

CS2212

Introduction to Software Engineering

Component-Level Design

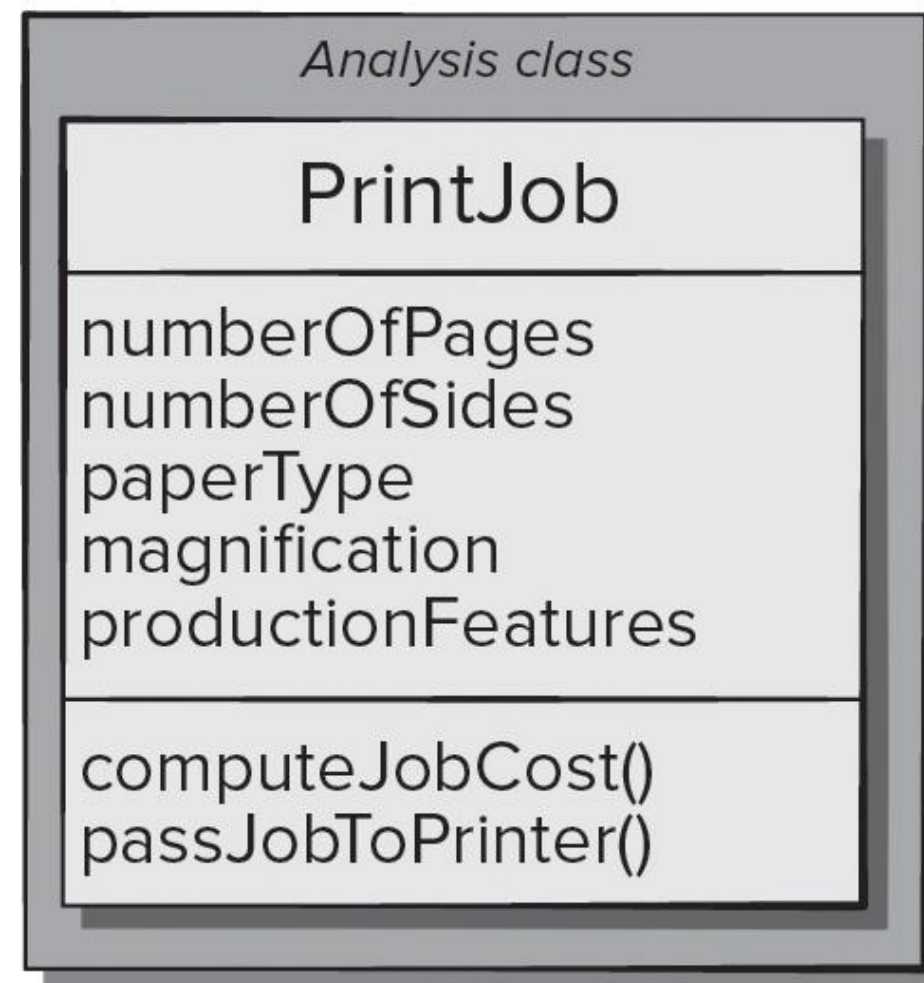
Quick Review: What is a Component?

Software Component:

- “A **modular, deployable, and replaceable** part of the system that **encapsulates implementation** and exposes a set of **interfaces**.”
- Parts of a system that **break the complexity into manageable parts**.
- **Hides (encapsulates) implementation details** behind an **interface**.
- Components can be **swapped in and out** so long as they share a common **interface**.

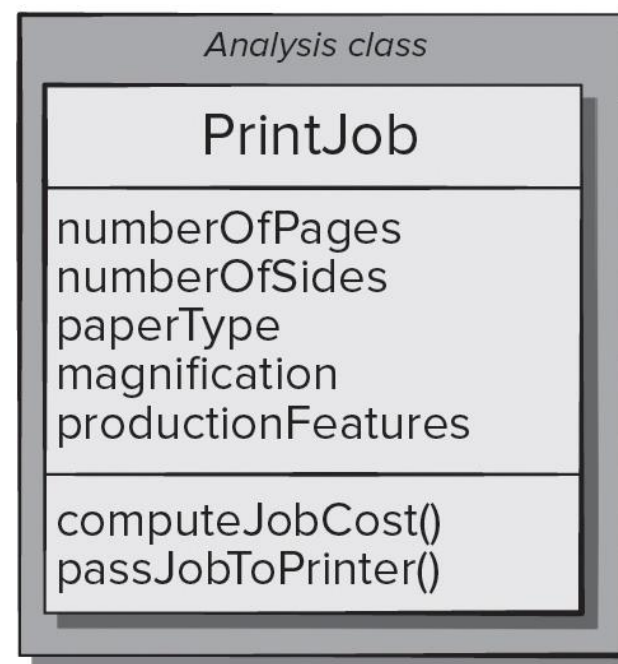
Component Elaboration

- Begin with the classes from the **analysis model**, divided into components and define **interfaces**.
- **Analysis classes** become design **components**.
- **Interfaces** identified based on **operations** in **analysis classes**.

