SOFTWARE ENGINEERING

CHAPTER-10
COMPONENT-LEVEL DESIGN

Software School, Fudan University
Spring Semester, 2016

Software Engineering: A Practitioner's Approach, 7th edition
Originated by Roger S. Pressman

Component-Level Design

- Architectural Design
  - A complete set of software components is defined
  - But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code

- Component-level design
  - defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each component

What is Component

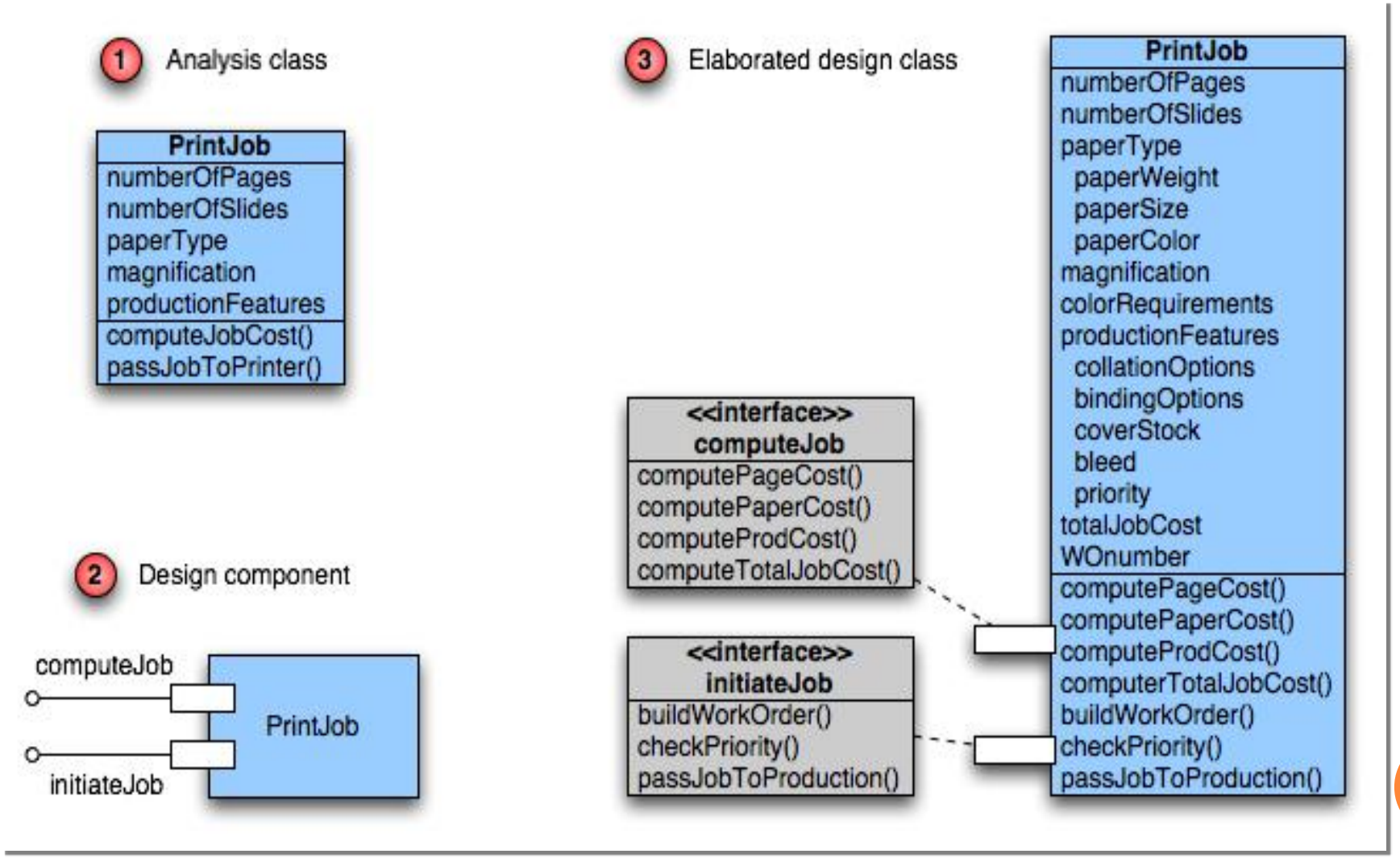
"A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

