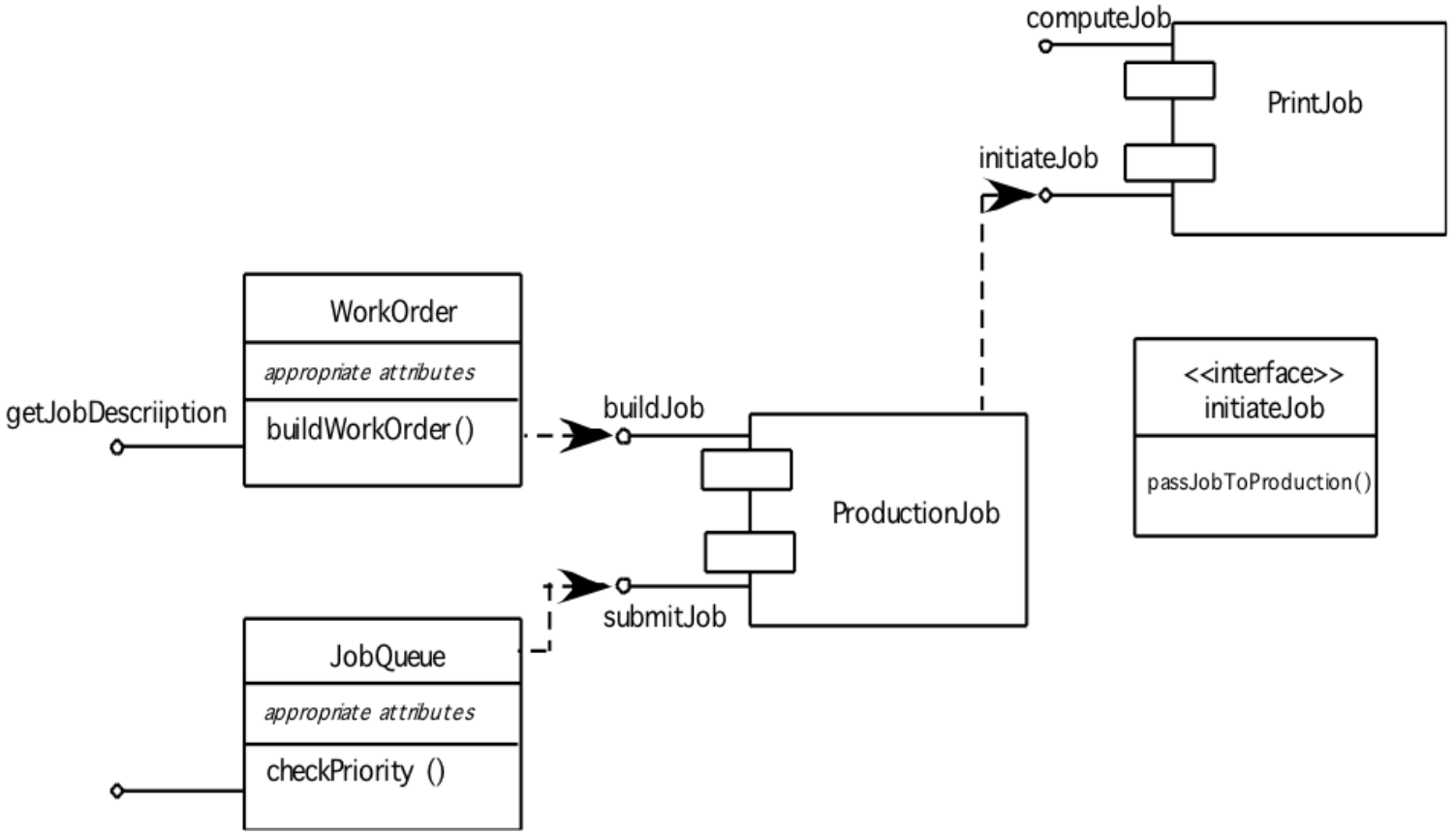
4. Describe persistent data sources (databases and files) and identify the classes required to manage them
5. Develop and elaborate **behavioral representations (state chart)** for a class or component
6. Elaborate **deployment diagrams** to provide additional implementation detail
7. Factor every component-level design representation and always consider **alternatives**

