
# Software Design and Architecture

## [Principles of Detailed Design] – Chapter 05, L03

# Lecture Outlines

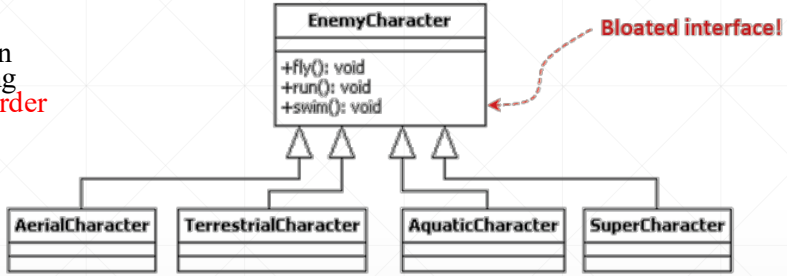
- **Previously we covered the following design principles for internal component design**
  - ✓ **S**ingle Responsibility Principle
  - ✓ **O**pen-Closed Principle
  - ✓ **L**iskov Substitution Principle

- **This lecture we will cover the following topics:**
  - ✓ **Continue with Component Design Principles:**
    - **I**nterface-Segregation Principle
    - **D**ependency-Inversion Principle
  - ✓ **Designing Internal Behavior of Components**
  - ✓ **Rules of Design**

# Interface Segregation Principle (ISP)

- Well-designed classes should have one (and only one) reason to change.

- When this concept is violated, there is a strong indication that the interfaces provided by these classes are providing more information than they should ➔ makes designs harder to maintain and reuse.
- The interface segregation principle (ISP) states that **classes/methods should not be forced to depend on methods that they do not use**.

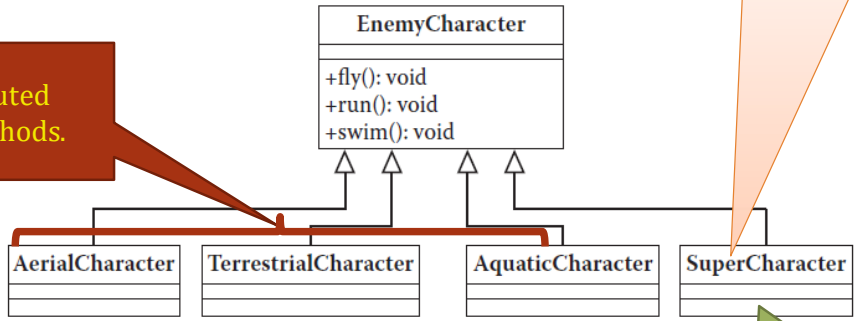



- Consider a gaming system that have different types of enemy character that is able to either **roam over land**, **fly**, or **swim**. The game also have a super enemy characters that can **do all** (roam over land, fly, or swim).
  - ✓ Some would be tempted to design the system as seen in the figure.

The ISP implies that many client-specific interfaces are better than one general purpose interface.

# ISP

the SuperCharacter class requires behaviour from all other enemy characters




These classes are polluted with unnecessary methods.

The design violates the LSP, why?

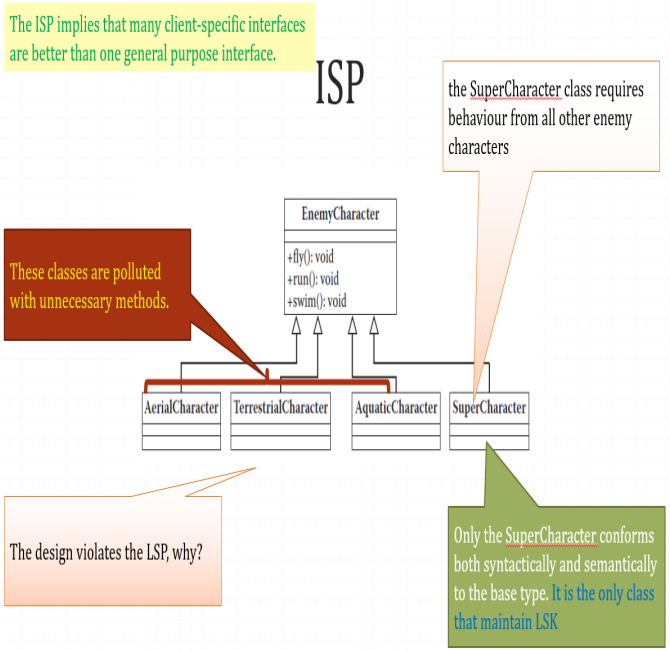
Only the SuperCharacter conforms both syntactically and semantically to the base type. It is the only class that maintain LSK