— *OMG UML Specification*

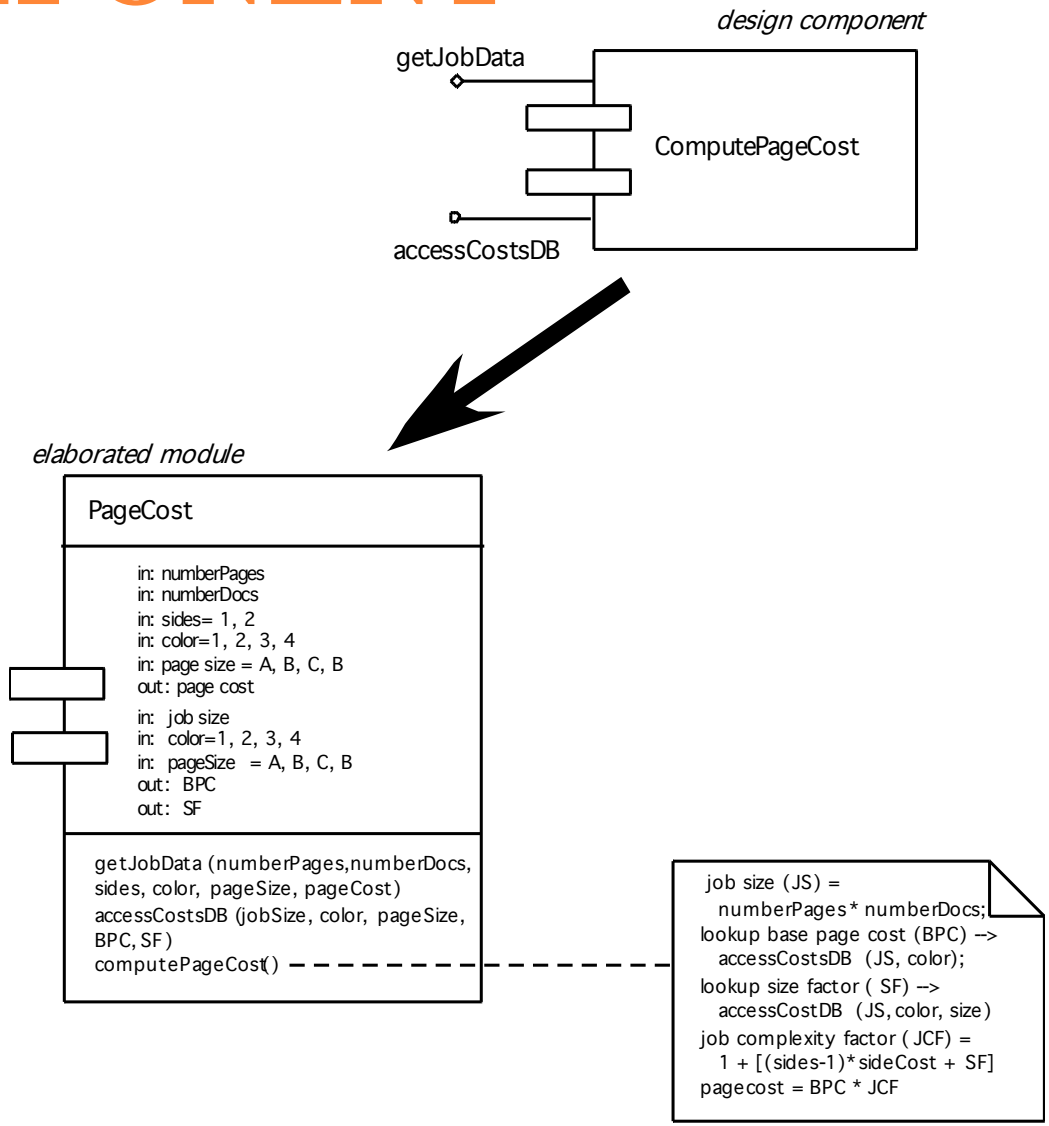
Component Views

- OO View: A component is a set of collaborating classes (class cluster)
- Conventional View
  - a functional element of a program that incorporates
    - **processing logic**
    - the **internal data structures** required to implement the processing logic
    - an **interface** that enables the component to be invoked and data to be passed to it

Class Elaboration



Conventional Component



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
- 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).

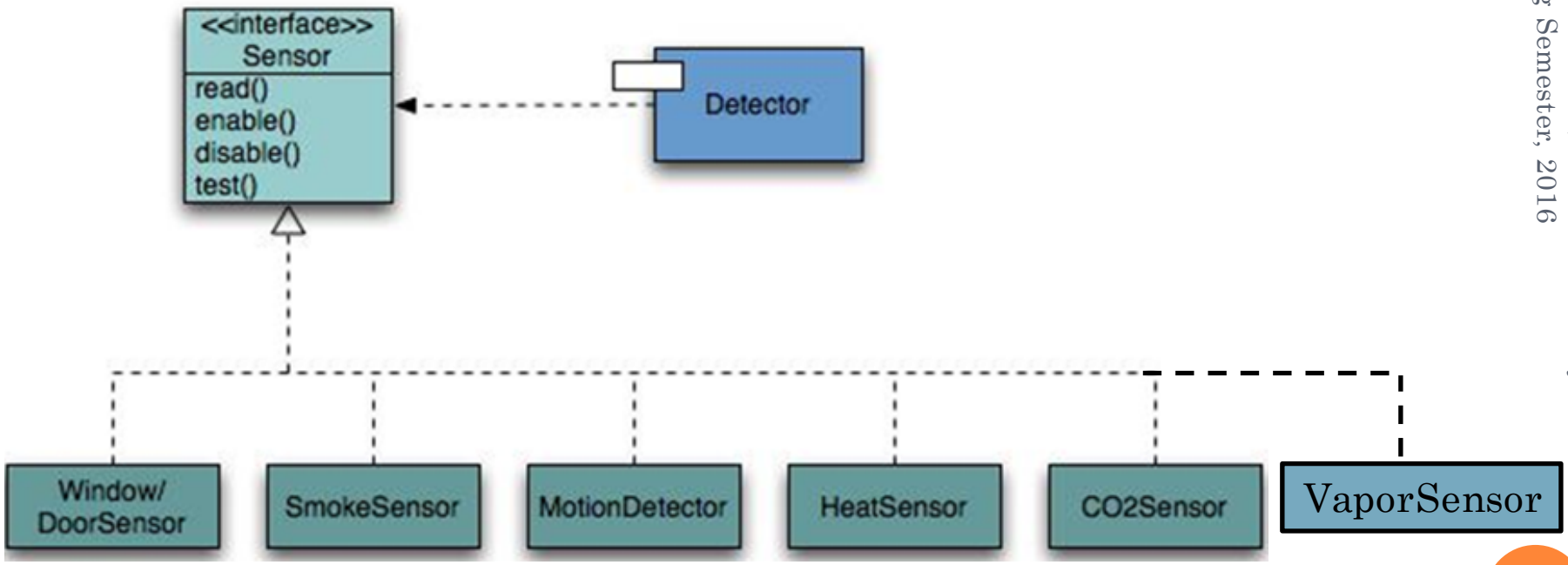
OO Design Principles

- 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."*

Design by Contract !!!

Open-Closed Principle

A module should be ***open for extension*** but ***closed for modification***.



SUBSTITUTABILITY

Subclasses should be substitutable for base classes

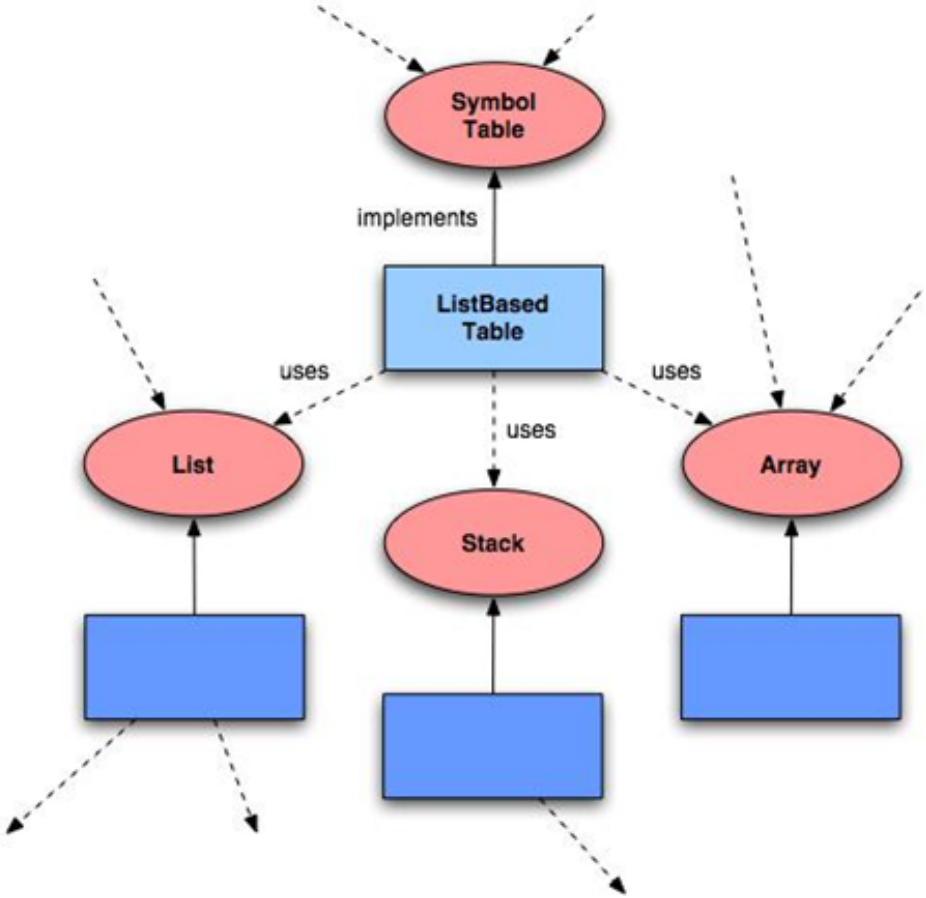
Is a circle a kind of ellipse?

