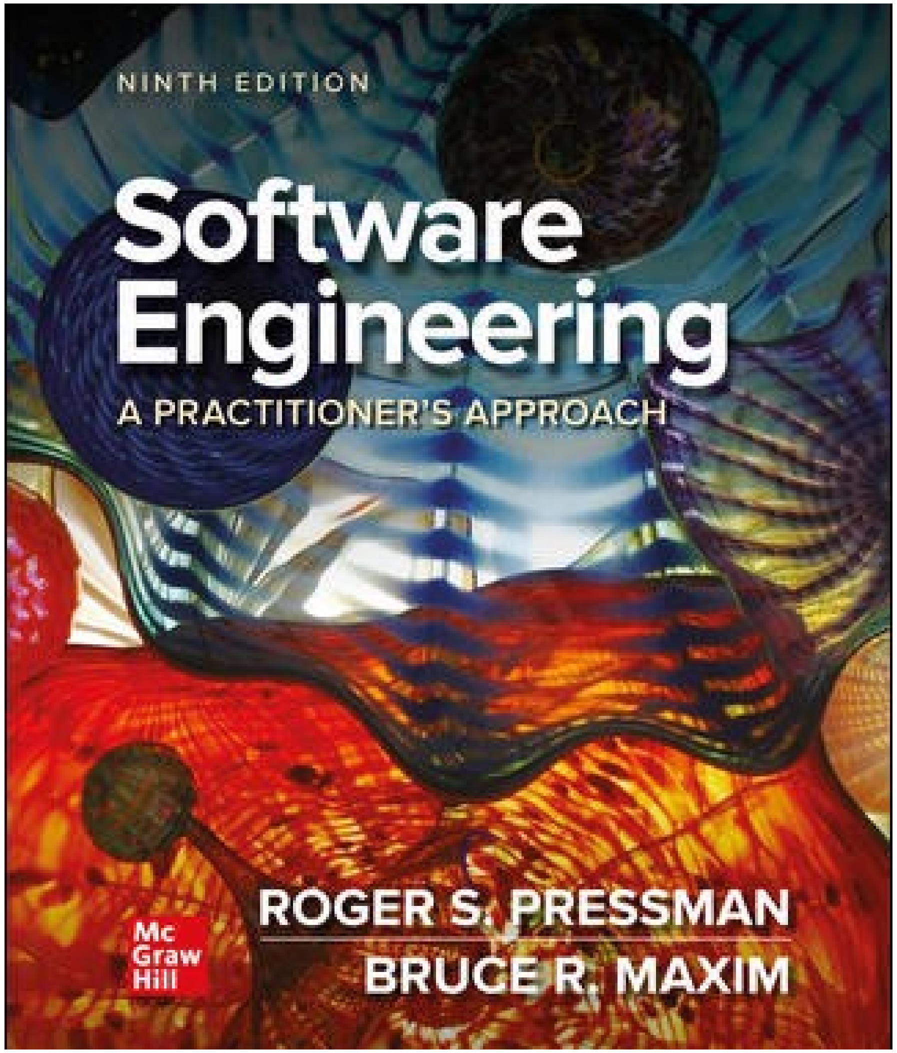
Because learning changes everything.®



# Chapter 11

Component-Level Design

**Part Two - Modeling**

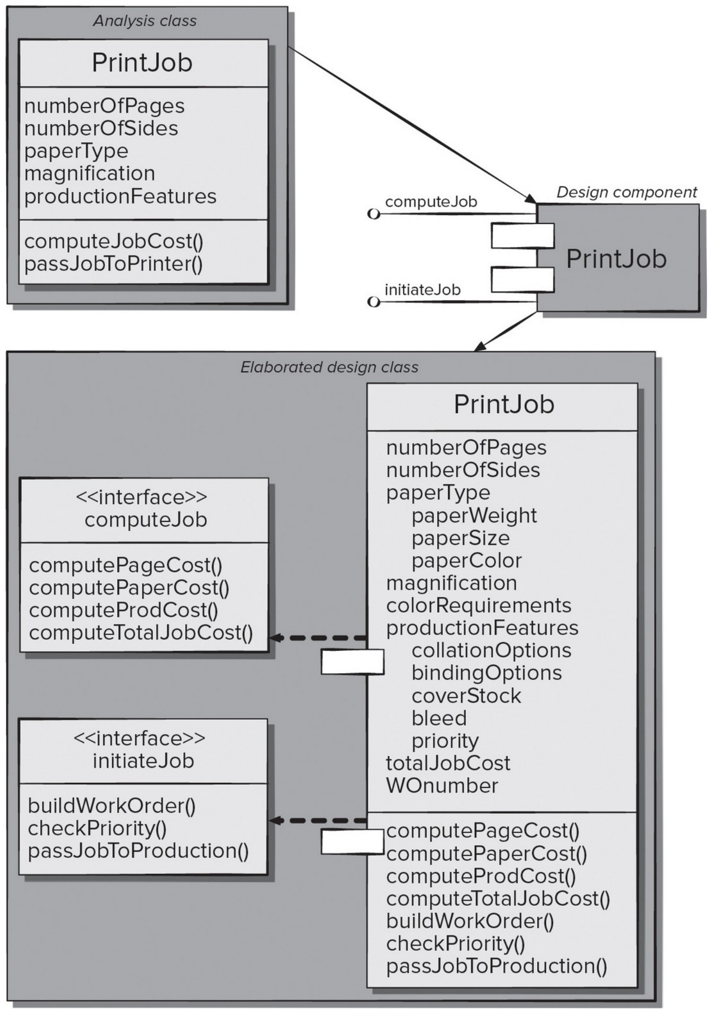
## What is a component?

- *OMG Unified Modeling Language Specification* defines a component as

