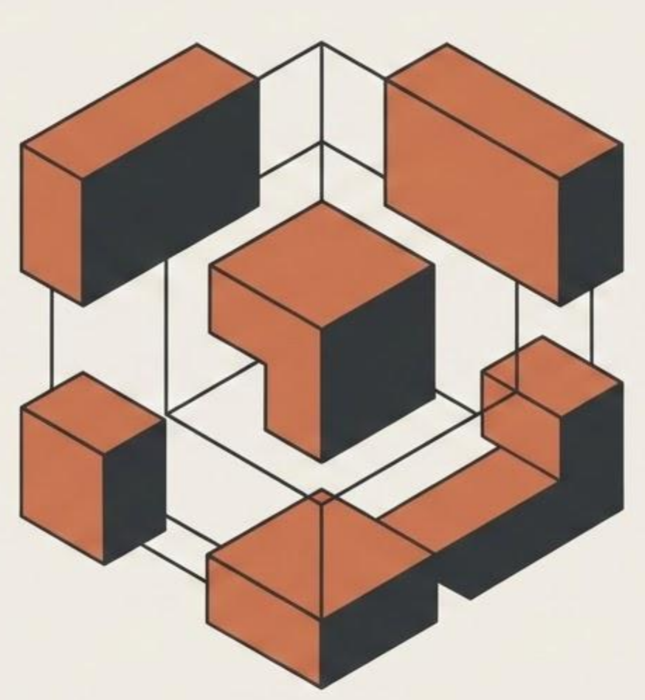
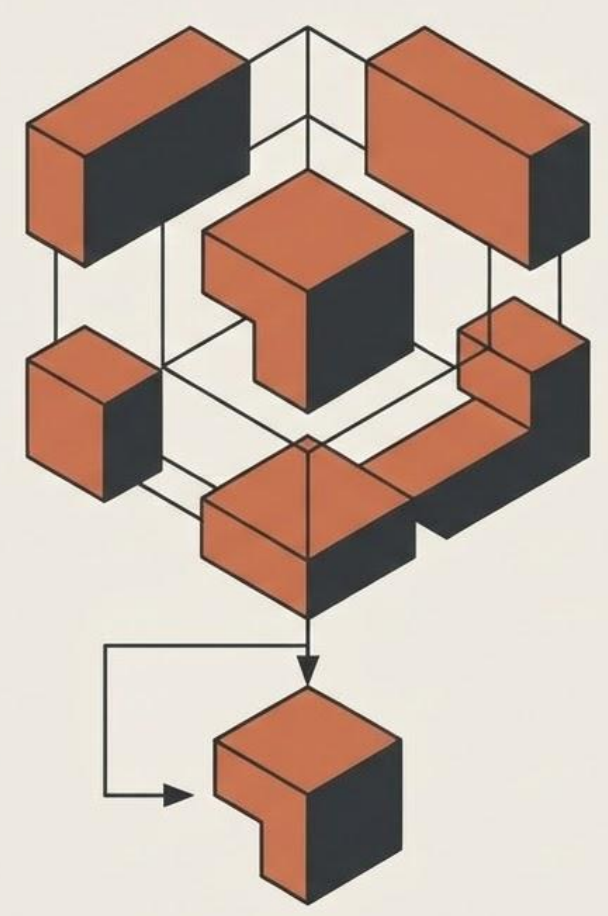


Chapter 10: Component-Level Design

Subject: Software Engineering
Program: BTech Computer Science and Engineering
Duration: 1 Hour

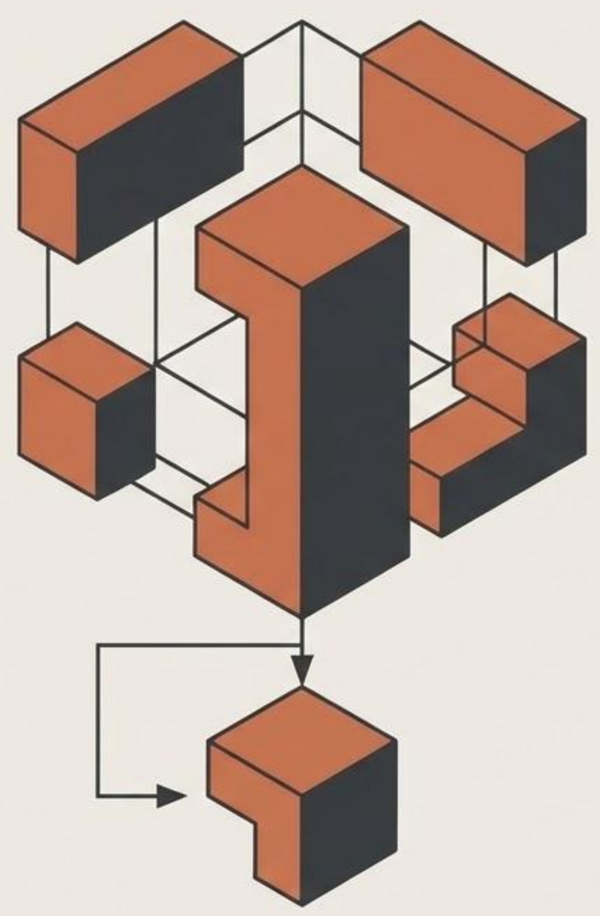


Component-Level Design

From Architecture to Code

Specifying the Building Blocks

Context: Zooming in from the “Big Picture” to the “Detailed Design.”



Zooming In

Moving Past the Big Boxes

- **The Context:** Architecture gave us the high-level structure—the subsystems and their connections.
- **The Question:** What happens inside those boxes?
- **The Goal:** How do we design the individual, concrete building blocks that developers will actually code?

