ARCHITECTURE DOCUMENT-VUE

Par Michel Michaud, nov.1999

Les applications produites par AppWizard sont dotées de deux classes importantes pour la structure générale du programme : ce sont les classes dérivées de `CDocument` et `CView`. La structure de programme qui en résulte peut et doit affecter la façon dont nous décomposons les opérations que le programme doit faire. Pour en tirer parti adéquatement, il faut respecter certains principes qui ne sont malheureusement pas imposés automatiquement. Au contraire, les outils de Visual C++ semblent nous indiquer une voie, mais ce n'est pas toujours la bonne. Le respect de certains principes rend non seulement le développement plus facile, mais permet aussi d'intégrer facilement certaines possibilités aux programmes. Il est donc primordial de commencer dès que possible à appliquer ces façons de faire, qui nous aideront encore plus par la suite.

Le schéma de conception «Observer»

L'architecture document-vue est une des variantes du schéma de conception (*design pattern*) *Observer*. Les schémas de conception sont théoriquement des solutions viables, utilisées et étudiées soigneusement, qu'on peut appliquer à des problèmes qui surviennent fréquemment en programmation orientée objet. Le but principal de *Observer* est de permettre une relation un à plusieurs entre des objets pour que lorsqu'un objet change, tous les autres objets dépendants soient mis à jour. On parle en principe du *sujet* et de ses *observateurs*. Dans le cas qui nous intéresse, la relation est entre un «document» et ses «vues». Même quand nous ne faisons qu'une seule vue, une approche document-vue permet de bien délimiter des responsabilités des objets. Cette description des responsabilités combinée aux possibilités offertes par les classes qu'on utilise (dérivées de `CDocument` et `CView`), ne laissera que peu de place à l'improvisation. C'est tant mieux : on peut programmer en toute sécurité, sachant que l'on arrivera à une architecture de programme qui sera fonctionnelle et extensible.

Responsabilité du document

La responsabilité première de l'objet document est de conserver les données, de les mettre à jour selon les demandes de changements et de répondre aux demandes d'information sur elles. Donc il a un rôle de «gardien» des données, prêt à accepter certaines demandes contrôlées à leur sujet. Lorsque les données sont modifiées, il doit avertir les vues. La première conséquence de ces responsabilités est que le document doit conserver ses données «jalousement» en particulier en les gardant toutes privées. Il ne doit pas non plus, par l'entremise de fonctions un peu trop générales, donner des accès indirects aux données qui permettraient des modifications des données sans que le document ne soit au courant. Ainsi le document ne devrait jamais donner de référence à des données, par des itérateurs ou des pointeurs, sauf si leur déréférencement donne une valeur `const`. Il ne faut surtout pas donner d'accès général aux collections de données qu'il conserve. Si la vue possède une valeur permettant d'indiquer une donnée précise (indice, itérateur, clef ou autres), le document doit simplement fournir des fonctions qui acceptent ces valeurs et qui font les opérations demandées sur la donnée correspondante.

De plus, le document devrait fournir des fonctions pour la plupart des opérations qui manipulent ou consultent l'ensemble des données. Par exemple, la vue et les boîtes de dialogue qui ont besoin de faire remplir des listes, pourront demander ce service au document. Si la vue a besoin de mettre à jour des contrôles qui dépendent des données du document, par des fonctions `OnUpdate` par exemple, elle demandera au document les informations nécessaires plutôt que de conserver une copie de ces informations. Ainsi la vue ne devrait pas savoir si elle affiche la dernière donnée d'une collection du document ou même en connaître le nombre : pour savoir cela, elle consultera toujours le document.

Responsabilité de la vue

La responsabilité des vues est l'interaction avec l'utilisateur du programme. Par opposition, le document ne doit pas communiquer directement avec les utilisateurs. Par exemple, même si le programme permet de changer la langue d'affichage, ceci ne devrait avoir aucun effet sur le document. D'ailleurs un programme pourrait très bien fournir en même temps deux vues différentes des données, une en français et l'autre en chinois ! Que pourrait faire le document ?...