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

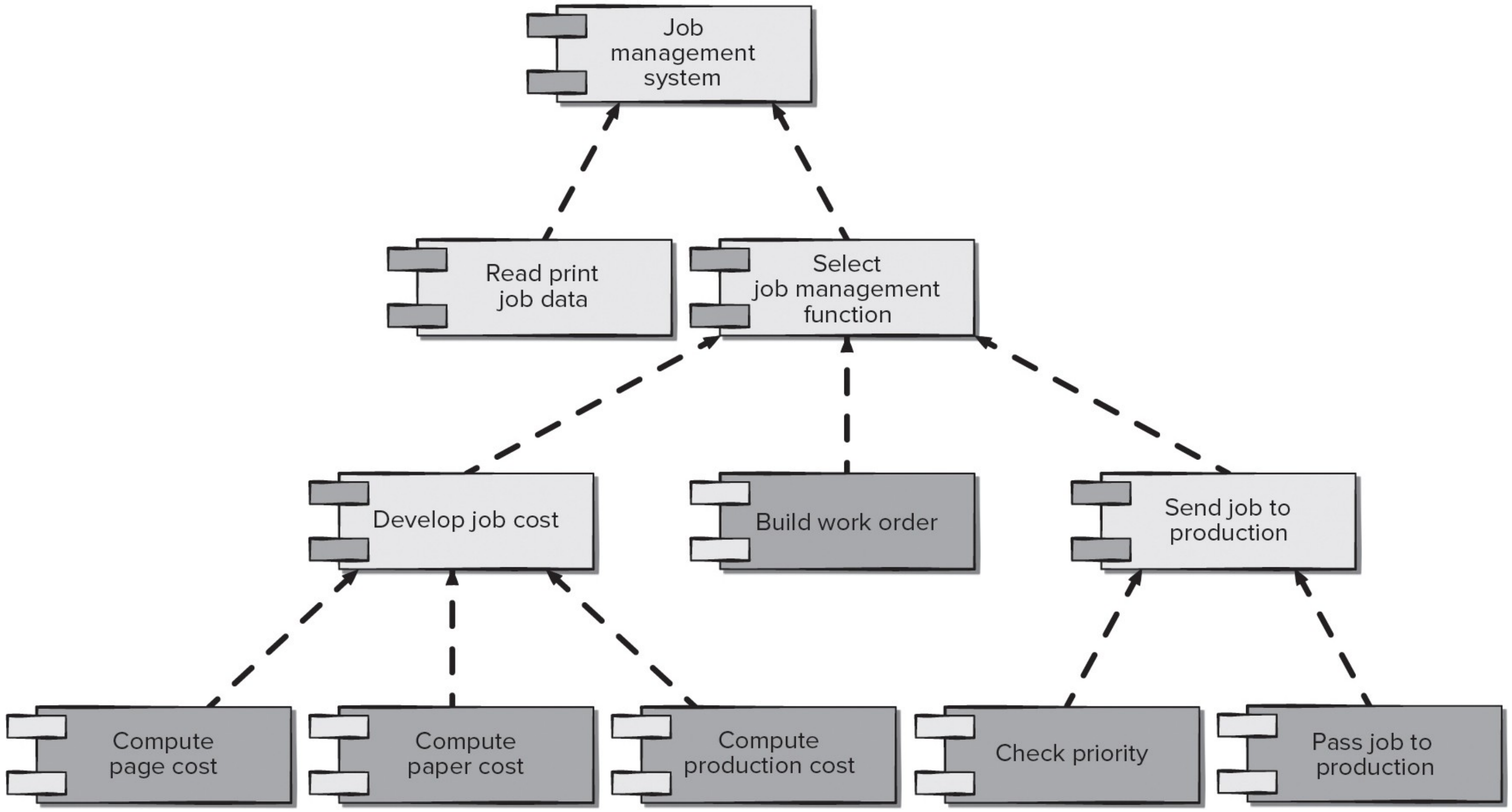
- *Object-Oriented view:* a component contains a set of collaborating classes.
- *Traditional view:* a component contains processing logic, internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.
- *Process-related view:* building systems out of reusable software components or design patterns selected from a catalog (component-based software engineering).

