Chapter 11

- **Component-Level Design**

Slide Set to accompany

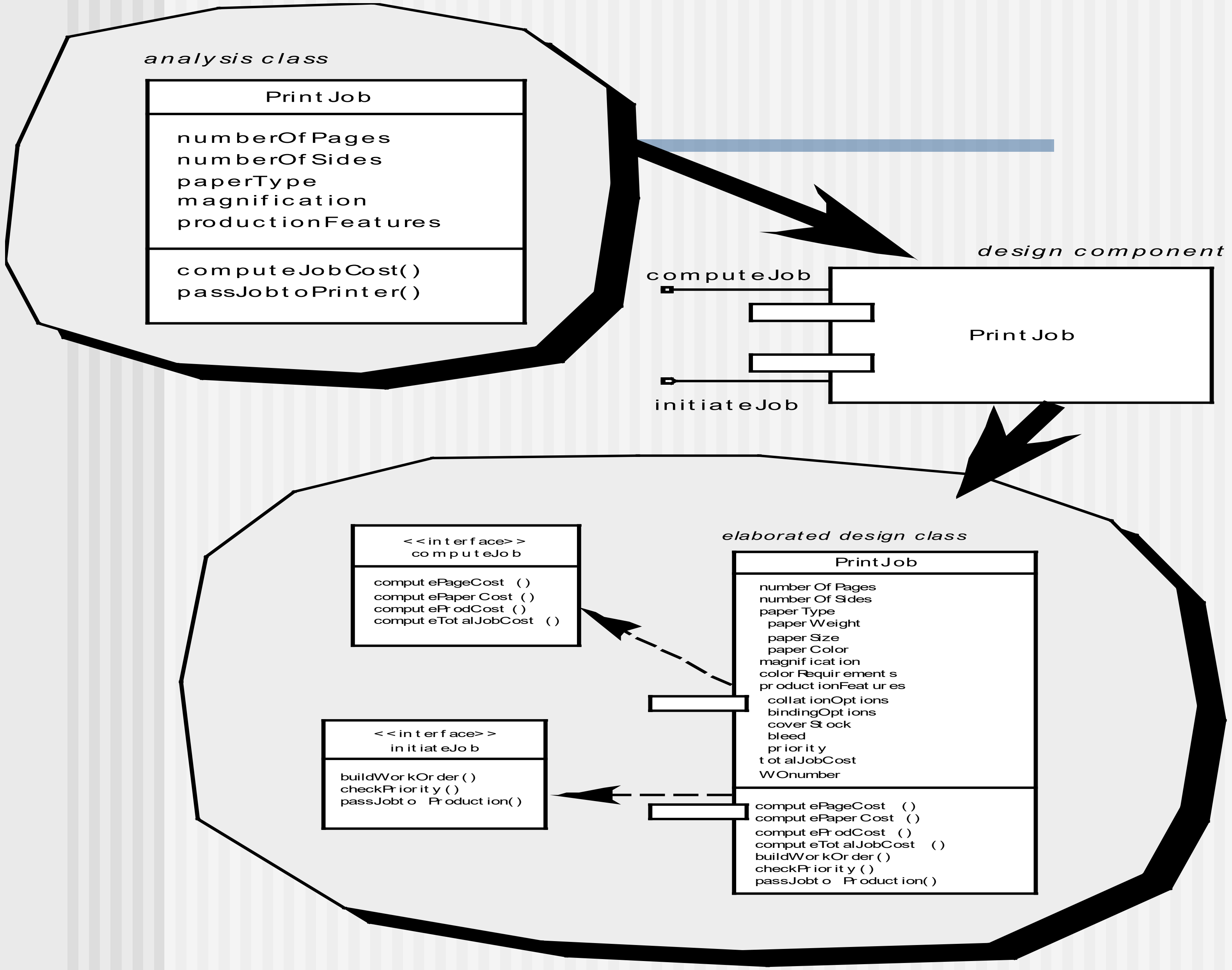
Software Engineering: A Practitioner's Approach, 8/e

by Roger S. Pressman and Bruce R. Maxim

What is a Component?

