Sistemas Informáticos
Curso 2005-2006

Estudio de un recolector de memoria generacional con referencias externas

Iván García Vicario

Dirigido por:
Prof. María Teresa Higuera Toledano
Dpto. Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de
Madrid

Resumen

Este proyecto se trata de un recolector de memoria que actúa siguiendo el algoritmo generacional, de manera que el heap está dividido en dos generaciones. Además del heap existen otros espacios de almacenamiento llamados regiones, donde el recolector no actúa, pero debe controlar las referencias al heap desde las regiones, dado que dichas referencias están permitidas. Las tareas cuya ejecución tiene lugar en las regiones, pueden ser críticas. Las tareas críticas ocupan poco espacio en memoria y no deben ser interrumpidas por el recolector, porque no pueden perder su plazo. El recolector de memoria se ejecuta en paralelo a la aplicación, por lo tanto debe existir comunicación entre ambos.

This project is about a garbage collector which works using the generational algorithm, with the heap divided into two generations. There are also others storage spaces besides the heap, which are called regions, where the garbage collector does not work, but it must control the references from the regions to the heap, due to these references are allowed. Tasks executed in the regions can be critical. Critical tasks require a little memory space and they must not be interrupted by the garbage collector, because they must not loose their deadlines. Garbage collector and aplication execute at the same time, therefore they must be comunicated.

Palabras clave

Recolección de memoria, generacionales, manejador, región de memoria, sistemas de tiempo real, gestión de memoria, lenguaje JAVA

La Universidad Complutense está autorizada a difundir y utilizar este proyecto con fines académicos no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Indice

1.-GARBAGE COLLECTOR 7
1.1.- Algoritmo de los colores 7
1.2.- Problemas en una ejecución concurrente del garbage collector 9
2.-GARBAGE COLLECTOR GENERACIONAL 11
2.1.-Tipos de recolección 11
2.2- Problema de relación intergeneracional en la minor collection 12
2.3.- Promoción de un objeto desde la generación nueva hasta la generación antigua 13
3.- SISTEMA BASADO EN REGIONES 13
3.1.- Relaciones entre el heap y las regiones 15
3.2.- Conflictos entre el programa y el garbage collector 15
4.- TIPOS DE TAREAS 17
5.- EJEMPLO DE FUNCIONAMIENTO: HEAP OF FISH 18
5.1.- Introducción 18
5.1.1.-Espacios de almacenamiento 20
5.1.2.- Sistema de direccionamiento 21
5.1.3.- Almacenamiento de punteros de la región al heap *(véase pag. 15)* 22
5.1.4.- Major collection y minor collection 23
5.1.5.- Distinción entre tarea normal y tarea crítica *(véase pag. 17)* 24
5.1.6.- Eliminación de la opción Compact Heap 24
5.1.7.- Control de concurrencia entre aplicación y garbage collector ... 24
5.2.- Diagrama de clases 26
5.3.- Funcionamiento de la aplicación 30
5.4.- Evolución del prototipo 39
5.4.1.- Primera versión 39
5.4.2.- Segunda versión 43
5.4.3.- Tercera versión 45
5.4.3.1.- Sistema de direccionamiento 46
5.4.3.2.- Incorporación de un motor de ejecución 47
5.5.- Pruebas realizadas 48
5.6.- Ajuste de parámetros 53
APÉNDICE I.- INSTRUCCIONES UTILIZADAS PARA REPRESENTAR LAS ACCIONES TÍPICAS DE LA APLICACIÓN EJEMPLO 54
APÉNDICE II.- CÓDIGO MODIFICADO SOBRE EL PROTOTIPO ORIGINAL. 55
BIBLIOGRAFÍA 135

1.-Garbage collector

Un garbage collector es una funcionalidad presente en la máquina virtual de Java que consiste en detectar todos aquellos objetos que son inalcanzables por el programa en ejecución y liberar la zona de memoria correspondiente a ellos. El garbage collector se ejecuta periódicamente, ya que en una aplicación media a lo largo del tiempo ciertos objetos se van volviendo inalcanzables. Es necesario que la ejecución del garbage collector sea transparente para la aplicación, ya que el programador de la aplicación no sabe nada de la ejecución del garbage collector, por lo que el garbage collector debe controlar todas aquellas acciones de la aplicación que puedan hacer ver al garbage collector que un objeto alcanzable por la aplicación no lo es y evitar que se libere un objeto necesario para la aplicación.

1.1.- Algoritmo de los colores

Para detectar que objetos son alcanzables y que objetos no son alcanzables, es necesario hacer un recorrido del grafo de objetos. Para determinar qué objetos son alcanzables y qué objetos no lo son, se procede así: Se parte de un conjunto denominado conjunto de variables raíz, que es el conjunto de variables que utiliza la aplicación para acceder a todos los datos que utiliza. A partir de ese conjunto se hace un recorrido en profundidad de la siguiente manera:

- Inicialmente, todos los objetos están coloreados de blanco.
- Cada vez que pasemos por un objeto, se colorea de gris.
- Cuando hayamos recorrido todos los hijos de un objeto, se colorea de negro.
- Al finalizar el recorrido, todos los objetos coloreados de blanco serán liberados.

Por lo tanto, el significado de los colores es el siguiente:
- Blanco: Aún no ha sido detectado como objeto vivo.
- Gris: Ha sido detectado como objeto vivo, pero el recorrido de su descendencia no ha sido completado.
- Negro: Ha sido detectado como objeto vivo y se ha completado el recorrido de su descendencia.

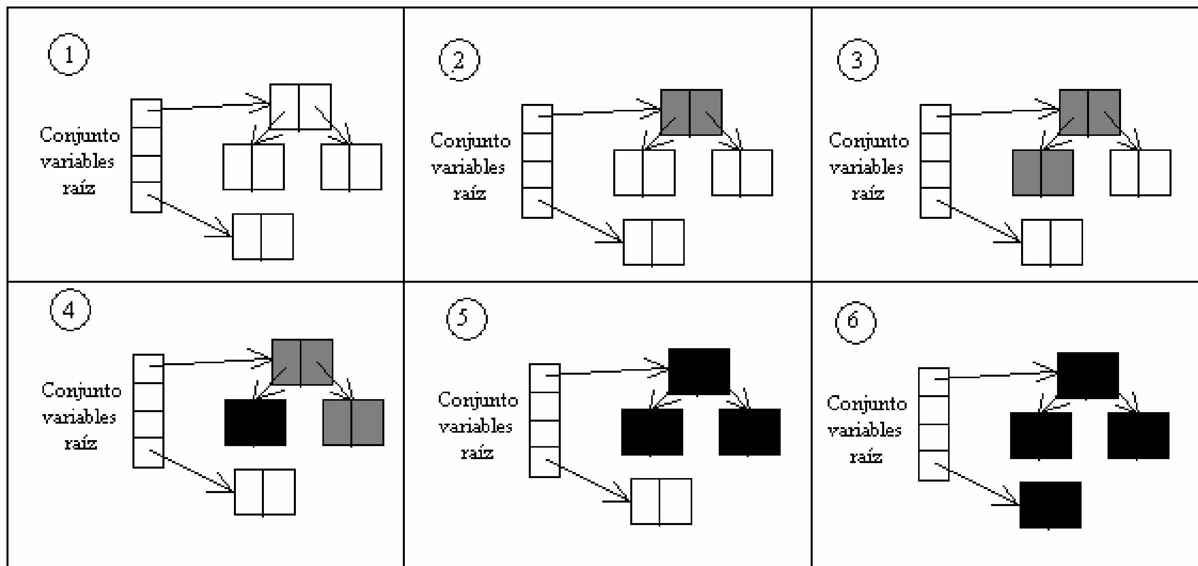


Fig. 1: Recorrido de los objetos siguiendo el algoritmo de los colores

En esta figura tenemos dos variables raíz. En los pasos del 1 al 5 se va mostrando como se va coloreando el árbol de objetos apuntado desde la primera variable. Nótese que cada vez que se alcanza alguna de las hojas del árbol se colorea de negro. Cuando se han alcanzado todas las hojas del árbol, se colorea todo el árbol de negro. En el paso 6 se muestra el recorrido desde la segunda variable, que, al encontrar un único objeto, se acaba la exploración desde esa variable y se colorea el objeto de negro.

1.2.- Problemas en una ejecución concurrente del garbage collector

Supongamos que nos encontramos ante una situación como la de la figura:

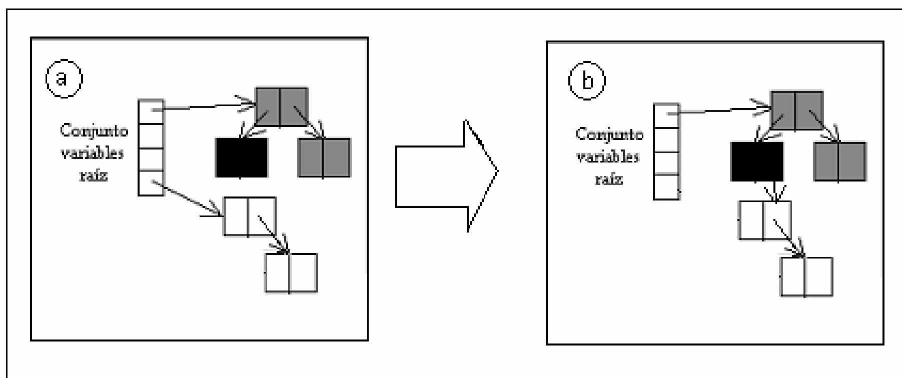


Fig. 2: Problema de concurrencia al aplicar el algoritmo de los colores

Como se puede observar en la figura *2a*, el penúltimo objeto mirando de arriba abajo está apuntado desde una de las variables locales y, debido a las acciones del programa, pasa a ser apuntado desde el objeto que ha sido coloreado de negro, dejando además de ser apuntado desde la variable local (figura *2b*). Por tanto, la única manera de acceder a este objeto es desde los atributos del objeto coloreado de negro. Esto genera el siguiente problema: Dado que el único camino hasta el objeto es a través del objeto coloreado de negro, y este último ya ha sido explorado por el garbage collector, nuestro objeto y su descendencia que no esté apuntada desde ningún otro objeto o variable van a ser recolectados, cuando se tratan de objetos vivos (figura *3*).

