

5. D - Dependency Inversion Principle (DIP)

What is the Dependency Inversion Principle (DIP)?

Code Example: Violating DIP

Problems with the Above Code

Code Example: Follows DIP

Key Benefits of the Refactored Code

Summary

▼ Resources

- Video → [1. SOLID Principles with Easy Examples \(Hindi\) | OOPs SOLID Principles - Low Level Design](#)

What is the Dependency Inversion Principle (DIP)?

*The Dependency Inversion Principle (DIP) states that “**high-level components should not depend on low-level components directly; instead, they should depend on abstractions**”.*

In other words, classes should depend on interfaces rather than concrete classes. Additionally, abstractions should not depend on details; details should depend on abstractions.

Code Example: Violating DIP

Utility Classes

```
1 public interface Keyboard {
2     void getSpecifications();
3 }
4 public interface Mouse {
5     void getSpecifications();
6 }
7
8 // Low-level module - concrete implementation
9 public class WiredKeyboard implements Keyboard {
10     private final String connectionType;
11     private final String company;
12     private final String modelVersion;
13     private final String color;
14
15     public WiredKeyboard(String connectionType, String company,
16 String modelVersion, String color) {
17         this.connectionType = connectionType;
18         this.company = company;
19         this.modelVersion = modelVersion;
20         this.color = color;
21     }
22
23     public void getSpecifications() {
24         System.out.println("====> Wired Keyboard");
25         System.out.println("Connection Type: " + connectionType);
26         System.out.println("Company: " + company);
27         System.out.println("Model Version: " + modelVersion);
28         System.out.println("Color: " + color);
29     }
30 }
31
32 // Low-level module - concrete implementation
33 public class BluetoothKeyboard implements Keyboard {
```

```

33     private final String connectionType;
34     private final String company;
35     private final String modelVersion;
36     private final String color;
37
38     public BluetoothKeyboard(String connectionType, String company,
String modelVersion, String color) {
39         this.connectionType = connectionType;
40         this.company = company;
41         this.modelVersion = modelVersion;
42         this.color = color;
43     }
44
45     public void getSpecifications() {
46         System.out.println("==> Bluetooth Keyboard");
47         System.out.println("Connection Type: " + connectionType);
48         System.out.println("Company: " + company);
49         System.out.println("Model Version: " + modelVersion);
50         System.out.println("Color: " + color);
51     }
52 }
53
54 // Low-level module - concrete implementation
55 public class WiredMouse implements Mouse {
56     private final String connectionType;
57     private final String company;
58     private final String modelVersion;
59     private final String color;
60
61     public WiredMouse(String connectionType, String company, String
modelVersion, String color) {
62         this.connectionType = connectionType;
63         this.company = company;
64         this.modelVersion = modelVersion;
65         this.color = color;
66     }
67
68     public void getSpecifications() {
69         System.out.println("==> Wired Mouse");
70         System.out.println("Connection Type: " + connectionType);
71         System.out.println("Company: " + company);
72         System.out.println("Model Version: " + modelVersion);
73         System.out.println("Color: " + color);
74     }
75 }
76
77 // Low-level module - concrete implementation
78 public class BluetoothMouse implements Mouse {
79     private final String connectionType;
80     private final String company;
81     private final String modelVersion;
82     private final String color;
83
84     public BluetoothMouse(String connectionType, String company,
String modelVersion, String color) {
85         this.connectionType = connectionType;
86         this.company = company;
87         this.modelVersion = modelVersion;
88         this.color = color;
89     }
90
91     public void getSpecifications() {
92         System.out.println("==> Bluetooth Mouse");
93         System.out.println("Connection Type: " + connectionType);
94         System.out.println("Company: " + company);
95         System.out.println("Model Version: " + modelVersion);
96         System.out.println("Color: " + color);
97     }
98 }
99
100 // Low-level module - concrete implementation

```

```

101 public class BluetoothMouse implements Mouse {
102     private final String connectionType;
103     private final String company;
104     private final String modelVersion;
105     private final String color;
106
107     public BluetoothMouse(String connectionType, String company,
String modelVersion, String color) {
108         this.connectionType = connectionType;
109         this.company = company;
110         this.modelVersion = modelVersion;
111         this.color = color;
112     }
113
114     public void getSpecifications() {
115         System.out.println("==> Bluetooth Mouse");
116         System.out.println("Connection Type: " + connectionType);
117         System.out.println("Company: " + company);
118         System.out.println("Model Version: " + modelVersion);
119         System.out.println("Color: " + color);
120     }
121 }