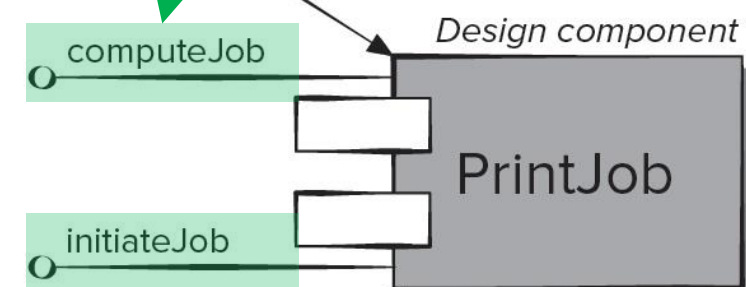


Component Elaboration

- Must now refine and elaborate on each **analysis class** to create our **design classes**.
- Detail is added for each **attribute**, **operation**, and **interface**.
- **Data structures** for each attribute are specified.
- **Algorithmic detail** for each operation is designed.
- The **mechanisms** required to implement the interface are designed

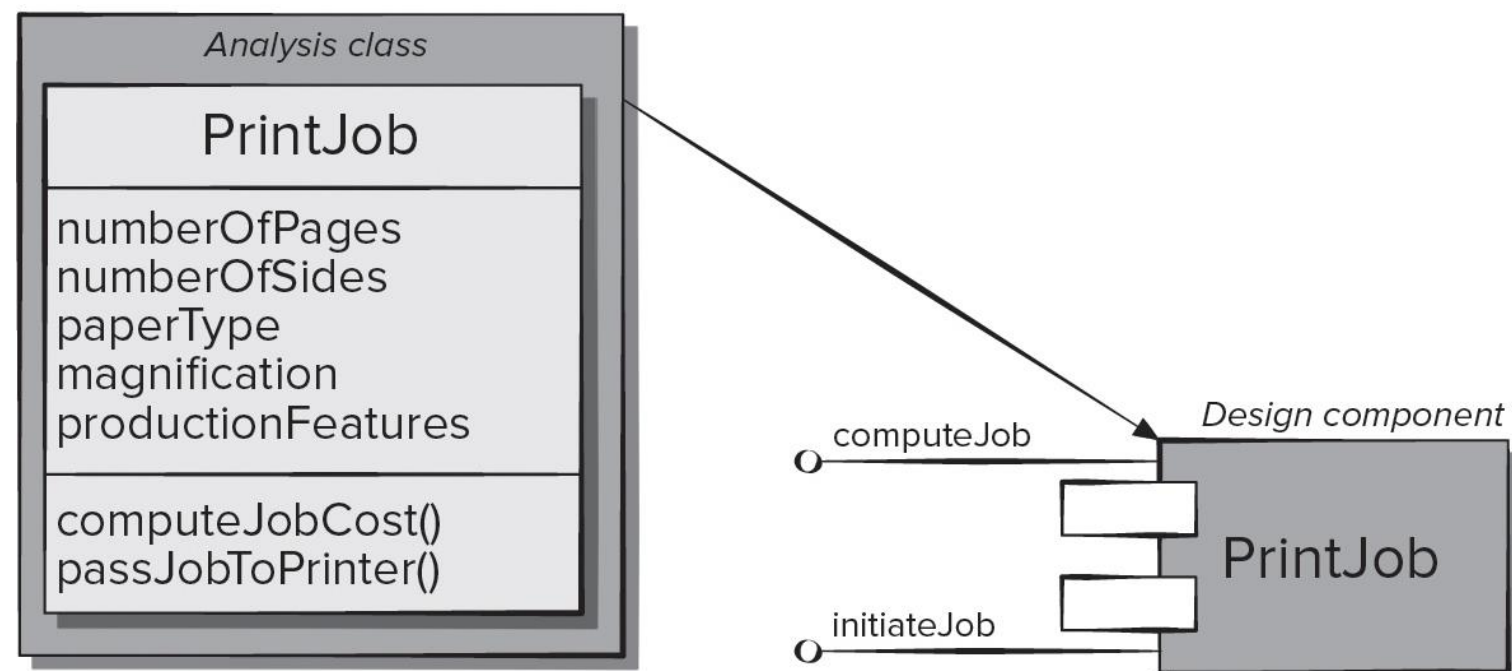


UML "lollipop" notation.
Denotes an interface.



