

OVERVIEW Principles of Detailed Design III: ISP, DIP, Internal Behaviour and Rules of Design

This lecture finishes SOLID. The Interface Segregation Principle says no class should be forced to depend on methods it does not use, so one bloated interface is split into small, role-specific ones. The Dependency Inversion Principle says high-level modules and low-level details should both depend on abstractions, so a Manager talks to an IWorker interface instead of a concrete Worker. It then shows how a component's internal behaviour is modelled (sequence diagrams, next to class diagrams for structure), and closes with sixteen rules of design and four common design mistakes.

ISP

Many client-specific interfaces are better than one general-purpose interface. Classes shouldn't implement methods they can't honour.

DIP

Depend on abstractions, not concretions: high-level policy and low-level detail both point at an interface.

Structure vs. behaviour

A component's internal structure is a class diagram; its internal behaviour is a sequence diagram.

Rules of design

What before how, external before internal, keep it simple, depend on abstractions, dependencies toward stability.

HOW THE LECTURE IS ORGANIZED

Slides 2–8 Interface Segregation Principle: the enemy-character game and the Worker example

Slides 9–14 Dependency Inversion Principle: charger, switch and Manager examples

Slide 15 Modelling the internal behaviour of a component

Slides 16–23 Rules of design and common design mistakes

Slide 24 From analysis models to design models

EXAM FOCUS

SRP and OCP from Lecture 2 have appeared on Final 252; ISP and DIP have not yet, so the questions here are predictions. Expect "which principle is violated?" scenarios: a class forced to implement a method it can't support (ISP), or a high-level class that holds a concrete low-level class (DIP).

HOW TO READ THIS PAGE

Every original slide is shown unchanged. **Yellow marks** on a slide point to a numbered note: hover or tap a mark to jump to its note, and hover a note to see its marks. Questions tagged **Past exam** appeared on a real paper; **Prediction** questions are likely angles, not confirmed.



Software Design and Architecture

[Principles of Detailed Design] – Chapter 05, L03

NOTES

- 1 Third detailed-design lecture: the last two SOLID principles (I and D), then how to model a component's behaviour and general rules of good design.

SLIDE 2 / 24 Lecture Outlines

Lecture Outlines

- **Previously we covered the following design principles for internal component design**
 - ✓ **Single Responsibility Principle**
 - ✓ **Open-Closed Principle**
 - ✓ **Liskov Substitution Principle**
- **This lecture we will cover the following topics:**
 - ✓ **Continue with Component Design Principles:**
 - **Interface-Segregation Principle**
 - **Dependency-Inversion Principle**
 - ✓ **Designing Internal Behavior of Components**
 - ✓ **Rules of Design**

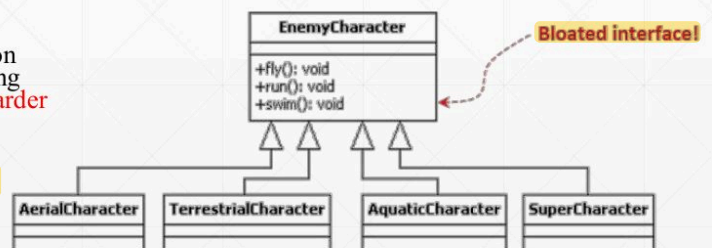
NOTES

- 1 Already covered in Lecture 2: **SRP**, **OCP** and **LSP**.
- 2 This lecture: the **Interface Segregation Principle** and the **Dependency Inversion Principle**, completing **SOLID**.
- 3 Then designing internal behaviour and the rules of design.

SLIDE 3 / 24 Interface Segregation Principle (ISP)

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.



NOTES

- 1 The starting point is SRP: a well-designed class has one reason to change.
- 2 When that is violated, the **interfaces** the class offers usually provide more than they should, which makes designs harder to maintain and reuse.
- 3 The **Interface Segregation Principle (ISP)**: classes (or methods) should not be **forced to depend on methods they do not use**.
- 4 The example: a game with enemy characters that roam over land, fly or swim, plus a super enemy that can do all three.
- 5 The tempting design: one `EnemyCharacter` with `fly()`, `run()` and `swim()`, inherited by `Aerial`, `Terrestrial`, `Aquatic` and `SuperCharacter`. The slide labels it a **bloated interface**.

