CAPOL207: Basics of Software Engineering

W11L1: Modular/Component Design Concepts

Dr. A. Joy Christy
School of Computing
SASTRA Deemed to be University

Outline

- Component Definition
- Designing Class-based Components
- Conducting Component-Level Design
- Designing Traditional Components

What is a Component?

- is a modular building block for computer software
- UML defines as
  - a modular, deployable, and replaceable part of a system
  - encapsulates implementation and exposes a set of interfaces
- reside within the software architecture
- differ depending on the point of view
  - Object-oriented view
  - Traditional view
  - Process-related view

Object-oriented view

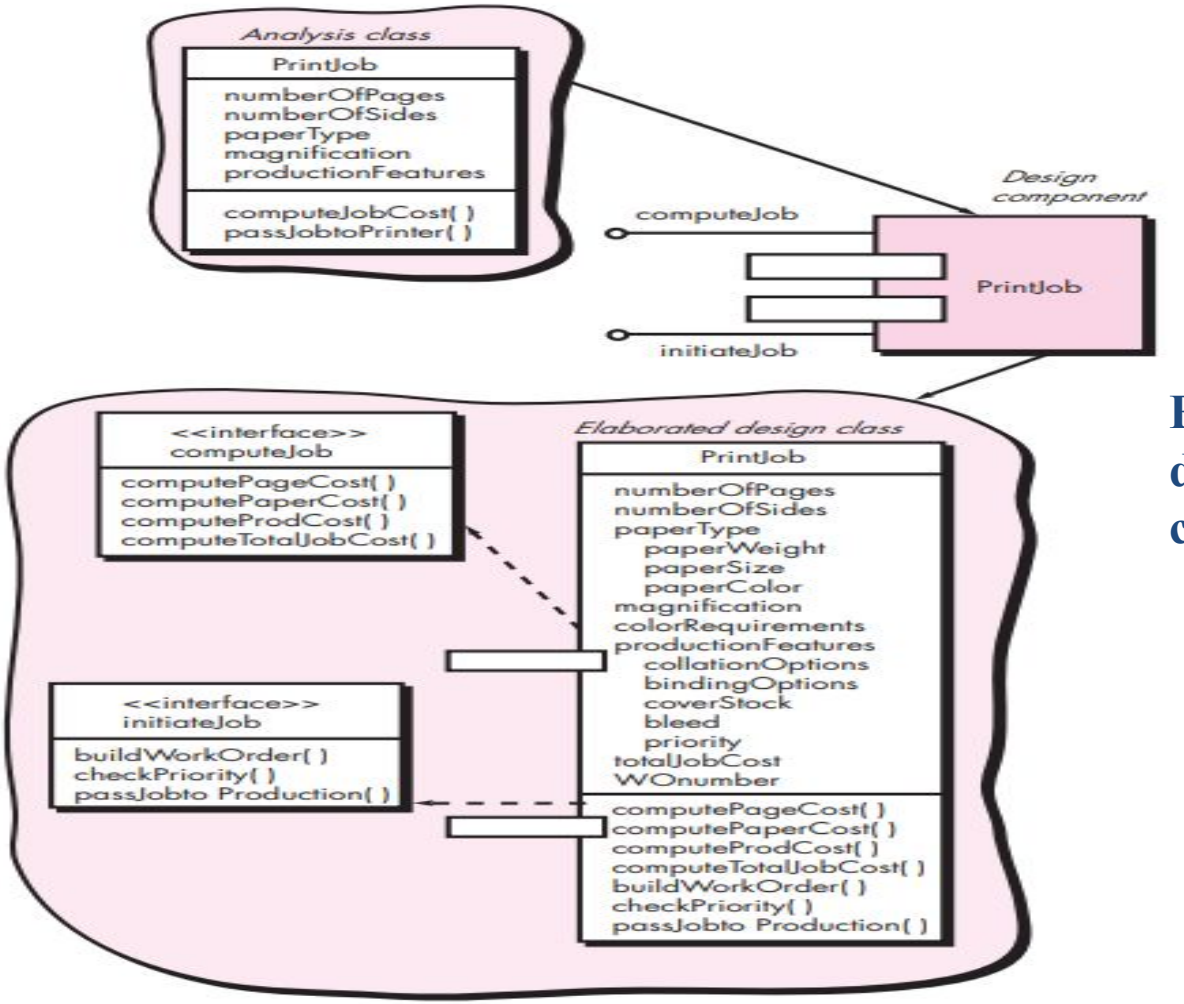
- set of collaborating classes
- each class has been fully elaborated with relevant attributes and operations
- all interfaces that enable the classes to communicate and collaborate with other classes must also be defined