Collaboration Diagram

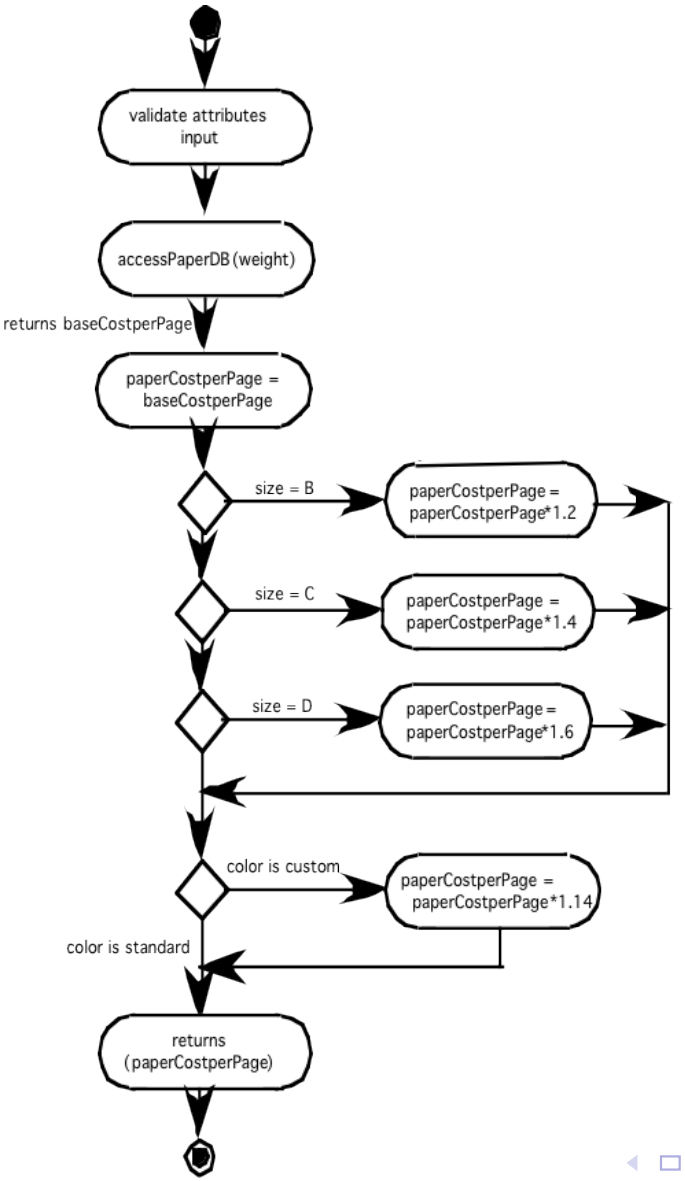


lab(se);

