① **Single Responsibility principle ?**

states that a class should have only one reason to change

( a class have only one Responsibility )

**Before**

```
class Bank {

public void Login ()

public void deposit ()

public void withdanaw()

}
```

**Now**

```
class Bank_Login {
}

class Bank_Deposit {
}

class Bank_withdnaw
{ }
```

---

**Violating SOLID consequence**

① Inconsistency between class

② Tight coupling

③ increase code complexity

④ reduce extensibility

⑤ Difficult in testing

⑥ inconnect behavior by supenclass subclass

⑦ unessary dependency

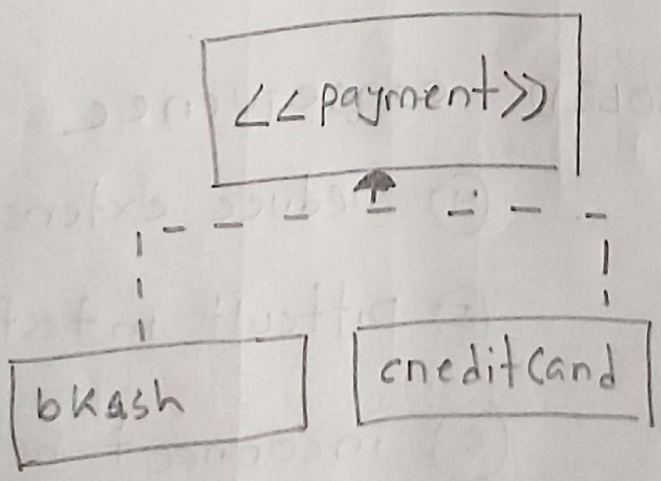
CS CamScanner

# Open-closed principle

states that software entities (class, module, functions) should be open for extension and closed for modification

## Before

```
class payment {

void bkash() {...}

void creditCard() {...}

}
```




## after

```
public interface
    payment {

    }

public class
bKash implements
    payment {

    }

public class
creditCard implements
    payment {

    }
```

CamScanner

# Liskov substitution principle

states that object of superclass must be replaceable with the object of its subclass without breaking the application

→ subclass (object) should behave the way superclass behave

→ violation :- throw exception / do nothing

superclass
```
class Bird {
  void fly();
  void run();
}
```

subclass
```
class Dove
  extends Bird
{
  void fly();
  void run();
}
```

subclass
```
class penquine
  extends
    Bird
{
  void fly();
  void run();
}
```
cannot fly

— o —

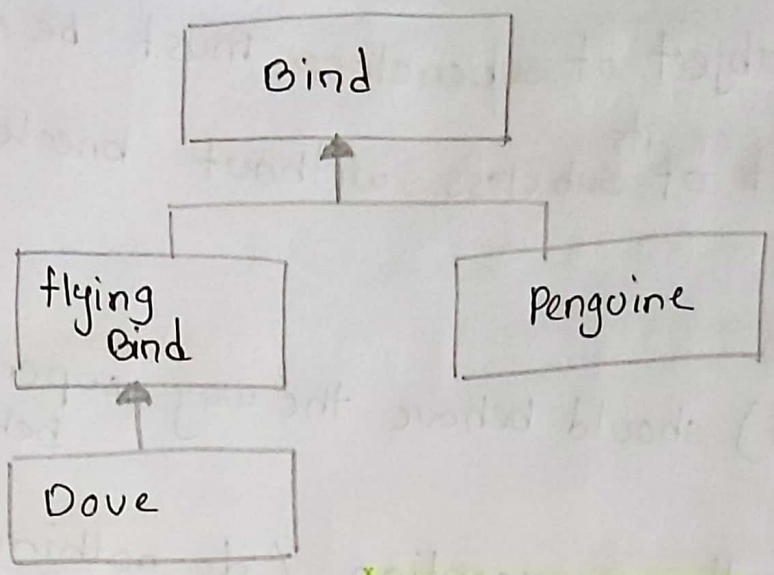
Solution :-

```
class Bird {
  void run();
}
```

both super and subclass
```
flying binds
class extends Bird {
  void run();
  void fly();
}
```

```
class Dove
  extends
    flyingBinds
{ void run();
  void fly();
}
```

```
class penguine
  extends Bird {
  void run();
}
```




## Interface segnation principle

states that client should not be forced to implements methods of an interface that they dont use.

```
class social media {
        void chatwithfriend();
        void sendphoto();
        void post();
        }
```

```
class facebook
    implements SocialMedia {
    void chatwithfriend();
    void sendphoto();
    void post(),
    }
```

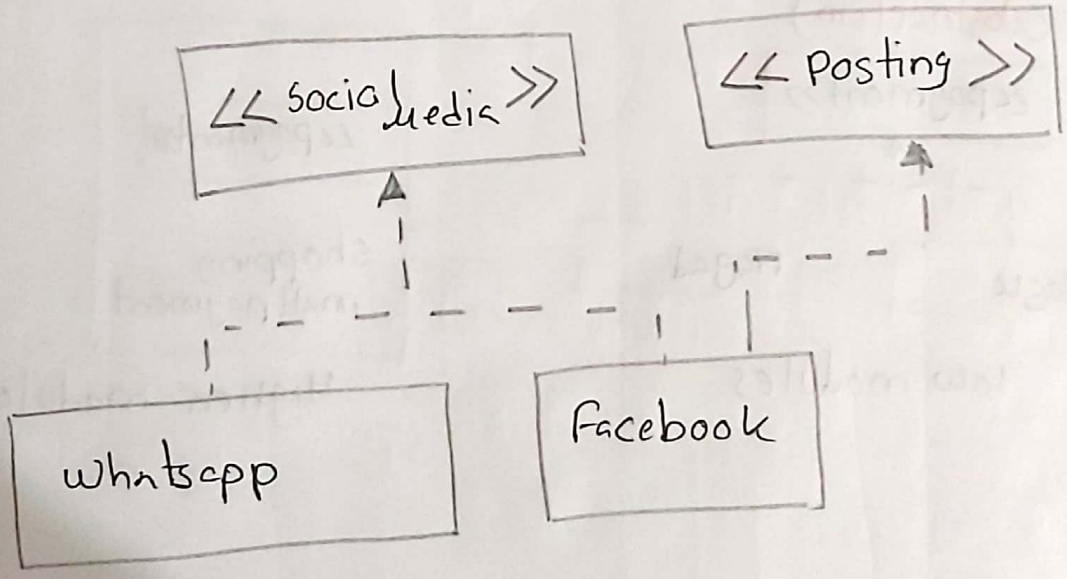
```
class whatsapp
    implements Social
    media {
    void chatwith friend
    void sendphoto()
    void post();
    }
```

```
public interface SocialMedia
{
  void chatwithfriend();
  void sendphoto
}
```

```
public Interface Posting
{
  void post();
}
```

