5

Construction calculette graphique

www.thierrycros.net

La première itération lors de la phase d'élaboration, version `iostream`, nous a permis de valider l'architecture, les classes `Calculateur` et `ContrôleCalcul`. Cette nouvelle itération consiste à construire une version graphique, la version « définitive », basée sur la bibliothèque standard Motif.

Phases / Activités	*Jul 99*										
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
Elaboration calculette											
Itération d'élaboration											
Expression de besoins											
Analyse											
Conception/Architecture											
Programmation/Test											
Construction calculette graphique											
Itération de construction											

Figure 5–1 : La phase et l'itération dans le cycle de développement

Points étudiés

Intégration Motif (langage C) et C++

Mise en oeuvre d'un pattern de conception (fabrication).

L'itération démarre par l'activité d'expression de besoins. Le modèle actuel des besoins est synthétisé dans le diagramme de contexte du système.

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

Figure 5–2 : Activités dans l'itération

Les activités d'expression de besoins et d'analyse n'apportent pas (ici) de valeur ajoutée aux modèles actuels issus de l'itération dans l'élaboration. Toutefois, il n'est pas inutile, pour le moins, de se poser les questions inhérentes à ces activités. Par exemple, les acteurs, les cas d'utilisation ont-ils été exhaustivement pris en compte ?

NOTE

L'un des avantages des méthodes est qu'elles permettent de se poser les « bonnes questions au bon moment ». Il faut alors accepter de suivre les démarches proposées. C'est un peu le « deal » avec une méthode. Ne pas suivre les démarches revient à remettre plus ou moins en cause la pertinence de l'outil. De ce fait, les adaptations inévitables des méthodes aux contextes des équipes, des projets, des domaines constituent un exercice difficile : jusqu'à quel point l'adaptation est compatible avec l'esprit de la méthode ?

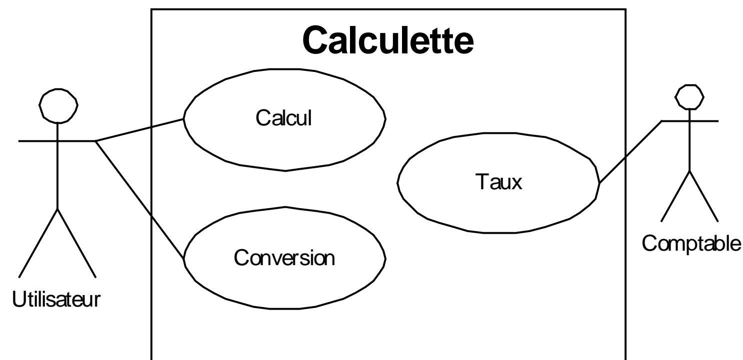


Figure 5–3 : Contexte de la calculette

Une itération de construction passe, plus ou moins rapidement, par une activité d'expression de besoins : la phase d'élaboration a pour finalité (de ce point de vue) d'obtenir la majorité des cas d'utilisation mais approximativement 20% de leur description détaillée. La phase de construction intervient ensuite pour compléter la liste des cas et surtout leurs descriptions.

Dans le cas de la calculette, l'ensemble des cas d'utilisation a été obtenu et même décrit dans la phase d'élaboration. Une évolution pourrait être de considérer désormais les cas d'utilisation de la calculette en tant que « services » rendus par des applications. Dans le cas d'un traitement de texte, l'un des services pourrait être cette calculette. Afin de faciliter l'insertion du modèle actuel dans un modèle plus général, il est possible d'empaqueter les cas d'utilisation dans une catégorie « Calculs ». Cette évolution n'est pas strictement obligatoire ici.

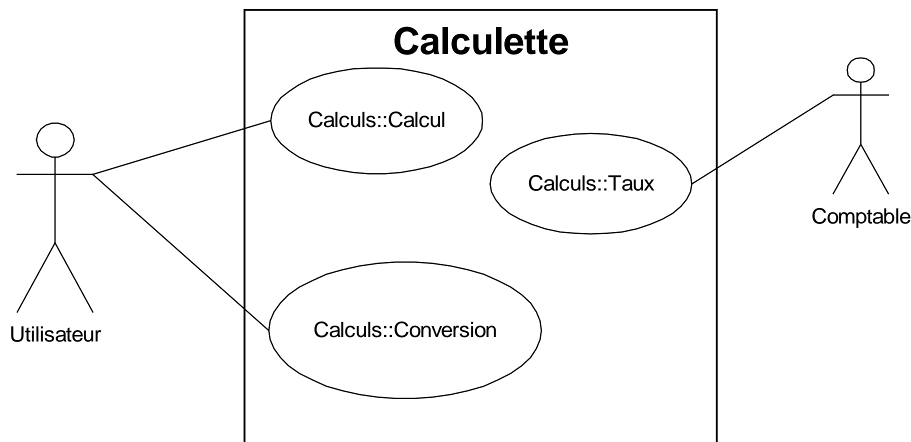


Figure 5–4 : Cas d'utilisation empaquetés dans « Calculs »

