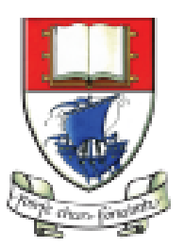
Dependency Inversion Principle (DIP)

Produced by:

Dr. Siobhán Drohan (sdrohan@wit.ie)

Eamonn de Leastar (edeleastar@wit.ie)

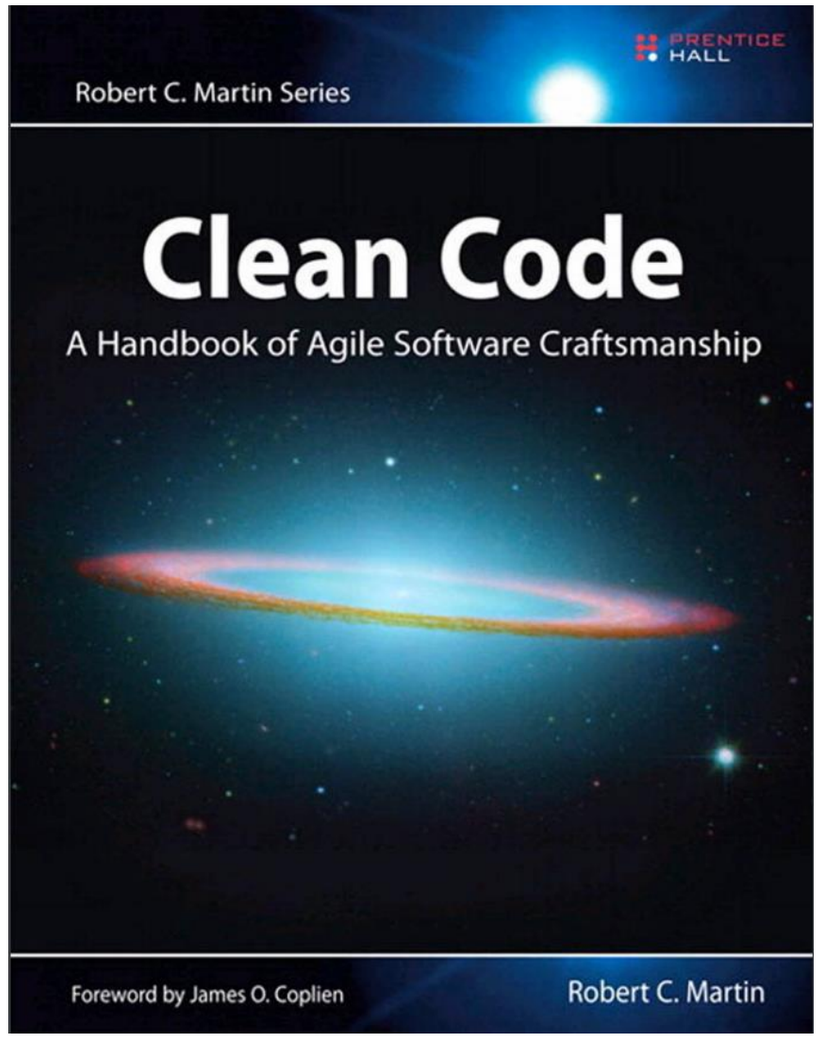


Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

SOLID Class Design Principles

In this talk, we will refer to the SOLID principles examples in this book.



SOLID → five principles for object-oriented class design i.e.

best guidelines for building a maintainable object-oriented system.

SOLID Class Design Principles

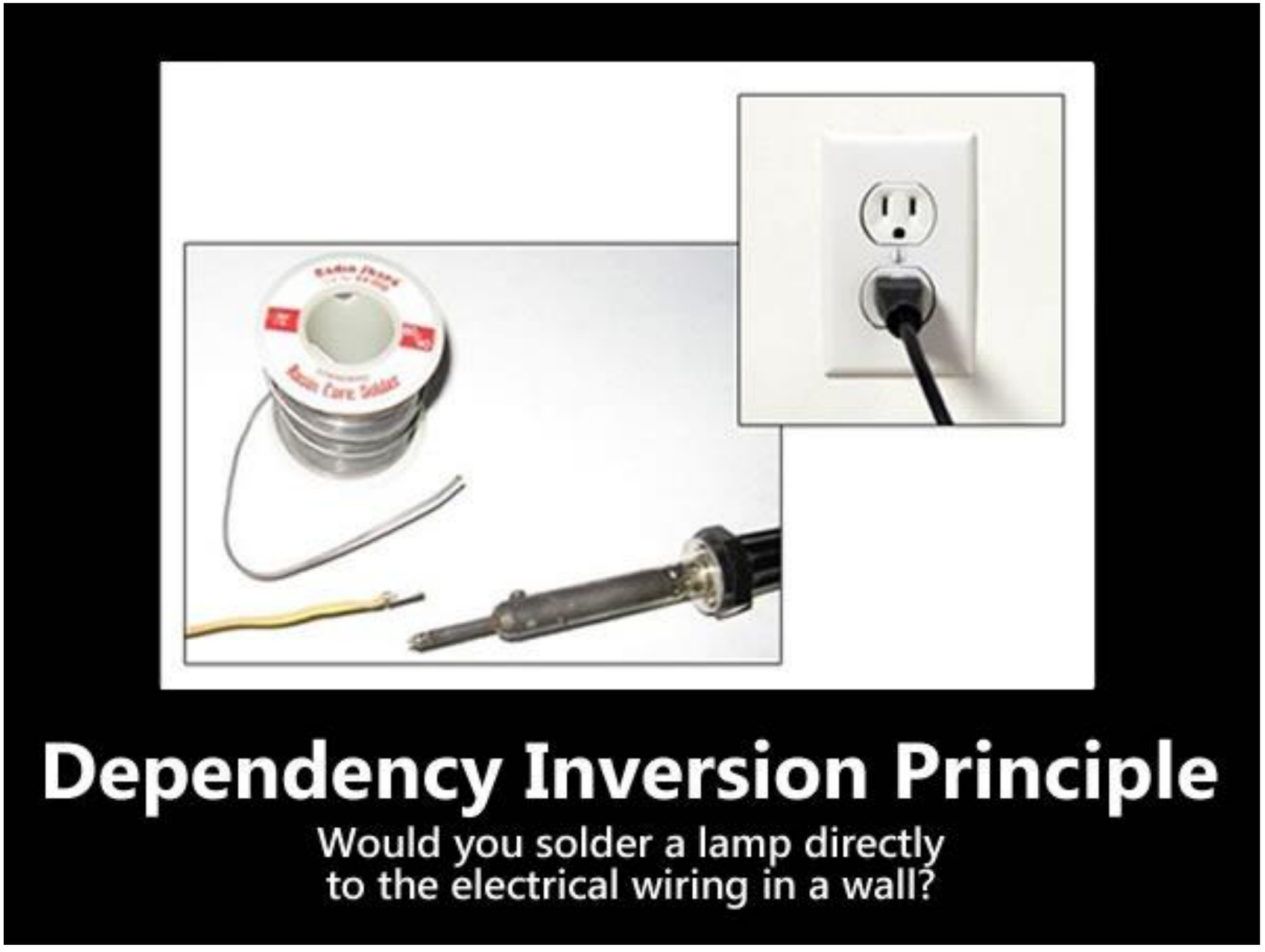
S Single Responsibility Principle (SRP). Classes should have one, and only one, reason to change. Keep your classes small and single-purposed.