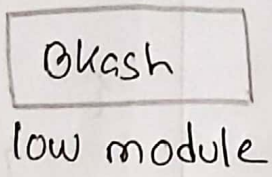
```
public facebook
  implements Social Media, Posting
  {

  }
```

```
public whatsapp
  implements
      SocialMedia
  {
  }
```

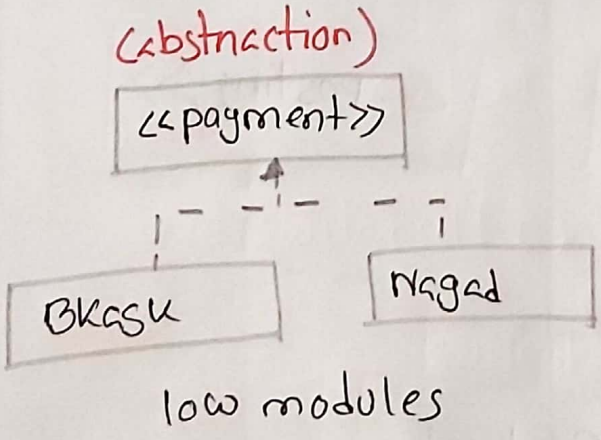


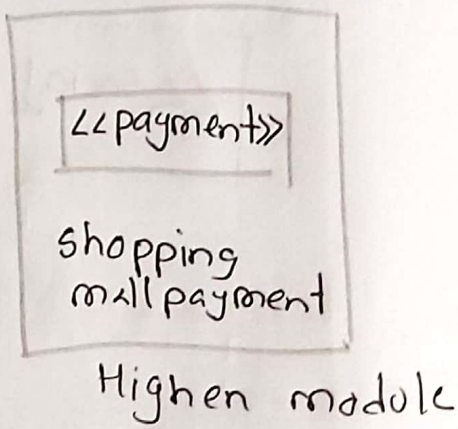
# Dependency Invension Principle

Higher level module should not depend on lower level module, but both should depend on abstraction.



— o —



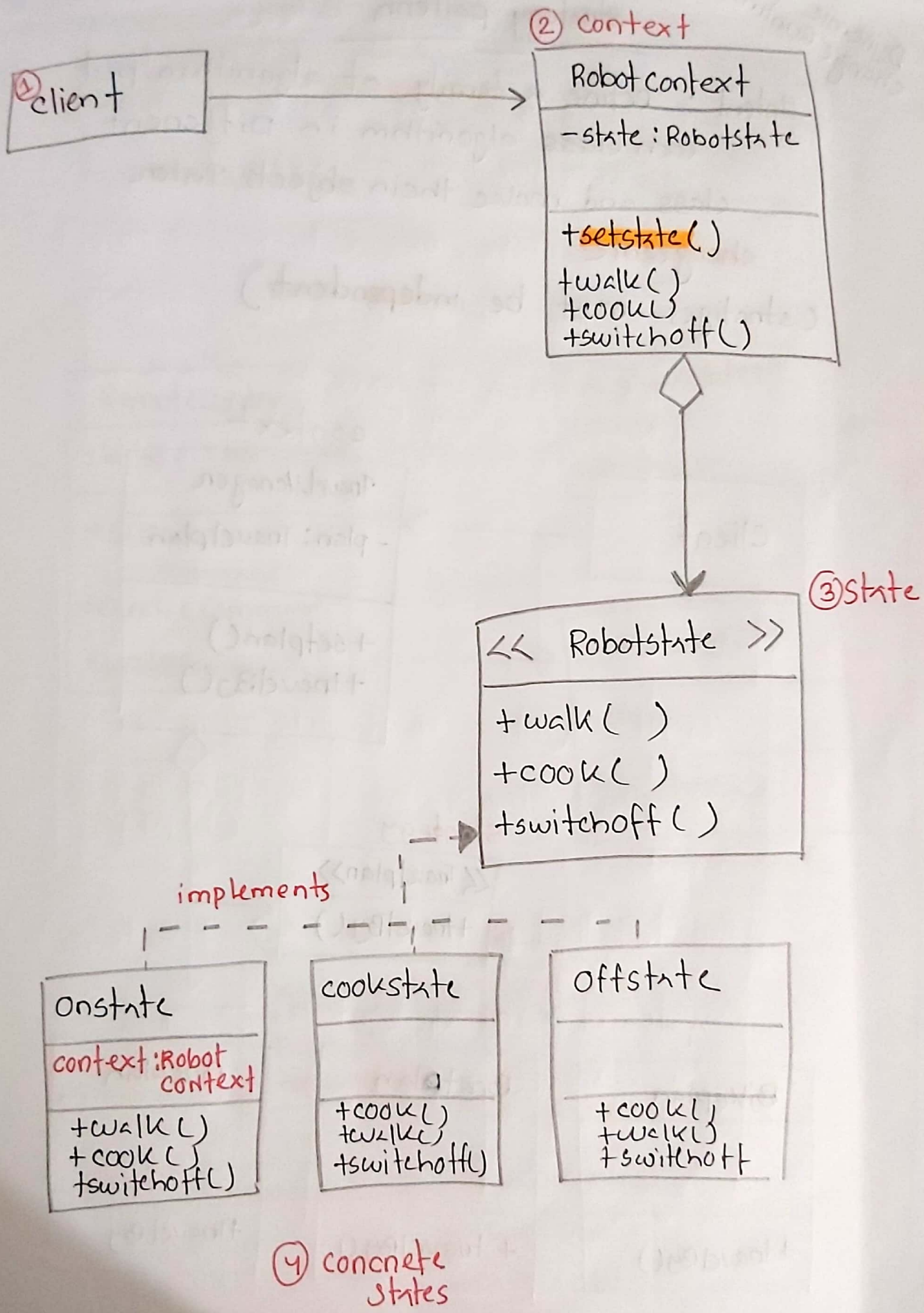



# State pattern

**Intent :-** allows an objects to alter its behaviour by changing its states.

( states can be dependent with each other)

```
class onstate implements Robotstate {

    Robot context context ;

    void walk ( ) { . . . }
    void cook ( ) {
            context.setstate (new cookstate());
            }
    void switchoff ( ) {
            context . setstate (new offstate());
            }

}

client {
    Robot context robot = new robotcontext();
```

