AspectC++ — extension de la Programmation Orientée Aspect pour C++

Olaf Spinczyk
Daniel Lohmann
Matthias Urban

Sur le CD:

La version d'évaluation de l'outil compagnon d'AspectC++ pour VisualStudio.NET, de l'outil de développement libre AspectC++ (Aspect C++ Development Tools, ou ACDT) pour Eclipse ainsi que les listings des exemples sont disponibles sur le CD joint au magazine.

De plus en plus de développeurs de logiciels sont confrontés à la Programmation Orientée Aspect (Aspect-Oriented Programming, ou AOP en anglais). En proposant des moyens de modulariser l'implémentation de préoccupations transverses, cette programmation permet une meilleure réutilisation, moins de couplage entre les modules, ainsi qu'une meilleure séparation entre les préoccupations d'une manière générale. À l'heure actuelle, des supports solides pour la Programmation Orientée Aspect sont disponibles, par exemple, celui de JBoss (JBoss AOP), de BEA (AspectWerkz), et d'IBM (AspectJ et AJDT pour Eclipse). Toutefois, l'ensemble de ces produits est basé sur le langage Java. En ce qui concerne les développeurs de C et C++, aucun des acteurs majeurs du marché n'offre toujours pas de support pour la Programmation Orientée Aspect.

Le présent article a pour objectif de présenter AspectC++, extension du langage orientée aspect pour C++. Un compilateur pour AspectC++, chargé de transformer le code source en code C++ classique, est développé sous forme de projet libre. Le projet AspectC++ a débuté avec un prototype issu de recherches en 2001 qui a gagné en maturité au cours des années. Aujourd'hui, le langage AspectC++ ainsi que le parseur ont été appliqués avec succès sur un certain nombre de projets à portée internationnale dans le domaine informatique et académique et une intégration IDE (Integrated Development Environment, ou Environnement de développement Intégré, en français) sous Eclipse et Microsoft VisualStudio.NET représente la première des nombreuses avancées à venir d'un projet encore naissant.

La présentation d'AspectC++ débutera par un exemple que l'on peut considérer de *Hello World* en matière de Programmation Orientée Aspect. Cet exemple illustrera les éléments de base du langage comme les aspects, les coupes transverses, ainsi que les méthodes d'aspect, que certains lecteurs connaissent sans doute déjà avec le langage AspectJ. Ensuite nous exposerons brièvement ces éléments semblables au langage AspectJ en observant une version du modèle observateur (observer pattern) sous AspectC++, relativement bien connue ainsi que les *méthodes d'aspect dites génériques*. Cette option unique sous AspectC++ permet d'allier la puissance des aspects à la programmation générique et générative de C++.

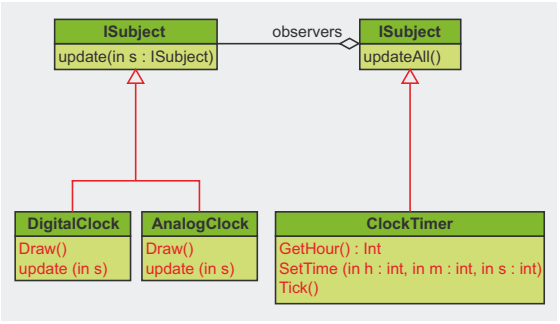

Figure 1. *Préoccupation transverse dans le modèle observateur*

Tracing — ou l'équivalent de l'application Hello World en Programmation Orientée Aspect

En guise d'introduction à la Programmation Orientée Aspect avec AspectC++, nous analyserons un exemple très simple d'aspect. L'aspect intitulé *Tracing* (*dépistage*) a pour but de modulariser l'implémentation des opérations de données de sortie, utilisées afin de pister le fux de contrôles du programme. À chaque démarrage de l'exécution d'une fonction, l'aspect exposé ci-dessous édite son nom :

```
#include <cstdio>
// Exemple de dépistage du flux de contrôles
aspect Tracing {
  // édite le nom de la fonction avant
  // le début de son exécution
  advice execution ("% ...::%(...)") : before () {
    std::printf ("in %s\n", JoinPoint::signature ());
  }
};
```

Même sans avoir une connaissance approfondie de tous les éléments syntaxiques exposés ci-dessus, la Programmation Orientée Aspect présente d'énor-

Le docteur Olaf Spinczyk est le fondateur ainsi que le principal développeur du projet AspectC++. Daniel Lohmann utilise quant à lui AspectC++ pour ses recherches sur le développement d'un système d'exploitation et travaille sur la conception et l'évolution des concepts du langage AspectC++. Tous deux travaillent à l'Université Friedrich-Alexander d'Erlangen-Nuremberg. Enfin, Matthias Urban est le principal développeur de l'analyseur syntaxique sous AspectC++. Il travaille chez la société pure-systems GmbH, où il est responsable du support pour AspectC++ ainsi que de l'outil compagnon d'AspectC++ pour VisualStudio.NET.
Pour contacter les auteurs :
os@aspectc.org, dl@aspectc.org, mu@aspectc.org

mes avantages sensés être évidents au vue de l'exemple ci-dessus. Sans le recours à ce simple aspect, qui ne prend que quelques lignes, le développeur serait obligé d'étendre l'ensemble des fonctions du programme au moyen d'une déclaration supplémentaire `printf` afin d'obtenir le même résultat. Dans un projet important, un guide de style devrait indiquer cette technique sous forme d'exigences et l'ensemble des programmeurs devrait le lire et observer cette règle générale. Ainsi, la solution proposée par la Programmation Orientée Aspect permet de gagner un temps précieux, des efforts d'organisation, et garantit l'absence d'oubli de fonction. De la même façon, le code, affecté par l'aspect, est entièrement découplé du code de dépistage, soit la déclaration `printf`. Il n'est même pas nécessaire d'inclure `<cstdio>`, dans la mesure où tout ce procédé est effectué à part par l'aspect.