public void setSize(int x, int y);
requires nothing
ensures after the call, the ellipse is
x units wide and y units high

public void setSize(int x, int y);
requires x = y
ensures after the call, the ellipse is
x units wide and y units high

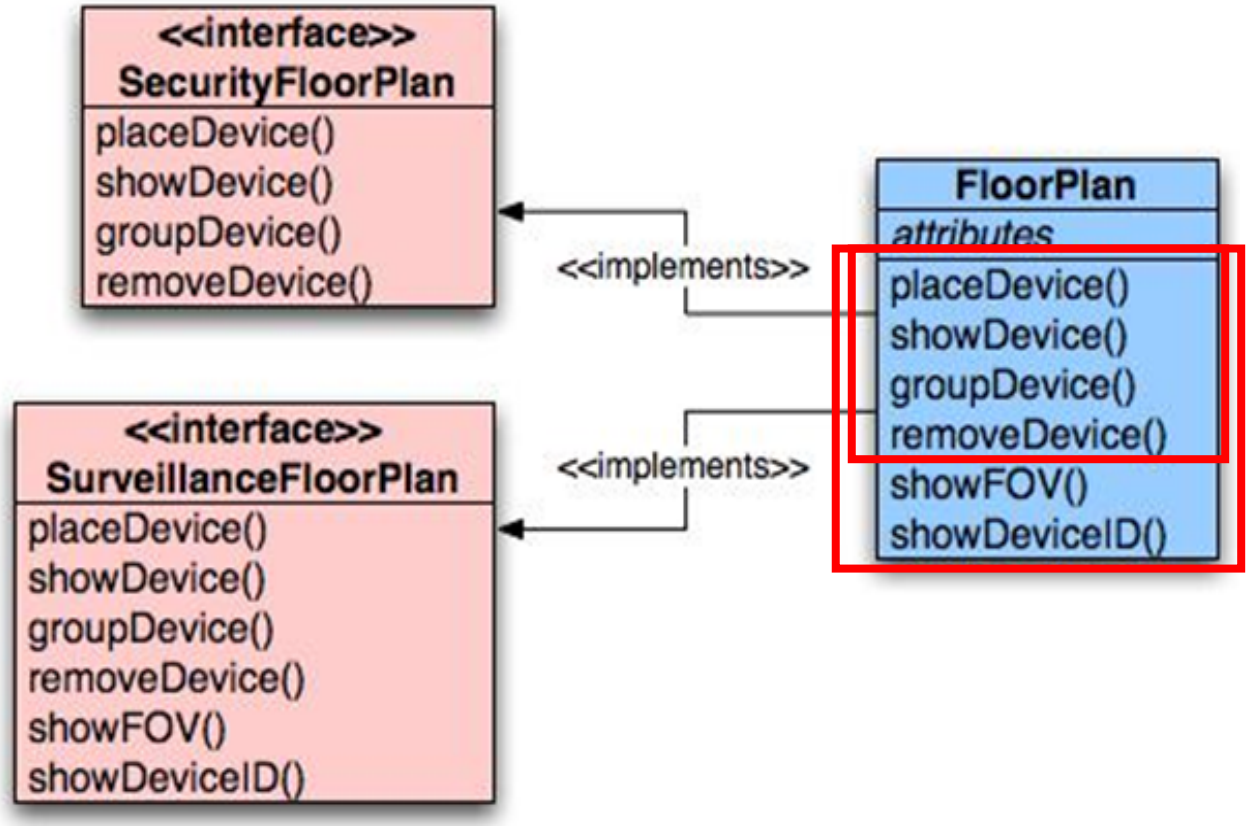
Dependency Inversion

Depend on abstractions (abstract class, interface…)
Do not depend on concretions.



Interface Segregation

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



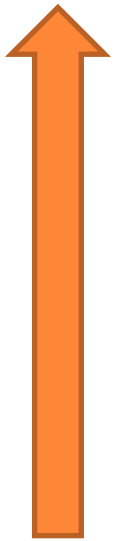
Design by Contract

- Contract
  - The relationship between a class and its clients can be viewed as a *formal agreement*, expressing each party's rights and obligations
- Consider the following operation in a List

  public Item remove(int index)
  - requires the specified index is in range (0 ≤ index < size( ))
  - ensures the element at the specified position in this list is removed, subsequent elements are shifted to the left (1 is subtracted from their indices), and the element that was removed is returned

Cohesion

- The "single-mindedness" of a module
- Cohesion implies that a single component or class encapsulates ***only*** attributes and operations that are ***closely related*** to one another and to the class or component itself.
- Examples of cohesion
  - Functional
  - Layer
  - Communicational
  - Sequential
  - Procedural
  - Temporal
  - Utility