Definition: Component-Level Design is the phase where we specify internal details with enough precision that coding becomes almost a mechanical translation.

Learning Objectives

By the end of this lecture, you will be able to:

- **Define Components**: Define a software component from three distinct perspectives:
  - Object-Oriented
  - Traditional
  - Process-Related
- **Apply Principles**: Apply basic design principles to create well-designed, class-based components.
- **Evaluate Quality**: Evaluate component quality using the critical metrics of Cohesion (sticking together) and Coupling (staying apart).
- **Describe the Process**: Describe the step-by-step activities involved in conducting component-level design.

Part 1: Defining the Building Block

Section 10.1: What Is a Component?

Focus:

- Modular
- Deployable
- Replaceable Units

Core Definition

The Physical Realization

Definition: A component is a modular, deployable, and replaceable part of a system.

Key Characteristics:

- Encapsulates implementation details.
- Exposes a set of defined interfaces.

Architectural Link: It is the "physical realization" of an abstraction defined during the architectural design phase.

An Object-Oriented View

Section 10.1.1: **Collaborating Classes**

Perspective: A component is viewed as a set of collaborating classes.

Structure:

- Responsibilities: Each class has a well-defined job.
- Collaboration: Classes work together to fulfill the component's goal.
- Inclusion: Includes the main design classes plus all auxiliary classes (UI, data access) needed to implement the abstraction.

Focus: Objects, Inheritance, Polymorphism.

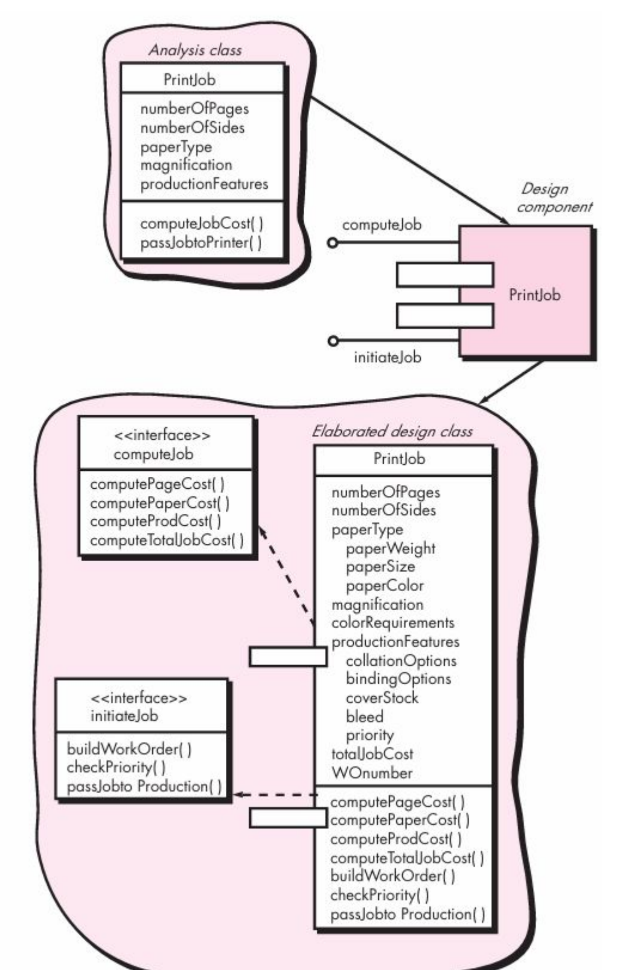
To illustrate this process of design elaboration, consider software to be built for a sophisticated print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility. During requirements engineering, an analysis class called **PrintJob** was derived. The attributes and operations defined during analysis are noted at the top of Figure 10.1. During architectural design, **PrintJob** is defined as a component within the software architecture and is represented using the shorthand UML notation[2] shown in the middle right of the figure. Note that **PrintJob** has two interfaces, *computeJob,* which provides job costing capability, and *initiateJob,* which passes the job along to the production facility. These are represented using the "lollipop" symbols shown to the left of the component box.

Recall that analysis modeling and design modeling are both iterative actions. Elaborating the original analysis class may require additional analysis steps, which are then followed with design modeling steps to represent the elaborated design class

Component-level design begins at this point. The details of the component **PrintJob** must be elaborated to provide sufficient information to guide implementation. The original analysis class is elaborated to flesh out all attributes and operations required to implement the class as the component **PrintJob.** Referring to the lower right portion of Figure 10.1, the elaborated design class **PrintJob** contains more detailed attribute information as well as an expanded description of operations required to implement the component. The interfaces *computeJob* and *initiateJob* imply communication and collaboration with other components (not shown here). For example, the operation *computePageCost()* (part of the *computeJob* interface) might collaborate with a **PricingTable** component that contains job pricing information. The *checkPriority()* operation (part of the *initiateJob* interface) might collaborate with a **JobQueue** component to determine the types and priorities of jobs currently awaiting production.

FIGURE 10.1

Elaboration of a design component



The Traditional View

Section 10.1.2: **Modules and Logic**

Perspective: A component is a functional element—a module.

Structure:

- Encompasses processing logic and internal data structures.
- Uses an interface for invocation and data passing.

Origin: Derived from structured programming.

- Procedural Abstraction: A single function or subroutine.
- Data Abstraction: A package of related functions (e.g., a standard Math library).

Focus: Functions, Data Structures, Decomposition.

would be derived during requirements modeling. Assume that these are mapped into an architecture shown in Figure 10.2. Each box represents a software component. Note that the shaded boxes are equivalent in function to the operations defined for the **PrintJob** class discussed in Section 10.1.1. In this case, however, each operation is represented as a separate module that is invoked as shown in the figure. Other modules are used to control processing and are therefore control components.