Les aspects, les méthodes d'aspect et les coupes transverses

L'exemple de l'aspect Tracing expose la majorité des éléments du langage chargés d'accomplir tous les avantages cités plus haut. Nous commencerons par l'*aspect*, qui a été conçu comme un module pour l'implémentation d'une préoccupation transverse. D'un point de vue syntaxique, un `aspect` sous AspectC++ est très proche d'une classe sous C++. Toutefois, en plus des fonctions membres et des éléments de données, un aspect peut définir une *méthode d'aspect*. Après le mot clé `advice`, une *expression de coupe transverse* définit l'emplacement où la méthode d'aspect est censée affecter le programme (autrement dit les *points de jonction*), alors que la partie suivant la colonne définit la *façon* dont le programme est censé être affecté sur ces points. Il s'agit de la règle générale pour l'ensemble des différents types de méthodes d'aspect sous AspectC++.

Les expressions de coupes transverses

L'expression de la coupe transverse donnée dans l'exemple (emplacement) est `execution("% ...::%(...)")`. Autrement dit, cette méthode d'aspect devrait affecter l'exécution de l'ensemble des fonctions qui ne satisfont pas l'expression `"% ...::%(...)"`. Dans les « expressions de correspondances » le signe `%` et les caractères `...` sont utilisés en tant que caractères de remplacement. Le signe pourcentage (`%`) correspond à n'importe quel type (par exemple `"% *"` correspond à tous type pointeurs) ainsi qu'à n'importe quelle séquence de caractères dans les identifiants (par exemple `"xdr_%"` correspond à l'ensemble des classes dotées d'un nom commençant par `xdr_`). Les points de suspension (`...`) équivalent à n'importe quelle séquence de types ou à des espaces de nommage (par exemple `"int foo(...)"` équivaut à n'importe quelle fonction générale chargée de retourner un entier et nommée foo). Enfin, l'expression de correspondance `"% ...::%(...)"` équivaut à n'importe quelle classe ou espace de nommage.

Les expressions de correspondances représentent des ensembles d'entités de programme nommées comme les fonctions ou les classes. Ainsi, les expressions de correspondances sont toujours des expressions de coupes transverses primitives, chargées de décrire un ensemble de points de jonction dans la structure statique du programme (dénommés « points de jonction statiques »). Toutefois, dans l'exemple exposé ci-dessus, nous souhaitons une méthode d'aspect pour décrire un évènement dans le flux dynamique de contrôles du programme, à savoir l'exécution des fonctions. La *fonction de coupe transverse* `execution()` est par conséquent utilisée. Celle-ci a pour objectif de produire l'ensemble des points de jonction d'exécution des fonctions données en argument.

Méthode d'aspect pour les points de jonction dynamiques

En ce qui concerne les points de jonction dits dynamiques, AspectC++ supporte trois types de méthodes d'aspect dans le code intitulées `before()`, `after()`, et `around()`. Ces méthodes implémentent toutes une partie supplémentaire au comportement du programme. Dans l'aspect Trace, évoqué plus haut, ce comportement est implémenté au moyen de la déclaration `printf()`, à la suite de `before()`. D'un point de vue syntaxique, cette technique est identique au corps d'une fonction et, de ce fait, il est d'ores et déjà possible de considérer *advice body (ou corps de la méthode d'aspect)* comme une fonction membre anonyme de l'aspect. À la place de `before()`, nous aurions pu très bien utiliser la méthode d'aspect `after()` (ou les deux) dans l'exemple. Auquel cas, le corps de la méthode d'aspect s'exécuterait une fois l'exécution d'une fonction achevée. Un corps de méthode d'aspect `around()` est exécuté à la place du flux de contrôles, censé suivre normalement le point de jonction dynamique.

Listing 1. *Modèle de l'observateur d'un aspect abstrait*

```
aspect ObserverPattern {
   // Structure des données pour gérer les sujets
   //et les observateurs
   ...
public:
   // Interfaces pour chaque rôle
   struct ISubject {};
   struct IObserver {
      virtual void update(ISubject *) = 0;
   };
   // À définir par l'aspect dérivé concret la fonction
   // subjectChange() apparie l'ensemble
   // des méthodes non-const
   pointcut virtual observers() = 0;
   pointcut virtual subjects() = 0;
   pointcut virtual subjectChange() =
      execution("% ...::%(...)" && !"% ...::%(...) const")
         && within(subjects());
   // Ajoute une nouvelle classe de base
   // à chaque classe sujet/observateur
   // et insère le code pour informer les observateurs
   advice observers() : baseclass(IObserver);
   advice subjects() : baseclass(ISubject);

   advice subjectChange() : after() {
      ISubject* subject = tjp->that();
    updateObservers(subject);
   }
   // Opération  pour ajouter,
   // faites les remarques observées
   void updateObservers(ISubject* sub) { ... }
   void addObserver(ISubject* sub, IObserver* ob) { ... }
   void remObserver(ISubject* sub, IObserver* ob) { ... }
};
```

