Utiliser le pattern Object Pool

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

Date de publication : 18/09/2006

Dernière mise à jour :

Le **pattern Object Pool** sert à gérer une quantité finie d'**objets** de même type utilisés couramment, sans qu'on veuille permettre de construction ou de destruction lors de l'utilisation d'une instance de ces objets. On peut aussi empêcher de cette manière la création de "trop" d'instances.

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

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Introduction

Le pattern **Object Pool** est un pattern permettant de stocker une quantité finie d'**instances** d'une **classe** afin de les distribuer à qui en a besoin et qui sont rendus en fin d'utilisation. Souvent, on les utilise pour gérer des threads, des connexions distantes, ... Après présentation de l'architecture, un exemple sommaire d'**implémentation** en C++ sera proposé.

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

I - L'architecture du pattern

I-A - Descriptif des nécessités des classes

Plusieurs instances d'une classe doivent être distribuées à des fonctions appelantes. Il faut donc une classe responsable de cette gestion, et ce sera donc un singleton qui en sera responsable. Celui-ci sera chargé de créer les instances des objets, de les distribuer et de les récupérer.

Les instances de la classe à partager n'ont pas d'impératif particulier.

I-B - Proposition de solution

La classe **Manager** sera la classe singleton. On voit qu'il y a une méthode statique pour récupérer le singleton, une fonction pour récupérer une nouvelle instance, une pour la rendre ainsi qu'une fonction pour rendre les instances dont le comportement n'est plus adéquat - par exemple une connexion qui se serait arrêtée de manière impromptue ou un objet corrompu -.

Un client récupère l'instance globale du manager et demande une instance d'**Objet** à utiliser. Après avoir utilisé cette instance, le client indique au manager qu'il peut redonner à un autre client l'instance donnée en paramètre. Si jamais l'objet est corrompu, on peut l'indiquer au manager en le redonnant par la fonction **releaseBadObject**.

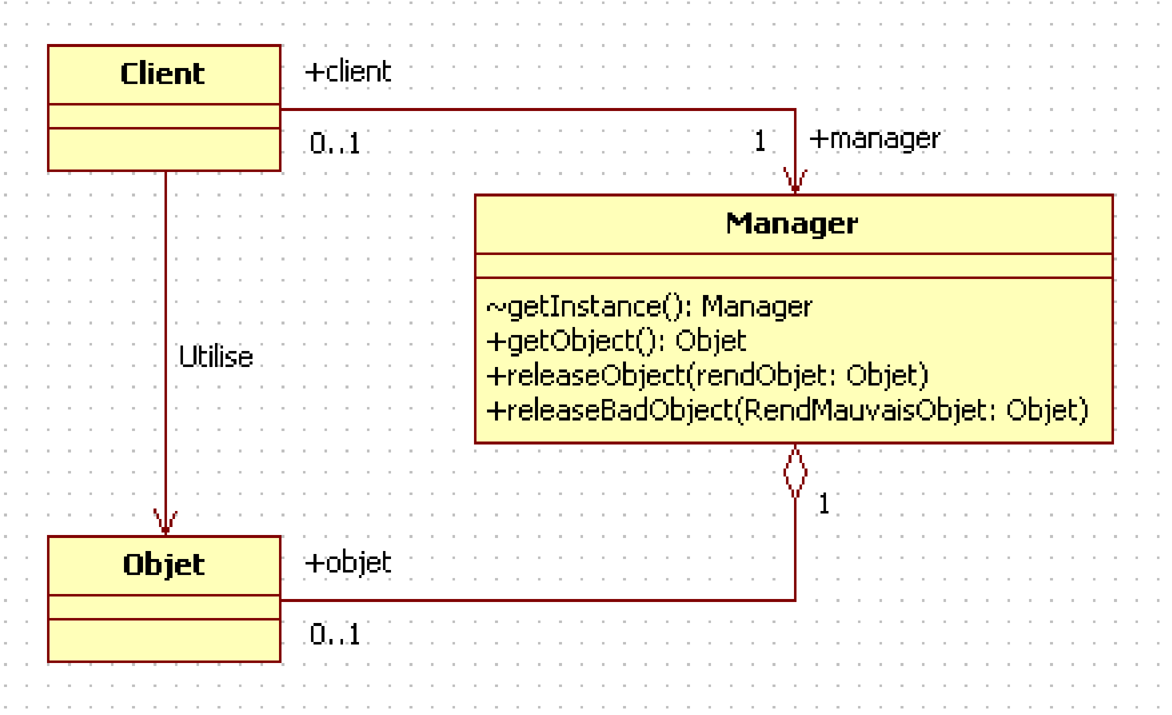

Diagramme UML du pattern Object Pool

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

II - Exemple d'implémentation en C++

Il existe plusieurs implémentations du pattern singleton. Ici, on choisira la plus simple, on laissera les modifications pour une version **multithread** à la discrétion du lecteur.