- begin with analysis model and elaborate
  - analysis classes
  - infrastructure classes

Object-oriented view

- To illustrate the process of design elaboration
  - Consider software to be built for a sophisticated print shop
  - Intent of the software is to
    - Collect customer's printing requirements
    - Cost a print job
    - Pass the job on to an automated production facility

Source: Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.



Elaboration of a design component

Object-oriented view

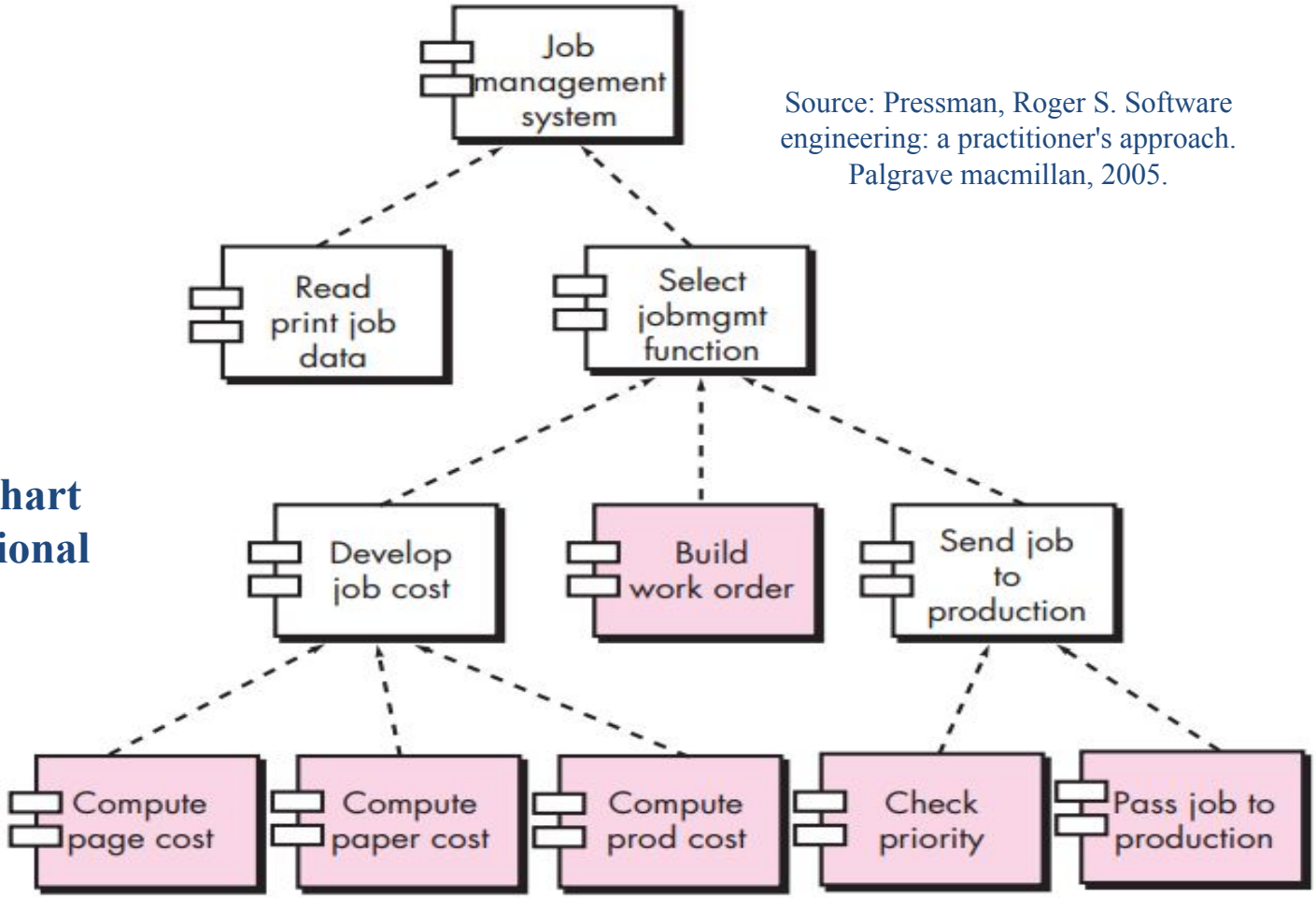
- The elaborated design class should contain
  - More detailed attribute information
    - Data structure appropriate for each attribute
  - Expanded description of operations to implement the component
    - Algorithmic detail to implement process logic