Expressions combinées de coupes transverses

Les expressions relatives aux coupes transverses peuvent être combinées au moyen des opérateurs set `&&` (intersection), `||` (union), et `!` (inversion). Par exemple, l'expression `"% foo(int, ...)" || "int bar(...)"` équivaut à n'importe quelle fonction générale intitulée `foo` prenant un `int` (entier) en premier paramètre et n'importe quelle fonction générale appelée `bar` chargée de retourner un `int`. Combinées à des fonctions de coupes transverses, nous obtenons ainsi des expressions plutôt puissantes afin de décrire l'emplacement où la méthode d'aspect est censée affecter le programme. Par exemple, nous pourrions changer l'expression de la coupe transverse par l'aspect Tracing comme suit :

```
advice call ("% ...::%(...)")
       && within ("Client") : before () {
  std::printf ("calling %s\n", JoinPoint::signature ());
}
```

La fonction `call()` produit l'ensemble des points de jonction des appels pour des fonctions données. Par opposition aux points de jonction d'exécution, les points de jonction d'appels prennent effet du côté de la routine appelante, c'est-à-dire avant, après ou en même temps que l'appel de la fonction concernée. La fonction de la coupe transverse `within()` se contente de retourner l'ensemble des points de jonction dans les classes ou fonctions indiquées. En introduisant la méthode d'appel à l'intersection de `call ("% ...::%(...)")` (soit n'importe quel appel de fonction) et de `within ("Client")` (n'importe quel point de jonction dans la classe Client), l'aspect ne pistera dés lors que les appels de fonction lancés à partir d'une méthode de la classe `Client`.

L'API point de jonction

Dans le corps de la méthode d'aspect, l'expression `JoinPoint::signature()` attend toujours une explication. Comme nous le savons à partir de l'exemple, cette expression produit le nom de la fonction que nous éditons avant que ne démarre son exécution. La fonction membre statique `signature()` est définie par l'*API point de jonction*. Il s'agit d'une API défini par AspectC++ dont l'objectif est de permettre au code aspect de retrouver les informations contextuelles à partir ou sur un point de jonction pour lequel elles fonctionnent. Nous verrons plus bas que de telles informations contextuelles sont indispensables pour les aspects réels.

Listing 2. *Implémentation d'un observateur concret*

```
#include "ObserverPattern.ah"
#include "ClockTimer.h"
aspect ClockObserver : public ObserverPattern {
   // définit les coupes transverses
   pointcut subjects()   = "ClockTimer";
   pointcut observers()  = "DigitalClock"||"AnalogClock";
public:
   advice observers() :
         void update( ObserverPattern::ISubject* sub ) {
      Draw();
   }
};
```

Notions apprises

Grâce à l'exemple de l'aspect `Tracing`, bien qu'implémenté au moyen de peu de lignes de code, nous avons pu présenter un grand nombre de concepts d'AspectC++. En voici un résumé :

- *Préoccupation transverse* : préoccupation liée à une implémentation, qui affecte de nombreuses parties différentes d'un même programme,
- *Aspect* : permet de fournir une implémentation modulaire d'une préoccupation transverse en définissant des méthodes d'aspect,
- *Point de jonction* : soit un évènement présent dans le flux de contrôles (on parle de points de jonction dynamiques), soit un élément présent dans une structure statique du programme (on parle de points de jonction statiques) sur lesquels une méthode d'aspect affecte le programme,
- *Coupes transverse* : ensemble de points de jonction,
- *Expressions de correspondances* : modèle auquel correspondent les signatures d'entités nommées du programme, par exemple des éléments de structure statique du programme. Les expressions de correspondances sont par conséquent des expressions primitives de coupes transverses, chargées de produire des points de jonction statiques,
- *Expression de coupes transverses* : est utilisée dans le but de définir une coupe transverse. On compose les expressions de coupes transverses en appariant les expressions et les fonctions de coupes transverses. Elles définissent l'emplacement où la méthode d'aspect est censée affecter le programme,
- *Méthode d'aspect* : définit la façon dont un aspect affecte le programme sur une coupe transverse donnée. Dans le cas des méthodes d'aspect pour les points de jonction dynamiques intitulées `before()`, `after()`, ou `around()`, des méthodes d'aspect peuvent être utilisées afin d'implémenter des comportements supplémentaires,
- *API de point de jonction* : peut être utilisé dans le code des méthodes d'aspect afin de restituer les informations contextuelles à partir du point de jonction concerné au moyen du pointeur intégré `JoinPoint *tjp`.

Quoi d'autre ?

L'aspect *Tracing* est un *aspect dit de développement* classique. Par opposition aux *aspects dits de production*, ces aspects ne sont utilisés qu'au cours du développement d'un programme, c'est-à-dire dans le but de déboguer, d'obtenir une assurance qualité et une optimisation du programme. Les aspects dits de production font partie du produit logiciel final, délivré aux utilisateurs. Nous recommandons donc de commencer la Programmation Orientée Aspect avec les aspects de développement afin d'acquérir en premier lieu une certaine expérience. Toutefois, nous ne nous limiterons pas, bien sûr, à la présentation dans le présent article d'un simple programme de type *Hello World* pour la Programmation Orientée Aspect ! La prochaine étape va consister à présenter un aspect dit de production, avec pour objectif d'exposer certaines caractéristiques plus sophistiquées d'AspectC++ et plus particulièrement celles relatives au préoccupations transverses dans la structure statique du programme.

