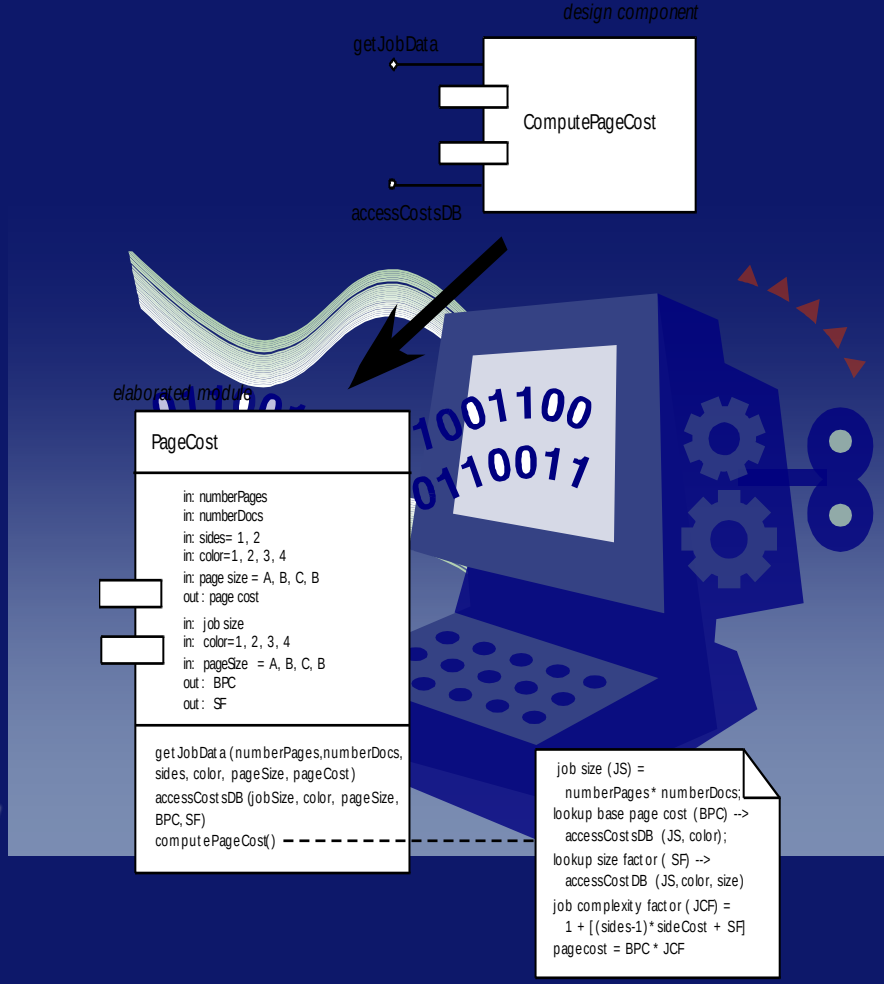



# Software Engineering

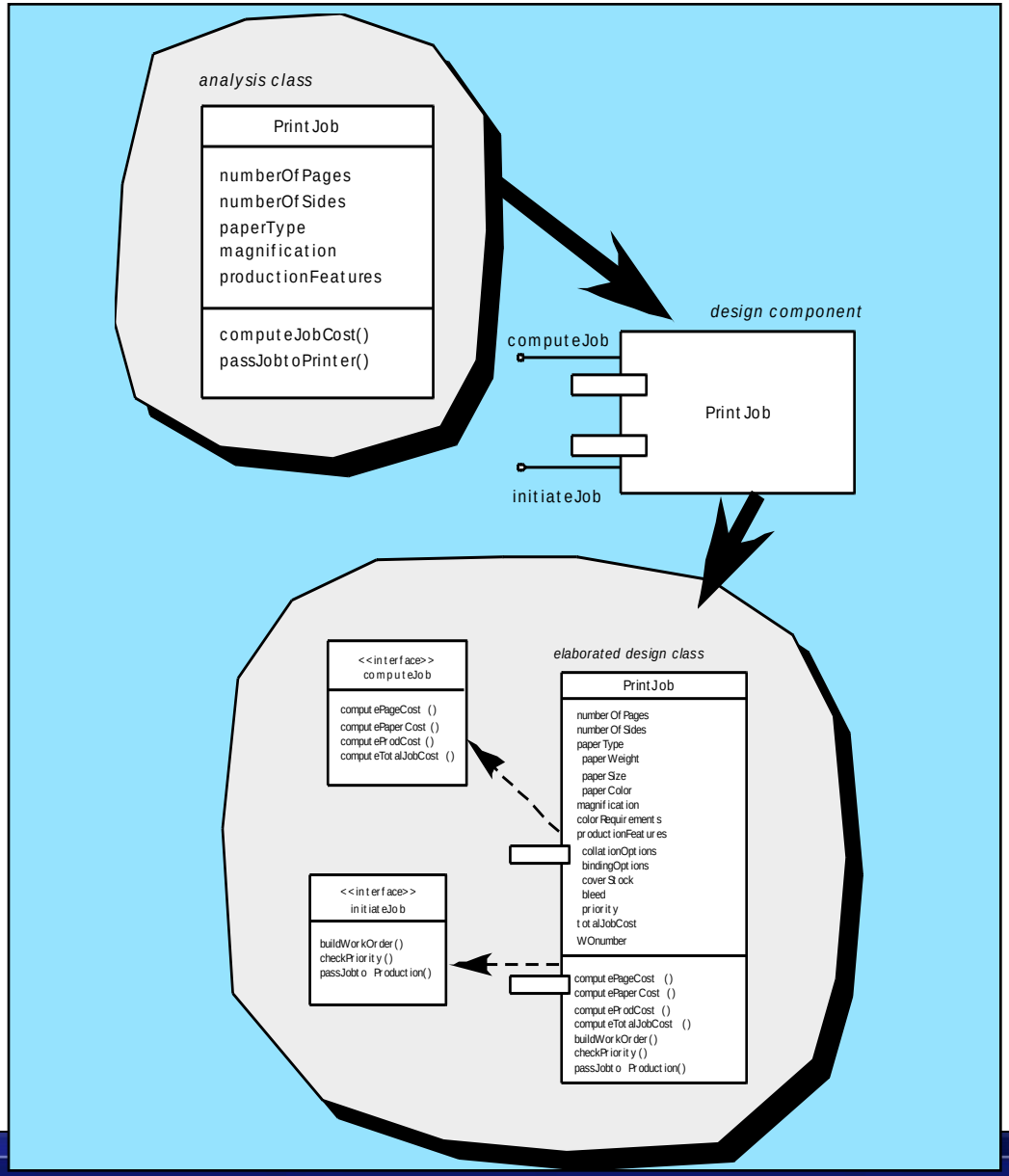
# Chapter 11 Component-Level Design

Moon kun Lee