En ce qui concerne l'analyse, la situation est inchangée.

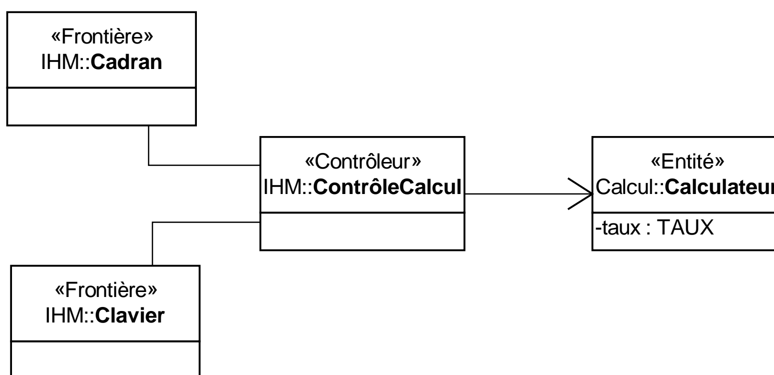


Figure 5–5 : Classes d'analyse de la calculette

5.1 Mécanisme d'intégration Motif - objet

5.1.1 Un mécanisme général

L'adaptation de Motif à la conception objet et à la programmation C++ n'est pas immédiate. En effet, les fonctions `Callback` déclenchées sur réception d'un événement émis par le serveur sont automatiquement invoquées par la Toolkit qui suppose une programmation C. Une méthode d'objet ne peut pas jouer ce rôle car elle reçoit implicitement un paramètre `this`. La solution consiste à fournir une fonction que la Toolkit peut invoquer : fonction classique type C ou bien fonction membre de portée classe et non objet ce qui supprime le problème lié au paramètre `this`.

Une classe `Grille` dont les objets doivent réagir à des événements issus du serveur présente des fonctions membres qui

Jouent le rôle de callback pour certaines,

Sont de véritables méthodes pour d'autres.

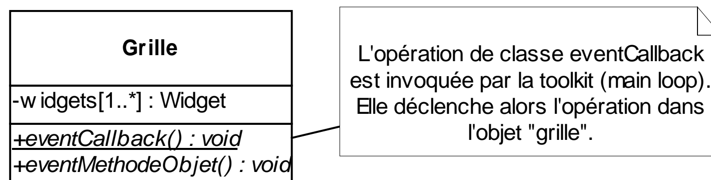


Figure 5–6 : Classe graphique Motif

```
class Grille {
public :
   static void eventCallback() ;
protected :
   void eventMethodeObjet() ;
} ;
```

5.1.2 Adaptation à la calculette

La classe `Fcalc`, qui est une grille de saisie, suit la trame proposée. Les événements à traiter sont les click souris sur les différents boutons : chiffres, opérateurs, etc.

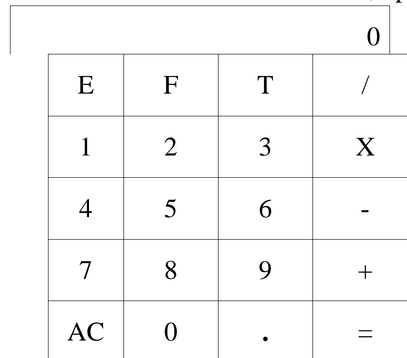


Figure 5–7 : Maquette calculette

La classe `Fcalc` reprend les responsabilités de la version `iostream` : affichage et saisie utilisateur.

L'affichage réalise l'interface `Icadran` et à ce titre est toujours constituée de deux méthodes. Une widget `cadran` participe également à cette responsabilité. C'est un label au sens Motif.

