Refactoring : C++

TP Refactoring/C++

Eric Ramat
ramat@lil.univ-littoral.fr

5 mars 2007

Durée : 9 heures

---

1 Objectif

L'objectif de ce TP est de restructurer un code existant. Cette restructuration a pour objectif de tendre vers un standardisation du code :
- du *coding style* et la documentation automatique,
- de l'utilisation des *design patterns*,
- des contrôles des contraintes (assertions),
- de l'utilisation systématiquement de bibliothèques de haut niveau (Boost et STL),
- l'encapsulation des pointeurs,
- ...

Cette démarche doit nous mener aux objectifs suivants :
- diminution du code (nombre total de lignes/instructions et nombre moyen de lignes/instructions par fonction);
- augmentation de la lisibilité du code (respect du *coding style* et documentation de chaque méthode publique);
- augmentation de la qualité de l'application ;
- réduction des bugs mémoires (zéro blocs perdus - valgrind);

Pour vérifier que les objectifs sont bien atteints un certain nombre de mesures vont évaluer votre code.

L'outil utilisé est CCCC (A code counter for C and C++). Son utilisation est la suivante : `cccc -lang=c++ -outdir=. *.cc *.hh` CCCC offre les mesures suivantes par fichier, par classe et/ou par projet :
- LOC (Lines of Code) : nombre de lignes de code (sans les commentaires et les lignes vides) ;
- MVG (McCabe's Cyclomatic Complexity) : nombre d'imbrications de boucles et nombre de chemins dans le flux de contrôle ; métrique applicable uniquement sur les fonctions ;
- COM (Comment Lines) : nombre de lignes de commentaires ;
- L_C (LOC/COM) : rapport entre le nombre de lignes de code et le nombre de lignes de commentaires ;
- M_C (MVG/COM) : rapport entre le MVG et le nombre de lignes de commentaires ;
- FO (Fan-out) : nombre de modules utilisés par le module considéré ;
- FI (Fan-in) : nombre de modules utilisant le module considéré ;

– IF4 (Information flow complexity ou HK - métrique d'Henry-Kafura/Shepperd) : mesure du flux d'information entre modules ; c'est une évaluation approximative du nombre de couplage inter-module (un module = un fichier) ; c'est la racine carrée du produit FO par FI ; il existe deux variantes : IF4v qui ne tient compte que des éléments accessibles à l'extérieur et IF4c qui mesure l'implication d'un changement dans l'interface fonctionnelle ;
– WMC (Weighted methods per class) : le nombre de méthodes (si le poids de chaque méthode est fixée à 1) ;
– DIT (Depth of inheritance tree) : profondeur de l'arbre d'héritage par chaque classe ;
– NOC (Number of children) : nombre de sous-classes de la classe ;
– CBO (Coupling between objects) : nombre de modules (ou fichiers ou classes si une classe par fichier) couplés en tant que client ou fournisseur à la classe considérée ;

Toutes ces mesures sont visualisables dans le fichier cccc.html qui a été généré dans le répertoire des sources C++.

Voici les valeurs limites pour chacune des métriques :
– LOC/fonction = 30
– LOC/module = 500
– MVG/fonction = 10
– MVG/module = 100
– L_C = 7
– M_C = 5
– FO = 12
– FI = 12
– IF4 = 100 ; IF4c = 6 ; IF4v = 6
– WMC = 30
– DIT = 3
– NOC = 4
– CBO = 12

NB : le fichier d'options est générable par la commande suivante : `cccc -lang=c++ -opt_outfile=opt`
Ce fichier d'options permet entre autre de fixer les seuils pour les métriques.

2 Analyse du code

Vous avez de la chance : le code vous le connaissait déjà puisqu'il s'agit du projet en C++ de l'année dernière. Le code qui est proposé est celui d'un binôme de l'année dernière (devinez lequel !).

Question 1. Réaliser le *reverse engineering* du code source en établissant le diagramme UML de l'application actuelle.

Question 2. En travaillant à partir du modèle, pouvez-vous identifier les *design patterns* qui ont été utilisés ?