During component-level design, each module in Figure 10.2 is elaborated. The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed using the stepwise refinement approach discussed in Chapter 8. The behavior of the module is sometimes represented using a state diagram.

FIGURE 10.2

Structure chart for a traditional system

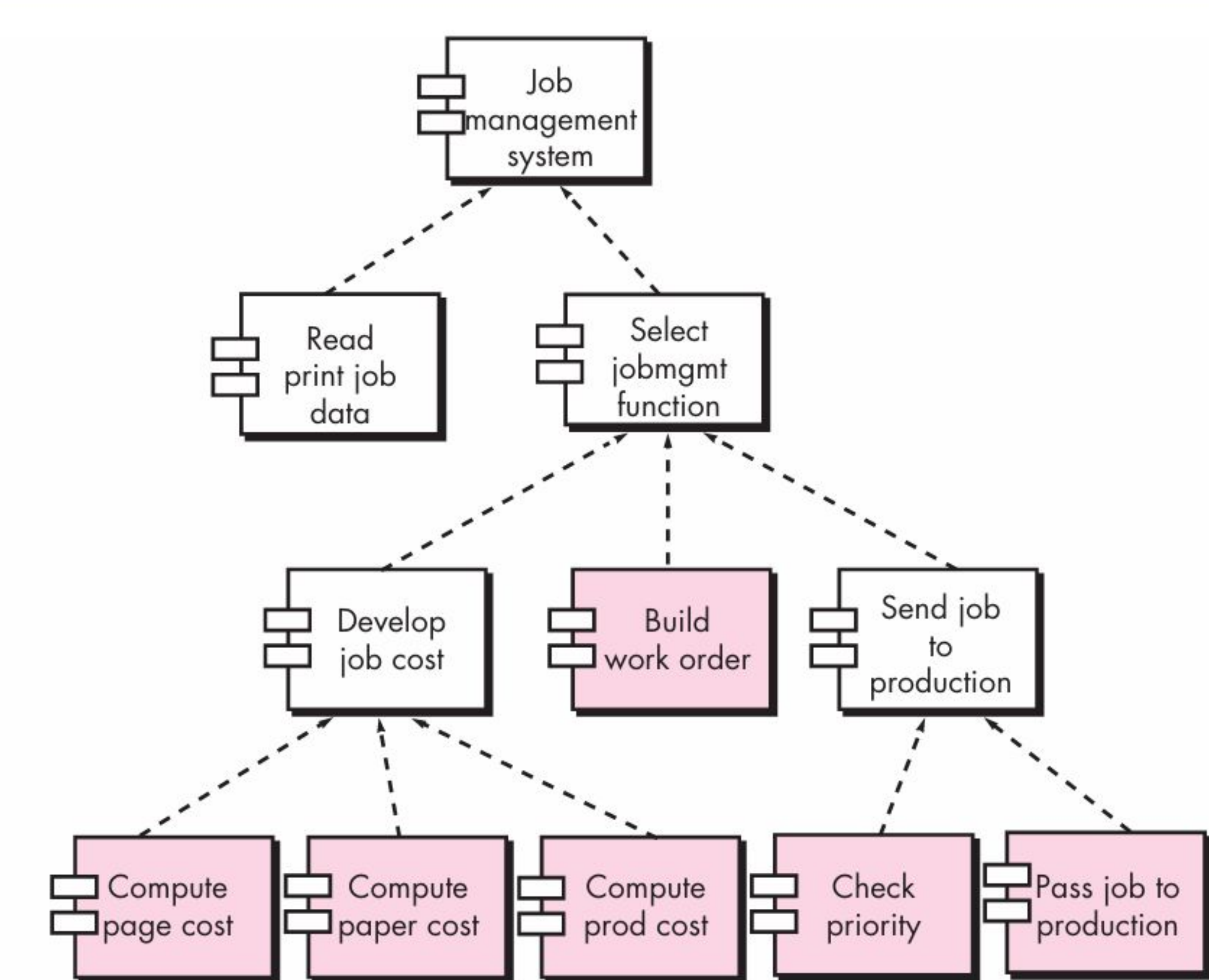


Figure 10.3 represents the component-level design using a modified UML notation. The *ComputePageCost* module accesses data by invoking the module *getJobData,* which allows all relevant data to be passed to the component, and a database interface, *accessCostsDB,* which enables the module to access a database that contains all printing costs. As design continues, the *ComputePageCost* module is elaborated to provide algorithm detail and interface detail (Figure 10.3). Algorithm detail can be represented using the pseudocode text shown in the figure or with a UML activity diagram. The interfaces are represented as a collection of input and output data objects or items. Design elaboration continues until sufficient detail is provided to guide construction of the component.

FIGURE 10.3 Component-level design for *ComputePageCost*

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

A Process-Related View

Section 10.1.3: **The Artifact**

Perspective: A component is a work product resulting from the design phase.

- Nature: It is a tangible design artifact (e.g., a UML diagram, detailed pseudocode) passed to the construction/coding phase.
- Agile Context: In modern processes, a component often corresponds to a specific User Story or feature designed and built within a single iteration.

Synthesis: What Makes a Good Component?

Universal Qualities

Regardless of the viewpoint (OO, Traditional, or Process), a good component must be:

- **Coherent:** It makes sense as a logical unit.
- **Encapsulated:** Internal details are hidden from the outside.
- **Interface-Driven:** It communicates only through clear, well-defined interfaces.

