DIP — Dependency Inversion Principle:

Após estudarmos os demais princípios da metodologia S.O.L.I.D, iremos, por fim, entender o último principio o “D” de Princípio da inversão de dependência (Dependency Inversion Principle).

Este é a base para se te um projeto com um excelente design orientado a objeto, voltado o domínio e uma arquitetura flexível.

Uma primícias do DIP (Dependency Inversion Principle) é:

- Dependa de abstrações e não de implementação.

O princípio afirma que:

1. Os módulos de alto nível não devem depender dos módulos de baixo nível. Ambos devem depender de abstrações.
2. Abstrações não devem depender de detalhes. Os detalhes devem depender de abstrações.

Vamos primeiramente entender um pouco mais sobre o que são módulos de baixo e alto nível, acoplamento e abstração.

Módulos de alto nível.

Um módulo de alto nível é qualquer módulo que dependa de outros módulos.

Por exemplo:

Temos a classe Sms abaixo com o método “enviarSms ()”:

A classe Sms depende da classe “Cliente”, por tanto pode ser considerada um módulo de alto nível, pois tem dependência de outra classe concreta.

```
public class Sms {

    public void enviarSms(Cliente cliente, String mensagem) {

        System.out.println("SMS enviado para:" + cliente.email + "\nMensagem: " + mensagem);

    }
}
```

Módulo de baixo nível.

No exemplo abaixo, temos a classe “Cliente” com seus próprios atributos e sem acoplamento a outros módulos. Ela não tem dependência há outras classes, o que a torna um módulo de baixo nível.

```java
public class Cliente {
    public String nome;
    public String email;
    public String celular;

    public Cliente(String nome, String email, String celular) {
        this.nome = nome;
        this.email = email;
        this.celular = celular;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getCelular() {
        return celular;
    }
    public void setCelular(String celular) {
        this.celular = celular;
    }


}
```

Acoplamento.

O projeto passa a ter acoplamento a partir do momento em que uma determinada classe passa a depender de outra. O grande problema no acoplamento é quando uma classe com um determinado comportamento especifico tende a mudar, pois classes que estão acopladas a esta acabam tendo que ser alteradas do mesmo modo.

Ao se executar um acoplamento de forma errada, pode-se dificultar mudanças, pois muitas delas afetam outras partes do sistema. Ao imaginarmos um sistema mal acoplado, podemos imaginar uma cena bastante inusitada como uma série de dominós enfileirados, onde ao movimentar uma das peças as demais tendem a se comportar de forma arbitraria, gerando assim quebras em seu projeto.

Sempre quando instanciamos um novo objeto em nosso projeto, significa que estamos gerando forte dependência no código e, assim, o deixamos fortemente acoplado.

Abstração.

A abstração, em essência, considera as qualidades e os comportamentos interdependentes dos objetos a que pertencem, isolando-se os atributos leva-se em consideração o que existe em comum no objeto de um determinado grupo.

A programação deve ser dirigida sempre para a abstração e não para a concreção, ou seja, não se deve programar para classes concretas, mas sim para interfaces(abstrações).

Aplicando o princípio.

Agora que entendemos um poco mais sobre as ferramentas usadas pelo princípio para sua implementação, vamos ver um exemplo fora do princípio e aplicarmos o mesmo para, visualmente, notarmos sua aplicação.

Abaixo vemos uma classe de Cliente concreta e seus atributos e métodos.

```
public class Cliente {
    public String nome;
    public String email;
    public String celular;

    public Cliente(String nome, String email, String celular) {
        this.nome = nome;
        this.email = email;
        this.celular = celular;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
```

Note que está classe é um módulo de baixo nível, pois não depende de acoplamento de outra classe concreta.

Vemos também uma classe Main que usaremos para a execução de alguns comandos.

```java
public class Main {

    public static void main(String[] args) {

    }
}
```

Note que o objeto cliente é instanciado da classe Cliente e assim gerando forte acoplamento assim como sms e email também geram forte acoplamento. Aqui a classe Main é um método de alto nível, pois depende de classes concretas.

Podemos notar que, mesmo não estando dentro dos requisitos do princípio, a execução do código gera o resultado esperado, mas viola o princípio que diz que os módulos de alto nível não devem depender dos módulos de baixo nível, mas ambos devem depender de abstrações.

```java
public class Main {

    public static void main(String[] args) {
        Cliente cliente = new Cliente("João Marcos Maciel", "joaoMaciel@dipjava.com", "1296659874");

        Email email = new Email();
        email.enviarEmail(cliente, "Sua solicitação de serviço", "Segue em axexo o seu número de solicitação de se

        Sms sms = new Sms();
        sms.enviarSms(cliente, "Segue sua solicitação de serviço...");

    }
}
```

```
Email enviado para: joaoMaciel@dipjava.com
Assunto: Sua solicitação de serviço
Mensagem:Segue em axexo o seu número de solicitação de serviço.
Sms enviado para: 1296659874
Mensagem:Segue sua solicitação de serviço...
```