Le clavier est formé de vingt boutons poussoirs de Motif, qui interviennent dans l'état de la classe ; les fonctions membres `clickCallback()` et `click()` ont pour but de gérer les événements sur ces boutons. Ainsi, une responsabilité est traduite en opérations et en attributs.

La gestion de la fenêtre (au sens classique de grille) est confiée à deux opérations : afficheFenêtre() et cacheFenêtre() qui ont pour but respectivement de faire apparaître et disparaître la grille. Cela correspond au « management » au sens Toolkit.

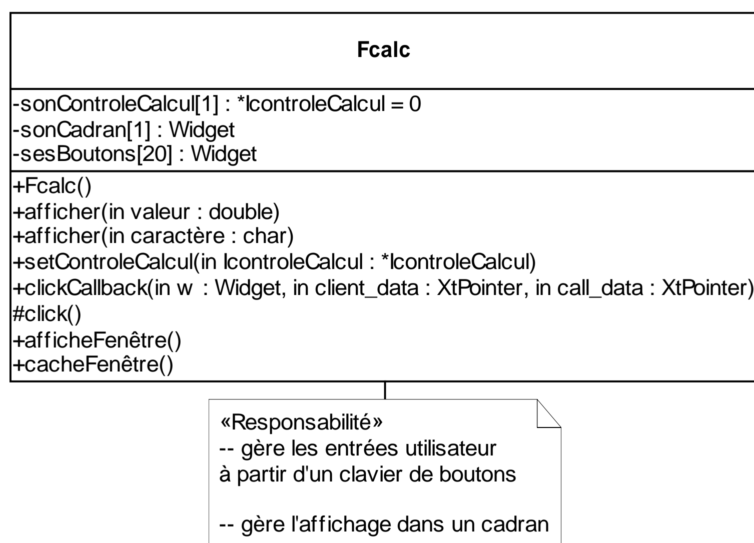


Figure 5–8 : Classe Fcalc version graphique Motif

La classe applicative est maintenant adaptée à Motif.

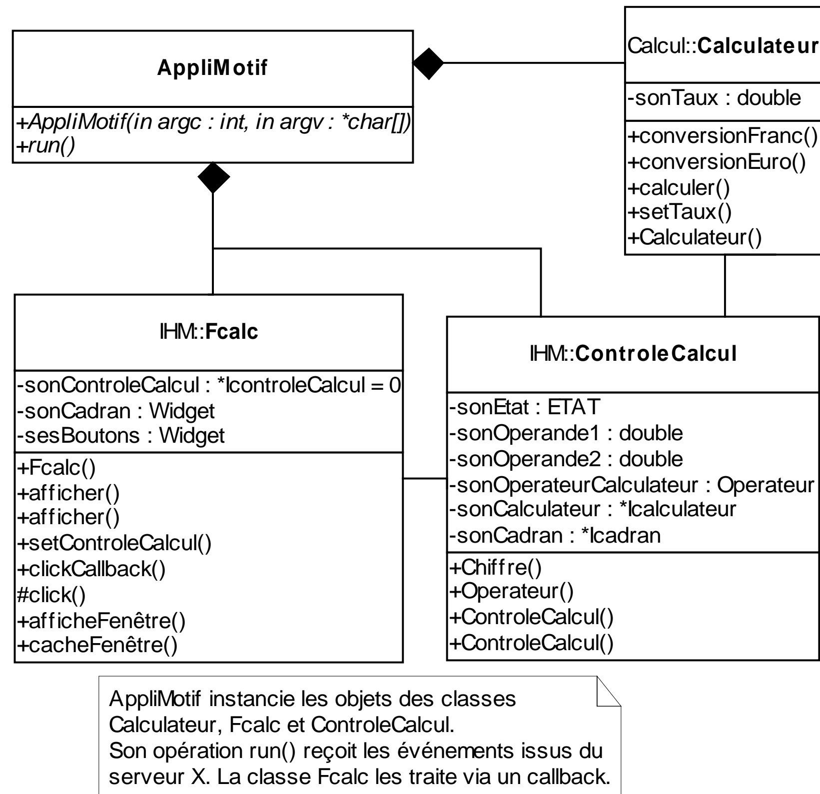


Figure 5–9 : Classes de la version Motif

Le fonctionnement de la calculette Motif fait intervenir constamment tous les objets, y compris `appliMotif` qui est la classe active. L'aspect dynamique est visualisé par un diagramme de séquence.

