PRINCIPES DE BASE DE L'APPROCHE ORIENTÉE OBJET

Par Michel Michaud, nov.1999

La programmation orientée objet est une technique qu'on caractérise habituellement par trois facettes : l'encapsulation, l'héritage et le polymorphisme. D'une certaine façon, seule l'encapsulation est une caractéristique fondamentale, car certains programmes (simples) n'ont pas besoin de l'héritage ou du polymorphisme. L'encapsulation est aussi la facette qui semble la plus simple, celle qui est «facile à régler». De façon générale cependant, il importe de comprendre le rôle général des classes dans les programmes afin de comprendre le sens exact et l'intérêt réel de l'encapsulation. C'est ce que ce document tentera de montrer.

Les classes, représentation d'un concept

Il est parfois difficile de déterminer exactement ce qui constitue une classe. Les classes les plus faciles à identifier sont celles qui correspondent à des concepts du monde réel que le programme veut simuler ou utiliser. Pour être plus formel cependant, on peut simplement penser qu'un *objet* (instance d'une classe) doit être quelque chose qui a des «comportements». On provoque ces comportements par les appels aux fonctions membres. Les comportements d'un objet peuvent être différents d'une fois à l'autre, car les objets ne sont pas toujours dans le même «état». Ce sont généralement la valeur des données membres qui définissent l'état de l'objet. Les comportements peuvent évidemment changer l'état d'un objet. Il faut se dire que les états, et donc les données membres, sont là simplement pour permettre d'implanter les comportements désirés. Ce qui importe pour les utilisateurs de la classe, c'est que les objets aient les comportements attendus[1].

L'encapsulation et données privées : interface vs implantation

Une des premières choses qu'il faut donc respecter, si on veut obtenir un programme ayant une architecture solide, est que les classes ne devraient laisser *aucune* donnée publique[2]. Le fait de cacher ainsi l'*implantation* de la classe est ce qu'on appelle généralement *encapsulation*[3]. Cette limitation des possibilités permet de s'assurer de l'intégrité des objets et ramène au principe même de la manipulation d'un *objet* : il doit être manipulé par son *interface* c'est-à-dire par les fonctions membres qui sont publiques. L'utilisation exclusive de l'interface de la classe peut être préconisée, mais ce n'est qu'en mettant toutes les données privées[4] qu'on peut la garantir. *C'est ce que vous devrez faire*[5].

Identification des classes

Pour découvrir quelles classes un programme devrait définir, il faut d'abord considérer tous les éléments du domaine du problème et se demander s'ils doivent participer aux traitements qui seront faits par l'application. Les concepts qu'on retiendra et qu'on voudra représenter dans le programme seront probablement les classes les plus importantes de notre application. D'autres classes pourraient être utiles pour appuyer les opérations des classes déjà dégagées. En pensant au fait que les classes peuvent fournir des services (des comportements «utiles») et représenter des informations (les

---

[1] En programmation orientée objet classique, on parle souvent de *méthode* (pour les fonctions membres) et d'*attribut* (pour les données membres). En C++ ce vocabulaire n'est pas généralement utilisé. On peut comprendre le mot méthode en se disant que c'est la méthode (manière) à utiliser pour demander un certain traitement à un objet. D'ailleurs, formellement, les appels de fonctions membres sont considérés comme un *envoi de message* à l'objet, pour lui demander de faire un traitement.

[2] Nous parlons ici de classes représentant des concepts tels que préconisés par la programmation orientée objet. Il pourrait parfois être utile de camoufler dans une classe un ensemble de données qui serait normalement une simple `struct`. Cependant, pour ces cas, nous préférerons utiliser des `struct`, ayant au besoin des constructeurs, et fournir des fonctions ordinaires pour les manipuler. Ce sont alors les données elles-mêmes qui constituent l'interface (de la “classe” ici représentée par une `struct`).

[3] Ce terme est parfois utilisé pour indiquer le regroupement des données et des fonctions, particulièrement dans les langages où les fonctions devraient donner l'interface, même si les données ne sont pas nécessairement cachées.

[4] Il faut aussi rendre privé toutes les fonctions membres ne devant pas faire partie de l'interface.