Prediction

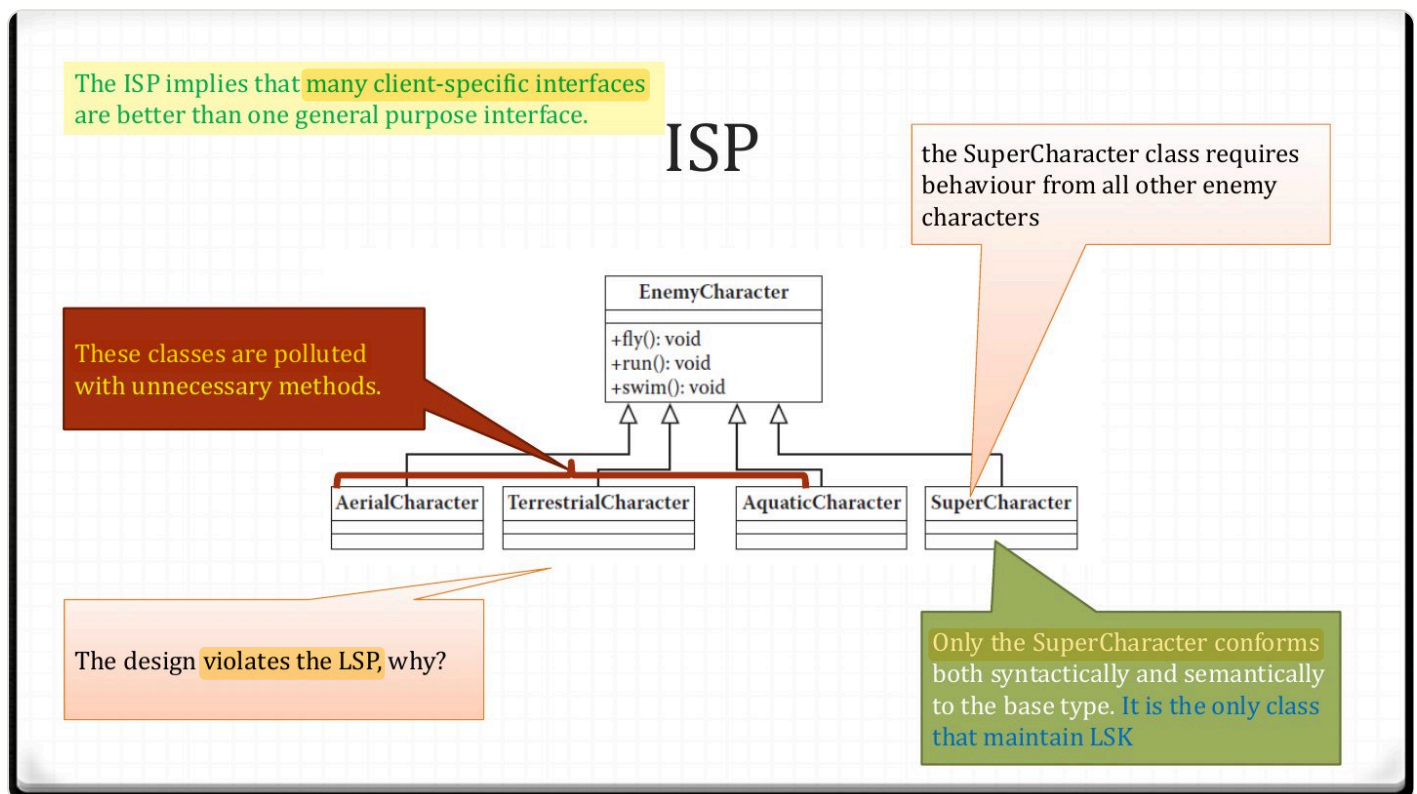
Q1

"Clients should not be forced to depend on methods they do not use" states the:

- A. Interface Segregation Principle
- B. Dependency Inversion Principle
- C. Open-Closed Principle
- D. Liskov Substitution Principle

Answer: A. Interface Segregation Principle

SLIDE 4 / 24 ISP: one general interface vs. many specific ones



NOTES

- 1 The rule of thumb: **many client-specific interfaces are better than one general-purpose interface.**
- 2 Aerial, Terrestrial and Aquatic characters are **polluted** with methods they can't perform: an aerial character inherits `run()` and `swim()`, a terrestrial one `fly()`.
- 3 The design also **violates LSP** (Lecture 2): code holding an `EnemyCharacter` may call `fly()`, but a `TerrestrialCharacter` put in its place can't fly, so it can't safely substitute for its parent.
- 4 Only **SuperCharacter** conforms to the base type syntactically and semantically. It is the only class that really needs (and honours) all three methods, so it is the only one that satisfies LSP.

Prediction

Q1

In the enemy-character design where every character extends EnemyCharacter with fly(), run() and swim(), which principles are violated and why?

Answer: ISP: Aerial, Terrestrial and Aquatic characters are forced to depend on methods they don't use. LSP: a TerrestrialCharacter cannot replace an EnemyCharacter where code expects fly(), so sub-classes don't behave like their parent. Only SuperCharacter conforms.

SLIDE 5 / 24 Extra: ISP and LSP explained

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.

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

ISP

These classes are polluted with unnecessary methods.

The design violates the LSP, why?

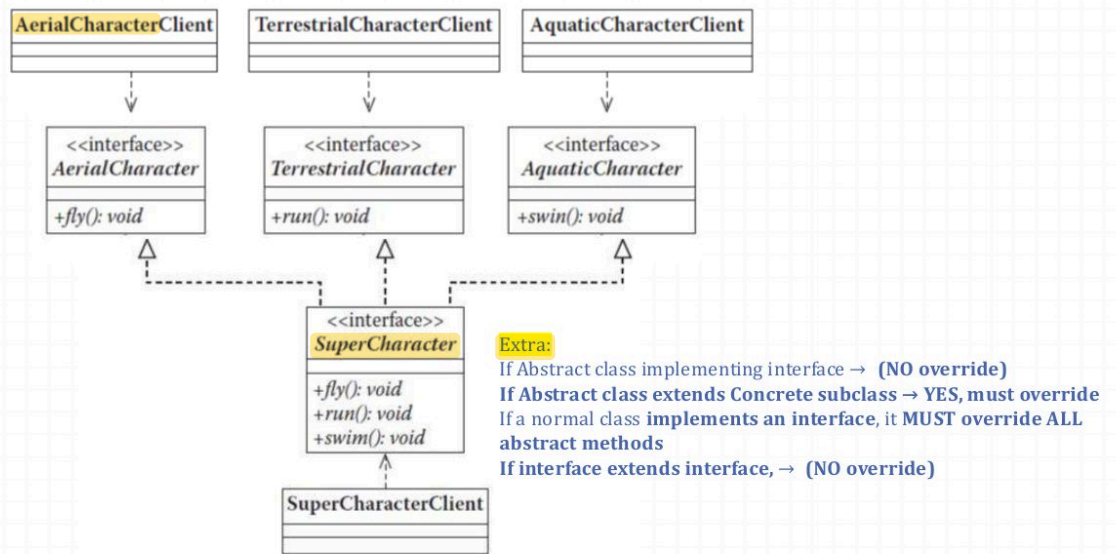
the SuperCharacter class requires behaviour from all other enemy characters

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

NOTES

- 1 ISP:** use small, specific interfaces instead of one big interface full of unused methods. Problem here: AerialCharacter and AquaticCharacter are forced to implement `run()`; the interface is polluted.
- 2 LSP:** a subclass must replace its parent without breaking correctness. Problem here: a TerrestrialCharacter can't replace an EnemyCharacter when the code expects `fly()`.
- 3 In short:** ISP → interfaces should be specific; LSP → subclasses must behave consistently with their base class. Fixing the interfaces (next slide) fixes both.