Component Elaboration

- **Infrastructure classes** to support the **design classes** may need to be created.
- Once complete, **all detail necessary for implementation** has been generated and is available to developers.



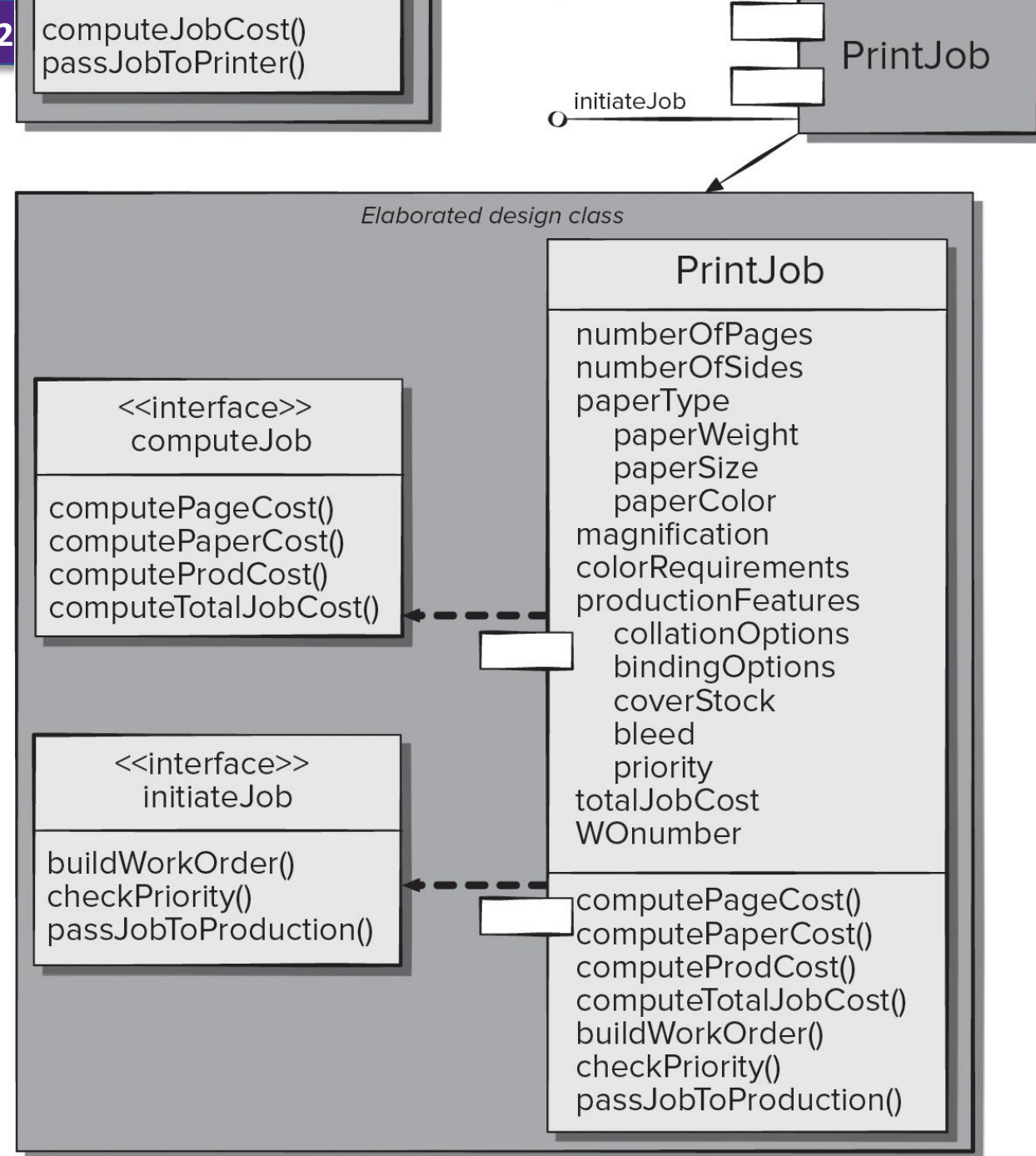
Component Elaboration

PrintJob now a Design Class

- **Interfaces** defined.
- Extra attributes that will be required added.
- Extra operations that will be required added.

Still need to do more:

- Define types and data structures.
- Define operations' algorithms using pseudocode and **stepwise refinement**.
- Consider grouping related attributes and operations into their own class.
- Elaborate on all other components in the software.