O Open-Closed Principle (OCP). Design classes to be open for extension but closed for modification; you should be able to extend a class without modifying it. Minimize the need to make changes to existing classes.

L Liskov Substitution Principle (LSP). Subtypes should be substitutable for their base types. From a client's perspective, override methods shouldn't break functionality.

I Interface Segregation Principle (ISP). Clients should not be forced to depend on methods they don't use. Split a larger interface into a number of smaller interfaces.

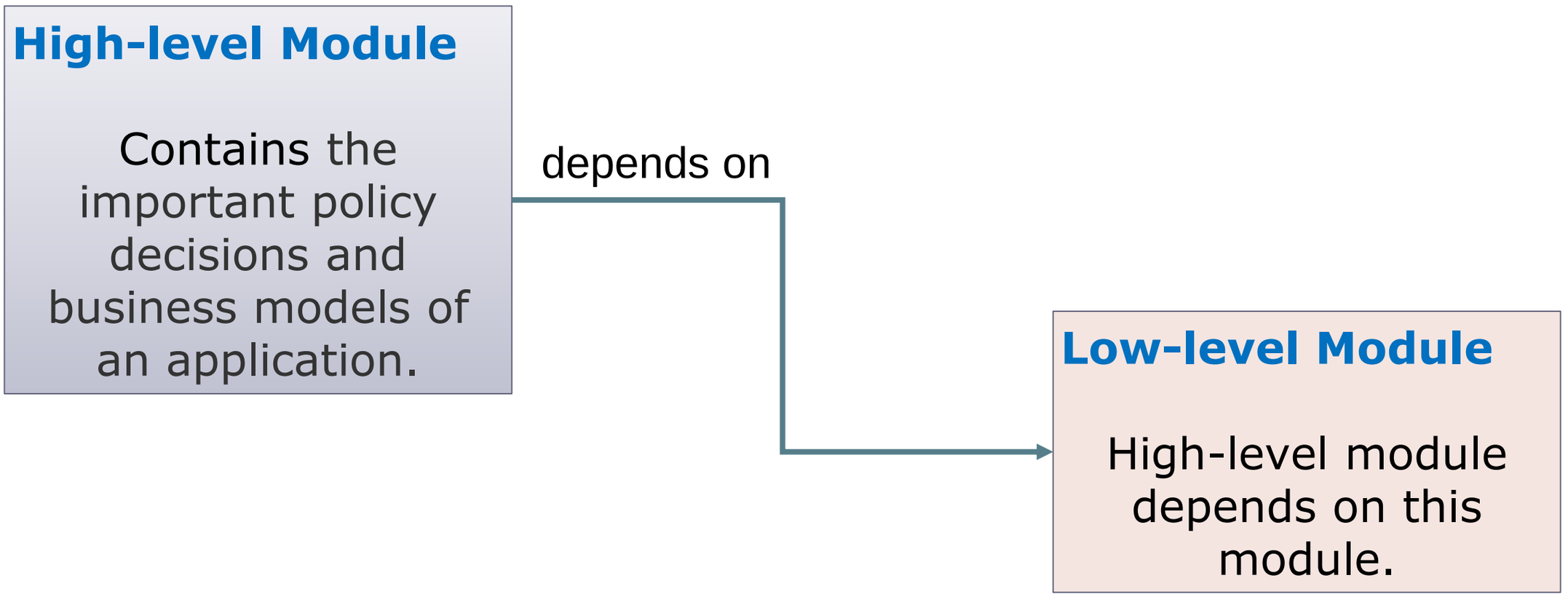
D **Dependency Inversion Principle (DIP). High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.**



https://zeroturnaround.com/rebellabs/object-oriented-design-principles-and-the-5-ways-of-creating-solid-applications/

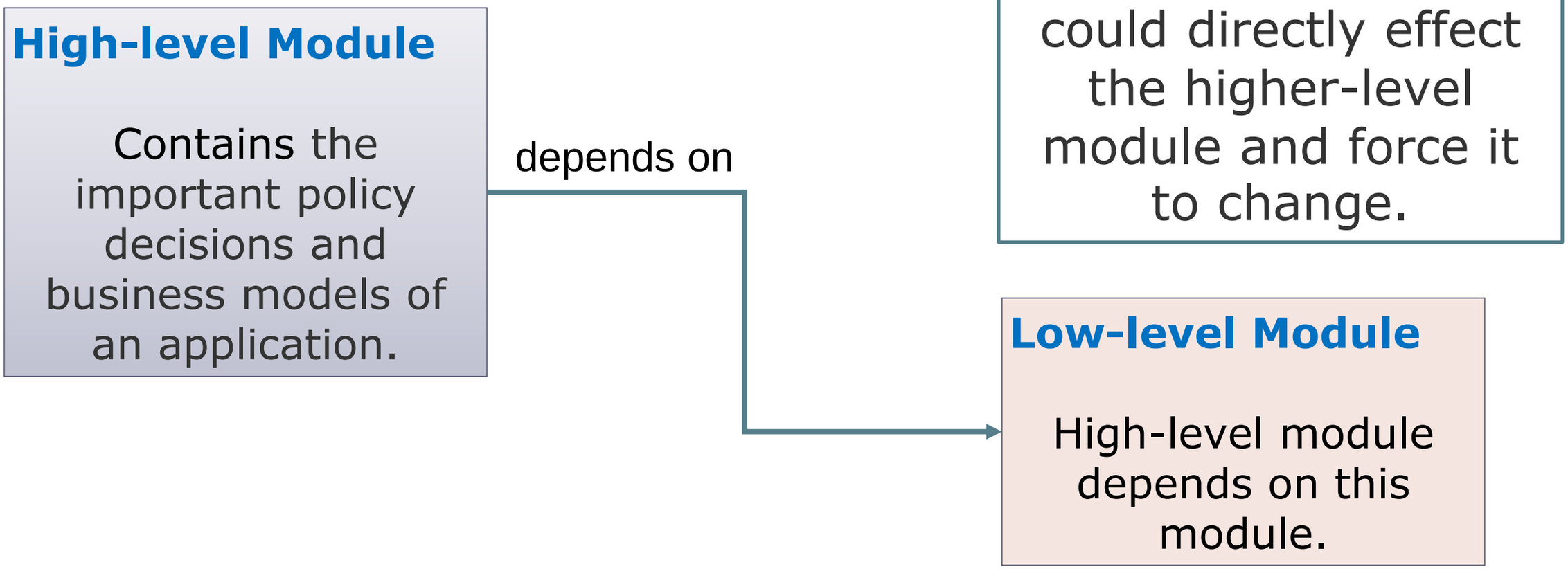
Dependency Inversion Principle (DIP) – Basic Idea

*Consider a **(bad!)** situation where a high-level module depends on a low-level module.*



Dependency Inversion Principle (DIP) – Basic Idea

*Consider a **(bad!)** situation where a high-level module depends on a low-level module.*



Dependency Inversion Principle (DIP) – Basic Idea

*Consider a **(bad!)** situation where a high-level module depends on a low-level module.*

High-level Module

Contains the important policy decisions and business models of an application.

depends on

Low-level Module

High-level module depends on this module.

