


Utiliser le pattern Registry

par Matthieu Brucher (<http://matthieu-brucher.developpez.com/>) (Blog)

Date de publication : 15/12/2006

Dernière mise à jour :

Le pattern Registry est un  **pattern** méconnu mais largement utilisé dans de nombreux cas. Il est utilisé pour gérer les paramètres d'un programme - exemple typique, la base de registres de Windows est un registre particulier -, les tests unitaires sont enregistrés dans un registre dans certaines implémentation - comme celles de JUnit -, ou plus simplement des menus.

Introduction

I - L'architecture du pattern




I-A - Descriptif des nécessités des classes

I-B - Proposition de solution

II - Exemple d'implémentation en C++

Conclusion

Introduction

Lorsque l'on désire enregistrer des  **objets** dans une structure et pouvoir les récupérer de manière transparente, on utilise d'habitude une carte associative. Dans certains cas, on veut plus, à savoir protéger les accès, utiliser une hiérarchie, ... Dans ces cas, on va utiliser un registre pour stocker les données. Traditionnellement un singleton, on peut aussi utiliser plusieurs registres répartis dans plusieurs  **instances** de  **classes** englobantes, mais cela est plus rare.

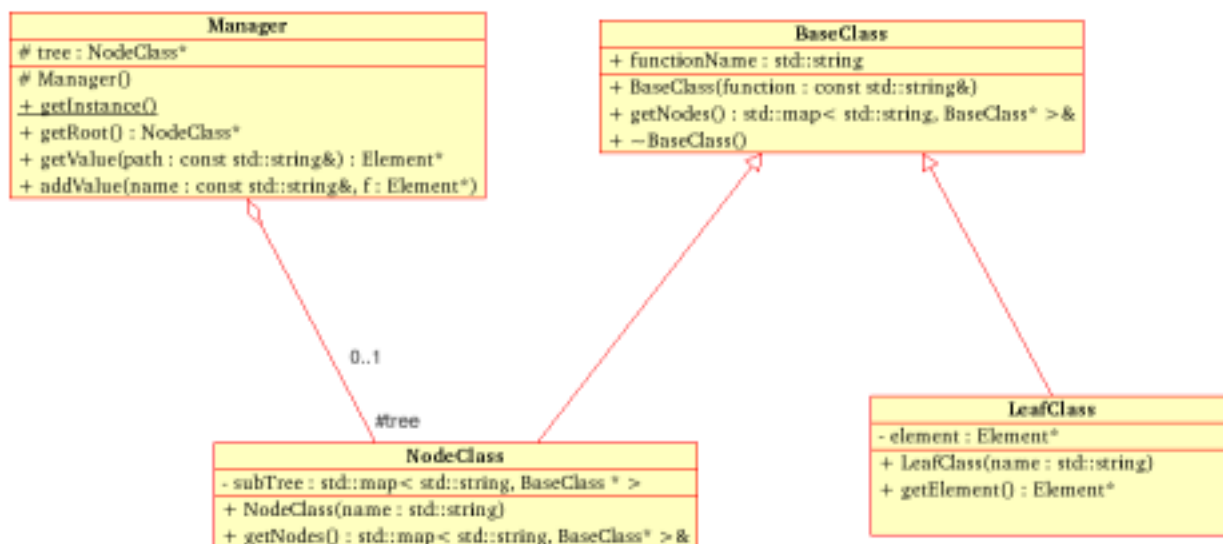
I - L'architecture du pattern

I-A - Descriptif des nécessités des classes

Selon les nécessités, on peut avoir besoin de plusieurs classes différentes. Dans sa version la plus simple, le registre ne fait qu'encapsuler une carte associative.

Dans une version plus complète, une hiérarchie complète que le registre gère est générée. Il faudra à ce moment, outre la classe Registre elle-même, une hiérarchie de classes, une classe de base virtuelle pure qui sera la base de la hiérarchie, une classe fille qui symbolisera les feuilles de l'arbre, et une autre classe fille qui permettra de gérer les noeuds de l'arbre.

I-B - Proposition de solution



Exemple de hiérarchie des classes

La version qu'on va présenter est une version hiérarchique.