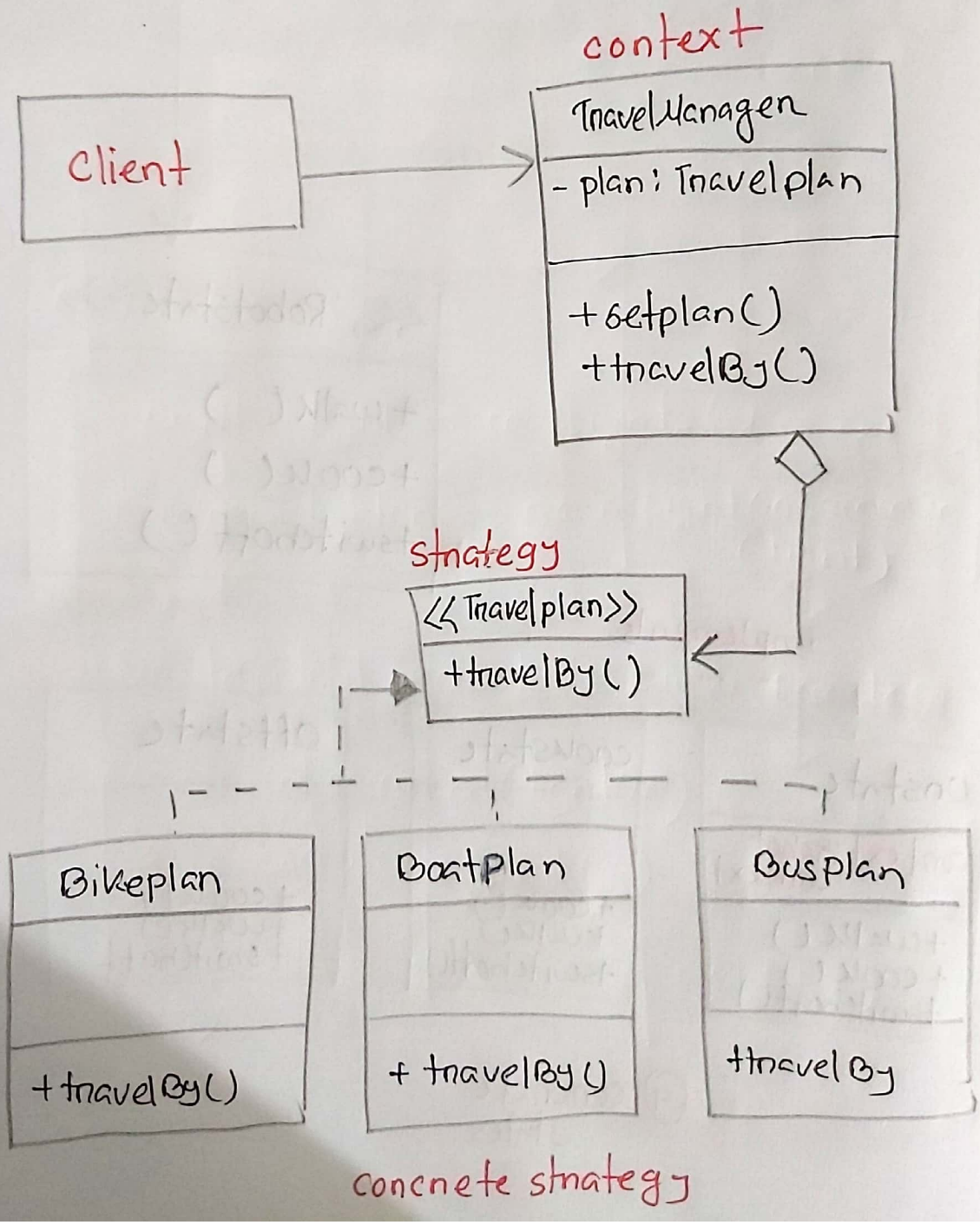

① client
② context
Robot context
-state : Robotstate
+setstate()
+walk()
+cook()
+switchoff()
③ State
<< Robotstate >>
+walk( )
+cook( )
+switchoff( )
implements
Onstate
context : Robot context
+walk()
+cook()
+switchoff()
cookstate
+cook()
+walk()
+switchoff()
offstate
+cook()
+walk()
+switchoff
④ concrete states