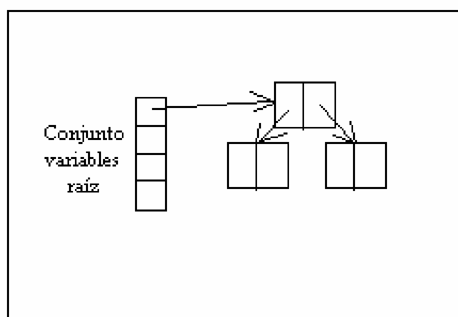


Fig. 3: Situación al finalizar la recolección

Este problema, tal como muestra la figura *4*, se resuelve coloreando de gris un objeto que no ha sido coloreado, es decir, que tiene el color blanco, cada vez que se establece una referencia hasta él desde un objeto coloreado de negro. Para que esta acción tenga sentido, es necesario que el garbage collector, antes de liberar la memoria correspondiente a objetos que se han quedado blancos, explore aplicando el algoritmo de los colores toda la descendencia de los objetos que han quedado coloreados de gris, ya que gris significa que el objeto ha sido visitado, pero no así su descendencia. De esta manera se evita que el objeto y su descendencia sean recolectados.

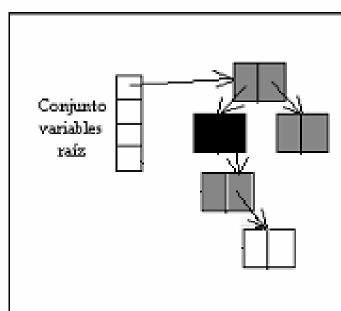


Fig. 4: Solución al problema

2.-Garbage collector generacional

Un garbage collector generacional es aquel que tiene dividido el heap en generaciones según el tiempo de vida de los objetos y que ejecuta dos tipos de recolección, denominadas minor collection y major collection. Su funcionamiento está basado en la teoría de la "mortalidad infantil", que dice que la mayoría de los objetos, a partir de cierta edad, permanecen vivos en memoria hasta el final de la ejecución del programa. Tenemos, por tanto el heap dividido tal que se muestra en la figura.

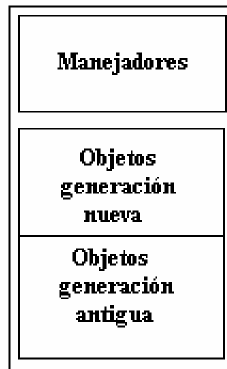


Fig.5: División del heap en un algoritmo generacional

Como se puede observar en la figura *5*, el heap está dividido en dos zonas, la zona de manejadores y la zona de objetos. Los manejadores contienen la dirección de memoria de cada objeto y alguna otra información acerca del mismo. Cuando un objeto contiene una referencia a un objeto, contiene la dirección de memoria de su manejador. Esto tiene como ventaja que si un objeto promociona, no es necesario modificar la dirección en todas las referencias al propio objeto, sino solamente la contenida en el manejador. La zona de objetos se divide en dos partes, la primera dedicada a los objetos nuevos y la segunda a los objetos antiguos. Los objetos de la generación nueva poseen un contador que cuenta el número de veces que ha sobrevivido el objeto al garbage collector y cuando este contador adquiere una cantidad determinada, este objeto promociona a la generación de objetos antiguos.

2.1.-Tipos de recolección

- Major collection. Este tipo de recolección actúa sobre todo el heap y consiste en un recorrido empezando desde el conjunto de variables iniciales aplicando el algoritmo de los colores. Los objetos que queden coloreados de blanco al finalizar el recorrido serán liberados.
- Minor collection. Este tipo de recolección actúa sobre la zona del heap perteneciente a la generación nueva y actúa aplicando el algoritmo de los colores empezando

desde el conjunto de variables iniciales y parando no solo cuando lleguemos a un objeto que no tenga referencias a ningún otro, sino también cuando lleguemos a un objeto de la generación nueva que tenga solamente referencias a objetos de la generación antigua. Los objetos de la generación nueva que queden coloreados de blanco al finalizar el recorrido serán liberados y no se hará este chequeo en la generación antigua.

2.2- Problema de relación intergeneracional en la minor collection

Al realizar la minor collection podemos tener problemas como el de la figura

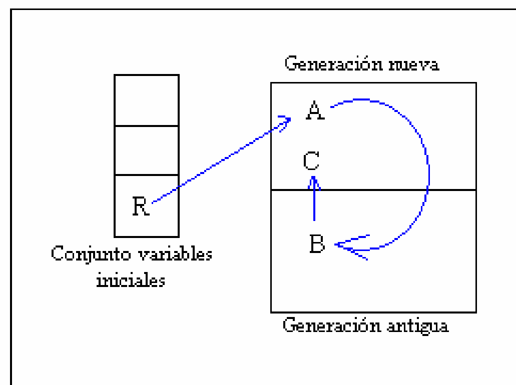


Fig.6: Ejemplo de relación intergeneracional

Obsérvese que la variable *R* apunta al objeto *A*, éste apunta al objeto *B* y, finalmente el objeto *B* apunta al objeto *C*. En este caso la manera de actuar del garbage collector, teniendo en cuenta que estamos ante una minor collection, sería así: empieza a buscar desde el conjunto de variables iniciales, que en este caso, solamente contamos con la variable *R*, llega al objeto *A*, no encuentra ninguna referencia a objetos de la generación nueva, se para el recorrido y se marca *A*. *C*, al no estar marcado será liberado a pesar de ser un objeto alcanzable. Para evitar este tipo de situaciones, hay que hacer lo siguiente:

1. Si se establece una referencia desde un objeto antiguo a un objeto nuevo, hay que añadir el objeto nuevo al conjunto de variables iniciales, y en caso de que estuviese, aumentar su cardinalidad en una unidad.

2. Si se anula una referencia que había desde un objeto antiguo a un objeto nuevo, disminuir en el conjunto de variables iniciales la cardinalidad en una unidad y, si se alcanza cardinalidad 0, eliminarlo del conjunto.

3. Si un objeto que pertenecía a la generación nueva promociona a la generación antigua, eliminarlo del conjunto de variables iniciales, sin importar la cardinalidad del mismo.

2.3.- Promoción de un objeto desde la generación nueva hasta la generación antigua

Ahora tenemos una pregunta clave. ¿Cuándo un objeto promociona desde la generación nueva hasta la generación antigua? Cada objeto tiene un contador y dicho contador se incrementa cada vez que se realiza una minor collection y el objeto sobrevive. Cuando este contador llega a una determinada cantidad, promociona. La promoción de este objeto se gestiona por el propio garbage collector. Para realizarla no hay más que buscar un bloque libre en la generación antigua, copiar todos los atributos del objeto, poner el bloque anteriormente utilizado como bloque libre y actualizar la dirección del objeto dentro del manejador.

3.- Sistema basado en regiones

En este prototipo tenemos al heap como único espacio para almacenar objetos dinámicos, pero también existen sistemas con regiones, donde además del heap se pueden crear objetos en dichas regiones. El sistema sería como en la figura:

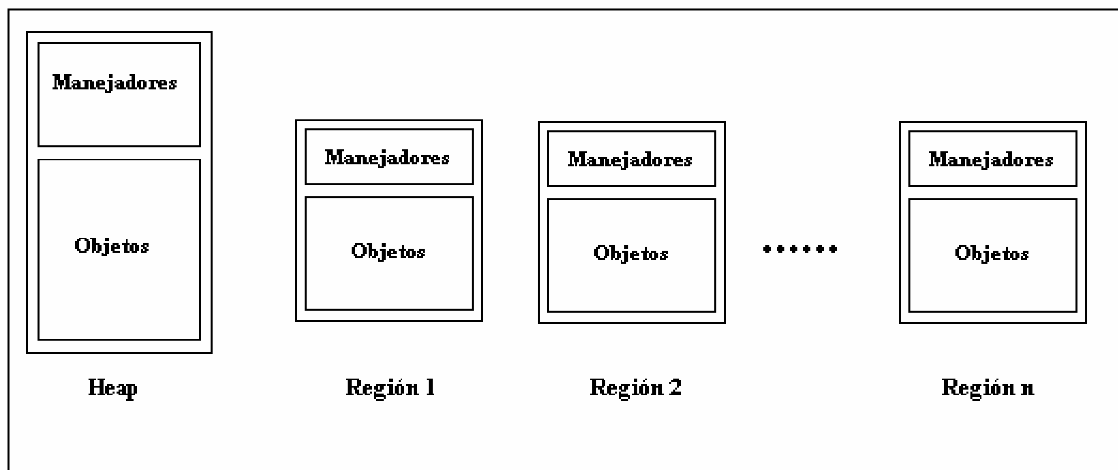


Fig. 7: Estructura del heap y de las regiones

Esto sería un sistema con n regiones, es decir, además del heap disponemos de n regiones para almacenar los objetos en memoria. Cada región, al igual que el heap, se divide en zona de manejadores y zona de objetos. El hecho de que cada región tenga su propia zona de manejadores proporciona diversas ventajas, tales como facilitar la gestión de direcciones de memoria o facilitar la tarea de destruir una región.

La utilidad que tiene esta forma de dividir la memoria es ofrecer la posibilidad de que hayan zonas de memoria donde el recolector no interrumpa a la aplicación. Esto es

especialmente importante en las aplicaciones de tiempo real y en aquellas que sean críticas, es decir que necesitan ser ejecutadas con determinismo temporal. En las tareas críticas, en términos generales, se utiliza muy poca memoria en su ejecución, pero es especialmente importante saber cuál es su tiempo de ejecución. Ya que el recolector puede provocar una parada en la aplicación, el cual es un suceso desfavorable, cuando se necesita ejecutar una tarea crítica, se ejecuta en la región, donde la memoria se libera solamente al final, cuando se destruye la región.

3.1.- Relaciones entre el heap y las regiones

Vamos a considerar las siguientes relaciones entre el heap y las regiones:

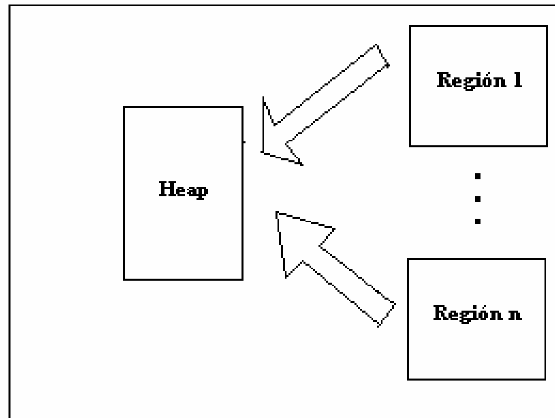


Fig. 8: Relación entre las regiones y el heap

Es decir, desde el heap solamente puede haber referencias a objetos del mismo y desde cada una de las regiones puede haber referencias a objetos de la misma región o del heap, pero no se permiten referencias entre objetos de distintas regiones. Esto evita que se produzcan "punteros colgados". Esta situación se produce cuando se destruye una región que tiene objetos referenciados desde el heap o desde otra región.

