

Validation de l'architecture

www.thierrycros.net

Ce chapitre présente la suite de l'itération d'élaboration dans laquelle nous obtiendrons un premier prototype chargé de valider la classe `Calculateur` et la classe `ContrôleCalcul` du sous-système `Calcul`, donc l'architecture (en particulier la mise en œuvre du pattern `Observateur`). L'interface utilisateur est basée sur la hiérarchie `ios` de la bibliothèque standard du C++.

Phases / Activités	Jul 99										
	1	2	3	4	5	6	7	8	9	10	11
Elaboration calculette											
<i>Itération d'élaboration</i>											
Expression de besoins		■									
Analyse			■								
Conception/Architecture				■							
Programmation/Test					■						

Figure 3-1 : Planning des activités d'implémentation et de test

Points étudiés

Création d'une classe applicative : la fonction `main()` devient uniquement le point d'entrée de l'exécutable

Programmation des interfaces et classes de réalisation en langage C++

Développement et test de composants réutilisables (ceux du sous-système `Calcul`).

Les sous-systèmes obtenus dans l'activité précédente sont repris. Avant de passer à la programmation elle-même, il est nécessaire de parachever la conception.

Le pattern Observateur est basé sur deux sur-classes : Sujet et Observateur. La mise en œuvre du pattern est particulièrement simplifiée.

```
# ifndef __SUJET_H_
#   define __SUJET_H_

class Observateur; // necessaire et suffisant pour Sujet.h
# include <iostream.h>

class Sujet
{
public:
    Sujet()
    { sonObservateur = 0; };
    virtual ~Sujet()
    {};
    virtual void attache(Observateur *o)
    { sonObservateur = o; };
    virtual void detache(Observateur *o)
    { sonObservateur = 0; };
    void notifie(void);

private:
    Observateur * sonObservateur;
};

# endif __SUJET_H_
```

Le fait d'écrire

```
class Observateur;
```

permet d'éviter une dépendance avec la définition de classe Observateur, ce qui est le cas dans

```
# include "Observateur.h"
```

Ainsi, les modules qui incluent "Sujet.h" n'incluent pas automatiquement "Observateur.h". Par contre, la mise en œuvre dans le corps du module Sujet impose la définition complète de la classe Observateur.

```
# include "Sujet.h"

# include "Observateur.h"

void Sujet :: notifie(void)
{
    if (sonObservateur != 0)
        sonObservateur -> miseAJour();
}
```

Ici, l'instruction

```
sonObservateur -> miseAJour();
```

doit être validée par le compilateur. Pour cela, il doit connaître la classe Observateur afin de vérifier que ce message est licite.

La classe `Observateur` déclare une opération `miseAJour(void)`. Ainsi, un objet doit fournir cette opération (sous forme de méthode) pour être considéré en tant qu'observateur.

```
# ifndef __OBSERVATEUR_H_
#   define __OBSERVATEUR_H_

class Observateur
{
public:
    virtual void miseAJour(void) = 0;
    virtual ~Observateur()
        {};
};

# endif
```

La validation de l'architecture s'intéresse

Au sous-système Calcul

Aux relations de type sujet-observateur.

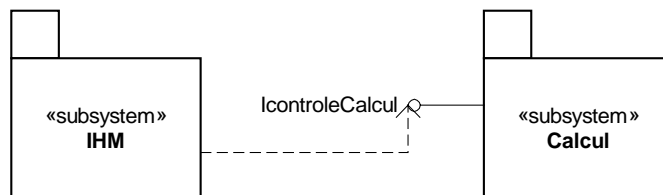


Figure 3-2 : Sous-systèmes

3.1 Conception détaillée Calculateur

Le concepteur s'intéresse maintenant aux éléments de modélisation qui constituent le cœur de l'application : les objets métier.

La classe `Calculateur` doit réaliser l'interface `Icalculateur`. Elle doit aussi répondre aux exigences non-fonctionnelles exprimées dans les cas d'utilisation sous forme de valeurs marquées. Cet aspect est traité ici car il correspond probablement à des choix de programmation uniquement. Habituellement l'architecte doit le prendre en compte beaucoup plus tôt, lors de la spécification de l'architecture.

Avant de traiter les détails de mise en œuvre de la classe, il est nécessaire de préciser les services de l'interface. Par exemple, les types des paramètres sont-ils déterminés ? Les retours `double` sont-ils définitifs ?

L'opérateur devient une énumération, ce qui est le plus adapté à une collection de constantes liées logiquement. Les paramètres et retours des opérations sont de type `double` afin de simplifier l'utilisation de cette interface (le taux se contente de `float` mais peu importe).