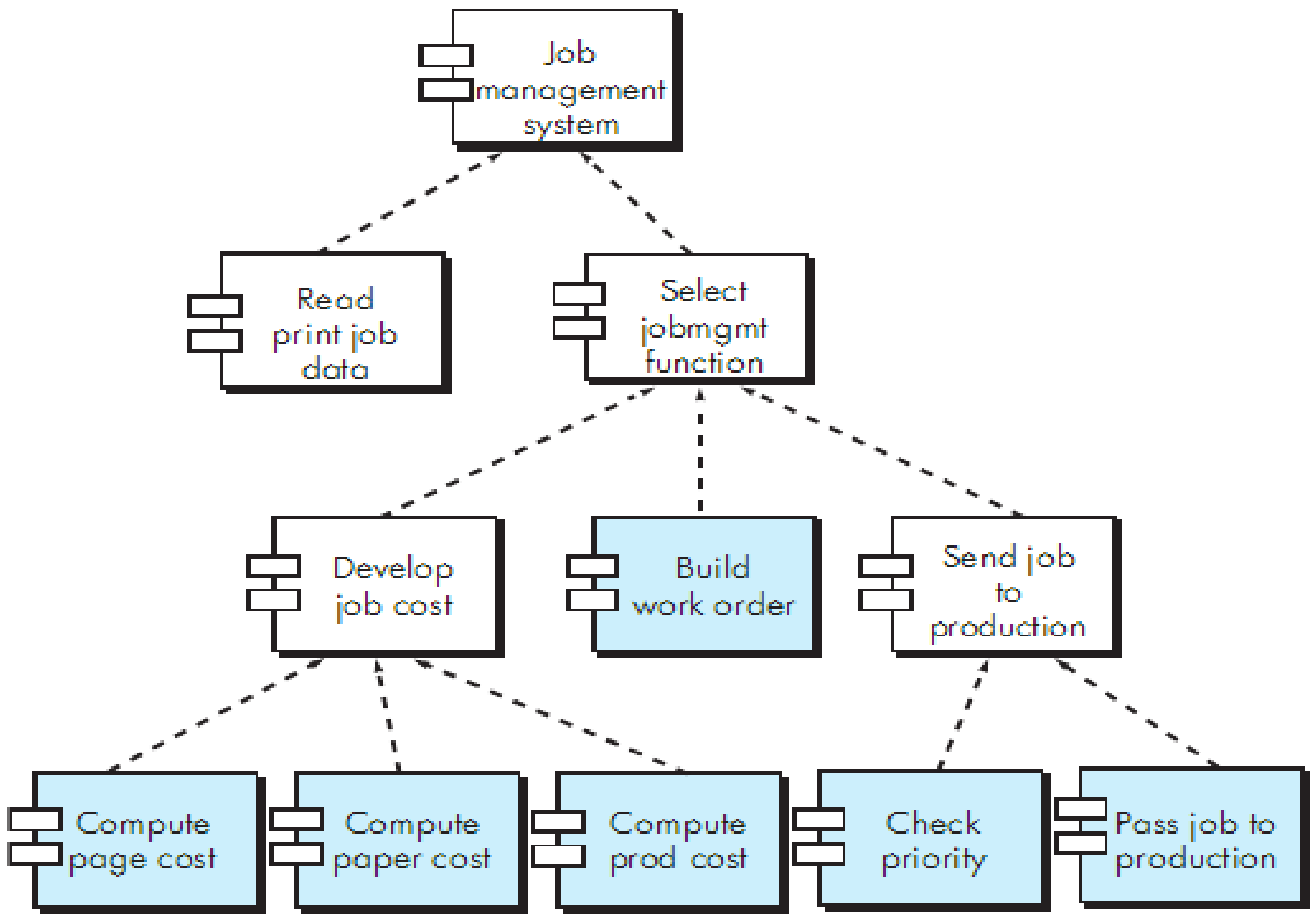
- *OMG Unified Modeling Language Specification* [OMG01] defines a component as
  - "… a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.""
- *OO view:* a component contains a set of collaborating classes.
- *Conventional view:* a component contains processing 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. Also called a module.
- *Process-related view:* a catalog of proven design or code-level components is made available to engineers as design work proceeds.