3.2.- Conflictos entre el programa y el garbage collector

Hay determinadas acciones que un programa puede realizar y que pueden ocasionar problemas en el garbage collector:

Caso 1: Establecer una referencia desde un objeto situado en la región a un objeto situado en el heap

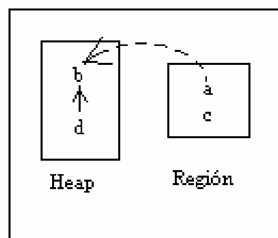


Fig. 9

En este caso se está asignando el objeto *b*, situado en el heap a uno de los atributos del objeto *a*, situado en la región. El objeto *d* contiene un atributo llamado *atrib1*, que es el que contiene la referencia a *b*. Imaginemos que después de asignar *b* a uno de los atributos de *a* se hiciese una asignación como la siguiente:

d.atrib1 = null;

Entonces *b* se quedaría sin ninguna referencia al mismo y sería recolectado por el garbage collector, siendo que está referenciado desde el objeto *a* de la región. Para evitar esto hay que insertar *b* en la tabla de referencias desde el momento en que se asigna *b* a un atributo de *a*.

<u>Caso 2</u>: Eliminar una referencia al heap desde un objeto situado en la región

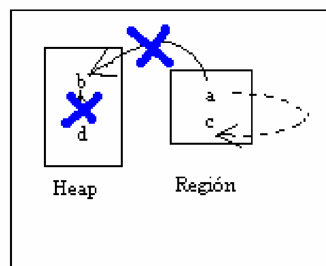


Fig. 10

Ahora en este caso, además de desasignar *b* a *d*, *a* pasa de apuntar a *b*, a apuntar a *c*, que como se observa en la figura *c* pertenece a la región, pero en el caso general también podríamos aceptar que pasase a apuntar otro objeto del heap, como, por ejemplo, *d* y en este caso se haría la combinación de acciones del caso anterior y del caso actual. En el caso de la figura se actuaría dependiendo si la tarea que está siendo ejecutada en la región es o no crítica.

- Si la tarea es crítica no es necesario realizar ninguna acción, ya que *b* será liberado cuando la región sea destruida.
- Si la tarea no es crítica hay que eliminar una instancia de *b* de la tabla de referencias, para que sea liberado si no está referenciado desde ningún otro lugar.

<u>Caso 3</u>: Cambiar una referencia procedente de un objeto de la región desde un objeto del heap hasta otro objeto también situado en el heap

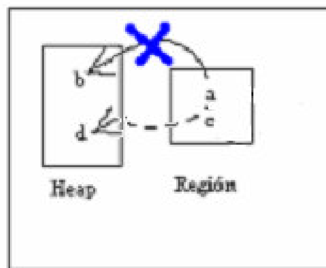


Fig.11

En este caso, *a,* objeto situado en la región tiene como atributo un puntero a *b* y ese atributo es sustituido por un puntero al objeto *d*. En este caso, al igual que el anterior debemos proceder de distinta manera dependiendo si la tarea es o no crítica.

- Si la tarea no es crítica, se borraría una instancia de *b* y se añadiría una instancia de *d* a la tabla de referencias.
- Si la tarea es crítica, es necesario añadir una instancia de *d* a la tabla de referencias, ya que, en caso contrario, el objeto *d* corre el riesgo de ser eliminado por el garbage collector.

Por último, cuando se destruya la región, la tabla de referencias será borrada, por lo que no será necesario hacer ninguna actualización. En este caso, los objetos del heap serán liberados en la siguiente ejecución del garbage collector si y solo si no están referenciados desde ningún otro objeto del heap.

4.- Tipos de tareas

En esta versión se han definido dos tipos de tareas:

- Tarea normal. Se lanza a ejecución en el heap o en una región y puede ser interrumpida por el garbage collector. Los objetos son creados en el heap o en una región y, en caso de que se ejecute en una región y se anule alguna referencia de la región al heap, se actualiza la lista de referencias de dicha región al heap, decrementando la cardinalidad de dicho elemento en dicha lista.
- Tarea crítica. Se lanza a ejecución en una de las regiones y no debe ser interrumpida por el garbage collector. Los objetos son creados solamente en alguna región y, en caso de que se anule alguna referencia de la región al heap no se hace nada porque la ejecución de esta tarea es lo más prioritario, la anulación de una referencia no supone un riesgo de pérdida de datos por una recolección de objetos necesitados por alguna aplicación, sino el efecto contrario: la presencia de la llamada "basura flotante", objetos cuya memoria no es liberada por el garbage collector al suponer que son objetos válidos. Dado que la lista de referencias será vaciada cuando la región sea destruida, en la siguiente ejecución del garbage collector, dicha memoria sería eliminada si así procediese.

5.- Ejemplo de funcionamiento: Heap Of Fish

5.1.- Introducción

En este proyecto pretendemos estudiar un recolector de memoria de Java y modificar algunas de sus características. En cuanto a decidir si modificamos algún ejemplo o implementamos algo nuevo, algo que influyó mucho en nuestra decisión fue un ejemplo que nos presentó la profesora supervisora, llamado Heap Of Fish. En él, a través diversos pasos, se crean objetos de 3 clases, llamadas: *RedFish (III), BlueFish (II)* y *YellowFish (I)*. Todos los objetos de cada una de estas clases tiene un atributo del mismo tipo, que se llama *myFriend*, las clases *RedFish* y *BlueFish* tienen un atributo llamado *myLunch*, que es de tipo *BlueFish* en la clase *RedFish* y de tipo *YellowFish* en la clase *BlueFish* y, por último la clase *RedFish* tiene un 3er atributo *mySnack*, de tipo *YellowFish*. Una vez creados dichos objetos, el usuario se encarga de asignar las referencias entre ellos, teniendo en cuenta que en la pila existe una referencia de cada clase. Una vez hecho este proceso, actúa el recolector de memoria, liberando la memoria no accesible y compactando el heap.

Esta es la implementación de cada una de las clases

```
class YellowFish
{
  YellowFish myFriend;
}
```

(I) .- Clase YellowFish

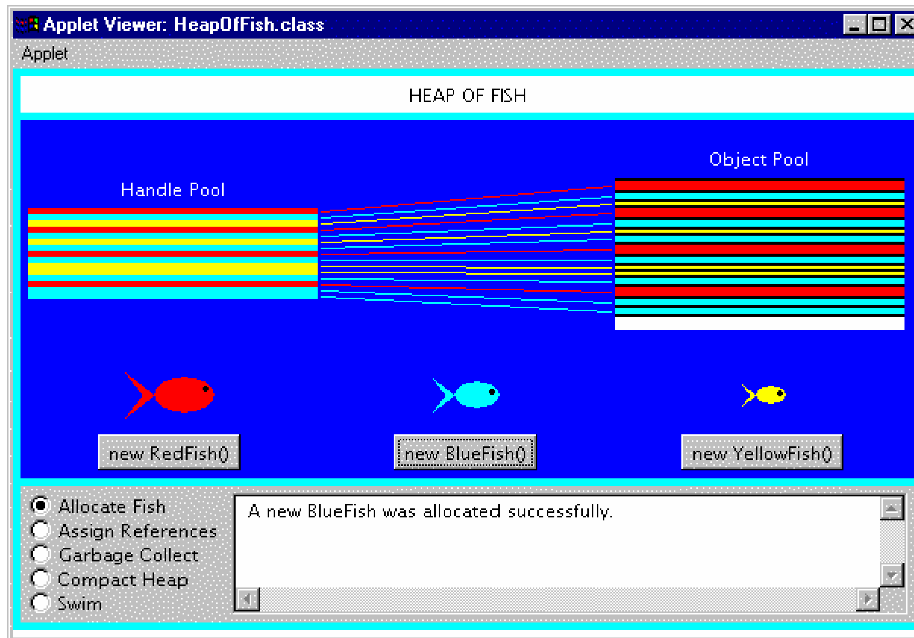
```
class BlueFish
{
  BlueFish myFriend;
  YellowFish myLunch;
}
```

(II).- Clase BlueFish

```
class RedFish
{
  RedFish myFriend;
  BlueFish myLunch;
  YellowFish mySnack;
}
```

(III).- Clase RedFish

Y aquí se muestra la representación del applet:



Cada objeto tiene un manejador, que contiene entre otras cosas la dirección del objeto en el heap. Con esto se consigue que cada vez que haya que mover un objeto dentro del heap no sea necesario actualizar cada una de las referencias a ese objeto, sino solamente la contenida en el propio manejador.

En esta imagen se muestra el applet, con las opciones disponibles, un dibujo que ilustra la manera de ir almacenando objetos en memoria, mostrando los manejadores y los objetos en cuestión y en la parte inferior los iconos utilizados para representar cada tipo de objeto.

Las opciones disponibles en el modelo original son:

- Allocate Fish. Esta opción se utiliza para crear peces. En el modelo original aparece representado el heap y su ocupación – disponibilidad tanto en la zona de manejadores como en la zona de objeto. Se dispone de un botón para crear cada tipo de peces: *YellowFish*, *BlueFish* y *RedFish*.
- Assign References. En esta opción los peces y las referencias entre ellos aparecen representados en estructura de grafo dirigido. Cuando un pez *b* es atributo de un pez *a*, aparece una línea desde la cabeza de *a* a la cola de *b*. Esta opción se divide en tres subopciones: *Move Fish*, *Link Fish* y *Unlink Fish*. *Move Fish* es una opción que permite mover los peces. Para ello hay que arrastrar los peces a través del applet. *Link Fish* es una opción para establecer referencias entre los peces. Si queremos que el pez *b* sea un atributo del pez *a*, tendremos que arrastrar el ratón desde el pez *a* hasta el pez *b*. *Unlink Fish* es una opción para anular referencias entre peces. Para hacer esto simplemente hay que hacer clic en la línea que representa dicha referencia.
- Garbage Collect. En esta opción se van representando los pasos de ejecución en el garbage collector. Su representación es un grafo igual que el de la opción *Assign*

References. En el modelo original hay 2 botones: *Reset* y *Step*. Hay que pulsar el botón *Reset* cada vez que se arranque el garbage collector y luego hay que pulsar el botón *Step* para mostrar cada uno de los pasos que va dando el garbage collector.

- Compact Heap. En esta opción aparece representado el heap de la misma manera que en la opción *Allocate Fish*, pero tiene como característica ser la opción que permite compactar el heap, es decir mover todos los objetos contiguamente al principio de la zona de objetos.