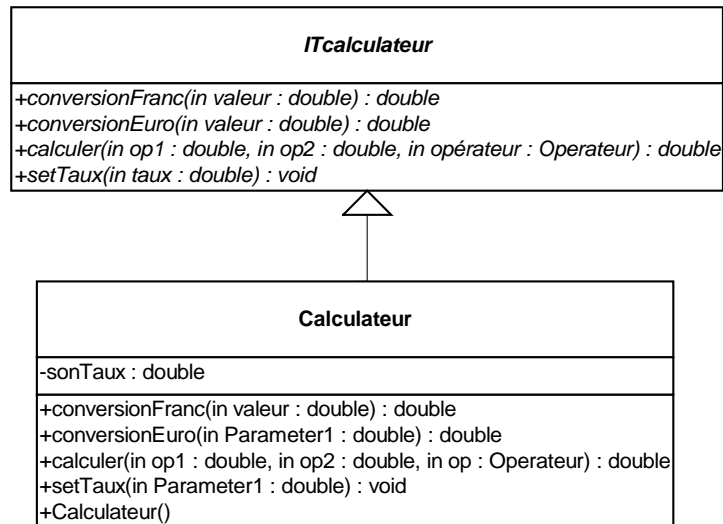


Figure 3-3 : Interface ITcalculateur et classe Calculateur

La classe Calculateur implémente ITcalculateur. En C++, cette relation correspond à un héritage public. En langage Java, les concepts d'interface et de classe, au sens réalisation, sont directement implantés dans la syntaxe. La visualisation de la relation de réalisation de l'interface est conforme à sa mise en œuvre en C++.

N La puissance du pré processeur du C++ permet de jouer avec sa syntaxe.
O // Quand le C++ danse la java
T # define interface class
E # define implements public
 ...
interface Icalculateur {...}; // dans Icalculateur.h
class Calculateur : **implements** Icalculateur {...}; // dans Calculateur.h

Les avantages de ces macro définitions sont-ils à la hauteur des inconvénients, le plus important étant peut-être le côté déroutant pour un programmeur débutant ?

L'interface ITcalculateur (interface du *type* Calculateur) est implémentée sous forme de classe abstraite dont les méthodes sont virtuelles pures (abstraites au sens UML). Par ailleurs, une telle classe ne possède pas d'attribut car il s'agit uniquement de spécification de service.

Le nom du sous-système est repris sous forme de préfixe des noms de fichiers (par exemple Calcul_Icalculateur.h) et de constantes (comme __CALCUL_ICALCULATEUR_H_). Les espaces de noms ne sont pas utilisés pour simplifier.

L'énumération Opérateur est publique car elle doit être connue des utilisateurs de l'interface.

Les lignes de code sont protégées par une directive de compilation conditionnelle, ce qui règle la question des includes d'include. Toutefois, inclure un fichier header dans un autre fichier header n'est pas anodin : cela constitue une dépendance.

```

// Calcul_Icalculateur.h
// Definition de l'interface dans le sous-système Calcul

#ifndef __CALCUL_ICALCULATEUR_H_
#define __CALCUL_ICALCULATEUR_H_

class Icalculateur
{
public:
    enum Operateur { PLUS, MOINS, MUL, DIV};
    virtual ~Icalculateur(void) {};
    virtual double conversionFranc(double valeur) = 0;
    virtual double conversionEuro(double valeur) = 0;
    virtual double calculer(double op1, double op2, Operateur op) = 0;
    virtual void setTaux(double taux) = 0;
};
#endif // __CALCUL_ICALCULATEUR_H_

```

N Le destructeur d'une classe qui représente en fait une interface doit être virtuel. En effet, la destruction d'un objet d'une sous-classe au travers d'un pointeur sur la sur-classe doit pouvoir invoquer le destructeur de la sous-classe. Si le destructeur de la sur-classe n'existe pas explicitement, le compilateur utilise celui par défaut.

T Si le destructeur existe mais n'est pas virtuel, le compilateur lie la destruction de l'objet, même d'une sous-classe, à ce destructeur. L'objet de la sous-classe n'est donc pas détruit correctement par le destructeur de sa véritable classe.

La classe `ITcalculateur` est la traduction de l'interface en C++. Point important : cette interface est *strictement identique* à celle de la classe. Autrement dit, la classe en tant qu'élément logique réalise l'interface au même titre que le composant (ici `Calcul_Calculateur.C`) en tant qu'élément physique.

Les opérations à concevoir ne posent pas de problème particulier. Pour simplifier, les erreurs sont renvoyées sous forme de retour égal à zéro. L'opération `calculer()` implante un algorithme basé sur un aiguillage.

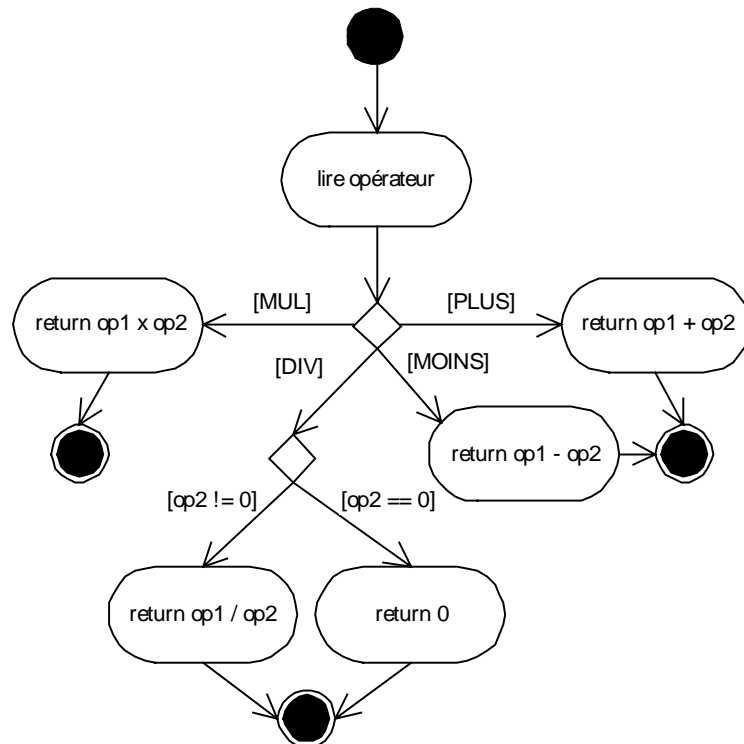


Figure 3-4 : Algorithme de l'opération calculer()