The traditional view

- A component is a functional element of a program
- Incorporates
  - Processing logic
  - Internal data structures
  - Interface to invoke the component and data to be passed to it

- A traditional component also called as module serves as one of the 3 roles
  - Control component: co-ordinates the invocation of problem domain components
  - Problem Domain component: complete or partial function required by customer
  - Infrastructure component: functions that support the processing of PD

Structure chart for a traditional system

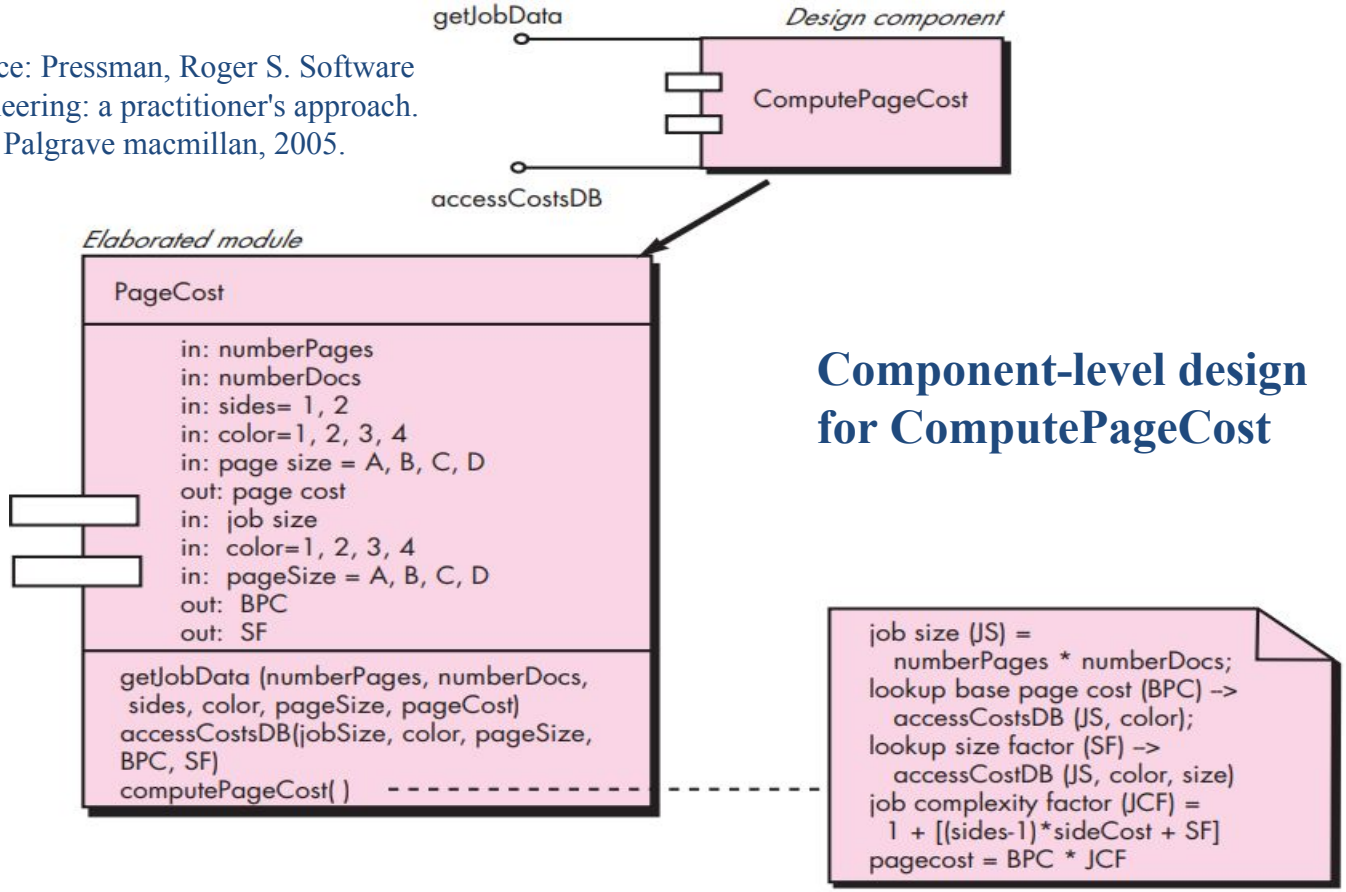


Source: Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.

The traditional view