A partir de este prototipo original hay que conseguir una aplicación que ejecute un garbage collector de tipo generacional, de manera que el heap esté divido en dos generaciones. Además del heap, dispondremos de otros espacios de almacenamiento como las regiones, que se pueden crear y destruir según demanda hasta 4 de ellas. Por otra parte también es necesario que el programa se ejecute de una manera similar a una máquina virtual, es decir sin que el usuario tenga que estar introduciendo órdenes de manera interactiva continuamente.

5.1.1.-Espacios de almacenamiento

Para almacenar los objetos se dispone de un heap, que se divide en zona de manejadores y zona de objetos. Su tamaño es de 15 unidades para la zona de manejadores, es decir, se pueden alojar 15 manejadores y de 50 unidades para la zona de objetos, que a su vez se divide dinámicamente en bloques, es decir se pueden alojar objetos dependiendo de la clase que sean, ya que a la cabecera de cada objeto se le asigna un espacio en la zona de objetos. Cada bloque puede estar libre u ocupado. Cada bloque tiene una cabecera que indica el tamaño del bloque y si está libre o no. Inicialmente el heap se divide en dos bloques libres de manera que el primero de ellos corresponde a la generación nueva de objetos y el segundo a la generación antigua. El heap está representado por la clase *GCHeap*, que deriva de la clase *EspacioAlmacenamiento*, que es la antigua implementación de la clase *GCHeap*. La zona de objetos se divide en dos partes, una de ellas correspondiente a la generación nueva de objetos y la otra correspondiente a la generación antigua. La manera de hacer esta división es mediante un atributo, denominado frontera, que contiene la dirección del primer bloque de la generación antigua.

El otro espacio de almacenamiento es la región, que se diferencia del heap en no tener tamaño estándar y en no tener la zona de objetos dividida en generación antigua y generación nueva. Al igual que el heap, también está dividida en zona de manejadores y zona de objetos y la zona de objetos se divide en bloques, pero inicialmente hay un solo bloque libre que abarca toda la zona de objetos de la región. El tamaño de la región, tanto de la zona de manejadores, como de la zona de objetos, lo decide el usuario cuando la crea. La región está representada por la clase *Region*, que deriva de la clase *EspacioAlmacenamiento*, al igual que el heap. El sistema dispone de 4 regiones como máximo y todas ellas van almacenadas dentro de la clase *EspacioRegiones*, que es la clase que gestiona las regiones.

5.1.2.- Sistema de direccionamiento

Para elegir un sistema de direccionamiento, nos enfrentamos a un problema, que es la imposibilidad de decidir un sistema de direccionamiento predeterminado, ya que no sabemos de antemano que regiones se van a utilizar ni que tamaño van a tener dichas regiones, por lo que no podemos decidir que asignación de direcciones vamos a hacer antes de que la región sea creada. Para solucionar esto, lo que se hace es que cada vez que se crea una nueva región, se le asigna una dirección de inicio y cada vez que se direccione un elemento de esa región tendrá asignada como dirección la dirección dentro de esa región sumada a esa dirección de inicio asignada en su creación.

La manera de asignar las direcciones de inicio para cada región es la siguiente: se busca el valor de más alto de todas las direcciones de inicio de cada región, y, una vez encontrado se le suma el tamaño de la zona de objetos de la región a la que pertenece. En el caso de que no haya ninguna región creada, se asigna como dirección de inicio el tamaño de la zona de objetos del heap.

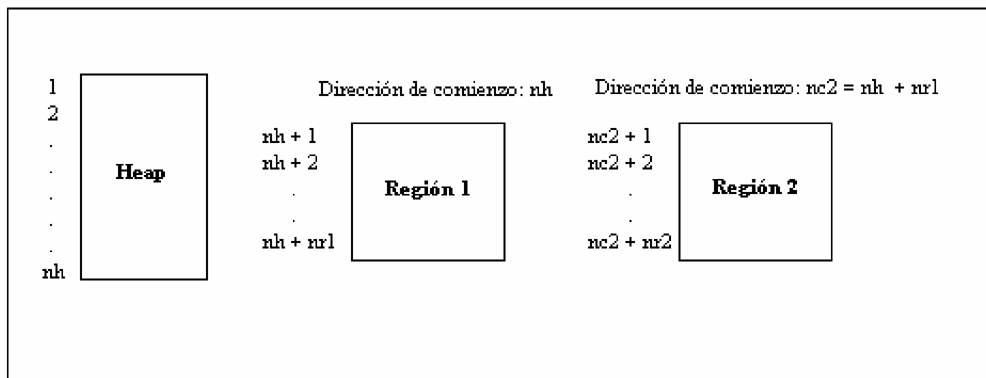


Fig. 12: Direccionamiento en un sistema con un heap y varias regiones

En esta figura se pueden observar dos regiones creadas. La región *1* fue creada en primer lugar. Al ser la primera región creada, se le asigna como dirección de inicio el tamaño del heap (*nh*). Cuando se crea la región *2*, el máximo valor de dirección de inicio hallado es el único que hay, es decir, el de la región *1*, que es *nh*. Para hallar el valor de la dirección de comienzo hay que sumar dicho valor al tamaño de la región, que es nr1. El resultado, etiquetado como *nc2* es *nh + nr1*. Como también se puede observar en la figura, para asignar las direcciones en cada región no hay más que sumar la dirección dentro de la región a la dirección de comienzo.

El significado de las direcciones es el mismo en todos los espacios de almacenamiento. Por esa razón en la clase *Region* han sido sobreescritos los métodos *allocateFish* y *getObjectHandle* de manera que, respectivamente, devuelvan y reciban la dirección que corresponda según este criterio. La dirección de comienzo es un atributo de la clase *Region*, para que cada región tenga conocimiento por si misma de su espacio de direcciones. La asignación de dirección de comienzo en cada región es la misma durante

toda la vida de la región y, por lo tanto, la destrucción de una región no implica una reasignación de direcciones en otras regiones.

5.1.3.- Almacenamiento de punteros de la región al heap *(véase pag. 15)*

Como habíamos comentado anteriormente, es posible establecer punteros desde cualquier región hasta el heap y también es posible que el objeto del heap hacia el cual está establecido el puntero, tenga éste como único puntero. Esto implica que los punteros de la región al heap sean almacenados y también recorridos en cada ejecución del garbage collector. La manera de almacenar dichos punteros va a ser una lista que contenga la tupla (*dh*, *np*), siendo *dh* la dirección del heap apuntada y *np* el número de punteros a la dirección. Así como no es necesario saber la dirección de procedencia de la región ni que el garbage collector realice el recorrido varias veces desde un mismo objeto, es necesario almacenar el número de punteros a un objeto, porque cada vez que se borre alguno interesa decrementar en una unidad el número, por si llega a cero, poder liberar el objeto del heap en la próxima ejecución del garbage collector, en caso de que dicho objeto no esté referenciado desde ningún otro sitio. Cada región tiene una lista de referencias distinta, ya que es necesario eliminar la lista entera en caso de que se destruya la región.

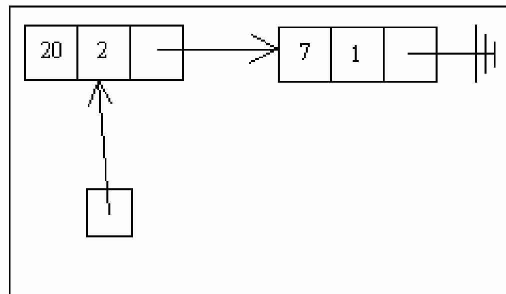


Fig. 13: Lista de referencias

Por ejemplo suponiendo que desde una misma región hay establecidas, desde dos direcciones distintas de la región, referencias al objeto en la dirección 20 del heap y una sola referencia al objeto en la dirección 7 del heap, la lista tendría el aspecto de la figura.

5.1.4.- Major collection y minor collection

Al tratarse de un garbage collector generacional, hay dos tipos de recolección, que son major collection y minor collection. Resumiendo, la major collection se ocupa de recolectar todo el heap y la minor collection se ocupa de recolectar solamente la zona del heap correspondiente a la generación nueva de objetos.

La minor collection aparece representada por la clase *MinorCollection* y la major collection por la clase *MajorCollection*. Ambas clases son hijas de la clase abstracta RecoleccionAbstracta, que implementa la interface *Recoleccion*.

La manera de actuar del garbage collector es la siguiente:

- Se dispone de un array de objetos de la interface *Recoleccion*, es decir que pueden apuntar a objetos de la clase *MajorCollection* o *MinorCollection*.
- En este array todos los objetos son de la clase *MinorCollection* menos el último, que es de la clase *MajorCollection*.
- Cada vez que se ejecuta el garbage collector va ejecutando el tipo de recolección que corresponda según el orden del array.
- Cuando se llegue a la última posición del array se vuelve al principio.

La manera de actuar de la minor collection es:

- En primer lugar se chequean las variables locales haciendo un recorrido en profundidad y aplicando el algoritmo de los colores, de manera que el recorrido es detenido cuando se alcanza un objeto que no tiene descendencia en la generación nueva de objetos.
- Finalizado este recorrido, se chequean de la misma manera todos los objetos de la generación nueva que son apuntados desde objetos de la generación antigua, que han sido almacenados en una lista igual a la utilizada para guardar las referencias desde las regiones al heap.
- Después, otra vez de la misma manera, se chequean todos los objetos de la generación nueva que son apuntados desde objetos procedentes de alguna región. Esto implica recorrer todas las listas de referencias correspondientes a cada región existente.
- Por último, se libera la memoria correspondiente a los objetos de la generación nueva que queden coloreados de blanco después de realizar este proceso, se incrementa en una unidad el contador que indica el número de veces sobrevividas al garbage collector de todos los objetos de la generación nueva que queden después de haber sido eliminados los anteriores y, en el caso de que algún contador de algún objeto alcance una determinada cantidad, el objeto es trasladado a la generación antigua.

Por otra parte, la major collection actúa de manera muy similar al garbage collector del programa original:

- Primero se chequean las variables locales, haciendo un recorrido en profundidad, aplicando el algoritmo de los colores y deteniendo el recorrido cada vez que nos encontremos con un objeto que no tenga descendencia.
- Hacemos lo mismo desde las listas correspondientes a cada región existente.
- Por último, se libera la memoria correspondiente a los objetos del heap que queden coloreados de blanco al finalizar este proceso.