Basic Design Principles

- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- Interface Segregation Principle (ISP)
- Release Reuse Equivalency Principle (REP)
- Common Closure Principle (CCP)
- Common Reuse Principle (CRP)

Basic Design Principles

- **Open-Closed Principle (OCP)**

Open-Closed Principle (OCP)

*“A module [component] should be **open for extension** but **closed for modification**.”*

- Specify the component in a way that allows it to be extended without the need to make internal (code/logic) modifications to the component.

REP)

- Common Closure Principle (CCP)
- Common Reuse Principle (CRP)

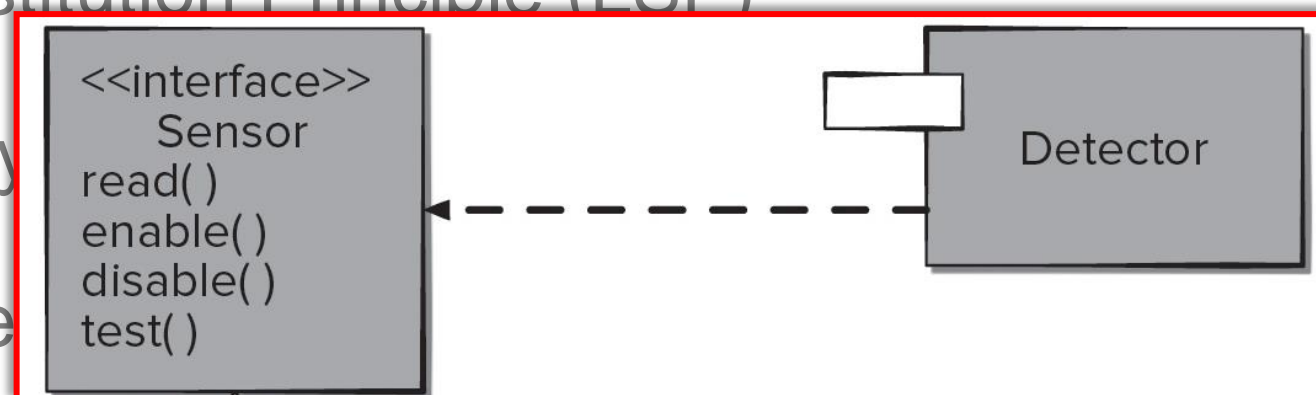
Basic Design Principles

- **Open-Closed Principle (OCP)**

- Liskov Substitution Principle (LSP)

- Dependency

- Interface Se



- Release Reuse Equivalency Principle (REP)

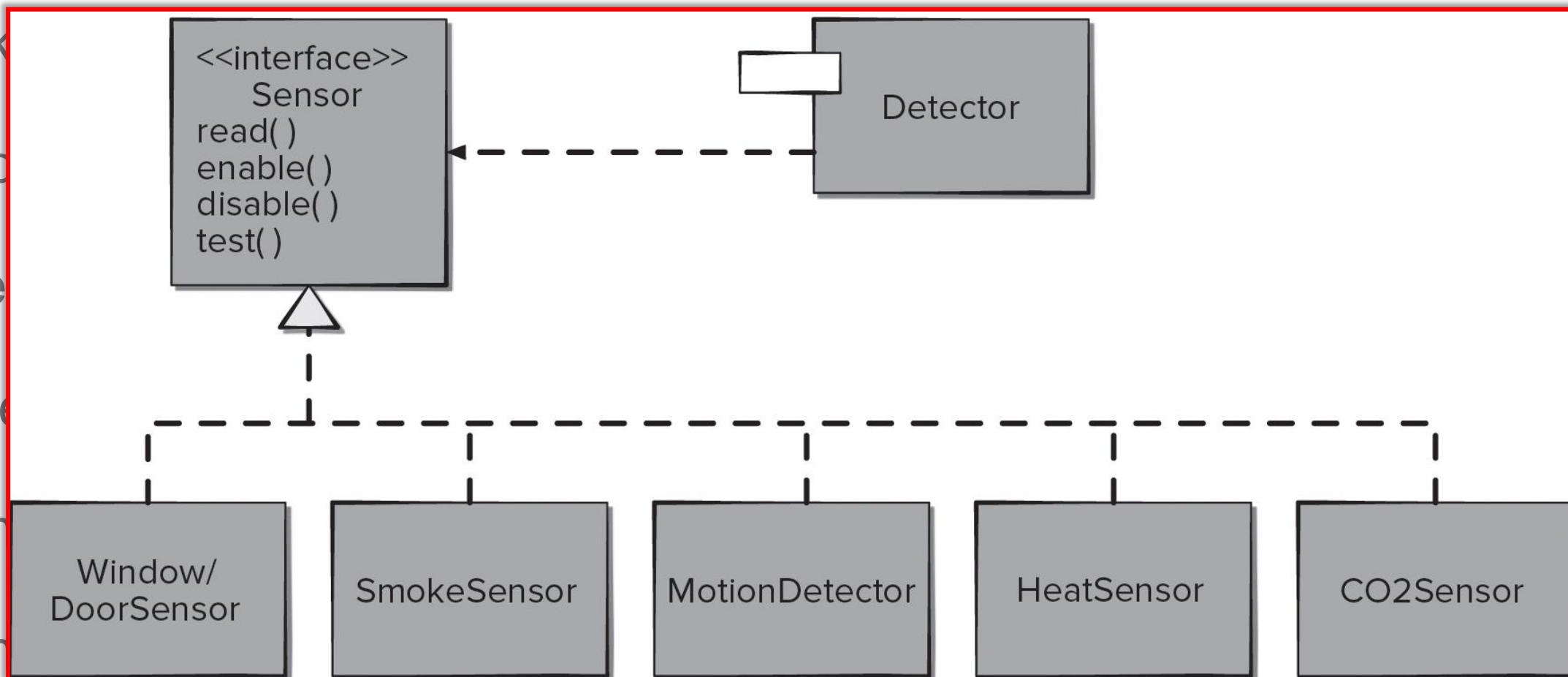
- Common Closure Principle (CCP)