```

```

1 // VIOLATION OF DIP
2 // High-level module directly depending on low-level module
3 public class MacBook {
4     private final WiredKeyboard keyboard;
5     private final WiredMouse mouse;
6
7     // Direct dependency on concrete class
8     public MacBook(WiredKeyboard wiredKeyboard, WiredMouse wiredMouse)
{
9         keyboard = wiredKeyboard; // Tight coupling
10        mouse = wiredMouse; // Tight coupling
11    }
12
13    public Mouse getMouse() {
14        return mouse;
15    }
16
17    public Keyboard getKeyboard() {
18        return keyboard;
19    }
20 }
21
22
23 public class DemoViolation {
24     public static void main(String[] args) {
25
26         // create keyboard and mouse objects
27         WiredKeyboard wiredKeyboard = new WiredKeyboard("USB", "Dell",
"F602", "Grey");
28         WiredMouse wiredMouse = new WiredMouse("USB", "Dell", "F602",
"Grey");
29         BluetoothKeyboard bluetoothKeyboard = new
BluetoothKeyboard("Bluetooth", "Logitech", "G102", "Black");
30         BluetoothMouse bluetoothMouse = new
BluetoothMouse("Bluetooth", "Logitech", "G102", "Black");
31
32         // create macbook
33         MacBook macBookWithWiredParts = new MacBook(wiredKeyboard,
wiredMouse);
34         macBookWithWiredParts.getKeyboard().getSpecifications();
35         macBookWithWiredParts.getMouse().getSpecifications();
36
37         // create macbook with bluetooth keyboard and mouse
38         // MacBook macBookWithBluetoothParts = new
MacBook(bluetoothKeyboard, bluetoothKeyboard);
39         // cannot create macbook with bluetooth keyboard and mouse
because

```

```

40     // macbook depends on wired keyboard and mouse - tight
    coupling - violation of DIP
41     }
42 }

```

Problems with the Above Code

- The `MacBook` is tightly coupled to the `WiredKeyboard` and the `WiredMouse`.
- Cannot create `MacBook` objects with different parts without modifying the `MacBook` class.
- Difficult to test `MacBook` in isolation
- High-level module `MacBook` depends on low-level module `WiredKeyboard` and the `WiredMouse`.

Code Example: Follows DIP

Utility Code remains the same

```

1  // Following DIP
2  // High-level module uses abstraction
3  public class MacBook {
4      private final Keyboard keyboard;
5      private final Mouse mouse;
6
7      // Abstraction - defines contract
8      // Dependency injection through constructor
9      public MacBook(Mouse mouse, Keyboard keyboard) {
10         this.keyboard = keyboard; // Works with any kind of keyboard
    and mouse
11         this.mouse = mouse;
12     }
13
14     public Mouse getMouse() {
15         return mouse;
16     }
17
18     public Keyboard getKeyboard() {
19         return keyboard;
20     }
21 }
22
23 // Usage example - following DIP
24 public class DemoSolution {
25     public static void main(String[] args) {
26         // create keyboard and mouse objects
27         WiredKeyboard wiredKeyboard = new WiredKeyboard("USB", "Dell",
    "F602", "Grey");
28         WiredMouse wiredMouse = new WiredMouse("USB", "Dell", "F602",
    "Grey");
29         BluetoothKeyboard bluetoothKeyboard = new
    BluetoothKeyboard("Bluetooth", "Logitech", "G102", "Black");
30         BluetoothMouse bluetoothMouse = new
    BluetoothMouse("Bluetooth", "Logitech", "G102", "Black");
31
32         // Can easily create macbooks with different combinations of
    keyboard and mouse
33
34         // create macbook with wired keyboard and mouse
35         MacBook macBookWithWiredParts = new MacBook(wiredMouse,
    wiredKeyboard);
36         macBookWithWiredParts.getKeyboard().getSpecifications();
37         macBookWithWiredParts.getMouse().getSpecifications();
38
39         // create macbook with bluetooth keyboard and mouse
40         MacBook macBookWithBluetoothParts = new
    MacBook(bluetoothMouse, bluetoothKeyboard);
41         macBookWithBluetoothParts.getKeyboard().getSpecifications();
42         macBookWithBluetoothParts.getMouse().getSpecifications();

```

```
43     }  
44 }
```

Key Benefits of the Refactored Code

- `BluetoothKeyboard` and `BluetoothMouse` depend only on the respective `Keyboard` and `Mouse` abstractions.
- Any different `MacBook` parts can be injected without changing the `MacBook`.
- Easy to mock for testing.
- New `MacBook` parts can be added without touching existing code.

Summary

The solution demonstrates true dependency inversion because the high-level module (`MacBook`) and low-level modules (`BluetoothKeyboard` , `BluetoothMouse` , `WiredKeyboard` , `WiredMouse`) both depend on the same abstraction (`Keyboard` and `Mouse`), and the abstraction doesn't depend on any concrete implementation details.