Conception orientée objet avancée

UML est un langage qui permet de répondre à un problème et qui propose des solutions.

Design patterns (cf ppt)

MVC = Model View Controler (différencier le model (la conception) et la view (les fenêtres qui permettront d'afficher les données))

MVC permet de représenter les liens entre le modèle et les vues afin de pouvoir faire les modifications adéquates.

3design pattern à connaitre :

- Creational
- Tructural
- Behavioral (comportementaux)

Design pattern :

Nom : singleton

Resout : garanti l'unicité d'une classe et l'unicité de son point d'accès.

(donc ça permet de s'assurer qu'à un moment T il n'existe qu'une seule instance de la classe)

Simulation thermomètre qui déclenche différentes alarmes selon la température (paramétrable).

Push(int temperature) ;

Public int T1 = ??, T2 = ??, T2 = ??;

Push(int température)

{

```
        Switch case temperature:
        Case T1:
        Lunch_alarm1;
        If(alarm2isOn == 1) switchOff(alarm2);
        Break;
        Case T2:
        Lunch_alarm2;
        If(alarm3isOn == 1) switchOff(alarm3);
        If(alarm2isOn == 1) switchOff(alarm1);
        Break;
        Case T3:
        Lunch_alarm3;
        If(alarm2isOn == 1) switchOff(alarm2);
        Break;
```

}

Ou

Abstract class **Observer**

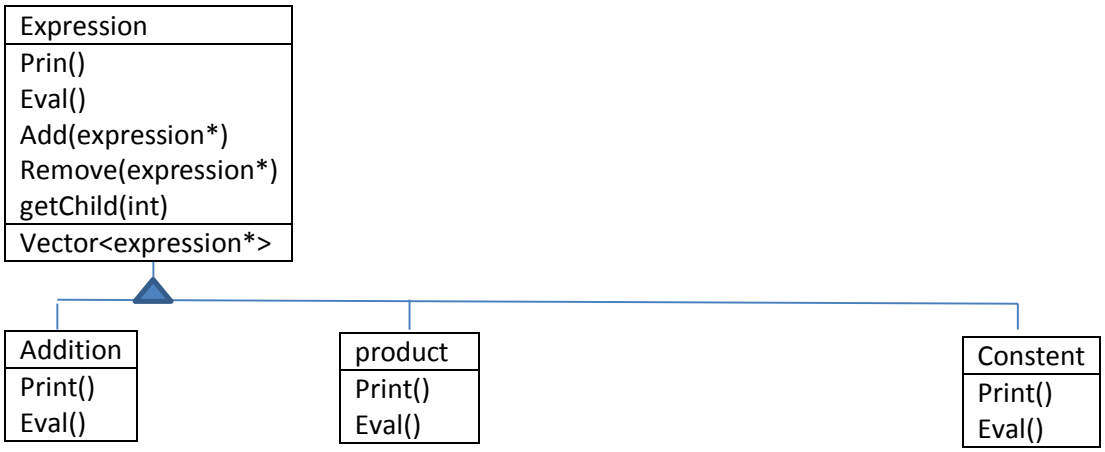
```
{
        Private int id;
        Private static int count =0;
        Private subject sujet;
        Public void update();
        Public getId()
        {
                Return this.id;
        }
        Public Observer(subject sujet)
        {
                Idd = Observer.count++;
                This.sujet = sujet;
        }
}
Abstract class subject
{
        Public Hashtble<int, observer>observers;
        Public attach(Observer obs)
        {
                This.observers.add(obs.id,obs);
        }
        Public detach(int id)
```

```
        {
                This.observers.remove(id);
        }
}
Class thermo extends subject
{
        Private float temperature;
        setState…
        getState…
}
Class AlarmA extends Observer
{
        Public update();
        Float temp = ((thermo)this.sujet).getState);
        (if temp > 35)
                Console.write("alarme A on");
        Else
                Console.write("alarme A off");
}
```

Design pattern: Composite (structurel)

Utile pour les interfaces graphiques (via arborescence structure)

Forme générale : 1classe mère avec des filles abstraites (chacune implemente une méthode)




```
class expression
{
        Vector <expression*> _exps;
        Public:
                Expression(){};
                Void add(epression* e)
                {       _exps.push_back(e);
                }
                Void remove(expression* e)
                {
                        Vector<expression*>::iterate it find(_exps,e);
```

```
                    If(it!= _exps.end())     _exps.remove(it);
                }
                Expression* getChild(int i)
                {
                    Return _xps[i];
                }
                Virtual void print()= null;
                Virtual double eval() = null;
};

Class addition : public expression
{
    Void print(){
    Getchild(0)->print();
    Cout <<" + ";
    Getchild(1)->print();
    }

    Double eval(){
    Return getchild(0)->eval()+getchild(1)->evl();
    }
};

Class constant : public expression{
    Double value;
    Public :
    Constant(double val): value(val){}
    Void print(){ cout<< value ;}
    Double eval() {return value ;}
};

Int main()
{
    Constant c(50),d(12,5);
    Addition a;

    a.add(&c);
    a.add(&d);
    a.print();
    cout<<a.eval()<<endl;
}
```