Division of Electronics & Information Engineering
Chonbuk National University

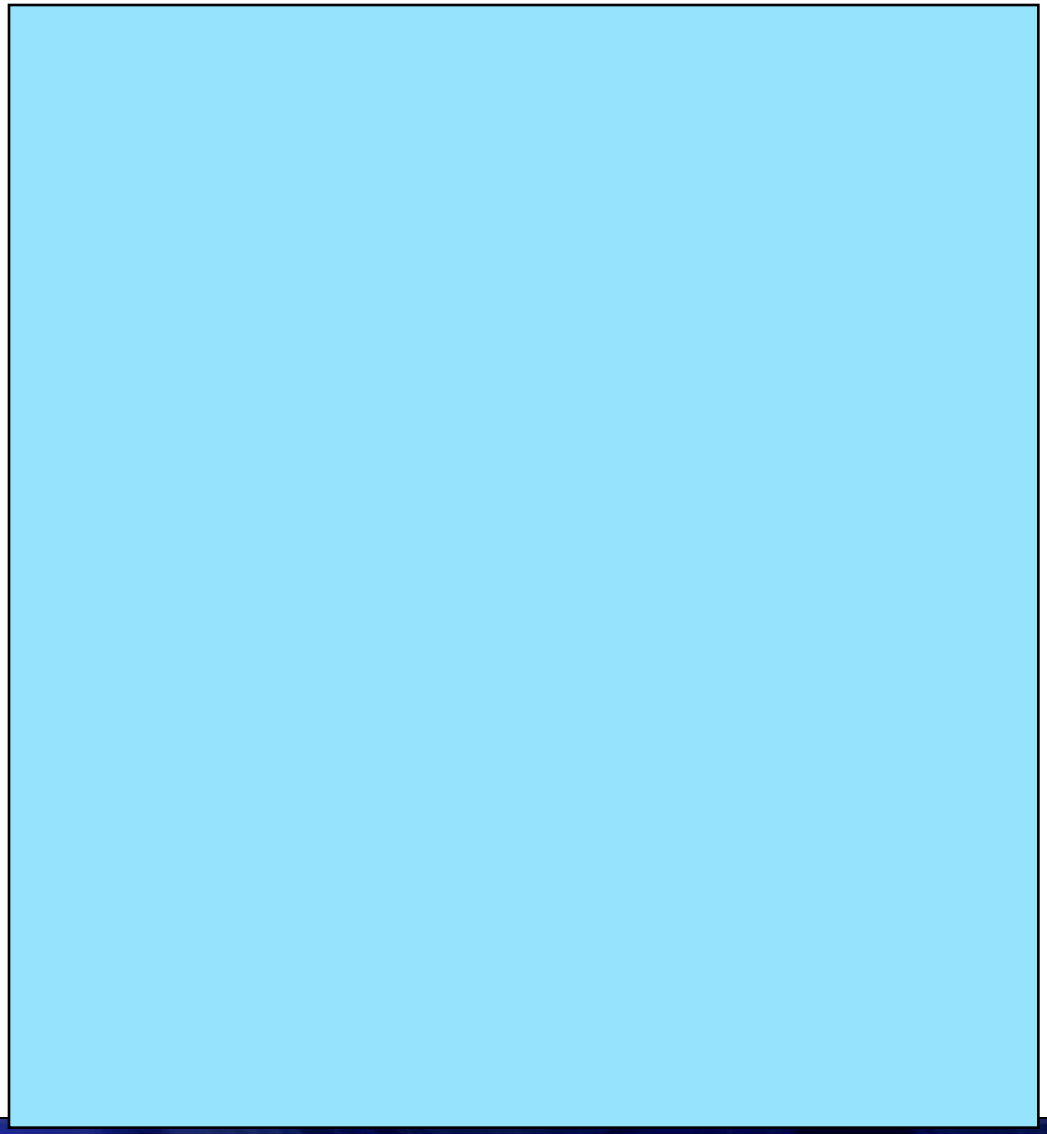
# What is a Component?

