

## Dependency Inversion Principle

In an application we have low level classes which implement basic and primary operations and high level classes which encapsulate complex logic and rely on the low level classes...

NEW Forum Discussions: NEW

What Design Pattern Should I Choose?

Design Principles and Patterns      Dependency Inversion Principle

Motivation

In an application we have low level classes which implement basic and primary operations and high level classes which encapsulate complex logic and rely on the low level classes. A natural way of implementing such structures would be to write low level classes and once we have them to write the complex high level classes. Since the high level classes are defined in terms of others this seems the logical way to do it. But this is not a flexible design. What happens if we need to replace a low level class?

Let's take the classical example of a copy module which read characters from keyboard and write them to the printer device. The high level class containing the logic is the Copy class. The low level classes are KeyboardReader and PrinterWriter.

In a bad design the high level class uses directly the low level classes. In this case if we want to change the design to direct the output to a new FileWriter class we have to change the Copy class. (Let's assume that it is a very complex class, with a lot of logic and really hard to test).

In order to avoid such problems we can introduce an abstraction layer between the high level classes and low level classes. Since the high level modules contains the complex logic they should not depend on the low level modules and that the new abstraction layer should not be created based on low level modules. The low level modules are created based on the abstraction layer.

According to this principle the way of designing a class structure is to start from high level modules to the low level modules:

High Level Classes --> Abstraction Layer --> Low Level Classes Intent

- 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.    Example

Below is an example which violates the Dependency Inversion Principle. We have the manager class which is a high level class, and the low level class Worker. We need to add a new module to our application because in the company there are some new specialized workers employed. We created a new class SuperWorker for this. Let's assume that the Manager class is a complex one containing a very complex logic. And now we have to change it in order to introduce the new SuperWorker. Let's see the disadvantages:

- we have to change the Manager class (remember it is a complex one and this will involve some time and effort).
- some present functionality from the manager class might be affected.
- the unit testing should be redone.

All those problems will take a lot of time to solve. Now it would be very simple if the application was designed following the Dependency Inversion Principle. That means that we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface. When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it.

In order to have more dramatic effect, just imagine that the Graphic Editor is a big class, with a lot of functionality inside, written and changed by many developpers, while the a shape might be a class implemented only by one developer. In this case it would be great improvement to allow the adding of a new shape without changing the GraphicEditor class. // Dependency Inversion Principle - Bad example

```
class Worker {
public void work() {
// ....working
}
}

class Manager {
Worker m_worker;

public void setWorker(Worker w) {
m_worker=w;
}
```

```

public void manage() {
    m_worker.work();
}
}

class SuperWorker {
    public void work() {
        //.... working much more
    }
}

```

Below is the code which supports the Dependency Inversion Principle. In this new design a new abstraction layer is added through the IWorker Interface. Now the problems from the above code are solved:

- Manager class should not be changed.
- minimized risk to affect old functionality present in Manager class.
- no need to redone the unit testing for Manager class.

```

// Dependency Inversion Principle - Good example
interface IWorker {
    public void work();
}

class Worker implements IWorker{
    public void work() {
        // ....working
    }
}

class SuperWorker implements IWorker{
    public void work() {
        //.... working much more
    }
}

class Manager {
    IWorker m_worker;

    public void setWorker(IWorker w) {
        m_worker=w;
    }

    public void manage() {
        m_worker.work();
    }
}

```

## Conclusion

When this principle is applied it means that the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer. In that case the creation of new low level objects inside the high level classes(if necessary) can not be done using the operator new. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory, Prototype.

The Template Design Pattern is an example where the DIP principle is applied.

Of course, using this principle implies an increased effort and a more complex code, but more flexible. This principle can not be applied for every class or every module. If we have a class functionality that is more likely to remain unchanged in the future there is not need to apply this principle.