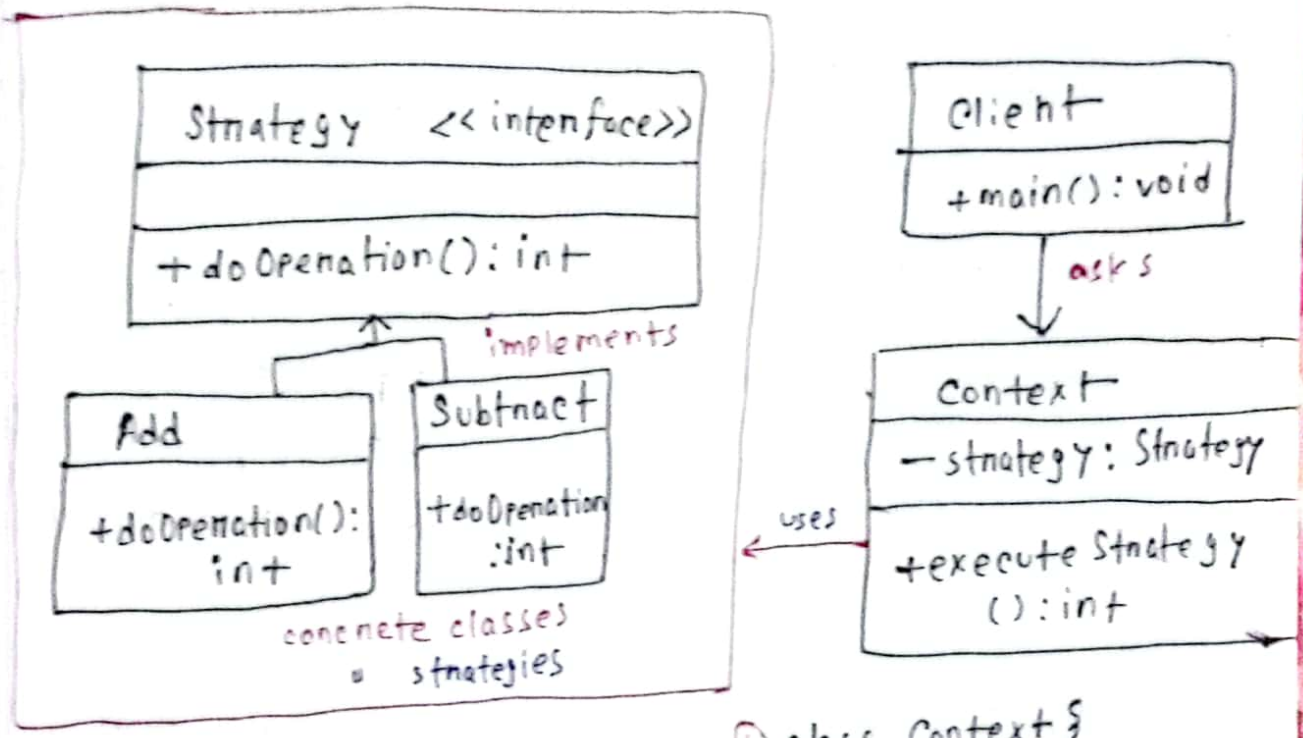
Strategy: (intent)
- define a family of algo
- put each algo in separate class
- interchange objects

UML:



code:

① interface Strategy {
int doOperation (n1, n2);
}

② Add implements Strategy{
doOperation (n1, n2){
ret n1+n2; }
}

③ Subtract implements Strategy{
doOperation (n1, n2) {
ret n1 - n2; }
}

④ class Context {
private Strategy strategy;
public Context (Strategy strategy)
{ this.strategy = strategy;
}
int executeStrategy (n1, n2)
{ ret strategy.doOperation (n1, n2); }

⑤ class Main() {
Context context = new
Context (new Add ());
context.executeStrategy(10,5

Pros

swap algo inside an object at runtime.

isolate the implementation details of an algo from client code

follows OCP

Cons

- clients should be awane of the diff. between the strategies to select a propen one.

- if only few algo (not changeable) strategy pattern makes code complicated.

Applicability:

when want to use diff. kinds of algo within an object and can switch ~~be~~ among them during runtime. when class has a massive control statements.

when you have a similan class with diff. behavion.

want to isolate business logic of class from implementation details of algo.