Modèle observateur sous AspectC++

Aujourd'hui, il est d'usage d'utiliser des modèles de conception à partir du « Gang of Four » (Gamma, Helm, Johnson, Vlissi-

des) dans le but de développer un logiciel orienté objet. Un de ces modèles parmi les plus connus est l'*Observateur (Observer)*, illustré ci-après par le diagramme des classes dans la Figure 1. Ce modèle peut être appliqué si un objet est chargé de gérer un état (Le *Sujet* – un objet `ClockTimer`) et si un nombre arbitraire d'autres objets (les *Observateurs* – instances `DigitalClock` et `AnalogClock`) est censé être informé de la modification de cet état. Comme l'illustre le diagramme des classes, la relation sujet/observateur entre nos trois classes d'application peut être établie en dérivant l'objet `ClockTimer` d'un objet réutilisable `ISubject`, chargé de gérer la liste des objets observateurs, et en dérivant les observateurs à partir de la classe abstraite `IObserver`. Par ailleurs, l'ensemble des fonctions altérant l'état (`SetTime()` et `Tick()`) doit être étendu par un appel vers la fonction `updateAll()` afin de spécifier à l'ensemble des observateurs cette modification. Du côté de l'observateur, les instances `DigitalClock` et `AnalogClock` doivent être également étendues par la fonction `update()` requise par la classe abstraite de base. D'une manière générale, un nombre assez important de modifications sujettes aux erreurs doit être réalisé sur le code de nos trois classes. La Figure 1 illustre en rouge les parties de l'implémentation qui sont affectées. Du point de vue de la programmation orientée aspect, la préoccupation du protocole de l'observateur (ou *observer protocol concern*, en anglais) traverse de manière à la fois statique et dynamique les classes participant au processus `ClockTimer`, `DigitalClock` et `AnalogClock`. Ainsi, il est plus judicieux de la diviser et d'en faire un aspect.

Comment résoudre les préoccupations transverses dynamiques ?

Nous connaissons déjà les éléments indispensables du langage AspectC++ pour implémenter une préoccupation dynamique dans l'exemple qui suit. Le protocole observateur exige que l'ensemble des méthodes altérant l'état dans la classe sujet appelle la fonction `updateAll()` avant de retourner. Sous C++, l'ensemble des fonctions membres non-`const` d'une classe peut être considéré comme une modification d'état. La définition suivante de la méthode d'aspect a pour objectif d'insérer les appels nécessaires vers la fonction `updateAll()` dans notre classe `ClockTimer` :

```
advice execution("% ClockTimer::%(...)") &&
      !execution("% ClockTimer::%(...) const") : after () {
   updateAll ();
}
```

L'expression de la coupe transverse de cette méthode d'aspect doit se comprendre comme suit : « un point de jonction est un élément de la coupe transverse résultante s'il s'agit de l'exécution d'une fonction membre `ClockTimer` *et non* de l'exécution d'une fonction membre `ClockTimer` déclarée comme `const` ». La raison pour laquelle ce genre d'expression « *et non* » est précisée est due au fait que dans une expression de correspondances, `const` est interprété comme une restriction. Si `const` n'est pas précisé, les fonctions `const` et non-`const` sont toutes deux appariées.

Introductions – Implémentation des préoccupations statiques

La préoccupation dite statique, exposée dans l'exemple, peut être implémentée au moyen d'une caractéristique d'AspectC++ appelée *introductions*. Une introduction n'est rien d'autre qu'un genre de méthode d'aspect pour laquelle l'emplacement est une expression de coupe transverse, représentant un ensemble de classes, alors que la façon d'affectation est une déclaration, censée être introduite dans les classes. Ainsi, par exemple, la fonction `update()` pourrait être introduite dans les classes observateur comme suit :

```
advice "DigitalClock"||"AnalogClock" : void update() {
   Draw();
}
```

Il est intéressant de remarquer que les membres introduits ne sont pas seulement visibles par l'aspect. La fonction `update()` peut être appelée, par exemple, par d'autres membres de `DigitalClock` ou `AnalogClock` comme s'il s'agissait d'une fonction membre des plus ordinaires. Toutefois, les introductions ne se limitent pas qu'aux fonctions membres. Elles peuvent être utilisées dans le but d'introduire des membres de données, des classes intégrées ainsi que tout autre élément syntaxiquement autorisé dans la définition d'une classe.

Les *introductions de classes de base* constituent des introductions d'un genre spécial, et sont chargées d'introduire de nouvelles classes dans la liste des classes de base. Elles s'avèrent très utiles dans notre exemple, puisque le sujet ainsi que les observateurs doivent dériver respectivement des rôles `ISubject` et `Iobserver` :

```
advice "DigitalClock"||"AnalogClock" : baseclass(IObserver);
advice "ClockTimer" : baseclass(ISubject);
```

Coupes transverses virtuelles et aspects abstraits

Nous disposons désormais de tous les éléments permettant d'assembler un aspect `ObserverPattern` pour notre exemple. Toutefois, appliquer le protocole de l'observateur à un ensemble de classes demeure une tâche périodique. Or nous souhaitons réaliser une implémentation réutilisable. Pour ce faire, il nous faut disposer de deux autres caractéristiques du langage AspectC++.

La première est la capacité de ce langage à nommer des coupes transverses. Par exemple, l'expression de la coupe transverse `"DigitalClock" || "AnalogClock"`, utilisée plusieurs fois, peut devenir une coupe transverse nommée `observers()`:

```
pointcut observers() = "DigitalClock"||"AnalogClock";
```

Une autre caractéristique des coupes transverses nommées encore plus intéressante est que ces dernières peuvent être déclarées *virtual* (*virtuelles*) ou *pure virtual (virtuelles pures)*. Les coupes transverses virtuelles pures peuvent être utilisées par une méthode d'aspect sous forme de coupes transverses ordinaires. Un aspect qui a recours à des coupes transverses dites virtuelles pures définit uniquement la façon dont une préoccupation transverse est implémentée, sans indiquer l'emplacement où cette dernière affectera le programme.

```
pointcut virtual observers() = 0;
pointcut virtual subjects() = 0;
advice observers() : baseclass(IObserver);
advice subjects() : baseclass(ISubject);
```

Par conséquent, l'aspect est incomplet. On appelle ce phénomène un *aspect abstrait*. Cette technique est très proche des classes dites abstraites dotées de fonctions membres virtuelles pures, ne pouvant pas être utilisées pour l'instanciation. Les aspects dits abstraits n'affectent pas le programme tant qu'il n'existe pas d'aspect dérivé, chargé de définir la coupe transverse virtuelle pure de son aspect de base.