Dynamic change Runtime

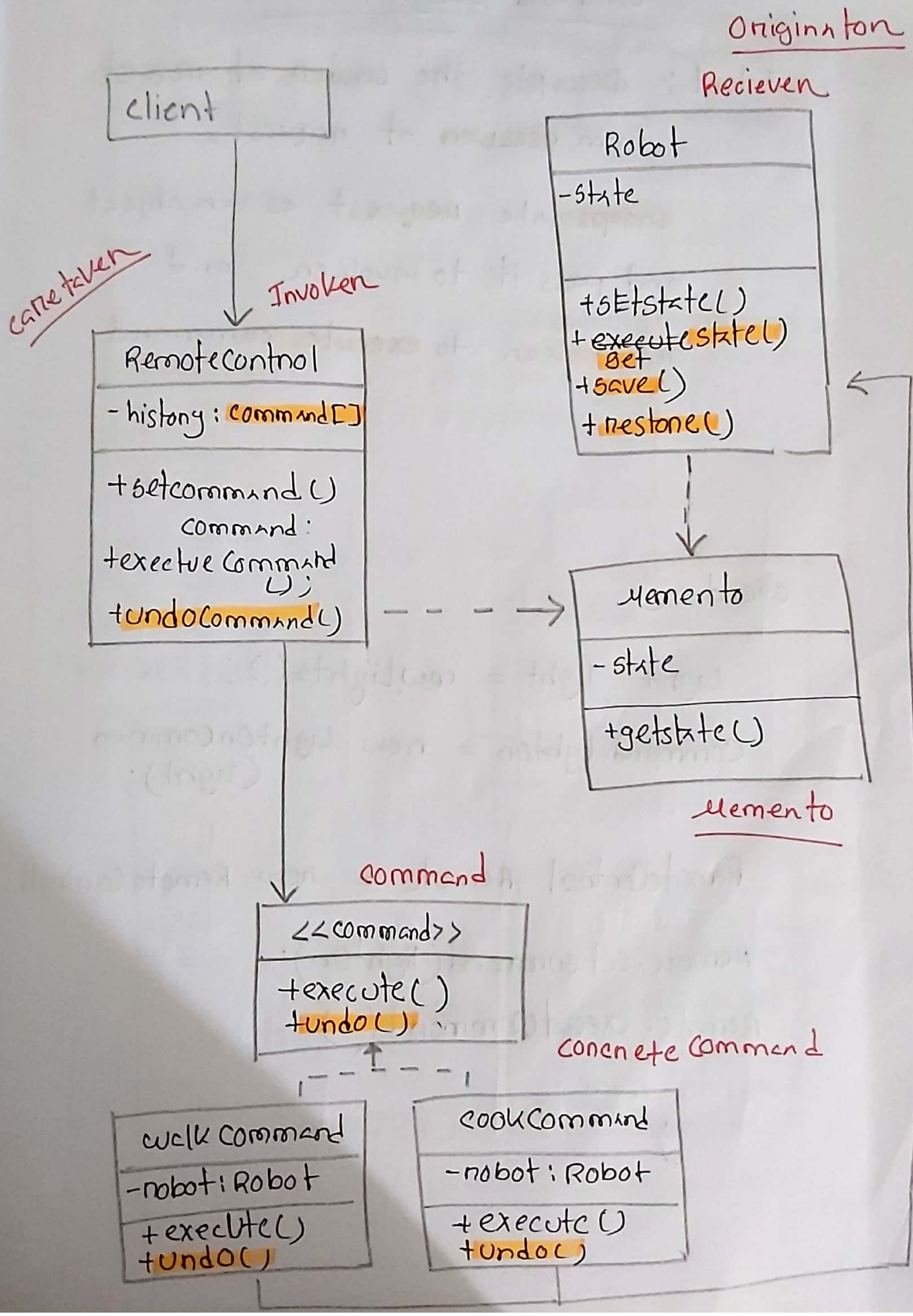
# Strategy pattern

Intent :- Define a family of algorithm put each ~~class~~ algorithm in Different class and make their objects Inter changeable

( strategies must be independent )



# Combining Memento & command

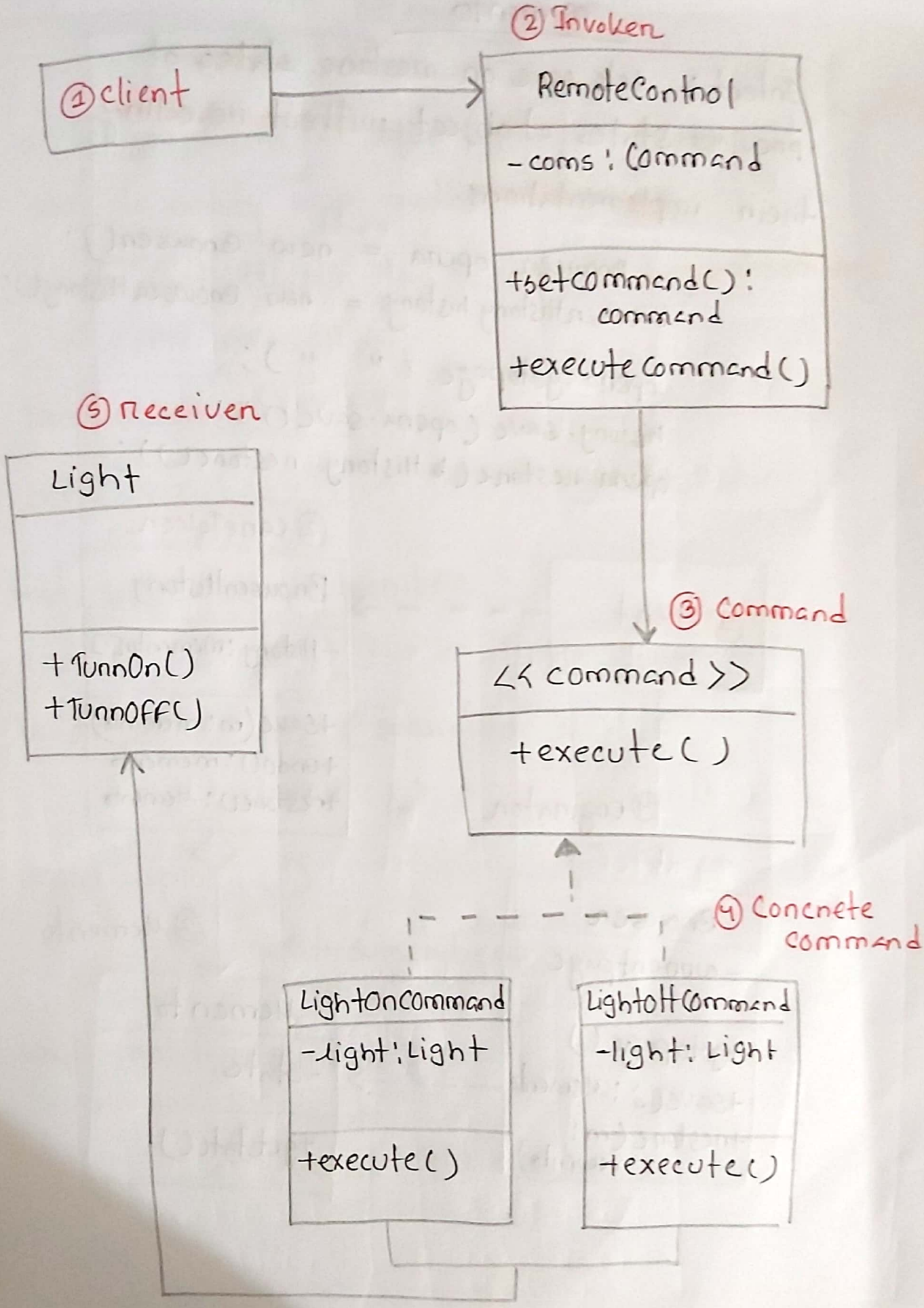


# Command Design Pattern

Intent :- Decouple the senden of nequest from necciven of nequest,

encapsulate nequest as an object and pass it to invoken, and let invoken to execute command

