Couplage entre classes

L'idée de base de la programmation objet est de créer des objets. Quand on fait cela, c'est déjà un bon début. Cependant, ce n'est généralement pas suffisant. Il faut également que les objets communiquent entre eux.

Lorsque qu'un objet-enfant est créé dans un objet-parent, il n'y a pas de problème pour que le parent puisse communiquer avec l'enfant. Comme le parent a créé l'enfant, il le connait bien. En particulier, il connait les fonctions publiques de l'objet-enfant et peut donc les appeler directement.

```
class Enfant {
public:
    void une_fonction() const {}
};

class Parent {
    Enfant mon_enfant;
public:
    void autre_fonction() const {
        mon_enfant.une_fonction(); // Ok
    }
};
```

Lorsque deux classes communiquent entre elles, on dit qu'elles sont couplées. Ici, le couplage est dit fort, puisque la classe `Parent` doit absolument connaitre la classe `Enfant`. Si ces deux classes sont déclarées dans des fichiers différents (`parent.h` et `enfant.h` par exemple), il sera nécessaire d'inclure le fichier `enfant.h` dans le fichier `parent.h`.

```
// dans parent.h
#include "enfant.h"
```

Ce couplage n'est pas symétrique. En effet, la classe `Enfant` ne connait pas (et n'a pas besoin de connaitre) la classe `Parent`. En revanche, si on souhaite appeler une fonction de `Parent` dans `Enfant`, on est face à deux difficultés.

Premièrement, il faut appeler la fonction sur un objet en particulier.

Comme l'objet `Enfant` est déjà un membre de la classe `Parent`, il ne serait pas possible de créer un nouvel objet `Parent` dans `Enfant`, puisque dans cette situation, l'objet Parent contiendrait un objet `Enfant`, qui contient à son tour un autre objet `Parent`, lui-même contenant un objet `Enfant` et ainsi de suite. On se retrouve dans une boucle infinie, cela n'a pas de sens.

```
class Enfant {
    Parent mon_parent;
};
```

Cette situation est résolue en ne créant pas un objet de type `Parent` dans la classe `Enfant`, mais créant simplement un “lien” vers un objet de type `Parent` existant (par exemple, l'objet de type `Parent` qui a créé l'objet de type `Enfant` comme variable membre). Ce type de “lien” est réalisé en C++ en utilisant une référence ou un pointeur. Il sera alors possible d'appeler une fonction de l'objet de type `Parent` dans l'objet de type `Enfant`.

```
// dans enfant.h
class Enfant {
    Parent const& mon_parent; // référence
public:
    void une_fonction() const {
        mon_parent.encore_une_fonction();
    }
};

// dans parent.h
#include "enfant.h"
class Parent {
    Enfant mon_enfant;
public:
    Parent() : mon_enfant(*this) { // beurk
    }

    void autre_fonction() const {
        mon_enfant.une_fonction(); // Ok
    }
```

```
    void encore_une_fonction() const {
    }
};
```

Si vous essayez de compiler ce code, vous obtiendrez une erreur dans la classe `Enfant` : “le type Parent n'est pas connu”. La raison est que le compilateur ne connait pas encore la classe `Parent`, il ne peut pas savoir à quoi correspond le terme “Parent” dans votre code. La solution est alors d'indiquer au compilateur que ce terme existe et qu'il correspond à une classe. Dans cette situation, on parle de “déclaration anticipée” de la classe `Parent`.

```
// dans enfant.h
class Parent; // on indique au compilateur que le terme
"Parent" existe et que c'est une classe
class Enfant {
    Parent const& mon_parent; // référence
public:
    Enfant(Parent const& p) : mon_parent(p) {
    }

    void une_fonction() const {
        mon_parent.encore_une_fonction();
    }
};

// dans parent.h
#include "enfant.h"
class Parent {
    Enfant mon_enfant;
public:
    Parent() : mon_enfant(*this) { // beurk
    }

    void autre_fonction() const {
        mon_enfant.une_fonction(); // Ok
    }

    void encore_une_fonction() const {
    }
```

```
};
```

Cependant, cela ne fonctionne toujours pas. En effet, trois lignes en dessous, on essaie d'appeler la fonction `encore_une_fonction` de la classe `Parent`. Or, pour le moment, le compilateur sait simplement qu'il existe une classe `Parent`, il ne sait pas encore ce qu'elle contient. Pour résoudre ce second problème, on va donc séparer la déclaration des classes de leur implémentation en les mettant dans des fichiers séparés (par exemple `enfant.cpp` et `parent.cpp`) :

```
// dans enfant.h
class Parent; // déclaration anticipée
class Enfant {
    Parent const& mon_parent; // référence
public:
    Enfant(Parent const& p);
    void une_fonction() const;
};

// dans enfant.cpp
#include "enfant.h"
#include "parent.h"
Enfant::Enfant(Parent const& p) : mon_parent(p) {
}
void Enfant::une_fonction() const {
    mon_parent.encore_une_fonction();
}

// dans parent.h
#include "enfant.h"
class Parent {
    Enfant mon_enfant;
public:
    Parent() : mon_enfant(*this);
    void autre_fonction() const;
    void encore_une_fonction() const;
};

// dans parent.cpp
#include "parent.h"
#include "enfant.h"
```

```
Parent::Parent() : mon_enfant(*this) { // beurk
}
void Parent::autre_fonction() const {
    mon_enfant.une_fonction(); // Ok
}
void Parent::encore_une_fonction() const {
}
```

Ce type de code est très classique en C++, on le retrouve partout. Mais cela pose un problème majeur : ce code n'est pas évolutif. Les classes et les fonctions appelées sont figées, si on souhaite appeler une autre fonction ou une autre classe, il faudra modifier le code existant de ces classes et ajouter toutes les déclarations de classes et de fonctions (imaginez le résultat sur un projet contenant plusieurs centaines de classes...) Il est possible d'améliorer la situation en mettant les fonctions et les objets comme paramètres des classes et fonctions (utilisation des templates, fonctions callback, etc), mais cela donne un code relativement lourd à maintenir et à faire évoluer.

Qt implémente également le système parent-enfant, avec les fonctions suivantes. Il est en particulier utilisé pour les destructions automatiques des hiérarchies d'objets. Voir : Object Trees & Ownership.

```
QObject * QObject::parent() const;
void setParent(QObject * parent);

const QObjectList & QObject::children() const;
T QObject::findChild(const QString & name = QString(),
    Qt::FindChildOptions options = Qt::
FindChildrenRecursively) const;
QList<T> QObject::findChildren(const QString & name =
QString(),
    Qt::FindChildOptions options = Qt::
FindChildrenRecursively) const
```