- *OMG Unified Modeling Language Specification* [OMG01] 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.

# OO Component



# Conventional Component



# Basic Design 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.*
- 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."*

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

# 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
- Interfaces
  - Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OCP(Open-Closed Principle))
- 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).

# Cohesion

- Conventional view:
  - the “single-mindedness” of a module
- OO view:
  - *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
- Levels of cohesion
  - Functional
  - Layer
  - Communicational
  - Sequential
  - Procedural
  - Temporal
  - utility

# 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
- Level of coupling
  - Content:  violate information hiding.
  - Common: uncontrolled error propagation, unforseen side effect
  - Control: consistency problem
  - Stamp: modification of system is more complex.
  - Data: increase bandwidth of comm. and complexity of the interface.
  - Routine call: increase the connectedness of a system.
  - Type use: modification is hard.
  - Inclusion or import:
  - External:

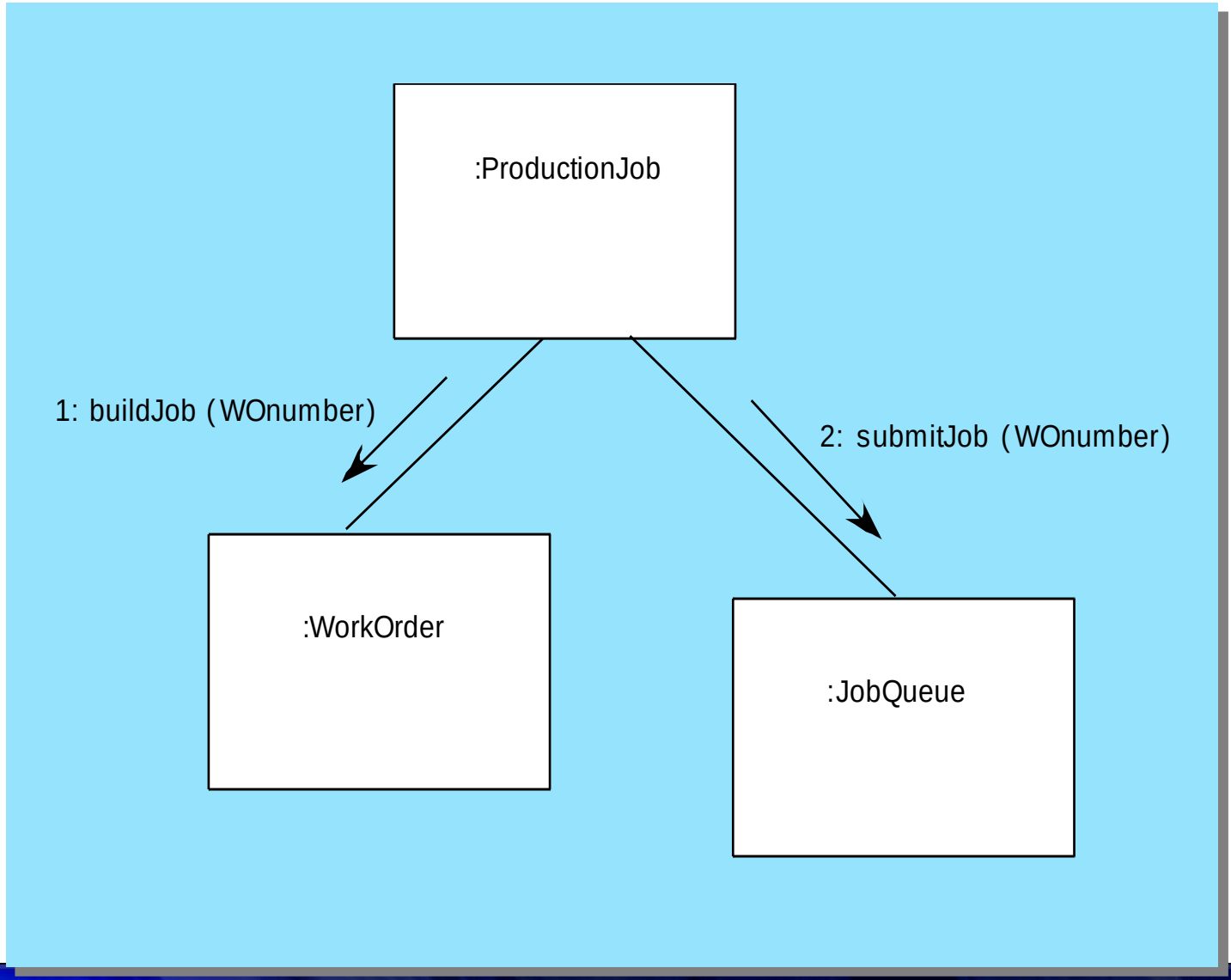
# Component Level Design-I

- 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 component collaborate.
- Step 3b. Identify appropriate interfaces for each component.

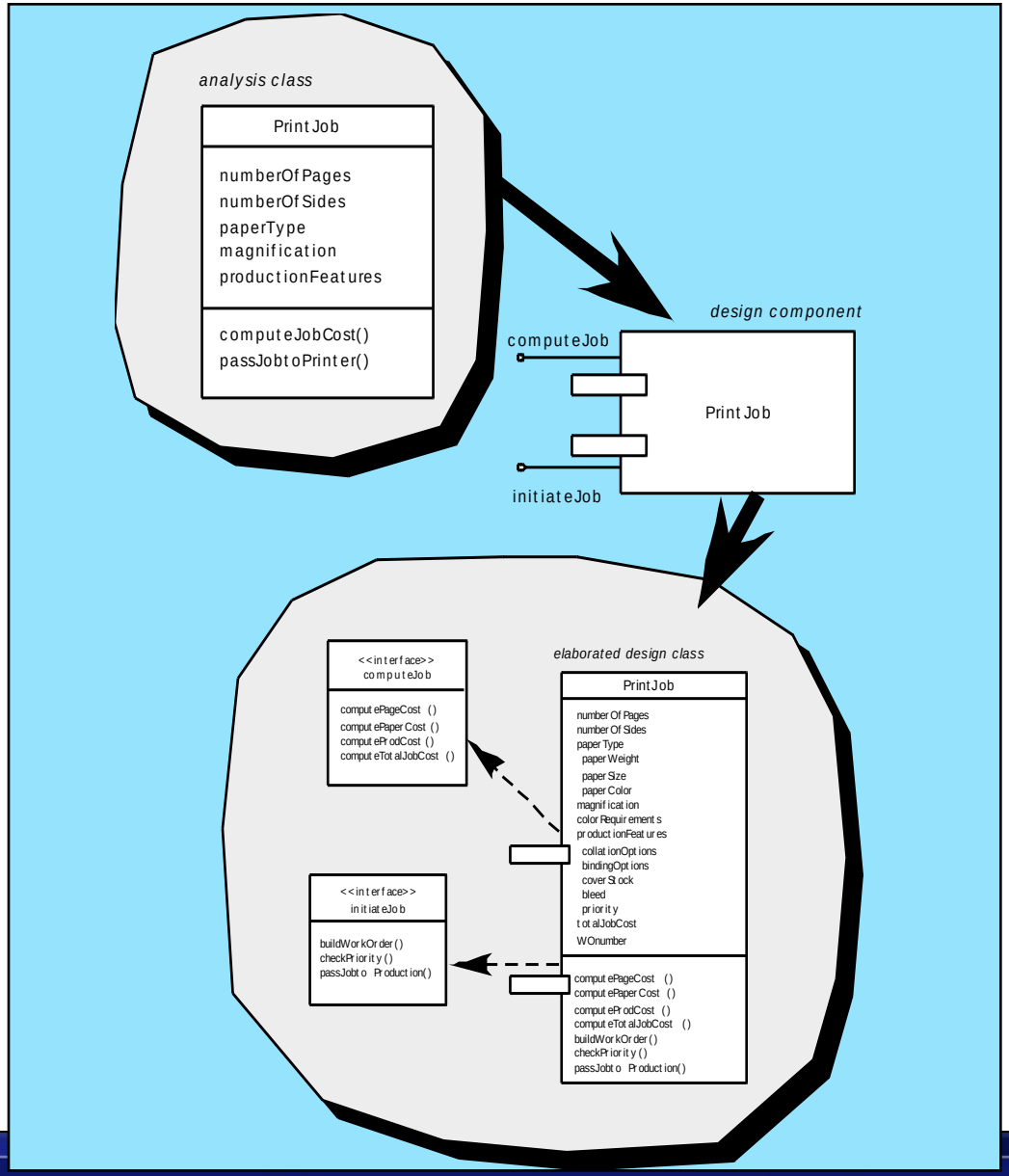
# Component-Level Design-II

- 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.
- 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. Factor every component-level design representation and always consider alternatives.