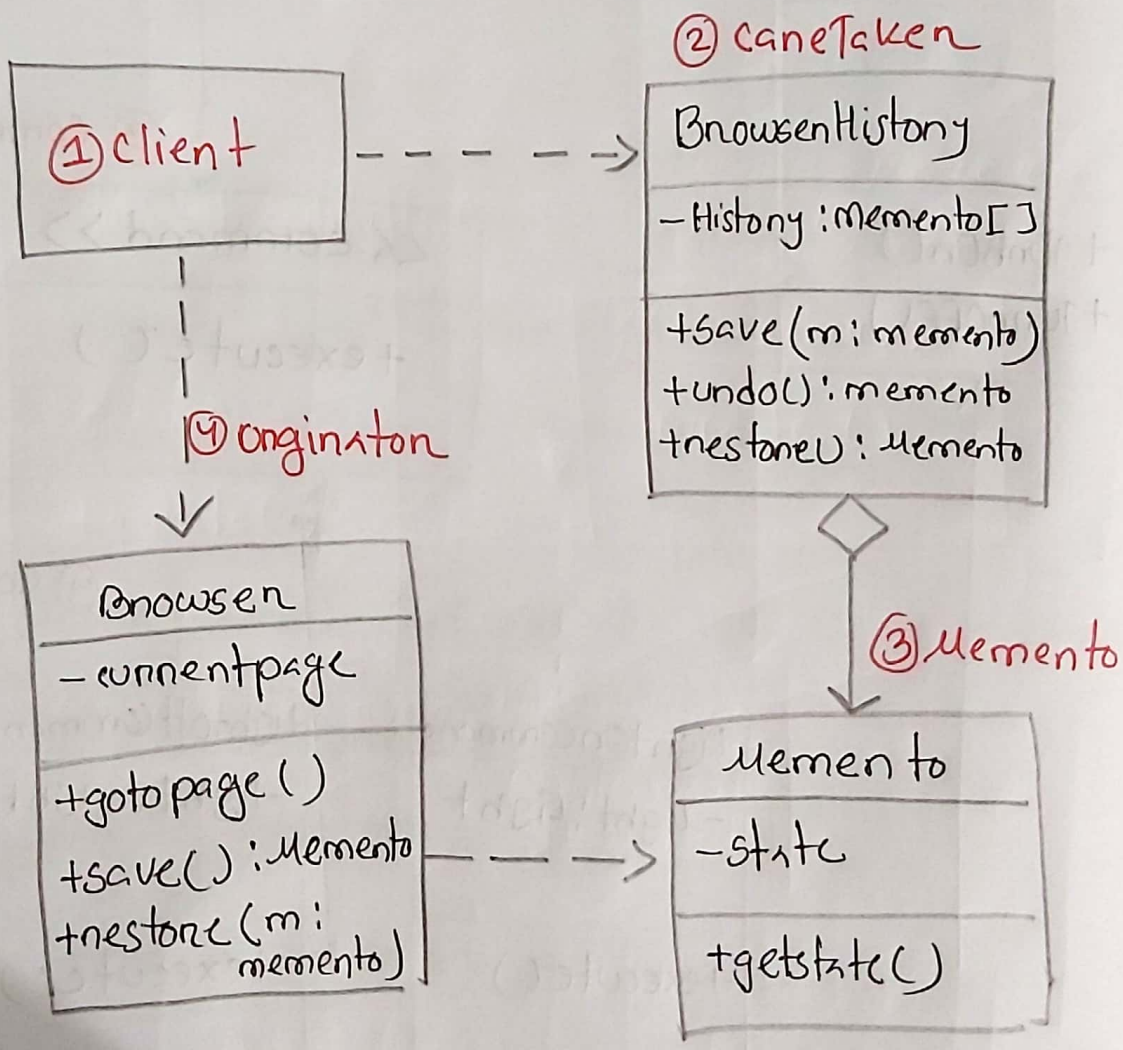
```
client {

    Light light = newLights();
    Command lighton = new LightOnCommen
                          (light);

    RemoteControl Remote = new RemoteControll

    Remote.setcommand(lightOn);
    Remote.executeCommand();

}
```


① client
② Invoker
RemoteControl
- coms : Command
+setcommand() : command
+execute command()
③ Command
<< command >>
+execute()
④ Concrete command
LightOnCommand
-light: Light
+execute()
LightoffCommand
-light: Light
+execute()
⑤ receiver
Light
+TurnOn()
+TurnOff()

# Memento

Intent :- Lets save or restore states of previous states of object without revealing their implementations.