3 Nettoyage du code

Question 3. Supprimer tous les commentaires dans les .cc et toutes les méthodes liées à de l'affichage en console. Les commentaires doivent être dans les .hh (voir Normalisation du code) et les affichages

doivent être remplacés par le contrôle des contraintes.

4 Restructuration des classes

Question 4. Identifier les *patterns* que l'on pourrait appliquer. On pourra aussi se réfèrer au site Web « *Refactoring To Patterns Catalog* » pour identifier la restructuration en terme de pattern que l'on peut appliquer au code.

5 Utilisation de Boost et de la STL

Question 5. Identifier les structures et algorithmes où Boost peut être utilisé ainsi que la STL. Par exemple, il ne doit plus y avoir de boucles mais seulement des *iterators* sur des structures. Autre exemple, les structures du type `vector < vector < X* > >` ne doivent plus exister. Jeter un coup d'oeil sur la classe `MultiMap` de la STL.

On en profitera pour encapsuler tous les pointeurs dans des `smart_pointers` de Boost.

6 Normalisation du code

Question 6. Vérifier que le *coding style* défini ci-dessous est respecté.

Le *coding style* employé est inspiré de celui du projet Linux de Linus Thorsvald. La plupart des questions engendrées par la lecture des paragraphes suivants trouvent leurs réponses dans ce codingStyle disponible sur le site http://www.kernel.org.

6.1 Format d'une classe

Dans une classe, nous définissons dans l'ordre les parties d'une classe : les définitions, les fonctions et les attributs avec comme priorité : *public*, *protected* et *private*. Cet ordre est défini dans ce sens car elle montre à l'utilisateur de votre classe les fonctions utiles en premier puisqu'il ne peut utiliser que les données et méthodes publiques et protégées. Par exemple :

```
#define CLASS_HPP
#define CLASS_HPP

class Class : public, protected, private class1, class2 etc.
{
public:
    typedefs ...
    variables ...   // les attributs publics sont très, très rares...
                    // Ne les utilisez jamais.
    functions ...

protected:
    typedefs ...
    variables ...
    functions ...
```

```
private:
    typedefs ...
    variables ...
    functions ...
};

#endif
```

Toutes les classes ou fonctions que vous déclarez doivent faire parti d'un espace de nom. Ne polluez jamais l'espace de nom avec des noms de classes ou de fonctions globales. Une manière simple et couramment employée dans les projets C++ est d'utiliser la règle : « un répertoire = un namespace ».

```
namespace gui { namespace plugins { namespace generator {

    class GeneratorPlugin : public Plugin
    {
        ...
    };

}}} // namespace gui plugins generator
```

6.2 Règles de nommage

– dossier : Les noms de répertoires sont en minuscules afin de respecter le nom des namespaces.

```
mkdir -p gui/plugins
```

– namespace : Tout en minuscule, comme le noms des dossiers.

```
namespace gui { namespace plugins {
    ...
}}
```

– classe : Les premières lettres de chaque nom en majuscule.

```
class Position
{
    ...
};
```

– méthode : La première lettre des noms de méthodes commence par une minuscule.

```
void getPosition(int& x, int& y) {
    ...
}
```

– attribut d'une classe : Commence par la lettre minuscule "'m"'.

```
double mX;
```

6.3 Règles de mise en page

6.3.1 Tabulations et espaces

Une fonction ne peut dépasser les limites de 50 lignes et 80 colonnes pour permettre une meilleure lecture du code. De même, les indentations se font sur 4 caractères alors que la taille d'une tabulation est

de 8 caractères. Utilisez toujours une tabulation à la place des espaces afin de donner aux lecteurs la possibilité d'adapter la tabulation à leurs préférences. Enfin, une fonction ne peut contenir plus de six variables locales, et une fonction ou un membre d'une classe ne peut contenir plus de six paramètres locaux.