Changes to the lower-level module could directly effect the higher-level module and force it to change.

Absurd!

Dependency Inversion Principle (DIP) – Basic Idea

*Consider a **(bad!)** situation where a high-level module depends on a low-level module.*

- It is the high-level, policy-setting modules that ought to be influencing the low-level detailed modules.
- The modules that contain the high-level business rules should take precedence over, and be independent of, the modules that contain the implementation details.

High-level modules simply <u>should not</u> depend on low-level modules in any way.

Dependency Inversion Principle (DIP) – Basic Idea

And taking this idea one step further…

High-level modules simply <u>should not</u> depend on low-level modules in any way.

Dependency Inversion Principle (DIP) – Basic Idea

High-level Module

We want to be able to re-use these!

If high-level modules are independent of low-level modules, the high-level modules can be easily reused.

Low-level Module

We are fairly good at reusing these e.g. utilities, libraries, components, etc.

Dependency Inversion Principle (DIP) – Basic Idea

High-level Module

We want to be able to re-use these!

If high-level modules are independent of low-level modules, the high-level modules can be easily reused.

Low-level Module

We are fairly good at reusing these e.g. utilities, libraries, components, etc.

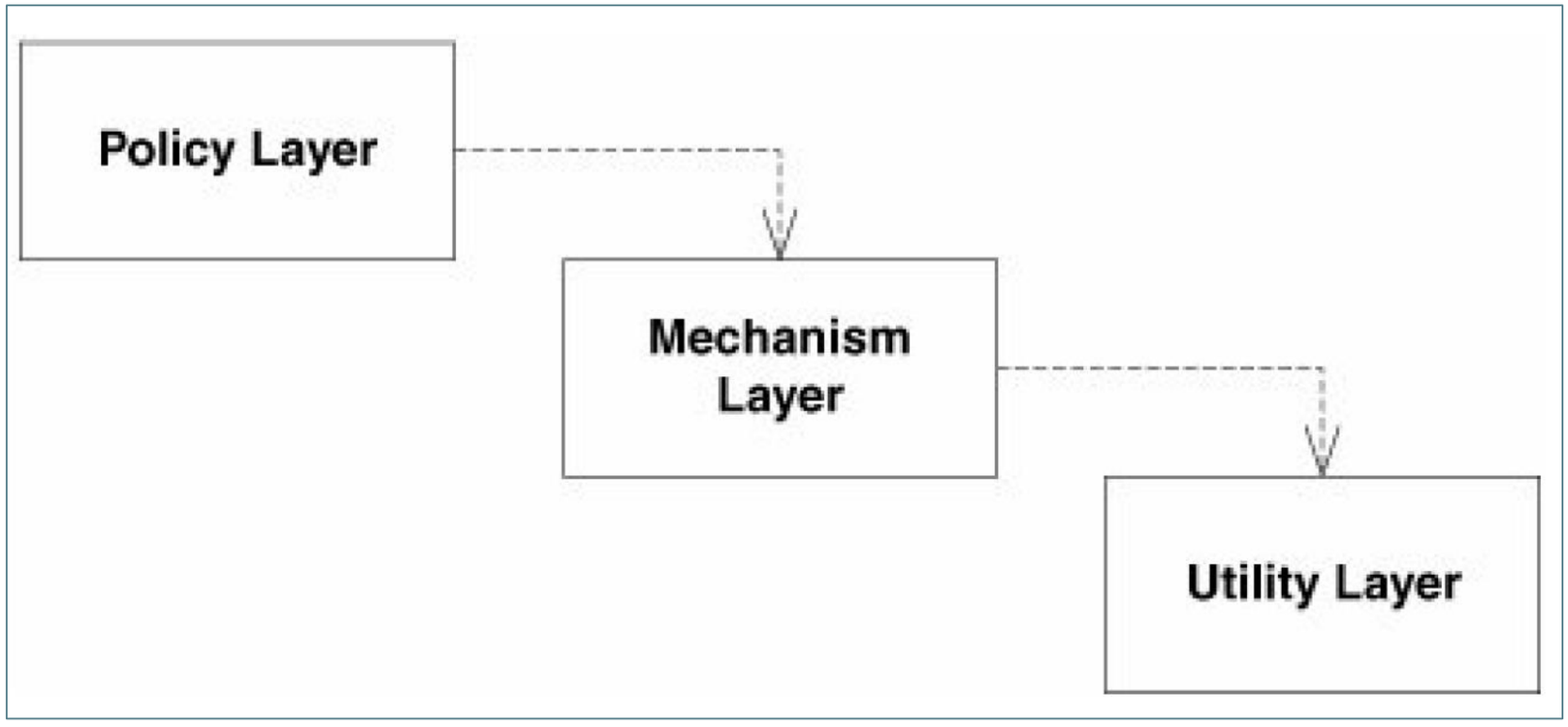
A. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*