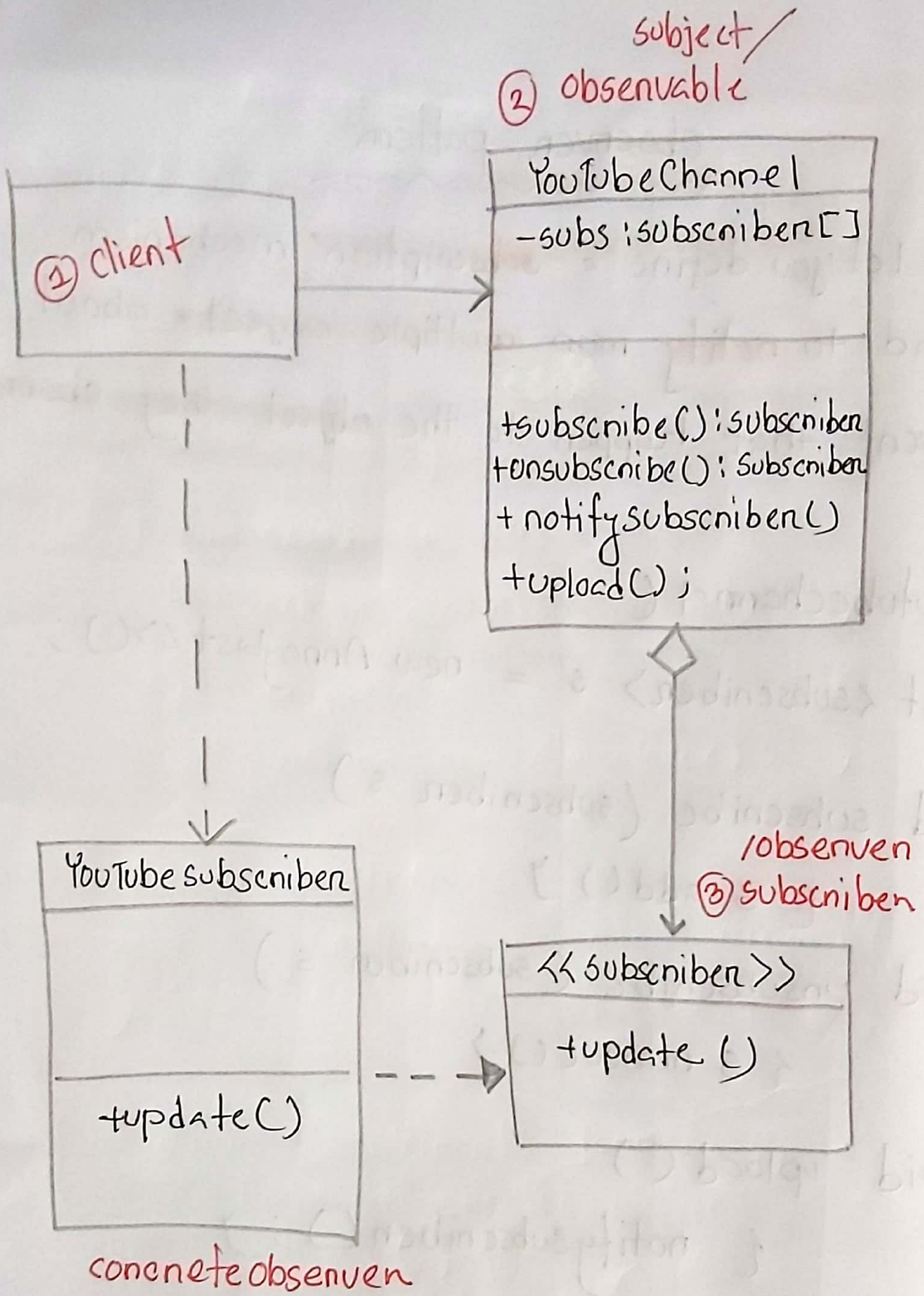
```
Browser opera = new Browser();
BrowserHistory history = new BrowserHistory()
opera.gotopage(" ");
history.save(opera.save());
opera.restore(History.restore());
```



# observer pattern

→ Let you define a subscription mechanism and to notify ~~map~~ multiple objects about an event that happen to the object they observing

```
Youtubechannel {
List <subscriber> s = new ArrayList<>();

void subscribe (subscriber s)
    { s.add() }
void unsubscribe (subscriber s)
    { s.remove() }
void upload ()
    { notifysubscriber(); }

void notifysubscriber (  ) {
        for (auto p : subscriber)
            p.update()
        }
```


subject/
② obsenvable
YouTubeChannel
-subs : subscriben[]
+subscribe() : subscriben
+unsubscribe() : subscriben
+ notifysubscriben()
+upload();
① client
/obsenven
③ subscriben
YouTube subscriben
+update()
<< subscriben >>
+update ()
conenete obsenven

# Abstract factory

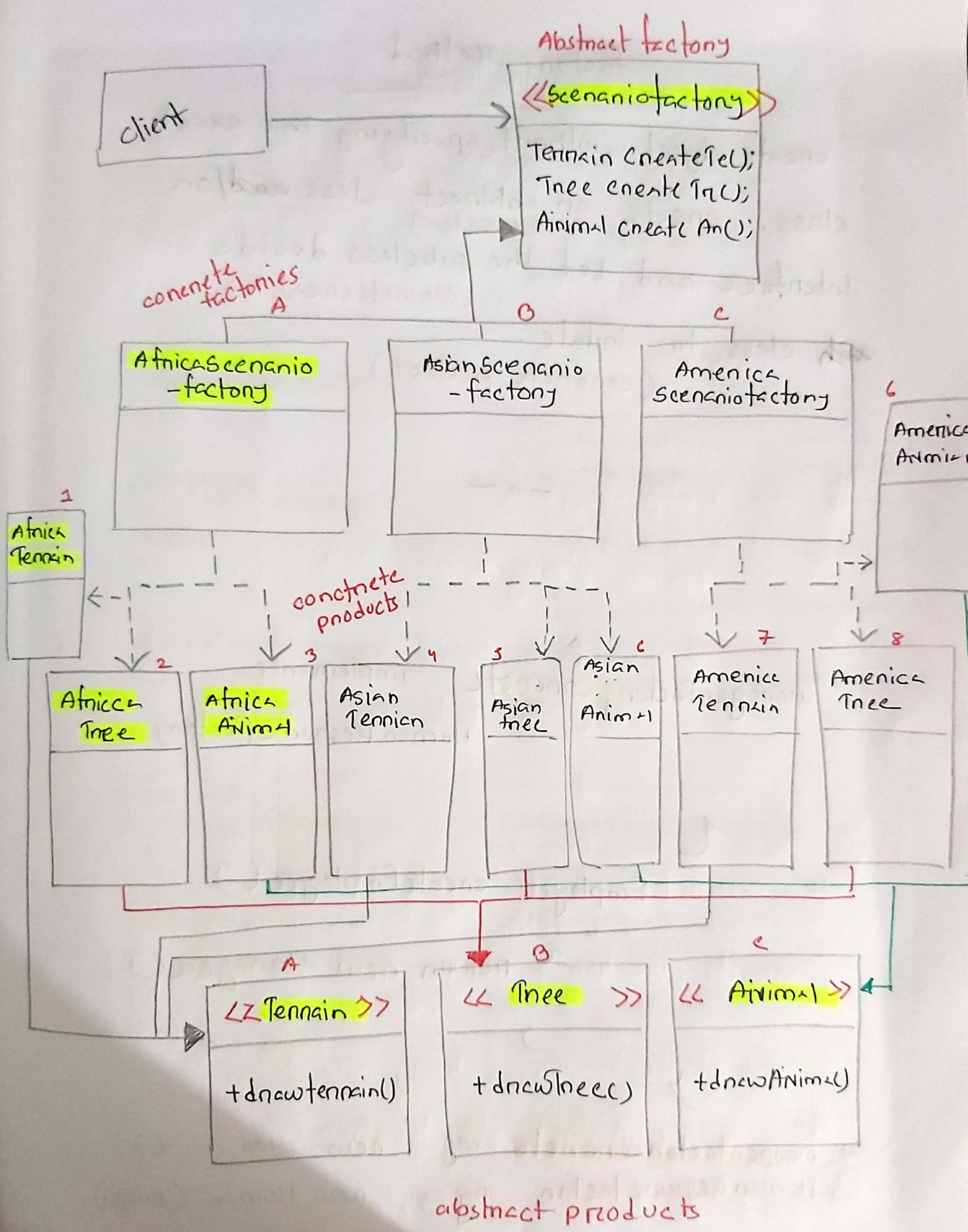
let produce families of related object without specifying their concrete class