Refactoring

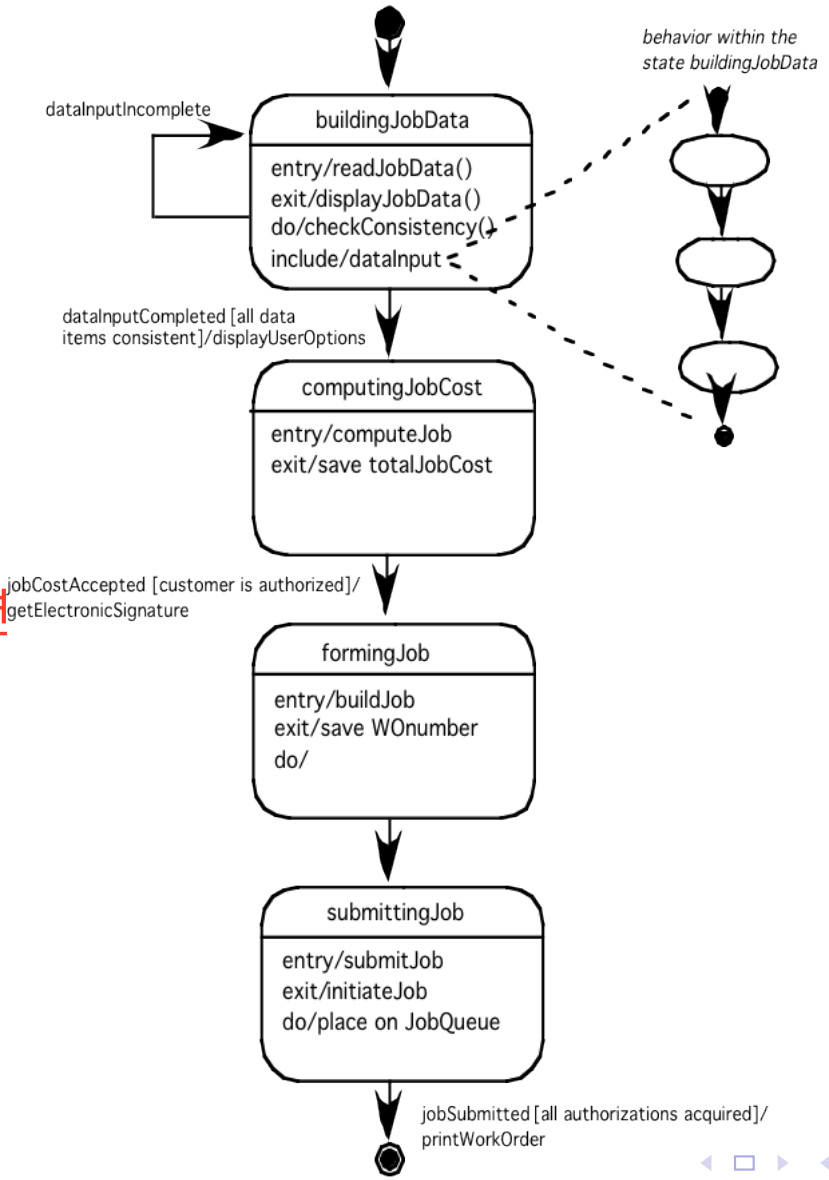


Processing Flow in Activity Diagram



lab(se);

Behavioral Representation in Statechart



() : 파라미터

[] : 가드 컨디션

/ : 액션

Object Constraint Language (OCL)

- complements UML by allowing a software engineer to use a formal grammar and syntax to construct unambiguous statements about various design model elements
- simplest OCL language statements are constructed in four parts:
  1. a **context** that defines the limited situation in which the statement is valid
  2. a **property** that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute)
  3. an **operation** (e.g., arithmetic, set-oriented) that manipulates or qualifies a property
  4. **keywords** (e.g., if, then, else, and, or, not, implies) that are used to specify conditional expressions

lab(se);