[5] Dans les programmes Windows, ceci signifie qu'il faudra indiquer qu'on veut des données privées lors de l'emploi de *Add Member Variables* et qu'il faudra modifier le fichier d'en-tête pour les variables associées à des contrôles (car ces variables sont toujours mises directement dans les données publiques par *ClassWizard*).

données/états), on devrait pouvoir obtenir assez de classes pour avoir une bonne idée de la structure du programme. Au cours du développement, on pourra bien sûr identifier d'autres classes, mais il est important de ne pas commencer la programmation avant d'avoir établi au moins quelques classes fondamentales : le programme résultant pourrait avoir une structure à peu près impossible à modifier facilement lorsqu'on s'apercevra qu'une classe importante est manquante.

Identification des fonctions et données membres

Une fois qu'on a déterminé quelques classes à implanter, on voudra implanter les comportements désirés. On pourra donc identifier un certain nombre de fonctions membres des classes en se demandant ce que les objets devront permettre de faire, mais aussi ce dont ils doivent être garants : on parle alors de la *responsabilité* des objets. Pour implanter ces fonctionnalités, on aura probablement à intégrer des données à notre classe de façon à conserver son «état». Une autre catégorie de fonctions membres peut alors être utile : des fonctions d'accès permettant simplement de consulter une partie de l'état des objets (aussi appelées *getter* ou *accessor*). Ces fonctions sont parfois nécessaires pour permettre au reste du programme de faire des traitements qui dépendent de ces valeurs.

Certaines classes peuvent aussi comporter des fonctions de modification directe des éléments de l'état des objets (on les appelle *setter*, *modifier* ou *mutator*). Cependant, en général, il faut éviter les fonctions qui modifieraient un état directement, alors que des opérations générales sur l'objet seraient plus appropriées. C'est donc habituellement en faisant «agir» l'objet, en lui demandant de faire «quelque chose», que son état va changer. Si le besoin paraît grand de fournir beaucoup de fonctions de consultation et modification des éléments de l'état, c'est probablement que la classe de l'objet, c'est-à-dire ce qu'elle représente comme concept dans le programme, a été mal choisie ou conçue. Le concept représenté est probablement trop simplifié ou, au contraire, on a essayé de mettre trop de possibilités non reliées dans la classe.

Un exemple

Par exemple, supposons qu'on décide de faire une classe pour représenter des clients d'une banque, `ClClient`. Quelle devrait être l'interface offerte par cette classe ? On peut se poser cette question sans savoir ce qu'un client doit comporter comme données, mais il faut alors savoir quelles sont les opérations qu'on peut vouloir faire. Il y a des méthodes d'analyse formelle pour parvenir à connaître ces éléments, mais on peut aussi y aller intuitivement. Parce que la banque voudra probablement écrire ou téléphoner au client, on imagine que les informations «adresse» et «numéro de téléphone»[6] doivent être disponibles à partir d'un client. Comme nous n'en sommes pas rendus à des systèmes informatiques complètement informatisés, on peut penser qu'une fonction `ClClient::Telephoner` (à laquelle on passerait la raison de l'appel en paramètre !) n'est pas encore possible. Le mieux qu'on peut alors faire, c'est faire afficher ce numéro de téléphone quelque part pour qu'un préposé fasse l'appel lui-même.

On peut alors imaginer une variété de scénarios : cet affichage peut se faire à l'écran, sur des rapports imprimés, etc. Il est possible de faire des fonctions pour chacun des cas, ou on peut peut-être les regrouper en donnant l'endroit où l'écriture devra se faire en paramètre (un `CDC` par exemple), mais ce serait probablement complexe. Notre classe `ClClient` n'a qu'à fournir une fonction pour connaître le numéro de téléphone et ceux qui utiliseront cette fonction pourront alors faire ce qu'ils veulent. C'est ce qu'on appelle une interface minimale. Ici cet aspect est justifié parce que les opérations possibles avec un numéro de téléphone paraissent simples. Si le numéro de téléphone était structuré de façon plus complexe, par exemple en gardant le code régional, le numéro de base et le poste (facultatif) séparément (parce que l'analyse nous indique que ce peut être nécessaire), il serait plus utile de faire une classe pour celui-ci et de mettre dans cette classe une fonction permettant d'avoir une *string* formatée correctement à partir d'un numéro de téléphone; cette classe pourrait fournir cette fonction (`ClNoTelephone::EnString()`) en plus de toute autre fonction de manipulation des différentes parties du numéro.

