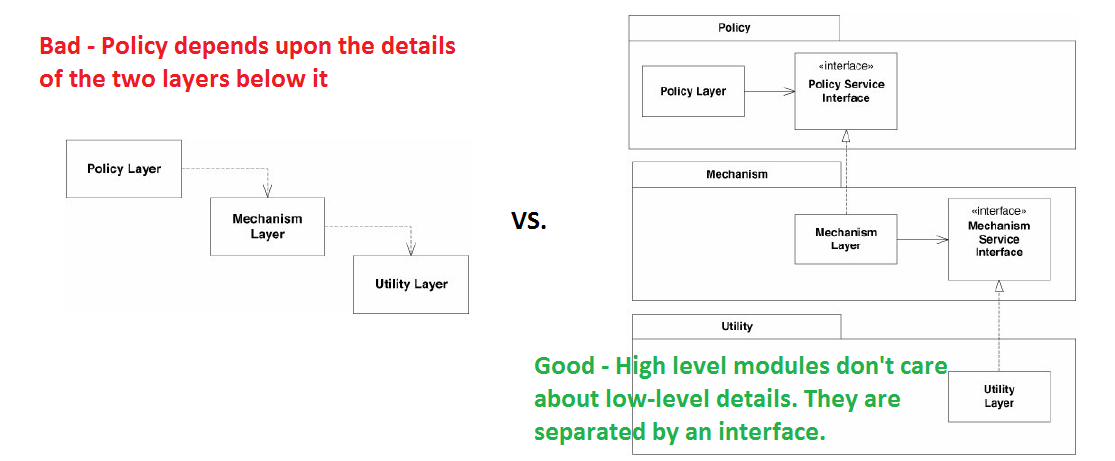
# Dependency-Inversion Principle

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend upon details. Details should depend upon abstractions

In brief, high-level policy should not be polluted by low level implementation details.

*Abstract Example:*

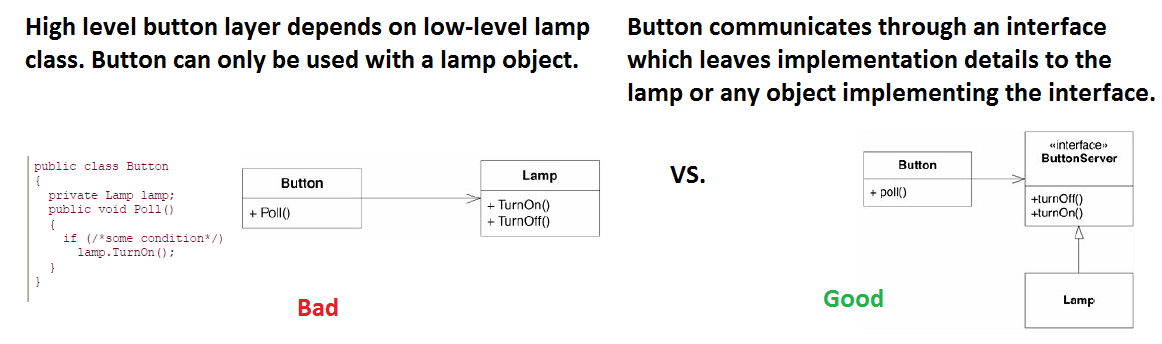


When the high-level modules are independent of the low-level modules, the high-level modules can be reused quite simply and be easily mocked for unit testing.

*Detailed example:*

*The Button object senses the external environment. On receiving the Poll message, the Button object determines whether a user has "pressed" it. It doesn't matter what the sensing mechanism is – it simply detects that a user has either activated or deactivated it.*

*The Lamp object affects the external environment. On receiving a TurnOn message, the Lamp object illuminates a light of some kind. On receiving a TurnOff message, it extinguishes that light. The physical mechanism is unimportant.*



*In the “good” portion of the above image, Button now holds an association to something called a ButtonServer, which provides the interface that Button can use to turn something on or off. Lamp implements the ButtonServer interface. Thus, Lamp is now doing the depending rather than being depended on. Because Button no longer depends on Lamp, Lamp can be swapped with an LED light, laser in a laser printer, or a mocked-up testing lamp.*

Source: (ch. 11, “Agile Principle, Patterns, and Practices”)

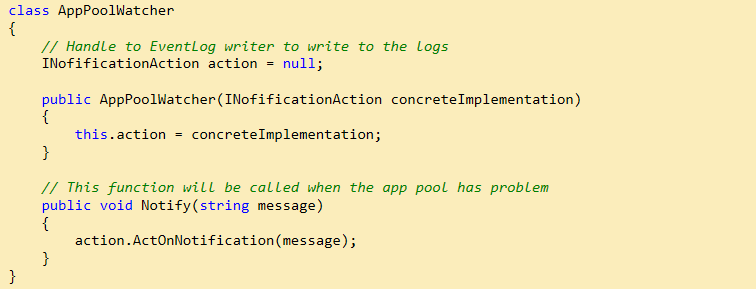
# Dependency Injection

Providing the objects that an object needs (its dependencies), rather than having it construct them itself.

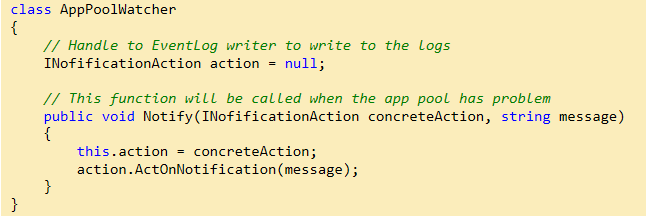
* reduce the coupling between classes
* move the binding of abstraction and concrete implementation out of the dependent class

1. **Constructor Injection** – pass the concrete class object to an interface field via constructor

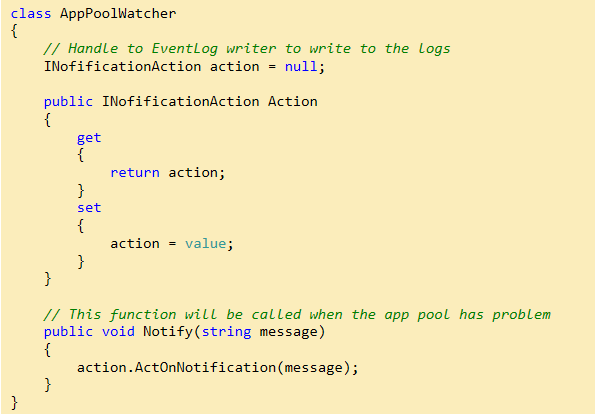
*Example:*

1. **Method injection** – passing the concrete dependency to an interface field via method parameter

*Example:*



1. **Property injection** - passing the concrete dependency to an interface field via class property



# Test Driven Design (TDD)

**Three Laws of TDD:**

First Law **-** You may not write production code until you have written a failing unit test.

Second Law **-** You may not write more of a unit test than is sufficient to fail, and not compiling

is failing.

Third Law **-** You may not write more production code than is sufficient to pass the currently.

failing test.

* These laws lock you into a cycle where tests are written before the production code
* TDD shifts the developer’s perspective to force modular production code that is written in a testable fashion
* TDD ensures virtually all our production code is covered by tests

Source: (“pg. 122-123, “Clean Code”)