Le travail des vues est complété par les boîtes de dialogue. Il faut voir celles-ci comme des extensions de la vue : elles doivent être sous sa responsabilité. C'est pourquoi les boîtes de dialogue ne doivent généralement pas faire des opérations qui sont sous la responsabilité de la vue. Par exemple, les opérations menant à l'ajout d'un nouveau client sont sous la

responsabilité de la vue. L'option du menu ou de la barre de bouton qui initie cette opération, va mener à une fonction de la vue. Celle-ci peut peut-être faire la saisie des données directement, si elle est dérivée d'un `CFormView`. Si par contre elle doit utiliser une boîte de dialogue, celle-ci ne s'occupera que de cette saisie, ce sera *sa* responsabilité, et c'est pourquoi on pourra l'appeler `CSaisieClient` par exemple. C'est ensuite la vue qui prendra les données saisies et terminera l'opération en demandant l'ajout au document, et en faisant toute autre opération subséquente. On comprend qu'il serait possible de faire l'ajout dans le document directement à partir de la fonction `OnOk` de la boîte de dialogue, puisque la boîte a probablement un pointeur sur le document pour permettre les validations, mais ce n'est pas une bonne idée, autant au niveau de la logique que la lisibilité du programme. L'interaction entre la vue et le document peut parfois devenir complexe et cacher une partie des opérations dans une fonction de la boîte de dialogue n'est définitivement pas une bonne idée.

Interaction vue-document

Un des aspects moins clairs de l'architecture document-vue, vient de la possibilité des vues de faire demander leur mise à jour par le document. Supposons simplement qu'une vue présente à l'écran, dans une zone de texte statique, le nombre de clients inscrits. Elle a bien sûr fait cet affichage en demandant la valeur au document. Après l'ajout d'un nouveau client, cette valeur devrait évidemment changer, mais il y a au moins trois façons de faire et il faut choisir la bonne. Une première serait pour la vue d'ajouter 1 à la valeur qui est déjà affichée. Nous avons dit que la vue ne devrait pas conserver cette valeur, mais il est possible que la valeur soit dans une variable associée au contrôle à l'écran. Il ne faut pas utiliser cette façon de faire : la méthode d'affichage pourrait changer et la valeur maintenant disponible pourrait bien ne plus l'être. Une deuxième façon serait pour la vue d'appeler directement sa fonction d'affichage qui demanderait la nouvelle valeur au document. Une dernière façon serait que la vue ne fasse rien directement, mais que le document, après l'ajout, demande la mise à jour des vues, ce qui ferait appeler indirectement la fonction d'affichage du nombre de clients.

L'avantage de la dernière façon de faire, qui constitue en fait le principe de base de l'approche document-vue (et de *Observer*) est qu'il est facile de l'étendre à une application multi-vues. Une modification dans une vue, ferait mettre à jour les autres vues, même si chaque vue ne connaît pas l'existence des autres vues de l'application. On sait qu'on implante cette façon de faire en laissant le document appeler `UpdateAllViews` et en mettant le code pour la mise à jour des vues dans leur fonction `OnUpdate`. À moins de contre-indication, cette façon de faire est la meilleure.

Par contre, dans certains cas, les opérations à faire dans une vue pour qu'elle se mette à jour complètement paraissent trop longues ou complexes, pour qu'on en fasse la demande lors d'une «petite» modification. Dans ce cas, l'architecture permet bien sûr de provoquer les modifications à partir de la vue (la deuxième façon de faire ci-haut). Elle permet en plus de provoquer quand même la mise à jour complète ou partielle des autres vues. Pour faire cela, il faut tirer parti des paramètres de `UpdateAllViews` et de `OnUpdate`. D'abord, le premier paramètre de `UpdateAllViews` peut être l'adresse d'une vue ou un pointeur nul (0 ou NULL). Le document demande la mise à jour (par appel de OnUpdate) de toutes les vues sauf celle dont l'adresse a été passée à `UpdateAllViews`. En fournissant l'adresse de la vue à une fonction du document (en passant `this`) pour indiquer que la mise à jour n'est pas nécessaire, on pourra utiliser adéquatement ce paramètre.