- Common Reuse Principle (CRP)

Basic Design Principles

- **Open-Closed Principle (OCP)**

- Link
- Dep
- Inte
- Rel
- Con
- Con



Basic Design Principles

- Open-Closed Principle (OCP)
- **Liskov Substitution Principle (LSP)**

Liskov Substitution Principle (LSP)

“Subclasses should be substitutable for their base classes.”

- A component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.
- Common Reuse Principle (CRP)

Basic Design Principles

- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- **Dependency Inversion Principle (DIP)**

Dependency Inversion Principle (DIP)

“Depend on abstractions. Do not depend on concretions.”

- **Abstractions** are the place where a design can be **extended without great complication**.
- The more components depend concrete components (rather than abstractions such as an interface), the more difficult it will be to extend.

Interface Segregation Principle (ISP)

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

- Only those operations that are relevant to an individual client category should be specified in the interface for that client.
- That is to say, **no code should be forced to depend on operations it does not use.**

- **Interface Segregation Principle (ISP)**

- Release Reuse Equivalency Principle (REP)
- Common Closure Principle (CCP)
- Common Reuse Principle (CRP)

Interface Segregation Principle (ISP)

“Many client-specific interfaces are better than one.”

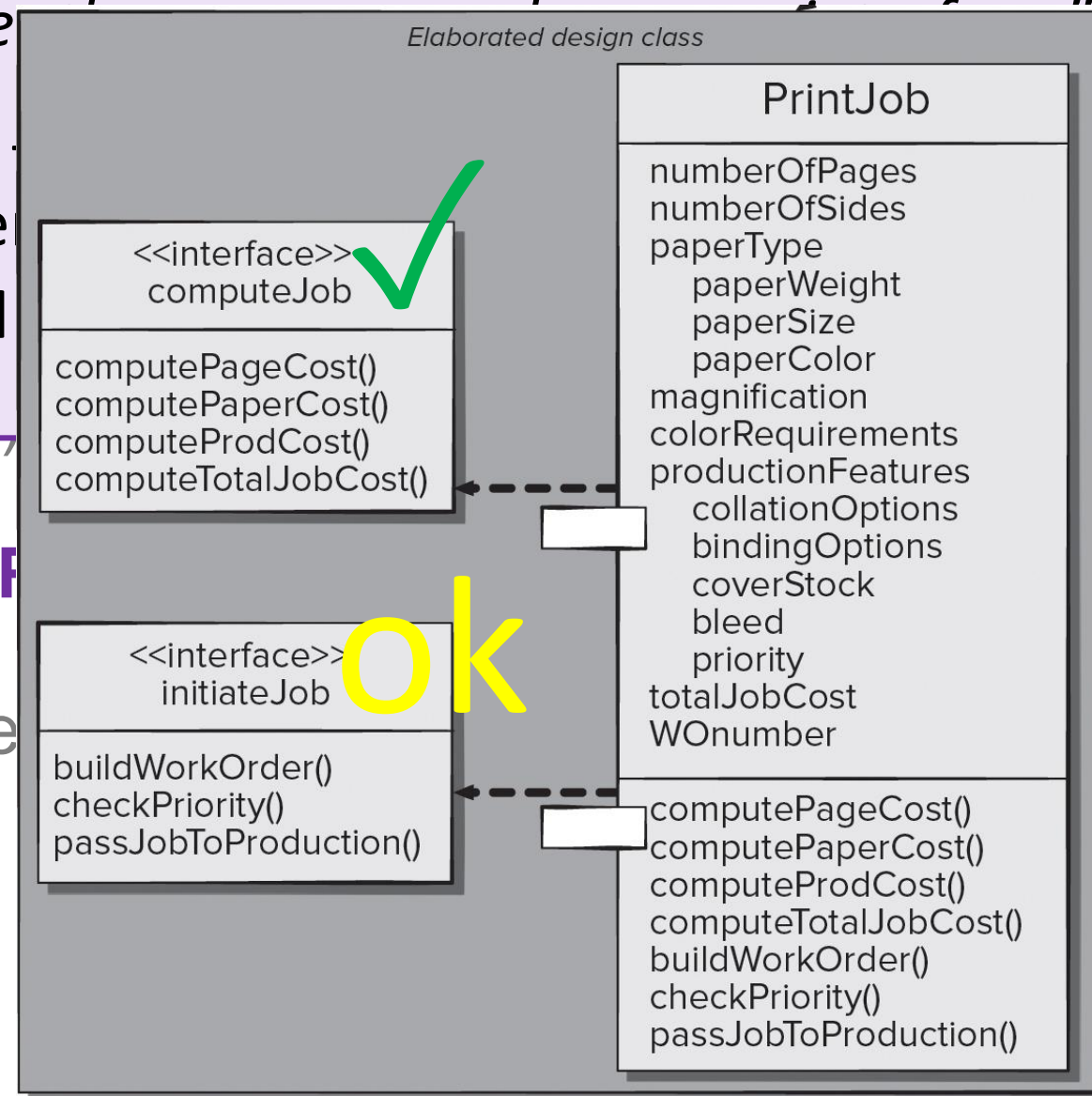
- Only those operations that are relevant to a client should be specified in the interface for that client.
- That is to say, **no code should be forced to use** an interface.