# Class-based Component-Level Design



Access the text alternative for slide images.

# Traditional Component-Level Design 1



Access the text alternative for slide images.

# Basic Component Design Principles

- **Open-Closed Principle (OCP)**. *"A module [component] 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."*

# Component-Level 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** - provide important information about communication and collaboration (as well as helping us to achieve the OPC).
- **Dependencies and Inheritance** – For readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

## Cohesion

**Traditional view** - the “single-mindedness” of a module.

**Object-Oriented view** - *cohesion* implies that a component encapsulates only attributes and operations that are closely related to one another and the component itself.

Levels of cohesion:

- Functional - module performs one and only one computation.
- Layer - occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.
- Communicational - All operations that access the same data are defined within one class.

## Coupling

**Traditional view** - degree to which a component is connected to other components and to the external world.

**Object-Oriented view** - qualitative measure of the degree to which classes are connected to one another.

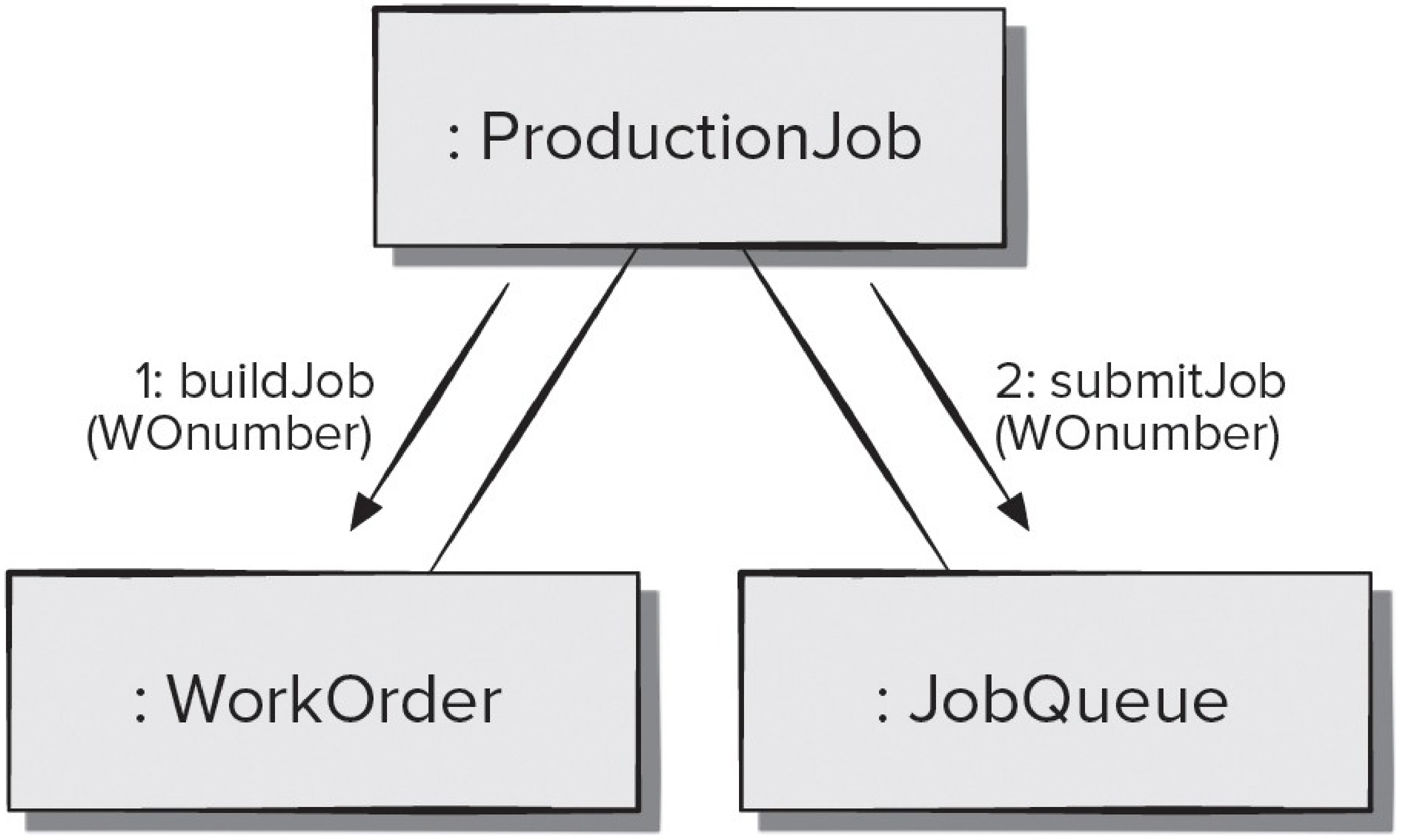
Levels of coupling.

- Content - occurs when one component "surreptitiously modifies data that is internal to another component.
- Control – occurs when control flags a passed to components to requests alternate behaviors when invoked.
- External - occurs when a component communicates or collaborates with infrastructure components.

