The target and dependent classes of Dependency Management can be designed by using the following principles of strategy DP

==========

==========

==========

Rule/Principle1 :: Favour Compsition (HAS-A Relation) over Inheritence (IS -A Relation) Rule/Principle2 :: Always code to interfaces/abstract classes i.e do not code to concreate classes to achieve the loose coupling in the implementation Dynamic Polymorphism Rule/Principle3 :: Our Code must be open for extension and must be closed for Modification..

Rule/Principle 2 :: Always code to interfaces/abstract classes i.e do not code to concreate classes to achieve the loose coupling in the implementation Dynamic Polymorphism

Problem:

is

becomes

if target calss having multiple possible dependent classes then the code tigtly coupled code if we take dependent class type HAS - A property in target calss.

Flipkart.java (target class)

```
public class Flipkart{
HAS-A property
private DTDC dtdc=new DTDC();
public String shopping(String items[],
```

DTDC.java (dependent calss1)

```
public class DTDC{
public String deliver(int oid){
double prices[]){
}
...
... //logic to calc BillAmt
}
"
String msg=dtdc.deliver(oid);
return "....";
}//method
}//class
...
//logics
```

BlueDart.java (dependent calss?)

```
public class BlueDart{
```

public String deliver(int oid){

//logics

}

}

oid :: order id

=>if the degree of dependency is more between two comps then they are called tightly coupled comps =>if the degree of dependency is less between two comps then they are called loosely coupled comps

use

=> Flipkart wants to BlueDart courier services instead of DTDC Courier service then we need to modify the the source of Flipkart class..(target class).. This makes these comps as the tightly coupled comps.

Solution using interfaces

==========

=> Make the all the possible dependent classes implementing the common interface and take that interface type HAS-A property having setter method for that property in target class to make the code as loosely coupled code

Flipkart.java (target class)

public class Flipkart{

//HAS-A property

Courier.java (common interface)

public interface Courier{

private Courier courier;

publietter method

To assing

Courier

obj to Flipkart }

object

void setCourier (Courier couerier){

this.courier-courier;

//b.method

public String shopping(String items[],

public String deliver(int oid);

}

DTDC.java (Dependent class1)

public class DTDC implements Courier{

double prices[]){

public String deliver(int oid){

...//logic to calc BillAmt

"

String msg=usier.deliver(sid);

}

}

return "....";

} //method

}//class

BlueDart.java (Dependent class2)

public class BlueDart implements Courier{

public String deliver(int oid){

""

..

} }

In client App (main class)

==========

Flipkart with DTDC

//create Target class obj Flipkart fpkt=new Flipkart(); ' // create Dependent class obj Courier courier=new DTDC(); fpkt.setCourier(courier);

//change courier

courier=new BlueDart();

fpkt.setCourier(courier);

Rule/Principle3 ::

Flipkart With

BlueDart

=>Here we can

change

the dependent class for target Flipkart class from the client Apps with out distrubing the source code of target class. (This makes the code as loosely coupled)

note:: Method overloading gives static /compile time polymorphism Method Overiding with the support of interfaces/abstact classes gives dynamic /rumtime polymorphism

Our Code must be open for extension and must be closed for Modification..

=> Making all the possible dependent classes implementing the common interface makes

"our code open for exention" becoz we can add more dependent classes by implementing that interface.

add

eg: we can more dependent classes like FirstFlight, DHL and etc.. implementing the common interface (Courier)

DHL.java

public class DHL implements Courier{

FirstFlight.java

public class FirstFlight implements Courier{

public String deliver(int oid){

public String deliver(int oid){

}

}

}

=> Either by taking the classes as the final classes or by taking the mehods as final methods we can make our code closed for modification..

note1:: Final classes can not have sub classes So code of the class can not be modified

note2:: final methods of super class can not overridden the sub classes .. So the code of the methods can not be modified.

=> After implementing rule1 and rule2 perfectly.. by just doing small modifications this rule3 can be implemented.

Flipkart.java (target class)

final

public class Flipkart{

//HAS-A property

Courier.java (common interface)

private Courier courier;

public interface Courier{

public String deliver(int oid);

public void setCourier(Courier couerier){

}

this.courier-courier;

}

public String shopping(String items[], double prices[]){

DTDC.java (Dependent class1)

final public class DTDC implements Courier{

public String deliver(int oid){

... //logic to calc BillAmt

...