• Interface Segregation Principle (ISP)

• Release Reuse Equivalency Principle (RREP)

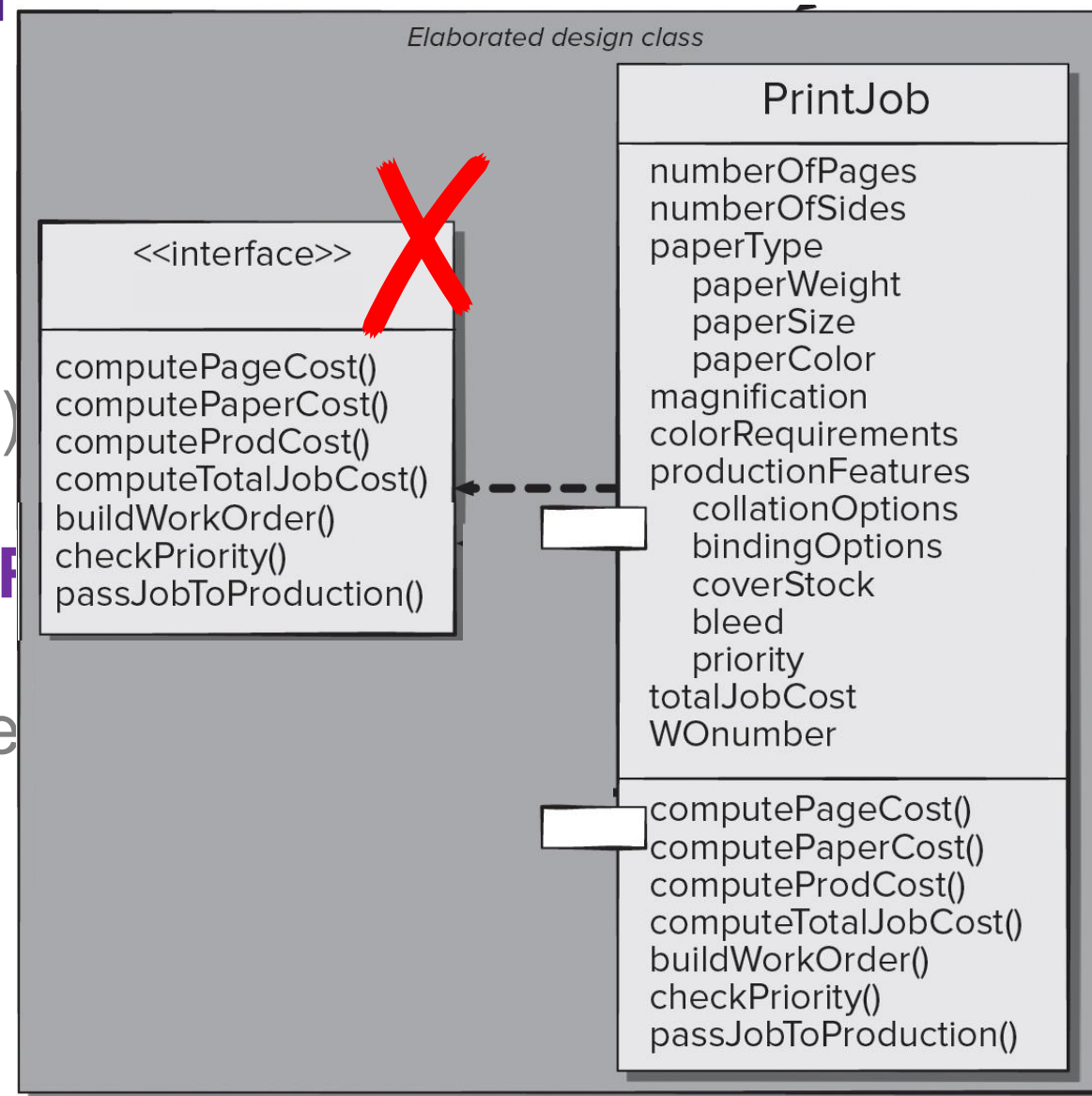
• Common Closure Principle (CCP)

