УДК 811.93(574.3)

В.Н. ХОРОШХИН

Использование паттерна Singleton в языках программирования С++, C#, Java

В инженерии программного обеспечения, шаблон проектирования (паттерн) является общим решением часто встречающейся проблемы в программном проекте. Однако паттерн – отнюдь не законченный вариант решения, который может быть преобразован непосредственно в код. Это всего лишь описание или образец для решения проблемы, которая может возникнуть во многих схожих ситуациях.

Паттерны ускоряют процесс разработки, обеспечивая проверенные способы написания кода. Многократное использование паттернов помогает предотвращать появление «тонких» проблем, которые могут причинять большие неприятности, и улучшает восприятие кода как программистами, так и системными аналитиками.

Таким образом, паттерны позволяют разработчикам использовать хорошо известную общепринятую терминологию для обеспечения взаимодействия проектировщиков программного обеспечения. В процессе работы общие решения, как правило, могут быть улучшены посредством их адаптации для каждого конкретного случая.

Паттерны классифицируются по разным критериям, наиболее распространенным из которых является назначение паттерна. Вследствие этого выделяются порождающие паттерны, структурные паттерны и паттерны поведения. Порождающие паттерны предназначаются для организации процесса создания объектов. Структурные паттерны отвечают за композицию объектов и классов. Паттерны поведения характеризуют способы взаимодействия классов или объектов между собой.

Паттерн Singleton относится к порождающим паттернам, назначение которого состоит в обеспечении наличия в системе только одного экземпляра заданного класса, позволяя другим классам получать доступ к этому экземпляру.

Предположим, что нам понадобился некий глобальный объект, то есть такой объект, доступ к которому можно было бы осуществить из любой точки приложения, но при этом необходимо, чтобы он создавался только один раз. То есть к этому объекту должны иметь доступ все элементы приложения, но работать они должны с одним и тем же экземпляром.

Примером такого объекта может быть хронологический список (history list), в котором хранится информация о всех действиях пользователя, которые он предпринимал, работая с приложением. Объект HistoryList по определению должен быть доступным для всех элементов приложения, чтобы они могли либо заносить в него сведения об очередной операции, выполненной пользователем, либо извлекать из него данные о последней операции для ее отмены.

Один из возможных методов решения этой задачи состоит в создании глобального объекта в главном модуле приложения с последующей передачей ссылки на этот объект всем другим объектам, которым это необходимо. Однако довольно трудно, приступая к разработке приложения, правильно определить способ передачи ссылки, который бы подходил для всех объектов, равно как и заранее предугадать, каким именно элементам приложения понадобится этот объект. Другим недостатком подобного решения является невозможность воспрепятствования другим объектам создавать дополнительные экземпляры глобального объекта (в нашем случае – HistoryList).

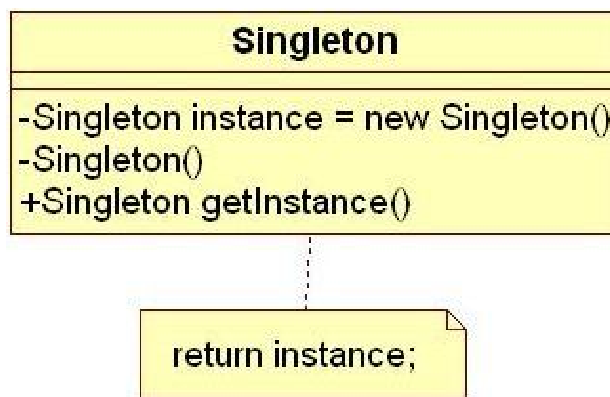
Существует и другой способ получения глобальных значений, основанный на применении статических переменных. Это позволяет приложению обращаться напрямую к нескольким специальным статическим объектам, заключенным внутри некоторого класса, но данный подход также имеет ряд недостатков.

Однако это не лучшее решение, так как статический объект создается во время загрузки класса, что лишает разработчика возможности передачи ему данных перед созданием экземпляра. При этом разработчик не может контролировать доступ к статическому объекту, который объявлен общедоступным. Кроме того, если разработчик решит, что вместо одного объекта ему понадобится, например, пять таких объектов, ему придется практически заново переписать весь код клиентской части приложения.

В подобной ситуации очень полезным становится паттерн Singleton, который обеспечивает удобный доступ всех элементов приложения к глобальному объекту.

При реализации паттерна Singleton используется класс. В этом классе определяется закрытый конструктор, имеется закрытая статическая переменная, в которой хранится ссылка на единственный экземпляр этого класса, а также определен статический метод доступа, возвращающий ссылку на этот экземпляр.

Остальные элементы класса не отличаются от элементов других классов. Статический метод доступа может реализоваться таким образом, чтобы он мог принимать решение о том, какой экземпляр создавать, базируясь на свойствах системы или значениях переданных ему параметров.



UML-диаграмма паттерна Singleton

Листинг 1 . Использование паттерна Singleton на языке C++

```
#include <iostream>
#include <string>
#include <stdlib.h>
using namespace std;
class Singleton
{
   public:
         static Singleton*
instance();
         static void setType(string
t)
         {
               type = t;
               delete inst;
               inst = 0;
         }
         virtual void setValue(int
in)
         {
               value = in;
         }
         virtual int getValue()
         {
               return value;
         }
   protected:
         int value;
         Singleton()
         {
               cout << ":ctor: ";
         }
   private:
         static string type;
         static Singleton* inst;
};
string Singleton::type = "decimal";
Singleton* Singleton::inst = 0;
class Octal: public Singleton
{
   public:
         friend class Singleton;
         void setValue(int in)
         {
               char buf[10];
               sprintf(buf, "%o",
in);
               sscanf(buf, "%d",
&value);
         }
   protected:
         Octal() {}
};
Singleton* Singleton::instance()
{
   if (!inst)
         if (type == "octal")
               inst = new Octal();
         else
               inst = new
Singleton();
      return inst;
   }
   void main()
   {
      Singleton::instance()-
>setValue(42);
      cout << "value is "
   << Singleton::instance()->getValue()
   << endl;
      Singleton::setType("octal");
      Singleton::instance()-
>setValue(64);
      cout << "value is "
   << Singleton::instance()->getValue()
   << endl;
   }
```

Листинг 2 . Использование паттерна Singleton на языке C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MainApp
{
 class Program
 {
 static void Main(string[] args)
 {
 Singleton s1 =
Singleton.Instance();
 Singleton s2 =
Singleton.Instance();
 if (s1 == s2)
 {
 Console.WriteLine
("Objects are the same instance");
 }
 Console.Read();
 }
 }
 class Singleton
 {
 private static Singleton instance;

 protected Singleton(){}
 public static Singleton Instance()
 {
 if (instance == null)
 {
 instance = new Singleton();
 }
 return instance;
 }
 }
}
```

Листинг 3 . Использование паттерна Singleton на языке Java

```
package p00;
public class Singleton {
 private Singleton() {
 }
 private static class
SingletonHolder {
 private static final Singleton
Instance =
new Singleton();
 }
 public static Singleton
getInstance() {
 return SingletonHolder.Instance;
 }
}

package p00;
```

```
import p00.Singleton.*;
public class NewMain {
 public static void main(String[]
args) {
 Singleton s1 =
Singleton.getInstance();
 Singleton s2 =
Singleton.getInstance();
 if (s1 == s2)
 {
 System.out.println
("Objects are the same instance");
 }
 }
}
```

СПИСОК ЛИТЕРАТУРЫ

1. Microsoft System Developer Network, 2008.
2. Java SE 6 Documentation, 2010.