ISP



6

NOTES

- 1 The ISP fix: three small interfaces, «interface» **AerialCharacter** (`fly()`), **TerrestrialCharacter** (`run()`) and **AquaticCharacter** (`swim()`). Each client class (dashed realization arrows) implements only the one it can honour.
- 2 «interface» **SuperCharacter** declares all three operations and is implemented by SuperCharacterClient. In the slide's version the three small interfaces extend SuperCharacter (hollow triangles pointing down to it); conceptually it is the combination of the three capabilities.
- 3 The Extra box: Java rules about overriding when types are combined:
 - Abstract class implements an interface**
No override needed: it can leave the methods abstract.
 - Abstract class extended by a concrete class**
Yes: the concrete subclass must override every abstract method.
 - Normal class implements an interface**
It must override all the interface's abstract methods.
 - Interface extends interface**
No override: interfaces only add declarations.

SLIDE 7 / 24 ISP: the segregated design in Java

```
1 // ===== Super Interface =====
2 interface SuperCharacter {
3     void fly();
4     void run();
5     void swim();
6 }
7
8 // ===== Specialized Interfaces =====
9 interface AerialCharacter extends SuperCharacter {
10     void fly();
11 }
12
13 interface TerrestrialCharacter extends SuperCharacter {
14     void run();
15 }
16
17 interface AquaticCharacter extends SuperCharacter {
18     void swim();
19 }
20
21 // ===== Clients =====
22
23 class SuperCharacterClient implements SuperCharacter {
24
25     public void fly() {
26         System.out.println("SuperCharacterClient is flying");
27     }
28
29     public void run() {
30         System.out.println("SuperCharacterClient is running");
31     }
32
33     public void swim() {
34         System.out.println("SuperCharacterClient is swimming");
35     }
36 }
37
38 class AerialCharacterClient implements AerialCharacter {
39     public void fly() {
40         System.out.println("SuperCharacterClient is flying");
41     }
42 }
43
44 class TerrestrialCharacterClient implements TerrestrialCharacter {
45
46     public void run() {
47         System.out.println("SuperCharacterClient is running");
48     }
49 }
50
51
52 class AquaticCharacterClient implements AquaticCharacter {
53     public void swim() {
54         System.out.println("AquaticCharacterClient is swimming");
55     }
56 }
57
58 // ===== Main =====
59 public class Main {
60     public static void main(String[] args) {
61
62         AerialCharacterClient    arialObject = new AerialCharacterClient();
63         arialObject.fly();
64         TerrestrialCharacterClient terresObject = new TerrestrialCharacterClient();
65         terresObject.run();
66         AquaticCharacterClient    aquaObject = new AquaticCharacterClient();
67         aquaObject.swim();
68         SuperCharacterClient      superCharacter = new SuperCharacterClient();
69         superCharacter.fly();
70         superCharacter.swim();
71         superCharacter.run();
72     }
73 }
```

NOTES

1 The code, top to bottom:

CODE	WHAT IT DOES
<pre>interface SuperCharacter { void fly(); void run(); void swim(); }</pre>	The full contract: all three capabilities.
<pre>interface AerialCharacter extends SuperCharacter { void fly(); }</pre>	A specialised interface that names the capability its clients use. (Because it <i>extends</i> SuperCharacter, a class implementing it must still provide all three methods; to apply ISP strictly, the small interfaces should not extend the big one.)
<pre>interface TerrestrialCharacter extends SuperCharacter { void run(); }</pre>	The same for running.
<pre>interface AquaticCharacter extends SuperCharacter { void swim(); }</pre>	The same for swimming.
<pre>class SuperCharacterClient implements SuperCharacter { fly(){...} run(){...} swim(){...} }</pre>	The only class that really supports all three; each prints what it is doing.
<pre>class AerialCharacterClient implements AerialCharacter { public void fly(){ ... "is flying" } }</pre>	Implements only <code>fly()</code> . As printed it would not compile, because the inherited <code>run()</code> and <code>swim()</code> are missing; with stand-alone small interfaces it would.
<pre>class TerrestrialCharacterClient implements TerrestrialCharacter { run() }</pre>	Runs only.
<pre>class AquaticCharacterClient implements AquaticCharacter { swim() }</pre>	Swims only.
<pre>AerialCharacterClient arialObject = new AerialCharacterClient(); arialObject.fly();</pre>	In <code>main</code> , each object is used through the capability it supports.
<pre>SuperCharacterClient superCharacter = new SuperCharacterClient(); superCharacter.fly(); ...swim(); ...run();</pre>	The super character can be asked for all three.

