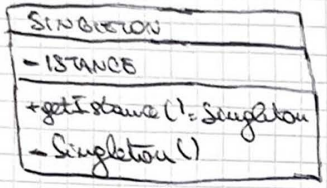
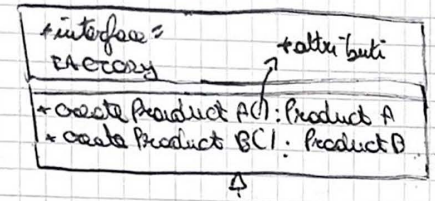
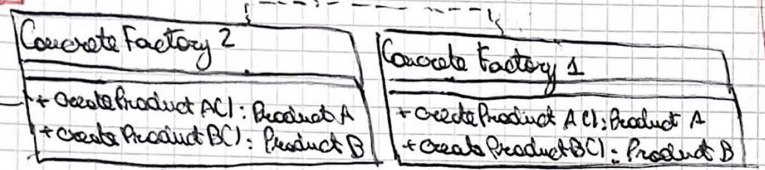
SINGLETON

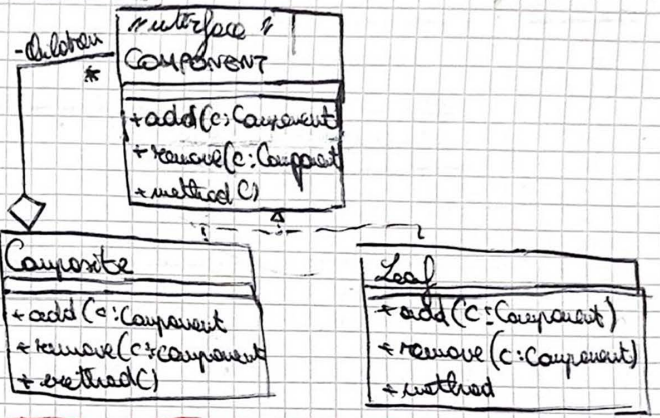


ABSTRACT FACTORY

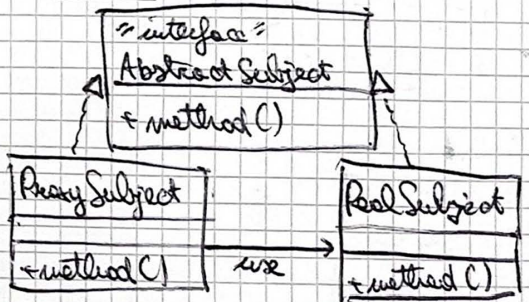




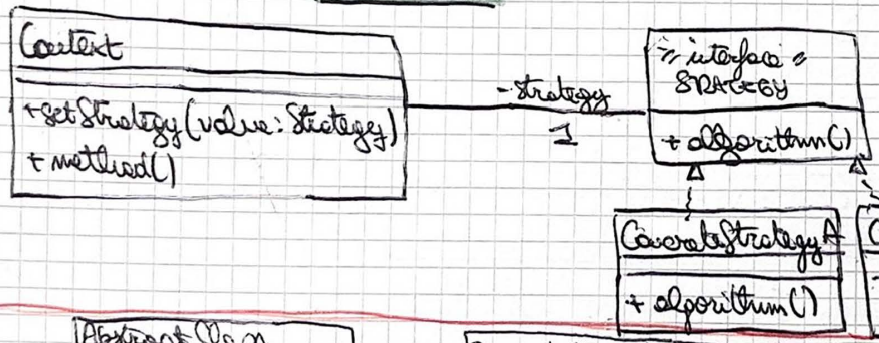
COMPOSITE



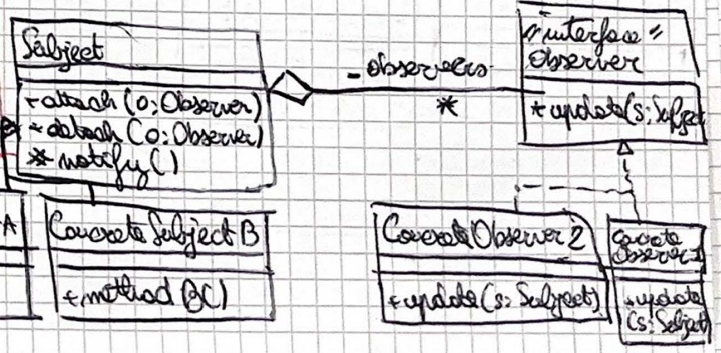