OO Component

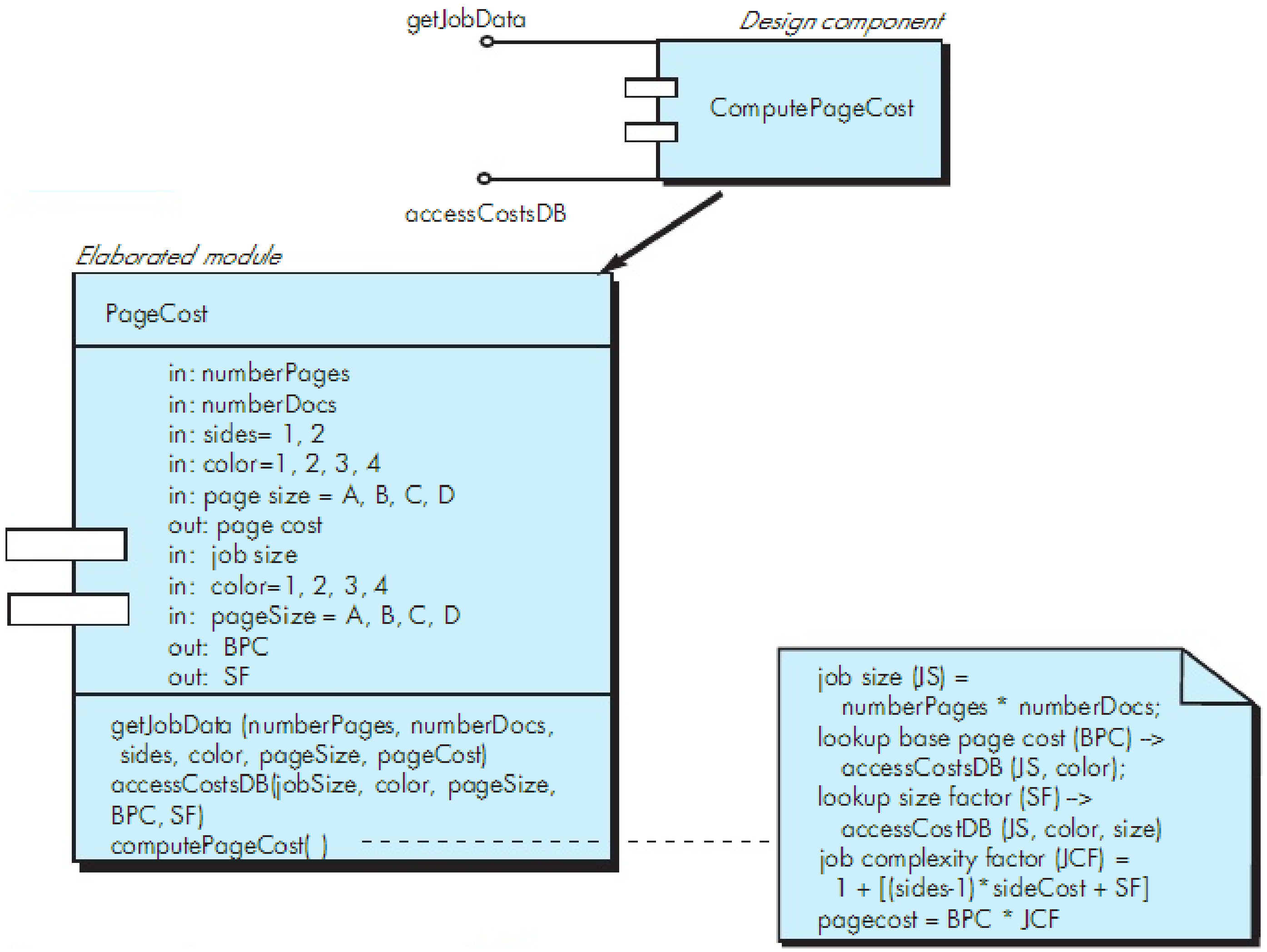


Conventional Component

Structure chart for a traditional system



Conventional Component


getJobData
Design component
ComputePageCost
accessCostsDB
Elaborated module
PageCost
in: numberPages
in: numberDocs
in: sides= 1, 2
in: color=1, 2, 3, 4
in: page size = A, B, C, D
out: page cost
in: job size
in: color=1, 2, 3, 4
in: pageSize = A, B, C, D
out: BPC
out: SF
getJobData (numberPages, numberDocs, sides, color, pageSize, pageCost)
accessCostsDB(jobSize, color, pageSize, BPC, SF)
computePageCost( )
job size (JS) = numberPages * numberDocs;
lookup base page cost (BPC) --> accessCostsDB (JS, color);
lookup size factor (SF) --> accessCostDB (JS, color, size)
job complexity factor (JCF) = 1 + [(sides-1) * sideCost + SF]
pagecost = BPC * JCF

Design Class-Based Components

- Component-level design draws on information developed as part of the requirements model and represented as part of the architectural model.
- Component-level design focuses on the elaboration of problem domain specific classes and the definition & refinement of infrastructure classes contained in the requirements model.
- The detailed description of the attributes, operations and interfaces used by these classes in the design detail required as a precursor to the construction activity.

Basic Design Principles

- The Open-Closed Principle (OCP).

  "A module [component] should be open for extension but closed for modification.

  Example：in the textbook

- The Liskov Substitution Principle (LSP).

  "Subclasses should be substitutable for their base classes.

  Example：http://blog.csdn.net/zhengzhb/article/details/7281833

- Dependency Inversion Principle (DIP).

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

  Example：http://www.cnblogs.com/gaochundong/p/dependency_inversion_principle.html

- The Interface Segregation Principle (ISP).

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

  Example：in the textbook

Basic Design Principles

- The Release Reuse Equivalency Principle (REP).

  "The granule of reuse is the granule of release" [Mar2000]

- The Common Closure Principle (CCP).

  "Classes that change together belong together" [Mar2000]

  Classes should be packaged cohesively. This leads to more effective change control and release management.

- The Common Reuse Principle (CRP).

  "Classes that aren't reused together should not be grouped together" [Mar2000]

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 OPC)
- 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
    - Common
    - Control
    - Stamp
    - Data
    - Routine call
    - Type use
    - 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. Fig.11.6 Collaboration Diagram
- Step 3b. Identify appropriate interfaces for each component. Fig. 11.7 Refactoring interfaces & definition of classes