B. *Abstractions should not depend upon details. Details should depend upon abstractions. (more on this later).*

Two Layering Approaches

Naïve and Inverted

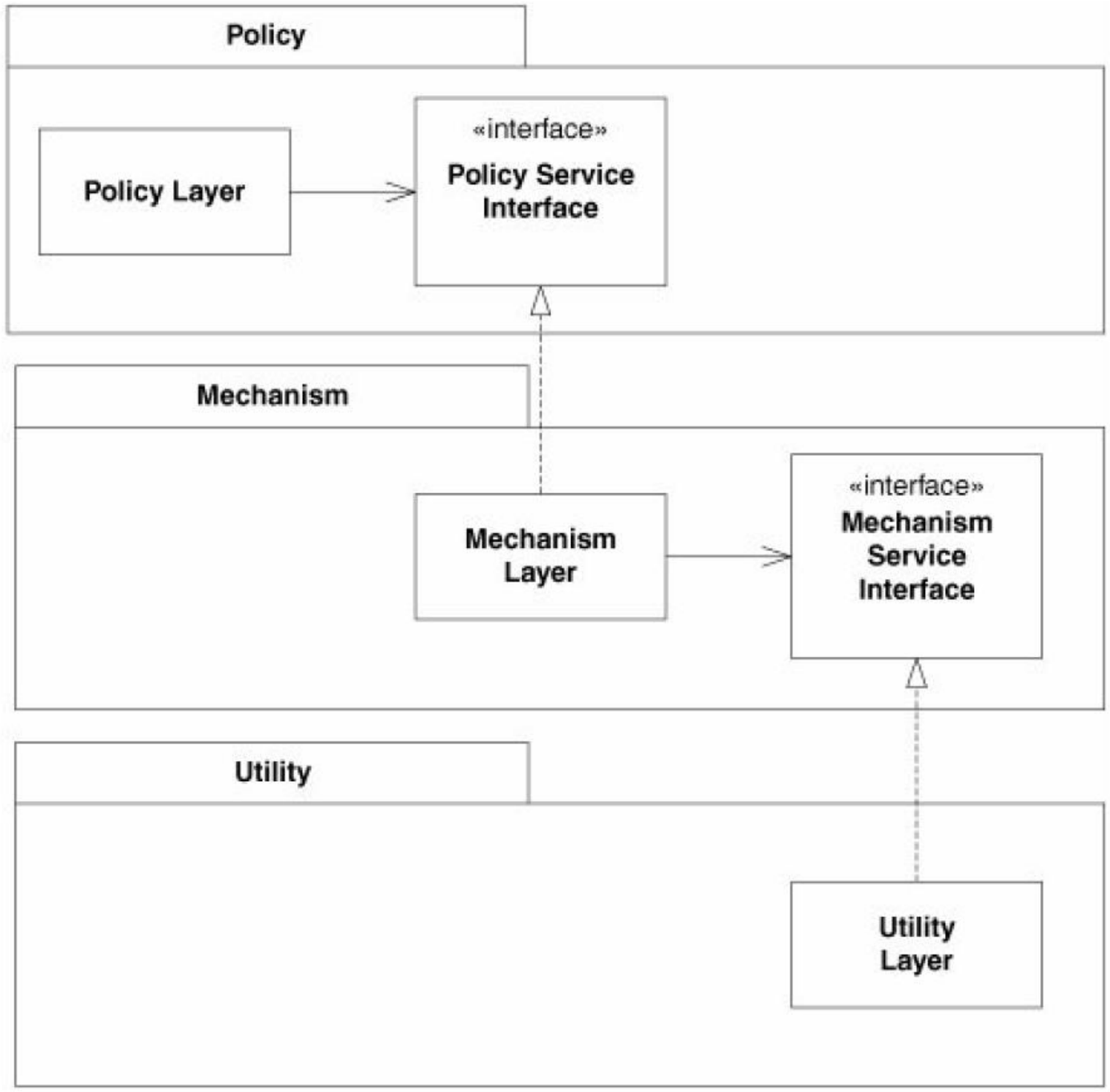
DIP – Naïve Layering



The high-level `Policy` layer uses a lower-level `Mechanism` layer, which in turn uses a `Utility` layer.

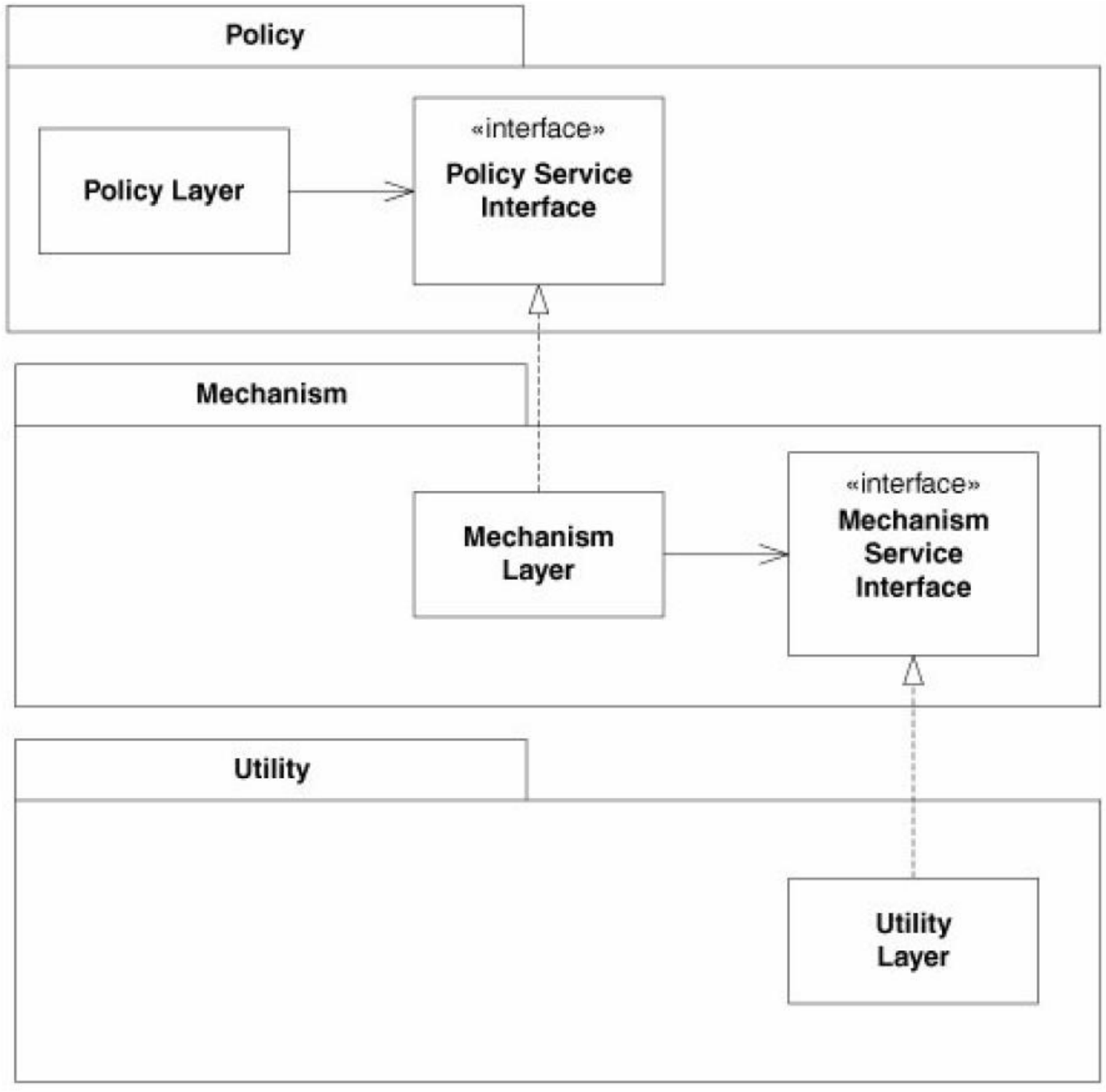
Problem: the `Policy` layer is sensitive to changes all the way down in the `Utility` layer.

DIP – Inverted Layering (more appropriate!)



Each upper-level layer declares an abstract interface for the services it needs.