- control components resides near the top of hierarchy
- Problem domain components reside toward the bottom of hierarchy

Source: Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.



Component-level design for ComputePageCost

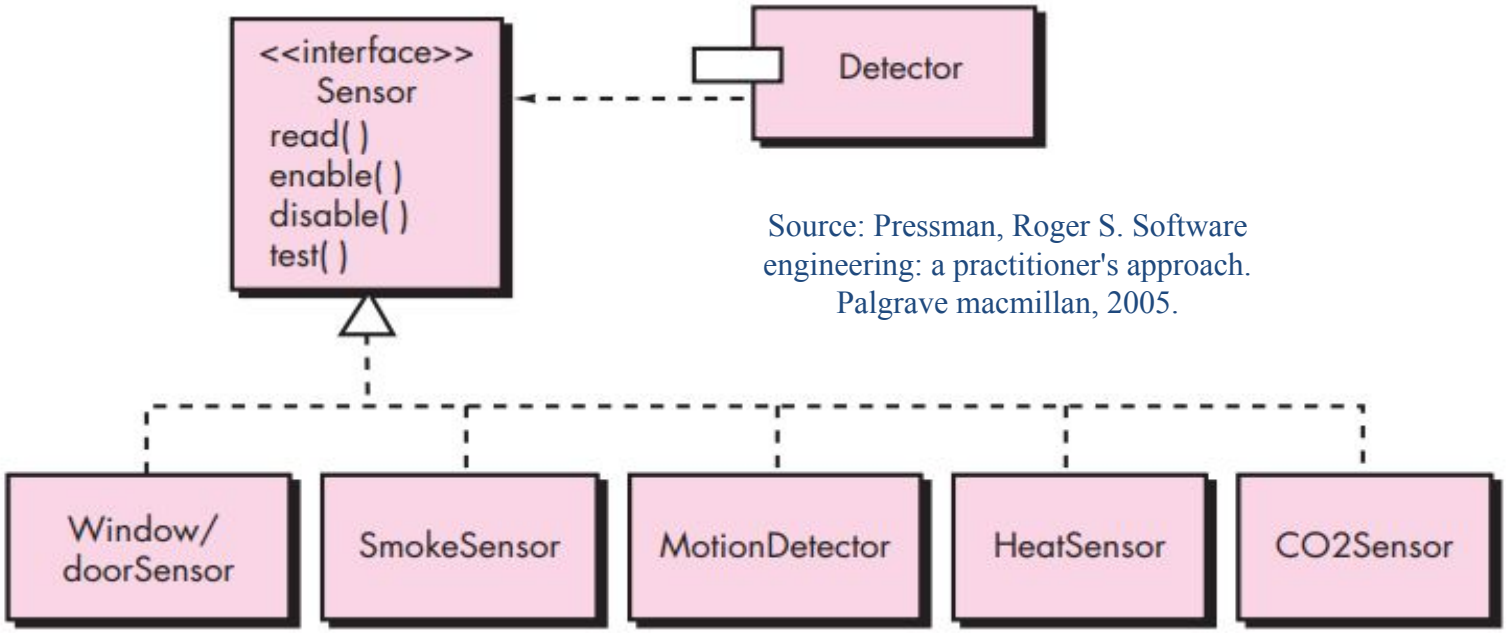
A process-related view

- Components have been created with reusability
- a catalog of proven design or code-level components is made available
- choose components or design patterns from the catalog

Designing Class-based Components

Basic Design Principles

- Open-Closed Principle (OCP)
  - *A module [component] should be open for extension but closed for modification*



Source: Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.

