Clases Gestoras

El uso de clases gestoras (manager) es una solución de diseño e implementación ampliamente extendida en Diseño y Programación Orientada a Objetos. En esencia un gestor es un objeto que maneja y organiza un conjunto de otros objetos. Usar clases gestoras es simple y fácil.

En un sistema siempre existen objetos simples (en Java se dicen POJO=plain oll java object).

Objetos simples

Son aquellos formados principalmente por atributos atómicos descriptivos y métodos set/get. Por ejemplo: Canción, fecha, hora, Persona, Articulo, Alumno, Materia, Domicilio, etc.

Por ejemplo, un objeto canción tendrá un titulo, un autor, una duración y talvés una clasificación (rock, pop, punk, tango, etc.)

Qué es una clase gestor o gestora?

Una clase gestora administra/maneja/gestiona un conjunto de objetos simples. La necesidad de manejar de manera centralizada objetos simples ocurre cuando en un sistema o aplicación, existen múltiples instancias de estos objetos simples. Cuando se dice "múltiples instancias", estas pueden ser cientos, miles y hasta millones.

Por ejemplo, ¿cuántos libros hay en una biblioteca?, ¿Cuántos poemas hay en un libro de poemas?, ¿Cuántas cuentas de ahorro se puede manejar un cajero automático?. En todos los casos, la respuesta es "muchos".

Funciones – Servicios

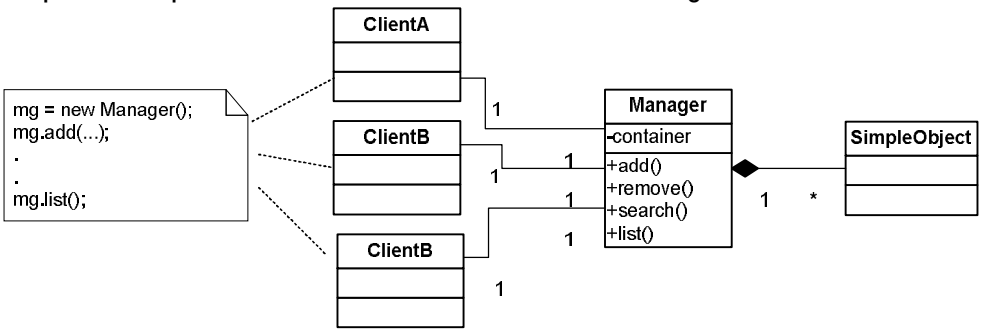
Así el gestor tiene la función de facilitar el almacenamiento, búsqueda y actualización de estos objetos, es decir proveer servicios (métodos) que permitan el acceso a cada instancia. A la vez, estos servicios son realizados de manera absolutamente transparente para aquellas otras clases (clientes) que requieren los servicios del gestor. Es decir, desconocen todo detalle de implementación.

Ejemplos de gestores:

1) Guía telefónica
2) Surtidor de combustible
3) Biblioteca
4) Guía de programación de canales
5) Sistema de Archivos
6) Libro
7) Caja de Supermercado
8) Cajero automático
9) Vuelo
10) Equipo de Futbol
11) Editor de línea
12) Etc.

Diseño e Implementación de Sistemas con Gestores

La clase gestora (Manager) "tiene" o se compone de un conjunto de objetos simples (SimpleObject), y provee métodos que manipular este conjunto. Por otro lado, el gestor ofrece sus servicios a otras clases (ClientA, ClientB y ClientC) que cumplen el rol de clientes. Es claro, que los clientes nunca mantienen interacción directa con los objetos simples, siempre la relación se establece a través del gestor.



Estructura

La estructura básica de cualquier gestor siempre es la misma, ya que se compone al menos de:

- un contenedor (container) de objetos
- un conjunto de operaciones (métodos-servicios) que permiten la manipulación de dicho contenedor.

El contenedor puede estar implementado mediante cualquier estructura de datos que soporte una colección de elementos, tales como: un array, una lista, un árbol, un archivo, una base de datos, etc.

Las operaciones básicas de un gestor son:

- Agregar o añadir nuevos objetos al contenedor
- Eliminar o remover objetos del contenedor
- Buscar un objeto en el contendor
- Recorrer el contenedor con diversos propósitos.