5.1.5.- Distinción entre tarea normal y tarea crítica *(véase pag. 17)*

En esta aplicación la manera de diferenciar si una tarea es normal o crítica es mediante un objeto situado en el hilo de ejecución de la misma. En las tareas normales, dicho objeto es de la clase *TareaNormal* y en las tareas críticas es de la clase *TareaCritica*. Estas dos clases, tanto *TareaNormal* como *TareaCritica* son implementaciones de la interfaz *Tarea*. Tiene el método chequearReferencias, que inserta en la lista de referencias la dirección del heap cuando un objeto de la región pasa a apuntar a uno del heap y en el caso en que la tarea no sea crítica, de la lista una instancia de la dirección a la que deja de apuntar la región. El motor de ejecución y cada hilo del mismo contiene una referencia de tipo *Tarea* y esa referencia apunta a un objeto de la clase *TareaNormal* o *TareaCritica* según si la tarea que se está ejecutando es una tarea normal o una tarea crítica. En las tareas críticas, el hilo de ejecución creado tiene prioridad alta, ya que el código ejecutado no debe ser interrumpido por el garbage collector, así que si entra en ejecución el garbage collector, tendrá menor prioridad y, por lo tanto, no podrá interrumpir la ejecución de dicha tarea.

5.1.6.- Eliminación de la opción Compact Heap

La opción *Compact Heap* ha sido eliminada porque, al estar el heap dividido en generaciones no es necesario hacer compactación, ya que en la generación nueva se crean y destruyen objetos muy frecuentemente y, por tanto, no es posible mantener esta generación compactada. En cuanto a la generación antigua, el hecho es completamente a la inversa, es decir, muy pocos objetos logran llegar a la generación antigua y de los que llegan ahí, muy pocos son destruidos. La compactación en la zona antigua se debe a la filosofía de su funcionamiento, ya que al desplazar un objeto desde la generación nueva se sitúa en el primer bloque libre de memoria con tamaño suficiente, que generalmente es contiguo al anterior objeto situado en dicha generación.

5.1.7.- Control de concurrencia entre aplicación y garbage collector

Dado el carácter atómico que tienen las instrucciones, es necesario que exista algo que garantice esa atomicidad. Es decir, el garbage collector no debe realizar ninguna acción hasta que se complete la ejecución de cada instrucción y no se debe ejecutar ninguna instrucción hasta que no se complete una etapa del garbage collector. Supóngase que la

aplicación está estableciendo una referencia entre dos objetos. Si antes de completarse el proceso de establecimiento de esta referencia el garbage collector finaliza su ejecución, es posible que se libere el objeto destino de dicha referencia, pudiendo ser alcanzable. Por lo tanto, es necesario que si se ha empezado a ejecutar alguna instrucción como establecer una referencia entre dos objetos se complete ese proceso antes que el garbage collector prosiga su ejecución.

Eso se consigue utilizando la sincronización de hilos de JAVA. Hay que declarar como *synchronized* todos aquellos métodos que correspondan a instrucciones y los que corresponden a una etapa del garbage collector, ya sea de la major collection o de la minor collection.

Es decir, están declarados como *synchronized* todos estos métodos:

En la clase *AllocateFishPanel*:

public synchronized void *crearPezRojo(Instruccion ins,PecesIniciales pi,MemoriaDeInstrucciones mem)*

public synchronized void *crearPezAzul(Instruccion ins,PecesIniciales pi,MemoriaDeInstrucciones mem)*

public synchronized void *crearPezAmarillo(Instruccion ins,PecesIniciales pi,MemoriaDeInstrucciones mem)*

En la clase *LinkFishCanvas*:

public synchronized void *asignarReferenciasEntrePecesIndirecto(Instruccion ins,Tarea t,**int** offset, MemoriaDeInstrucciones mem)*

public synchronized void *asignarReferenciasEntrePecesInmediato(Instruccion ins,Tarea t,**int** offset)*

public synchronized void *asignarReferenciasEntrePecesDestinoInmediato(Instruccion ins,Tarea t,**int** offset, MemoriaDeInstrucciones mem)*

public synchronized void *anularReferenciasPeces(Instruccion ins, Tarea t, **int** offset, MemoriaDeInstrucciones mem)*

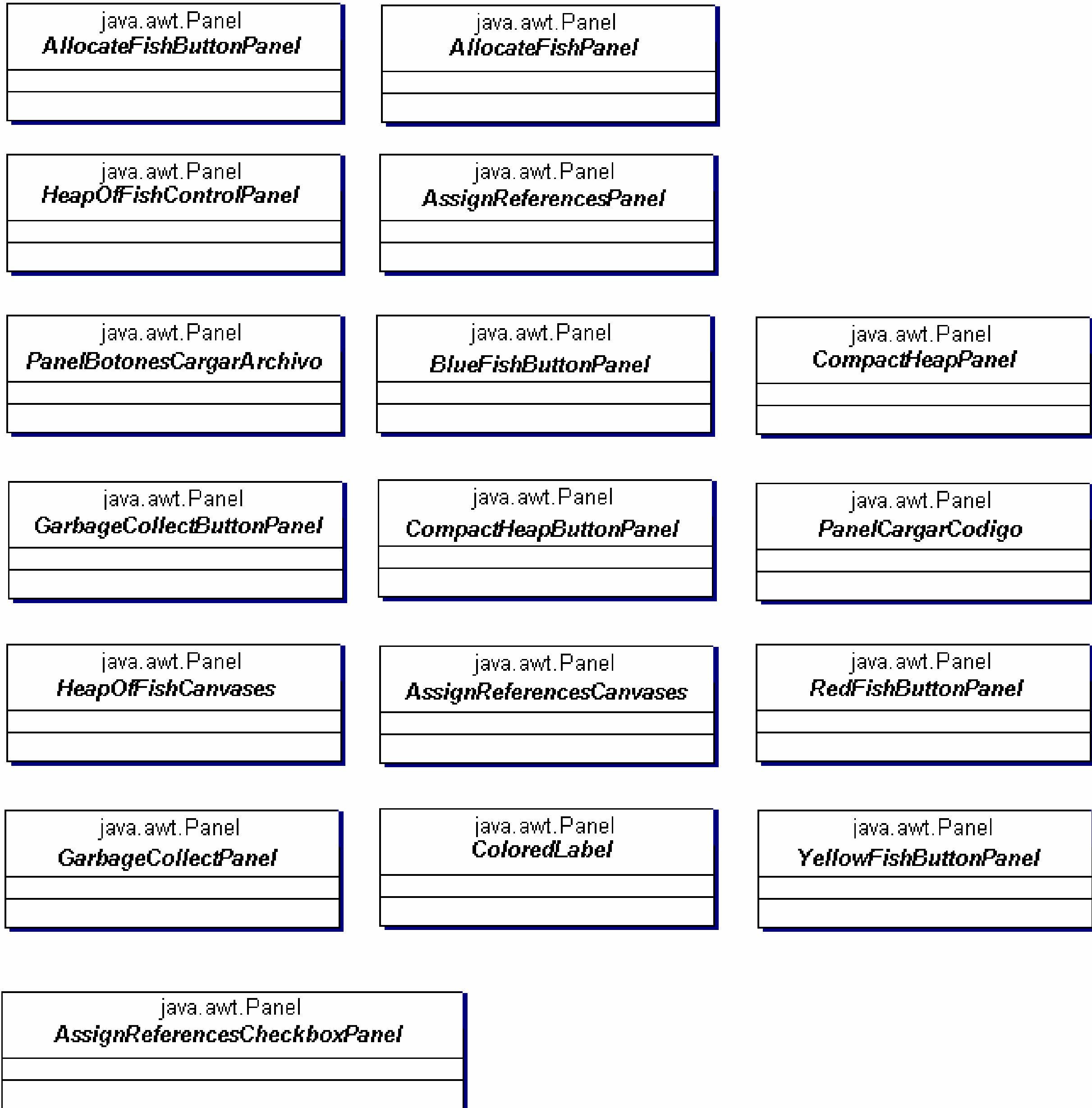
public synchronized void *asignarPecesAReferencias(Instruccion ins, MemoriaDeInstrucciones mem)*

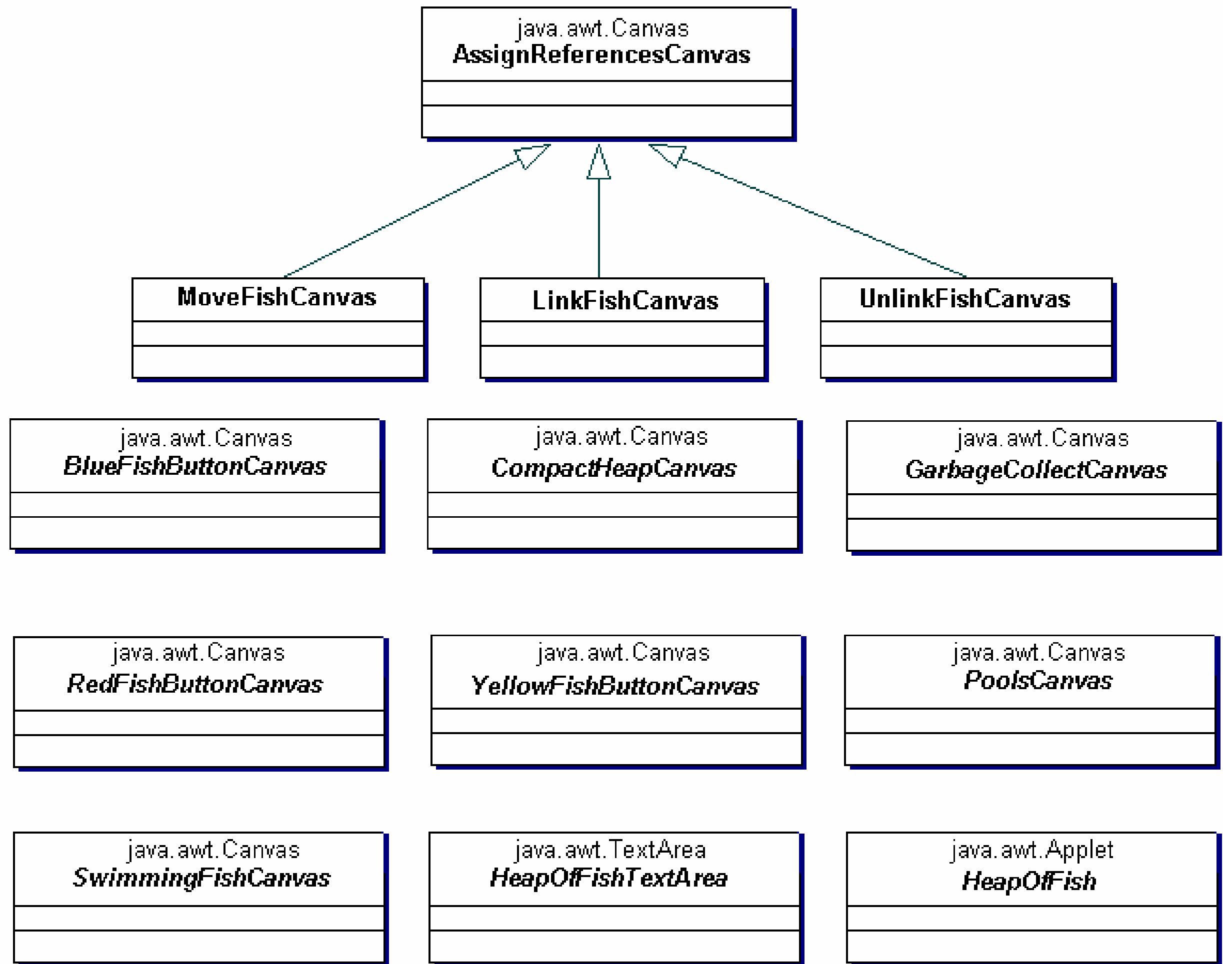
public synchronized void *anularReferencias(Instruccion ins)*