Chain of Responsibility:

- pass requests along a chain of handlers.

handler —receive requests→ process request / pass to next handler

Pros

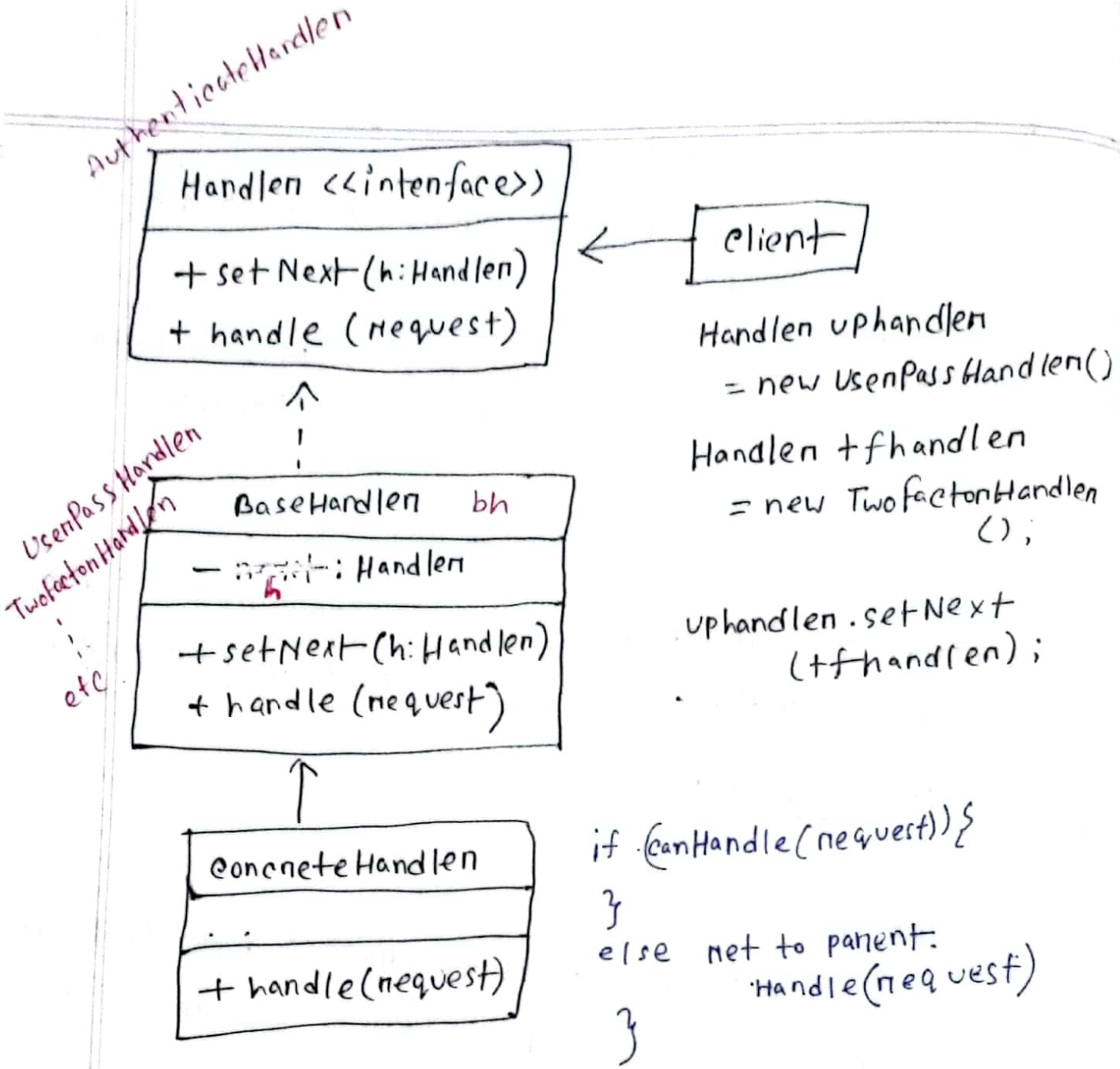
- control the order of request handling
- follow OCP and SRP

Cons

- some request may end up unhandled

Applicability:

- when your program expects to process diff. kinds of request, but type and sequence of requests are unknown.
- order can change at runtime.