## Component-Level Design 1

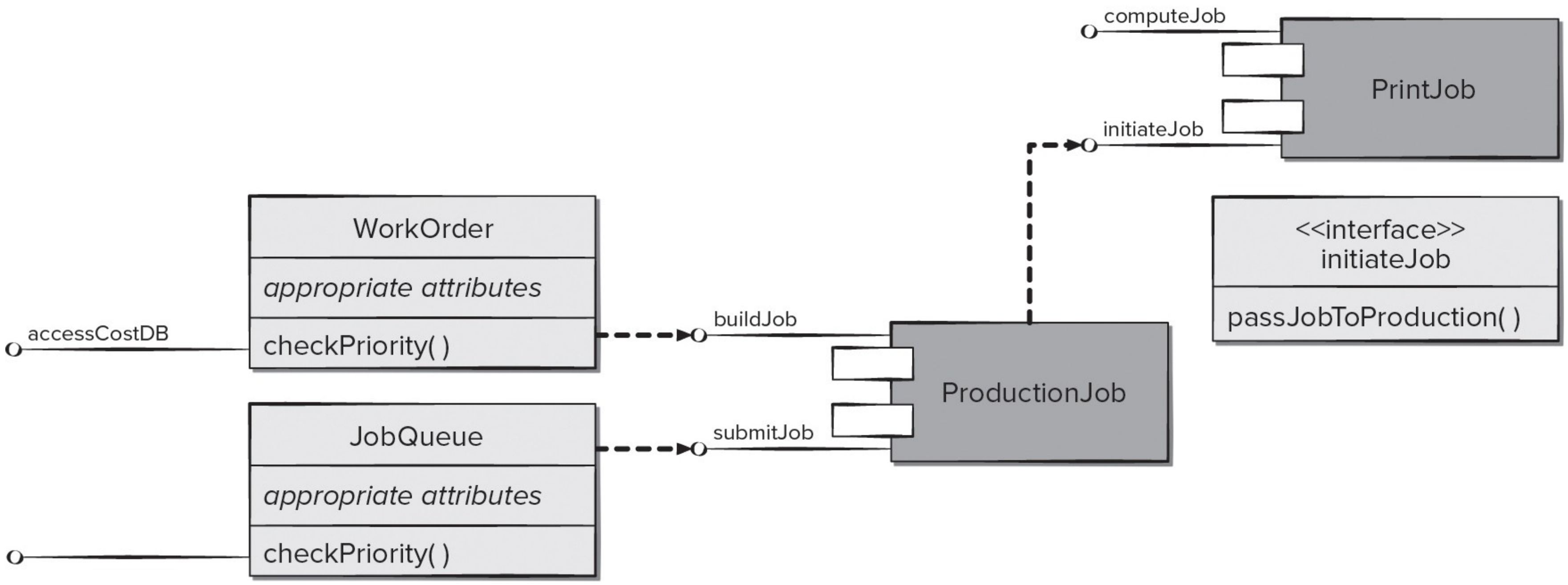
- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain.
- Step 3. Elaborate all design classes that are not acquired as reusable components.
- Step 3a. Specify message details when classes or component collaborate.
- Step 3b. Identify appropriate interfaces for each component.

# Collaboration Diagram with Message Detail



Access the text alternative for slide images.