Dans le même ordre d'idée :
– sur un ligne se trouve une seule instruction :

```
if (condition)
   fait_quelque_chose();
```

– Pas d'espace entre un nom de fonction et la parenthèse :

```
Mauvais :   func (arg);
Bon :       func(arg);
```

– Entrez un espace après if, while, switch, etc. :

```
Mauvais :   if(arg)     for(;;)
Bon :       if (arg)    for (;;)
```

– Entrez un espace après une virgule et un point-virgule :

```
Mauvais :   func(arg1,arg2);    for (i = 0;i < 2;++i)
Bon :       func(arg1, arg2);   for (i = 0; i < 2; ++i)
```

– Entrez un espace avant et après $=$, $+$, $/$, etc. :

```
Mauvais :   var=a*5;
Bon :       var = a * 5;
```

– Pas d'espace entre le type de la variable et les cractères référence (&) et pointeur (*):

```
Mauvais :   int * a;
Bon :       int* a;
```

– Pas d'espace entre les opérateurs d'adresses et les variables :

```
Mauvais :   int b = * a;
Bon :       int b = *a;
```

6.3.2 Placement des accolades

La politique de placement des accolades est celle définit par Kernighan et Ritchie c'est à dire, une accolade se trouve sur l'instruction de la ligne sauf dans le cas d'une définition de fonction. Les exemples suivants montrent les différents cas de placement :

```
if (x == y) {
    ...
}

int fonction(int x)
{
    ...
}

do {
    ...
} while (condition);
```

```
if (x == y) {
    ...
} else if (x > y) {
    ...
} else {
    ...
}

switch (test.getX()) {
case 0:
    ...
    break;
case 1:
    {
        ...
    }
default:
    ...
}
```

6.4 Commentaires

Les commentaires font partis intégrante d'un programme. Toutes les fonctions doivent contenir des commentaires décrivant le rôle de la fonction et non son fonctionnement. Aucun commentaire ne doit venir s'insérer dans le code de la fonction.

Le programme *doxygen*, à partir des sources commentées et de balises spéciales, réalise une documentation complète.

```
/**
 * Description du rôle de la fonction ...
 * ...
 * @param n1 description du paramètre 1 très longue description, bla bla
 * blablabla bla bla blabla.
 * @param n2 description du paramètre 2
 * @throw indique les exceptions pouvant être levées par la fonction.
 * @return description du retour de la fonction.
 */
int max(int n1, int n2)
{
    ...
}
```

Les fichiers du projet d'extension ".cc" et ".hh" possèdent une entête identique définissant la licence (de type GNU : GPL) et les informations sur les auteurs :

```
/**
 * @file filename.extension
 * @brief description du fichier
 * @author The I2L Development Team
 * @date lun, 23 jan 2006 14:00:40 +0100
 */

/*
 * Copyright (C) 2004, 05, 06 - The I2L Development Team
```

```
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
 */
```

7 Contrôle des contraintes

Question 7. Intégrer les contrôles des contraintes par les assertions.

Une méthode très simple pour détecter les erreurs dans un programme est d'utiliser les fonctions d'assertions définies dans le fichier `Debug.hpp`. Par exemple :

```
int getX(const Point* p)
{
    Assert(Exception::Internal, p != NULL, "p == NULL ?");
    return p->x;
}
```

Trois types d'assertions sont définis :
- `Assert(except, test, msg)` : Pour reporter une exception « except » si le « test » échoue, l'exception est initialisée avec le message « msg » ;
- `AssertS(except, test)` : Pour reporter une exception « except » si le « test » échoue ;
- `AssertI(test)` : Pour reporter une exception `Exception::Internal` : si le « test » échoue.

Le code de ces assertions est disponible : http://vle.univ-littoral.fr/gitweb?p=vle.git;a=blob;f=src/vle/utils/Debug.hpp;h=a20c24ad27ab7bf12ed6ca50898bd63c878d23e4;hb=HEAD

8 Références

- Boost : http://www.boost.org
- STL : http://www.sgi.com/tech/stl/
- Refactoring : « Refactoring To Patterns Catalog » - http://www.industriallogic.com/xp/refactoring/catalog.html
- Valgrind : http://valgrind.org/
- Doxygen : http://www.stack.nl/~dimitri/doxygen/
- CCCC : http://cccc.sourceforge.net/ ; la documentation est dans /usr/share/doc/cccc