Lors de l'ajout d'une nouvelle valeur, de nouveaux noeuds seront créés si besoin est, et un noeud feuille sera créé. Naturellement, s'il faut pouvoir modifier les valeurs entrées dans le registre, il faudra ajouter de nouvelles fonctions pour cela. Ici, nous ferons un simple registre, sans ces fonctions - qui ne sont pas utiles pour un menu ou pour un logiciel de tests -.

Tout d'abord la classe Manager possède une méthode statique pour récupérer l'instance du registre. Ensuite, elle dispose de deux méthodes permettant d'enregistrer une instance d'une classe avec une valeur associée - ici, nous prendrons une chaîne de caractères - et une méthode retournant l'instance stockée lorsqu'on lui présente la chaîne de caractères associée.

La classe de base de la hiérarchie **BaseClass** n'a pas besoin de contenir grand chose. Elle doit contenir un nom qui est le nom du noeud - utile pour parcourir la hiérarchie à la recherche d'un élément -, une fonction retournant les noeuds fils - s'il y en a -. Dans la classe fille **NodeClass** qui est responsable de la hiérarchie, on surcharge l'opérateur retournant les noeuds fils pour parcourir l'arbre. Ensuite, la classe fille **LeafClass** est celle qui contiendra l'instance. Pourquoi utiliser 3 classes alors qu'une seule suffirait ? La raison principale est qu'une seule classe devrait gérer

l'arbre et la valeur, ce qui est contraire à un principe bien connu, une classe une responsabilité. Ensuite, le cas ici est très simple, le registre élémentaire. Si celui-ci doit contenir des méthodes supplémentaires, elles seront peut-être différentes selon qu'on soit dans un noeud de l'arbre ou à une feuille.

II - Exemple d'implémentation en C++

On se propose d'utiliser un singleton simple pour la classe Manager, comme pour le précédent article [sur le pattern object Pool](#). Ce singleton permettra d'enregistrer une instance d'un objet associée à une valeur, ou de récupérer l'instance associée à une valeur. On pourra aussi ajouter d'autres méthodes si besoin est.