Héritage d'un aspect

Si nous combinons tous ces éléments ensemble, nous obtenons l'aspect réutilisable `ObserverPattern` exposé dans le Listing 1. L'aspect `ObserverPattern` est complètement indépendant de nos trois classes données dans l'exemple. Il ne définit que la façon dont le modèle observateur traverse une implémentation, mais non son emplacement. Ce dernier devra être définit au moyen d'un aspect dérivé, illustré dans le Listing 2.

Listing 3. *Aspect chargé d'émettre des erreurs de type Win32 en tant qu'exceptions*

```
#include <strstream>
#include "win32-helper.h"
aspect ThrowWin32Errors {
   using namespace std;
   // modélise le métaprogramme afin de générer du code pour
   // transmettre en continu une séquence
   // d'arguments de type CSV
   template< class TJP, int N >
   struct stream_params {
      static void process( ostream& os, TJP* tjp ) {
         os << *tjp->arg< TJP::ARGS - N >() << ", ";
         stream_params< TJP, N - 1 >::process( os, tjp );
      }
   };
   // spécialisation afin de mettre fin à la récursivité
   template< class TJP >
   struct stream_params< TJP, 1 > {
      static void process( ostream& os, TJP* tjp ) {
         os << *tjp->arg< TJP::ARGS - 1 >();
      }
   };
   advice call( win32::Win32API() ) : after() {
      if( win32::IsErrorResult( *tjp->result() ) ) {
         ostringstream os;
         DWORD code = GetLastError();
         os << "WIN32 ERROR " << code << ": "
               << win32::GetErrorText(code) << endl;
         os << "WHILE CALLING: "
               << tjp->signature() << endl;
         os << "WITH: " << "(";
         // Génère une séquence d'opérations propre au point
         // de jonction afin de transmettre en continu
         // l'ensemble des valeurs des arguments
         stream_params< JoinPoint,
               JoinPoint::ARGS >::process( os, tjp );
         os << ")";
         throw win32::Exception( os.str(), code ); }
   }
};
```

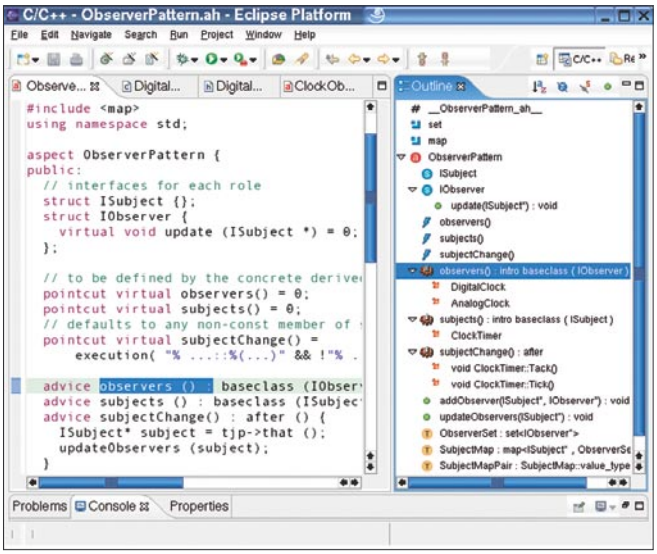

Figure 2. *ACDT en pleine action*

Notre aspect dérivé `ClockObserver` accomplit cette tâche en définissant les deux coupes transverses virtuelles pures héritées. Cet aspect se charge également d'implémenter la fonction introduite `update()` d'une manière propre à l'observateur.

Notions apprises

Nous avons fini par obtenir deux aspects. L'aspect de base abstrait réutilisable `ObserverPattern` encapsule l'implémentation du protocole de l'observateur. Voilà un avantage évident, puisque autrement, la décision de conception correspondante serait câblée dans une douzaine de classes. Par exemple, en modifiant cet aspect, nous aurions pu facilement basculer entre une implémentation, chargée de stocker une liste d'observateurs dans chaque instance de sujets, et une implémentation, capable de gérer une structure centrale de données à cet effet.

L'aspect `ClockObserver` hérite de `ObserverPattern` et, de ce fait, réalise la liaison entre l'observateur abstrait et les rôles sujets avec les classes d'application concrètes `ClockTimer`, `DigitalClock` et `AnalogClock`. Cette technique présente à elle seule un autre avantage, dans la mesure où il n'est plus nécessaire de polluer les classes d'applications elles-mêmes avec un code propre au modèle. Ces classes peuvent désormais rester telles quelles, en conservant, par conséquent, toutes leurs propriétés facilitant leur compréhension ainsi que leur réutilisation.