Le fichier `Calcul_Calculateur.h` représente la spécification du module d'implémentation de l'interface `Icalculateur`.

```

# ifndef __CALCUL_CALCULATEUR_H_
# define __CALCUL_CALCULATEUR_H_

# include "CALCUL_Icalculateur.h"

class Calculateur : public Icalculateur
{
public:
    Calculateur () {sonTaux = 1.0; };
    ~Calculateur() {};
    virtual double conversionFranc(double valeur);
    virtual double conversionEuro(double valeur);
    virtual double calculer(double op1, double op2, Operateur op);
    virtual void setTaux(double taux) ;

private:
    double sonTaux; // de conversion Franc / Euro
};

# endif // __CALCUL_CALCULATEUR_H_
  
```

La définition de la classe `Calculateur` est basée sur l'héritage de la classe `Icalculateur` qui joue le rôle d'interface. Le fichier `Icalculateur.h` fait partie de la réalisation et doit à ce titre être inclus avec précaution car il rend l'incluant dépendant d'une mise en œuvre donnée (en particulier la partie privée).

3.2 le sous-système "Calcul"

Ce sous-système est constitué de deux classes.

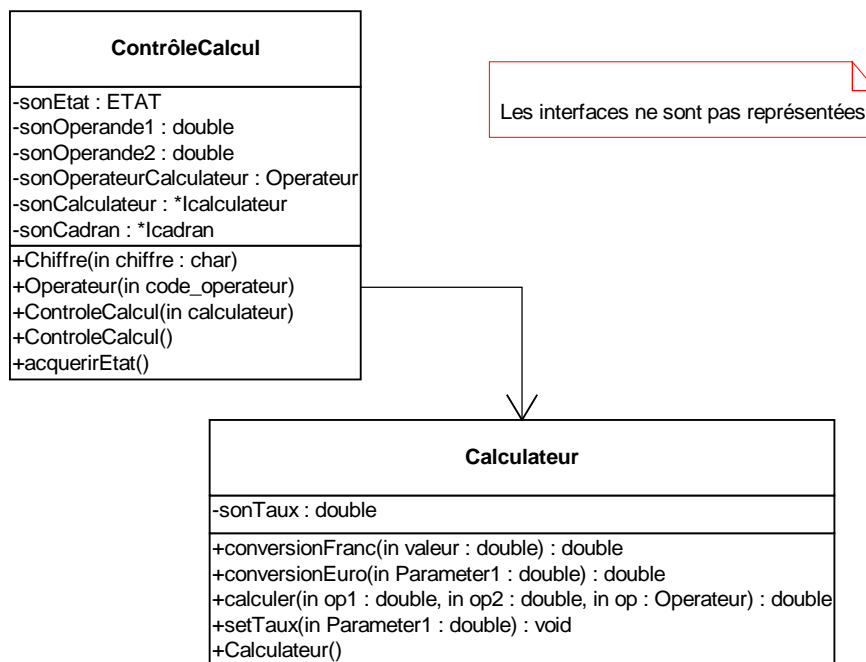


Figure 3-5 : Classes du sous-système Calcul

Le corps du module `calculateur` réalise les opérations sous forme de méthodes ou fonctions membres. La suite de ce paragraphe présente le fichier `calculateur.c` qui est la réalisation des services. Le constructeur est défini dans le fichier header.

Calculateur

```

#include "CALCUL_Calculateur.h"

double Calculateur :: conversionFranc(double valeur)
{
    return valeur * sonTaux;
}
  
```

```
double Calculateur :: conversionEuro(double valeur)
{
    if (sonTaux != 0.0) // test sur flottant... à surveiller
        return valeur / sonTaux;
    else
        return 0.0;
}
```

Le test sur flottant est, de façon générale, à surveiller : les égalités sont de la forme $(x - y) < \text{epsilon}$ plutôt que $x == y$.

```
double Calculateur :: calculer(double op1, double op2, Operateur op)
{
    double resultat = 0.0;

    switch (op)
    {
        case PLUS:
            resultat = op1 + op2;
            break;
        case MOINS:
            resultat = op1 - op2;
            break;
        case MUL:
            resultat = op1 * op2;
            break;
        case DIV:
            if (op2 == 0.0) // test sur flottant
                resultat = 0.0; // solution rustique
            else
                resultat = op1 / op2;
            break;
    }

    return resultat;
}
```

Le cas de la division par zéro est traité sous forme de retour nul. Une solution à étudier serait de placer la calculette dans un état « erreur ». Alors, les relations entre le sous-système et ses utilisateurs devraient évoluer pour faire apparaître des affichages alphanumériques et non plus uniquement numériques.

```
void Calculateur :: setTaux(double taux)
{
    sonTaux = taux;
}
```

Ici, le paramètre pourrait être vérifié en terme de cohérence de la valeur (supérieur à zéro...).