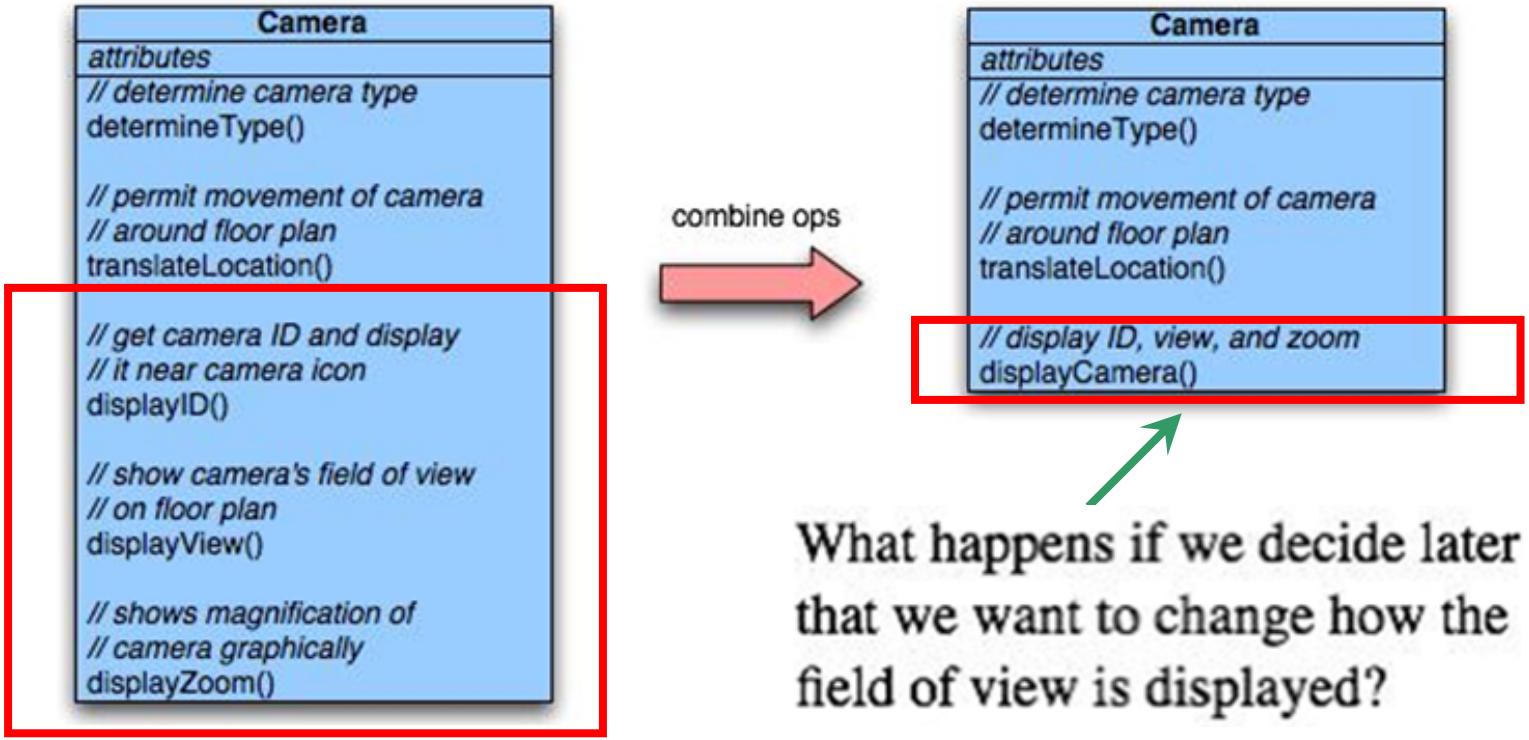


High

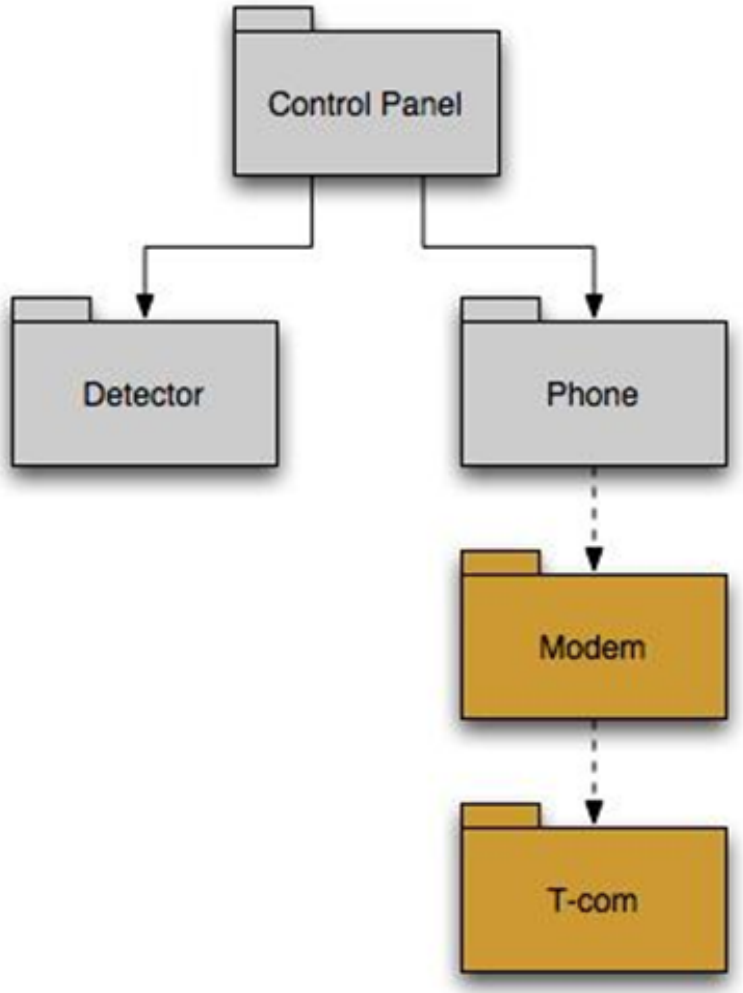
Low

Functional Cohesion

- Typically applies to operations
  - Occurs when a module performs one and only one computation and then returns a result



Layer Cohesion



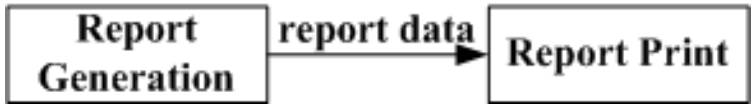
- Applies to packages, components, and classes
  - Occurs when a higher layer can access a lower layer, but lower layers do ***not*** access higher layers

Communicational Cohesion

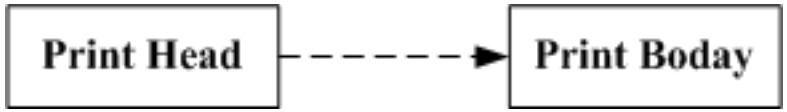
- All operations that access the same data are defined within one class
- In general, such classes focus solely on the data in question, accessing and storing it
- Example
  - A ***StudentRecord*** class that adds, removes, updates, and accesses various fields of a student record for client components.

Other Cohesion Types

- Sequential

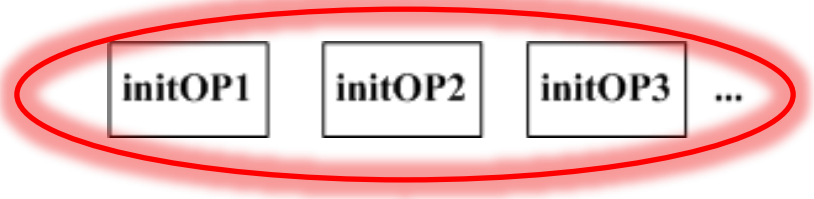


- Procedural



- Temporal

a set of operations performed in initiation



- Utility

Random	An instance of this class is used to generate a stream of pseudorandom numbers.
ResourceBundle	Resource bundles contain locale-specific objects.
SimpleTimeZone	SimpleTimeZone is a concrete subclass of TimeZone that represents a time zone for use
Stack	The Stack class represents a last-in-first-out (LIFO) stack of objects.
StringTokenizer	The string tokenizer class allows an application to break a string into tokens.
Timer	A facility for threads to schedule tasks for future execution in a background thre
TimerTask	A task that can be scheduled for one-time or repeated execution by a Timer.
TimeZone	TimeZone represents a time zone offset, and also figures out daylight savings.
TreeMap	Red-Black tree based implementation of the SortedMap interface.
TreeSet	This class implements the Set interface, backed by a TreeMap instance.
Vector	The Vector class implements a growable array of objects.
WeakHashMap	A hashtable-based Map implementation with *weak keys*.