Tout comme le précédent tutoriel sur le pattern Object Pool, on laissera au lecteur la compréhension des différentes méthodes utilisées.

```
/**
 * \file registry.h
 * Déclaration des classes gérant le registre qui stocke des pointeurs vers des Element
 */
#ifdef REGISTRY
#define REGISTRY
#include <string>
#include <map>

class Element;

// Classe de base des éléments enregistrés dans le registre
struct BaseClass
{
    /**
     * Constructeur de base de la classe
     * @param nom est le nom du noeud courant
     */
    BaseClass(const std::string& nom);
    /**
     * Retourne la liste des noeuds fils du noeud courant
     * @return une référence sur la carte des fils
     * \warning lance une exception dans le cas de base
     */
    virtual std::map<std::string, BaseClass*> getNodes();
    /**
     * Retourne un pointeur vers l'élément stocké dans le noeud courant
     * @return un pointeur vers l'élément stocké
     * \warning lance une exception dans le cas de base
     */
    virtual Element* getElement();
    /**
     * Destructeur par défaut
     */
    virtual ~BaseClass(){}
private:
    /// Le nom du noeud
    std::string nom;
};

// Classe composant un noeud avec enfants sans élément
struct NodeClass : public BaseClass
{
    /**
     * Constructeur de base de la classe
     * @param nom est le nom du noeud courant
     */
    NodeClass(const std::string& nom);
    /**
     * Retourne la liste des noeuds fils du noeud courant
     * @return une référence sur la carte des fils
     */
    virtual std::map<std::string, BaseClass*> getNodes();
    /**
```

```
* Destructeur par défaut
* \warning devrait être complété pour supprimer les noeuds fils
*/
~NodeClass(){}
private:
    /// Carte des noeuds fils
    std::map<std::string, BaseClass*> subTree;
};

/// Classe feuille de l'arbre
struct LeafClass : public BaseClass
{
    /**
     * Constructeur de base de la classe
     * @param nom est le nom du noeud courant
     * @param element est le pointeur à stocker dans la feuille
     */
    LeafClass(const std::string& nom, Element* element);
    /**
     * Retourne un pointeur vers l'élément stocké dans le noeud courant
     * @return un pointeur vers l'élément stocké
     */
    virtual Element* getElement();
    /**
     * Destructeur par défaut
     */
    ~LeafClass(){}
private:
    /// Pointeur vers l'élément fils
};

/// La classe gérant le registre
struct Manager
{
    /**
     * Retourne l'instance du manager et la crée au besoin
     * @return l'instance du manager
     */
    Manager& getInstance();
    /**
     * Ajoute un élément dans le registre avec un nom associé. Les différentes parties du nom
     hiérarchique sont séparées par des ':'
     * @param nom est le nom associé au nouvel élément
     * @param element est un pointeur vers l'élément à ajouter
     * \warning attention, si vous ajoutez un élément avec un nom existant, il faudra ajouter une
     vérification
     */
    void addElement(std::string nom, Element* element);
    /**
     * Retourne un élément associé à un nom donné en paramètre
     * @param nom est le nom de l'élément à retrouver
     * @return un pointeur vers l'élément recherché
     */
    Element* getElement(const std::string& nom) const;
private:
    /// Constructeur par défaut
    Manager();
    /// Constructeur par copie interdit
    Manager(const Manager&);

    /// Pointeur vers l'arbre des éléments
    NodeClass* tree;
};

#endif
```

```
#include "registry.h"
#include <list>
#include <stdexcept>

/**
 * Découpe une chaîne en morceau avec ':' comme séparateur
 * On peut changer facilement en ajoutant le séparateur comme paramètre à la fonction
 * @param nom est la chaîne à découper
 * @return la liste des sous-chaînes
 */
std::list<std::string> split(const std::string& nom)
{
    std::list<std::string> liste;
    size_t start = 0;
    for(size_t end = 0; end < nom.size(); ++end)
    {
        if(nom[end] == ':')
        {
            liste.push_back(nom.substr(start, start - end - 1));
            start = end + 1;
        }
    }
    liste.push_back(nom.substr(start, start - end - 1));
    return liste;
}

BaseClass::BaseClass(const std::string& nom)
:nom(nom)
{
}

std::map<std::string, BaseClass*>& BaseClass::getNodes()
{
    throw std::runtime_error("Pas de noeuds fils");
}

Element* BaseClass::getElement()
{
    throw std::runtime_error("Pas d'élément");
}

NodeClass::NodeClass(const std::string& nom)
:BaseClass(nom), subTree()
{
}

std::map<std::string, BaseClass*>& NodeClass::getNodes()
{
    return subTree;
}

LeafClass::LeafClass(const std::string& nom, Element* element)
:BaseClass(nom), element(element)
{
}

Element* LeafClass::getElement()
{
    return element;
}

Manager::getInstance()
{
    static Manager manager;
    return manager;
}

Manager::Manager()
```

```
:tree("Root")
{
}

void Manager::addElement(std::string nom, Element* element)
{
    std::list<std::string> liste = split(nom);
    NodeClass* node = getRoot();

    // On récupère le dernier élément de la liste
    std::list<std::string>::iterator endList = liste.end();
    --endList;

    // On travaille sur chaque élément de la liste sauf le dernier pour construire la hiérarchie
    for(std::list<std::string>::iterator sousNom = liste.begin(); sousNom != endList; ++sousNom)
    {
        if(node->getNodes().find(*sousNom) == node->getNodes().end())
        {
            node->getNodes()[*sousNom] = new NodeClass(*sousNom);
        }
        std::map<std::string, BaseClass*>::iterator element = node->getNodes().find(*sousNom);
        node = static_cast<NodeClass*>(element.second);
    }
    // On crée la feuille de l'arbre
    node->getNodes()[*endList] = new LeafClass(*endList);
}
```

Conclusion

Ce pattern est une base de travail. En général, on veut plus qu'enregistrer ou récupérer une valeur, on peut pouvoir récupérer un sous-arbre, toutes les instances du sous-arbre, modifier l'instance, ... Pour cela, on utilisera d'autres patterns pour aller plus loin.