PROXY



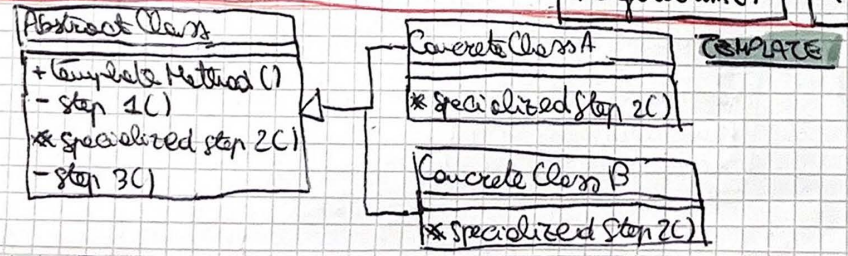
STRATEGY



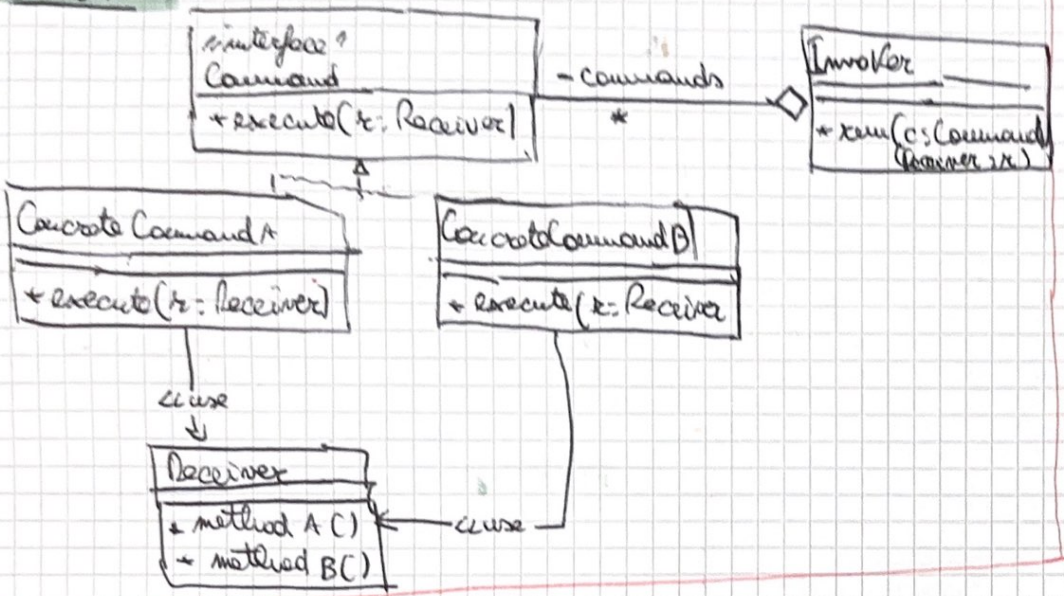
OBSERVER



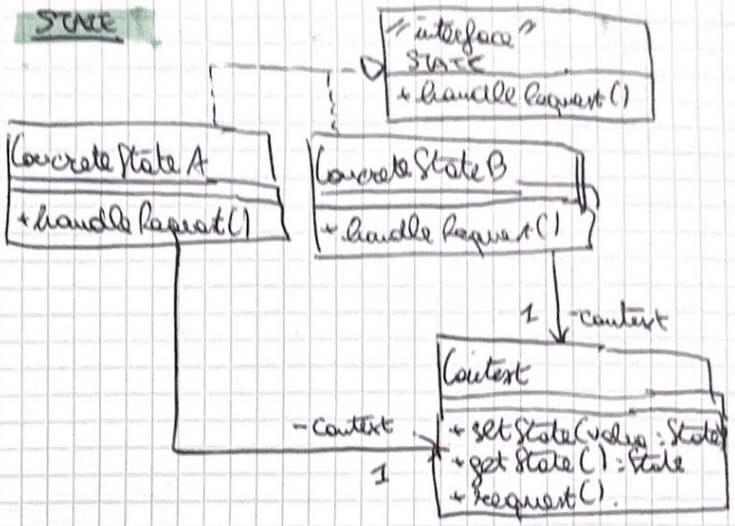
TEMPLATE



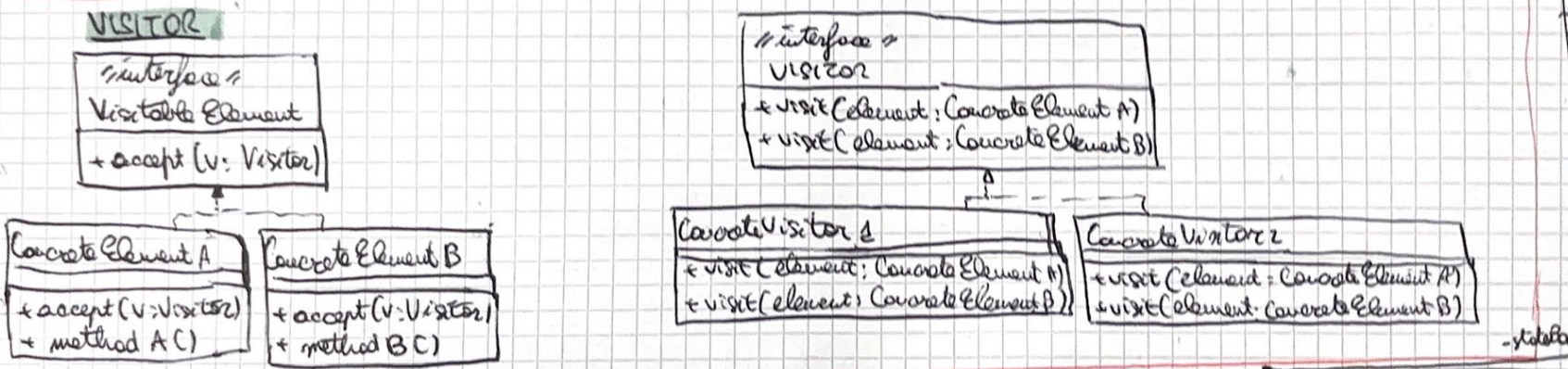
COMMAND

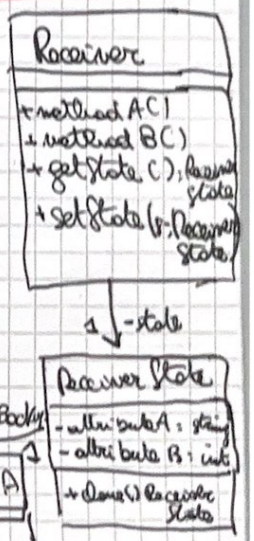


STATE

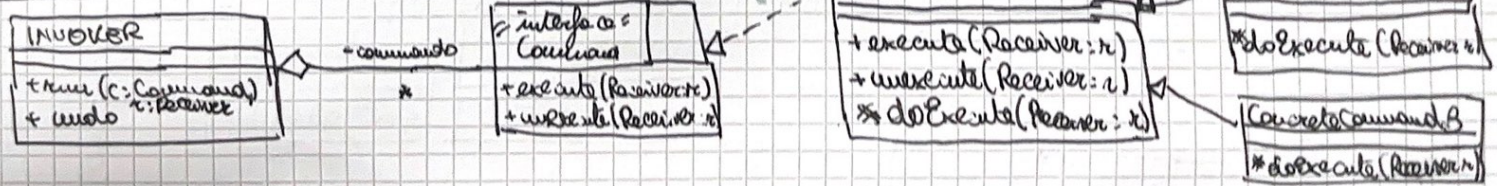


VISITOR