Part 2: Designing Class-Based Components

Section 10.2: Principles and Metrics

Focus: Creating Robust, Maintainable Classes

Basic Design Principles (SOLID)

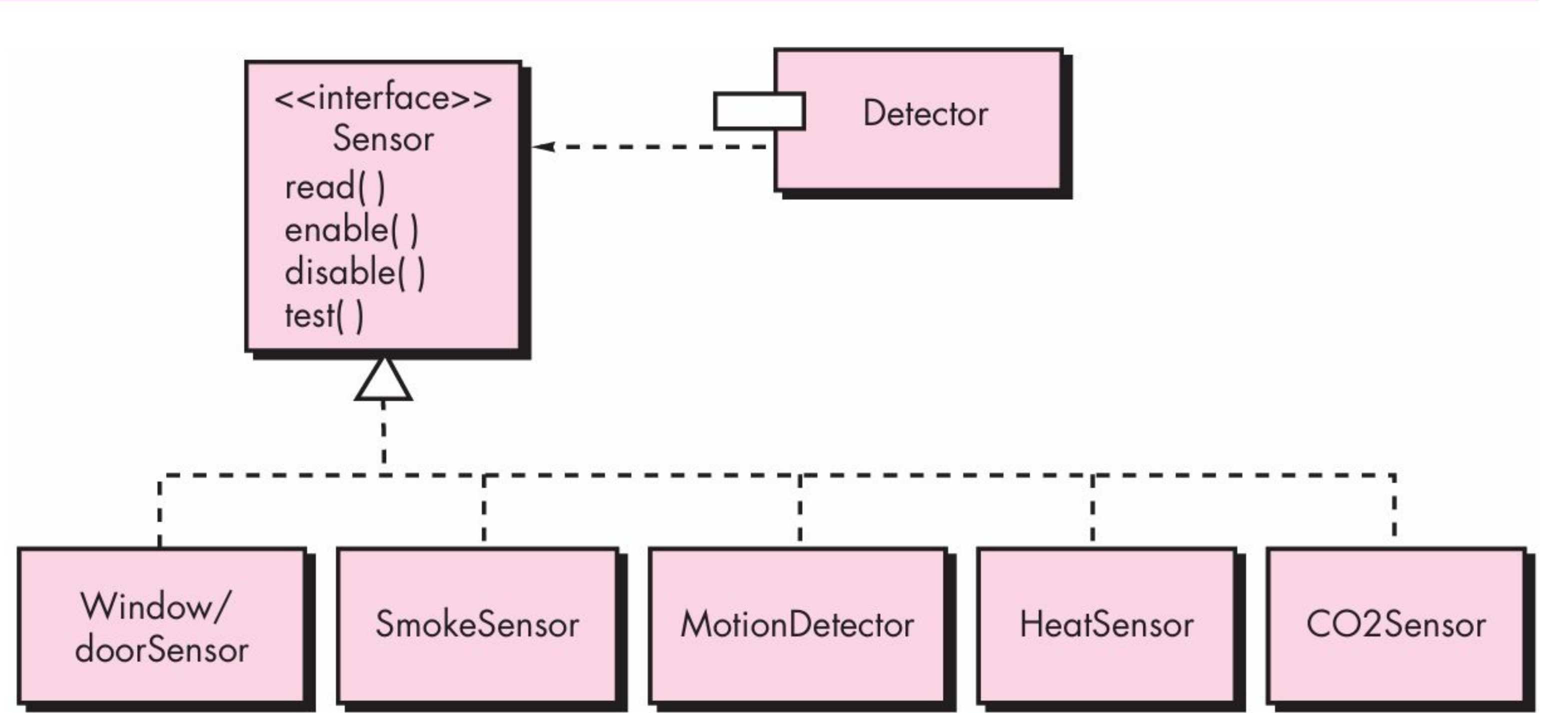
Section 10.2.1: The Foundation

- **Open-Closed Principle (OCP):** Open for extension, closed for modification. Add new code; don't change old code.
- **Liskov Substitution Principle (LSP):** Subtypes must be substitutable for their base types without breaking the program.
- **Dependency Inversion Principle (DIP):** Depend on abstractions (interfaces), not concretions. High-level modules should not depend on low-level modules.
- **Interface Segregation Principle (ISP):** Many specific interfaces are better than one general-purpose one. Clients shouldn't depend on methods they don't use.
- **Release Reuse Equivalency Principle (REP):** The granule of reuse is the granule of release. If you reuse it, you must be able to track and release it.

One way to accomplish OCP for the **Detector** class is illustrated in Figure 10.4. The *sensor* interface presents a consistent view of sensors to the detector component. If a new type of sensor is added no change is required for the **Detector** class (component). The OCP is preserved.

FIGURE 10.4

Following the OCP



Component-Level Design Guidelines

Section 10.2.2: Practical Rules

- **Naming:** Name components for their overall function (e.g., AccountManager).
- **Interfaces:**
    - Use Nouns for data-only interfaces.
    - Use Verbs for operation-only interfaces.
- **Dependencies:** Must be explicitly modeled (dashed arrows in UML) and minimized.
- **Contract:** Functionality defined entirely by preconditions and postconditions.
- **Error Handling:** Designed as a core feature, not an afterthought.

Cohesion (Sticking Together)

Section 10.2.3: From Worst to Best

Definition: The degree to which the responsibilities of a single component are related.

