Remarque : lors de la publication de cet article sur mon blog, beaucoup de personnes ont signalé que l'exemple choisi n'était probablement pas le plus pertinent. En effet, il n'est pas insensé d'avoir des accesseurs pour une classe aussi simple qu'un point 3D.

Philippe Dunski (koala01) a détaillé cette problématique dans son livre Coder efficacement - Bonnes pratiques et erreurs à éviter (en C++), je conseille fortement sa lecture.

Les accesseurs et les détails d'implémentation

C'est une discussion qui revient régulièrement sur le chat de Developpez.com. Une personne demande comment fait-on pour accéder aux variables membres privées d'une classe et on lui répond de créer des getter et setter. Viens alors un C++ien moyen (c'est-à-dire un casse-pied, en général moi) qui hurle au scandale et sort l'adage bien connu : « les accesseurs, c'est le mal ». S'en suit une discussion sur pourquoi les accesseurs sont à éviter, quand j'ai le temps et l'humeur. :)

Dans cet article, je vais présenter les problèmes que posent les accesseurs concernant l'exposition des détails d'implémentation.

Un exemple simple

Imaginons que l'on souhaite créer une classe représentant un point en 3D. On écrit alors le code suivant :

```
class Point3D {
    float x, y, z;
public:
    inline float x() const { return x; }
    inline float y() const { return y; }
    inline float z() const { return z; }
```

```
    inline void setX(float v) { x = v; }
    inline void setY(float v) { y = v; }
    inline void setZ(float v) { z = v; }
};
```

On utilise ensuite cette classe à plusieurs endroits dans notre code, par exemple dans une fonction glVertex (remarque : pour ceux, qui comme LittleWhite bloquent sur l'utilisation de glVertex, je précise que ce n'est qu'un code d'exemple ; glVertex est une fonction dépréciée et ne doit plus être utilisée) :

```
Point3D p { 1.0f, 1.0f, 1.0f };
glVertex3f(p.x(), p.y(), p.z());
```

Jusque là, tout va bien.

(Ou presque. Si on modifie les coordonnées d'un point, on obtient un nouveau point, bien distinct du premier. Les setters sont donc contraire au respect de la sémantique de valeur et doivent être évités.)

Là où les choses se gâtent

Nos besoins évoluent. Au lieu de simplement vouloir faire de la représentation 3D sur nos points, on doit également faire des calculs pour de la simulation physique. On a la chance d'avoir une carte graphique prenant en charge l'extension fp64 et on décide donc d'utiliser double au lieu de float :

```
class Point3D {
    double x, y, z;
public:
    inline double x() const { return x; }
    inline double y() const { return y; }
    inline double z() const { return z; }

    inline void setX(double v) { x = v; }
    inline void setY(double v) { y = v; }
    inline void setZ(double v) { z = v; }
};
```

Malheureusement, le code qui utilise cette classe Point3D doit également être modifié pour pouvoir compiler. En effet, glVertex3f prend comme paramètres des float, il faut maintenant utiliser glVertex3d qui prend comme paramètres des double :

```
Point3D p { 1.0, 1.0, 1.0 };
glVertex3d(p.x(), p.y(), p.z());
```

La modification est relativement facile. Maintenant imaginons que cette classe Point3D est utilisée dans des centaines de ligne de code. Imaginons aussi que toutes vos classes présentent le même problème d'exposer des détails d'implémentation interne. À chaque fois que l'on doit modifier une classe, on se retrouve avec plein de bugs à la compilation et on doit perdre du temps à corriger toutes les lignes de code utilisant notre classe. C'est un problème que connaissent beaucoup de débutants (et d'autres développeurs plus anciens... mais aiment faire des erreurs de débutants). Quelle perte de temps ! On parle de couplage fort entre deux classes quand la modification de l'une de classe implique la modification de l'autre classe. Sinon, on parle de couplage faible.

En utilisant les templates, on améliore un peu la situation. On laisse la responsabilité de choisir le type utilisé en interne à l'utilisateur de notre classe. Il sait donc quelle fonction appeler selon le contexte :

```
template<class T = float>
class Point3D {
    T x, y, z;
public:
    inline T x() const { return x; }
    inline T y() const { return y; }
    inline T z() const { return z; }

    inline void setX(T v) { x = v; }
    inline void setY(T v) { y = v; }
    inline void setZ(T v) { z = v; }
};

Point3D p { 1.0, 1.0, 1.0 };
glVertex3f(p.x(), p.y(), p.z());
Point3D<double> p2 { 1.0, 1.0, 1.0 };
```

```
glVertex3d(p2.x(), p2.y(), p2.z());
<code>

Même si le code est meilleur que le précédent, il est encore
améliorable. L'idéal serait de ne plus avoir à choisir
manuellement la fonction à appeler et laisser le compilateur
faire le travail pour nous.

===== Comment corriger ce problème ? =====

Le premier principe qui n'est pas respecté dans ce cas est
le principe de ségrégation des interfaces. Ce principe dit «
Une classe ou une fonction cliente ne doit pas dépendre
d'interfaces dont elle n'a pas l'utilité » (source). Dit
autrement, cela veut dire que si une classe A utilise une
classe B et que cette classe B utilise une classe C, A n'a
pas à connaître C. Dans notre exemple, il faudrait que le
code qui utilise la classe Point3D n'a pas à connaître le
type utilisé en interne (float ou double).

Mais le problème est plus profond (et critique) que cela. Le
problème vient en fait d'une mauvaise compréhension de ce
qu'est l'encapsulation. La règle est la suivante : « On
encapsule un comportement, pas des propriétés » (source).
Voyons ce que cela implique en pratique pour notre classe
représentant un point en 3D. Si on pense en termes de
propriétés, comme on l'a fait au début, un point dans un
espace 3D est effectivement un objet représenté par ses
trois composantes réelles x, y et z. Si on pense en terme de
comportement, l'implémentation sera différente. Quels sont
les comportements attendus pour notre point ? En suivant
notre code d'exemple précédant, le seul comportement que
l'on souhaite implémenter est de pouvoir l'afficher. On
écrit donc simplement le code suivant :

<code cpp>
class Point3D {
    float x, y, z;
public:
    void draw() const { glVertex3f(x, y, z); }
};
```

```
Point3D p { 1.0f, 1.0f, 1.0f };
p.draw();
```