La classe ContrôleCalcul

La classe `ContrôleCalcul` réalise l'interface `IcontrôleCalcul`. Elle offre ainsi les méthodes de traitement des chiffres et opérateurs. Cette classe est indépendante de la bibliothèque IHM utilisée dans l'interface utilisateur.

L'interface `IcontrôleCalcul` joue au passage le rôle de facade du sous-système. En effet, `CALCUL` est constitué de deux classes `ContrôleCalcul` et `Calculateur`. Rien n'empêche l'utilisation directe de la classe `Calculateur`. Toutefois, un utilisateur (au sens classe utilisatrice) est guidé par l'interface de la classe `IcontrôleCalcul`.

```
# ifndef __CALCUL_ICONTROLECALCUL_H_
#  define __CALCUL_ICONTROLECALCUL_H_

# include "Sujet.h"

class IcontroleCalcul : public Sujet
{
public:
    IcontroleCalcul(){};
    virtual ~IcontroleCalcul(){};
    virtual void Chiffre(char chiffre) = 0;
    virtual void Operateur(char operateur) = 0;
    virtual double acquerirEtat(void) = 0;
};

# endif // __CALCUL_ICONTROLECALCUL_H_
```

Le code rappelle qu'un `contrôleCalcul` est un objet sujet. Autrement dit, il peut être observé - et il sait qu'il peut l'être - sans toutefois connaître la nature précise des observateurs. A ce titre, les objets de ce type doivent fournir une méthode `acquerirEtat(void)`.

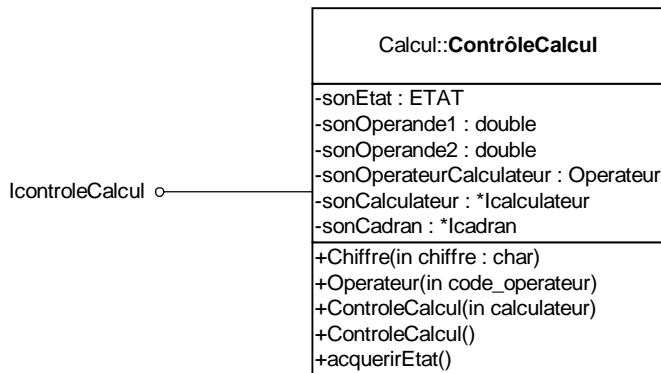


Figure 3-6 : Interface et classe ContrôleCalcul

La réalisation de l'interface `IcontroleCalcul` est confiée à la classe `ControleCalcul`. Elle est définie dans le fichier header suivant.

```

# ifndef __CALCUL_CONTROLECALCUL_H_
# define __CALCUL_CONTROLECALCUL_H_

# include <string>

# include "Sujet.h"
# include "CALCUL_IcontroleCalcul.h" // pour implementation

# include "CALCUL_Icalculateur.h" // pour utilisation, pas implementation

class ControleCalcul : public IcontroleCalcul
{
public:
    ControleCalcul();
    ControleCalcul(Icalculateur *leCalculateur);
    ~ControleCalcul() {};
    void Chiffre(char chiffre);
    void Operateur(char operateur);
    virtual double acquerirEtat(void);
private:
    enum Etat {OP1_0chiffre, OP1_nchiffres,
               OP2_0chiffre, OP2_nchiffres,};
private:
    Etat sonEtat;
    string sonOperande1;
    string sonOperande2;
    double sonResultat;
    Icalculateur::Operateur sonOperateurCalculateur;
    Icalculateur *sonCalculateur;
};

# endif // __CALCUL_CONTROLECALCUL_H_

```

Le résultat des opérations sous-traitées à l'objet calculateur est conservé dans l'attribut `sonResultat` car la méthode `acquerirEtat()` l'utilise.

Le listing du corps du module `ControleCalcul` reprend la mise en œuvre du pattern (mise en œuvre simplifiée) du côté observateur.

```

# include <iostream.h>
# include <stdlib.h>

# include "CALCUL_ControleCalcul.h" // pour implementation

ControleCalcul :: ControleCalcul()
    : sonEtat (OP1_0chiffre), sonOperande1(""), sonOperande2(""),
      sonOperateurCalculateur(Icalculateur::PLUS),
      sonCalculateur(0) , sonResultat(0.0)
{
    cerr << "ControleCalcul :: ControleCalcul()\n";
}

```

Le constructeur sans paramètre permet d'initialiser proprement l'objet. Le message envoyé à l'acteur via l'objet `cerr` de la bibliothèque standard a pour but de « tracer » la création des objets.

```
ControleCalcul :: ControleCalcul(Icalculateur *leCalculateur)
: sonEtat (OP1_0chiffre), sonOperande1(""), sonOperande2(""),
  sonOperateurCalculateur(Icalculateur::PLUS),
  sonCalculateur(leCalculateur), sonResultat(0.0)
{
  cerr << "ControleCalcul :: ControleCalcul(leCalculateur)\n";
}
```