OCL Example

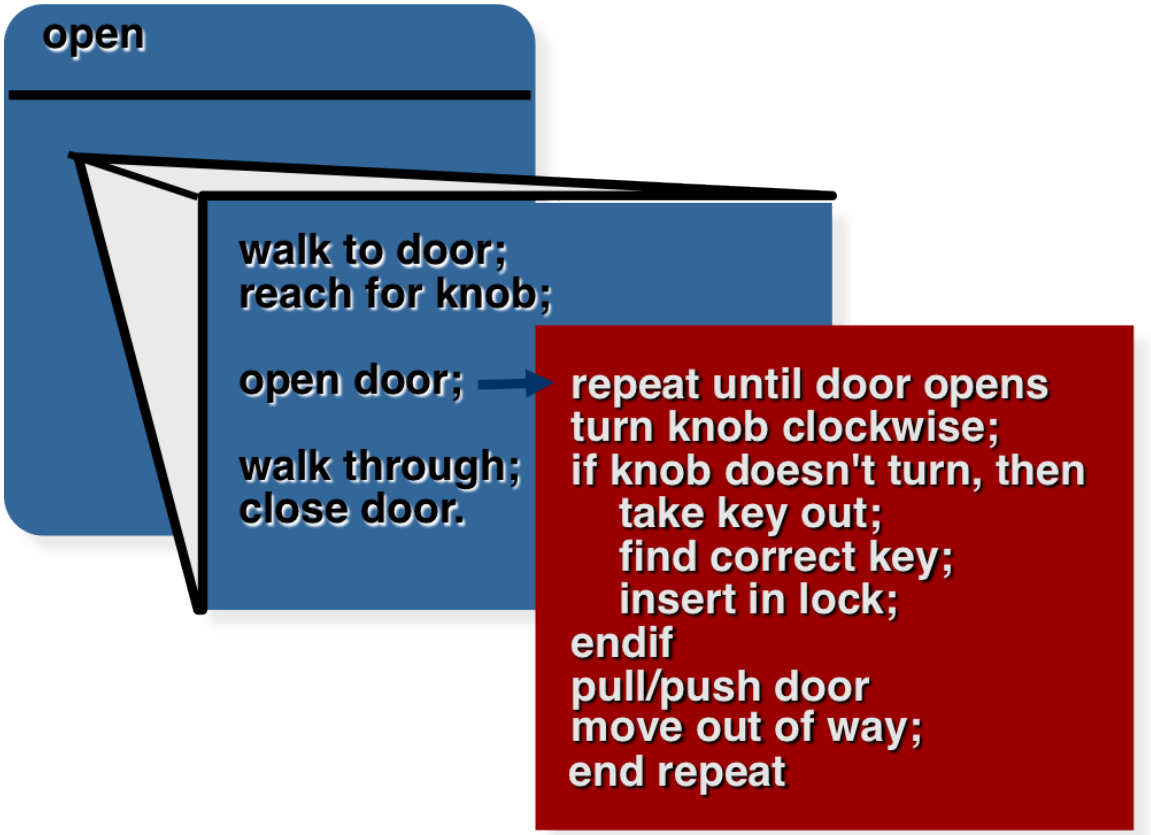
```
context PrintJob::validate(upperCostBound :
                    Integer, custDeliveryReq : Integer)
  pre: upperCostBound > 0
       and custDeliveryReq > 0
       and self.jobAuthorization = 'no'
  post: if self.totalJobCost <= upperCostBound
           and self.deliveryDate <= custDeliveryReq
        then
           self.jobAuthorization = 'yes'
        endif
```

lab(se);

Algorithm Design

- The closest design activity to coding
- The approach:
  - review the design description for the component
  - use stepwise refinement to develop algorithm
  - use **structured programming** to implement procedural logic
  - use '**formal methods**' to prove logic

Stepwise Refinement



Algorithm Design Model

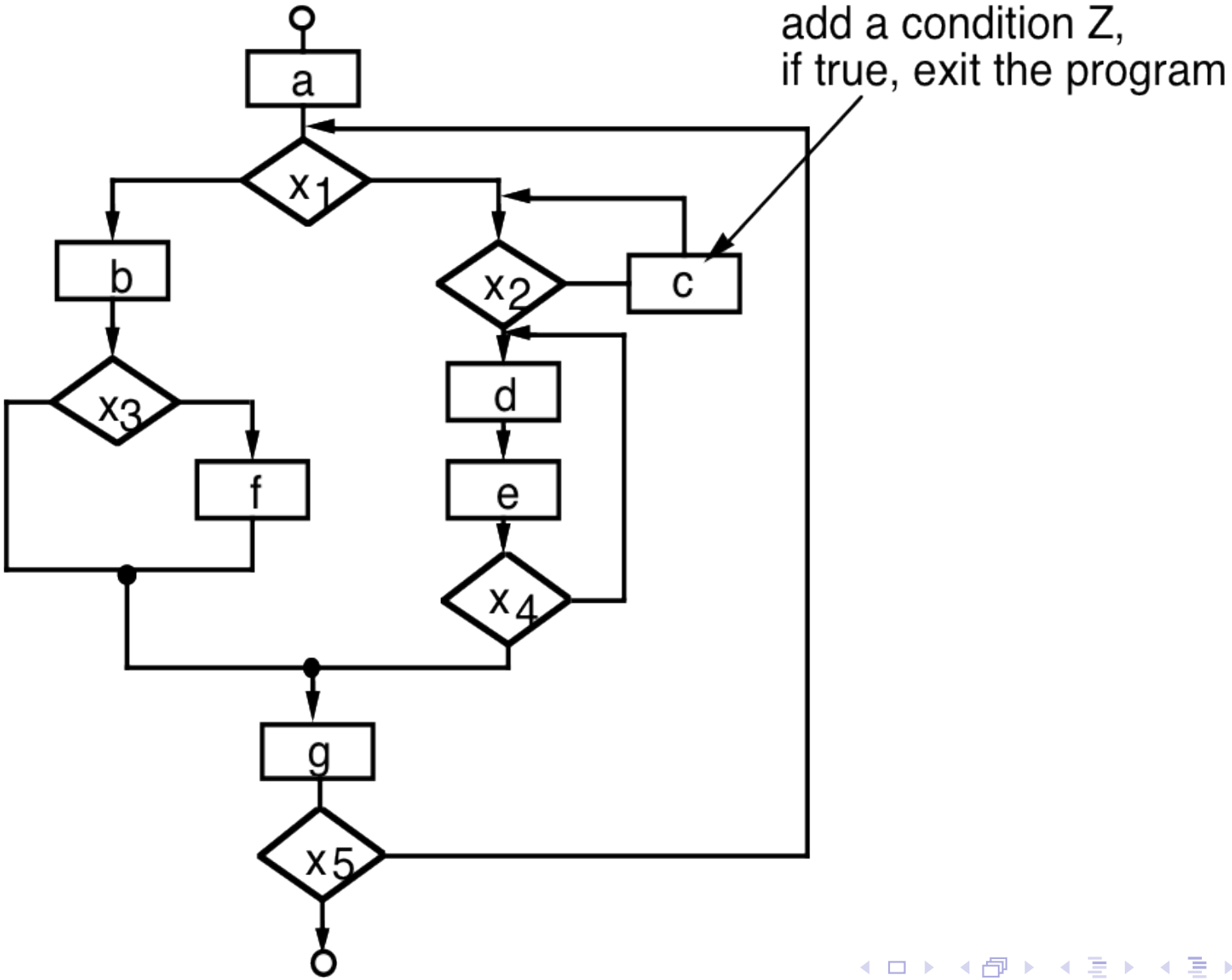
- Represents the algorithm at a level of detail that can be reviewed for quality
- Options:
  - graphical (e.g. flowchart, box diagram)
  - pseudocode (e.g., PDL) ... choice of many
  - programming language
  - decision table