# Define Interfaces

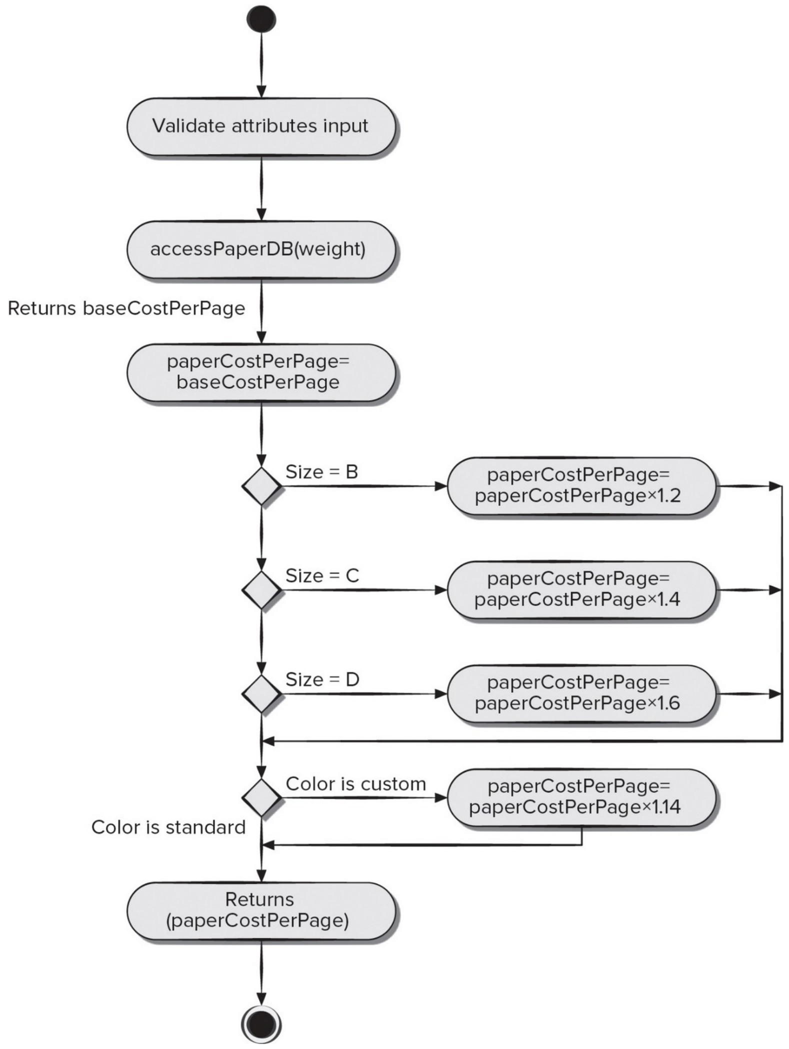


Access the text alternative for slide images.

## Component-Level Design 2

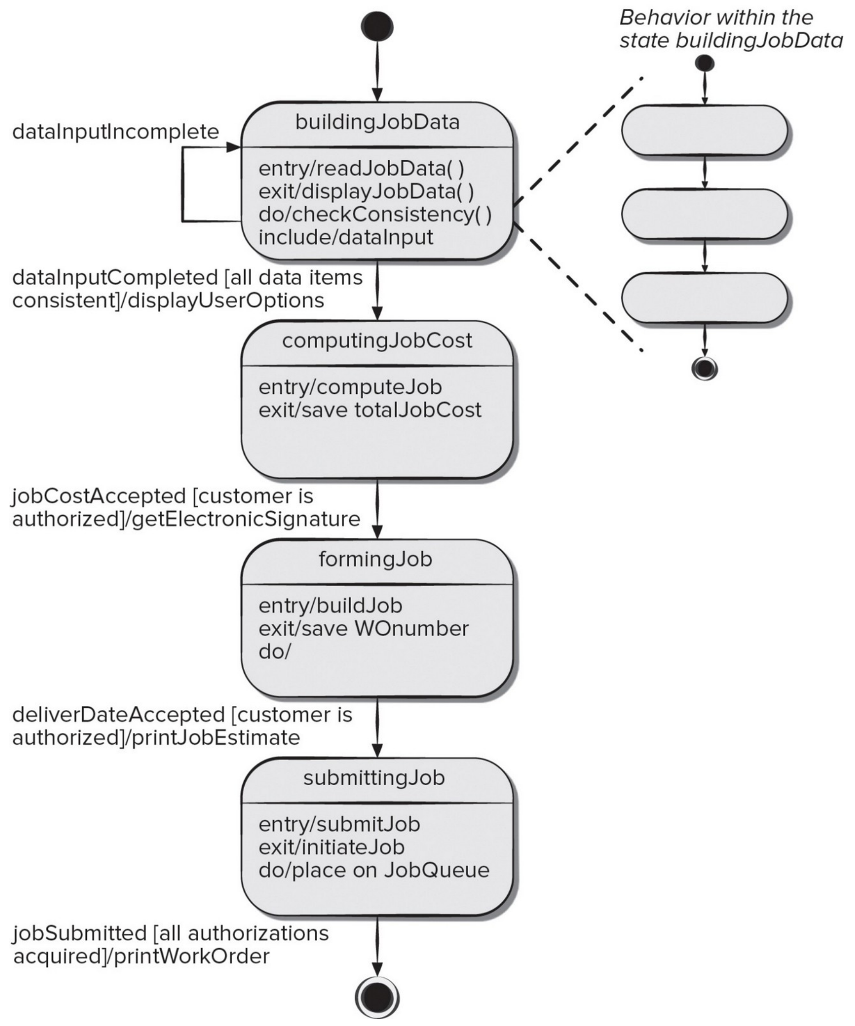
- Step 3c. Elaborate attributes and define data types and data structures required to implement them.
- Step 3d. Describe processing flow within each operation in detail.
- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop 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.