#

```
AfricanScenarioFactory implements ScenarioFactory
{   Terrain  CreateTe {
        return new AfricaTerrain() ; }

    Tree  CreateTr {
        return new AfricaTree() ;  }

    Animal  CreateAn {
        return new Africa Animal () ;}
}
```

#

```
AfricaTerrain  implements Terrain {

    void drawTerrain( ) {

        }

    }
```


Abstract factory
client
<<Scenaniofactory>>
Terrain CreateTe();
Tree create Tr();
Animal create An();
concrete factories
A
B
C
Africa Scenanio -factory
Asian Scenanio -factory
America Scenaniofactory
6
America Animal
1
Africa Terrain
concrete products
2
Africa Tree
3
Africa Animal
4
Asian Terrian
5
Asian tree
6
Asian Animal
7
America Terrain
8
America Tree
A
<< Terrain >>
+drawterrain()
B
<< Tree >>
+drawTree()
C
<< Animal >>
+drawAnimal()
abstract products

# factory method

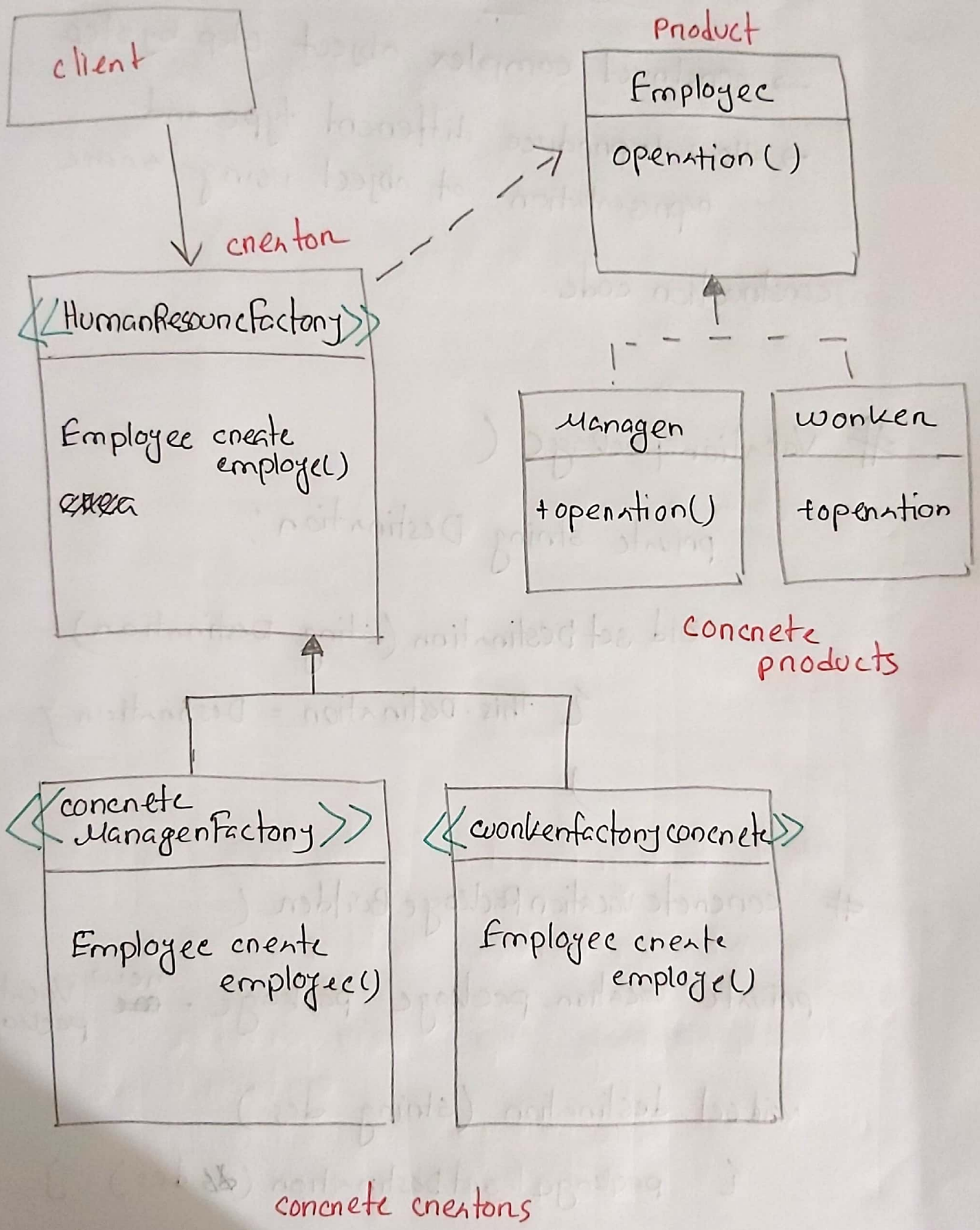
create object without specifying the exact class. create an abstract class (creator) or Interface and Let the subclass (concrete creator) decide class to initate. (concrete product)