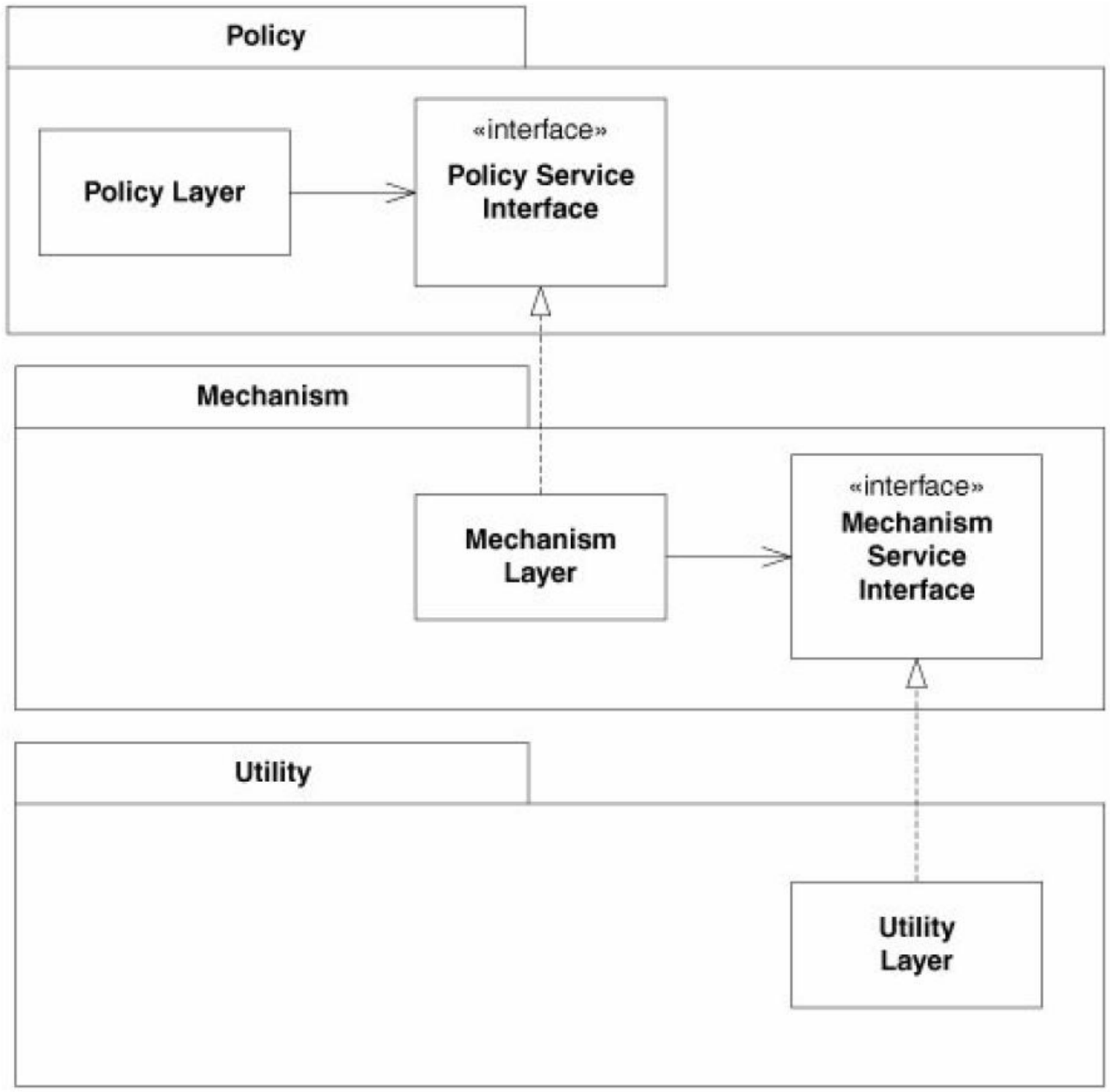
DIP – Inverted Layering (more appropriate!)



The lower-level layers are then realized from these abstract interfaces.

Each higher-level class uses the next lowest layer through the abstract interface.

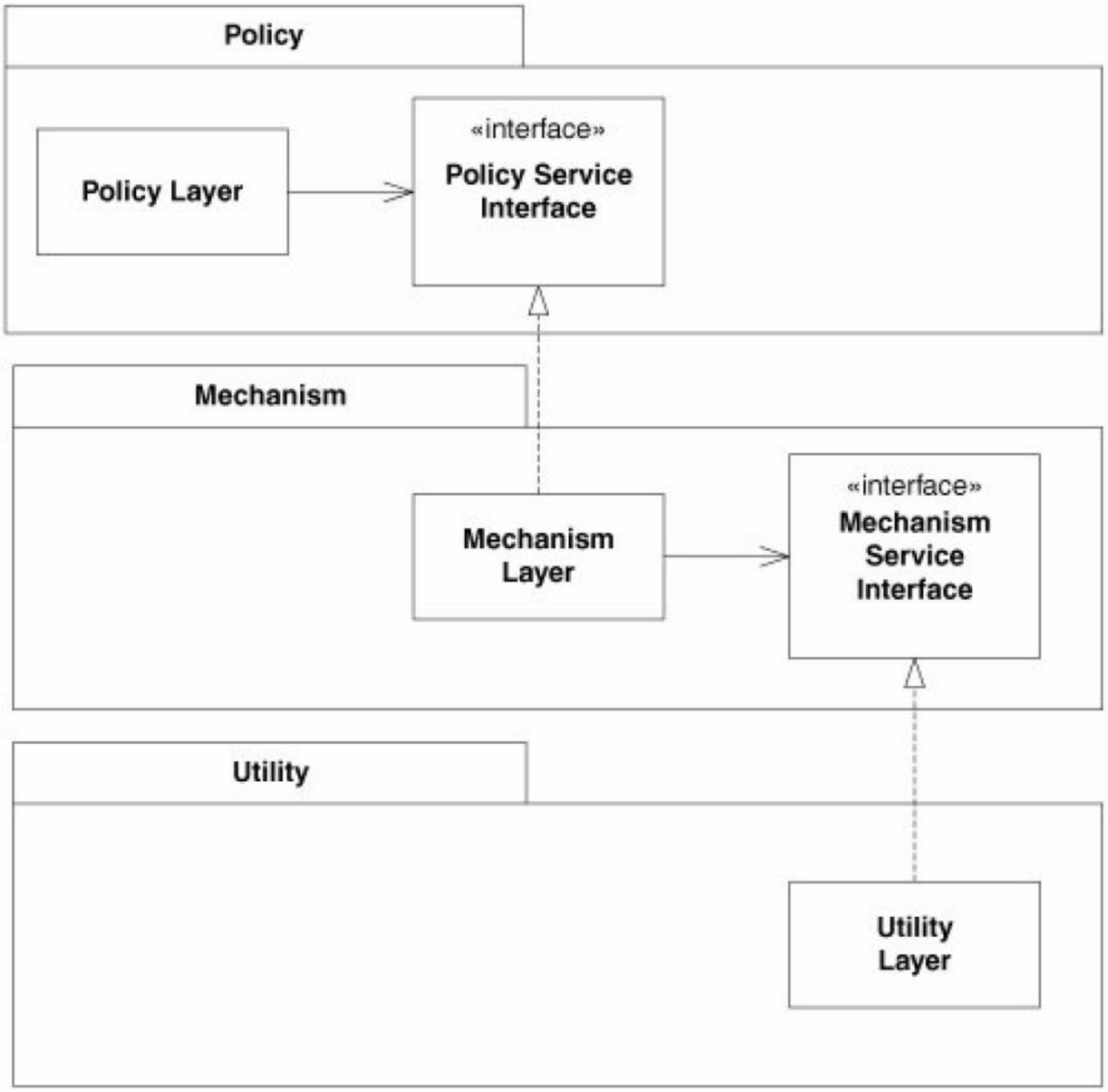
DIP – Inverted Layering (more appropriate!)



Now, the upper layers do not depend on the lower layers.

Instead, the lower layers depend on abstract service interfaces *declared in* the upper layers.

DIP – Inverted Layering (more appropriate!)



PolicyLayer can be reused in any context that defines lower-level modules that conform to the PolicyService-Interface.

→ This is called Dependency Inversion.

Dependency inversion can be applied wherever one class sends a message to another.

Consider this simple example that violates DIP.

A Simple Example (that violates DIP)

A Simple Example (that violates DIP)

The `Button` object, receives a `Poll` message and determines whether the user has pressed the button.