Following the OCP

- The Liskov Substitution Principle (LSP)

  "Subclasses should be substitutable for their base classes"

  - Components that uses a base class should continue to function with derived class
  - derived classes must honor the contract between the base class and components

- Dependency Inversion Principle (DIP)
  - *Depend on abstractions. Do not depend on concretions*
  - abstractions are the place where a design can be extended
  - Reduces complications

- The Interface Segregation Principle (LSP)

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

  - multiple client components use the operations provided by a server class
  - create specialized interface to serve each major category of clients

- The Release Reuse Equivalency Principle (REP)

  "The granule of reuse is the granule of release"

  - group reusable classes into packages
  - manage and control as newer versions evolve

- The Common Closure Principle (CCP)

  ""Classes that change together belong together"

  - Classes should be packaged cohesively

- The Common Reuse Principle (CRP)

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

  - only classes that are reused together should be included within a package

- The Common Closure Principle (CCP)

  ""*Classes that change together belong together*"

  - Classes should be packaged cohesively

- The Common Reuse Principle (CRP)

  ""*Classes that aren't reused together should not be grouped together*"

  - only classes that are reused together should be included within a package

Component-Level Design Guidelines

Components

- Establish proper naming conventions
- Architectural component names should be drawn from problem domain

Ex:

FloorPlan

- Use stereotypes to help identify the nature of components

Ex:

<<infrastructure>> <<database>> <<table>>

Interfaces

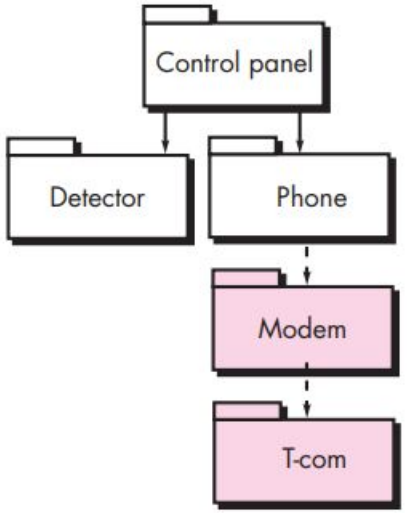
- provide important information about communication and collaboration
  - lollipop representation of an interface should be used
  - Should flow from left-hand side of the component
  - Interfaces that are relevant to the component should be shown

Dependencies and Inheritance

- To improve readability, model
  - dependencies from left to right
  - Inheritance from bottom to top

Cohesion

- 'Single-mindedness' of a component
- encapsulates only attributes and operations that are closely related to one another
- Different types of cohesion
  - Functional:
    - module performs one and only computation and returns a result
  - Layer:
    - Higher layer accesses the service of a lower layer
  - Communicational:
    - Operations using the same input data or output data



Coupling

- is a qualitative measure of the degree to which classes are connected to one another
- As classes (and components) become more interdependent, coupling increases
- the amount of communication and collaboration increases system complexity
  - Implementation
  - Testing
  - Maintenance
- designer should work to reduce coupling whenever possible

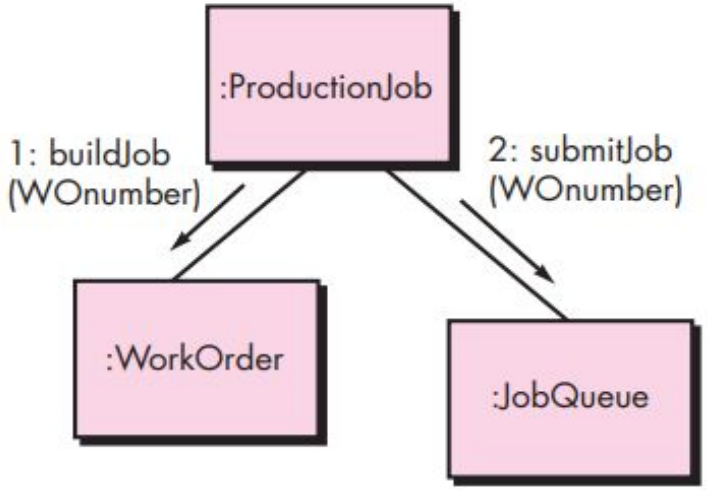
- Content coupling
  - One component modifies data internal to another component
- Common coupling
  - number of components make use of global variable
- Control coupling
  - controls the flow of another, by passing info on what to do
  - Operation A() invokes operation B()
- Stamp coupling
  - classA is declared as a type for an argument of an operation of classB