""

String msg=dtdc.deliver(oid);

-

}

}

return "....";

```
} //method
}//class
DHL.java
final
public class DHL implements Courier{
public String deliver(int oid){
}
}
Strategy DP Implemenation in core java
==========================
BlueDart.java (Dependent class2)
final
public class BlueDart implements Courier{
public String deliver(int oid){
}
}
"""
FirstFlight.java
public class FirstFlight implements Courier{
final
public String deliver(int oid){
}
CoreProj4- strategyDP
---->src
|----com.nt.comp]
|--->Courier.java (Common interface for depdent classes) |---> DTDC.java (dependent classes)
|--->BlueDart.java
|--->Flipkart.java (target class)
com.nt.factory]
|----->FlipkartFactory.java
----com.nt.test
CoreJavaProj4-StrategyPattern
> JRE System Library [JavaSE-17]
✓ src
>
com.nt.comp
BlueDart.java
```

> Courier.java

> DTDC.java

com.nt.factory

> Flipkart.java

> FlipkartFactory.java

✓ com.nt.test

(factory pattern class)

----->StrategyDPTest.java (Client App)

//Courier.java (Common Interface)

```
package com.nt.comp;
public interface Courier {
}
```

> StrategyPattern Test.java

public String deliver(int oid);

//BlueDart.java (dependent class2)

```
package com.nt.comp;
public final class BlueDart implements Courier {
```

Ctrl+shift+o:: To import the packages automatically

It is recomandded to take java package names with multiple words having company names, project names, module names and etc...

sample package names are :: com.nt.<category> org.nt.<category>

<proj name>.<mod name>.<category>

we can take the

=>In the implementation one Design Pattern support of multiple other design patterns like we can use Factory Pattern in the implementation of strategy DP to create and return target class obj having the client choice dependent class obj for the main class/client app ..

```
}
}
```

//DTDC.java (dependent class1)

```
package com.nt.comp;
@Override
public String deliver(int oid) {
return oid+" order items are kept for delivery by BlueDart";
public final class DTDC implements Courier {
@Override
public String deliver(int oid) {
return oid+" order items are kept for delivery by DTDC";
}
```

```
}
//Flipkart.java (target class)
package com.nt.comp;
import java.util.Arrays; import java.util.Random;
public final class Flipkart {
//HAS-A property
private Courier courier;
public void setCourier(Courier courier) {
System.out.println("Flipkart.setCourier()");
this.courier-courier;
}
// b.method
public String shopping(String items[], double prices[]) {
//calculate bill amount
double billAmount=0.0;
for(double p: prices) {
}
billAmount-billAmount+p;
// generate the order id (random number as the order)
int oid=new Random().nextInt(1000);
}
}
// deliver the products using courier
String msg=courier.deliver(oid);
return Arrays.toString(items)+" items with billAmount:::"+billAmount+" ----
//FlipkartFactory.java (Factory Pattern class)
package com.nt.factory;
import com.nt.comp.BlueDart;
import com.nt.comp.Courier;
import com.nt.comp.DTDC;
import com.nt.comp.Flipkart; public class FlipkartFactory {
//static factory method having factory pattern logic
public static Flipkart getInstance(String courierType) { //create Dependent class obj
Courier courier=null;
if(courierType.equalsIgnoreCase("dtdc"))
courier=new DTDC();
else if(courierType.equalsIgnoreCase("blueDart"))
```

```
else
courier=new BlueDart();
throw new IllegalArgumentException("invalid courier type");
//create target class obj
Flipkart fpkt=new Flipkart();
//assign dependent class object to target class obj fpkt.setCourier(courier);
return fpkt;
}
}
//StrategyPatternTest.java
package com.nt.test;
import com.nt.comp.Flipkart; import com.nt.factory.FlipkartFactory;
public class StrategyPatternTest {
public static void main(String[] args) {
// use Factory Pattern to target class obj
Flipkart fpkt=FlipkartFactory.getInstance("dtdc");
//invoke the b.method
String resultMsg=fpkt.shopping(new String[] {"shirt","trouser"},
System.out.println(resultMsg);
new double[] {5000.0,6000.0});
}//main
}//class
"+msg;
```

(target class) (dependent classes) Assigment:: implement strategy DP on Vechile and Engine classes.

(PetrolEngine, DieselEngine

EltricEngine)

b.method

make all these classes implemnenting common IEngine(1) having two methods declaration |---> public void startEngine(); |----> public void endEngine();

```
public void journey(String startPlace, String destPlace){
.
}
```