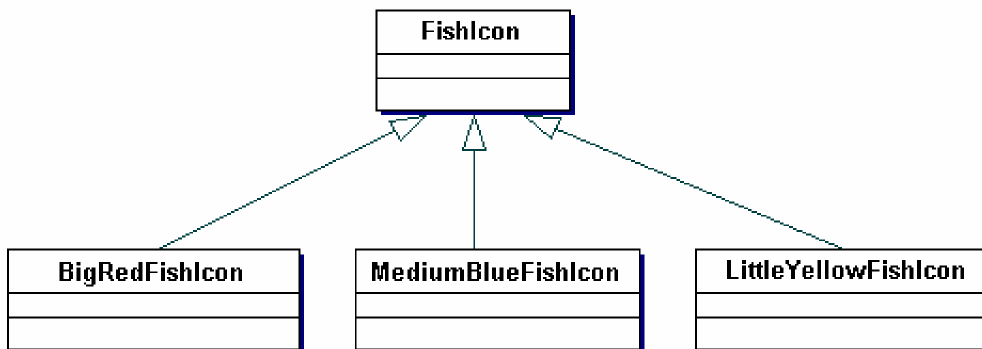
En las clases *MinorCollection* y *MajorCollection*:

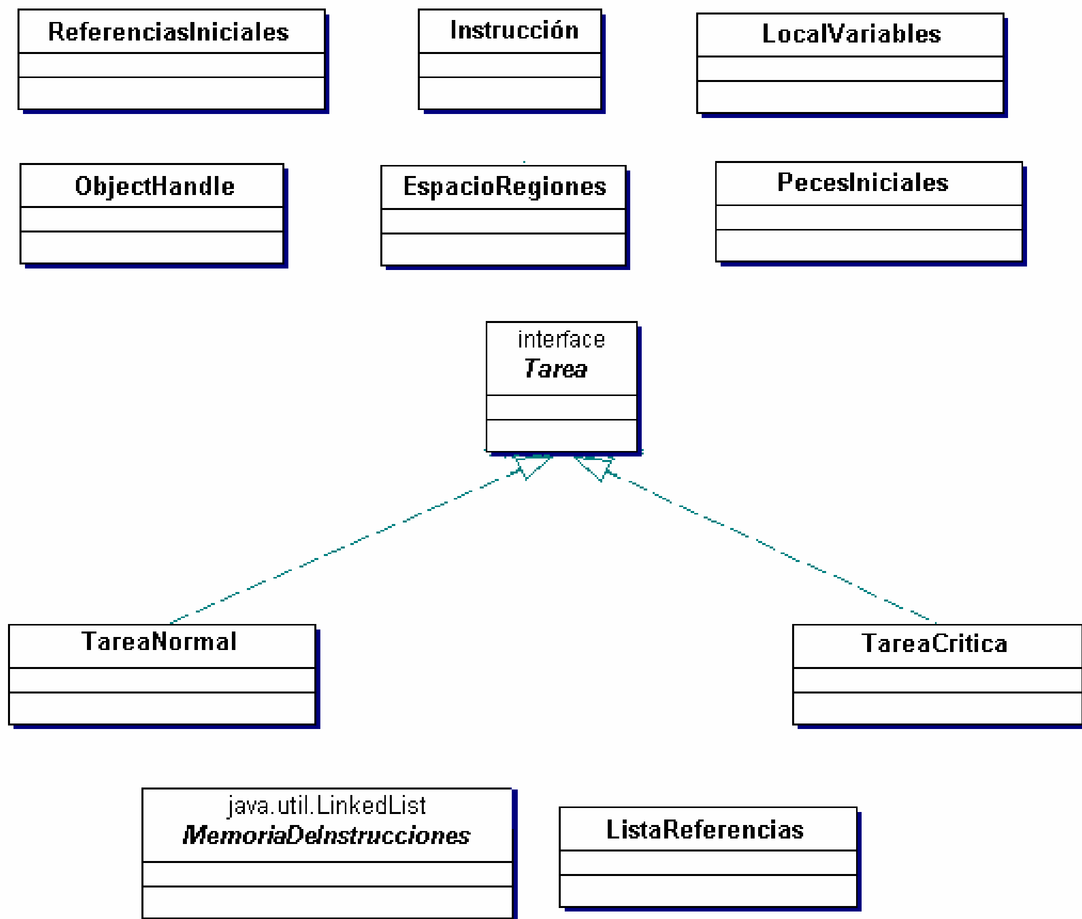
public synchronized void *nextGCStep()*

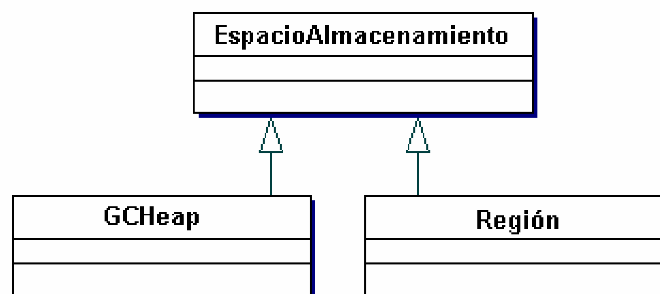
5.2.- Diagrama de clases

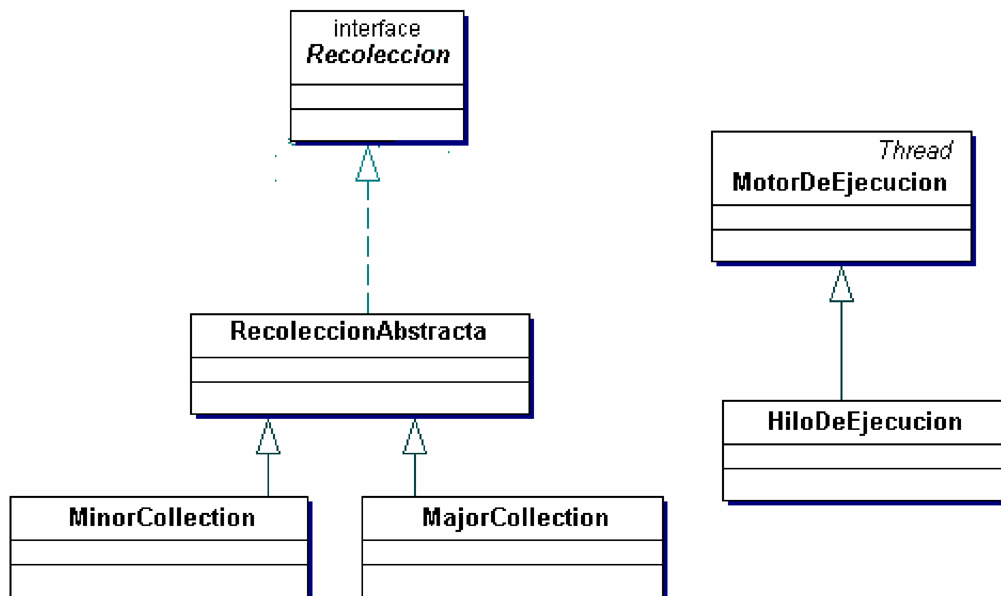



java.awt.Canvas
AssignReferencesCanvas
MoveFishCanvas
LinkFishCanvas
UnlinkFishCanvas
java.awt.Canvas
BlueFishButtonCanvas
java.awt.Canvas
CompactHeapCanvas
java.awt.Canvas
GarbageCollectCanvas
java.awt.Canvas
RedFishButtonCanvas
java.awt.Canvas
YellowFishButtonCanvas
java.awt.Canvas
PoolsCanvas
java.awt.Canvas
SwimmingFishCanvas
java.awt.TextArea
HeapOfFishTextArea
java.awt.Applet
HeapOfFish


FishIcon
BigRedFishIcon
MediumBlueFishIcon
LittleYellowFishIcon


ReferenciasIniciales
Instrucción
LocalVariables
ObjectHandle
EspacioRegiones
PecesIniciales
interface
Tarea
TareaNormal
TareaCritica
java.util.LinkedList
MemoriaDeInstrucciones
ListaReferencias


EspacioAlmacenamiento
GCHeap
Región



Las clases que han sido añadidas al proyecto son las siguientes:

- *PanelCargarCodigo*. Proporciona una interfaz para poder realizar la carga de archivos para ser lanzados a ejecución, pudiendo proporcionar el usuario parámetros como el propio archivo, el espacio donde va ser ejecutado, si es una tarea crítica y el tamaño de la región creada, en caso de que sea ejecutado en una región.
- *PanelBotonesCargarArchivo*. Contiene algunos de los componentes usados en el panel anterior.
- *Instruccion*. Clase que contiene el código de operación y los operandos, como si de una instrucción máquina se tratase.
- *MemoriaDeInstrucciones*. Clase donde se almacenan las instrucciones que van a ser ejecutadas.
- *ListaReferencias*. Lista que almacena referencias y el número de repeticiones que aparecen de cada una.
- *PecesIniciales*. Clase que contiene los métodos para crear objetos, que en este caso son peces.
- *ReferenciasIniciales*. Clase que contiene un objeto de tipo *LocalVariables*, que contiene las referencias del programa, y otro objeto de tipo *ListaReferencias*, que contiene las referencias desde objetos de la generación antigua a objetos de la generación nueva, y representa al conjunto raíz de variables.
- *EspacioAlmacenamiento*. Clase que se ha utilizado con el mismo código que la original *GCHeap*, ya que es el código común a sus hijos.
- *Region*. Clase que representa una región y tiene como particularidad que debe contener la dirección de memoria a partir de donde representa su espacio.