classes in java.util package

Coupling

- A qualitative measure of the degree to which classes or components are connected to each other
- Avoid
  - Content coupling
- Caution
  - Common coupling
  - Control coupling
- Be aware
  - Routine call coupling
  - Type use coupling
  - Inclusion or import coupling

Avoid

Content Coupling

- Occurs when a compon
  "surreptitiously" modif
  data that is internal to
  another component"
- Violates information
  hiding

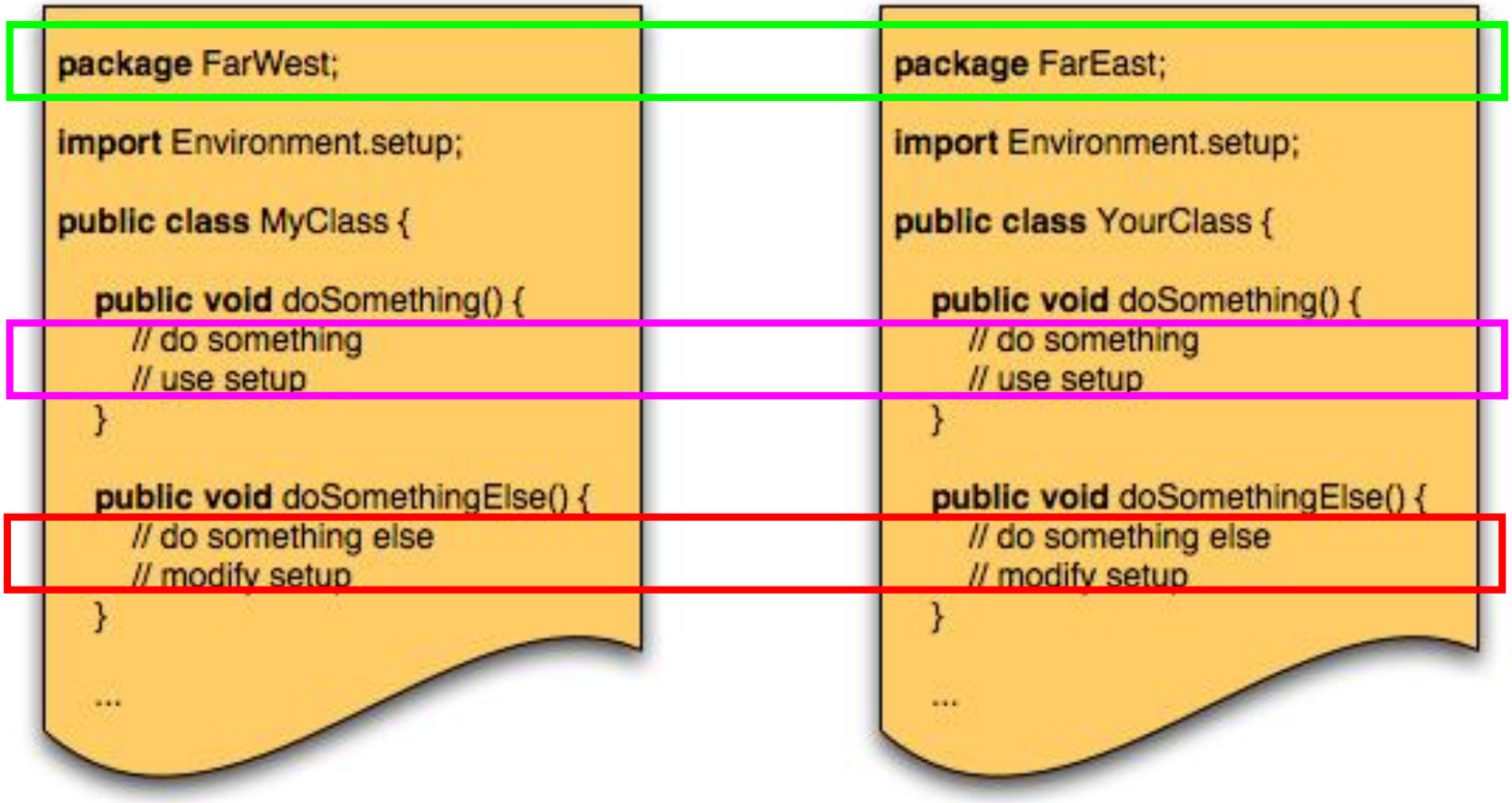
```
public class StudentRecord {

    private String name;
    private int[ ] quizScores;

    public String getName(){
        return name;
    }
    public int getQuizScore(int n) {
        return quizScores[n];
    }
    public int[ ] getAllQuizScores(){
        return quizScores;
    }

    ....
```

??

Caution

Common Coupling

- Occurs when a number of components all make use of a ***global variable***



Caution

CONTROL COUPLING

- Occurs when operation ***A*** invokes another operation ***B*** and passes a control flag that directs logical flow in ***B***

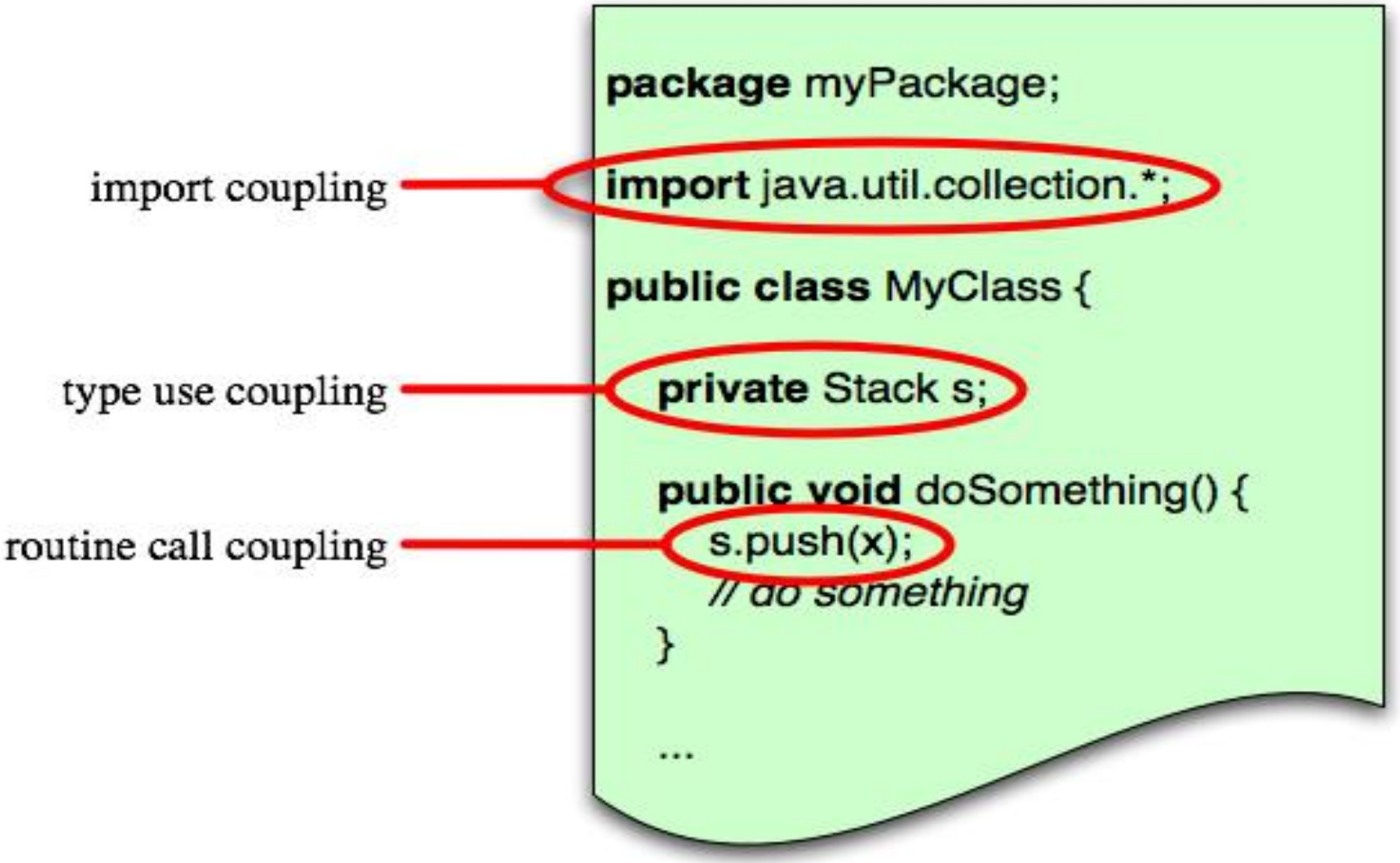
```
public void studentRegister(Student stu) {
    …
    if (stu.isGraduate==true) // graduate
       …

    else  // undergraduate student
       …
}
….
```