Fig.11.6 Collaboration Diagram

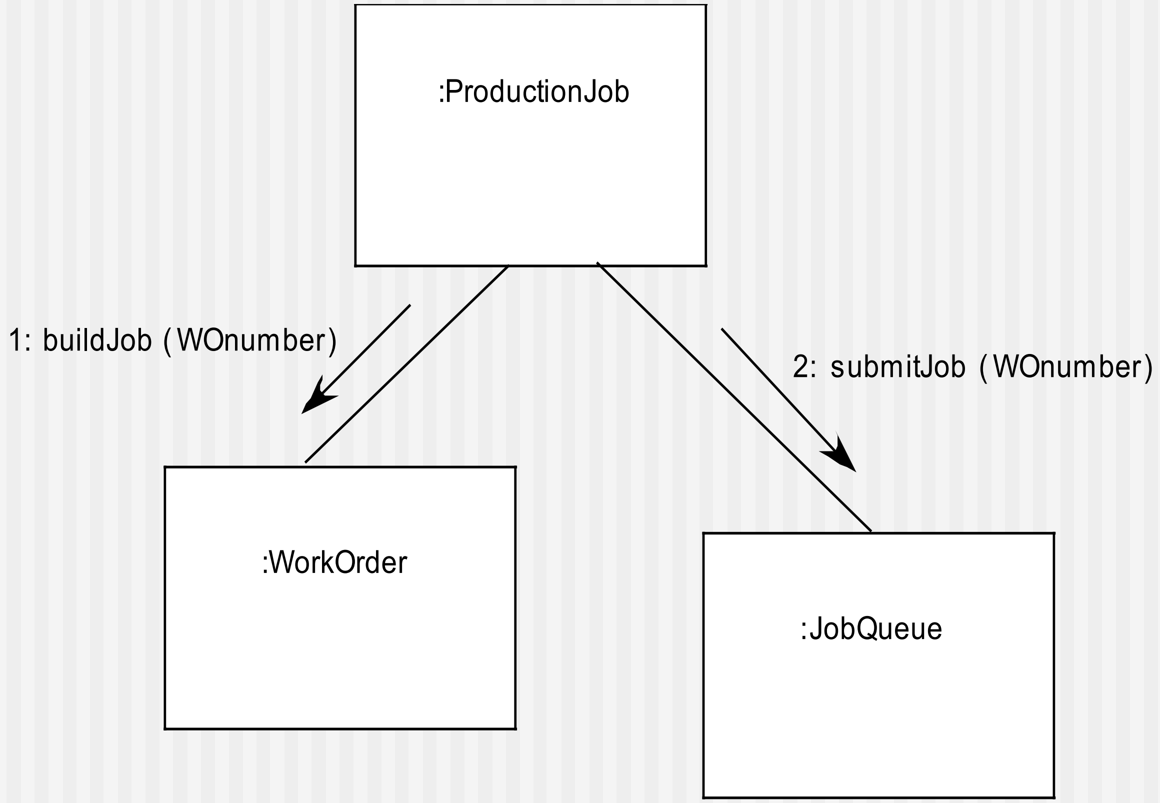
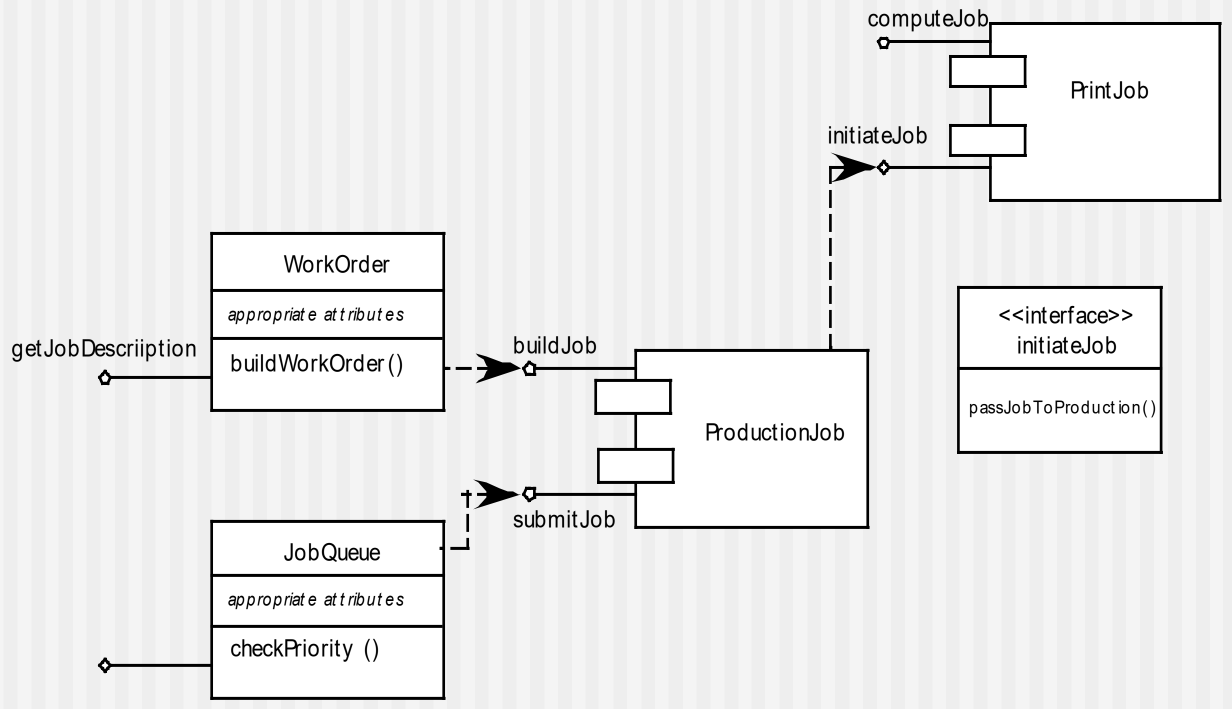