The Spectrum: (From Worst to Best)

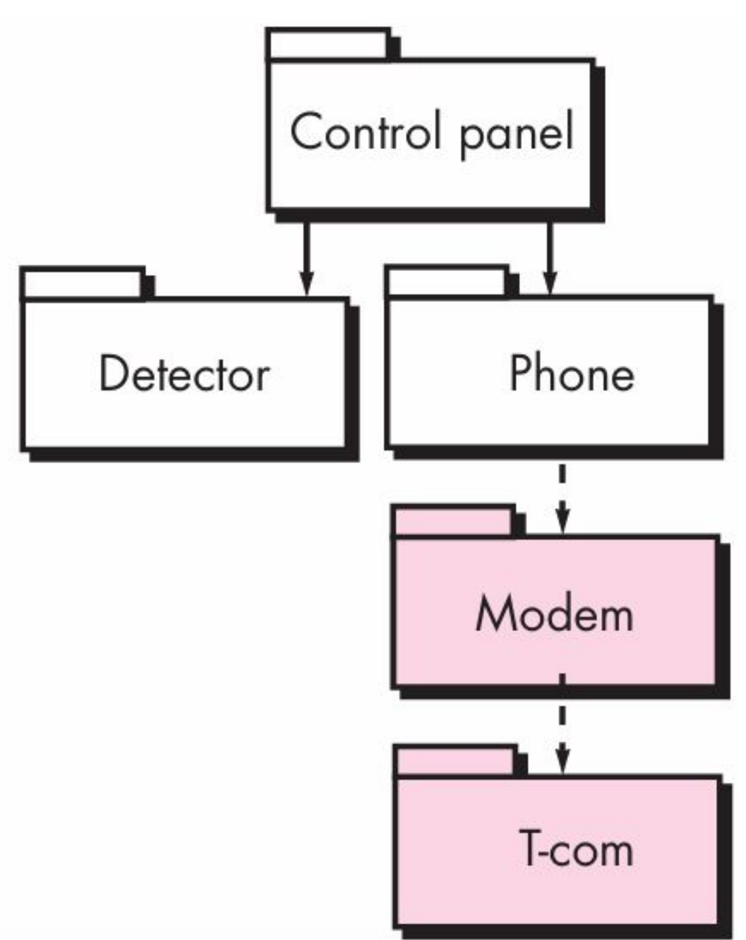
- **Coincidental (Worst):** Random, unrelated parts (e.g., a "Utils" class).
- **Logical:** Logically related (all inputs), but functionally different.
- **Temporal:** Related by timing (e.g., initialize()).
- **Procedural:** Steps in a specific algorithm.
- **Communicational:** Operate on the same data.
- **Sequential:** Output of one is input to the next.
- **Functional (Best):** Every part performs one single, well-defined task.

Goal: MAXIMIZE Functional Cohesion.

Layer. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers. Consider, for example, the *SafeHome* security function requirement to make an outgoing phone call if an alarm is sensed. It might be possible to define a set of layered packages as shown in Figure 10.5. The shaded packages contain infrastructure components. Access is from the control panel package downward.

FIGURE 10.5

Layer cohesion



Coupling (Staying Apart)

Section 10.2.4: From Worst to Best

Definition: The degree of interdependence between components.

The Spectrum:

- **Content (Worst):** Modifying another component's internal data or logic.
- **Common/Global:** Sharing global data (side-effect prone).
- **Control:** Passing flags to control another's logic.
- **Stamp:** Passing a whole data structure when only part is needed.
- **Data (Best):** Communicating via simple data parameters.

Goal: MINIMIZE Coupling. Aim for Data Coupling.

The Golden Rule & Exercise

Applying the Concepts

The Rule:

Strive for High Cohesion and Low Coupling.

In-Class Exercise:

Scenario: A class **StudentProcessor** has methods:

- **registerForCourse()**
- **calculateGPA()**
- **printTranscript()**
- **sendEmailToParents()**

- **Question 1:** What is its likely cohesion level? (Hint: It's doing too much).
- **Question 2:** How would you refactor this to improve cohesion?

Part 3: Conducting Component-Level Design

Section 10.3

Focus: The Step-by-Step Process

The Design Process

Inputs & Outputs

Goal: Create detailed specifications for each component/class.

Inputs:

- **Architectural Model** (The "Big Picture").
- **Analysis Classes** (Refined from requirements).
- **Use Cases** (Functional scenarios).

Outputs: Detailed specifications ready for coding.

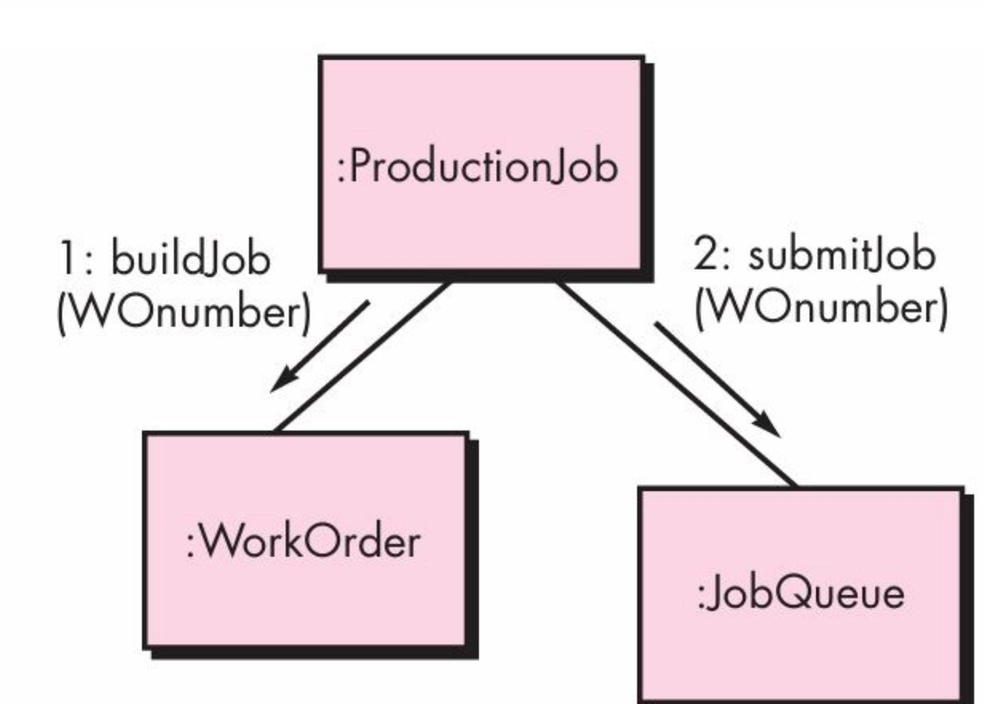
Figure 10.6 illustrates a simple collaboration diagram for the printing system discussed earlier. Three objects, **ProductionJob, WorkOrder,** and **JobQueue,** collaborate to prepare a print job for submission to the production stream. Messages are passed between objects as illustrated by the arrows in the figure. During requirements modeling the messages are specified as shown in the figure. However, as design proceeds, each message is elaborated by expanding its syntax in the following manner [Ben02]:

[guard condition] sequence expression (return value) :=
message name (argument list)

where a **[guard condition]** is written in Object Constraint Language (OCL)[5] and specifies any set of conditions that must be met before the message can be sent; **sequence expression** is an integer value (or other ordering indicator, e.g., 3.1.2) that indicates the sequential order in which a message is sent; **(return value)** is the name of the information that is returned by the operation invoked by the message; **message name** identifies the operation that is to be invoked, and **(argument list)** is the list of attributes that are passed to the operation.

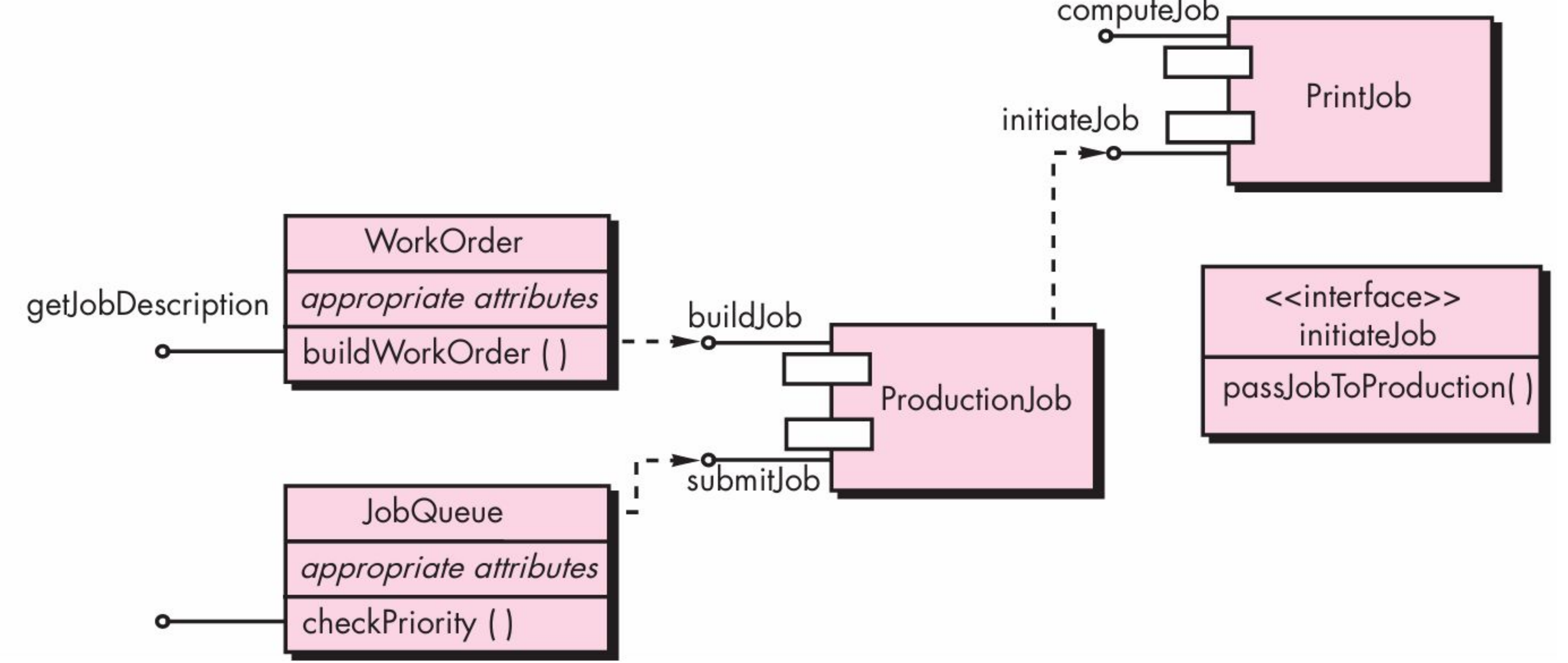
FIGURE 10.6

Collaboration diagram with messaging



Referring to Figure 10.1, it can be argued that the interface *initiateJob* does not exhibit sufficient cohesion. In actuality, it performs three different subfunctions—building a work order, checking job priority, and passing a job to production. The interface design should be refactored. One approach might be to reexamine the design classes and define a new class **WorkOrder** that would take care of all activities associated with the assembly of a work order. The operation *buildWorkOrder()* becomes a part of that class. Similarly, we might define a class **JobQueue** that would incorporate the operation *checkPriority()*. A class **ProductionJob** would encompass all information associated with a production job to be passed to the production facility. The interface *initiateJob* would then take the form shown in Figure 10.7. The interface *initiateJob* is now cohesive, focusing on one function. The interfaces associated with **ProductionJob, WorkOrder,** and **JobQueue** are similarly single-minded.