## Extra Explanation:

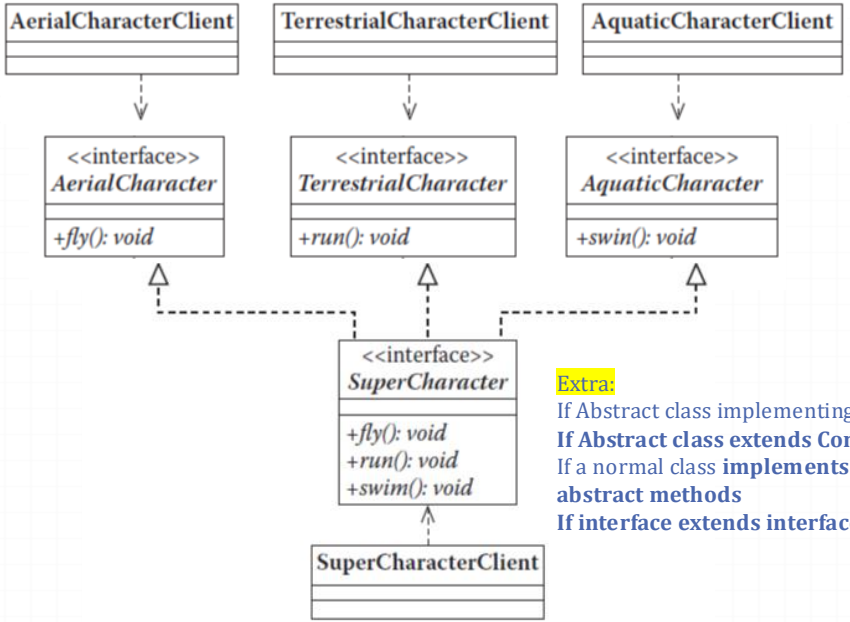
**1. ISP (Interface Segregation Principle)**
**What it means:** Use small, specific interfaces instead of one big interface with many unused methods.
**Problem:** Classes (e.g., AerialCharacter, AquaticCharacter) are forced to implement methods they don't need (like run() ).
**Why it's bad:** Interfaces get "polluted" with unnecessary methods → violates ISP

**2. LSP (Liskov Substitution Principle)**
**What it means:** Subclasses should replace their parent class without breaking program correctness.
**Problem:** A TerrestrialCharacter cannot replace an EnemyCharacter if code expects fly().
**Why it's bad:** Subclasses don't fully match parent behavior. Only SuperCharacter correctly follows LSP.

👉 In short:
ISP → Interfaces should be specific.
LSP → Subclasses must behave consistently as their base class.



# ISP




Extra:

If Abstract class implementing interface → **(NO override)**

**If Abstract class extends Concrete subclass → YES, must override**

If a normal class **implements an interface**, it **MUST override ALL abstract methods**