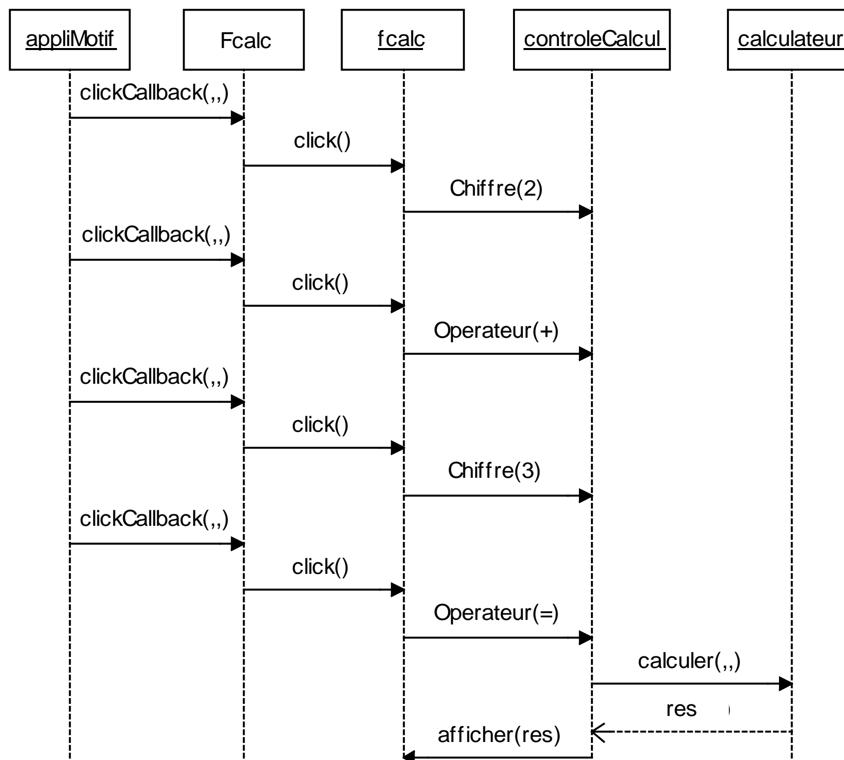


Figure 5–10 : Fonctionnement typique de la calculette Motif

Il reste maintenant à déterminer l'ordre de création des objets ainsi que leurs visibilités respectives.