A Simple Example (that violates DIP)

The `Button` object, receives a `Poll` message and determines whether the user has pressed the button.

Button messages the lamp. On receiving a:

- `TurnOn` message, the `Lamp` object turns on a light.
- `TurnOff` message, it turns off that light.

A Simple Example (that violates DIP)

Code Example:

```
public class Button
{
   private Lamp lamp;

   public void Poll()
   {
      if (/*some condition*/)
         lamp.TurnOn();
   }
}
```

A Simple Example (that violates DIP)

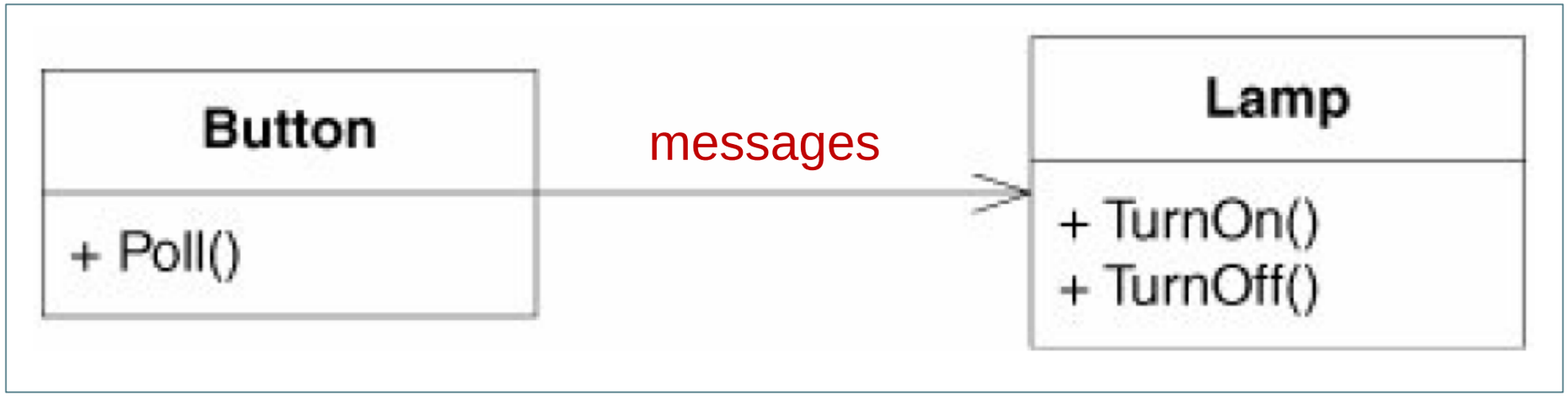
Naïve Implementation!

The Button class depends directly on the Lamp class.

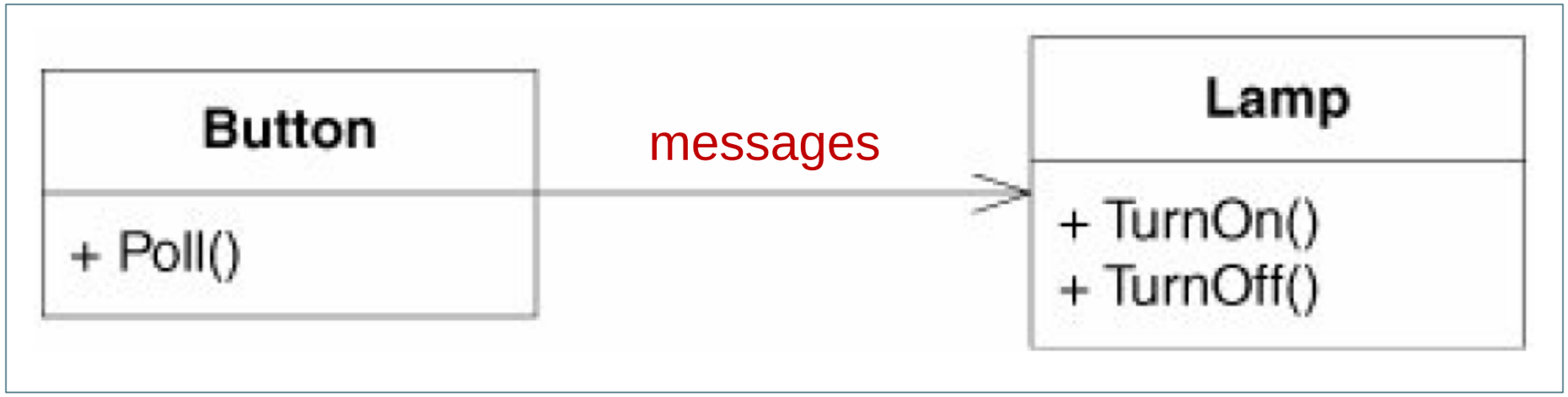
A Simple Example (that violates DIP)

DEPENDENCY: This dependency implies that `Button` will be affected by changes to `Lamp`.

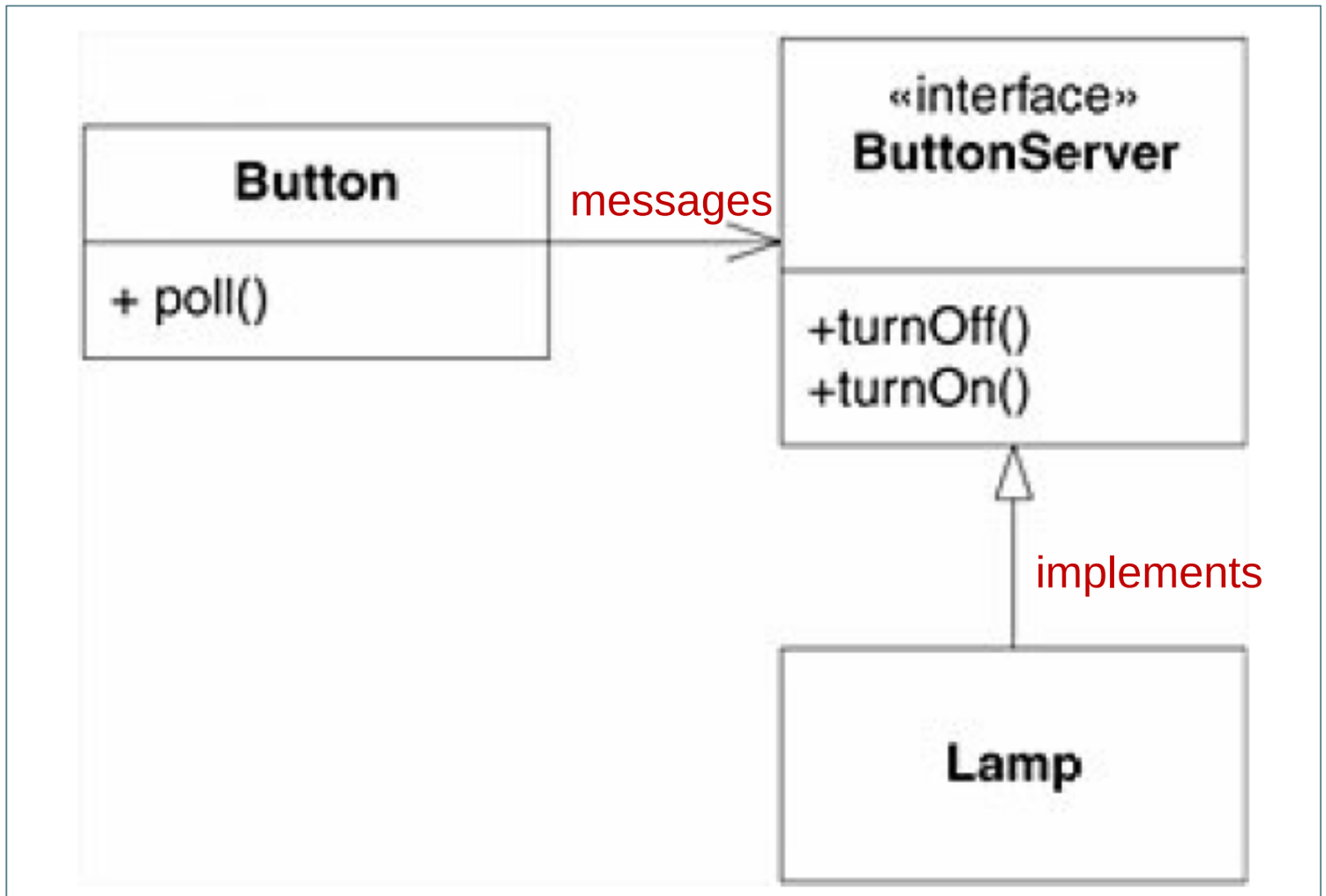
REUSE: Also, it will not be possible to reuse `Button` to control, say, a `Motor` object. In this model, `Button` objects control `Lamp` objects and *only* `Lamp` objects.