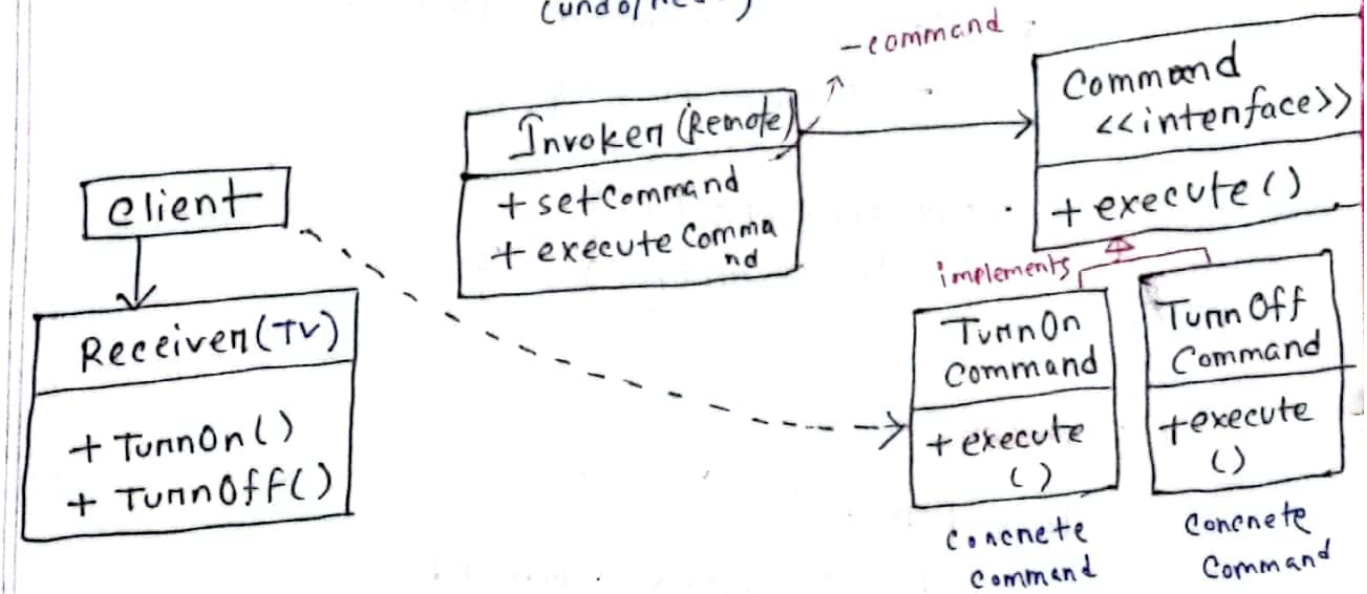

AuthenticateHandler
Handler <<interface>>
+ setNext(h:Handler)
+ handle(request)
client
Handler uphandler
= new UsernPassHandler()
Handler tfhandler
= new TwofactorHandler
();
uphandler.setNext
(tfhandler);
UsernPassHandler
TwofactorHandler
etc
BaseHandler bh
- next: Handler
+ setNext(h:Handler)
+ handle(request)
ConcreteHandler
+ handle(request)
if (canHandle(request)){
}
else net to parent.
Handle(request)
}

Command: turns a request into separate object (contains all info about the request)

Pros	Cons
follows SRP and OCP	complicated

Applicability:

1) parametrize objects with operations
2) execute operations remotely / schedule execution / queue operations
3) implement reversible operations. (on/off) (undo/redo)



// command interface

```
interface Command {
        void execute();
}
```

// Receiver

```
class TV {
        void Turnoff() { };
        void TurnOn() { };
}
```

// main

```
TV tv = new TV();
Command toncommand
= new TurnOnCommand
        (tv);
Remote RC = newRemote
        ();
RC.setCommmand
    (tOn command);
RC.executeCommand
        ();
```

// Concrete Command

```
class TurnOnCommand (TurnOffCommand) implements Command {

    private TV tv;
    public TurnOnCommand (TurnOffCommand) (TV tv) {
        this.tv = tv; }

    void execute() {
        tv.TurnOn() (Off); }
}
```

// Invoker

```
class Remote() {
    private Command command;
    public void set(Command) (Command command) {
        this.command = command; }
    public void executeCommand() {
        command.execute(); } }
```