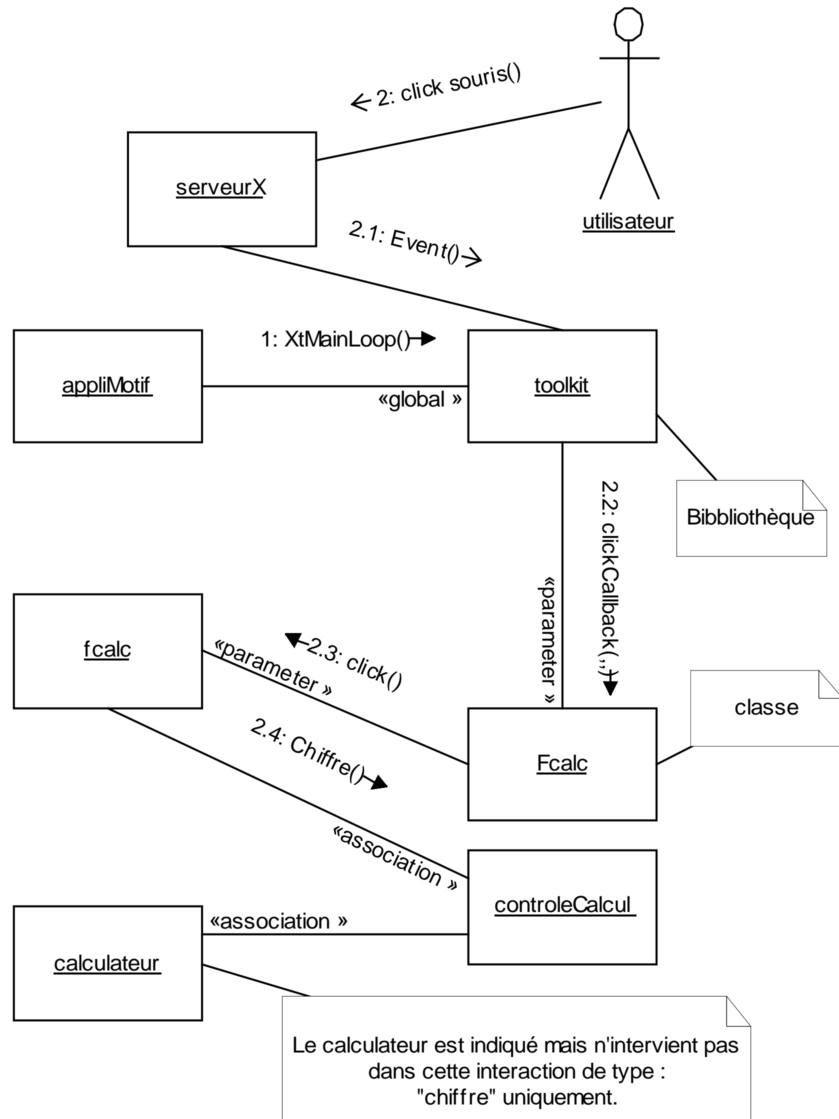


Figure 5–11 : Collaboration entre objet de la calculette Motif

L'initialisation des objets, leurs créations, sont visualisées par un diagramme de séquence.

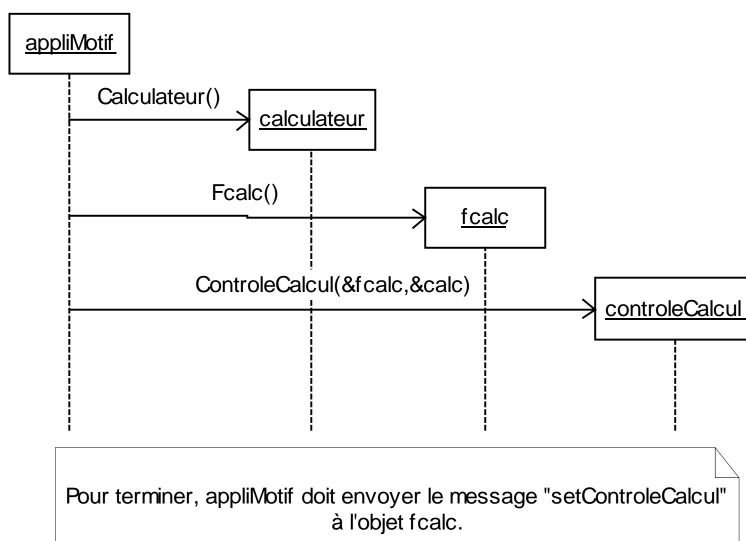


Figure 5–12 : Création des objets de la calculette Motif

5.1.3 Avant de programmer

Les algorithmes des méthodes ne doivent plus poser de problème en programmation. Pour cela, examinons chaque méthode de chaque classe.

AppliMotif

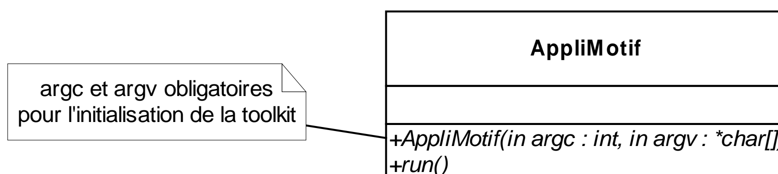


Figure 5–13 : Classe AppliMotif

Cette classe représente le client X. A ce titre, elle prend en charge la relation avec le serveur graphique.

Les classes `AppliCaractere` et `FcalcCaractere` d'une part, `AppliMotif` et FcalcMotif d'autre part sont liées car elles partagent le contrôle de l'application en fonction du type d'entrée-sortie. Nous reviendrons sur ce point à propos du pattern fabrique abstraite.

Le constructeur créé la communication (la connexion) avec le serveur.
Il crée aussi les objets de l'application.

Fcalc {version = Motif}
+Fcalc(){sequential} +afficher(in valeur : double){sequential} +afficher(in caractère : char){sequential} +setControleCalcul(in IcontroleCalcul : *IcontroleCalcul){sequential} +clickCallback(in w : Widget, in client_data : XtPointer, in call_data : XtPointer){sequential} #click(){sequential} +afficheFenêtre(){sequential} +cacheFenêtre(){sequential}

Figure 5–14 : La classe Fcalc

Fcalc

Les valeurs marquées indiquent la version de la classe (ici Motif) et le fait que les opérations sont séquentielles. Le constructeur ne pose pas de problème particulier. Les opérations `afficher()` modifient l'état du cadran, c'est de la programmation Motif. Les opérations de gestion du click souris sur un bouton du clavier de la calculette constituent la difficulté majeure de cette mise en œuvre. En effet, l'opération de classe doit être un callback : elle possède donc les trois paramètres cités.