- Data coupling
  - Occurs when components share the data through
- Routine call coupling
  - Occurs when one operation invokes another
- External coupling
  - Occurs when two components communicate or collaborate with infrastructure components

Conducting Component-Level Design

1. Identify all design classes that correspond to the problem domain
   - Using requirements and architectural model
2. Identify all design classes that correspond to the infrastructure domain
   - GUI components
   - operating system components
   - object and data management components
3. Elaborate all design classes that are not acquired as reusable components
   - interfaces, attributes, and operations

3. a). Specify message details when classes or components collaborate



Collaboration diagram with messaging

Source: Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.

- As design proceeds
  - Each message is elaborated by expanding its syntax

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

3. b). Identify appropriate interfaces for each component

3. c). Elaborate attributes and define data types and data structures

name : type-expression = initial-value {property string}

paperType-weight: string = "A" { contains 1 of 4 values – A, B, C, or D}

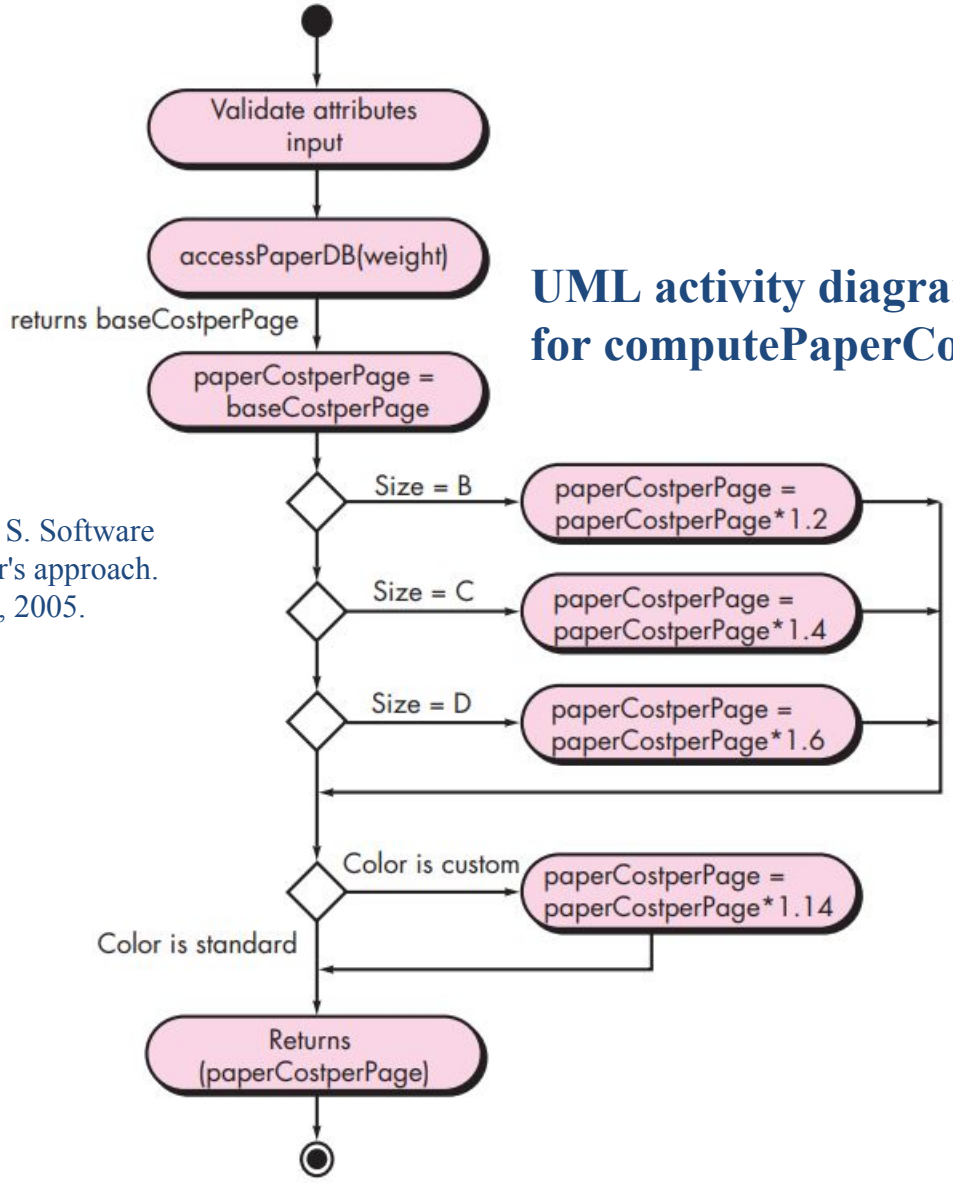
3. d). Describe processing flow within each operation in detail