La classe `ClClient` devra donc fournir une fonction permettant de connaître le numéro de téléphone. Selon qu'on utilise ou non une classe `ClNoTelephone`, le prototype serait du genre `xxx ClClient::NoTelephone()` avec `xxx` qui serait le type `string` ou `ClNoTelephone`. Ceci détermine l'interface de la classe pour le numéro de téléphone. Ce n'est qu'après avoir établi cette interface qu'il faut penser aux données membres. Le fait de conserver comme donnée une `string` ou un `ClNoTelephone` ne devrait être qu'une conséquence des besoins de l'interface. En fait, il serait tout

[6] Pour simplifier, on suppose ici qu'on ne conserve qu'un seul numéro de téléphone par client. En réalité, on aurait probablement un numéro de téléphone personnel et un autre, au travail.

à fait possible qu'on ne garde pas du tout une telle donnée : notre classe pourrait peut-être fouiller dans une base de données servant de bottin téléphonique et, à partir du nom et de l'adresse du client, retrouver le numéro de téléphone (vous pouvez penser à *http://canada411.sympatico.ca*). Il est fort probable que ça ne soit pas le cas (à cause des numéros de téléphone privés en particulier), mais rien ne nous oblige à ce que le numéro de téléphone soit conservé dans la classe de la même façon que les fonctions membres le rendront disponible. Par exemple, le numéro de téléphone pourrait être gardé dans une `string`, mais la fonction ClClient::`NoTelephone` composerait au besoin un objet de type `ClNoTelephone` pour le renvoyer. Ou vice-versa.

L'intérêt de l'encapsulation, c'est qu'on peut justement changer cette implantation de la fonction sans changer l'interface de la classe. Les données étant privées, on est certain que ces changements n'affecteront aucun utilisateur de la classe.

Pour revenir aux opérations qu'on veut permettre à partir d'un client, le cas de l'adresse peut paraître semblable à celui du numéro de téléphone. Cependant une adresse est clairement quelque chose de plus complexe qu'un simple numéro de téléphone. Selon toute probabilité, il ne s'agit pas d'une variable simple. Il faudrait peut-être un vecteur de lignes d'adresse ou une `struct` ou, mieux encore, une classe contenant l'équivalent du vecteur ou de la structure et fournissant les opérations courantes et utiles sur les adresses, par exemple `vector<string> ClAdresse::EnString()`, `string ClAdresse::CodePostal()` ou même `void ClAdresse::Afficher(CDC* p_pDC, int p_largeur, int p_hauteur)`. La classe `ClClient` ne serait alors plus la classe responsable de la manipulation des adresses. Le mieux qu'elle pourrait faire serait une fonction du style `ClAdresse ClClient::Adresse()` pour connaître l'adresse et une autre `void ClClient::Adresse(const ClAdresse& p_adresse)`, pour la modifier globalement.

Quelles seraient donc les opérations vraiment fournies par `ClClient` ? Un constructeur permettrait évidemment de donner un état initial comprenant plusieurs attributs : le nom, l'adresse, le numéro de téléphone, etc. Comme le nom d'une personne est généralement fixe, on n'aurait peut-être aucune fonction permettant de le modifier (*setter*), mais on pourrait certainement fournir une fonction donnant ce nom (*getter*). Ce qui rendra la classe particulière à une banque sera probablement la gestion des comptes. Un client pouvant avoir plusieurs comptes, il est probable que la classe conserve ces comptes (une classe `ClCompte` par exemple) dans une collection quelconque, comme un vecteur ou une liste. Par contre, cette implantation devrait être complètement cachée (et donc modifiable) et l'interface de la classe devra fournir une façon de gérer les comptes sans l'exposer. D'ailleurs il n'est pas dit que la collection contiendrait réellement des comptes. Elle pourrait simplement contenir les numéros de comptes et retrouver l'information complète dans une base de données à partir de ce numéro.