**If interface extends interface, → (NO override)**

```java
// ===== Super Interface =====
interface SuperCharacter {
    void fly();
    void run();
    void swim();
}

// ===== Specialized Interfaces =====
interface AerialCharacter extends SuperCharacter {
    void fly();
}

interface TerrestrialCharacter extends SuperCharacter {
    void run();
}

interface AquaticCharacter extends SuperCharacter {
    void swim();
}

// ===== Clients =====

class SuperCharacterClient implements SuperCharacter {

    public void fly() {
        System.out.println("SuperCharacterClient is flying");
    }

    public void run() {
        System.out.println("SuperCharacterClient is running");
    }

    public void swim() {
        System.out.println("SuperCharacterClient is swimming");
    }
}
```

```java
class AerialCharacterClient implements AerialCharacter {
   public void fly() {
        System.out.println("SuperCharacterClient is flying");
    }
}

class TerrestrialCharacterClient implements TerrestrialCharacter {

public void run() {
        System.out.println("STerrestrialCharacterClient is running");
    }

}

class AquaticCharacterClient implements AquaticCharacter {
    public void swim() {
        System.out.println("AquaticCharacterClient is swimming");
    }
}

// ===== Main =====
public class Main {
    public static void main(String[] args) {

        AerialCharacterClient         arialObject =new AerialCharacterClient();
          arialObject.Fly();
        TerrestrialCharacterClient    terresObject =new TerrestrialCharacterClient();
          terresObject.run();
        AquaticCharacterClient        aquaObject =new AquaticCharacterClient();
          aquaObject.swim();
        SuperCharacterClient          superCharacter = new SuperCharacterClient();
          superCharacter.fly();
          superCharacter.swim();
          superCharacter.run();
    }
}
```

## ❌ ISP Violation (bad)

java

```java
interface Worker {
    void work();
    void eat();
}
```

java

```java
class Robot implements Worker {
    public void work() {
        System.out.println("Robot working");
    }

    public void eat() { }   // ❌ Robot does not eat
}
```

## ✅ ISP Applied (good)

java

```java
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}
```

java

```java
class Robot implements Workable {
    public void work() {
        System.out.println("Robot working");
    }
}
```

java

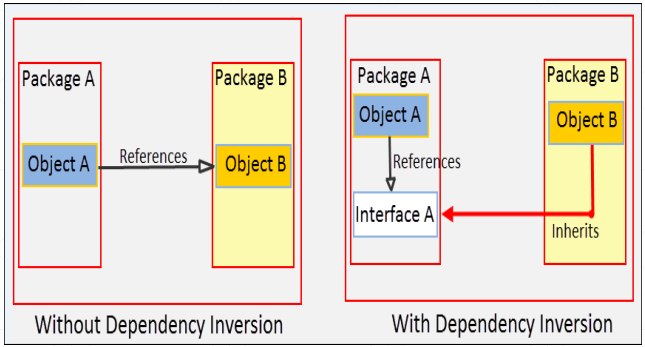
```java
class Human implements Workable, Eatable {
    public void work() {
        System.out.println("Human working");
    }

    public void eat() {
        System.out.println("Human eating");
    }
}
```

# Dependency Inversion Principle (DIP)

- The Dependency Inversion Principle (DIP) tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.
- In a statically typed language, like Java, this means that the use, import, and include statements should refer only to source modules containing interfaces, abstract classes, or some other kind of abstract declaration. **Nothing concrete should be depended on**.
- We need to avoid depending on modules that are actively developing, and that are undergoing frequent change.

# Dependency Inversion Principle (DIP)




**1- What is the Dependency Inversion Principle ?**

- **High-level modules should NOT depend on low-level modules.**
- **Both should depend on abstractions (interfaces).**

Also:

- **Abstractions should NOT depend on details.**
- **Details should depend on abstractions.**

# Extra:

### Imagine this real-life idea first

❌ **Bad idea**

You buy a **charger that works ONLY with Samsung phones**.

If you change your phone → charger is useless.

✅ **Good idea**

You buy a **charger with USB**.

Any phone can use it → Samsung, iPhone, anything.

👉 **USB = interface (abstraction)**

### ❌ WITHOUT Dependency Inversion (bad)

```java
class Fan {
    void turnOn() {
        System.out.println("Fan is ON");
    }
}

class Switch {
    Fan fan = new Fan();   // directly depends on Fan

    void press() {
        fan.turnOn();
    }
}
```

### ❌ Problem

- `Switch` works **ONLY with Fan**
- If you want a **Light** → must change `Switch`

## Extra: WITH Dependency Inversion (good)

### Step 1: Interface (USB idea)

```java
interface Device {
    void turnOn();
}
```

### Step 2: Devices implement it

```java
class Fan implements Device {
    public void turnOn() {
        System.out.println("Fan is ON");
    }
}

class Light implements Device {
    public void turnOn() {
        System.out.println("Light is ON");
    }
}
```