• Common Reuse Principle (CRP)



Basic Design Principles

- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- **Interface Segregation Principle (ISP)**
- Release Reuse Equivalency Principle
- Common Closure Principle (CCP)
- Common Reuse Principle (CRP)



Basic Design Principles

- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- Interface Segregation Principle (ISP)
- **Release Reuse Equivalency Principle (REP)**
- **Common Closure Principle (CCP)**
- **Common Reuse Principle (CRP)**

**Component Placing and
Packaging Principles**

Basic Design Principles

Release Reuse Equivalency Principle (REP)

“The granule of reuse is the granule of release.”

- Effective reuse requires tracking of releases from a change control system. The package is the effective unit of reuse and release.
- When designing for reuse, don't address classes individually, group them into packages that can be more easily managed.

Interface Segregation Principle (ISP)

- **Release Reuse Equivalency Principle (REP)**

- Common Closure Principle (CCP)

- Common Reuse Principle (CRP)

Basic Design Principles

Common Closure Principle (CCP)

“Classes that change together belong together.”

- Classes should be packaged **cohesively**.
 - Classes in the same package should address the same functional or behavioural area.
 - When characteristics of that area change, it is likely that only those classes within the package will require modification.
-
- Release Reuse Equivalency Principle (REP)
 - **Common Closure Principle (CCP)**
 - Common Reuse Principle (CRP)

Basic Design Principles

- Open-Closed Principle (OCP)

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 the same package.
 - Unrelated classes lead to extra integration work and testing.
 - Also means new releases of the package even if changes have no impact on some systems.
- Common Closure Principle (CCP)
 - **Common Reuse Principle (CRP)**

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.

Types of Cohesion:

- 1. Functional:** Module performs **one and only one computation**.
- 2. Layer:** Occurs when a **higher layer accesses the services of a lower layer**, but **lower layers do not access higher layers**.
- 3. 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.

Types of Coupling:

1. **Content:** Occurs when one component “*surreptitiously*” **modifies data that is internal to another component.**
2. **Control:** Occurs when control flags are passed to components to **request alternate behaviours** when invoked.
3. **External:** Occurs when a component communicates or **collaborates with infrastructure components.**

Coupling

Traditional View: Degree to which a component is connected to the external world.

Object-Oriented View: Quantity of components connected to one another.

Types of Coupling:

1. **Content:** Occurs when one component “*surreptitiously*” **modifies data that is internal to another component.**
2. **Control:** Occurs when control flags are passed to components to **request alternate behaviours** when invoked.
3. **External:** Occurs when a component communicates or **collaborates with infrastructure components.**

Control Coupling

```
public int doMath(int a, int b, int flag) {  
    switch(flag) {  
        case ADDITION: return a + b;  
        case MULTIPLICATION: return a * b;  
        case SUBTRACTION: return a - b;  
        default: return 0;  
    }  
}
```

Component-Level Design

- **Step 1:** Identify all **design classes** that correspond to the **problem domain**.
- **Step 2:** Identify all **design classes** that correspond to the **infrastructure domain**.