On considère que les instances à gérer sont de la classe objetGere, déclarée dans le fichier *objetGere.hpp*. Cinquante instances seront créées, si le nombre des instances dépend de la session du programme, on peut ajouter une fonction *init* qui prendra en paramètre le nombre d'instances, et d'autres si besoin est.

manager.hpp

```
/**
 * \file manager.hpp
 * Ce fichier contient les déclarations de la classe Manager
 */

// On va utiliser les pointeurs intelligents de Boost pour gérer les objets alloués dynamiquement
#include <boost/shared_ptr.hpp>
#include "objetGere.hpp"

// On décide de gérer 50 objets.
const unsigned int nbObjets = 50;

class Manager
{
public:
  /// Type des objets retournés, pour simplifier l'écriture
  typedef boost::shared_ptr<objetGere> pointeurObjetGere;

private:
  /// Le constructeur par défaut du manager est déclaré privé
  Manager();

  /// La liste des objets qu'on peut donner
  std::list<pointeurObjetGere> listeObjetsGeres;
  /// La liste des objets corrompus, si on veut les conserver
  std::list<pointeurObjetGere> listeObjetsCorrompus;
public:
  /**
   * Retourne la seule instance du manager
   * @return l'instance en question
   */
  static Manager& getInstance();
  /**
   * Retourne une instance de l'objet géré
   * @return une instance
   */
  const pointeurObjetGere getObject();
  /**
   * Rend un objet
   * @param rendObject est un pointeur vers l'objet rendu
   */
  void releaseObject(const pointeurObjetGere& rendObject);
  /**
   * Rend un objet corrompu
   * @param rendMauvaisObject est un pointeur vers un un objet corrompu
   */
  void releaseBadObject(const pointeurObjetGere& rendMauvaisObject);
};
```

manager.cpp

```
#include <manager.hpp>
```


Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

manager.cpp

```
Manager::Manager()
 :listeObjetsGeres(nbObjets), listeObjetsCorrompus()
{
 for(std::list<pointeurObjetGere>::iterator objectInitialize = listeObjetsGeres.begin();
  objectInitialize != listeObjetsGeres.end(); ++objectInitialize)
 {
  objectInitialize->reset(new objetGere);
 }
}

Manager& Manager::getInstance()
{
 static Manager instance;
 return instance;
}

const pointeurObjetGere Manager::getObject()
{
 if(listeObjetsGeres.empty())
 {
  throw std::exception;
 }
 pointeurObjetGere front = listeObjetsGeres.front();
 listeObjetsGeres.pop_front();
 return front;
}

void Manager::releaseObject(const pointeurObjetGere& rendObject)
{
 listeObjetsGeres.push_back(rendObject);
}

void Manager::releaseBadObject(const pointeurObjetGere& rendMauvaisObject)
{
 listeObjetsCorrompus.push_back(rendMauvaisObject);
}
```

En ce qui concerne la dernière fonction, on peut la vider entièrement et ne pas conserver les instances corrompues, puisqu'elles sont gérées par les pointeurs intelligents. On peut aussi en profiter pour créer de nouvelles instances. De manière générale, on devrait pouvoir se passer de recréer de nouveaux objets mais si les objets sont régulièrement corrompus, il faudrait plutôt songer à revoir la classe pour la corriger.

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Conclusion

Ce pattern ne figure pas parmi les plus utilisés ou les plus connus puisqu'il n'est pas exposé dans le **livre du GoF**, mais dans le livre de *Mark Grant*, Patterns in Java, volume 1 ainsi que celui de *Clifton Nock*, Data Access Patterns.

Utilisez ce Pattern quand :

- vous voulez vous constituer une liste d'objets réutilisables (et ainsi éviter les allocations/désallocations inutiles d'où gain de temps et limitation du risque de fuite mémoire)
- vous voulez facilement contrôler le nombre d'instances de ces objets en mémoire

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Copyright © 2006 - Matthieu Brucher. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.