Grâce à l'exemple `ObserverPattern`, nous avons pu présenté certains concepts d'une plus grande importance dans le langage AspectC++. En voici ci-dessous un résumé :

- *Introduction :* méthode d'aspect pour des points de jonction statiques. Les introductions peuvent être utilisées dans le but d'étendre la structure statique des classes au moyen d'éléments supplémentaires comme les fonctions membres, les membres de données, ou les classes dites locales,
- *Introduction de classe de base :* forme spéciale d'introduction dont le but est d'étendre la liste des classes de base qu'une classe hérité,
- *Point de jonction nommé :* expression de coupe transverse pouvant être référée par un identifiant. Les coupes transverses nommées peuvent être déclarées virtuelles ou virtuelles pures, afin de pouvoir les intégrer dans un aspect hérité,

- *Aspect abstrait :* aspect contenant au moins une coupe transverse ou une méthode virtuelle pure. Les aspects abstraits sont de nature incomplète, et n'affectent donc pas le programme jusqu'à son accomplissement au moyen d'un aspect dérivé,
- *Héritage d'aspect :* à l'instar de l'héritage des classes, les aspects peuvent hériter d'autres aspects.

Quoi d'autre ?

Les applications relatives aux aspects ne sont pas limitées au dépistage ou aux modèles. Il existe de nombreuses préoccupations transverses et, une fois maîtrisés les principes de base de la Programmation Orientée Aspect, les programmeurs seront bientôt capables de les identifier automatiquement et désireront disposer d'outils pouvant implémenter ces préoccupations de manière modulaire. Nous allons donc dès à présent nous intéresser à ce sujet dans notre dernier exemple, chargé de décrire un nouvel aspect dit de production. D'un point de vue technique, cet exemple permettra d'illustrer comment les implémentations d'aspects peuvent tirer profit des diverses informations fournies par l'API de point de jonction ainsi que par les techniques de programmation génératives.

Les aspects sous un C++ sophistiqué

Les programmeurs de C++ doivent souvent travailler avec les bibliothèques patrimoniales C telles que l'API Win32. Outre le fait que l'API ne soit pas orienté objet, la gestion des erreurs proposée par les fonctions de la bibliothèque ne répond pas aux critères d'une gestion des erreurs basée sur les exceptions, système que de nombreux programmeurs préfèrent à l'heure actuelle. La transformation du système de gestion des erreurs de C en une approche basée sur les exceptions serait sans aucun doute une tâche laborieuse et génératrice d'erreurs. Par ailleurs, il s'agit d'une préoccupation transverse, dans la mesure où l'ensemble des fonctions contenues dans l'API Win32 est censé être englobé par une fonction chargée de contrôler le résultat et de générer une exception en cas de détection d'une erreur.

Listing 4. *Fichier win32helper.h*

```
namespace win32 {
   struct Exception {
      Exception( const std::string& w, DWORD c ) { ... }
   };
   // Vérifie la présence de "valeur magique"
   // synonyme d'erreur
   inline bool IsErrorResult( HANDLE res ) {
      return res == NULL || res == INVALID_HANDLE_VALUE;
   }
   inline bool IsErrorResult( HWND res ) {
      return res == NULL;
   }
   inline bool IsErrorResult( BOOL res ) {
      return res == FALSE;
   }
   ...
   // Traduit le code de l'erreur Win32 en un texte lisible
   std::string GetErrorText( DWORD code ) { ... }

   pointcut  Win32API() =  "% CreateWindow%(...)"
         || "% BeginPaint(...)"
         || "% CreateFile%(...)"
         || ...
} // espace de nommage Win32
```

L'aspect intitulé `ThrowWin32Errors`, exposé dans le Listing 3 accomplit la même tâche tout en sollicitant moins d'efforts de la part du programmeur. En compilant l'application au moyen de cet aspect, l'API Win32 se comporte comme si des erreurs lui étaient signalées en émettant des exceptions. Toutefois, cette implémentation n'est pas chose aisée et mérite donc une explication.

Détection des erreurs de type Win32

La première étape pour réaliser une propagation des erreurs basée sur les exceptions consiste à détecter l'échec éventuel de l'invocation d'une fonction Win32. Les fonctions de l'API Win32 indiquent une situation d'erreur en retournant une *valeur magique* spéciale. La détection des appels de l'API qui ont échoué relève par conséquent, encore une fois, d'une préoccupation transverse dite dynamique. L'idée générale ici consiste à doter d'une méthode d'aspect de type after advice l'ensemble des appels vers les fonctions Win32. Dans le corps de la méthode d'aspect, la valeur de retour est censée être contrôlée afin d'émettre une exception en cas de détection d'une erreur :

```
aspect ThrowWin32Errors{
   advice call(win32::Win32API()) : after() {
      if(<magic value> == *tjp->result())  throw ...
   }
};
```

La méthode d'aspect affecte l'ensemble des fonctions de l'API décrites par la coupe transverse nommée (et définie de manière externe) `win32::Win32API()`, et contenant toutes les fonctions de l'API Win32 (voir le Listing 4). Dans le corps de la méthode d'aspect, la valeur de retour de la fonction Win32 appelée est restituée au moyen de la méthode intitulée `tjp->result()` de l'API du point de jonction. Cette méthode a pour objectif de retourner un pointeur vers la valeur actuelle du résultat, rendant de ce fait possible la modification du résultat. Nous le comparerons ici avec la valeur magique retournée par la fonction de l'API afin d'indiquer une erreur.

La définition de cette méthode, toutefois, ne fonctionne pas encore. Il subsiste le problème selon lequel cette valeur magique, devant être comparée avec le résultat, n'est pas toujours la même. Cette dernière dépend du type de retour émis par la fonction de l'API appelée. De nombreuses fonctions Win32 ne relèvent que du simple type `BOOL` et signalent une erreur en retournant la valeur `FALSE`. Toutefois, d'autres fonctions de cette API ont recours à des types comme `HWND`, `ATOM`, `HDC`, ou `HANDLE`. Pour chacun de ces types, il existe une valeur magique correspondante retournée en cas d'erreur. Les fonctions dites `ATOM`, par exemple, retournent la valeur `0` ; et les fonctions `HANDLE` retournent soit la valeur `NULL` soit `INVALID_HANDLE_VALUE`.