Une autre façon de faire, probablement meilleure en général, est d'élaborer un protocole entre le document et les vues pour que celui-ci indique quelles sont les modifications qui ont été apportées aux données et donc quelles sont les choses que les vues devraient mettre à jour. Cette information peut être fournie aux vues en utilisant les deuxième et troisième paramètres de `UpdateAllViews`. Ces paramètres ont des valeurs par défaut et c'est pourquoi on voit souvent des appels à cette fonction utilisant un seul paramètre. Cependant les valeurs que l'on donne à ces paramètres sont récupérées par les paramètres de `OnUpdate`, en fait même le premier paramètre s'il n'est pas nul. Le deuxième paramètre permet de passer une valeur numérique (un `LPARAM` correspond à un `long`, soit un entier 32 bits). Le troisième permet de passer un pointeur sur un objet dérivé de `CObject`. C'est donc plus facilement le deuxième paramètre qui pourra être utile. Par exemple on pourrait passer une valeur d'un type par énumération, un numéro d'enregistrement, etc., qu'on pourrait ensuite convertir dans le bon type au besoin, dans la fonction `OnUpdate`, afin de prendre connaissance de l'information véhiculée. Si l'on peut ainsi éviter des modifications inutiles, cette façon de faire est certainement à étudier sérieusement.

Par ailleurs, on peut parfois retarder les mises à jour en demandant des modifications simples au document, puis en appelant directement à partir de la vue le `UpdateAllViews` du document. Cette solution est pratique lorsqu'on fait des transactions complexes, en plusieurs étapes, mais il est dangereux d'oublier certaines mises à jour des vues.

Cependant, avant d'établir que la vue est «trop lente» à se mettre à jour, il faut s'en assurer. Il est inutile de faire un programme inutilement complexe, si la vue peut se mettre globalement à jour en une fraction de seconde. Dans ce cas, la

meilleure solution demeure pour la vue d'attendre les appels de `OnUpdate` pour se mettre à jour et, pour le document, de demander la mise à jour des vues après chaque modification. Le programme résultant sera d'une grande clarté.

Les boîtes de dialogue vs la vue

Comme on l'a indiqué précédemment, les boîtes de dialogue ne devraient pas demander directement des modifications au document. Dans certaines boîtes de dialogue, il faut parfois en appeler une autre pour faire l'ajout d'une donnée utilisée par la première (par exemple, pour entrer l'information sur un code postal qui n'est pas déjà inscrit dans la liste proposée pour l'ajout d'un client). Plutôt que de faire cette opération directement et devoir faire l'ajout au document ensuite, on choisira plutôt d'appeler une fonction d'ajout du code postal qu'on aura mise *dans la vue*. Cette fonction peut déjà être prévue dans la vue (si on veut permettre directement l'inscription des codes postaux par exemple) ou on pourra l'ajouter spécifiquement pour ce besoin (on trouvera probablement plus tard qu'on en avait besoin de toute façon). Il faudra que la boîte de dialogue ait alors l'adresse de la vue : on l'aura passée au constructeur comme on passait déjà celle du document...

Par ailleurs, les boîtes de dialogue doivent communiquer les données saisies à la vue et, inversement, lors des modifications de données c'est la vue qui doit fournir des données aux boîtes de dialogue. La chose est assez simple quand les données sont de simples structures, mais elle demande plus de préparation lorsque les données sont normalement manipulées comme des classes dans le programme.

Rappelons d'abord la façon de faire que nous utilisons habituellement.

1) Nous utilisons autant que possible la même classe de boîte de dialogue pour la saisie des nouvelles données que pour les modifications à des données existantes. Il est même possible d'utiliser la même classe pour la simple présentation des données. Afin d'être général, nous nommons donc ces classes `CSaisieXXX`, par exemple `CSaisieClient`.