COMMAND + TEMPLATE (command sur state)



Codici Java dei DESIGN PATTERN

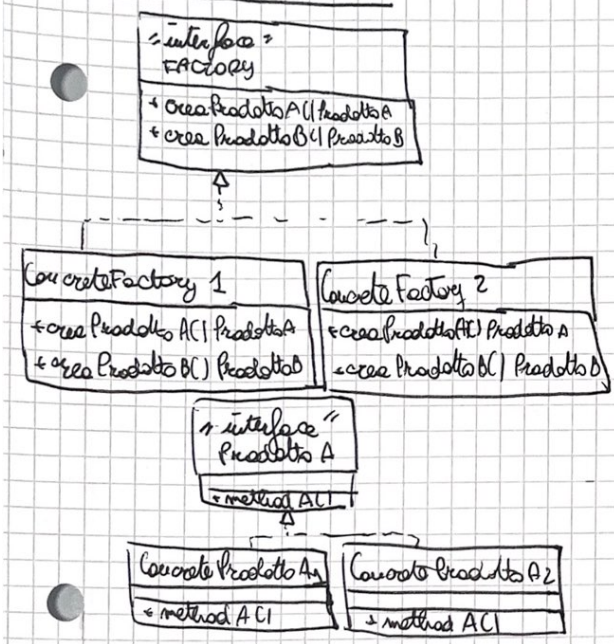
SINGLETON


```
private static BITCOIN istance = null;
public static Bitcoin getIstance() {
    if (istance == null)
        istance = new Bitcoin();
    return istance;
private Bitcoin () {
    COSTRUTTORE PRIVATO.
}
```