Structured Programming for Procedural Design

- Uses a limited set of logical constructs:
  - sequence
  - conditional : if-then-else, select-case
  - loops : do-while, repeat until
- leads to more readable, testable code
- can be used in conjunction with 'proof of correctness'
- important for achieving high quality, but not enough

lab(se);

A Structured Procedural Design (Flow Chart)



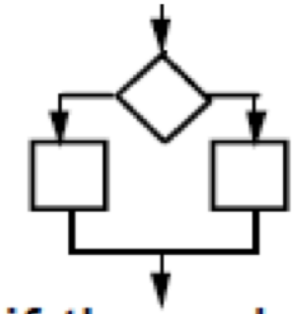
Decision Table

- Use a decision table when a complex set of conditions and actions are encountered within a component

	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
Rules						
no discount	✓					
apply 8 percent discount			✓	✓		
apply 15 percent discount					✓	✓
apply additional x percent discount		✓		✓		✓

lab(se);

Program Design Language (PDL)



if-then-else

```
if condition x
  then process a;
  else process b;
endif
```

PDL

- Easy to combine with source code
- Can be represented in great detail
- Machine readable, no need for graphics input
- Graphics can be generated from PDL
- Enables declaration of data as well as procedure
- Easier to review and maintain

lab(se);

Component-Based Development

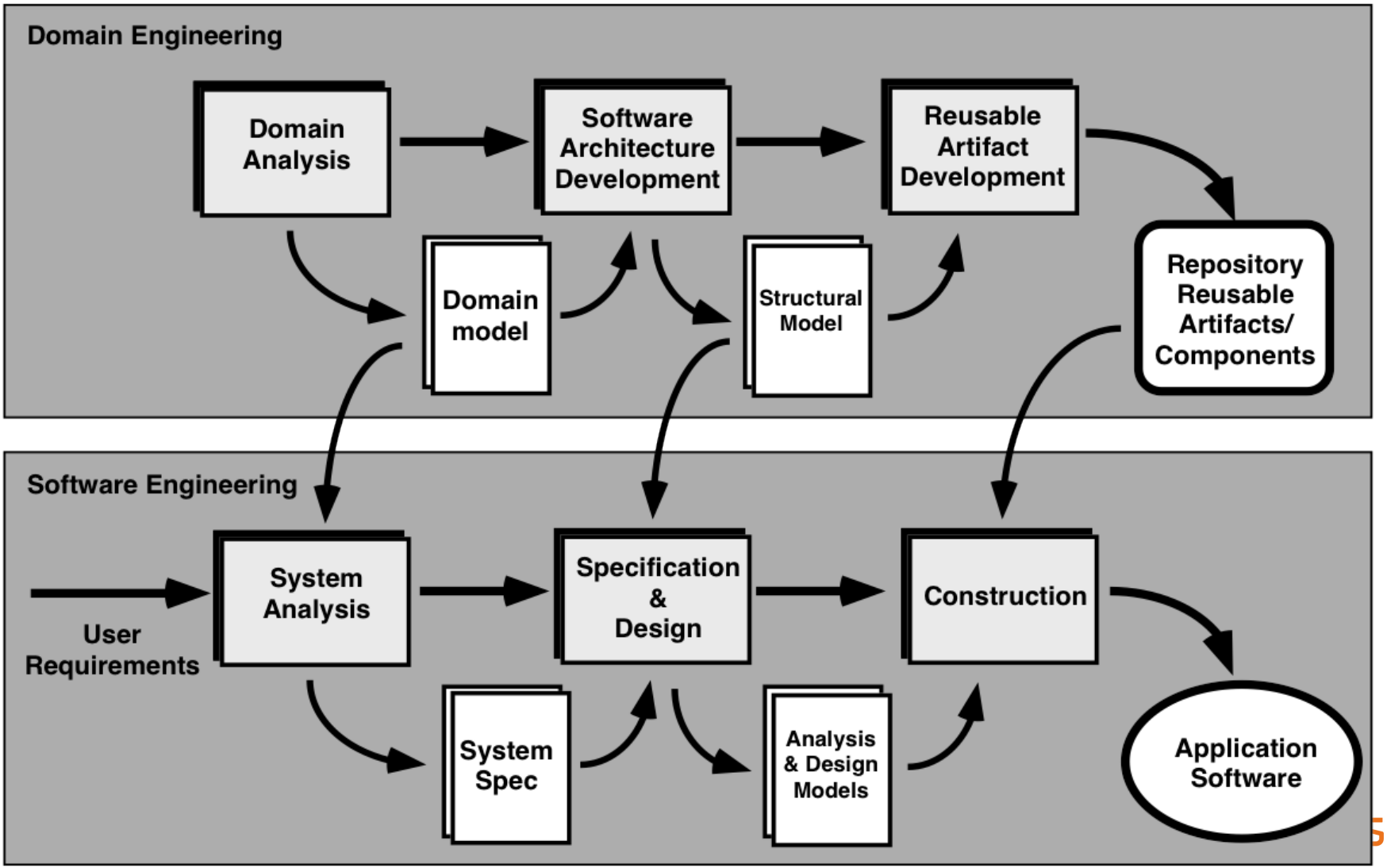
- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally-developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible within the architecture of the system to be built?
- At the same time, they are faced with the following impediments to reuse ...

Impediments to Reuse

- Few companies and organizations 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 them
- Relatively little training is available to help software engineers and managers understand and apply reuse
- Many software practitioners continue to believe that reuse is "more trouble than it's worth"
- Many companies continue to encourage of software development methodologies which do not facilitate reuse
- Few companies provide an incentives to produce reusable program components

lab(se);

Component-Based Software Engineering Process



Domain Engineering

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

lab(se);

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 a non-reusable component be parameterized to become 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?

lab(se);

Component-Based Software Engineering

- a library of components must be available
- components should have a consistent structure
- a standard should exist, such as :
  - OMG/CORBA
  - Microsoft COM
  - Sun JavaBeans

lab(se);