# Describe Processing Flow


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)

Access the text alternative for slide images.

# Elaborate Behavioral Representations


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/
deliverDateAccepted [customer is authorized]/printJobEstimate
submittingJob
entry/submitJob
exit/initiateJob
do/place on JobQueue
jobSubmitted [all authorizations acquired]/printWorkOrder

Access the text alternative for slide images.

## Component-Level 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.

Component-level design for WebApps often incorporates elements of content design and functional design.

## WebApp Content Design

Focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end-user

Consider a Web-based video surveillance capability within **SafeHomeAssured.com** potential content components can be defined for the video surveillance capability:

1. the content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras;
2. the collection of thumbnail video captures (each an separate data object), and
3. the streaming video window for a specific camera.

Each of these components can be separately named and manipulated as a package.

# WebApp Functional Design

Modern Web applications deliver increasingly sophisticated processing functions that:

1. perform localized processing to generate content and navigation capability in a dynamic fashion;
2. provide computation or data processing capability that is appropriate for the WebApp's business domain;
3. provide sophisticated database query and access, or.
4. establish data interfaces with external corporate systems.

To achieve these (and many other) capabilities, you will design and construct WebApp functional components that are identical in form to software components for conventional software.

# Component-Level Design for Mobile Apps

Thin web-based client.

- Interface layer only on device.
- Business and data layers implemented using web or cloud services.

Rich client.

- All three layers (interface, business, data) implemented on device.
- Subject to mobile device limitations.

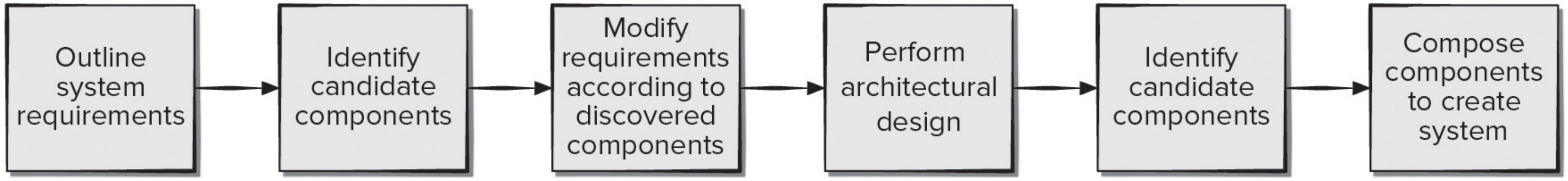
## Traditional Component-Level Design 2

- Design of processing logic is governed by the basic principles of algorithm design and structured programming.
- Design of data structures is defined by the data model developed for the system.
- Design of interfaces is governed by the collaborations that a component must effect.

## Component-Based Software Engineering (CBSE)

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?


Outline system requirements
Identify candidate components
Modify requirements according to discovered components
Perform architectural design
Identify candidate components
Compose components to create system

Access the text alternative for slide images.

## CBSE Benefits

- **Reduced lead time.** It is faster to build complete applications from a pool of existing components.
- **Greater return on investment (ROI).** Sometimes savings can be realized by purchasing components rather than redeveloping the same functionality in-house.
- **Leveraged costs of developing components.** Reusing components in multiple applications allows the costs to be spread over multiple projects.
- **Enhanced quality.** Components are reused and tested in many different applications.
- **Maintenance of component-based applications.** With careful engineering, it can be relatively easy to replace obsolete components with new or enhanced components.

## CBSE Risks

- **Component selection risks.** It is difficult to predict component behavior for black-box components, or there may be poor mapping of user requirements to the component architectural design.
- **Component integration risks.** There is a lack of interoperability standards between components; this often requires the creation of "wrapper code" to interface components.
- **Quality risks.** Unknown design assumptions made for the components makes testing more difficult, and this can affect system safety, performance, and reliability.
- **Security risks.** A system can be used in unintended ways, and system vulnerabilities can be caused by integrating components in untested combinations.
- **System evolution risks.** Updated components may be incompatible with user requirements or contain additional undocumented features.

