COR :- Chain of Responsibility. / Linked list.
If, else but in a modular way of classes.

abstract Handler class which contains next type Handler object.

next is initialized through constructor. to it's next handler type.

Has an abstract function which takes request object to handle the request depending upon object's state.

```
//cannot make this classe's object as its abstract
public abstract class Handler {
    protected Handler next;

    public Handler(Handler next) {
        this.next = next;
    }

    abstract void handleRequest(Request obj);
}
```

Now we can make any number of handles to handle a single type of responsibility depending upon the Request object that is passed to the handle.

RealHandler 1 — if we cannot handle the case, we delegate.

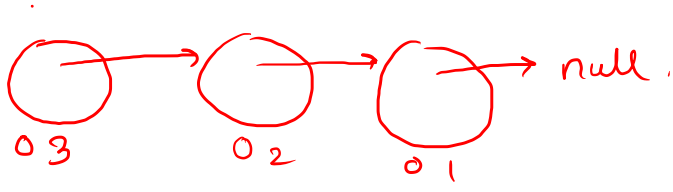
```java
public class RealHandler1 extends Handler {

    public RealHandler1(Handler next) {
        super(next); //as it derives from handle we can directly
                     //inititalize next to super's next; || Calls parent's constructor and passes next.
    }

    @Override
    void handleRequest(Request obj) {
        if(obj.state > 0) {
            System.out.println(x: "Handling the positive state request");
        } else if(next != null){
            System.out.println(x: "Cannot handle the request hence deligating");
            next.handleRequest(obj);
        }

    }

}
```

```java
public class RealHandler2 extends Handler {

    RealHandler2(Handler next) {
        super(next);
    }
    @Override
    void handleRequest(Request obj) {
        if(obj.state == 0) {
            System.out.println(x: "Handling the zero state request");
        } else if(next != null){
            System.out.println(x: "Cannot handle the request hence deligating");
            next.handleRequest(obj);
        }
    }

}
```

```java
public class RealHandler3 extends Handler{

    public RealHandler3(Handler next) {
        super(next);
    }

    @Override
    void handleRequest(Request obj) {
        if(obj.state < 0) {
            System.out.println(x: "Handling the negative state request");
        } else if(next != null){
            System.out.println(x: "Cannot handle the request hence deligating");
            next.handleRequest(obj);
        }

    }
}
```

```java
public class Request {
    int state;

}
```

```java
public class Test {
    Run | Debug
    public static void main(String[] args) {

        Handler o1 = new RealHandler1(next: null);
        Handler o2 = new RealHandler2(o1);
        Handler o3 = new RealHandler3(o2);

        Request request = new Request();
        request.state = 20;

        o3.handleRequest(request);
    }
}
```



Strategy Design Pattern :- If we contain a Family of Algorithms, we use Composition instead of Inheritance to make the code maintainable and to not let there be a burst of classes.

There are 2 ways we can reuse code in class B
1: Inheritance - B derives from A.
2: Composition - B contains an object of A inside it.

We should always prefer Composition over Inheritance.

Strategy Pattern is used to lower the number of class files created to make a programme work.

* This is one of the most used Design Pattern.

```java
public class Fighter {
    IKickBehaviour kb;
    IPunchBehaviour pb;
    IMagicBehaviour mb;

    Fighter() {
        this.kb = new NoKickBehaviour();
        this.pb = new NoPunchBehaviour();
        this.mb = new NoMagicBehaviour();
    }

    void fight() {
        kb.kick();
        mb.magicAttack();
        pb.punch();
    }
}
```

Fighter has 3 behaviours / Families of algorithms
1) Kicking
2) Magic
3) Punch.

We do not put these behaviours directly in the class itself.

Instead we create Interfaces of these Behaviours and then specialize them according to our needs.

```java
package IKickBehaviour;

public interface IKickBehaviour {
    void kick();
}
```

```java
package IMagicBehaviour;

public interface IMagicBehaviour {
    void magicAttack();
}
```

```java
package IPunchBehaviour;

public interface IPunchBehaviour {
    void punch();
}
```

Kick Family

```java
package IKickBehaviour;

public class NoKickBehaviour implements IKickBehaviour {
    @Override
    public void kick() {
        System.out.println(x: "Sorry Cannot Kick ./././");
    }
}
```

```java
package IKickBehaviour;

public class BackKick implements IKickBehaviour {
    @Override
    public void kick() {
        System.out.println(x: "BackKick woooh");
    }
}
```

```java
package IKickBehaviour;

public class Dolichegi implements IKickBehaviour {
    @Override
    public void kick() {
        System.out.println(x: "chigi chigi chegi chegi");
    }
}
```

```java
package IKickBehaviour;

public class RoundKick implements IKickBehaviour {
    @Override
    public void kick() {
        System.out.println(x: "Round Kick heeyah!");
    }
}
```

Magic Family

```java
package IMagicBehaviour;

public class DarkMagic implements IMagicBehaviour {

    @Override
    public void magicAttack() {
        System.out.println(x: "Dark Chakra Ball!!");

    }

}
```

```java
package IMagicBehaviour;

public class FinalFlash implements IMagicBehaviour {
    @Override
    public void magicAttack() {
        System.out.println(x: "White Final Flash .!.!.!.!.");
    }
}
```

```java
package IMagicBehaviour;

public class GallicGun implements IMagicBehaviour {
    @Override
    public void magicAttack() {
        System.out.println(x: "Gaaaaaa Lick Gunnnn");
    }
}
```

```java
package IMagicBehaviour;

public class NoMagicBehaviour implements IMagicBehaviour{
    @Override
    public void magicAttack() {
        System.out.println(x: "Cant do Magic Attacks");
    }
}
```

Punch Family

```java
package IPunchBehaviour;

public class jabPunch implements IPunchBehaviour {
    @Override
    public void punch() {
        System.out.println(x: "Hit With a Jab");
    }
}
```

```java
package IPunchBehaviour;

public class NoPunchBehaviour implements IPunchBehaviour {
    @Override
    public void punch() {
        System.out.println(x: "Sorry Cant Punch :(");
    }
}
```

```java
package IPunchBehaviour;

public class OneInchPunch implements IPunchBehaviour {
    @Override
    public void punch() {
        System.out.println(x: "Bruce Lee OIP");
    }
}
```

```java
public class OnePunch implements IPunchBehaviour {
    @Override
    public void punch() {
        System.out.println(x: "You are dead with One Punch");
    }
}
```

```java
package IPunchBehaviour;

public class UpperCutPunch implements IPunchBehaviour {
    @Override
    public void punch() {
        System.out.println(x: "Upper Cuts ! -d-");
    }
}
```

```java
import IKickBehaviour.Dolichegi;
import IMagicBehaviour.Kamehameha;
import IPunchBehaviour.UpperCutPunch;

public class Test{
    Run | Debug
    public static void main(String[] args) {
        Fighter Goku = new Fighter();
        Goku.kb = new Dolichegi();
        Goku.mb = new Kamehameha();
        Goku.pb = new UpperCutPunch();
        Goku.fight();

        System.out.println(x: " ");

        Fighter defaultFighter = new Fighter();
        defaultFighter.fight();
    }
}
```

Hence Kick, Punch, Magic Behaviours have 4,5 variations each and are represented by IKick, IPunch, IMagic interfaces respectively.

Only 15 files are needed and now we can create $4 \times 4 \times 4 = 64$ types of Fighter objects through these files.

No code smell (Inherited directly behaviours from main Fighter class).

[Interface reference variable can store objects of any classes who implements them].