TEST

```
public void test {
  Bitcoin istance =
      Bitcoin.getIstance();
  istance.operation();
  ...
}
```

ABSTRACT FACTORY

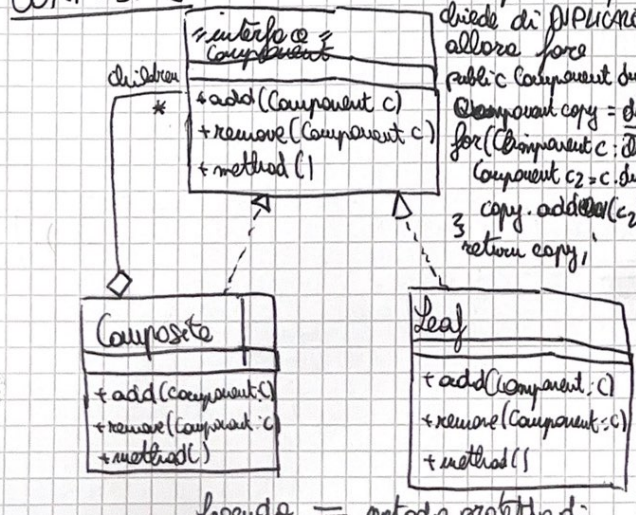



```
public ProdottoA creaProdottoA() {
    return new ProdottoA();
}
public ProdottoB creaProdottoB() {
    return new ProdottoB();
}
```

TEST

```
public void test {
  Factory factory = new ConcreteFactory1();
  ProdottoA prodA = factory.creaProdottoA();
  prodA.methodA();
  prodA.methodB();
  ProdottoB prodB = factory.creaProdottoB();
  prodB.methodB();
}
```

COMPOSITE



sul componente + template chiede di DUPLICARE allora fare

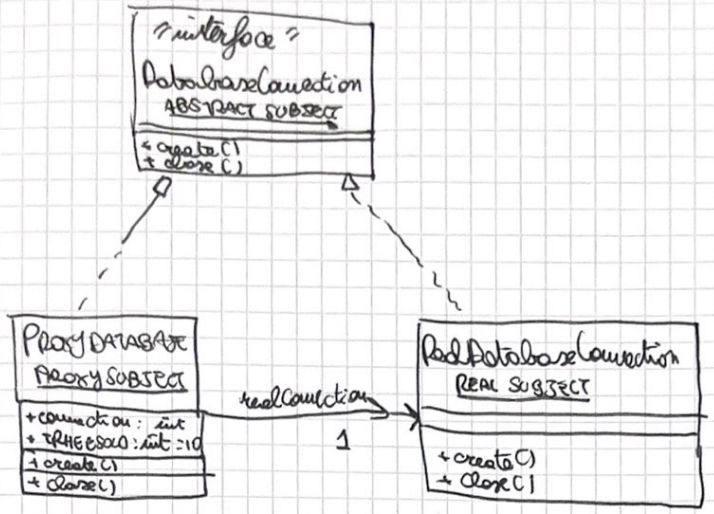
```
public Component duplicate(){
  Component copy = doDuplicate();
  for(Component c : children){
    Component c2 = c.duplicate();
    copy.add(c2);
  }
  return copy;
```

COMPOSITE

```
public void method() {
  for (Component c : children){
    c.method();
  }
  ...
}

public void add(Component c){
    children.add(c);
}

public Iterator<Component> getChildren(){
    return children.iterator();
}
```