What happens if the client is extended to handle registration of the third kinds of students?

Be aware

Routine Coupling

- Certain types of coupling occur routinely in object-oriented programming.



Component-Level Design

Component-level design is elaborative in nature

should provide sufficient detail for construction activity

General Steps

1. Identify design classes in problem domain (using analysis model and architecture)
2. Identify infrastructure design classes
3. Elaborate all design classes that are not acquired as reusable components
4. Describe persistent data sources
5. Elaborate behavioral representations
6. Elaborate deployment diagrams
7. Refactor design and consider alternatives

Steps 1 & 2 – Identify Classes

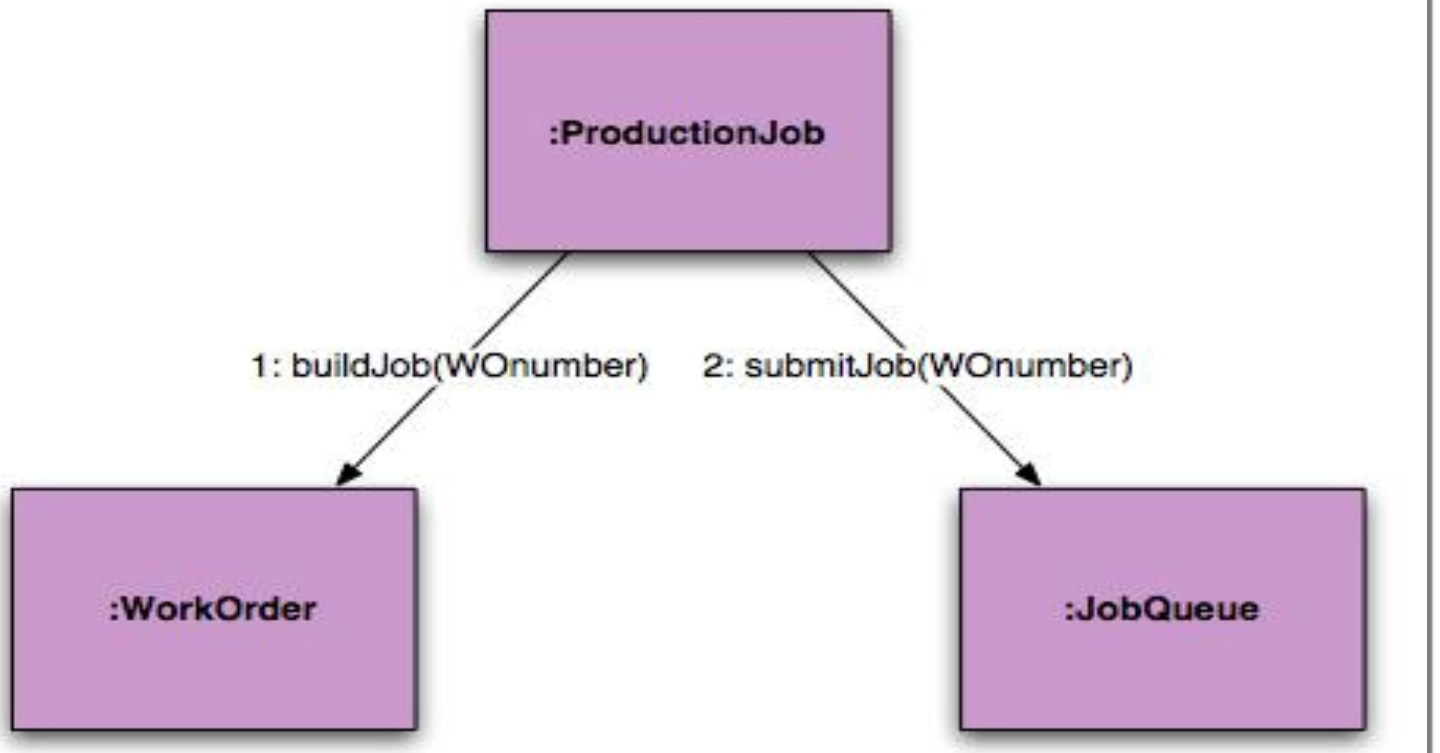
- Most classes from the **problem domain** are analysis classes created as part of the analysis model
  - but there could be *some* classes found in designing
- The **infrastructure** design classes are introduced as components during architectural design

Step 3 – Class Elaboration

- Specify ***message details*** when classes or components collaborate
- Identify appropriate ***interfaces*** for each component
- Elaborate ***attributes*** and define ***data structures*** required to implement them
- Describe ***processing flow*** within each operation in detail