- use a
  - programming language-based pseudocode or UML activity diagram

- Each software component is elaborated through a number of iterations
  - The first iteration defines each operation as part of the design class
  - The next iteration does little more than expand the operation name

```
computePaperCost (weight, size, color): numeric
```

  - If the algorithm to implement computePaperCost() is simple
    - No further design elaboration is necessary
  - if the algorithm is more complex
    - further design elaboration is required at this stage



UML activity diagram for computePaperCost()

Source: Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.

4. Describe persistent data sources (databases and files) and identify the classes required to manage them

   - Databases and files is described as an individual component

5. Develop and elaborate behavioral representations for a class or component

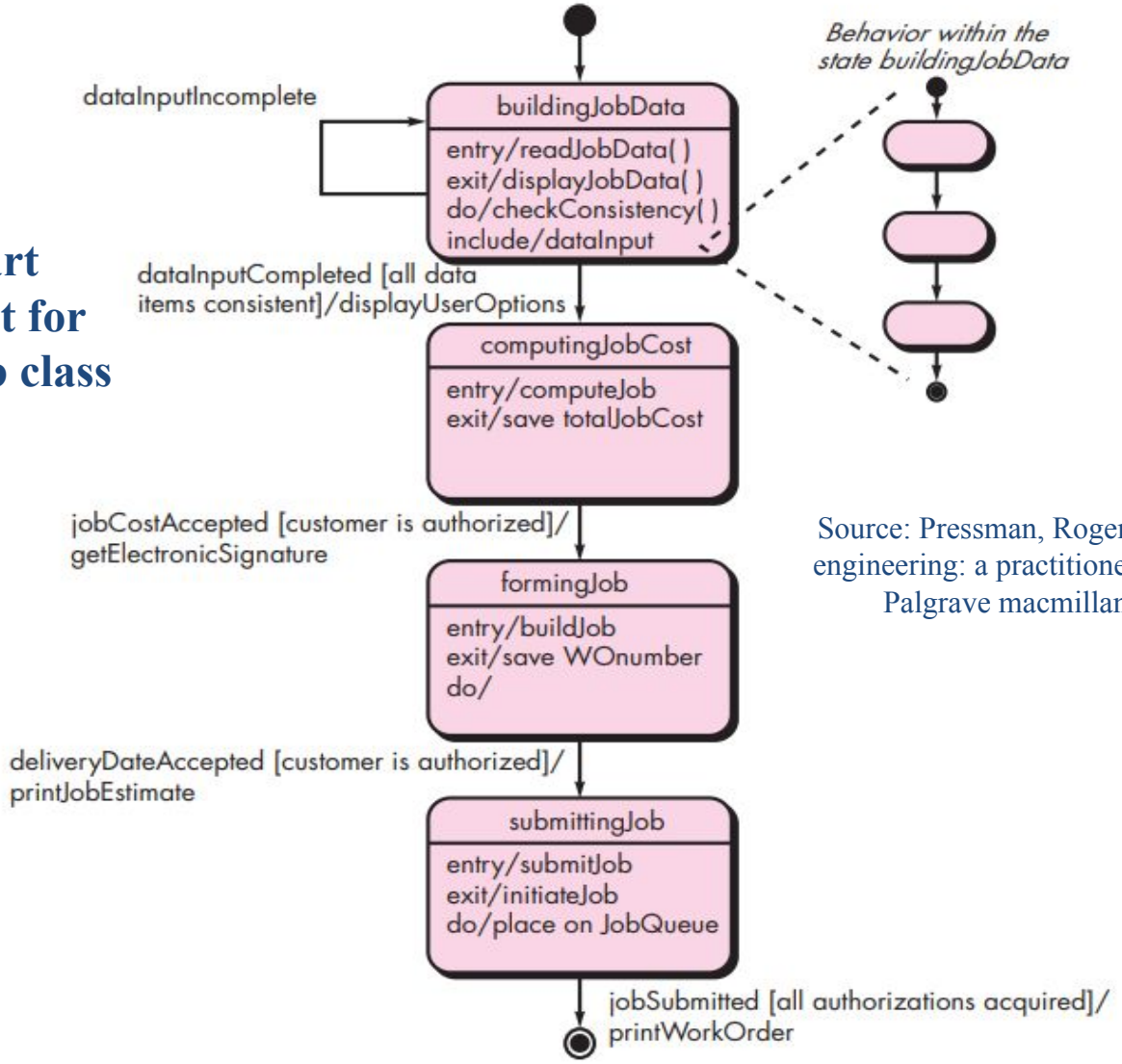
   - To understand the dynamic behavior of an object
     - examine all use cases that are relevant to the design class