Fig.11.7 Refactoring

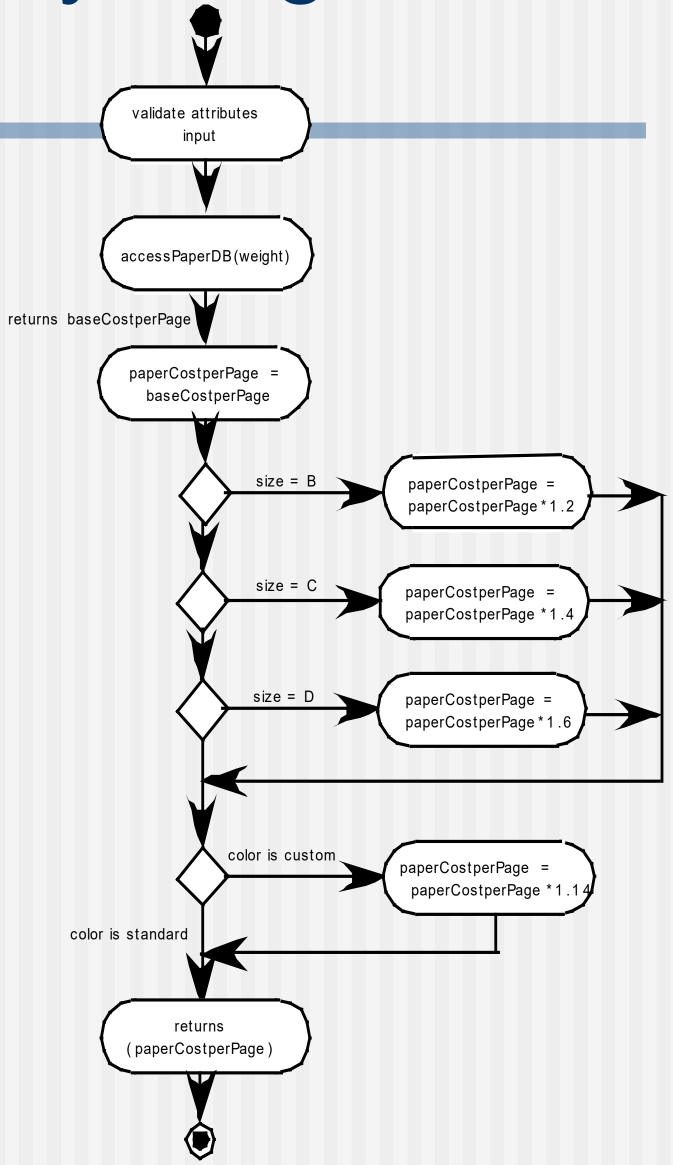


Component-Level Design-II

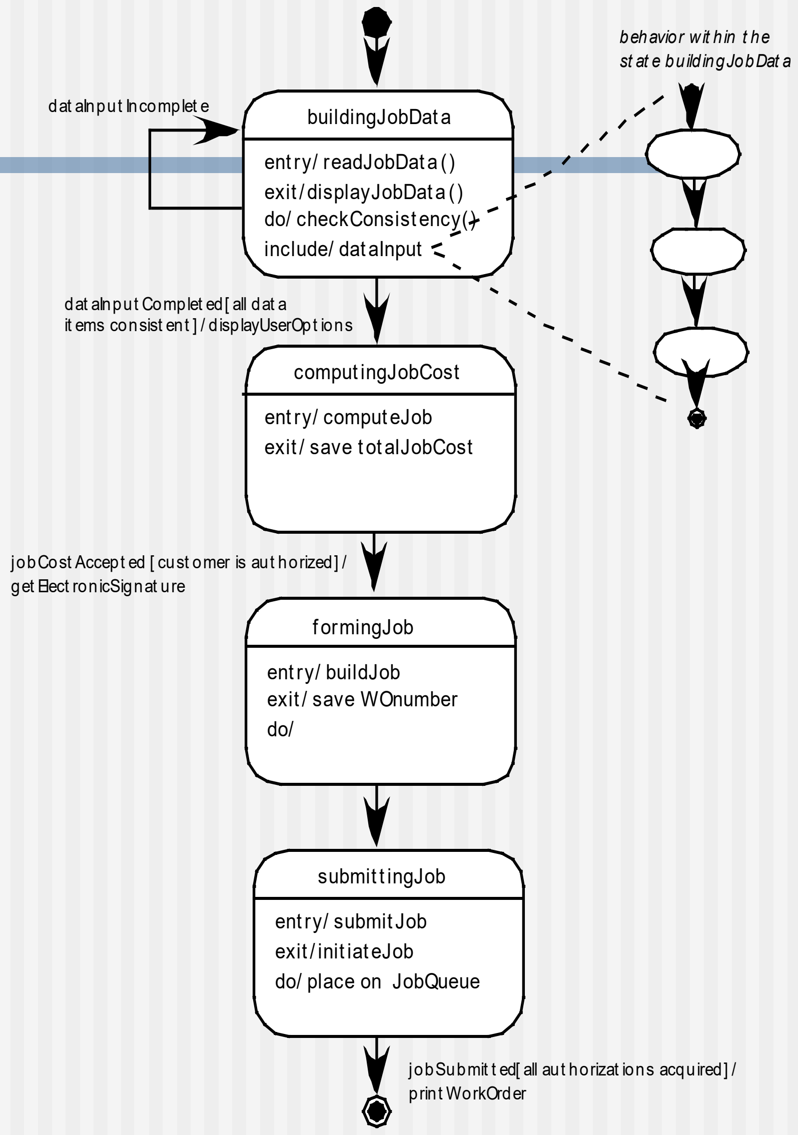
- Step 3c. Elaborate attributes and define data types and data structures required to implement them.
- Step 3d. Describe processing flow within each operation in detail.

  Fig. 11.8 Activity Diagram
- 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. Fig. 11.9 State Chart
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Factor every component-level design representation and always consider alternatives.

Fig.11.8 Activity Diagram


validate attributes input
accessPaperDB(weight)
returns baseCostperPage
paperCostperPage = baseCostperPage
size = B
paperCostperPage = paperCostperPage * 1.2
size = C
paperCostperPage = paperCostperPage * 1.4
size = D
paperCostperPage = paperCostperPage * 1.6
color is custom
paperCostperPage = paperCostperPage * 1.14
color is standard
returns (paperCostperPage)