— o —

```
managerFactory concrete implements
                    HumanResourceFactory
{
    Employee createEmployee()
    {
        return new Manager();
    }
}
```

1. → mangerfactorconcrete mg = new Man... ();
2. → HumanResourcefactor rg = new Human (mg);
3. → Employee em = rg.create
4. → em.operation()

~~rg.operation();~~

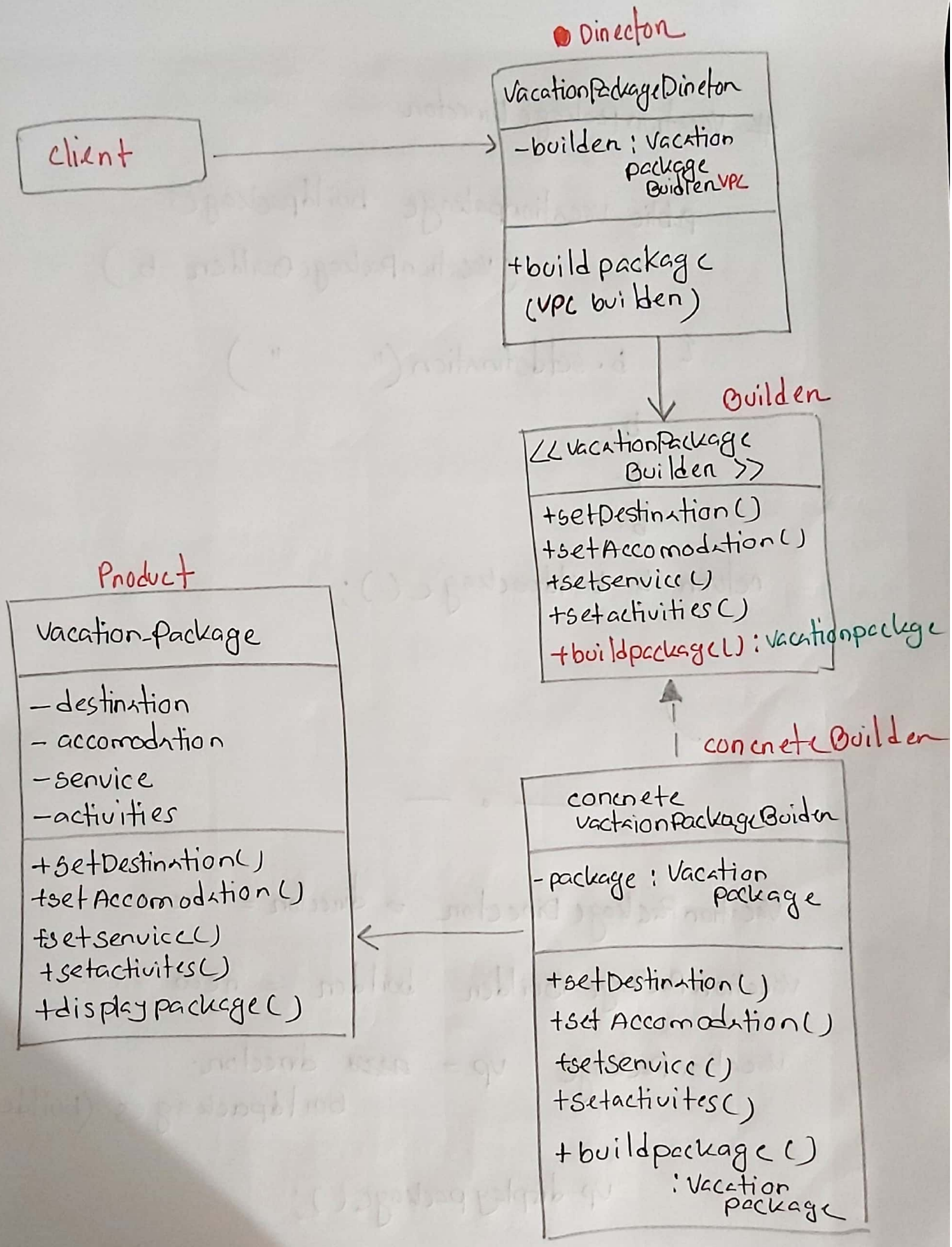

client
Product
Employee
operation()
creator
<<HumanResourceFactory>>
Employee create employee()
Manager
+operation()
worker
+operation
concrete products
<<concrete ManagerFactory>>
Employee create employee()
<<workerfactory concrete>>
Employee create employee()
concrete creators

# Builder Pattern

→ construct complex object step by step

→ allow to produce different type and representation of object using same construction code.

```
# Vacation package (
        private string Destination;
        void setDestination (sting Destination)
            { this.Destination = Destination }
        )

# concrete Vacation Package Builder (
    private Vacation_package package = new Vacation package()
    void set destination (string des)
        { package.setDestination (des) }

#void Vacationpackage Buildpackage ()
        { return Package }
```


Director
VacationPackageDirector
-builder : Vacation package Builder VPC
+build package (VPC builder)
client
Builder
<< VacationPackage Builder >>
+setDestination()
+setAccomodation()
+setservice()
+setactivities()
+buildpackage() : vacationpackage
Product
Vacation_Package
-destination
-accomodation
-service
-activities
+setDestination()
+setAccomodation()
+setservice()
+setactivites()
+displaypackage()
concrete Builder
concrete VactaionPackageBuilder
-package : Vacation package
+setDestination()
+set Accomodation()
+setservice()
+setactivites()
+buildpackage() : Vacation package

# Vacation Package Director

```
public VacationPakage buildpackages
        (VacationPackageBuilder b)
{
    b.setdestination("      ")
    b.
    b.
    b.
return b.buildpackage();

}
```

---

```
VacationPackageDirector director = new ...
VacationPackageBuilder builder = new ...
VacationPackage vp = director.
                     buildpackages(builde
vp.displaypackage();
```