FIGURE 10.7 Refactoring interfaces and class definitions for PrintJob

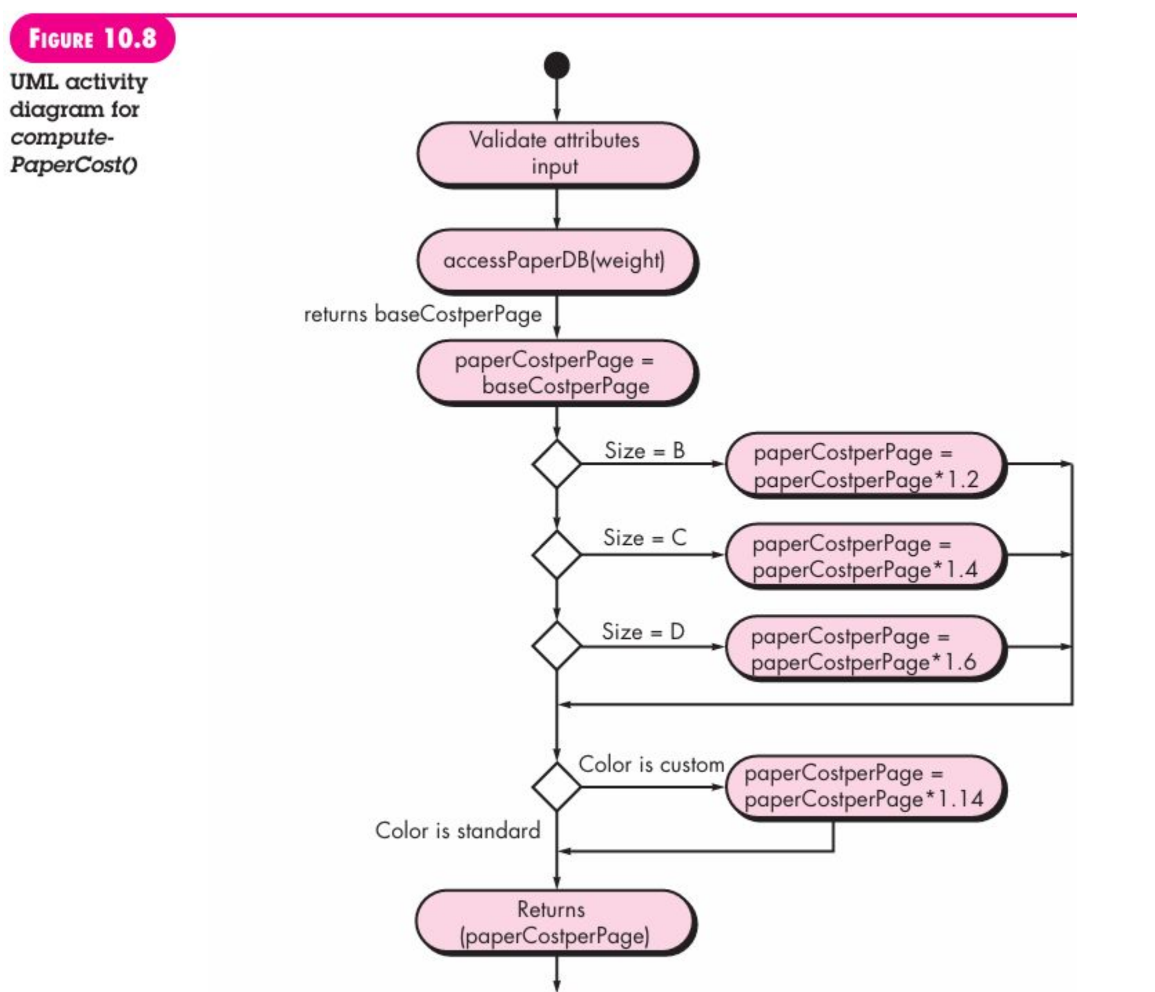


Use stepwise elaboration as you refine the component design. Always ask, "Is there a way this can be simplified and yet still accomplish the same result?"

If the algorithm required to implement *computePaperCost()* is simple and widely understood, no further design elaboration may be necessary. The software engineer who does the coding will provide the detail necessary to implement the operation. However, if the algorithm is more complex or arcane, further design elaboration is required at this stage. Figure 10.8 depicts a UML activity diagram for *computePaperCost().* When activity diagrams are used for component-level design specification, they are generally represented at a level of abstraction that is somewhat higher than source code. An alternative approach—the use of pseudocode for design specification—is discussed in Section 10.5.3.

FIGURE 10.8

UML activity diagram for *compute-PaperCost()*



(mode of behavior) of the object. To understand the dynamic behavior of an object, you should examine all use cases that are relevant to the design class throughout its life. These use cases provide information that helps you to delineate the events that affect the object and the states in which the object resides as time passes and events occur. The transitions between states (driven by events) are represented using a UML statechart [Ben02] as illustrated in Figure 10.9.

The transition from one state (represented by a rectangle with rounded corners) to another occurs as a consequence of an event that takes the form:

Event-name (parameter-list) [guard-condition] / action expression

where **event-name** identifies the event, **parameter-list** incorporates data that are associated with the event, **guard-condition** is written in Object Constraint Language (OCL) and specifies a condition that must be met before the event can occur, and **action expression** defines an action that occurs as the transition takes place.