Statechart


behavior within the state buildingJobData
dataInputIncomplete
buildingJobData
entry/ readJobData()
exit/displayJobData()
do/ checkConsistency()
include/ dataInput
dataInputCompleted[all data items consistent]/displayUserOptions
computingJobCost
entry/ computeJob
exit/ save totalJobCost
jobCostAccepted [customer is authorized]/ getElectronicSignature
formingJob
entry/ buildJob
exit/ save WOnumber
do/
submittingJob
entry/ submitJob
exit/initiateJob
do/ place on JobQueue
jobSubmitted[all authorizations acquired]/ printWorkOrder

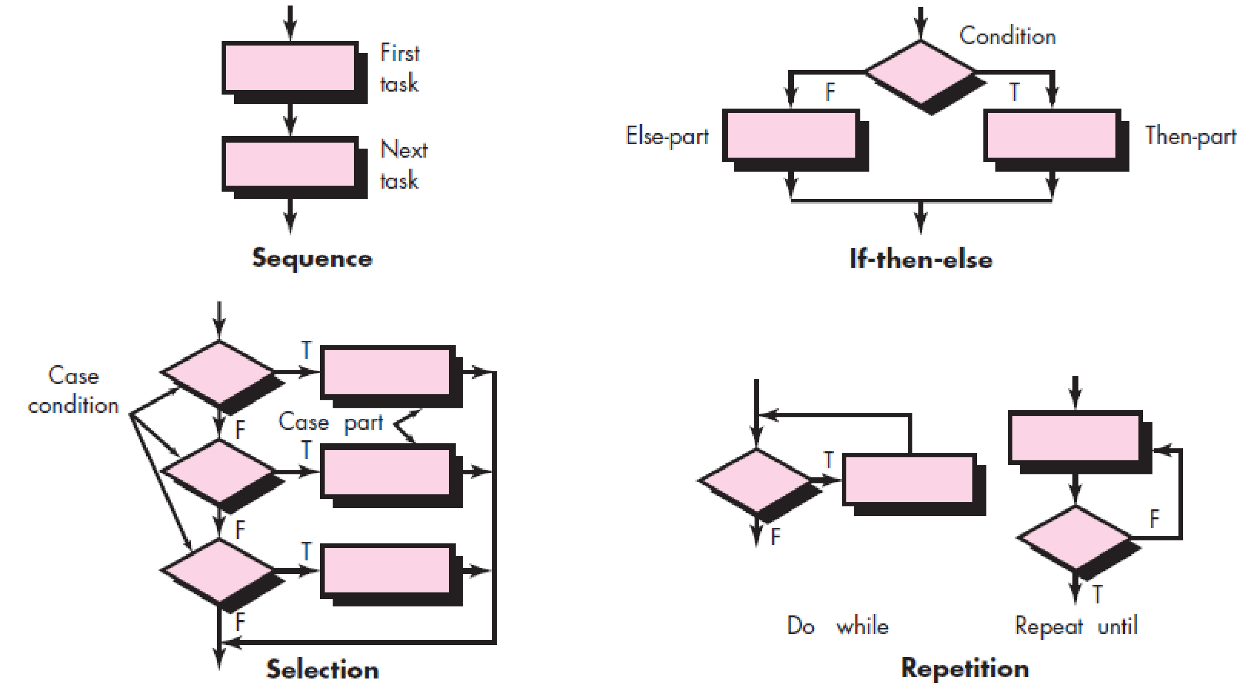
Component Design for WebApps

- WebApp component is
  - (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end-user, or
  - (2) a cohesive package of content and functionality that provides end-user with some required capability.
- Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

Designing Conventional Components

FIGURE 10.10

Flowchart constructs



Summary

- The purpose of component-level design is to define data structures, algorithms, interface characteristics, and communication mechanisms for each software component identified in the architectural design.
- Component-level design occurs after the data and architectural designs are established. The component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built. The work product produced is a design for each software component, represented using graphical, tabular, or text-based notation.
- Design walkthroughs are conducted to determine correctness of the data transformation or control transformation allocated to each component during earlier design steps.

Assignment

- Preview

  《Software Engineering》(9th Edition)

  Chapter 12　User Interface Design

  by R.S.Pressman