2 The intent: a reference of a narrow type can only request what that character can do, so unsupported calls become impossible. Exam answers should draw the small interfaces as independent, each implemented by the classes that need it.

✗ ISP Violation (bad)

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

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

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

✓ ISP Applied (good)

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

interface Eatable {
    void eat();
}

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

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

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

NOTES

1 **Violation:** one `Worker` interface with `work()` and `eat()`.

CODE	WHAT IT DOES
<code>interface Worker { void work(); void eat(); }</code>	One fat interface for two unrelated capabilities.
<code>class Robot implements Worker {</code>	A robot can work but cannot eat...
<code> public void work() {</code> <code> System.out.println("Robot working"); }</code>	...so <code>work()</code> is meaningful,
<code> public void eat() { } // Robot</code> <code>does not eat</code>	but <code>eat()</code> is forced on it and left empty: a lie in the type.

2 **Applied:** split the interface by capability.

CODE	WHAT IT DOES
<code>interface Workable { void work(); }</code>	The working capability.
<code>interface Eatable { void eat(); }</code>	The eating capability.
<code>class Robot implements Workable {</code> <code>work() }</code>	The robot implements only what it can do.
<code>class Human implements Workable,</code> <code>Eatable { work() ... eat() }</code>	A human implements both: a class may implement several small interfaces.

Prediction

Q1

A Robot class implements a Worker interface that declares work() and eat(), and leaves eat() empty. Which principle is violated, and how do you fix it?

Answer: The Interface Segregation Principle: Robot is forced to depend on eat(), which it does not use. Split Worker into Workable (work()) and Eatable (eat()); Robot implements Workable only, Human implements both.

SLIDE 9 / 24 Dependency Inversion Principle (DIP)

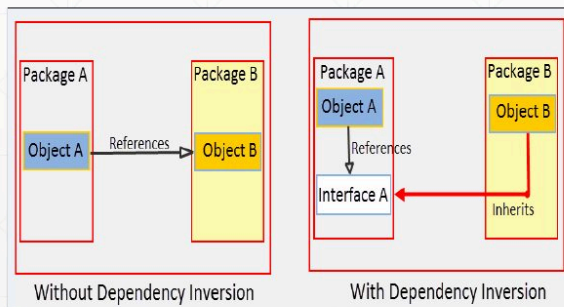
Dependency Inversion Principle (DIP)

- 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**.

NOTES

- 1 The **Dependency Inversion Principle (DIP)**: the most flexible systems are those whose source-code dependencies refer **only to abstractions**, not to concretions.
- 2 In a statically typed language like Java, that means **use, import and include** statements should name only interfaces, abstract classes or other abstract declarations.
- 3 **Nothing concrete should be depended on**: concrete classes change often, abstractions rarely.
- 4 In particular, avoid depending on modules that are **actively being developed** and change frequently: every change there would ripple into your code.

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.

NOTES

- 1 The two rules. **(1)** High-level modules (policy, e.g. a Manager) must not depend on low-level modules (details, e.g. a Worker). **Both** depend on an abstraction (an interface).
- 2 **(2)** Abstractions must not depend on details; details depend on abstractions.
- 3 The diagram: **without** inversion, Object A in Package A references Object B in Package B directly. **With** inversion, Object A references Interface A, and Object B implements (inherits) Interface A. The arrow from B now points *toward* the abstraction, the opposite ("inverted") direction of the original dependency.