Memento: - lets you save and restore previous state of an object

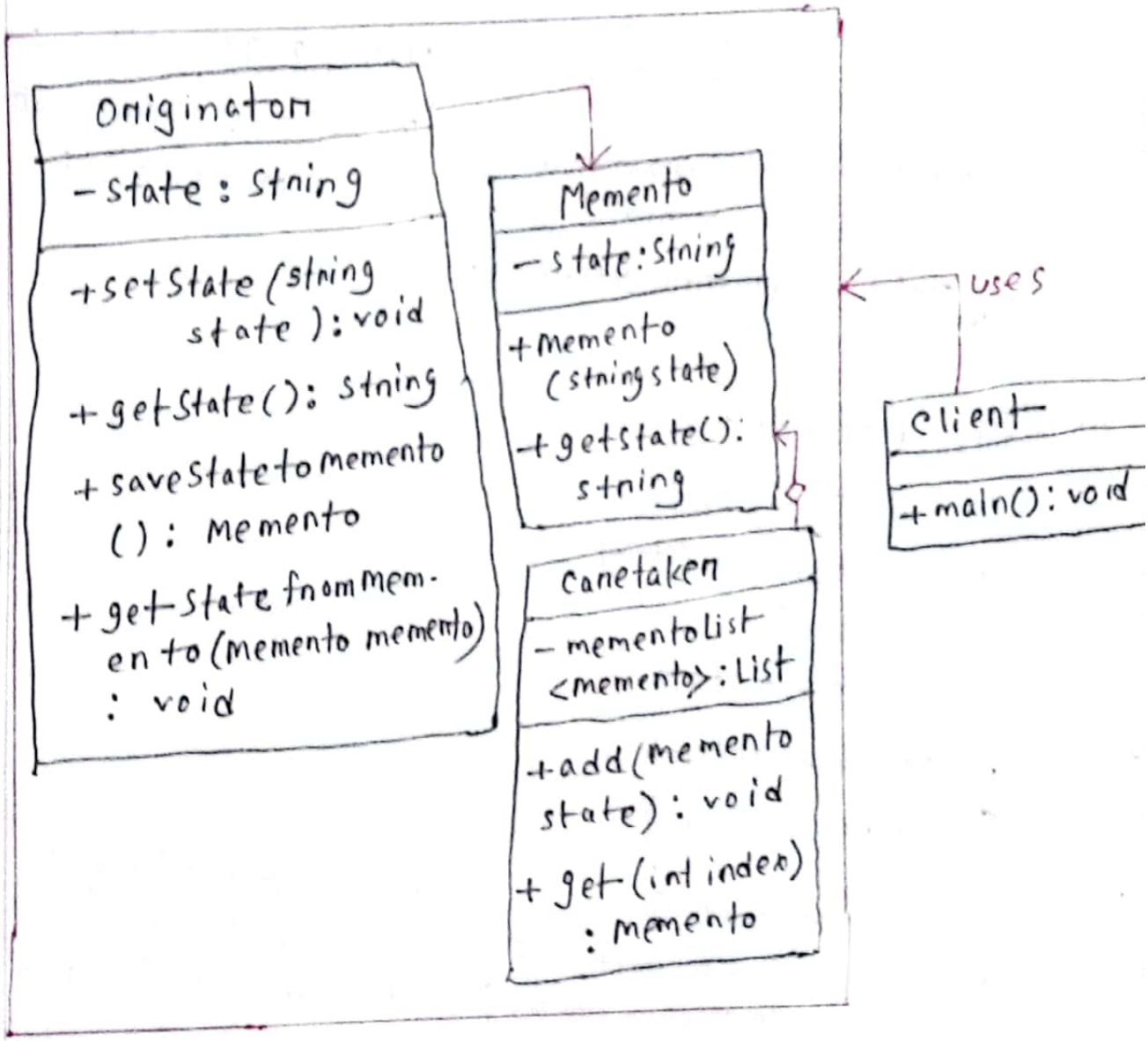
— not reveal implementation details.

Pros

- not violate encapsulation

Cons

- consume lots of RAM



(1)
```
class Originator {
    private String state;

    void setState(String state)
    { this.state = state; }

    String getState {
      return state; }

    Memento saveStateToMemento
    { return new Memento
          (state); }

    void getStateFromMemento
    (Memento memento) {
    state = Memento.getState();
}
```

(2)
```
class Memento {
  private String state;
  public Memento (String
     state) {
  this.state = state; }

  public String getState()
  { return state; }
```

(3)
```
class Caretaker {
 private List <Memento> ml;
 void add (Memento state) {
  ml.add (state); }

public Memento get
      (int idx) {
 ret ml.get(idx);
 }
```