3A. Collaboration Details

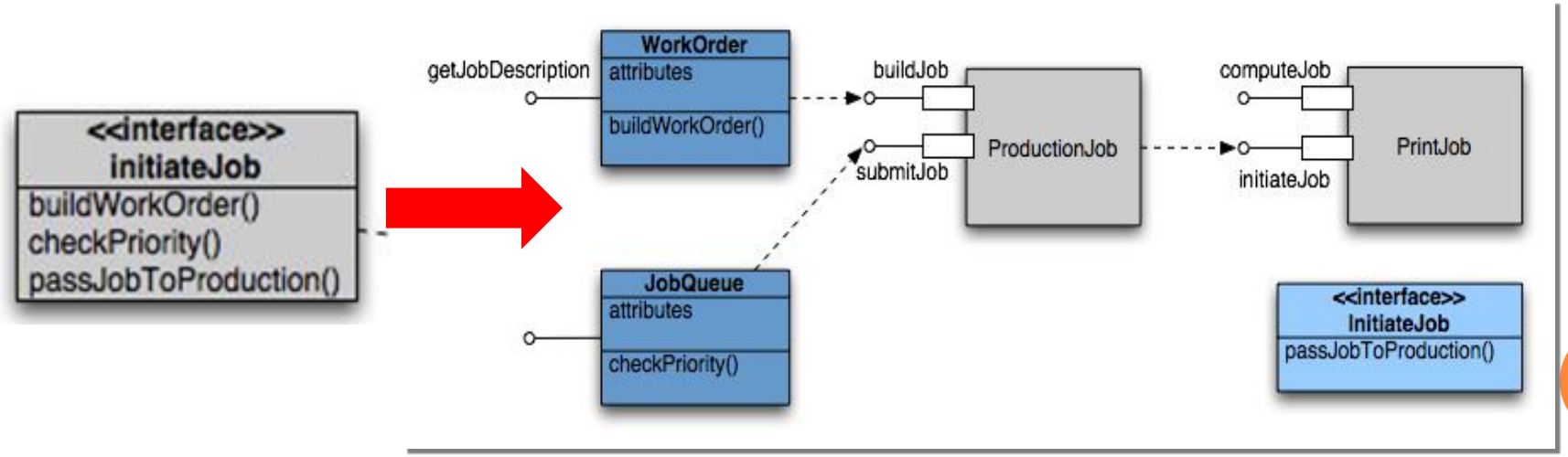
- Messages can be elaborated by expanding their syntax in the following manner:
  - [guard condition] sequence expression (return value) : message name (argument list)



3B. Appropriate Interfaces

- The PrintJob interface "initiateJob" does not exhibit sufficient cohesion because it performs three different subfunctions

- The following refactoring is suggested



3C. ELABORATE ATTRIBUTES

- Analysis classes will typically only list names of *general attributes* (ex. paperType)
- List all attributes during component design
  - UML syntax:
    - name: type-expression = initial-value {property string}
  - For example, *paperType* can be broken into *weight*, *size*, and *color*.
    - The *size* attribute would be:

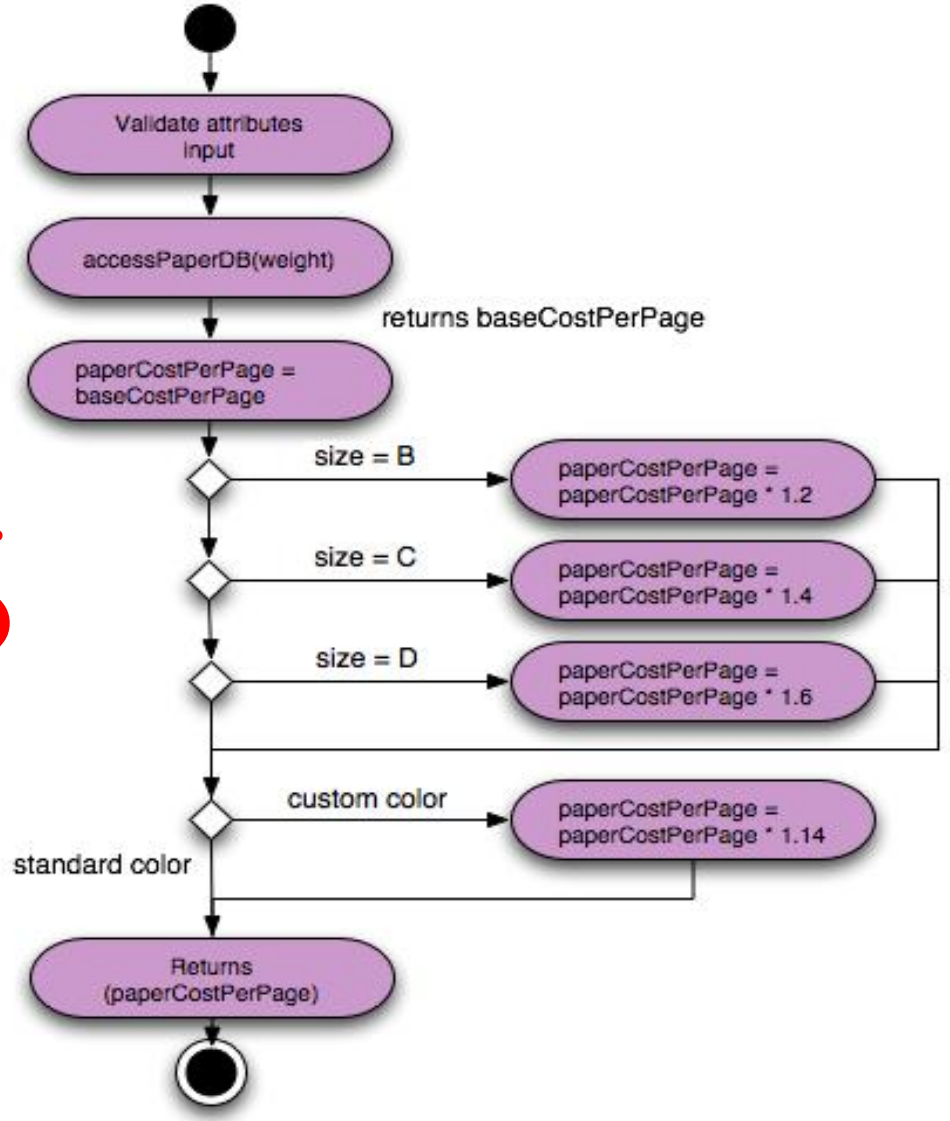
paperType_size: string =

"A" { contains 1 of 4 values – A, B, C, or D }

If an attributes appears repeatedly across a number of design classes and has a Relatively complex structure, it is best to create a separate class for it.