La différence par rapport au code précédant est ridicule en terme de travail à fournir pour l'implémentation. On a simplement refactorisé l'appel à glVertex dans une fonction membre de Point3D. Par contre, en terme de sémantique, la différence est énorme : le code client n'a plus besoin de connaître les détails d'implémentation, notre code est plus facilement évolutif et donc efficace. Les modifications à apporter à notre code en cas de changement est localisé : on sait que si l'on modifie une variable membre d'une classe, on n'a que les fonctions membres de la classe à modifier et rien d'autre.

Si on est paresseux (et donc intelligent), on va utiliser la version template pour plus de souplesse, par exemple avec des spécialisations :

```
template<class T = float>
class Point3D {
    T x, y, z;
public:
    void draw() const;
};

template<>
void Point3D<float>::draw() const { glVertex3f(x, y, z); }

template<>
void Point3D<double>::draw() const { glVertex3d(x, y, z); }

Point3D p { 1.0, 1.0, 1.0 };
p.draw();
Point3D<double> p2 { 1.0, 1.0, 1.0 };
p2.draw();
```

La version template demande un peu plus de ligne de code que la version non template et peu donc demander un peu plus de travail pour le développeur. Pour autant, elle est préférable puisqu'il ne sera plus nécessaire de modifier le code en fonction des besoins du code client (respect du principe ouvert-fermé). Si on a plusieurs fonctions qui dépendent du type utilisé en interne, on peut également utiliser une

classe de traits et polices :

```
template<class T = float>
struct gl_trait {
    typdef T internal;
    // static inline glVertex  (internal x, internal y,
internal z) const {}
    // static inline glNormal  (internal x, internal y,
internal z) const {}
    // static inline glTexCoord(internal x, internal y,
internal z) const {}
};

<code>template<>
struct gl_trait<float> {
    static inline glVertex  (internal x, internal y,
internal z) const { glVertex3f(x, y, z); }
    static inline glNormal  (internal x, internal y,
internal z) const { glNormal3f(x, y, z); }
    static inline glTexCoord(internal x, internal y,
internal z) const { glTexCoord3f(x, y, z); }
};

<code>template<>
struct gl_trait<double> {
    static inline glVertex  (internal x, internal y,
internal z) const { glVertex3d(x, y, z); }
    static inline glNormal  (internal x, internal y,
internal z) const { glNormal3d(x, y, z); }
    static inline glTexCoord(internal x, internal y,
internal z) const { glTexCoord3d(x, y, z); }
};

template<class T>
class Point3D {
    gl_trait<T>::internal x, y, z;
public:
    void draw() const { gl_trait<T>::glVertex(x, y, z); }
};

Point3D p { 1.0, 1.0, 1.0 };
```

```
p.draw();
Point3D<double> p2 { 1.0, 1.0, 1.0 };
p2.draw();
```

Avec ce code, la liste des fonctions à appeler en fonction du type utilisé en interne est localisé dans une même classe de traits. Et si on souhaite ajouter un nouveau type, il suffit d'ajouter une nouvelle spécialisation pour la classe de traits, sans rien modifier au code existant.

Pour terminer, un peu de lecture

Le respect de ces principes est une méthode pour éviter les couplages trop forts entre les classes. Il existe d'autres méthodes pour découpler des classes (ie diminuer la force du couplage). Tout le monde connait par exemple la séparation du code des classes dans un fichier d'en-tête (partie la moins susceptible d'être modifiée) et un fichier d'implémentation (partie plus facilement modifiable). On peut également citer l'idiome Pimpl (Pointer To Implementation) ou l'utilisation des signaux et slots, deux techniques très utilisée dans Qt.

En complément, Emmanuel Deloget a publié quelques articles intéressants sur les principes de programmation objet :

- Valider et corriger une architecture objet première partie et seconde partie par Emmanuel Deloget ;
- Le principe d'encapsulation, le principe de ségrégation des interfaces et le principe ouvert-fermé par Emmanuel Deloget ;
- La guerre des accesseurs par Emmanuel Deloget ;
- Présentation des classes de Traits et de Politiques en C++ par Alp Mestan ;
- L'idome pImpl (WikiBooks) ;
- Boost.Signals (documentation de Boost) ;
- Les signaux et slots dans Qt (documentation de Qt 4.8).

C++