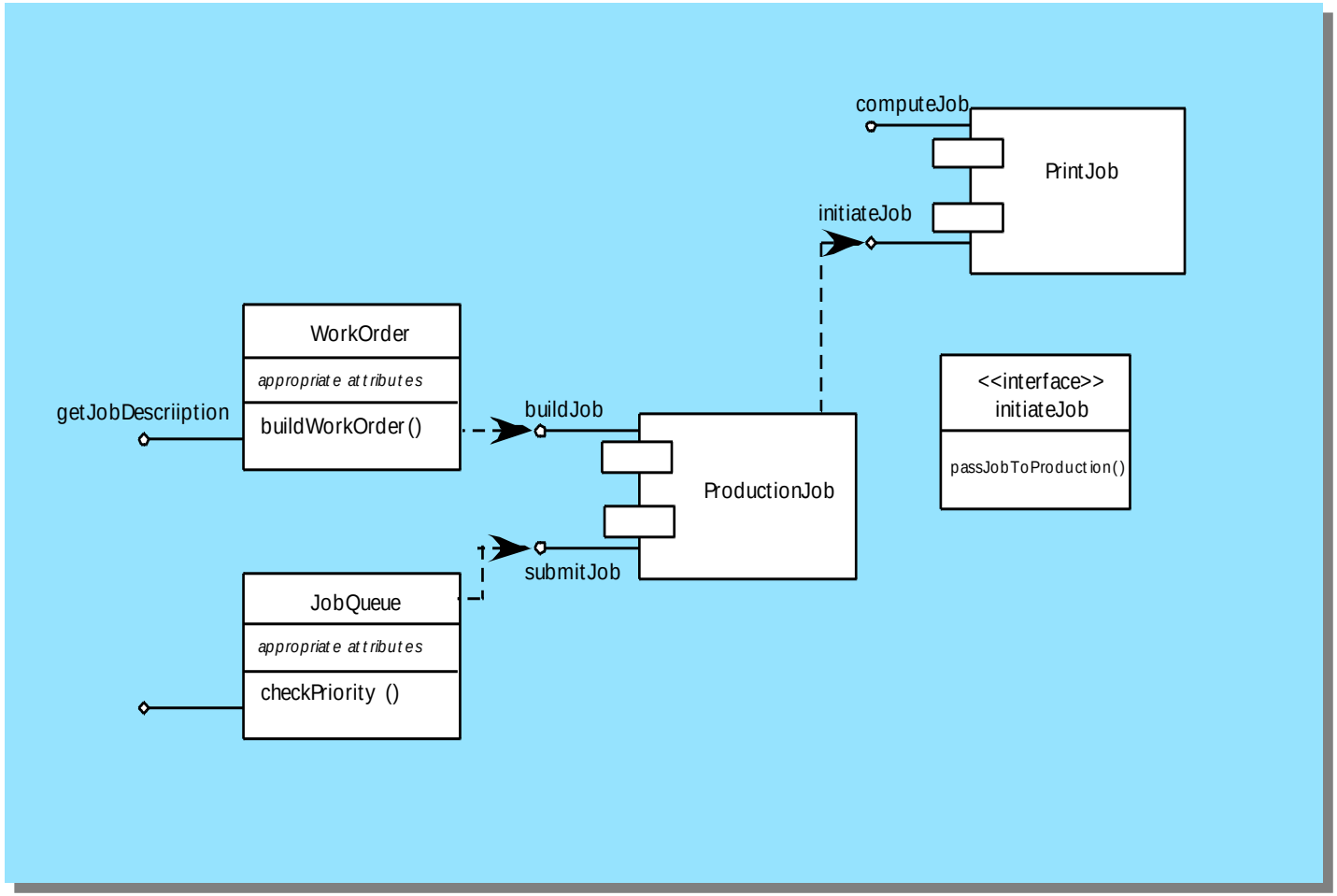
# Collaboration Diagram (3.a)