A classe Email é responsável pelo envio da email ao cliente e é um módulo de alto nível, pois depende da classe concreta de Cliente.

```java
public class Email {
    public void enviarEmail(Cliente cliente, String assunto, String mensagem) {
        System.out.println("Email enviado para: "+ cliente.getEmail() + "\nAssunto: " + assunto + "\nMensagem:" + mensagem);
    }
}
```

O mesmo acontece com a classe Sms que é responsável pelo envio das mensagens via sms.

```java
public class Sms {
    public void enviarSms(Cliente cliente, String mensagem) {
        System.out.println("Email enviado para: "+ cliente.getCelular() + "\nMensagem:" + mensagem);
    }
}
```

Vamos primeiro criar abstrações, interfaces, para colocarmos nosso programa dentro do primeiro requisito do princípio.

Primeiro iremos criar a interface cliente (InterfaceCliente), que contém os métodos e geram assim baixo acoplamento dentro do projeto.

```java
public interface InterfaceCliente {
    public void saveCliente(Object String, Object String2, Object String3);
    public String getEmail();
    public String getCelular();
}
```

Aplicando ao Main temos:

```java
public class Main {

    public static void main(String[] args) {
        InterfaceCliente cliente = new Cliente("João Marcos Maciel", "joaoMaciel@dipjava.com", "1296659874");

        Email email = new Email();
        email.enviarEmail(cliente, "Sua solicitação de serviço", "Segue em axexo o seu número de solicitação de serviço.");

        Sms sms = new Sms();
        sms.enviarSms(cliente, "Segue sua solicitação de serviço...");

    }
}
```

Iremos submeter as classes Email e Sms ao mesmo processo.

InterfaceEmai:

```java

public interface InterfaceEmail {

    void enviarEmail(InterfaceCliente cliente, String string, String string2);

}

```

```java

public class Email implements InterfaceEmail{
    public void enviarEmail(InterfaceCliente cliente, String assunto, String mensagem) {
        System.out.println("Email enviado para: "+ cliente.getEmail() + "\nAssunto: " + assunto + "\nMensagem:" + mensagem);
    }
}
```

InterfaceSms:

```java
public interface InterfaceSms {

    void enviarSms(InterfaceCliente cliente, String string);

}
```

```java

public class Sms implements InterfaceSms{
    public void enviarSms(InterfaceCliente cliente, String mensagem) {
        System.out.println("Email enviado para: "+ cliente.getCelular() + "\nMensagem:" + mensagem);
    }
}
```

Temos agora um baixo acoplamento na Main atendendo ao primeiro requisito do princípio.

```java
public class Main {

    public static void main(String[] args) {
        InterfaceCliente cliente = new Cliente("João Marcos Maciel", "joaoMaciel@dipjava.com", "1296659874");

        InterfaceEmail email = new Email();
        email.enviarEmail(cliente, "Sua solicitação de serviço", "Segue em axexo o seu número de solicitação de serviço.");

        InterfaceSms sms = new Sms();
        sms.enviarSms(cliente, "Segue sua solicitação de serviço...");

    }
}
```

Para atender o segundo requisito que diz que abstrações não devem depender de detalhes, os detalhes devem depender de abstrações, iremos criar uma interface Factory

com métodos estáticos que irão instanciar novos objetos gerando assim dependência das abstrações e não o contrário.

```
public interface Factory {

    public static Cliente getCliente(String string, String string2, String string3) {
        return new Cliente(string, string2, string3);
    }

    public static Email getEmail() {
        return new Email();
    }

    public static Sms getSms() {
        return new Sms();
    }
}
```

Assim temos:

```
public class Main {

    public static void main(String[] args) {
        InterfaceCliente cliente = Factory.getCliente("João Marcos Maciel", "joaoMaciel@dipjava.com", "1296659874");

        InterfaceEmail email = Factory.getEmail();
        email.enviarEmail(cliente, "Sua solicitação de serviço", "Segue em axexo o seu número de solicitação de serviço.");

        InterfaceSms sms = Factory.getSms();
        sms.enviarSms(cliente, "Segue sua solicitação de serviço...");

    }
}
```

```
Email enviado para: joaoMaciel@dipjava.com
Assunto: Sua solicitação de serviço
Mensagem:Segue em axexo o seu número de solicitação de serviço.
Sms enviado para: 1296659874
Mensagem:Segue sua solicitação de serviço...
```

Como podemos ver, conseguimos com as mudanças remover o alto acoplamento que tínhamos no início do projeto por dependermos das classes concretas Cliente, Email e Sms. Neste formato a aplicação pode evoluir de forma mais robusta, estável e segura.