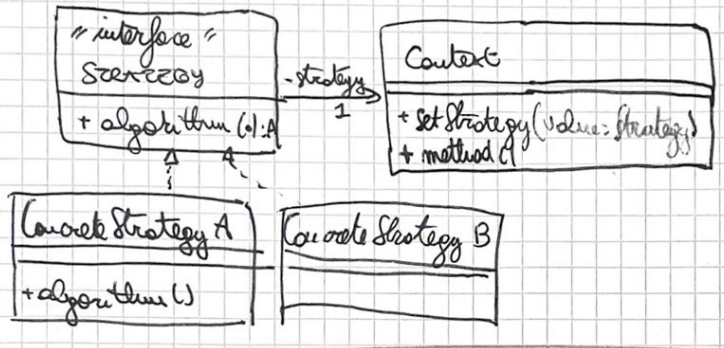
Legenda = metodo protetto di TEMPLATE

PROXY



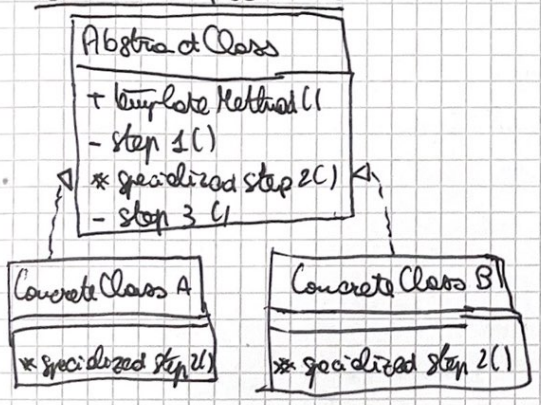

```
public void create() {
  if(connections == THRESHOLD) {
    return new Exception("Too many connections");
  }
  connections++;
  realConnection = new RealDatabase(connection);
  realConnection.create();
}
public void close() {
  realConnection.close();
  connections--;
}
```

STRATEGY




```
public A algorithm() {
```

TEMPLATE METHOD

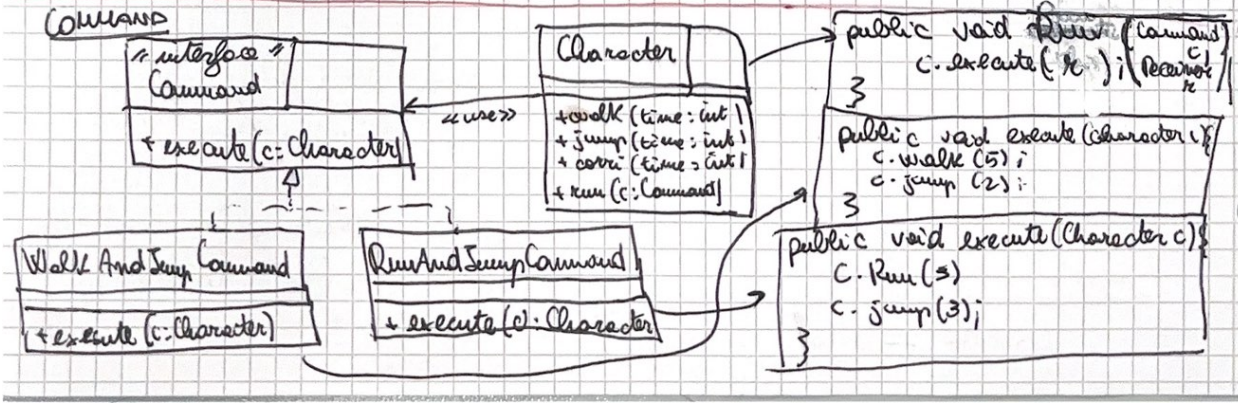



```
public void templateMethod() {
private step1();
protected specializedStep2();
private step3();
prote
```

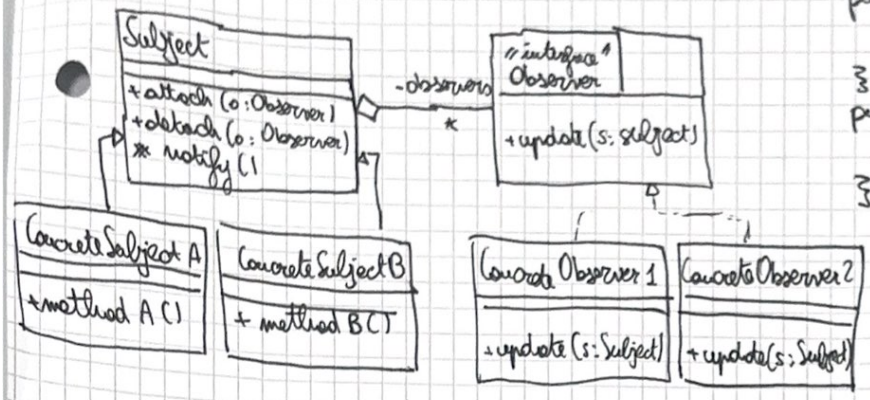
Se TEMPLATE diventa di duplicare public AbstractClass duplicate()

```
RealClass
protected void specializedStep2() {
    return new ClasseFiglia();
}
```