EXAM ANGLE

Prediction

Q1

According to DIP, a high-level module should depend on:

- A. Abstractions (interfaces)
- B. Low-level modules
- C. Concrete classes
- D. Global variables

Answer: A. Abstractions (interfaces)

Extra:

Imagine this real-life idea first

✗ Bad ideaYou 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

NOTES

- 1 The real-life idea: a charger that only works with Samsung phones is useless when you change phones (bad). A **USB** charger works with any phone (good): USB is the interface, the abstraction.
- 2 The same mistake in code, the **charger that only fits one phone**:
- 3 Switch without dependency inversion:

CODE**WHAT IT DOES**

```

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

```

A concrete low-level device.

```

class Switch {

```

The high-level module.

```

    Fan fan = new Fan(); // directly
    depends on Fan

```

The switch creates and holds a concrete Fan.

```

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

```

Pressing can only ever turn on a fan.

- 4 The problem: Switch works **only** with Fan. To control a Light you 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();
```

NOTES**1** The fix in three steps:**CODE**

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

```
class Fan implements Device { public
void turnOn() { ..."Fan is ON"... } }
```

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

```
class Switch { Device device;
Switch(Device device) { this.device =
device; } void press() {
device.turnOn(); } }
```

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

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

WHAT IT DOES

Step 1: the abstraction (the "USB port").

Step 2: the details implement the abstraction.

A second device, added without touching Switch.

Step 3: Switch knows only Device; the concrete device is passed in through the constructor.

Wiring a fan: prints "Fan is ON".

Wiring a light: prints "Light is ON". Same Switch class.

2 Passing the dependency in from outside (instead of `new Fan()` inside) is called dependency injection; it is the usual way to apply DIP.

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();
    }
}
```

Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

13

NOTES

- 1 The textbook bad example, line by line:

CODE	WHAT IT DOES
class Worker { public void work() { // working } }	A concrete worker.
class Manager {	The high-level module (the boss).
Worker worker;	The manager is fixed to one concrete type.
public void setWorker(Worker w) { worker = w; }	It can only be given a Worker (the circled parameter type).
public void manage() { worker.work(); } }	It tells that worker to work.

- 2 Manager is the high-level module and depends directly on the low-level Worker.
- 3 **Problem:** the boss only understands Worker. A Robot or an Intern can't be managed without changing Manager, and tests can't substitute a fake worker.

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.
- **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.

```
// 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();
    }
}
```

14

NOTES**1** The good example:

CODE	WHAT IT DOES
<pre>interface IWorker { public void work(); }</pre>	The contract: any worker must know how to work.
<pre>class Worker implements IWorker { public void work() { // working } }</pre>	The original worker now follows the contract.
<pre>class RobotWorker implements IWorker { public void work() { ..."Robot is working..." } } </pre>	A new kind of worker, added without touching Manager.

2 Manager now depends only on the abstraction:

CODE	WHAT IT DOES
<pre>class Manager { IWorker worker; public void setWorker(IWorker w) { worker = w; } public void manage() { worker.work(); } }</pre>	The boss only cares about the interface, not the exact type. Any worker (human, robot, AI, intern) can be assigned. At run time the actual object's <code>work()</code> runs.

- 3** Bad code: Manager → Worker (tightly coupled). Good code: Manager → IWorker ← Worker, RobotWorker (loosely coupled and flexible). New workers such as **RobotWorker**, **AIWorker** or **InternWorker** need no change to Manager, which also makes Manager open for extension (OCP).