Ce constructeur reçoit un objet calculateur en paramètre afin d'initialiser le lien.

```
void ControleCalcul :: Chiffre(char chiffre)
{
  switch (sonEtat)
  {
    case OP1_0chiffre:
      sonOperande1 += chiffre ;
      sonEtat = OP1_nchiffres;
      Sujet :: notifie();
      break;
    case OP1_nchiffres:
      sonOperande1 += chiffre ;
      Sujet :: notifie();
      break;
    case OP2_0chiffre:
      sonOperande2 += chiffre ;
      sonEtat = OP2_nchiffres;
      Sujet :: notifie();
      break;
    case OP2_nchiffres:
      sonOperande2 += chiffre ;
      Sujet :: notifie();
      break;
    default:
      cerr << "ControleCalcul :: Chiffre(char chiffre) Error\n";
      break;
  }
}
```

La réception d'un chiffre provoque un changement d'état au sens strict du terme : l'objet évolue car l'un de ses attributs est modifié. Lorsque l'objet `controleCalcul` est dans l'état `OP2_0chiffre` et qu'il reçoit un nouveau chiffre, il ne subit pas de transition au sens de la modélisation visualisée dans le diagramme d'état suivant.

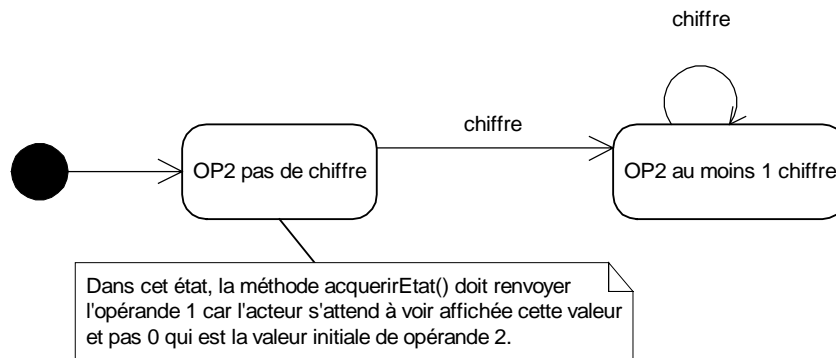


Figure 3-7 : Détails de l'état Opérande 2 de ControleCalcul

```

void ControleCalcul :: Operateur(char operateur)
{
    switch (operateur)
    {
        case '+':
            sonOperateurCalculateur = Icalculateur :: PLUS;
            sonEtat = OP2_0chiffre, sonOperande2 = "";
            Sujet :: notifie();
            break;

        case '=':
            {
                double op1 = atof (sonOperande1.c_str());
                double op2 = atof (sonOperande2.c_str());
                sonResultat = sonCalculateur -> calculer(op1, op2,
                    sonOperateurCalculateur);
                sonEtat = OP1_0chiffre, sonOperande1 = "";
                Sujet :: notifie();
            }
            break;

        case 'A': // Re init
            sonEtat = OP1_0chiffre, sonOperande1 = sonOperande2 = "",
            sonResultat = 0.0;
            Sujet :: notifie();
            break;

        case 'T': // saisie Taux de conversion
            sonCalculateur -> setTaux ( atof (sonOperande1.c_str()));
            sonEtat = OP1_0chiffre, sonOperande1 = "";
            Sujet :: notifie();
            break;

        case 'F': // Conversion en Franc
            sonResultat = sonCalculateur
                -> conversionFranc (atof(sonOperande1.c_str ()));
            sonEtat = OP1_0chiffre, sonOperande1 = "";
  
```

```

    Sujet :: notifie();
    break;

    case 'E': // Conversion en Euro
        sonResultat = sonCalculateur
            -> conversionEuro (atof(sonOperandel.c_str ()));
        sonEtat = OP1_0chiffre, sonOperandel = "";
        Sujet :: notifie();
        break;
    default:
        cerr << "Pb ControleCalcul :: Operateur(char operateur)\n";
        break;
} // fin switch
}

```

Seul l'opérateur «+» est traité ici. Les deux méthodes

```

void ControleCalcul :: Chiffre(char chiffre)
void ControleCalcul :: Operateur(char operateur)

```

modifient l'état de l'objet. Cet état est d'ailleurs explicitement défini dans l'attribut sonEtat. Au contraire, la méthode `acquerirEtat(void)` consulte l'état de l'objet sans le modifier.

```

double ControleCalcul :: acquerirEtat(void)
{
    double retour = 0.0;
    switch (sonEtat)
    {
        case OP1_0chiffre:
            retour = sonResultat;
            break;
        case OP1_nchiffres:
            retour = atof(sonOperandel.c_str ());
            break;
        case OP2_0chiffre:
            retour = atof(sonOperandel.c_str ());
            break;
        case OP2_nchiffres:
            retour = atof(sonOperande2.c_str ());
            break;
        default:
            cerr << "Pb ControleCalcul ::acquerirEtat(void) \n";
    } // switch (sonEtat)

    return retour;
}

```

Cette première version permet de tester les relations entre les objets du sous-système IHM, bien que la méthode `ControleCalcul::Operateur(char chiffre)` n'est pas implémentée.

3.3 Programmation de l'IHM de validation

Le sous-système IHM de validation de l'architecture est basé sur une classe `FcalcCaractère` qui est associée à la classe `ContrôleCalcul` du sous-système Calcul.