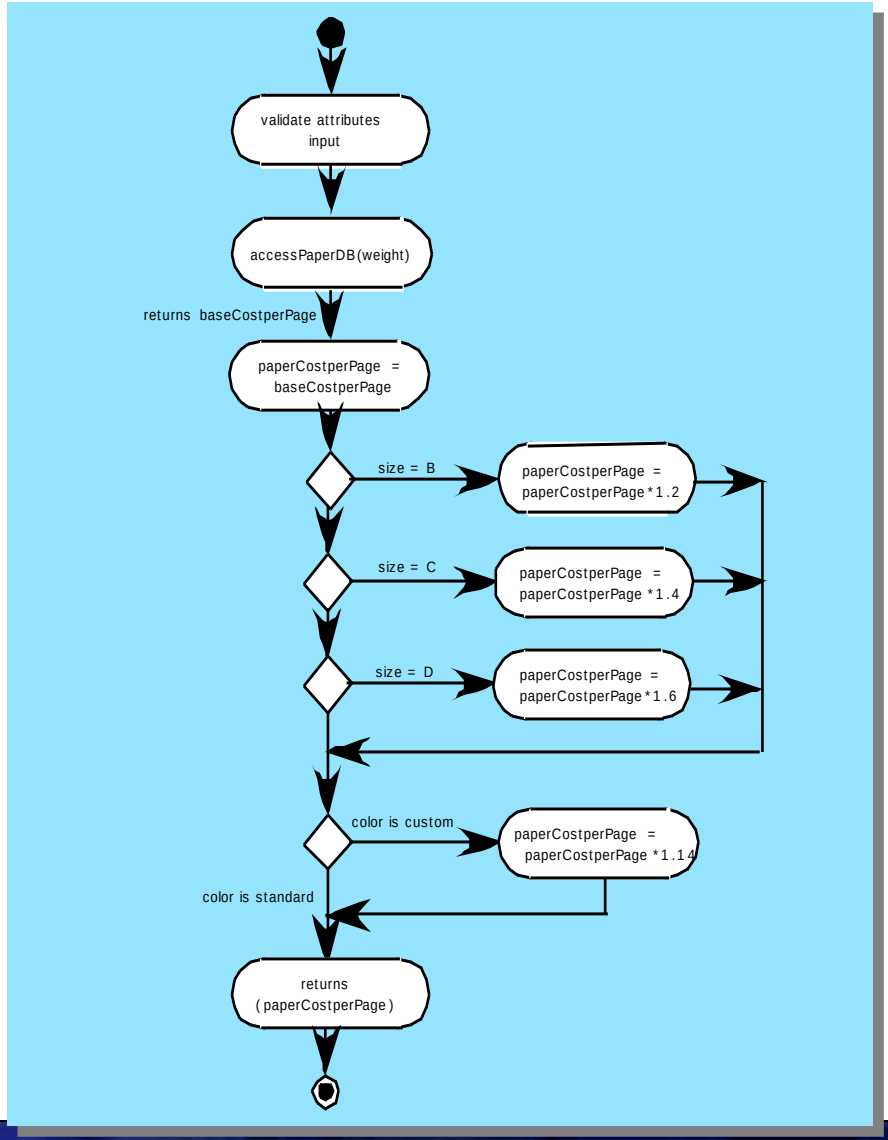
# OO Component (3.b)



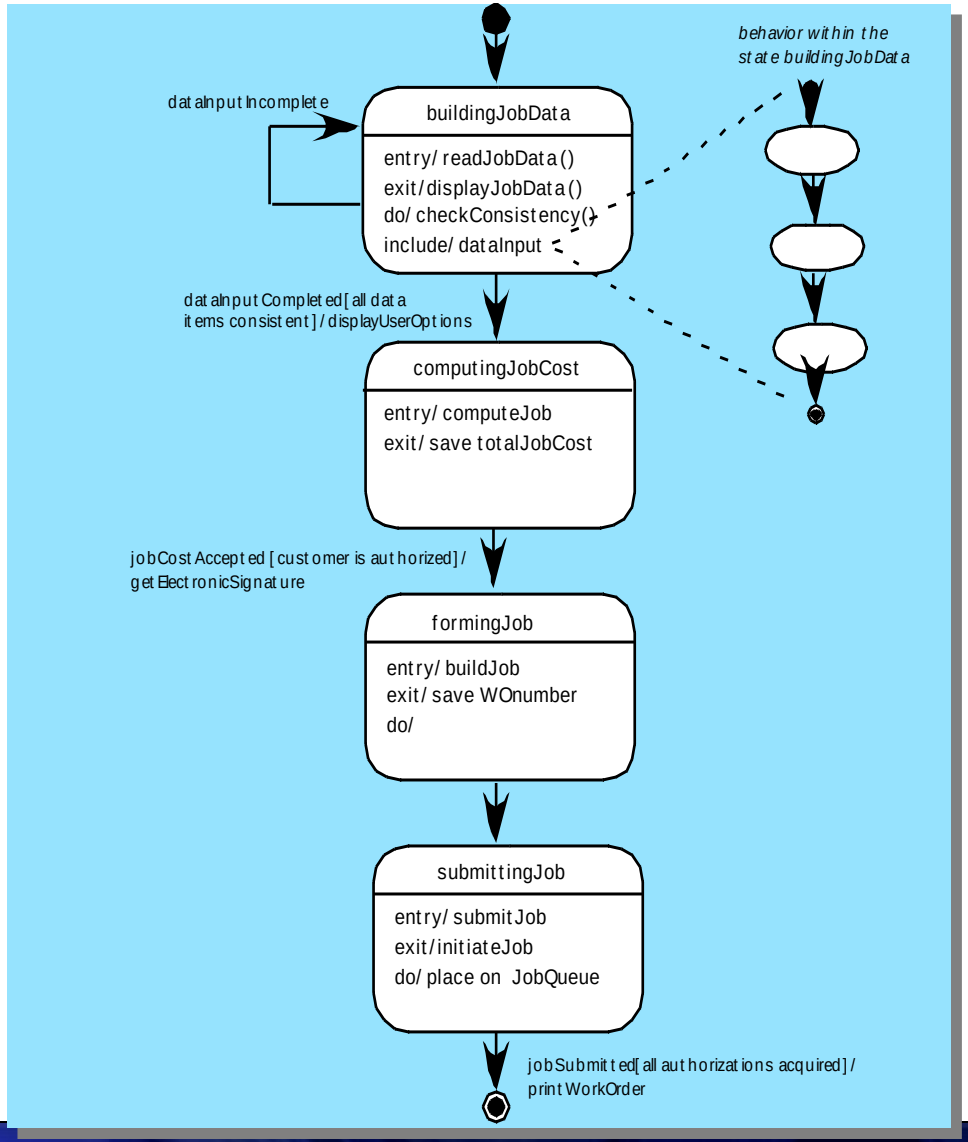
# Refactoring (3.b)