Le constructeur a pour rôle de créer la fenêtre et donc ses composants : le cadran, les boutons, etc. Cette classe est confiée à un programmeur Motif. Les détails de mise en œuvre seront réglés en programmation, il s'agit de développement graphique sous Unix, la conception a été réalisée dans ce paragraphe. Une autre solution consiste à générer le code à partir d'un générateur d'interface. La méthode `run()` invoque simplement la fonction `XtMainLoop()` de la bibliothèque Toolkit.

5.1.4 Programmation et test

La classe `AppliMotif` est définie dans le fichier AppliMotif.h.

```
# ifndef __APPLI_MOTIF_H_
#  define __APPLI_MOTIF_H_

# include "IHM_Icadran.h"
# include "IHM_IcontroleCalcul.h"
# include "Calcul_Icalculateur.h"
# include "IHM_FcalcMotif.h"

class AppliMotif
{
   public:
      AppliMotif(int argc, char * argv[]);
      ~AppliMotif();
      void run(void);
   private:
      IcontroleCalcul *sonControleCalcul;
      FcalcMotif  *sonFcalc;
      Icalculateur *sonCalculateur;
      Widget toplevel;
      XtAppContext app;
};
```

```
# endif
```

Les méthodes sont codées dans le fichier AppliMotif.C.

```
# include  <Xm/XmAll.h>
# include "AppliMotif.h"

// Ici les includes des classes concretes
# include "Calcul_Calculateur.h"
# include "IHM_ControleCalcul.h"

AppliMotif :: AppliMotif(int argc, char * argv[])
{
   toplevel  =XtAppInitialize(&app, "Twam", NULL, 0,
             &argc, argv, NULL, NULL, 0);
   // A ce niveau, la connexion serveur est correcte
   // sinon la toolkit est sortie en erreur "Can't open display"

   sonCalculateur = new Calculateur;
   sonFcalc = new FcalcMotif(toplevel);
   sonControleCalcul = new ControleCalcul(sonCalculateur, sonFcalc);
   sonFcalc -> setControleCalcul(sonControleCalcul);

   sonFcalc -> afficheFenetre();
   sonFcalc->afficher( (float) 0);
   XtRealizeWidget(toplevel);
}

AppliMotif :: ~AppliMotif(void)
{
   delete sonCalculateur;
   delete sonFcalc;
   delete sonControleCalcul;
   XtDestroyWidget(toplevel);
}

void AppliMotif :: run(void)
{
   XtAppMainLoop(app);
}
```

La classe `FcalcMotif` est définie dans le fichier IHM_FcalcMotif .h

```
/*
   Projet Calculette
   Fichier : IHM_FcalcMotif.h
   Specification de la classe IHM_FcalcMotif

 */

# ifndef __IHM_FCALC_MOTIF_H_
#  define __IHM_FCALC_MOTIF_H_

# include <Xm/XmAll.h>
```

```
# include "IHM_Icadran.h"
# include "IHM_IcontroleCalcul.h"

class FcalcMotif : public Icadran
{
   public:
      FcalcMotif();
      FcalcMotif(Widget parent);
      ~FcalcMotif();
      virtual void afficher(char c);
      virtual void afficher(float valeur);
      void setControleCalcul(IcontroleCalcul *);
      void afficheFenetre(void);
      void cacheFenetre(void);
   private:
      static void clickCallback(Widget, XtPointer, XtPointer);
   protected:
      void click(Widget, XtPointer);
   private:
      IcontroleCalcul *sonControleCalcul;
      enum Constantes { BOUTONS = 20};
      Widget contenant; // la grille elle-même
      Widget cadran;
      Widget boutons[BOUTONS];
};
# endif
```