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

6. Elaborate deployment diagrams to provide additional implementation Detail

7. Refactor every component-level design representation and always consider alternatives

Statechart fragment for PrintJob class



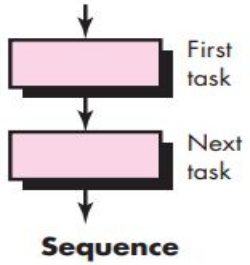
Source: Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.

Designing Traditional Components

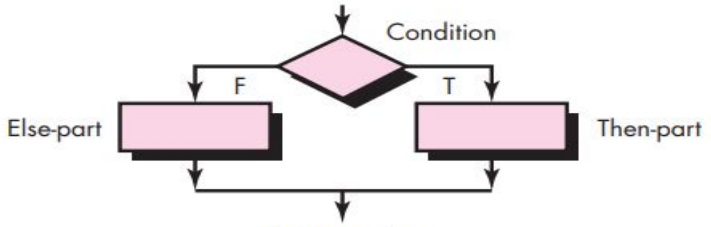
Graphical Design Notation

- Flowchart provides useful pictorial patterns for depicting procedural detail

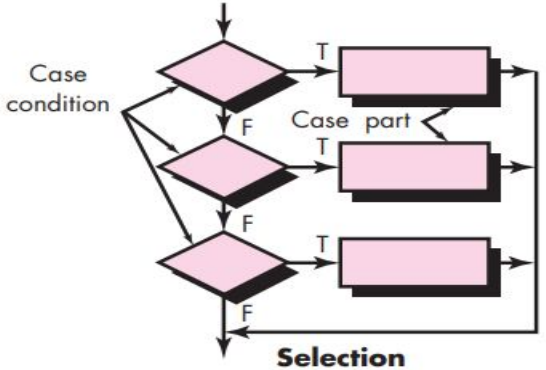
Source: Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.



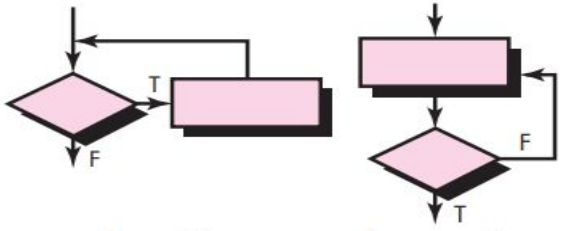
Sequence



If-then-else



Selection



Repetition

Flowchart constructs

Tabular Design Notation

- Decision Table provides a notation that translates actions and conditions in a tabular form

Source: Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.

	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
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional *x* percent discount		✓		✓		✓

Decision table nomenclature

Program Design Language

- Pseudo code
  - Free-form expressive ability of a natural language

Source: Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.

```
component alarmManagement;
The intent of this component is to manage control panel switches and input from sensors by
type and to act on any alarm condition that is encountered.
    set default values for systemStatus (returned value), all data items
    initialize all system ports and reset all hardware
    check controlPanelSwitches (cps)
        if cps = "test" then invoke alarm set to "on"
        if cps = "alarmOff" then invoke alarm set to "off"
        if cps = "newBoundingValue" then invoke keyboardInput
        if cps = "burglarAlarmOff" invoke deactivateAlarm;
        •
        •
        •
```

Pseudo Code

THANK YOU

Reference:

Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave macmillan, 2005.