Prediction

Q1

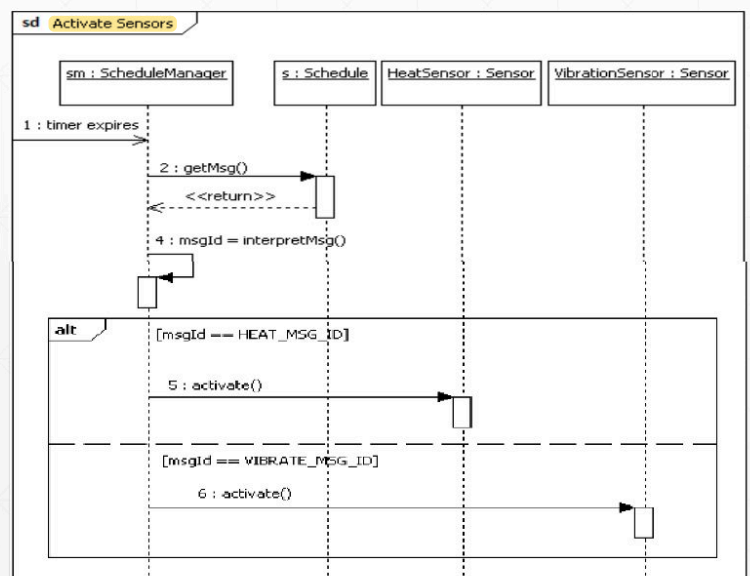
A Manager class holds a field of the concrete type Worker and calls worker.work(). Which principle does this violate? Draw or describe the fix.

Answer: The Dependency Inversion Principle: the high-level Manager depends on the concrete low-level Worker. Fix: introduce interface IWorker { work(); }; Worker, RobotWorker, etc. implement it; Manager holds an IWorker and receives it through setWorker(IWorker). In UML: Manager → IWorker (dependency/association), Worker and RobotWorker → IWorker (realization).

SLIDE 15 / 24 Modeling Internal Behavior of

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**



NOTES

- 1 Component design models both the **internal structure** and the **behaviour** of a component.
- 2 The internal **structure** is modelled with a **class diagram** (as in Lecture 2).
- 3 The internal **behaviour** is modelled with a **sequence diagram**.

4 Reading the "Activate Sensors" example:

1: timer expires

An outside event reaches the SchedulerManager (`sm`).

2: `getMsg()` / 4: `interpretMsg()`

It fetches the scheduled message from `s : Schedule` and interprets it (a self-call).

alt frame

A choice: if `msgId == HEAT_MSG_ID` it sends 5: `activate()` to the HeatSensor; if `msgId == VIBRATE_MSG_ID` it sends 6: `activate()` to the VibrationSensor. Only one branch runs.

EXAM ANGLE

Prediction

Q1

Which UML diagrams model a component's internal structure and its internal behaviour?

- A. Class diagram for structure, sequence diagram for behaviour
- B. Deployment diagram for structure, use case diagram for behaviour
- C. Package diagram for both
- D. Sequence diagram for structure, class diagram for behaviour

Answer: A. Class diagram for structure, sequence diagram for behaviour

SLIDE 16 / 24 Rules of Design

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.

NOTES

- 1 **Make sure the problem is well defined:** know all design criteria, requirements and constraints before starting the design.

- 2 **What comes before how:** at every level of abstraction, define the functions to be performed before choosing the structures that implement them.
- 3 **Separate orthogonal concerns:** don't connect what is independent (orthogonal = unrelated, changing for different reasons).

SLIDE 17 / 24 Rules of Design

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.

NOTES

- 1 **External before internal:** first treat the solution as a black box and decide how it interacts with its environment; then decide how the box is organised inside, usually as smaller black boxes refined the same way.
 - 2 **Keep it simple:** fancy designs are buggier, harder to implement and verify, and often slower. Problems that look complex are often simple problems huddled together; the designer's job is to separate them and solve each one.
-

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).

NOTES

- 1 **Work at multiple levels of abstraction:** good designers move quickly between the big picture and the details.
- 2 **Design for extensibility:** a good design is “open-ended”, easy to extend. Don’t introduce what is immaterial (unimportant) and don’t restrict what is irrelevant (unrelated): both make later extension harder.

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

NOTES

- 1 **Rapid prototyping:** before implementing, build a high-level prototype and check that the design criteria are met.
- 2 **Details depend on abstractions; abstractions don't depend on details:** the DIP stated as a rule of design.
- 3 **Common closure:** classes in a released component should change together; if one needs to change, they all probably do ("what affects one, affects all"). Group classes by the reasons they change.