# Activity Diagram (3.d)



# Statechart (5)



# 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, and
  (4) keywords (e.g., if, then, else, and, or, not, implies) that are used to specify conditional expressions.

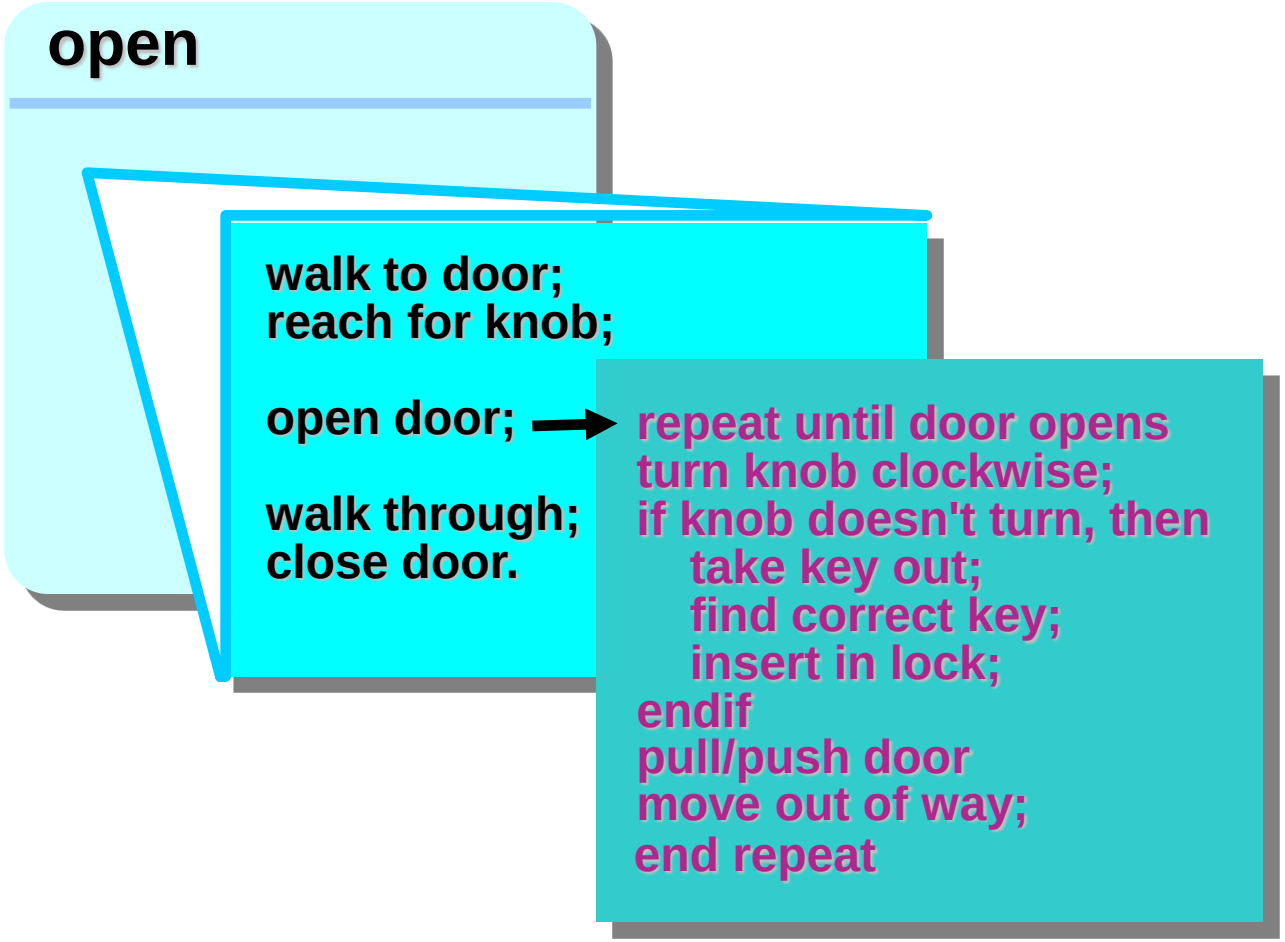
# OCL Example

```
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
```