Let’s now invert this dependency on Lamp and see what happens!

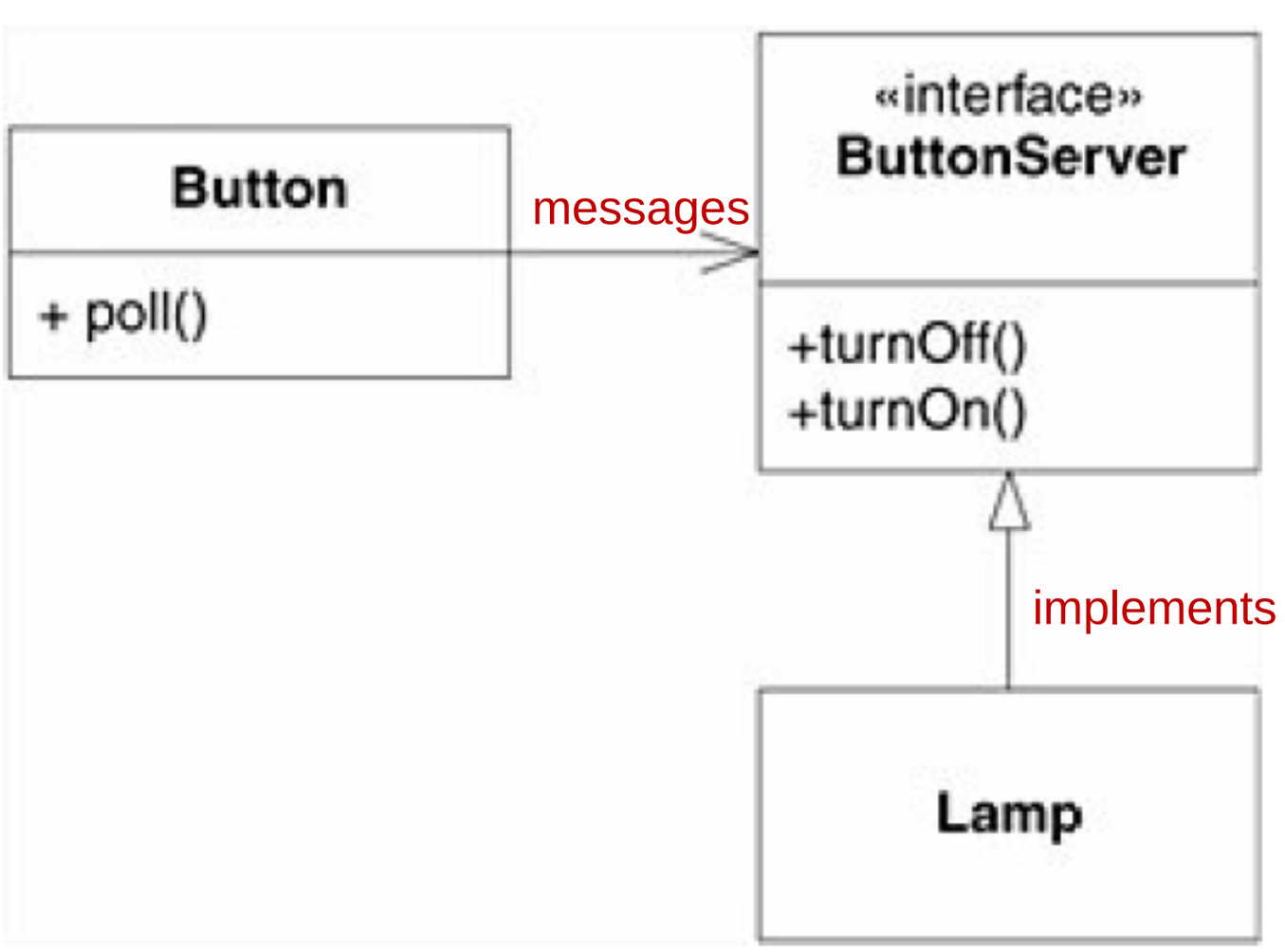

Button
+ Poll()
messages
Lamp
+ TurnOn()
+ TurnOff()

becomes


Button
+ poll()
messages
«interface»
ButtonServer
+turnOff()
+turnOn()
implements
Lamp

`Button` now holds an association to something called a `ButtonServer`, which provides the interfaces that `Button` can use to turn ***something*** on or off.

`Button` can now control ***anything*** implementing `ButtonServer` → flexibility and reuse!