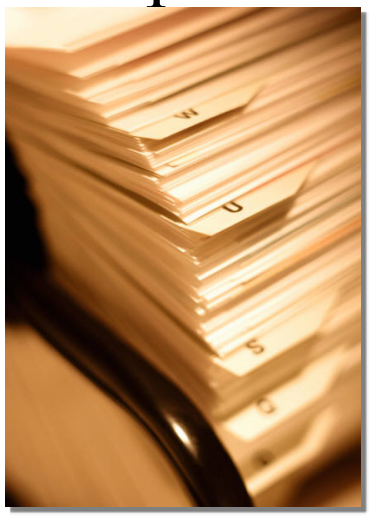
3D. Describe Processing Flow

Activity diagram for *computePaperCost( )*



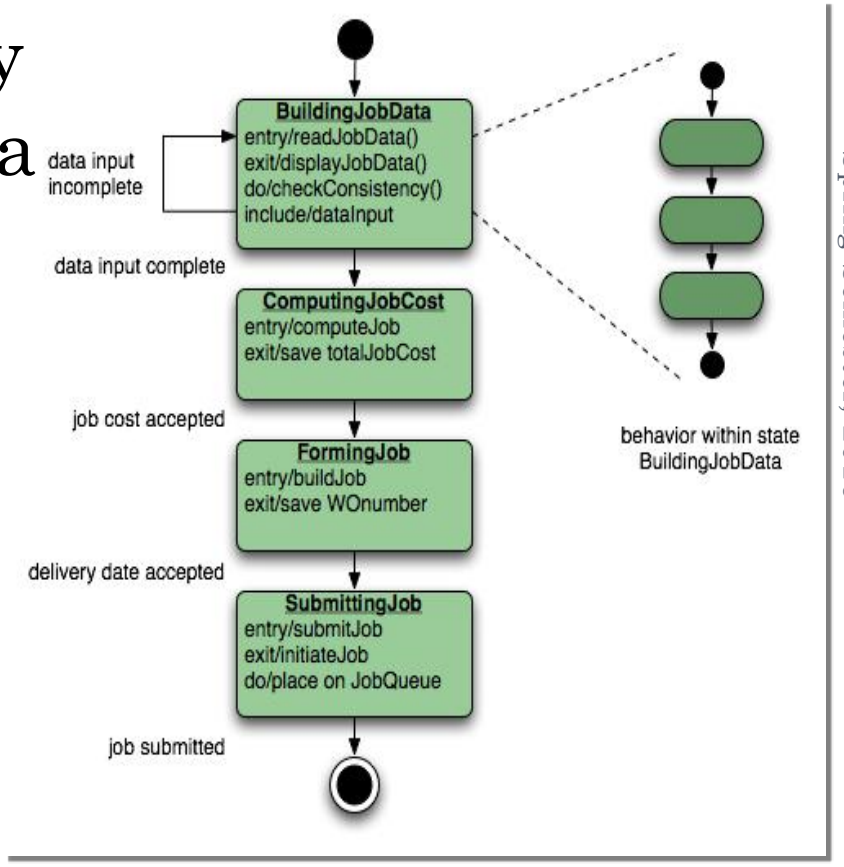
Step 4 – Persistent Data

- Usually persistent data stores are initially specified in architectural design
- It is useful to provide additional detail in component-level design
- Describe persistent data sources (databases and files) and identify the classes required to manage them
  - Classes dedicated to
    - DB operations
    - File access



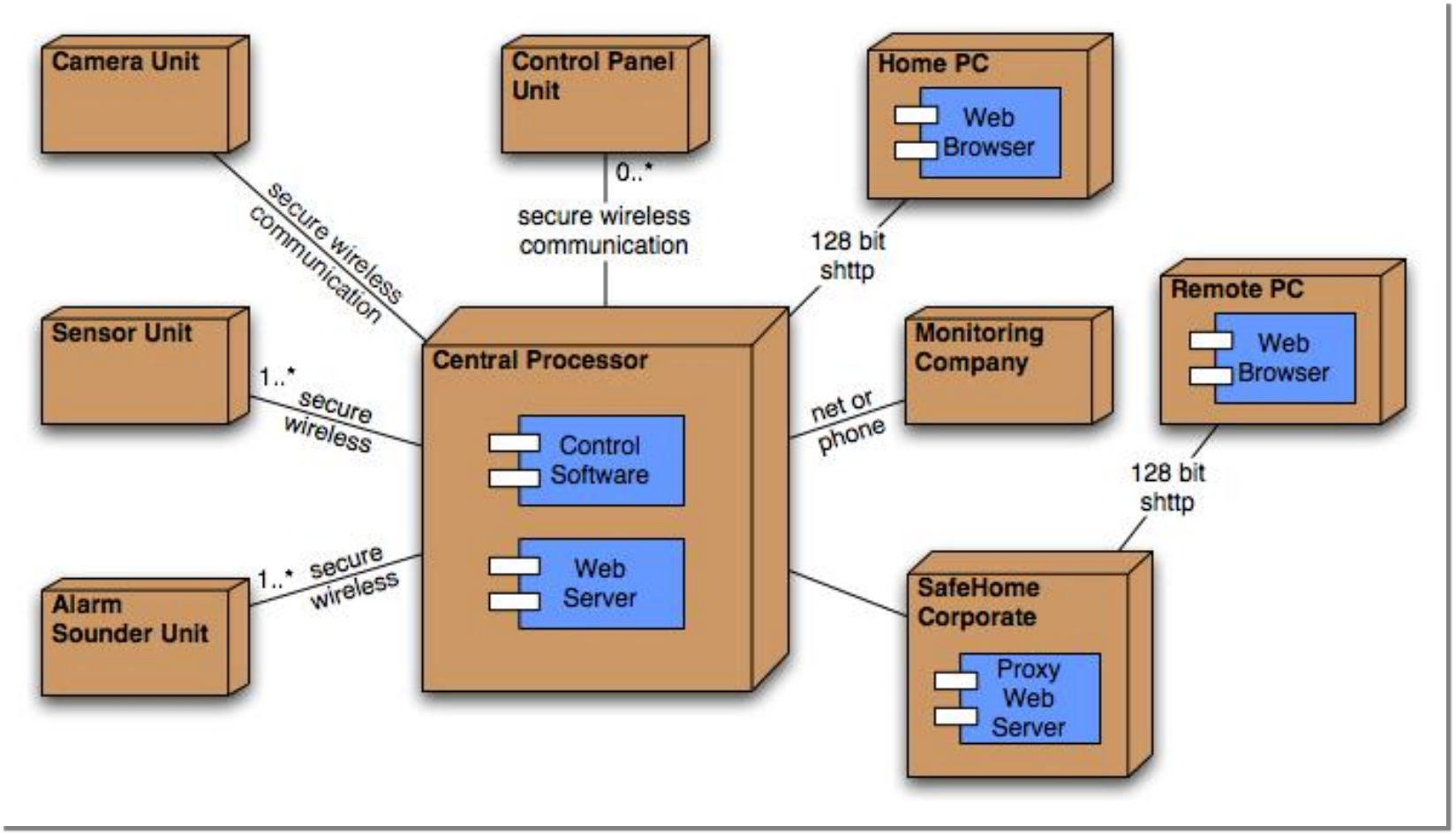
Step 5 – Elaborate Behavior

- It is sometimes necessary to model the behavior of a design class
- Transitions from state to state have the form:
  - Event-name (parameter-list) [guard-condition] / action expression
  - Sometimes simplified



Step 6 – Elab. Deployment

- Deployment diagrams are elaborated to represent the location of key packages or components



Step 7 – Redesign/Reconsider

- The *first* component-level model you create will not be as complete, consistent, or accurate as the ***nth iteration*** you apply to the model

- The best designers will consider many *alternative design solutions* before settling on the final design model

End of Chapter 10