Fichier corps de la classe : IHM_FcalcMotif.C.

```
/*
   Projet Calculette
   Fichier :  IHM_FcalcMotif.C
   Corps de la classe FcalcMotif
 */

# include <iostream.h>
# include <iomanip.h>
# include "IHM_FcalcMotif.h"

FcalcMotif :: FcalcMotif(void)
   : sonControleCalcul(0)
{
   cerr << "FcalcMotif :: FcalcMotif(void)\n";
}

FcalcMotif :: ~FcalcMotif(void)
{
   cerr << "FcalcMotif :: ~FcalcMotif(void)\n";
   XtDestroyWidget(contenant);
}

FcalcMotif :: FcalcMotif(Widget parent)
   : sonControleCalcul(0)
{
   static char * noms[BOUTONS] =    {
```

```
                 "E",  "F", "T", "/",
                                    "1",  "2", "3", "x",
                                    "4",  "5", "6", "-",
                                    "7",  "8", "9", "+",
                                    "AC", "0", ".", "="
                };
   cerr << "FcalcMotif :: FcalcMotif(Widget parent)\n";
   contenant = XmCreateRowColumn(parent, "contenant", NULL,0);
   // pour les nombres cadrés à droite
   XtVaSetValues(contenant,
      XmNentryAlignment, XmALIGNMENT_END,
      NULL);
   cadran  = XmCreateLabel(contenant, "cadran", NULL, 0);
   Widget separateur = XmCreateSeparator(contenant, "sep", NULL, 0);
   Widget clavier = XmCreateRowColumn(contenant, "clavier", NULL, 0);
   for (int i=0; i<BOUTONS; ++i)
   {
      boutons[i] = XmCreatePushButton(clavier, noms[i], NULL,0);
      XtAddCallback(boutons[i], XmNactivateCallback,
            &FcalcMotif :: clickCallback,
            (XtPointer) this);
   }
   // réglage du clavier
   XtVaSetValues(clavier,
      XmNorientation, XmHORIZONTAL,
      XmNpacking, XmPACK_COLUMN,
      XmNnumColumns, 5,
      NULL);
   XtManageChildren(boutons, XtNumber(boutons));
   XtManageChild(clavier);
   XtManageChild(separateur);
   XtManageChild(cadran);
}

void FcalcMotif :: setControleCalcul(IcontroleCalcul *le_controleCalcul)
{
   sonControleCalcul = le_controleCalcul;
}

void FcalcMotif :: afficher(char c)
{
   char chaine[32];
   sprintf(chaine, "%c", c);
   XmString xms = XmStringCreateLocalized(chaine);
   XtVaSetValues(cadran,
         XmNlabelString, xms,
         NULL);
   XmStringFree(xms);
}

void FcalcMotif :: afficher(float valeur)
{
   char chaine[32];
   sprintf(chaine, "%15.2f", valeur); // à améliorer…
   XmString xms = XmStringCreateLocalized(chaine);
```

```
   XtVaSetValues(cadran,
         XmNlabelString, xms,
         NULL);
   XmStringFree(xms);
}

void FcalcMotif :: afficheFenetre(void)
{
   XtManageChild(contenant);
}

void FcalcMotif :: cacheFenetre(void)
{
   XtUnmanageChild(contenant);
   // la grille devient invisible et donc tout le contenu
}

// Methode de classe
void FcalcMotif :: clickCallback(Widget w, XtPointer clientData,
               XtPointer callData)
{
   FcalcMotif * objet = (FcalcMotif *) clientData;
   objet -> click(w, callData);
}

void FcalcMotif :: traiterTouche(char caractere)
{
   switch(caractere)
   {
     case '0':
     case '1':
     case '2':
     case '3':
     case '4':
     case '5':
     case '6':
     case '7':
     case '8':
     case '9':
     case '.':
      sonControleCalcul -> Chiffre(caractere);
      break;
     default:
      sonControleCalcul -> Operateur(caractere);
      break;
   }
}

void FcalcMotif :: click(Widget w, XtPointer callData)
{
   String nom = XtName(w);
   traiterTouche(nom[0]);
}
```

Un objet applicatif est créé dans la fonction `main()`.

```
// # include "AppliCaractere.h"
# include "AppliMotif.h"

void main(int argc, char *argv[])
{
      AppliMotif *appli = new AppliMotif(argc, argv);
      appli->run();
}
```

5.2 Intégrer les versions iostream et Motif

5.2.1 Deux versions cohabitent

Les deux versions de la calculette (caractère et graphique) doivent désormais cohabiter. Une première solution consiste à compiler puis lier les fichiers. Les directives de compilation conditionnelle offrent la possibilité de gérer une seule version de fichier source, quelle que soit la cible.

```
# ifdef GRAPHIQUE_UNIX
#     include "AppliMotif.h"
# else
#     include "AppliCaractere.h"
# endif

void main(int argc, char *argv[])
{
# ifdef GRAPHIQUE_UNIX
      AppliMotif *appli = new AppliMotif(argc, argv);
      appli->run();
# else
      AppliCaractere *appli = new AppliCaractere;
      appli->run();
# endif
}
```

Dans ce cas précis, le fichier `main` est seul concerné. Toutefois les directives `#ifdef` ont tendance à proliférer dans les projets, les fichiers source deviennent alors moins lisibles. Cela est du au fait que les objets, de façon générale, sont créés dans de nombreux modules. Pire encore, l'ajout d'une cible provoque de nombreuses et laborieuses évolutions.