### Step 3: Switch depends on interface ONLY

```java
class Switch {
    Device device;

    Switch(Device device) {
        this.device = device;
    }

    void press() {
        device.turnOn();
    }
}
```

### Use it

```java
Switch s1 = new Switch(new Fan());
s1.press();

Switch s2 = new Switch(new Light());
s2.press();
```

# Dependency Inversion Principle (DIP)

- **Extra**: violates DIP
- class Worker → A worker who can do work.
- public void work() → The worker does the job (prints "working").
- class Manager → The boss who manages workers.
- Worker worker; → The boss is fixed to only one type: Worker.
- setWorker → Assigns a worker to the boss.
- **manage**() → Boss tells the worker to work.
- **Problem** → Boss only understands **Worker**, can't manage other worker types (like Robot).

```
// Dependency Inversion Principle - Bad example

class Worker {

        public void work() {

                // ....working
        }
}
class Manager {

        Worker worker;


        public void setWorker(Worker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}
```

# Dependency Inversion Principle (DIP)

```
class RobotWorker implements IWorker {
    public void work() {
        System.out.println("Robot is working...");
    }
}
```

**Good Example**

- **Manager depends on IWorker (interface), not on Worker.**
- **You can add new workers (e.g., RobotWorker, AIWorker, InternWorker) without changing Manager.**
- **High-level module (Manager) depends on abstraction, not low-level implementation.**

```
// Dependency Inversion Principle - Good example
interface IWorker {
        public void work();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
}

class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}
```

- interface IWorker → A contract: **any worker must know how to work**.
- class Worker implements IWorker → Worker follows the contract and knows how to work.
- Manager → The boss.
- IWorker worker; → Boss only cares about the **interface**, not the exact type.
- setWorker → Assign any worker (human, robot, etc.) to the boss.
  manage() → Boss tells the worker (whatever type) to work.
- **Good** → Now Manager can handle **any worker type** that implements IWorker.

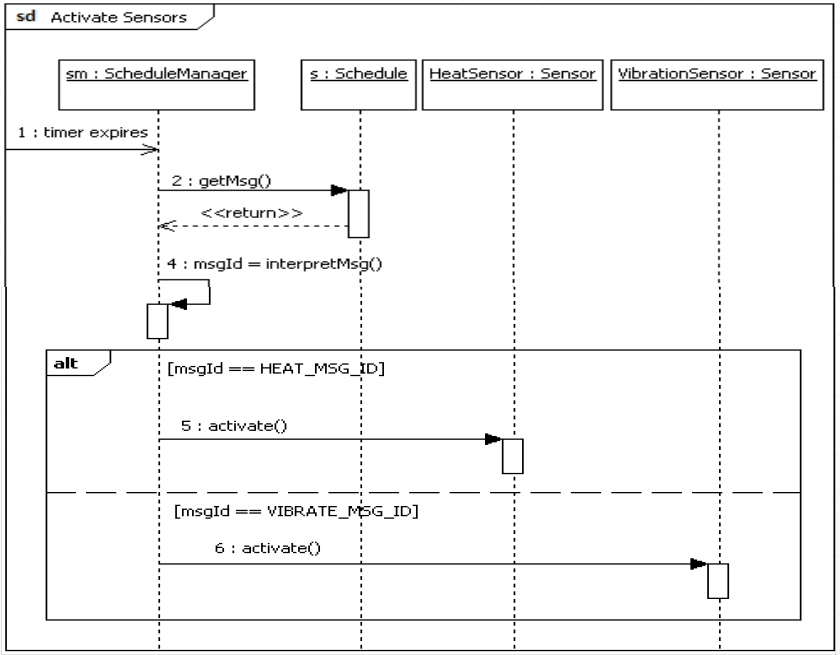
So:

•**Bad code**: Manager depends on **Worker** → tightly coupled.

•**Good code**: Manager depends on **IWorker** → loosely coupled, flexible.