# 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



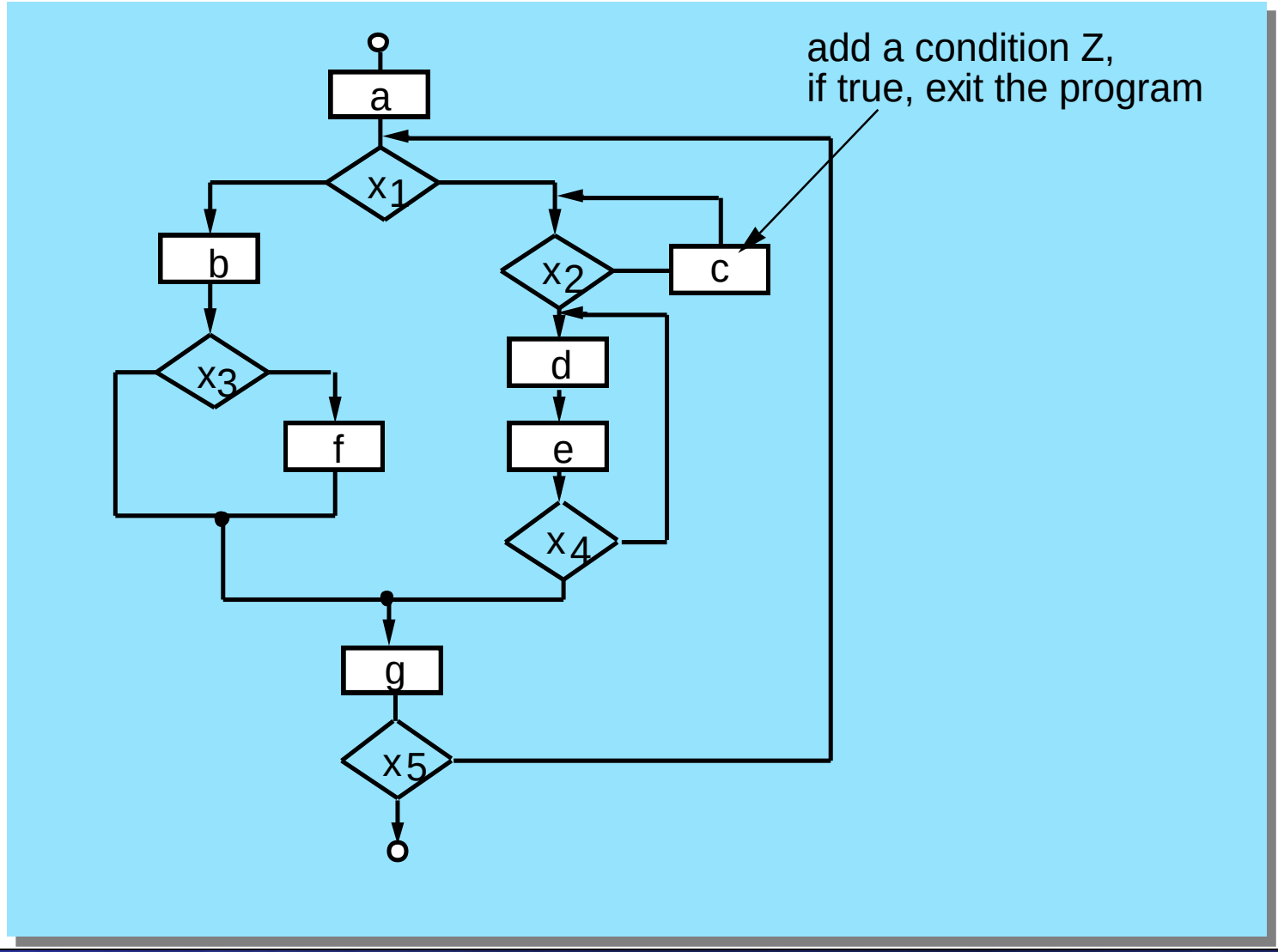
# Algorithm Design Model

- 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
  - conduct walkthrough to assess quality

# Structured Programming

- 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

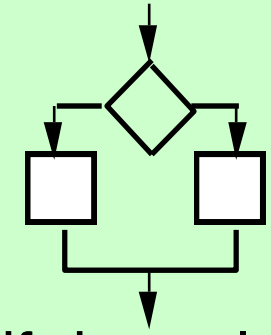
# A Structured Procedural Design



# Decision Table

| | 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 | | ✓ | | ✓ | | ✓ |

# Program Design Language (PDL)



if-then-else

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

PDL

- easy to combine with source code
- machine readable, no need for graphics input
- graphics can be generated from PDL
- enables declaration of data as well as procedure
- easier to maintain

# Why Design Language?

- can be a derivative of the HOL of choice e.g., Ada PDL
- machine readable and processable
- can be embedded with source code, therefore easier to maintain
- can be represented in great detail, if designer and coder are different
- easy to review

# Representation Tools

- PDL/81
- DocGen
- PowerPDL