Le sous-système IHM basé sur iostream du C++

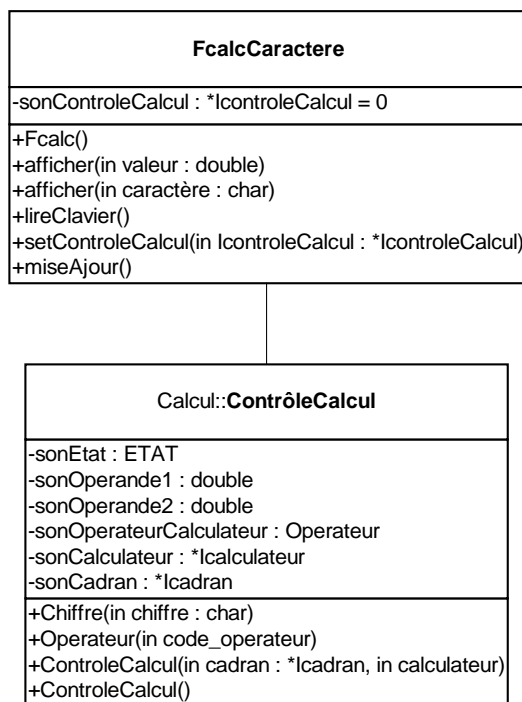


Figure 3–8 : Classes de l'IHM

La classe Fcalc

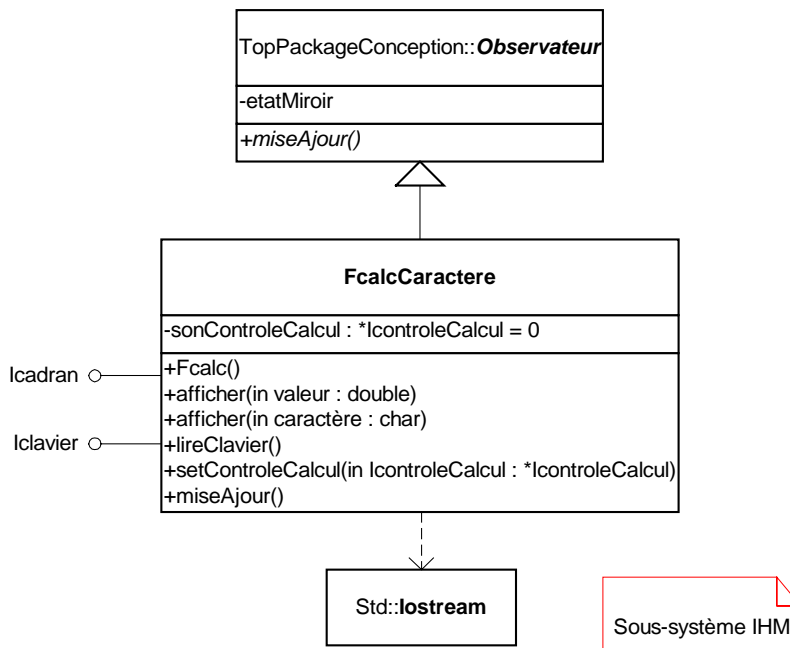


Figure 3-9 : Classe Fcalc[TWAM1]

La définition de la classe FcalcCaractère (fenêtre calculatrice, version caractère) est dans le fichier IHM_FcalcCaractere.h.

```

# ifndef __IHM_FCASC_CARACTERE_H_
#   define __IHM_FCASC_CARACTERE_H_
#   include "Observateur.h"
#   include "CALCUL_IcontroleCalcul.h"
class FcalcCaractere : public Observateur
{
public:
    FcalcCaractere();
    ~FcalcCaractere() {};
    virtual void afficher(char c);
    virtual void afficher(float valeur);
    void lireClavier(void);
    void setControleCalcul(IcontroleCalcul *);
    void miseAJour(void);
private:
    void afficherClavier(void);
private:
    IcontroleCalcul *sonControleCalcul;
};
# endif // __IHM_CONTROLECALCUL_H_

```

Cette classe définit la méthode associée à l'opération `miseAJour()`. La méthode `lireClavier()` correspond à l'aspect clavier de cette classe. Une sur-classe `Iclavier` permettrait de spécifier cette interface. L'attribut `sonControleCalcul` implante l'association au niveau des classes.

Le corps du module est le fichier `IHM_FcalcCaractere.C`. Il contient les définitions des méthodes de la classe.

```
# include <iostream.h>
# include <iomanip.h>
# include "IHM_FcalcCaractere.h"

FcalcCaractere :: FcalcCaractere(void)
    : sonControleCalcul(0)
{
    cout << "FcalcCaractere :: FcalcCaractere(void)\n";
}

void
FcalcCaractere :: setControleCalcul(IcontroleCalcul *le_controleCalcul)
{
    sonControleCalcul = le_controleCalcul;
    if (sonControleCalcul == 0)
        cerr << "FcalcCaractere :: setControleCalcul() erreur\n";
    else
        sonControleCalcul -> attache(this);
}
```

Cette méthode de finalisation de la construction (ou de façon générale de modification de l'objet `controleCalcul` lié) est nécessaire en phase d'initialisation. Elle envoie le message `attache` au sujet observé.

```
void FcalcCaractere :: afficherClavier(void)
{
    cout << "| E F T | \n";
    cout << "| 1 2 3 / | \n";
    cout << "| 4 5 6 * | \n";
    cout << "| 7 8 9 - | \n";
    cout << "| A 0 = + | \n";
    cout << "\t";
    cout.flush();
}
```