Méthodes d'aspect génériques

Une solution possible pour résoudre ce problème consiste à filtrer les fonctions dans la coupe transverse `win32::`

`Win32API()` pour chaque type de retour et de les doter de méthodes d'aspect spécifiques de la façon suivante :

```
aspect ThrowWin32Errors {
   advice call(win32::Win32API() && "BOOL %(...)") : after() {
      if(FALSE == *tjp->result())  throw ...
   }
   advice call(win32::Win32API() && "HANDLE %(...)") : after() {
      if((NULL == *tjp->result())
            || (INVALID_HANDLE_VALUE ==
            *tjp->result())) throw ...
   }
... // ainsi de suite
};
```

Cette solution comporte toutefois certains inconvénients. Il faut en effet écrire presque la même définition de méthode d'aspect encore. Pire encore, en cas d'oubli d'un type ou d'introduction d'un nouveau par Microsoft, les fonctions de l'API associées seront omises par l'aspect, dans la mesure où ces dernières ne correspondent à aucune définition existante des méthodes d'aspect. Il nous faut donc opter pour une meilleure solution, présentant moins de fragilité :

```
aspect ThrowWin32Errors {
   advice call(win32::Win32API()) : after() {
      if(win32::IsErrorResult(*tjp->result()))  throw ...
   }
};
```

Une fois séparé tout le code dépendant des types (soit la comparaison avec les valeurs magiques dépendant du type) dans une fonction propre `win32::IsErrorResult()`, devant être surchargée pour chaque type de retour (voir le Listing 4), et selon le type statique du moment de `*tjp->result()`, le compilateur cherche une version compatible de `win32::IsErrorResult()` et, plus important encore, nous signale s'il ne peut en trouver une. De cette façon, il n'est plus possible que certaine fonctions passent sous silence, en raison de leur type de retour oublié.

La définition de la méthode d'aspect exposée ci-dessus est un exemple de *méthodes d'aspect dites génériques*. Cette méthode est qualifiée de générique dans la mesure où elle s'adapte à l'implémentation du moment (valeur magique à contrôler) tout en respectant certaines informations (type de retour de la fonction correspondante) issues du contexte du point de jonction concerné. Ce procédé est très proche des techniques utilisées dans les modèles des bibliothèques pour la programmation générique, comme STL (Standard Template Library).

Rapporter les erreur de type Win32

Une fois maîtrisée la manière de détecter des appels de l'API Win32 qui ont échoué grâce à la fiabilité de nos aspects, l'étape suivante consiste à en faire état sous forme d'exceptions. L'objet exception doit comprendre l'ensemble des informations contextuelles, pouvant être utiles pour connaître la raison actuelle d'un tel échec. En plus du code relatif à l'erreur Win32, il doit également comprendre une chaîne lisible écrite en langage humain, chargée de décrire l'erreur en question, la signature de la fonction appelée (restituée au moyen de `JoinPoint::signature()`) ainsi que les valeurs du paramètre concerné passées dans la fonction.

Méthodes d'aspect génératives

La génération d'un chaîne décrivant les valeurs du paramètre concerné demeure difficile. La solution consiste à transmettre en continu chaque paramètre en un objet std::ostringstream. Toutefois, comme la méthode d'aspect affecte les fonctions au moyen de signatures différentes, son implémentation doit être générique afin de respecter le nombre et les types d'arguments des fonctions. Les appels de la séquence de `operator <<(std::ostream&, T)` doivent donc être générés en fonction de la signature de la fonction affectée. Ce procédé est réalisé (voir le Listing 3) en renseignant les informations fournies par l'API du point de jonction (voir le Tableau 1) en un petit modèle de métaprogramme. Ce modèle de métaprogramme est instancié au moyen du code de la méthode d'aspect avec le type JoinPoint et des itérateurs, grâce à l'instanciation du modèle sur la liste des type d'arguments propres au point de jonction `JoinPoint::Arg<I>`. Pour chaque type d'arguments, une classe `stream_params` dotée d'une méthode `process()` est générée, avec pour objectif de transmettre en continu ultérieurement lors de l'exécution la valeur de l'argument indiquée en type (restituée au moyen de `tjp->arg<I>()`) et d'appeler de manière récursive `stream_params::process()` pour l'argument suivant.

Notions apprises

Comme l'illustre `ThrowWin32Errors`, les aspects peuvent non seulement être utilisés avec un logiciel orienté objet, mais fournissent également certains avantages permettant d'amé-

Tableau 1. *Exceptions issues de l'API du point de jonction d'AspectC++*

Types émis lors de la compilation et énumérateurs	
`That`	type de la classe affectée
`Target`	type de la classe de destination (pour les points de jonction d'appel)
`Arg<i>::Type` `Arg<i>::ReferredType`	type de l'argument i'th avec 0 \leq i < ARGS
`Result`	Type du résultat
`ARGS`	Nombre d'arguments
Runtime méthodes statistiques	
`const char *signature()`	signature de la fonction affectée
Runtime méthodes non-statistiques	
`void proceed()`	Code original d'exécution (méthode d'aspect around)
`That *that()`	Instance d'objet à laquelle se réfère this
`Target *target()`	Instance de l'objet cible d'un appel (pour les points de jonction d'appel)
`Arg<i>::ReferredType *arg()`	Instance de la valeur argument de l'argument i'th
`Result *result()`	Instance de la valeur résultat

liorer le code patrimonial C. Ils peuvent par ailleurs être implémentés de façon complètement générique en exploitant d'autres techniques sophistiquées de C++ telles que la programmation générique et générative. Les concepts sous AspectC++ pour cette combinaison sont les suivants :

- *Méthode d'aspect générique :* méthode d'aspect ayant recours à des informations de type statique issues du contexte du point de jonction dans le but d'instancier ou de relier du code générique,
- *Méthode d'aspect générative :* méthode d'aspect dotée d'une implémentation en partie générée au moyen de l'instanciation du modèle des métaprogrammes grâce aux informations de typa statique issues du contexte du point de jonction concerné.