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

NOTES

- 1 **Common reuse:** classes in a released component are reused together; you can't take part of a component.
- 2 **Stable dependencies:** dependencies between components must run in the direction of stability. The component depended on (dependee) must be more stable than the one depending on it.
- 3 **Stable abstractions:** the more stable a component is, the more it should consist of abstract classes; a completely stable component contains nothing but abstractions. That is why interfaces sit at the stable end of DIP diagrams.

EXAM ANGLE

Prediction

Q1

State the stable-dependencies rule and explain why stable components should be abstract.

Answer: Dependencies must point toward more stable components (the dependee is more stable than the depender). Stable components are hard to change because many others depend on them, so they should consist of abstractions that can be extended without modification.

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

NOTES

- 1 **Depth-first design:** perfecting one part early and only partially satisfying the requirements overall. Experience is the best cure.
- 2 **Directly refining the requirements specification** into a design: produces overly constrained, inefficient designs, because the structure copies the document instead of the problem.
- 3 **Failing to consider potential changes:** always design for extension and contraction (adding and removing features).
- 4 **Making the design too detailed:** over-constrains the implementation and leaves programmers no sensible choices.

EXAM ANGLE

Prediction

Q1

Name two common design mistakes and their consequences.

Answer: Any two: depth-first design (only partially satisfies requirements); directly refining the requirements specification (overly constrained, inefficient design); failing to consider potential changes (hard to extend); making the design too detailed (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**

NOTES

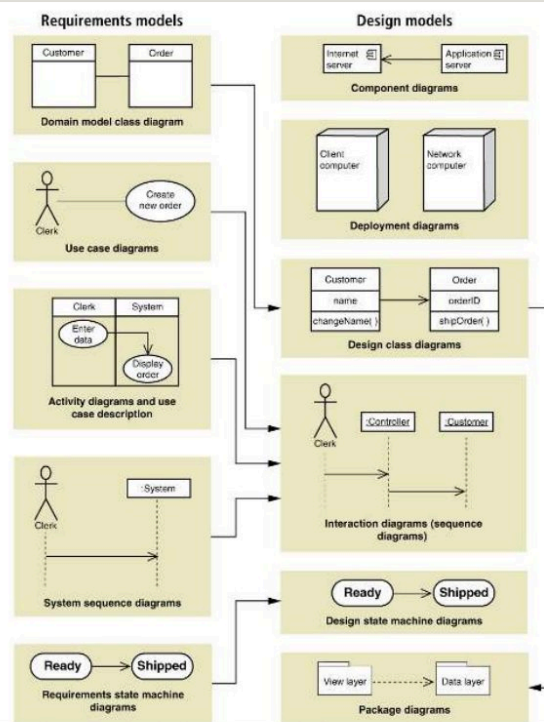
- 1 Recap: the internal-component design principles (the slide lists SRP and DIP; this lecture added ISP and DIP to Lecture 2's SRP, OCP and LSP).
- 2 Internal behaviour is modelled with sequence diagrams, and the rules of design guide every decision.
- 3 Next chapter: **creational design patterns**.

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.

NOTES

- 1 The rules from slides 16–20 in one list. Problem and requirements first; **what before how**; separate independent concerns; design external behaviour first.
- 2 **Keep it simple**; work at several abstraction levels; design for extensibility; avoid the unnecessary; prototype early.
- 3 **Depend on abstractions, not details** (DIP).
- 4 Component rules: classes that change together and are reused together belong in one component; **dependencies go toward stability**; stable components should be abstract.

Analysis Models to Design Models



NOTES

- 1 How analysis (requirements) models turn into design models. Left: the analysis models describe the problem.
- 2 The **domain model class diagram** (Customer, Order) becomes the **design class diagrams** (with attributes and operations such as `changeName()`, `shipOrder()`); use case diagrams and activity diagrams feed the component and interaction (sequence) diagrams.
- 3 System sequence diagrams become detailed **interaction diagrams**; requirements state machines (Ready → Shipped) become design state machines; and package diagrams organise the result into layers. Each design model traces back to an analysis model.

Original slides from the SE322 lecture deck, with study notes. Check exact wording against the [original deck](#). Notes and questions are study aids. Made by Shoug Alomran.