## Dependency Inversion Principle (DIP)

This principle is about dependencies among components. The definition of DIP is given by Robert C. Martin is as follows:

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.

### **Understanding**

The principle says that high-level modules should depend on abstraction, not on the details, of low-level modules. In simple words, the principle says that there should not be a tight coupling among components of software and to avoid that, the components should depend on abstraction.

The terms Dependency Injection (DI) and Inversion of Control (IoC) are generally used as interchangeably to express the same design pattern. The pattern was initially called IoC, but Martin Fowler (known for designing the enterprise software) anticipated the name as DI because all frameworks or runtime invert the control in some way and he wanted to know which aspect of control was being inverted.

Inversion of Control (IoC) is a technique to implement the Dependency Inversion Principle in C#. Inversion of control can be implemented using either an abstract class or interface. The rule is that the lower level entities should join the contract to a single interface and the higher-level entities will use only entities that are implementing the interface. This technique removes the dependency between the entities.

#### **Note:**

In below implementation, I have used interface as a reference, but you can use abstract class or interface as per your requirement.

### **Implementation**

In below code, we have implemented DIP using IoC using injection constructor. There are different ways to implement Dependency injection. Here, I have use injection thru constructor but you inject the dependency into class's constructor (Constructor Injection), set property (Setter Injection), method (Method Injection), events, index properties, fields and basically any members of the class which are public.

public interface IAutomobile

{

void Ignition();

void Stop();

}

public class Jeep : IAutomobile

{

#region IAutomobile Members

public void Ignition()

{

Console.WriteLine("Jeep start");

}

public void Stop()

{

Console.WriteLine("Jeep stopped.");

}

#endregion

}

public class SUV : IAutomobile

{

#region IAutomobile Members

public void Ignition()

{

Console.WriteLine("SUV start");

}

public void Stop()

{

Console.WriteLine("SUV stopped.");

}

#endregion

}

public class AutomobileController

{

IAutomobile m\_Automobile;

public AutomobileController(IAutomobile automobile)

{

this.m\_Automobile = automobile;

}

public void Ignition()

{

m\_Automobile.Ignition();

}

public void Stop()

{

m\_Automobile.Stop();

}

}

class Program

{

static void Main(string[] args)

{

IAutomobile automobile = new Jeep();

//IAutomobile automobile = new SUV();

AutomobileController automobileController = new AutomobileController(automobile);

automobile.Ignition();

automobile.Stop();

Console.Read();

}

}