Cette méthode simule l'affichage d'un clavier. L'instruction

```
cout.flush();
```

a pour but de provoquer l'affichage de la tabulation bien qu'il n'y ait pas de fin de ligne.

```
void FcalcCaractere :: afficher(char c)
{
    cout << endl;
    cout << "[" << setw(10) << c << "]\n";
}
```

```

void FcalcCaractere :: afficher(float valeur)
{
    cout << endl;
    cout << "[" << setw(10) << valeur << "]\n";
}

void FcalcCaractere :: lireClavier(void)
{
    while (true)
    {
        FcalcCaractere :: afficherClavier();
        char c = 0;
        cin >> c;
        if ( (c >= '0') && (c <= '9') ) // ascii lineaire
            sonControleCalcul -> Chiffre (c);
        else
            sonControleCalcul -> Operateur (c);
    }
}

```

Cette méthode est le cœur du contrôle de l'application. Elle est basée sur une boucle infinie `while(true)`. En fonction du code ascii reçu, l'objet `fcalcCaractere` envoie le message `Chiffre()` ou bien `Operateur()` à l'objet `controleCalcul` lié par le pointeur `sonControleCalcul`.

```

void FcalcCaractere :: miseAJour(void)
{
    double valeur = sonControleCalcul -> acquerirEtat();
    cout << "[ " << setw(10) << valeur << "]\n";
}

```

Cette méthode est déclenchée sur réception du message `miseAJour()` émis par l'objet sujet pointé. L'objet `fcalcCaractere` interroge alors systématiquement le sujet pour obtenir son nouvel état.

3.4 Conception de AppliCaractère

Pour terminer cette première itération de construction de la calculette, il reste à concevoir l'application elle-même, la fonction `main()` du langage C++ se réduisant schématiquement à :

```

void main(void)
{
    Calculette *calculette = new AppliCalculette ;
    Calculette->run() ;
}

```

L'application est une classe active qui sous-traite les entrées à la classe `FcalcCaractere`. Pour cela, elle envoie un message (en début d'exécution) `lireClavier()` qui boucle indéfiniment sur les caractères lus au clavier.

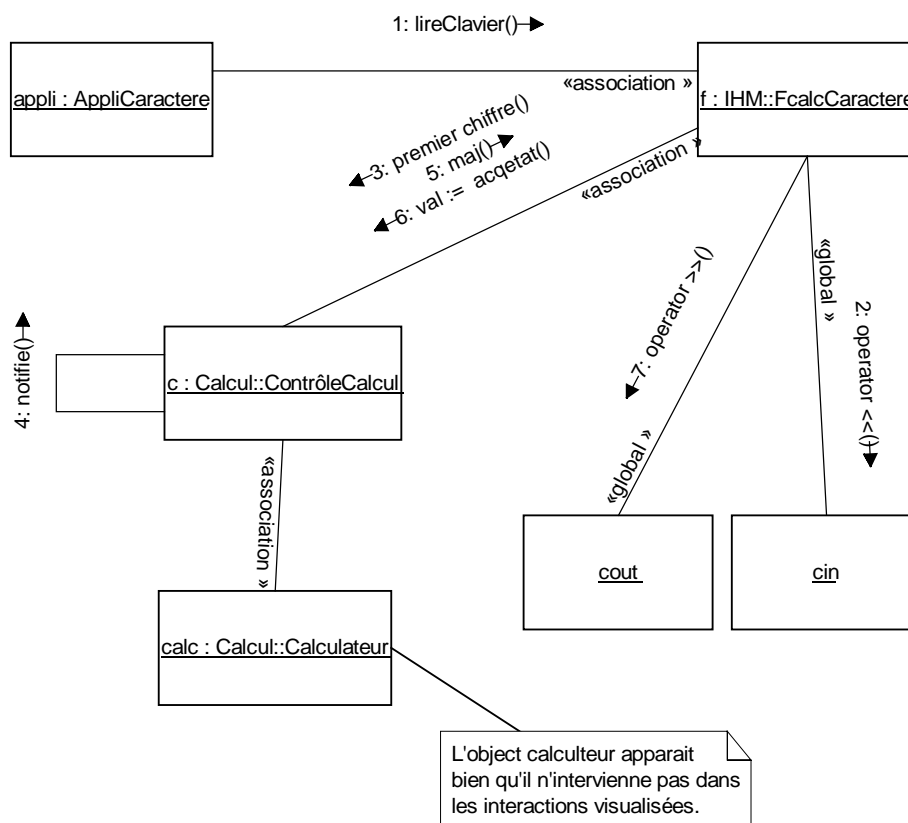


Figure 3–10 : Collaboration entre objets de la solution caractère

Le diagramme de collaboration précédent est essentiel : il présente un début de scénario obtenu dans le modèle des cas d'utilisation et visualise les interactions entre objets de conception, en particulier le type de visibilité. Ici `<<association>>` est la valeur par défaut mais est indiqué pour éviter toute mauvaise interprétation. L'objet `cin` est par contre global.

Ce diagramme permet d'une part de vérifier l'adéquation de la conception, d'autre part les visibilités sont nécessaires à l'implémentation.

Enfin, l'ordre de création des objets reste à déterminer.