# Modeling Internal Behavior of a Component

- The component design considers modeling the internal structure and the behavior of the component
- As discussed in earlier slides, the internal structure of a component is modeled using a **class diagram**
- We model the internal behavior of a component using a **sequence diagram**



# Rules of Design

- **Make sure that the problem is well-defined**
  - ✓ All design criteria, requirements, and constraints, should be known before a design is started.
- **What comes before how**
  - ✓ i.e. Define the functionalities to be performed at every level of abstraction before deciding which structures should be implementing these functionalities.
- **Separate orthogonal concerns**
  - ✓ Do not connect what is independent.

# Rules of Design

- **Design external functionality before internal functionality.**
  - ✓ First consider the solution as a black-box and decide how it should interact with its environment.
  - ✓ Then decide how the black-box can be internally organized. Likely it consists of smaller black-boxes that can be refined in a similar fashion.
- **Keep it simple.**
  - ✓ Fancy designs are buggier than simple ones; they are harder to implement, harder to verify, and often less efficient.
  - ✓ Problems that appear complex are often just simple problems huddled together.
  - ✓ Our job as designers is to identify the simpler problems, separate them, and then solve them individually.

# Rules of Design

- **Work at multiple levels of abstraction**
  - ✓ Good designers must be able to move between various levels of abstraction quickly and easily.
  - ✓ Design for extensibility
- **Design for extensibility**
  - ✓ A good design is “open-ended,” i.e., easily extendible.
  - ✓ Do not introduce what is immaterial (not important).
  - ✓ Do not restrict what is irrelevant (not related).

# Rules of Design

- **Use rapid prototyping when applicable**
  - ✓ Before implementing a design, build a high-level prototype and verify that the design criteria are met.
- **Details should depend upon abstractions**
  - ✓ Abstractions should not depend upon details
- **Classes within a released component should share common closure**
  - ✓ That is, if one needs to be changed, they all are likely need to be changed
  - ✓ i.e., what affects one, affects all

# Rules of Design

- **Classes within a released component should be reused together**
  - ✓ That is, it is impossible to separate the components from each other in order to reuse less than the total
- **Dependencies between released components must run in the direction of stability**
  - ✓ The dependee must be more stable than the depender
- **The more stable a released component is, the more it must consist of abstract classes**
  - ✓ A completely stable component consist of nothing but abstract classes

# Common Design Mistakes

- **Depth-first design**
  - ✓ Only partially satisfy the requirements (Experience is best cure for this problem)
- **Directly refining requirements specification**
  - ✓ Leads to overly constrained, inefficient designs
- **Failure to consider potential changes**
  - ✓ Always design for extension and contraction
- **Making the design too detailed**
  - ✓ This over-constrains the implementation

# Summary

- **Design Principles for Internal Component Design (Continue)**
  - ✓ Single Responsibility Principle,
  - ✓ Dependency Inversion principle,
- **Designing Internal Behavior of Components**
- **Rules of Design**
- **Next …**
  - We will start the following chapter →**Creational Design Patterns**

1. Extra:
2. **Define the problem first**
   Understand *what* you are solving before designing.
3. **Know requirements and constraints early**
   Don't start design without clear requirements.
4. **What before how**
   Define *what the system should do* before *how it will do it*.
5. **Separate independent concerns**
   Don't connect things that are unrelated.
6. **Design external behavior first**
   Treat the system as a **black box** → then design its inside.
7. **Keep it simple**
   Simple designs are easier, safer, and more reliable.
8. **Work at multiple abstraction levels**
   Move easily between high-level view and low-level details.
9. **Design for extensibility**
   Allow easy future changes and additions.
10. **Avoid the unnecessary**
    Don't add what is unimportant or unrelated.
11. **Use rapid prototyping when possible**
    Build a quick model to validate the design early.
12. **Depend on abstractions, not details**
    Details should depend on interfaces, not the opposite.
13. **Classes in one component change together**
    If one changes, related ones likely change too.
14. **Classes in one component are reused together**
    You reuse the component as a whole, not partially.
15. **Dependencies go toward stability**
    Less stable components depend on more stable ones.
16. **Stable components should be abstract**
    The most stable components contain mostly abstractions.

# Analysis Models to Design Models