5.2.2 Mise en œuvre d'un pattern de conception : fabrication

Ce pattern permet d'encapsuler la création des objets. Ainsi, le problème est isolé. C'est une technique classique : lorsqu'un aspect du système pose problème, un objet sacrifié est créé pour protéger les autres.

NOTE

La notion de sacrifice est particulièrement apparente dans le cas du médiateur. Ce pattern (qui est la contrepartie conceptuelle des contrôleurs en analyse) crée un objet, le médiateur, qui prend en charge l'ensembles des liens. De ce fait, les autres objets sont moins couplés et donc plus réutilisables.

Pratiquement, seule la classe qui représente l'application doit être créée par la fabrication. En effet, l'objet de la classe Fcalc est directement créé par l'application. De plus son constructeur reçoit des paramètres qui dépendent fortement du type.

Ce pattern est basé sur une hiérarchie de classes qui créent les objets.

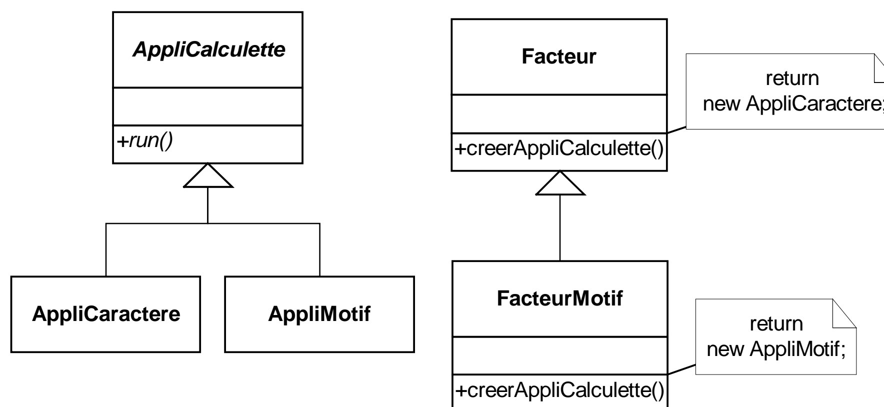
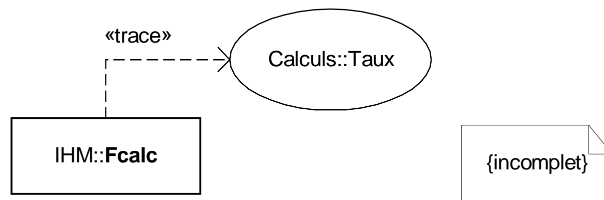


Figure 5–15 : Classes de mise en œuvre du pattern fabrication

5.3 Vérification par les cas d'utilisation

Un retour vers les cas d'utilisation permet d'assurer la conformité de la conception. Des relations stéréotypées « trace » montrent les relations entre éléments du modèle d'expression de besoins et modèle de conception.

De même, des relations « trace » pourraient être visualisées entre analyse et conception.



Une dépendance <<trace>> parmi beaucoup d'autres.
Visualiser la traçabilité entre éléments de modélisation a un double intérêt.
- cette relation démontre la pertinence de l'élément source
- elle facilite aussi la mesure de l'impact d'une évolution de l'élément cible.

Figure 5–16 : relation « trace » entre éléments de modélisation

CONSTRUCTION CALCULETTE GRAPHIQUE......1

5.1 Mécanisme d’intégration Motif - objet......4

5.1.1 Un mécanisme général......4

5.1.2 Adaptation à la calculette......4

5.1.3 Avant de programmer......9

5.1.4 Programmation et test......10

5.2 Intégrer les versions iostream et Motif......15

5.2.1 Deux versions cohabitent......15

5.2.2 Mise en œuvre d’un pattern de conception : fabrication......15

5.3 Vérification par les cas d’utilisation......16