2) Pour différencier les opérations dans cette classe, nous définissons un type par énumération, que nous utilisons au niveau du constructeur afin d'identifier la fonction de la boîte de dialogue. Cette valeur permet à OnInitiDialog d'adapter la boîte selon l'opération à effectuer. Par exemple :

```
CSaisieClient d(CSaisieClient::AJOUT, ...);
```

3) Si la boîte sert pour des modifications, on change la valeur initiale en utilisant une fonction membre, qu'on nomme `Initialiser` :

```
CSaisieClient d(CSaisieClient::MODIFICATION, ...);
d.Initialiser(clientAModifier);
```

4) Dans tous les cas, on récupère le résultat par une fonction `Extraire` :

```
if (IDOK == d.DoModal())
    {
    ClClient client;  // ou TypeClient client;
    d.Extraire(client);
    ...
```

5) Au besoin, on aura aussi passé le document à la boîte de dialogue, par le constructeur.

On pourrait penser passer la valeur à modifier au constructeur plutôt que d'utiliser une fonction séparée. C'est possible, mais il y a deux problèmes. D'abord, il faudrait aussi passer une valeur, bidon, lors de l'utilisation de la boîte pour un simple ajout. On pourrait résoudre ce problème en créant deux constructeurs différents, ce qui permettrait aussi de se passer du type par énumération, sauf pour les cas où la boîte sert aussi pour la simple présentation des données (avec la valeur du type par énumération `CSaisieClient::AFFICHAGE` par exemple) ou pour d'autres opérations. Pour pouvoir toujours utiliser le même principe, le type par énumération est plus flexible et la déclaration de la boîte de dialogue est aussi plus claire. L'autre raison pour avoir une fonction `Initialiser` est qu'on peut vouloir réutiliser la même boîte de dialogue en changeant ses valeurs, en particulier dans les boîtes servant à l'affichage, ou même pour des boîtes servant à l'ajout, si on veut partir d'une copie des données déjà saisies. On ne veut donc pas limiter l'initialisation au constructeur et il paraît finalement plus simple de toujours se servir de la fonction `Initialiser`. On comprend quand même que le constructeur initialise les données à des valeurs utiles pour la saisie d'un nouvel élément.

L'écriture des fonctions `Initialiser` et `Extraire` n'est pas toujours aussi simple qu'il n'y paraît. Notons d'abord que c'est dans ces fonctions que devra avoir lieu la conversion entre la représentation des données dans la boîte de dialogue et

leur représentation dans les variables des types `struct` ou `class`. Par exemple, les types par énumération sont habituellement représentés par des groupes de boutons radios ou des listes non modifiables.

Pour une `struct TypeClient`, on pourrait donc avoir :

```
void CSaisieClient::Initialiser(const TypeClient& p_client)
    {
    m_nom= p_client.nom;
    m_etat= p_client.etat; // Les valeurs numériques du type par énumération sont correctes...
    ...
```

et

```
void CSaisieClient::Extraire(TypeClient& p_s_client)
    {
    p_s_client.nom= m_nom;
    p_s_client.etat= static_cast<TypeEtatClient>(m_etat);
    ...
```

Malheureusement les choses ne sont pas aussi simples si on a une classe `ClClient`, parce qu'on ne peut pas aussi facilement accéder aux divers champs des classes. À moins bien sûr d'avoir des fonctions d'accès pour chacun des champs, ce qui peut devenir très lourd et qu'on préfère éviter. Une «solution» est alors de demander à la classe le «service» de transfert de ses données vers et depuis les variables des boîtes de dialogue. Par exemple :

```
void CSaisieClient::Initialiser(const ClClient& p_client)
    {
    p_client.CopierDansDlg(m_nom, m_etat, ...);
    ...
```

et

```
void CSaisieClient::Extraire(ClClient& p_s_client)
    {
    p_s_client.AffecterDeDlg(m_nom, m_etat, ...);
    ...
```