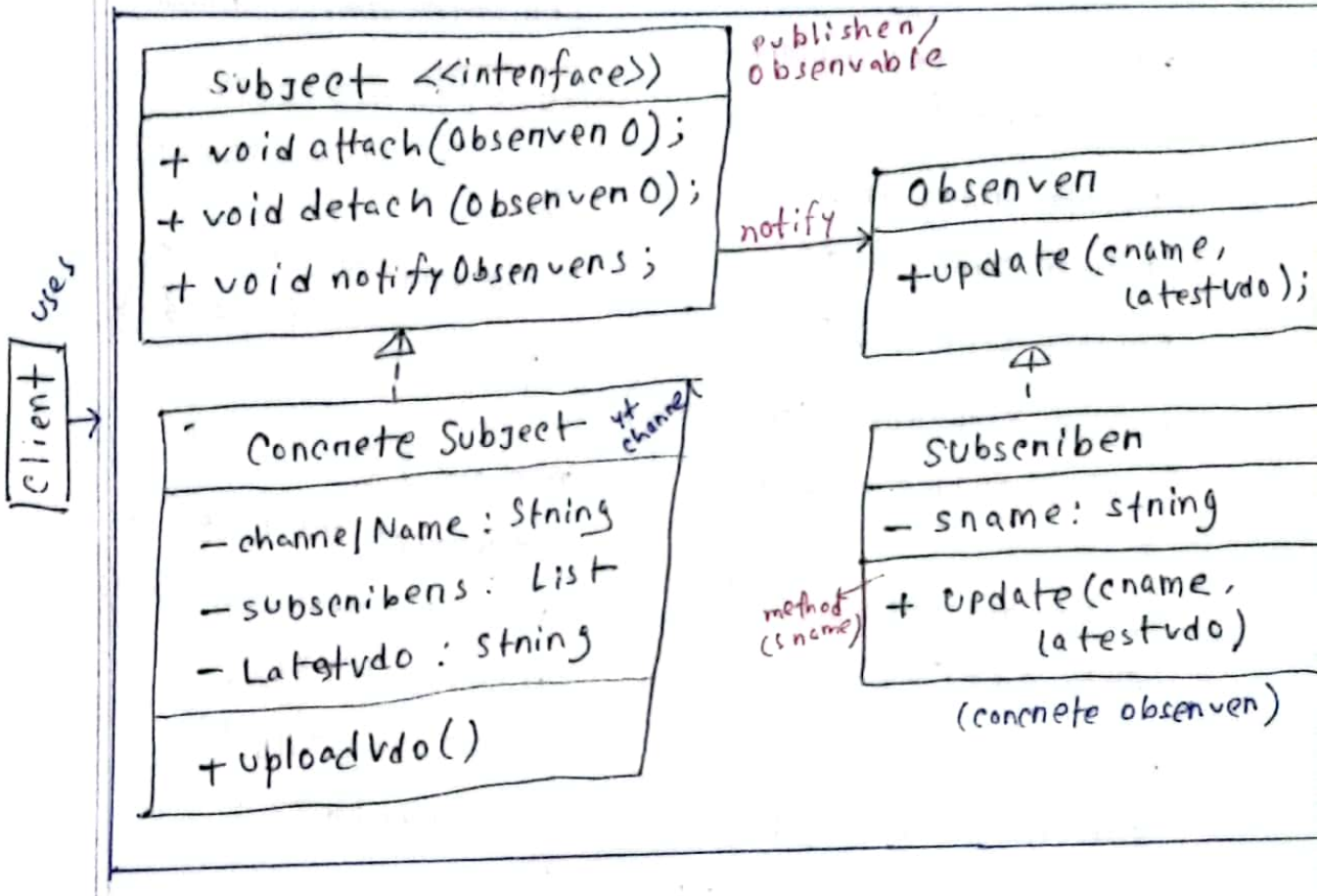
(4)
```
main() {
 Originator o = new O();
 Caretaker c = new C();

 o.setState("S1");
 c.add (o.saveStateToMemento());
 o.setState("S2");
 c.add (o.saveStateToMemento());
 print (o.getState());
 o.getState fromMemento
     (c.get(1));
 }
```

pub-sub system

subscribens

Obsenven: lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're obsenving.

a yt channel



Applicability:

- change of one object's state → change other objects, change dynamically.
- object observe others for a limited time.