## Component Refactoring

- Most developers would agree that refactoring components to improve quality is a good practice.
- It is hard to convince management to expend resources fixing components that are working correctly rather than adding new functionality to them,
- Changing software and failing to document the changes can lead to increasing technical debt.
- Reducing this technical debt often involves architectural refactoring, which is generally perceived by developers as both costly and risky.
- Developers can make of tools to examine change histories to identify the most cost-effective refactoring opportunities.

End of Main Content

Because learning changes everything.®

www.mheducation.com

## Accessibility Content: Text Alternatives for Images

# Class-based Component-Level Design – Text Alternative

Return to parent-slide containing images.

The illustration shows a class-based component level design. The diagram has two class levels. The first is labeled Analysis class and has the class (print job) with the attributes and operations. The second class is labeled elaborated class and also has a class (print job). The class has more attributes and operations. The class also has two interfaces labeled compute job and initiate job. The analysis class links to the design component labeled print job and which relays the process to the elaborated design class.

Return to parent-slide containing images.

# Traditional Component-Level Design [1] – Text Alternative

Return to parent-slide containing images.

The diagram shows a traditional component-level design. The diagram has 4 levels. The components and the connections in each level from bottom to top are. Level: 4 compute page cost; compute paper cost; compute production cost; check priority; pass job to production. Level 3: develop job cost; build work order; send job to production. Level 2: read print job data; select job management function. Level 1: job management system. The connections are as follow: compute page cost; compute paper cost; compute production cost connect to develop job cost. Check priority; pass job to production connect to send job to production. Develop job cost; build work order; send job to production connect to select job management function. Read print job data; select job management function connect to job management system.

Return to parent-slide containing images.

## Collaboration Diagram with Message Detail – Text Alternative

Return to parent-slide containing images.

The diagram shows a collaboration diagram with message detail. A production job collaborates with work order and job queue. Production job relays a message, build job with work order number to work order. Production job relays a message, submit job with work order number to job queue.

Return to parent-slide containing images.

# Define Interfaces – Text Alternative

Return to parent-slide containing images.

The flow diagram shows how to define interfaces. The diagram shows two classes on the left: work order and job queue. The classes have appropriate attributes and a check priority operation. work order accesses cost from data base. Work order relays a build job order to production job while job queue relays a submit job order to production job component. Production job relays an initiate job order to print job component. The print job component also has a compute job order.

Return to parent-slide containing images.

## Describe Processing Flow – Text Alternative

Return to parent-slide containing images.

The diagram shows how to describe a process flow. The flow starts with an initial state. The next process are: validate attributes input, access paper D B for weight, (this returns base cost per page), paper cost per page equals base cost per page. The next flow in the diagram shows three conditions: if size = B; if size = C; and if size = D. Each cost returns a paper cost per page. B = 1,2; C = 1.4; D = 1.6. The next flow in the diagram is a condition for color. If color is custom, then paper cost per page = 1.14 and if color is standard. The process then returns the final paper cost per page. End state.

Return to parent-slide containing images.

## Elaborate Behavioral Representations – Text Alternative

Return to parent-slide containing images.

The diagram shows a process flow with elaborate behavioral representation. From an initial state the process flows to a building job data class which has varying attributes. If the data input is incomplete the process loops back and repeats. When data input is completed then display user options. The next class is computing job cost which further has multiple attributes. If job cost is accepted, then costumer is authorized and system gets an electronic signature. The next class is forming job with its attributes. If deliver date is accepted customer is authorized and job estimate is printed. The next class is submitting job which has attributes. If job is submitted that means all authorizations are acquired the print work order.

Return to parent-slide containing images.

# Component-Based Software Engineering (CBSE) – Text Alternative

Return to parent-slide containing images.

A component-based software engineering (cbse) has the following flow: outline system requirement, identify candidate components, modify requirements according to discovered components, identify candidate components and compose components to create system.

Return to parent-slide containing images.