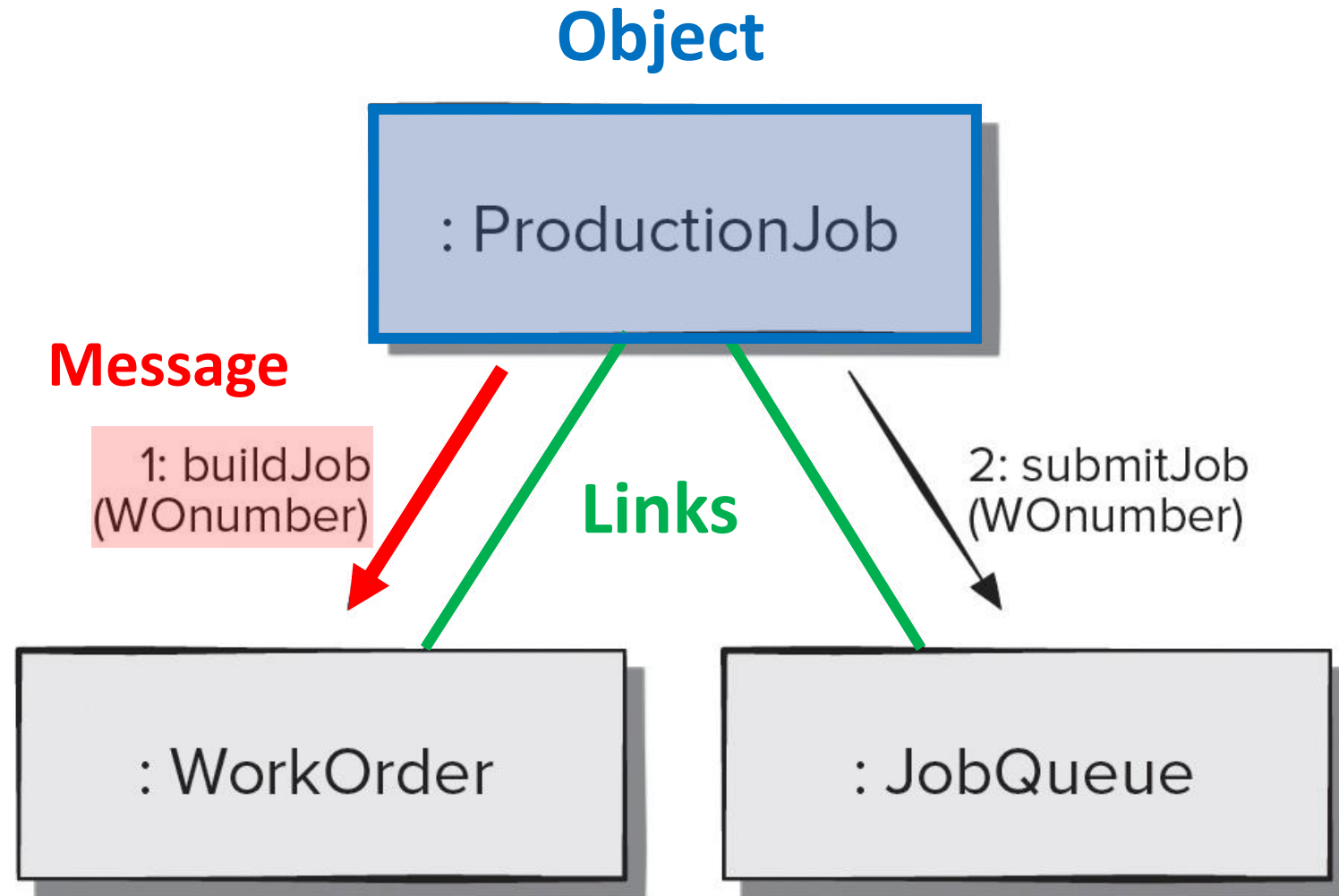
Component-Level Design

- **Step 3: Elaborate** all **design classes** that are not acquired as reusable components.
 - **Step 3a:** Specify **message details** when classes or component collaborate. **Collaboration diagrams** commonly uses.
 - **Step 3b:** Identify appropriate **interfaces** for each component.
 - **Step 3c:** Elaborate **attributes**, define **data types**, and **data structures**.
 - **Step 3d:** Describe **processing flow** within each operation in detail, using **pseudocode**, an **activity diagram**, or a **flowchart**.

Collaboration Diagram

Collaboration diagram from our earlier printing example, with messaging details shown.

- Sequence numbers show the order of messages
- Names of operations are given (buildJob and submitJob)
- Argument list is provided (a WOnumber).



Component-Level Design

- **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; **state diagrams** can be useful for this.
- **Step 6: Elaborate deployment diagrams** to provide additional implementation detail.
- **Step 7: Refactor** every component-level design representation and always consider alternatives.