Component-Based Software Engineering Activities

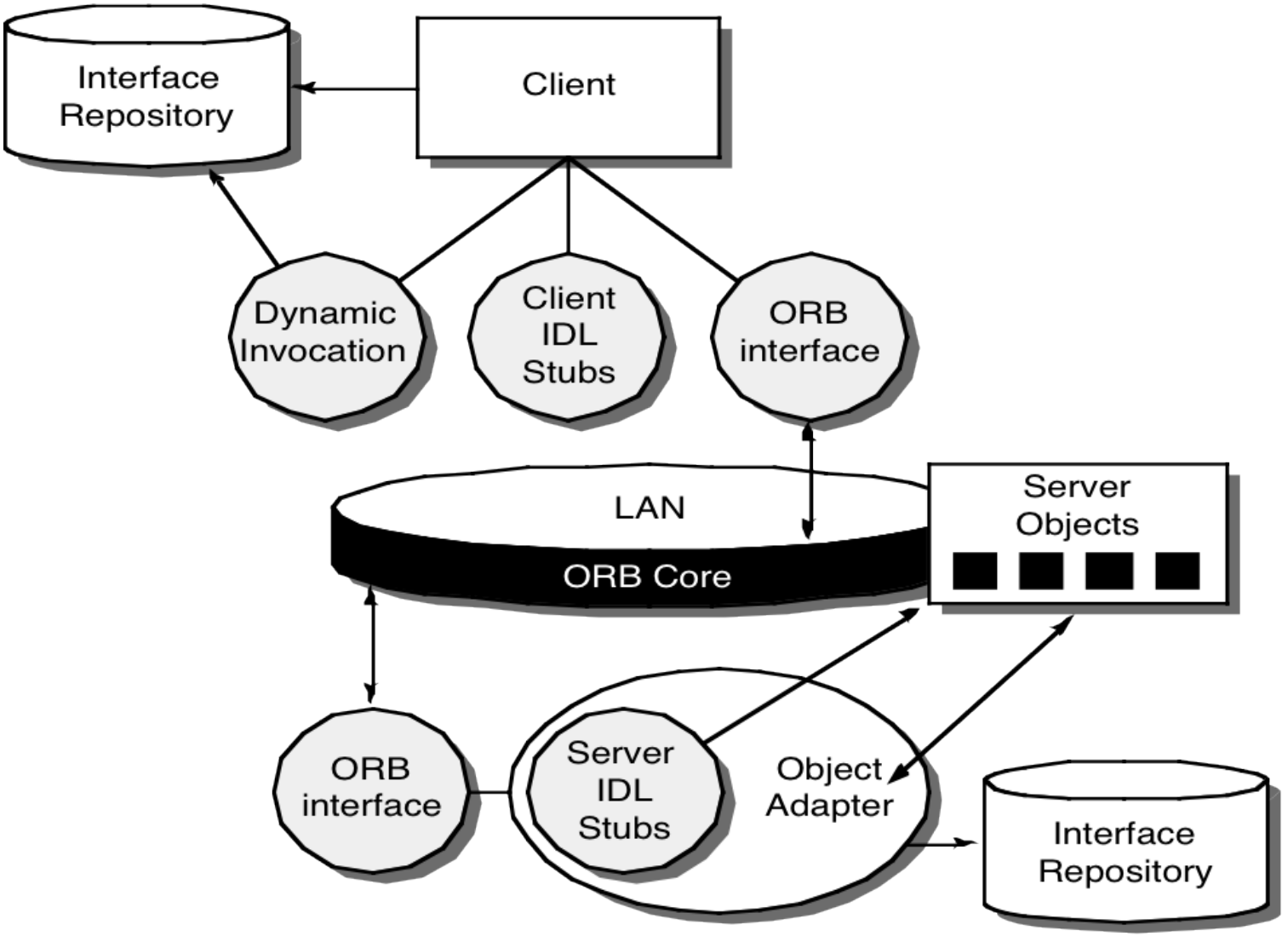
- Component qualification
- Component adaptation
- Component composition
- Component update

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

lab(se);

ORB Architecture



lab(se);

Microsoft COM

- Component Object Model(COM) provides a specification for using components produced by various vendors within a single application running under the Windows operating system
- COM encompasses two elements:
  - COM interfaces (implemented as COM objects)
  - a set of mechanisms for registering and passing messages between COM interfaces

Sun JavaBeans

- JavaBeans component system is a portable, platform independent CBSE infrastructure developed using the Java programming language
- The JavaBeans component system 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.

lab(se);