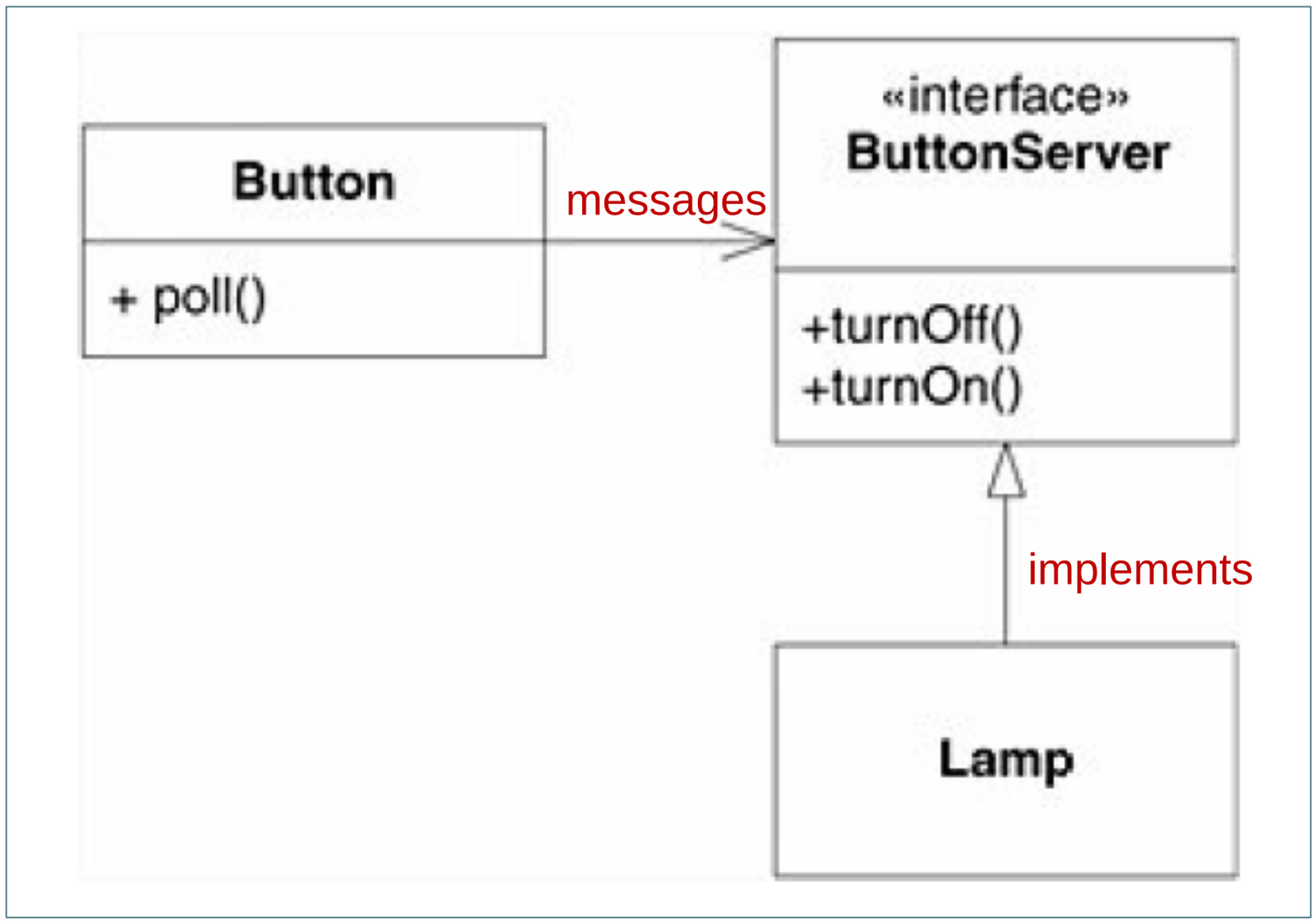


`Button` now holds an association to something called a `ButtonServer`, which provides the interfaces that `Button` can use to turn ***something*** on or off.

`Button` can now control ***anything*** implementing `ButtonServer` → flexibility and reuse!

`Lamp` implements the `ButtonServer` interface.

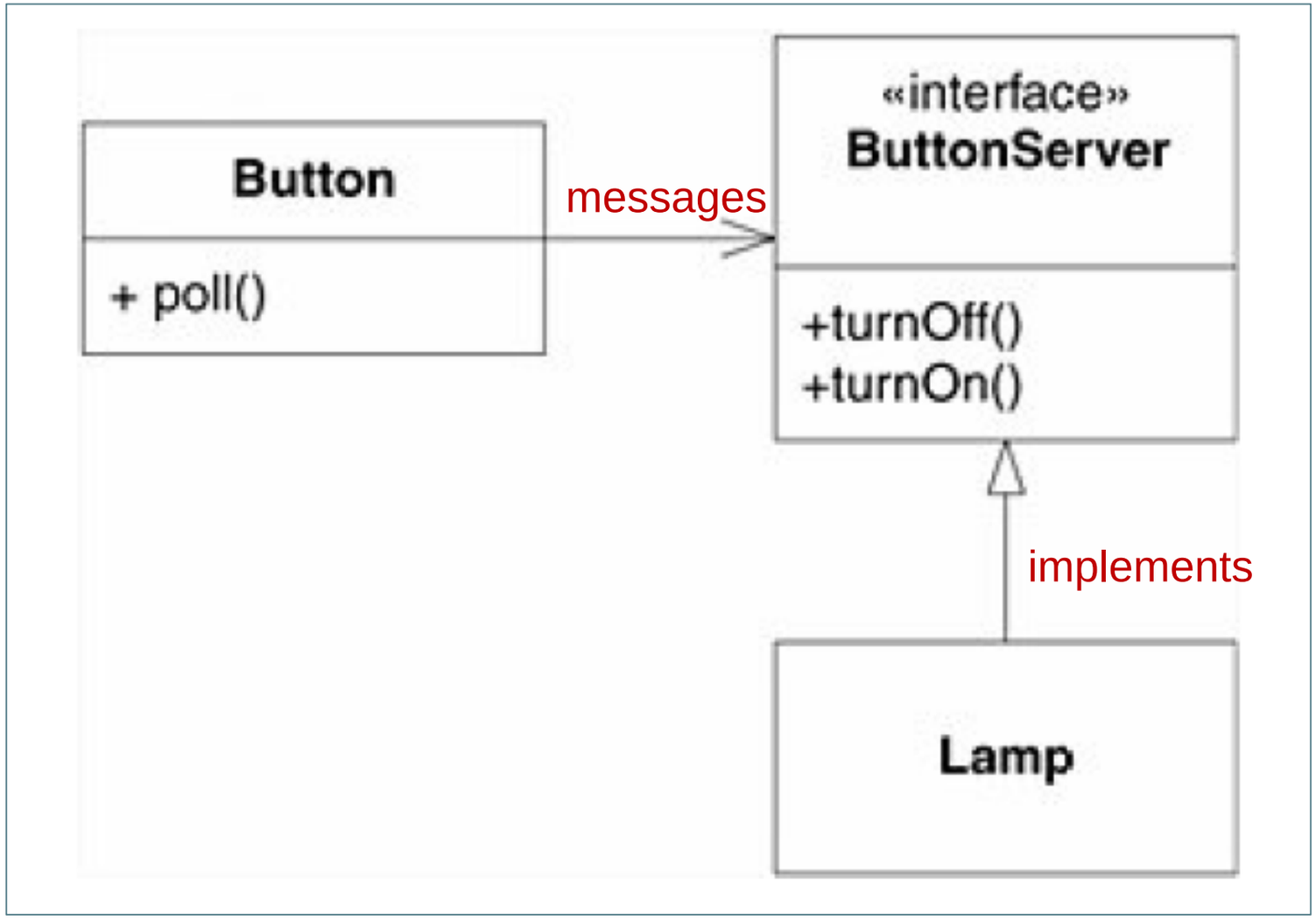
`Lamp` is now doing the **depending** rather than being depended on.



A. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*

B. *Abstractions should not depend upon details. Details should depend upon abstractions.*

Dependency Inversion Principle (DIP)



A. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*

B. *Abstractions should not depend upon details. Details should depend upon abstractions.*

This approach is critically important for the construction of code that is resilient to change.

Since abstractions and details are isolated from each other, the code is much easier to maintain.