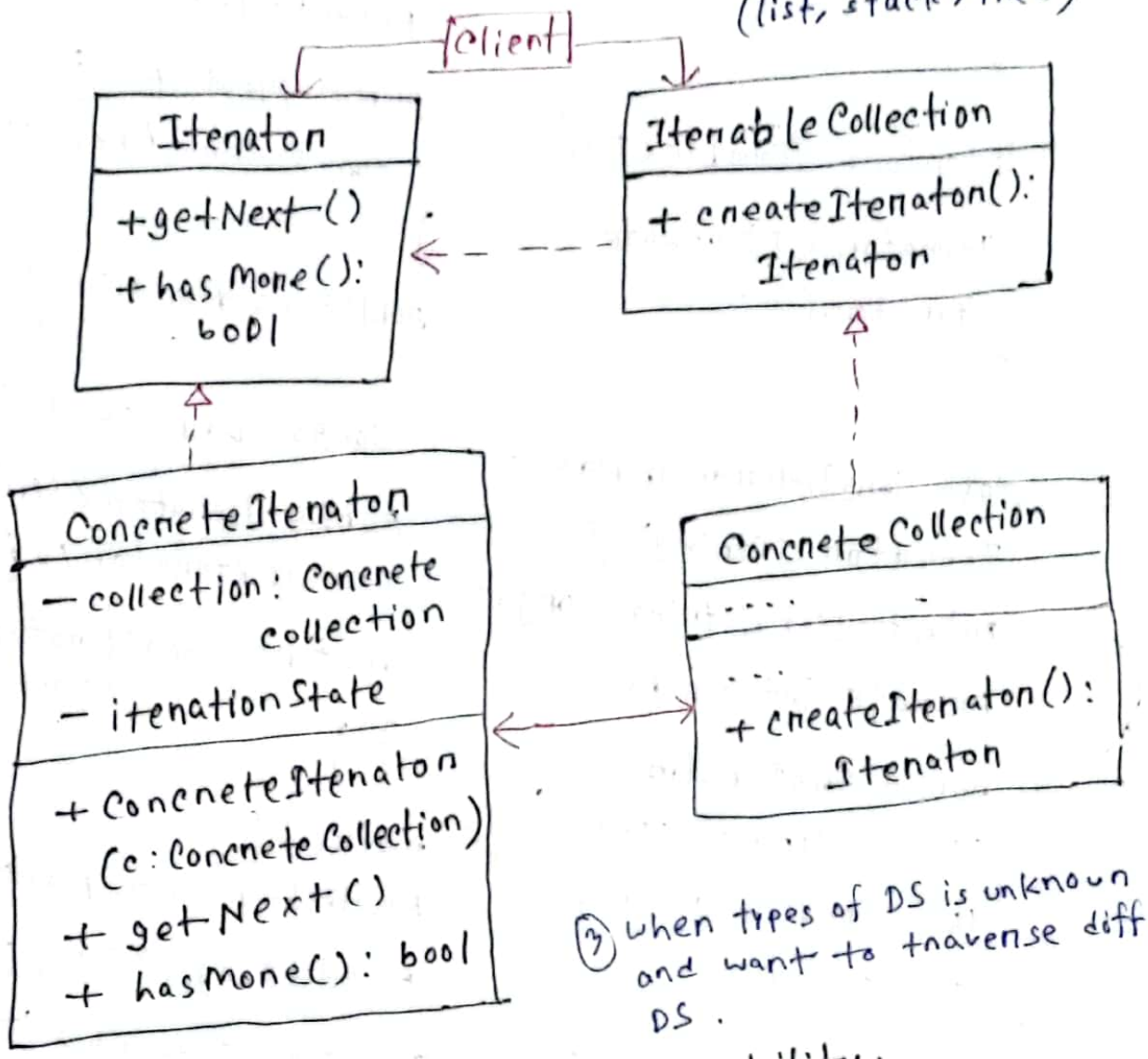
Pros: - Follows OCP. - Establish relation between objects at Puntime.

Cons: subscribens notified in random order.

```
Interface Subject {
    void attach(O);
    void detach(O);
    void notifyObservens();
}

class Ytchannel implements
 subject {
 private String cname;
 private string Lvdo;
 private List<O> sub;
 public Ytchannel(Stn cname)
{ this.cname = cname;
 }
 public void attach(O){
  sub.add(O); }
 void detach(O) {
  sub.remove(O); }
 void notifyObservens(){
  for(O o : sub) {
   o.update(cname, Lvdo);
  } } void upVdo(Stn vname){
}       Lvdo = vname;
}       notifyObservens();
      }
```

```
interface O {
void update(cname,
            lvdo);
 }

class sub implements O{
private string Sname;
(method)
sub(string sname){
 this.sname = sname;
}
void update(cname,
            lvdo){ }
 }

main() {
 Ytchannel Y = new();
 sub S1 = new sub("S1);
 Y.attach(S1);
 Y.upVdo("AI");
 S1.update();
}
```

Iterator: traverse elements of a collection not expose its underlying representation (list, stack, tree)



Applicability:

1. when your collection has a complex data structure but you want to hide its complexity from clients.
2. reduce duplication of traversal code
3. when types of DS is unknown and want to traverse diff. DS.

Pros:

- follows OCP and SRP

Cons:

- overskill if collection simple
- less efficient

(1)
```
interface Iterator<T>{
  bool hasNext();
  T next();
}
```

(2)
```
interface Iterable<T>{
  Iterator<T> create
  Iterator();
}
```

(3)
```
class SongIterator implements
Iterator<string>{
private List<string> songs;
private int curidx;

public SongIterator
(List<string> songs){
this.songs = songs;
 curidx = 0;
}

public String next(){
if(hasNext()){
String song = songs.
    get(curidx);
 curidx++;
 ret song; }
ret null; } }
```