Dans l'interface de la classe, la façon d'accéder aux comptes ne devrait pas exposer l'implantation. Dans le plus simple des cas, on pourrait simplement avoir des fonctions du style `ClCompte& ClClient::CompteCheque()` où la fonction déterminerait le compte précis. Mais s'il peut y avoir plusieurs comptes d'un type particulier, une façon plus générale sera nécessaire. Par exemple, on pourrait fournir une fonction donnant le nombre de comptes du client (`int ClClient::NbComptes()`) et une fonction donnant un compte particulier par son numéro séquentiel `ClCompte& ClClient::Compte(int p_no)`. C'est une bonne approche en général, mais si les comptes sont gardés dans une liste (ou une autre structure sans accès direct), on pourrait avoir des problèmes d'efficacité s'il y en a beaucoup. Si on identifie que le problème peut survenir, il faut le prévenir même si la classe n'utilise présentement pas une liste, car nous ne voulons pas que la façon de garder les comptes soit exposée. On pourra donc fournir des fonctions plus générales, par exemple, transférant les noms de compte (ou plus) dans un vecteur ou ailleurs (dans un «list box» par exemple) pour utilisation ultérieure par le demandeur.

Dans certains cas, on pourrait même fournir un genre d'«itérateur» qui permettra de conserver une position (abstraite) dans les comptes tout en permettant un accès plus rapide. Cependant ces itérateurs ne devraient pas permettre des modifications directes de l'ensemble des comptes, pour ne pas briser l'encapsulation. C'est facile à faire si on ne fournit pas d'accès à la collection elle-même, ce qu'il faut évidemment faire. Dans certains cas, le déréférencement de ces itérateurs pourraient être interdits à l'extérieur de la classe hôte (`ClClient` dans notre exemple) ou simplement donner une valeur constante. Par contre, les fonctions de la classe feront des opérations sur les comptes en acceptant cette valeur. Par exemple, en fournissant les fonctions du type `ClClient::ClIterCompte ClClient::PremierCompte()` et `ClClient::ClIterCompte ClClient::FinDesComptes()`, on permettra le classique `for` qui passe à travers de tous les comptes. Par exemple :

```
double soldeTotal= 0.0;

for (ClClient::ClIterCompte it= client.PremierCompte(); it != client.FinDesComptes(); ++it)
    {
    soldeTotal+= client.Compte(it).Solde();
    client.FermerCompte(it);
    }
```

On notera qu'il n'y aurait probablement pas de fonction `ClCompte::Solde(double p_nouveauSolde)`. Au lieu de faire «manuellement» `compte.Solde(client.Solde()+montant)`, on fournira une fonction plus directe permettant de faire `compte.AjusterSolde(montant)`.

En général, on pourra appliquer la règle suivante : lorsque des opérations importantes peuvent s'appliquer sur un ensemble de données qui est conservé dans une classe, c'est la classe qui devrait fournir cette opération, puisque c'est elle qui est la mieux placée pour connaître la meilleure façon de faire. Cette règle peut évidemment être nuancée, d'abord parce qu'il est difficile de définir exactement ce que signifie «opérations importantes». Par exemple, c'est en sachant la fréquence des opérations qu'on pourrait décider si `ClClient` devrait fournir, ou non, une fonction `double ClClient::SoldeTotal()` ou `void ClClient::FermerTousLesComptes()`.

En général, il faut essayer d'éviter un couplage trop grand entre les classes, autant en évitant de mettre des paramètres de fonctions trop spécifiques, qu'en faisant faire le travail d'un classe dans une autre. Par exemple, la fonction `void ClClient::EcrireNomDans(CEdit& p_edition)` n'est pas très utile, en particulier si on a déjà une fonction donnant le nom en *string* et qu'on l'utiliser pour transférer la valeur dans le champs d'édition. Une telle fonction force un couplage avec la classe `CEdit`, et ce couplage n'apporte rien d'intéressant. On éviterait aussi de faire une fonction de modification des comptes qui partirait d'un client : on ne veut pas faire `client.AjusterCompte(it, montant)` alors que la responsabilité du traitement des comptes est celle de la classe `ClCompte`. Tout ce que pourrait faire la fonction `AjusterCompte`, c'est appeler `Compte(it).AjusterSolde(montant)` alors on laissera l'utilisateur faire cette opération lui-même (`client.Compte(it).AjusterSolde(montant);`).

***