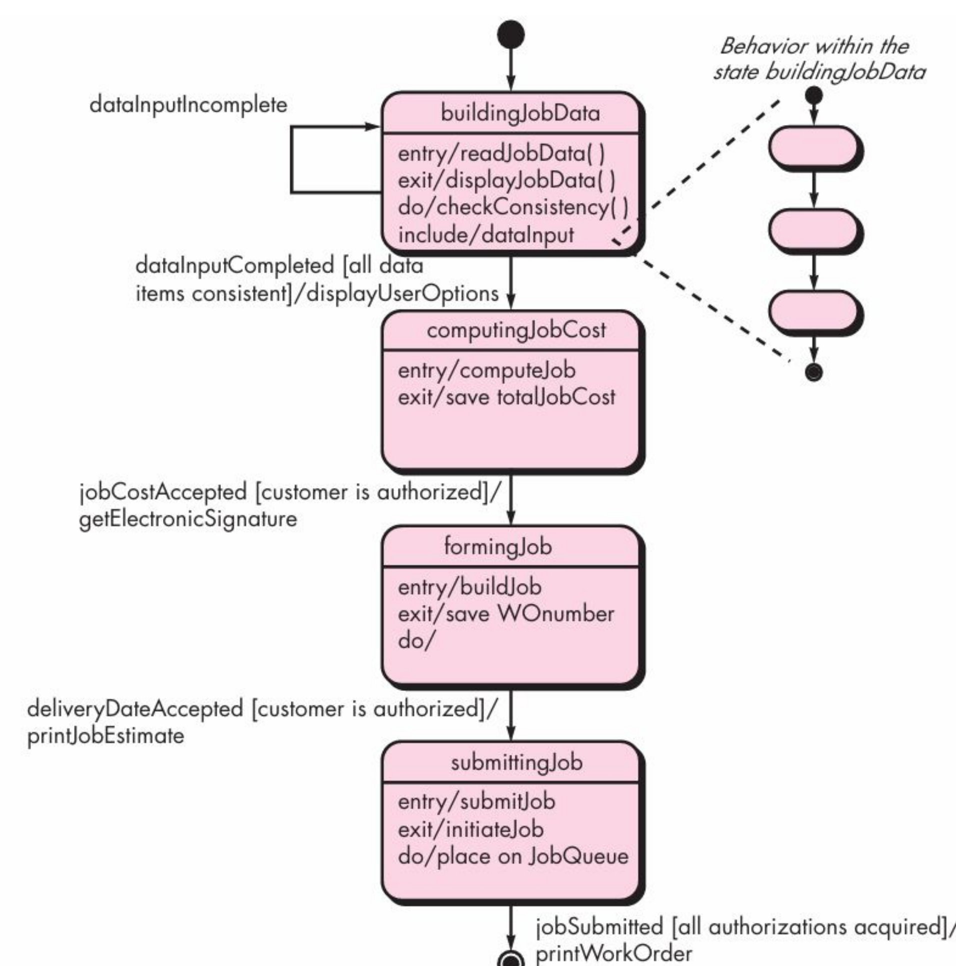
Referring to Figure 10.9, each state may define *entry/* and *exit/* actions that occur as transition into the state occurs and as transition out of the state occurs, respectively. In most cases, these actions correspond to operations that are relevant to the class that is being modeled. The *do/* indicator provides a mechanism for indicating activities that

occur while in the state, and the *include/* indicator provides a means for elaborating the behavior by embedding more statechart detail within the definition of a state.

It is important to note that the behavioral model often contains information that is not immediately obvious in other design models. For example, careful examination of the statechart in Figure 10.9 indicates that the dynamic behavior of the **PrintJob** class is contingent upon two customer approvals as costs and schedule data for the print job are derived. Without approvals (the guard condition ensures that the customer is authorized to approve) the print job cannot be submitted because there is no way to reach the *submittingJob* state.

FIGURE 10.9

Statechart fragment for PrintJob class



Key Activities (Step 1-4)

Elaborating the Model

- **Identify Components:** Select the “big boxes” from the architectural model for elaboration.
- **Elaborate Classes:**
  - Add detailed attributes (types, visibility: public/private).
  - Define operations (signatures: name, parameters, return types).
- **Specify Interfaces:** Define precisely how components communicate.
- **Specify Persistence:** Determine how objects are stored (Database schemas, File formats).

Key Activities (Step 5-8)

Logic & Refinement

- **5. Specify Algorithms:** Use Pseudocode, Activity Diagrams, or Decision Tables for complex logic.
- **6. Apply Patterns:** Implement design patterns (Strategy, Factory) to solve problems elegantly.
- **7. Refine Design:** Analyze Cohesion and Coupling. Refactor weak areas.
- **8. Review:** Conduct a technical review to catch errors before a single line of code is written.

Tools & Notation

Documenting the Design

- **UML Class Diagrams:** For static structure (attributes, methods, relationships).
- **UML Sequence Diagrams:** For interaction details (who calls whom?).
- **UML State Diagrams:** For components with complex state lifecycles (e.g., an Order moving from "Pending" to "Shipped").
- **Pseudocode (PDL):** The primary tool for specifying internal algorithm logic in a language-agnostic way.

The Last Checkpoint Before Code

Principles, Metrics, and Specification

Wrapping up Component-Level Design

Summary of Core Concepts

Defining the Building Blocks

- What is a Component? A modular, replaceable building block.
- OO View: Collaborating classes.
- Traditional View: Functional modules.
- Process View: A work product / user story.
- Guiding Principles: Design is not random; it is guided by SOLID principles (OCP, LSP, DIP) and practical guidelines.

The Golden Rule of Design

Metrics that Matter

The Goal: We strive for High Functional Cohesion and Low Data Coupling.

- Cohesion: Measures internal relatedness (Does this class make sense as a unit?).
- Coupling: Measures external interdependence (How hard is it to rip this class out and use it elsewhere?).

The Process

From Concept to Specification

Systematic Elaboration: Component-level design is the process of taking “big boxes” and filling them with details.

Tools:

- UML: For static structure and relationships.
- Pseudocode: For specifying algorithms and logic.
- Result: Precise specifications for classes, data, algorithms, and interfaces.

Final Thought

Design vs. Debugging

"Component-level design is where the rubber meets the road. It is the last checkpoint where we can think deeply about structure and clarity before diving into the syntax programming language. A minute spent perfecting cohesion here saves an hour of debugging later.