Quoi d'autre ?

Vous avez vu maintenant les plus importantes constructions du langage AspectC++. Les aspects `Tracing`, `ObserverPattern`, et `ThrowWin32Error` ne sont bien sûr que des exemples illustrant les nombreuses variations différentes de préoccupations transverses que la Programmation Orientée Aspect peut résoudre. Il est probable que vous ayez déjà certaines idées pour utiliser le Programmation Orientée Aspect dans vos propres projets écrits en C++. Considérons dès à présent les outils disponibles d'AspectC++ destinés à cet effet.

Support d'outils

La Programmation Orientée Aspect propose de nombreux moyens afin de modulariser l'implémentation de préoccupations transverses dans les aspects. De ce fait, le code des aspects doit être tissé dans les composants affectés afin d'élaborer le programme final. Un parseur d'aspect est donc requis pour accomplir cette tâche.

Par ailleurs, le support pour l'outil consistant à visualiser les points de jonction n'est pas obligatoire mais fortement recommandé. Les aspects peuvent modifier potentiellement le programme n'importe où. Sur de gros projets, ceci sous-entend le danger de comportements de programme surprenants, si les développeurs chargés de travailler sur le code du composant ne sont pas conscients des aspects. Par conséquent, l'ensemble des points de jonction, réellement affectés par un aspect, est censé être marqué automatiquement dans le code. Les développeurs peuvent alors facilement voir les emplacements où certains aspects affectent leur code.

Sur le réseau

- Page d'accueil du projet AspectC++
  http://www.aspectc.org/
- Outil de développement AspectC++ (ACDT) pour Eclipse
  http://acdt.aspectc.org/
- Portail Web sur tous ce qui concerne le développement de logiciel orienté aspect (aspect-oriented software development, ou AOSD en anglais)
  http://www.aosd.net/
- Société proposant l'outil compagnon d'AspectC++ pour Visual Studio .NET ainsi qu'un support commercial pour les utilisateurs d'AspectC++
  http://www.pure-systems.com/

Parseur AspectC++

Le parseur sous AspectC++ s'appelle ac++. Il s'agit d'un parseur source à source capable de transformer les programmes d'AspectC++ en programmes C++. Il peut donc être utilisé en combinaison avec n'importe quel compilateur conforme au standards de C++ comme compilateur principal, g++ (version 3.x) et Microsoft C++ (VisualStudio.NET) étant plus particulièrement supportés.

Afin d'identifier les points de jonction correctement, ac++ réalise une analyse syntaxique et sémantique complète de ses données d'entrée en AspectC++. Compte tenu de la complexité du langage C++, le projet peut être considéré comme très ambitieux. Toutefois, ac++ peut déjà analyser un code de C++ réel et même des modèles complexes tels que définis par la bibliothèque STL ou Microsofts ATL, sans aucun problème. Des modèles de bibliothèques plus sophistiqués tels que Boost seront supportés dans un proche avenir.

Plugiciel ACDT pour Eclipse

L'outil de Développement d'AspectC++ pour Eclipse (ACDT) est un plugiciel Eclipse basé sur le code du projet CDT (C/C++ Development Tool). Il est chargé d'étendre les outils de développement C++ en ajoutant une syntaxe soulignant les mots clés d' AspectC++, une description étendue (montrant les aspects, les méthodes d'aspect, et les coupes transverses, voir la Figure 2), un constructeur pour les projets *Managed Make*, et la visualisation des points de jonction dans l'affichage de la description ainsi que dans l'éditeur du code source même pour les projets *Standard Make* basés sur votre propre fichier Makefile.

Compagnon d'AspectC++ pour VisualStudio.NET

Une extension commerciale VisualStudio.NET pour AspectC++ est disponible sur le site www.pure-systems.com. Cet outil supporte diverses extensions de langages spécifiques à Visual C++ et constitue donc un choix de premier ordre pour les utilisateurs habitués à l'IDE VisualStudio ainsi qu'au compilateur Microsoft Visual C++. Une version d'évaluation gratuite est disponible.

Résumé et conclusion

Principalement connue uniquement dans le monde Java, la Programmation Orientée Aspect est tout aussi bien adaptée à des projets C++. Le présent article a présenté les concepts ainsi que les caractéristiques les plus importants du langage AspectC++. Les programmeurs peuvent tirer profit de diverses manières d'une extension de langage orienté aspect. Les aspects de développement comme Tracing constituent un bon début pour ceux qui souhaitent faire appel à la Programmation Orientée Aspect et peuvent d'ores et déjà faire gagner aux programmeurs un temps précieux. Dans certains projets à grandeur réelle, nous avons évalué à 25% le nombre de lignes de code consacré au dépistage, au profilage, ou aux contrôles de contraintes. Les aspects dits de production sont présents partout. Comme l'ont bien illustré les exemples, ces aspects sont capables de simplifier la conception, l'implémentation, et même la gestion des bibliothèques patrimoniales.

Après avoir lu le présent article, vous saurez déjà maîtriser les premières étapes *menant* à la Programmation Orientée Aspect. ■