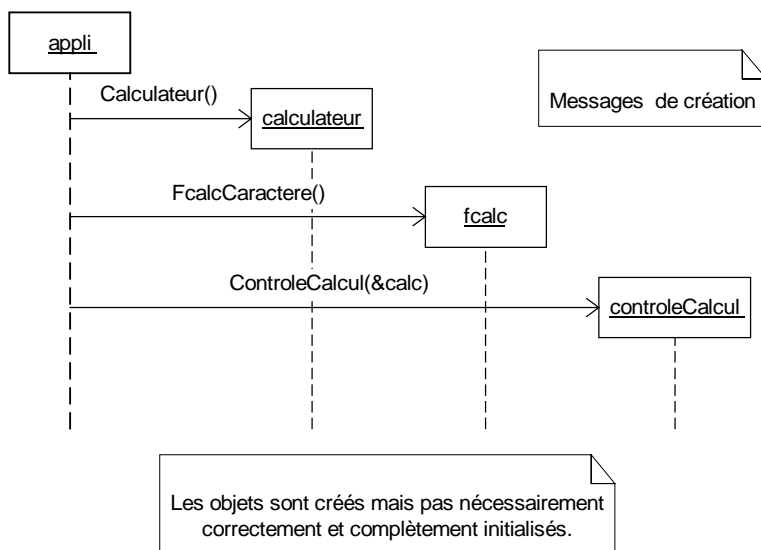


Figure 3-11 : Création des objets

Les objets sont maintenant créés, il reste à finaliser les liens. Pour cela, nous jouons avec les paramètres des constructeurs et des méthodes de modification d'état qui initialisent des pointeurs dans les objets.

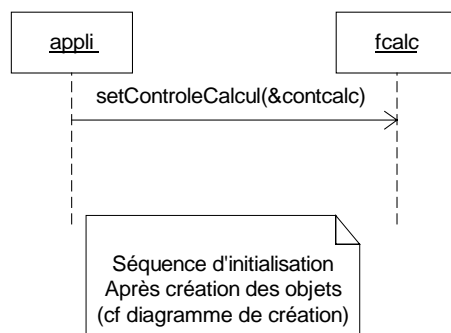


Figure 3-12 : Fin de la séquence d'initialisation

```

# ifndef __APPLI_CARACTERE_H_
#   define __APPLI_CARACTERE_H_

# include "CALCUL_IcontroleCalcul.h"
# include "CALCUL_Icalculateur.h"
# include "IHM_FcalcCaractere.h"

class AppliCaractere
{

```

```

public:
    AppliCaractere();
    ~AppliCaractere();
    void run(void);
private:
    IcontroleCalcul *sonControleCalcul;
    FcalcCaractere *sonFcalc;
    Icalculateur *sonCalculateur;
};

# endif // __IHM_CONTROLECALCUL_H_

```

La classe `AppliCaractere` est relativement luxueuse : elle permet d'obtenir une fonction `main()` dont le seul but est de l'instancier. Ainsi, l'application elle-même devient un objet. La méthode `run()` correspond au lancement à proprement parler de l'exécution. La fonction `main()` est alors :

```

# include "AppliCaractere.h"

void main(void)
{
    AppliCaractere *appli = new AppliCaractere;
    appli->run();
}

```

Les méthodes de la classe `AppliCaractere` ont essentiellement pour but

D'instancier les objets de l'application

De passer le contrôle à l'objet interface utilisateur.

```

# include <iostream.h>
# include "AppliCaractere.h"

// Ici les includes des classes concretes
# include "CALCUL_Calculateur.h"
# include "CALCUL_ControleCalcul.h"
AppliCaractere :: AppliCaractere(void)
{
    sonCalculateur = new Calculateur;
    sonFcalc = new FcalcCaractere;
    sonControleCalcul = new ControleCalcul(sonCalculateur);
    sonFcalc -> setControleCalcul(sonControleCalcul);
}

```

Le constructeur de l'application crée aussi les objets applicatifs. De ce fait, le module doit inclure les définitions des classes concrètes.

```

AppliCaractere :: ~AppliCaractere(void)
{
    delete sonCalculateur;
    delete sonFcalc;
    delete sonControleCalcul;
}

```

```

void AppliCaractere :: run(void)
{
    sonFcalc -> lireClavier();
}

```

3.5 Conclusion

A l'issue de cette itération, la calculette (version caractère `iostream`) doit être opérationnelle. Les classes `Calculateur` et `ControleCalcul` sont validées, le mécanisme sujet / observateur également. En supposant a priori qu'il n'y a strictement aucun problème à la mise en œuvre d'une interface graphique Motif, l'architecte peut désormais considérer que l'architecture est validée.

Le sous-système calcul n'est pas complet. Toutes les opérations de la calculette ne sont prises en compte. Ce n'est pas un problème dans la mesure où l'objectif de cette itération d'élaboration n'était pas d'obtenir une version opérationnelle de quelques cas d'utilisation mais de valider l'architecture.

VALIDATION DE L'ARCHITECTURE	
THIERRY CROS	1
3.1 Conception détaillée Calculateur	3
3.2 le sous-système "Calcul"	7
3.3 Programmation de l'IHM de validation	13
3.4 Conception de AppliCaractère	17
3.5 Conclusion	21