On laisserait alors à ces fonctions le soin de convertir les valeurs entre les deux représentations afin de ne pas exposer le type des données membres. Cependant cette solution comporte plusieurs problèmes. D'abord elle expose à tout le programme les diverses valeurs conservées dans la classe. Même sans *getter* et *setter*, le reste du programme peut faire tous les accès et toutes les modifications qu'on voulait pourtant empêcher. Une façon de contourner ce problème serait de rendre ces fonctions privées et de mettre la classe de boîte de dialogue `friend`, mais il y a un autre problème de toute façon. Le couplage entre les deux classes est trop grand. Si la façon de faire la saisie change, on risque de devoir changer les fonctions de la classe `ClClient`, ce qui paraît anormal. On remarque cependant que le couplage en sens inverse est normal : si la classe `ClClient` change (ajoute ou retire une des données conservées) il faudra évidemment modifier la façon de faire la saisie.

Une meilleure solution est donc de laisser la boîte de dialogue faire tout le travail, en lui permettant de le faire. Si on pense aux `operator<<` et `operator>>` qu'on utilise pour les streams, on peut voir que les boîtes de dialogue de saisie ont à peu près le même rôle. Par conséquent, la solution est simplement de déclarer la classe de boîte de dialogue `friend` :

```
class ClClient
    {
    ...
    friend class CSaisieClient;
```

Un autre avantage de cette solution est que si on réutilise la classe `ClClient` dans un programme où il n'est pas nécessaire d'utiliser des boîtes de dialogue de saisie, on n'aura pas de fonctions inutiles dans la classe. La déclaration `friend` peut rester là, elle n'aura simplement aucun effet.

Les fonctions `Initialiser` et `Extraire` de la classe `CSaisieClient` seront maintenant semblable à celle qu'on utilisait avec les `struct`.

Les fonctions «call back»

On notera aussi que le même raisonnement peut s'appliquer à une fonction «call back» qui doit remplir un contrôle de liste. Comme la fonction doit fournir chacun des champs séparément, il faudrait une fonction d'accès (*getter*) pour chaque élément de données, ce qu'on veut éviter. Considérant que c'est aussi un fonction de «sortie» de la classe, on pensera la mettre `friend` au besoin, c'est-à-dire si la liste doit afficher assez de champs pour lesquels il n'y a pas déjà de fonction d'accès. En principe, c'est assez simple :

```
class ClClient
    {
    friend class CSaisieClient;  // Boîte de dialogue de saisie
    friend void CXyzView::OnGetDispInfoXyz(NMHDR* pNMHDR, LRESULT* pResult);
```

Mais en réalité, dans un programme Windows tel que généré par *AppWizard*, il n'y aura probablement pas eu de déclaration de la classe de vue (`CXyzView` dans mon exemple) avant la déclaration de `ClClient`. Et il ne sera pas facile de faire cette déclaration puisque la classe de vue fait référence à la classe du document, qui fait référence à `ClClient` ! Sans compter que la classe de vue peut directement faire référence à `ClClient` elle aussi ! Heureusement, il y a une façon de s'en sortir…

```
// Dans le fichier d'en-tête spécifique pour ClClient ou avant la déclaration de la classe document
// dans le fichier d'en-tête de cette classe :

//...
class CXyzDoc;       // Prédéclaration pour l'utilisation de CXyzDoc dans GetDocument de la vue
class ClClient;      // Au cas où la vue utilise ClClient dans son interface
#include "XyzView.h" // Pour la déclaration de
                     //  void CXyzView::OnGetDispInfoXyz(NMHDR* pNMHDR, LRESULT* pResult);

class ClClient
    {
    friend class CSaisieClient;  // Boîte de dialogue de saisie
    friend void CXyzView::OnGetDispInfoXyz(NMHDR* pNMHDR, LRESULT* pResult);
    ...
```

* * *