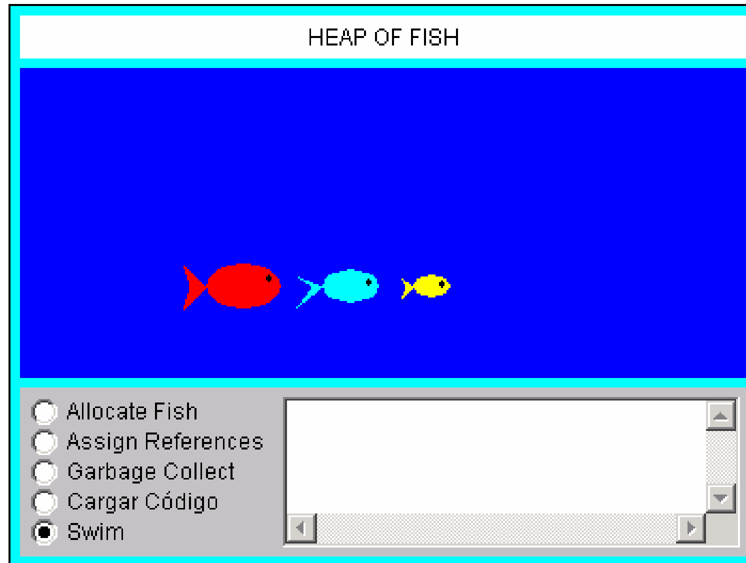
- *EspacioRegiones*. Clase que contiene las regiones que hayan sido creadas, así como las listas de referencias al heap para cada una de ellas.
- *Recoleccion*. Interface que representa cada uno de los tipos de recolección.
- *RecoleccionAbstracta*. Clase abstracta que contiene los atributos y métodos comunes a cada uno de los tipos de recolección.
- *MajorCollection*. Clase hija de *RecoleccionAbstracta* que implementa la major collection.
- *MinorCollection*. Clase hija de *RecoleccionAbstracta* que implementa la minor collection.
- *Tarea*. Interface que representa cada uno de los tipos de tarea en ejecución.
- *TareaNormal*. Clase que implementa la interface *Tarea* y contiene los métodos exclusivos de una tarea no crítica.
- *TareaCritica*. Clase que implementa la interface *Tarea* y contiene los métodos exclusivos de una tarea crítica.

Por otra parte, las principales modificaciones que han tenido lugar en las demás clases son las siguientes:

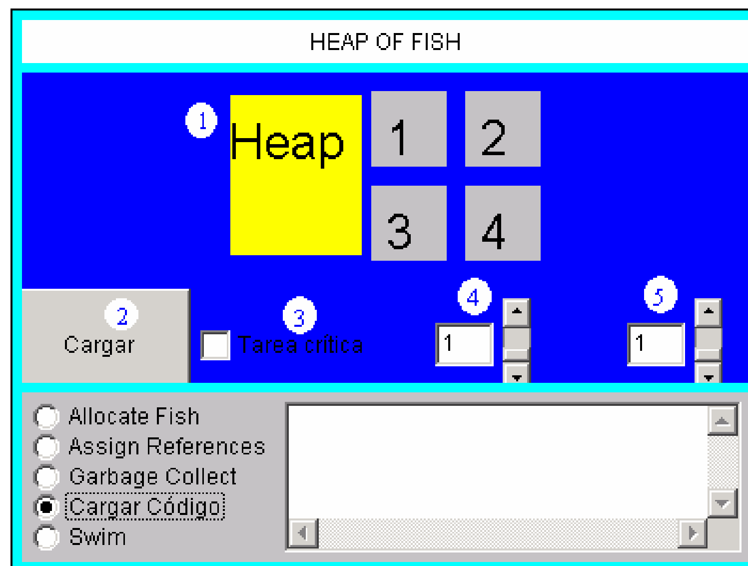
- *GCHeap*. Pasa a ser hija de la clase *EspacioAlmacenamiento*, que contiene la antigua implementación, pero, al tratarse de un algoritmo generacional, es necesario dividirlo en dos generaciones, contemplar en el constructor que, en lugar de un solo bloque de espacio libre, haya dos, de los cuales cada uno de ellos corresponde a una generación, e implementar un método que permita traspasar bloques de la generación nueva a la generación antigua.
- *LinkFishCanvas*. Se han añadido métodos para establecer y anular referencias por programa, en lugar de interactivamente y permitir al usuario mover los peces arrastrando el ratón.
- *AllocateFishPanel*. Se ha suprimido la interactividad en la creación de peces y la hace por programa y además ahora se representan las regiones que hayan sido creadas.
- *AssignReferencesCanvas*. Se ha modificado el método paint para que pinte los peces de las regiones y pinte los peces de la generación antigua de un color más oscuro.
- *GarbageCollectCanvas*. Se ha movido y modificado todo el código que ejecuta las acciones correspondientes a recolección a las clases *RecoleccionAbstracta*, *MajorCollection* y *MinorCollection* y se ha creado un sistema que determina el tipo de recolección que va ejecutada la próxima vez que el garbage collector sea ejecutado.
- *HeapOfFishCheckModePanel*. Se ha añadido la opción cargar código y se ha suprimido la opción compact heap.
- *ObjectHandle*. Se ha añadido un atributo que cuenta el número de minor collection a las que ha sobrevivido el objeto.
- *FishIcon*. Se ha añadido un atributo que contiene el color en que se pintan los peces cuando están en la generación antigua, así como los métodos para pintarlo.

5.3.- Funcionamiento de la aplicación

La aplicación, cuando se arranca, tiene este aspecto:



Como se había comentado anteriormente, la función *Compact Heap* ha sido suprimida y ha sido añadida la opción *Cargar Código*. Si hacemos clic en la opción *Cargar Código* se mostrará lo siguiente:



Este es el panel para lanzar un programa a ejecución. Un programa puede ser ejecutado en el heap o se puede crear una nueva región para ejecutarlo. Cuando se crea una región se puede elegir el tamaño y si la tarea que vamos a ejecutar en ella es o no crítica.

Los elementos que se muestran en este panel son los siguientes:

1) Muestra los espacios de almacenamiento, el que está seleccionado y en el caso de las regiones nos muestra el número que identifica a cada región y si ha sido creada o no. Los programas solamente pueden ser lanzados a ejecución en una región que no haya sido creada. El rectángulo amarillo etiquetado como Heap corresponde al heap y cada uno de los cuadrados grises etiquetados con un número corresponden a la región etiqueta con ese número. El espacio de almacenamiento seleccionado se resalta en magenta y las regiones se muestran en rojo cuando han sido creadas y la ejecución en las mismas no ha sido finalizada y en verde cuando han sido creadas y la ejecución ya ha finalizado. Para seleccionar un espacio basta con hacer clic en él y si hacemos clic en una región que está representada en verde, la región es destruida.

2) Botón cargar. Cuando tenemos seleccionado el espacio de almacenamiento donde va a ser ejecutado un programa, podemos seleccionar el archivo que queremos ejecutar con solamente pulsar este botón.

3) Tarea crítica. Cuando queremos ejecutar un programa en una región podemos seleccionar si el programa a ejecutar en dicha región es una tarea normal o una tarea crítica. La manera de seleccionarlo es mediante este cuadro de confirmación. Si está activado al pulsar el botón *Cargar*, la tarea ejecutada será crítica y si está desactivado será una tarea normal.

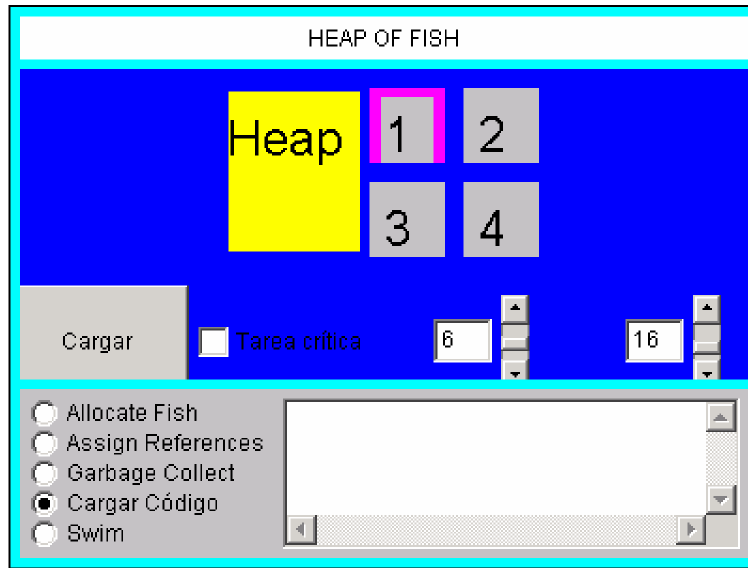
4) Tamaño de la zona de manejadores de una región. Como habíamos comentado anteriormente, cuando se crea una región, se puede especificar el tamaño de la misma. El tamaño que esté marcado en esta zona al pulsar el botón *Cargar* será el tamaño de la zona de manejadores de la región creada. El tamaño puede ser fijado tanto con las flechas como escribiéndolo directamente en el cuadro.

5) Tamaño de la zona de objetos de una región. Al igual que la zona de manejadores, aquí se fija el tamaño de la zona de objetos cuando se crea una región.

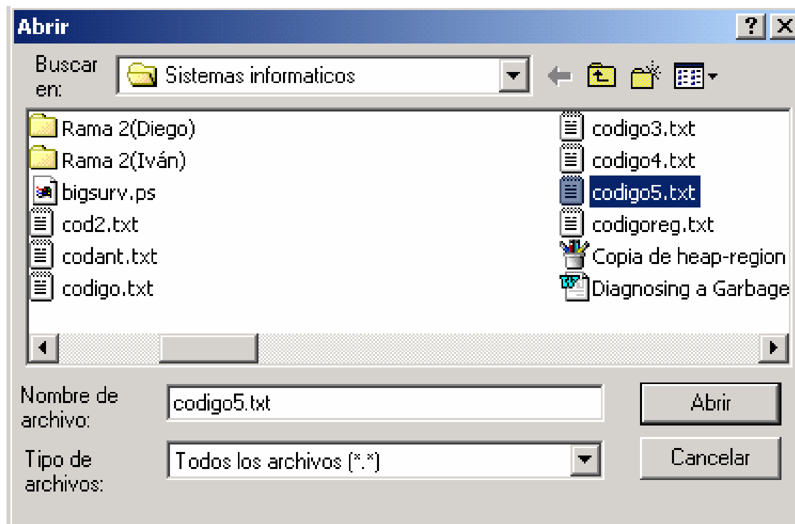
Ahora supongamos que queremos ejecutar en la región 1 el archivo *codigo5.txt* , cuyo contenido es el siguiente:

```
craz p6
craz p7
craz p8
cram p9
am p6,p7
asr p6,r2
am p7,p8
cm p6,p9
```