(4)
```
class Playlist implements
Iterable<String>{

private List<String> songs;

public Playlist(){
 songs = new ArrayList<>();
}

public void addSong(String
          song){
 songs.add(song); }

public Iterator<String>
create Iterator(){
ret new SongIterator
          (songs);
} }
```

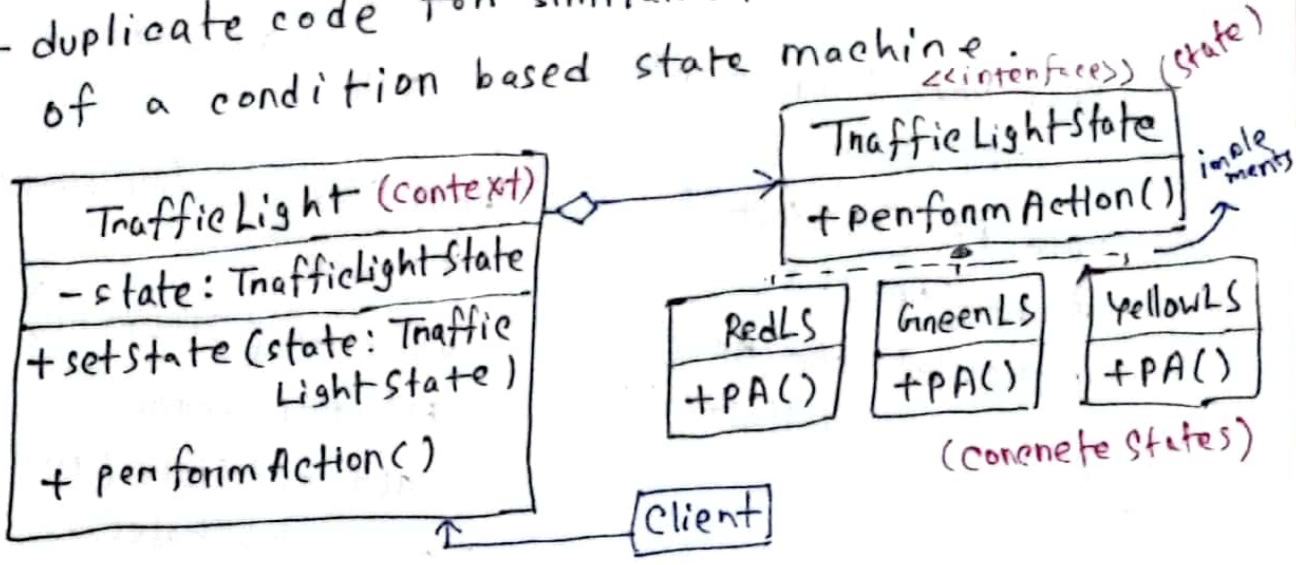
(5)
```
main(){
Playlist p = new Playlist();
p.addSong("S1");

Iterator<String> iterator
= playlist.createIterator();

while(iterator.hasNext()){
 String song = iterator.next();
 print(song);
} } }
```

State: object alter its behavior when its internal state changes.

Pros
- follows OCP and SRP
- simplify code
  eliminate bulky
  state machine
  conditionals

Cons
Overkill if a state machine has
- few states
- states rarely change

Applicability:
- object behaves differently depending on its current states, the number of state is huge, the state specific code changes frequently.
- class polluted with a lot conditionals (how class behaves according to class field's current value)
- duplicate code for similar states and transitions of a condition based state machine.



// context class

(1)
```
class TL {
    private TLState state;
    public TL() {
        state = new RLstate();
    }
    void setstate(TLState state){
        this.state = state; }
    void PA() {
        state.PA(this); }
}
```

// state interface

(2)
```
interface TLState{
void PA(TL Light);
}
```

// concrete state

(3)
```
class RedLS implements
        TLstate {
    void PA(TL Light){
    print("Red");
    Light.setstate(new
                GLstate());
    }
}
```

(4)
```
class GreenLS implements
    TLstate {
    void PA(TL light) {
        print("Green");
        light.setstate(new
                YLstate());
    } }
```

(5)
```
class YellowLS implements
    TLState {
    void PA(TL Light) {
        print("yellow");
        light.setstate(new
                RLstate());
    } }
```

(6)
```
main() {
    TL tL = new TL(
    tL.PA(); //red
    tL.PA(); // gree
    tL.PA(); // yel
    tL.PA(); // re
} }
```

Visiton: seperate algo. from the objects ← on which algo. operate

Pros:

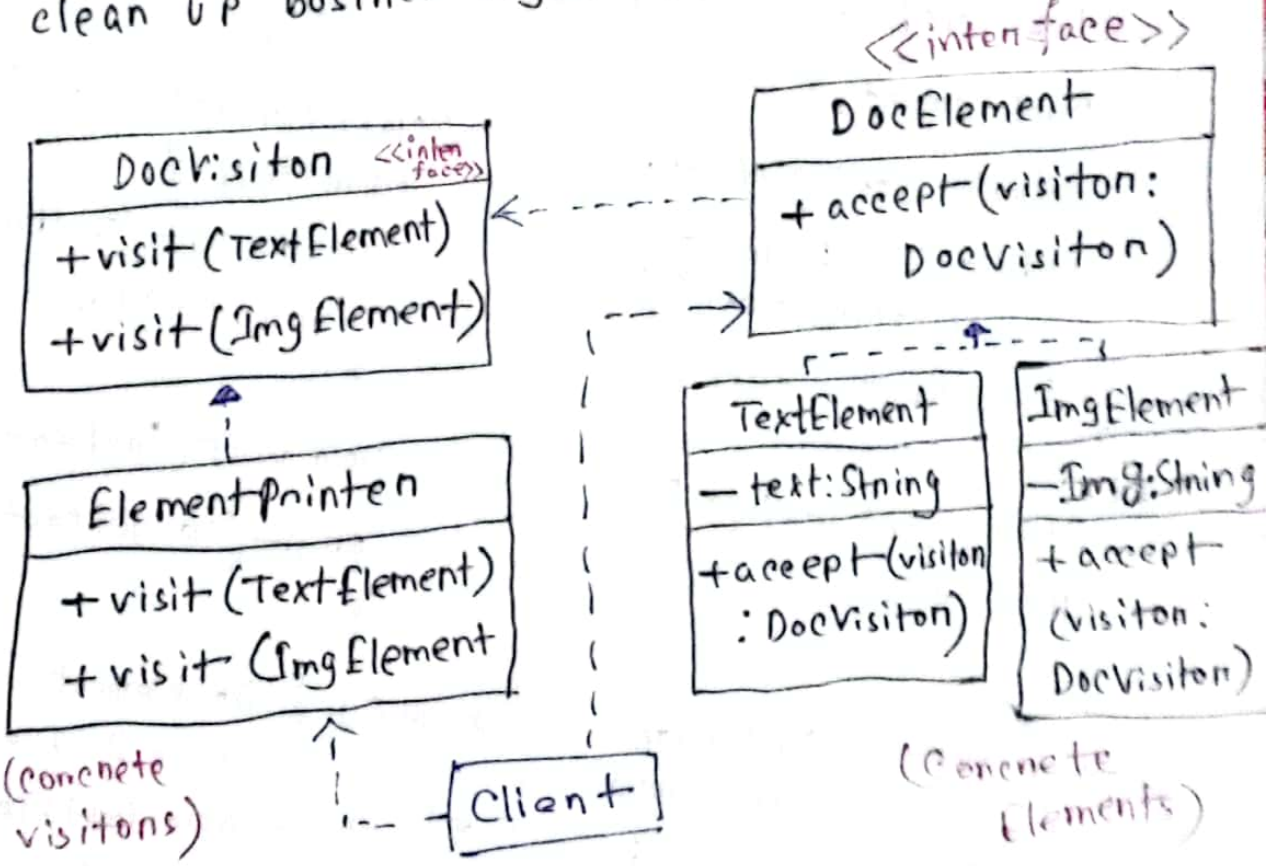
follow OCP and SRP

Cons

update all visitons each time a class add/remove to the element hiananchy

Applicability

- when perfonm an openation on all elements of a complex object stnuctune.
- when behavion makes sense only in some classes of a class hienanchy, but not in othens.
- clean up busines logic of auxiliany behavions



```
for (DocElement ele : ele);
    ele.accept(ep);
} } }
```

① interface DocVisitor {
void visit (TextElement te);
void visit (ImgElement ie);
}

~~class~~

② interface DocElement {
void accept (DocVisitor visitor);
}

③ class TextElement implements DocElement, {
private String text;
public TextElement (String text) {
this.text = text;
}
String getText() {
return text;
}
void accept (DocVisitor visitor) {
visitor.visit (this);
} }

④ class ImgElement implements DocElement
{ private String img;
public ImgElement (String img) {
this.img = img; }
String getImg() {
return img; }
void accept (DocVisitor visitor)
{ visitor.visit(this);
} }

⑤ class ElementPrinter implements DocVisitor {
void visit (TextElement te) {
print(te.getText());
}
void visit (ImgElement ie) {
print(ie.getText());
}

⑥ main() { DocElement[] ele =
new DocElement[3];
ele[0] = new TextElement ("hi");
ele[1] = new ImgElement ("img");
DocVisitor ep = new EP();