Implementación

En Java y con los conocimientos que hoy tenemos, un gestor básicamente será una clase que encapsula un vector que sirve de contenedor. Más adelante se podrán usar otros tipos de contenedores (listas o árboles u archivos). La relación que se da entre el gestor y los objetos que maneja, es la composición.

Ejemplo

Un contacto refiere a una persona u empresa cuya identificación y correo electrónico conforman la descripción mínima necesaria. En los sistemas "clientes de correo" como Outlook o servicios de mail en internet (Yahoo-Hotmail) los usuarios tienen muchos contactos agendados o guardados. Una Lista de Contacto permite acceder a estos, crear nuevos, buscar, y eliminar contactos existentes.
A continuación se presenta el diagrama, y el código del gestor (ContactList) y el menú que permite acceder a los servicios del gestor.

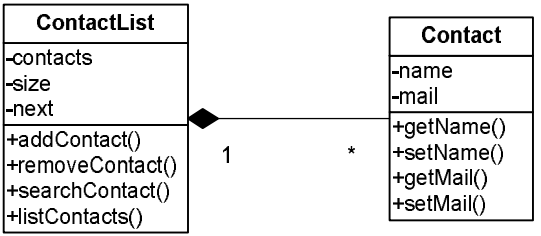

ContactList
-contacts
-size
-next
+addContact()
+removeContact()
+searchContact()
+listContacts()
1
*
Contact
-name
-mail
+getName()
+setName()
+getMail()
+setMail()

Diagrama 2.

```
public class Contact {
     private String name, mail;
     Contact(String name, String mail){
      this.name=name;
      this.mail=mail;     }
     public String getName(){
      return this.name;
     }
     public String getMail(){
      return this.mail;
     }
     public void setName(String name){
      return this.name=name;
     }
     public void setMail(String mail){
      return this.mail=mail;
     }
}
```

```
public class ContactList {
   private Contact [] contacts;
   private int size, next;

public ContactList(int n){
  this.size=n;
  this.contacts = new Contact[size];
  this.next=0;
}

public void addContact(Contact contact) {
  if (!(this.next < this.size))
      this.enlarge();
  contacts[next]=contact;
  this.next++;
}
```

```
public void removeContact(String contactName){
   boolean found=false;
   int count=0;
   Contact aux;
   while(!found && count<next){
     if (contacts[count] != null) {
       aux=contacts[count];
       if (aux.getName().equals(contactName))
         found=true;
       }
       count++;
   }
   if (found)
    contacts[count-1]= null;
}
```

```
public boolean searchContact(String contactName){
   boolean found=false;
   int count=0;
   Contact aux;
   while(!found && count<next) {
      if (contacts[count] != null)  {
         aux=contacts[count];
         if (aux.getName().equals(contactName))
             found=true;
      }
      count++;
   }
   return found;
}
```

```
public void listContacts() {
  int count=0;
  Contact aux;
  while(count<next) {
     if (contacts[count] != null) {
       aux=contacts[count];
       System.out.println(aux.getName());
       System.out.println(aux.getMail());
     }
     count++;
  }
}
```

```
private void enlarge() {
  int count2=0;
  Contact [] aux=new Contact[size+100];
  for(int count1=0; count1<size; count1++)
     if (contacts[count1] != null)  {
         aux[count2]=contacts[count1];
         count2++;
     }
  this.contacts = aux;
  this.next=count2;
  this.size=size+100;
}
}
```

En la implementación propuesta, el método *enlarge* tiene dos funciones muy importantes. Primero permite redimensionar el arreglo, de manera tal de superar las restricciones estáticas de los arreglos, que impiden agregar elementos a los mismos cuando se completa su capacidad. Por otro lado, permite recuperar el espacio asignado a los objetos eliminados de manera eficiente (sin corrimiento de elemento cada vez que sucede una operación de remover)

```
public class MainMenu {

  public static void main(String[] args) {
    int op = 0;
    ContactList myContactList =new ContactList(50);
    do {
     // mesnajes menu
     op = Console.readInt("enter option: ");
     switch (op){
     case 1: {
       System.out.println("Enter Contact Details");
       System.out.println("-----------------");
       name = Console.readString("Name: ");
       String mail = Console.readString("Mail: ");
       Contact contact = new Contact(name, mail);
       myContactList.addContact(contact);
       break;
     }
```