COMMAND

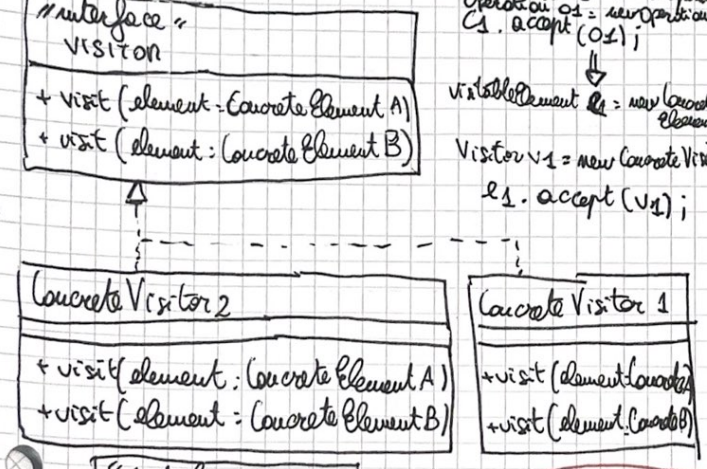


OBSERVER



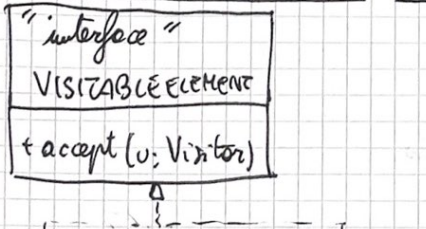

```
PER SUBJECT
public void attach(Observer o){
    observers.add(o);
}
public void detach(Observer o){
    observers.remove(o);
}
protected void notify(){
    for(Observer o: observers){
        o.update(this);
    }
}

PER CONCRETE SUBJECT A
public void method A(){
...
notify();
}
```

VISITOR

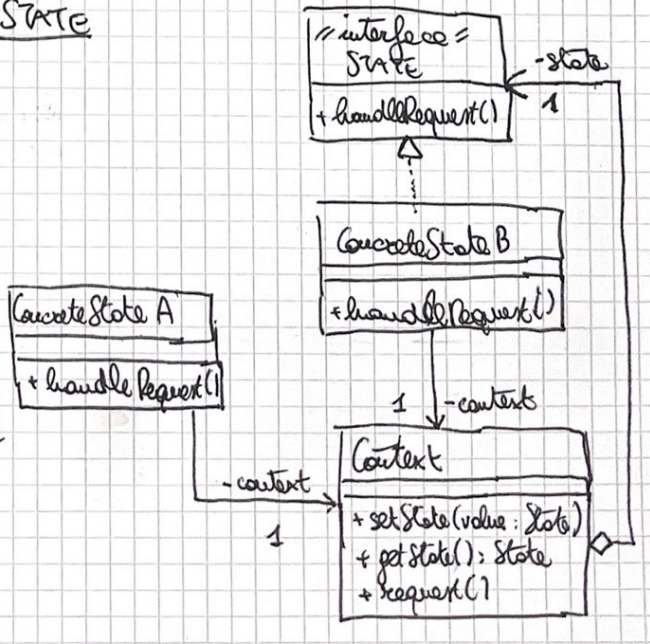



```
TEST
compositeA c1 = new(compositeA);
operationA o1 = new operationA();
c1.accept(o1);
⇓
visitableElement e1 = new(concrete elementA);
Visitor v1 = new ConcreteVisitor1();
e1.accept(v1);
```

```
PER CONCRETE ELEMENT A
public void accept(Visitor v){
    v.visit(this);
}
public void visit(concrete ElementA element){
...
element.method A();
...
}
public void visit(concrete ElementB element){
...
element.method B();
...
}
```



STATE




```
PER CONCRETE STATE B
public void handleRequest(Context context){
...
State nextState = ...;
context.setState(nextState);
}
PER CONTEXT
public void request(){
    state.handleRequest();
}
```