**Dependency Inversion Principle**

The dependency inversion principle refers to a specific form of decoupling software modules.

The principle states:

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

2.   Abstractions should not depend on details. Details should depend on abstractions.

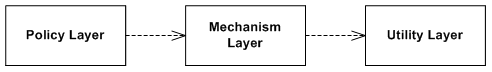
By dictating that both high-level and low-level objects must depend on the same abstraction this design principle inverts the way some people may think about object-oriented programming.

The idea behind points 1 and 2 of this principle is that when designing the interaction between a high-level module and a low-level one, the interaction should be thought of as an abstract interaction between them. This not only has implications on the design of the high-level module, but also on the low-level one: the low-level one should be designed with the interaction in mind and it may be necessary to change its usage interface.

# Traditional layers pattern

When the discovered abstract interaction schema(s) between two modules is/are generic and generalization makes sense, this design principle also leads to the following dependency inversion coding pattern.

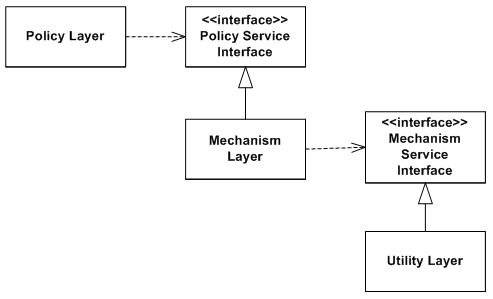
In conventional application architecture, higher-level components depend directly upon lower-level components to achieve some task. This dependency upon lower-level components limits the reuse opportunities of the higher-level components.



The goal of the dependency inversion pattern is to avoid this highly coupled distribution with the mediation of an abstract layer, and to increase the re-usability of higher/policy layers.

# Dependency inversion pattern

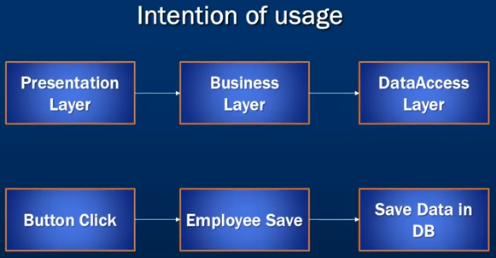
With the addition of an abstract layer, both high- and lower-level layers reduce the traditional dependencies from top to bottom. Both layers should depend on abstractions that draw the behavior needed by higher-level layers.



In a direct application of dependency inversion, the abstracts are owned by the upper/policy layers. This architecture groups the higher/policy components and the abstractions that define lower services together in the same package. The lower-level layers are created by inheritance/implementation of these abstract classes or interfaces.

# Example

**Problem**



Here we are coupling different layers and any further changes are complicated

BusinessLogicLayer is directly dependent on lower level DATAAccessLayer and its hard to perform any unit test on BusinessLogicLayer as both are coupled.

class BusinessLogicLayer {

DATAAccessLayer DAL;

Public:

BusinessLogicLayer { DAL = new DATAAccessLayer(); }

void save(Object details) { DAL.save(details); }

};

class DATAAccessLayer {

public:

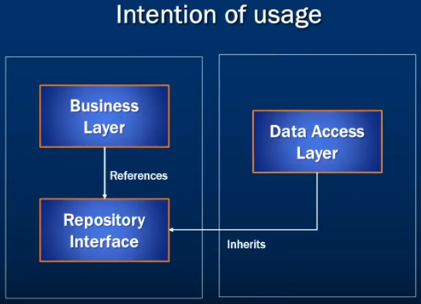
void save (Object details) {}

};

**Solution**

Based on DIP we apply an absraction to decouple these layers.

Inoredr to achieve this we introduce an interface that acess abstraction so bith these modules are decoupled.



class BusinessLogicLayer {

RepositaryLayer DAL;

public:

BusinessLogicLayer(RepositaryLayer repositorylayer) {

DAL = repositorylayer;

}

void save(object details) { DAL.save(details); }

};

class RepositaryLayer {

public:

void save(Object details) = 0;

};

class DATAAccessLayer : public RepositaryLayer {

public:

void save (Object details) {}

};

# END