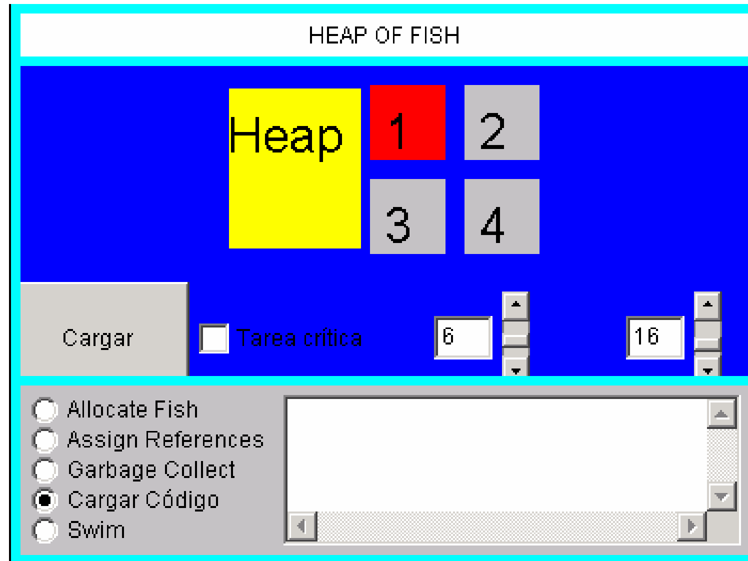
Para ejecutarlo en el región 1 tenemos que seleccionar la región 1, y elegir el tamaño de la región. Vamos a poner tamaño 6 para la zona de manejadores y tamaño 16 para la zona de objetos. Como queremos ejecutarlo como tarea normal, hay que cerciorarse que la casilla Región crítica aparece desactivada. Hecho esto, el applet tiene este aspecto:



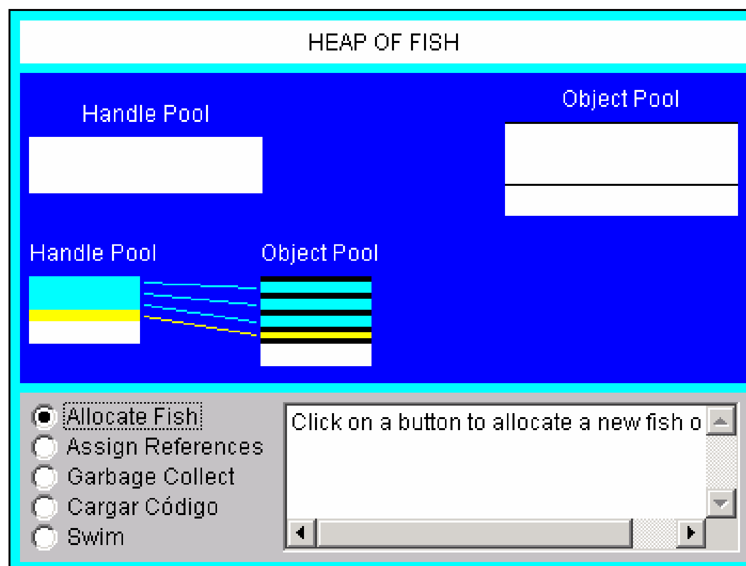
Una vez hecho esto, al pulsar el botón *Cargar* se desplegará el siguiente diálogo:



Se selecciona el archivo *codigo5.txt* y se pulsa el botón *Abrir*.



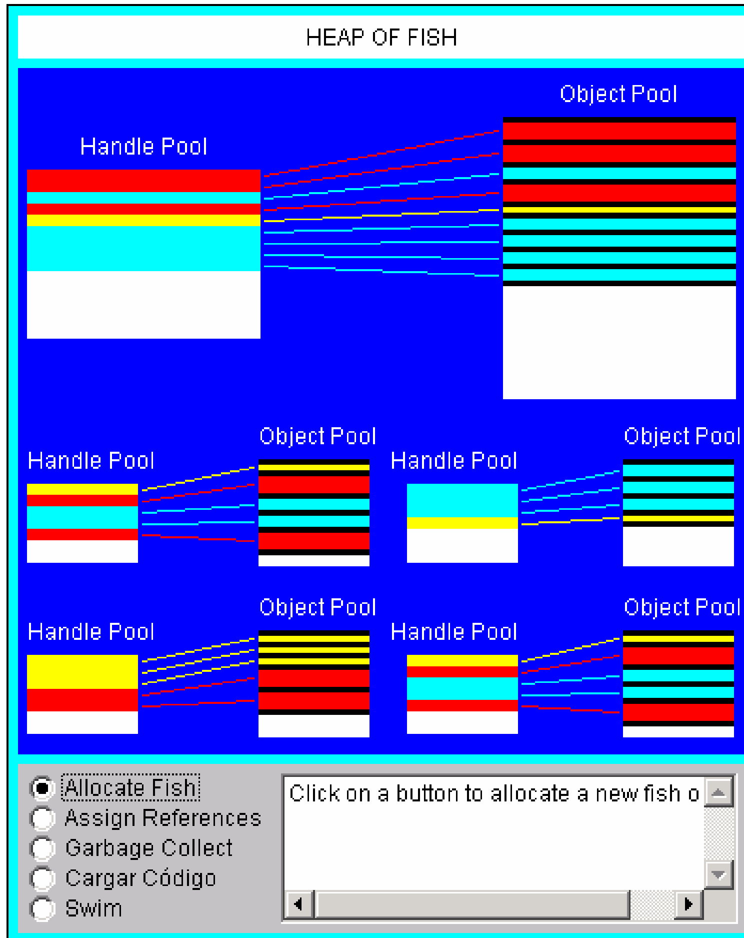
La región 1 aparece coloreada de rojo porque ha sido creada y la ejecución en la misma no ha finalizado aún. Observando las opciones *Allocate Fish* y *Asign References* se puede ver cómo va evolucionando la ejecución. Al llegar a su final, el panel *Allocate Fish* tiene este aspecto:



En la parte superior se representa el heap, que está vacío, ya que solamente hemos ejecutado una tarea en la región. Si existe alguna región, se representan en la parte inferior solamente las regiones existentes. Tanto en el heap como en las regiones el rectángulo etiquetado como *Handle Pool* se utiliza para representar la zona de manejadores y el rectángulo etiquetado como *Object Pool* se utiliza para representar la

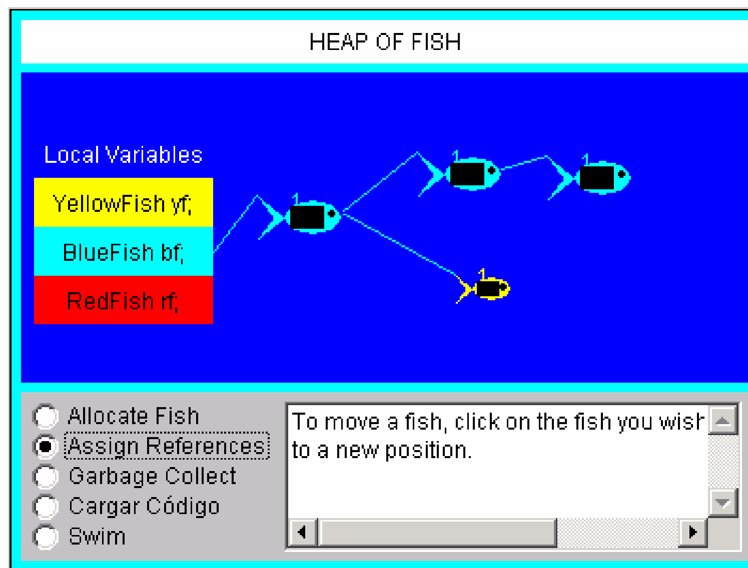
zona de objetos. Las líneas que aparecen entre ambos nos indican que objeto corresponde a que manejador.

En el caso de que haya ejecución en el heap y en las cuatro regiones, el applet se presentará de la siguiente manera:



En la parte superior se muestra el heap, en la fila superior de la parte inferior se muestra el contenido de las regiones 1 y 2 respectivamente y en la fila inferior el contenido de las filas 3 y 4. Cada región existente aparece siempre en su posición, existan o no las demás regiones. Las regiones no existentes no se muestran en el applet.

El panel *Assign References* tiene este aspecto, también al finalizar la ejecución:

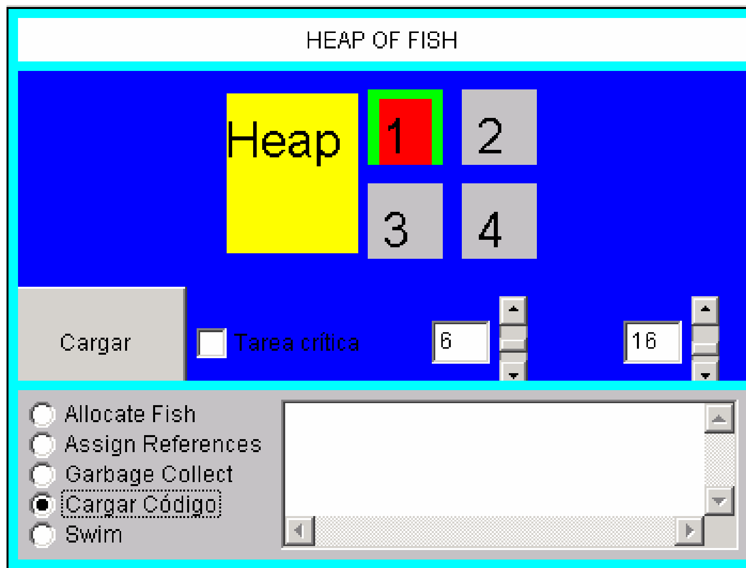


Aquí se muestra tanto el pez al que apunta la variable local *bf*, como las referencias que hay entre los peces. Como ya habíamos comentado anteriormente, si el pez *a* es un atributo del pez *b*, habrá una línea desde la cabeza del pez *b* hasta la cola del pez *a*. En esta imagen se muestra que la variable *bf* apunta a un pez azul, y de este pez salen dos atributos, que son el pez azul que está justo debajo de él y el pez amarillo. El tercer pez azul es un atributo del segundo.

Observese que en los peces aparece dibujado un rectángulo negro y encima aparece el número *1*. Esta es la manera en que se muestran los peces que se crean en una región, con un rectángulo negro en su interior y un número que indica el número de la región donde se ha creado. Cuando se crean peces en el heap, no aparece nada de esto.

En este panel es posible mover los peces y así verlo con más claridad. Para hacer esto no hay más que arrastrar los peces uno a uno hasta la posición deseada. A diferencia del modelo original ya no hay las tres subopciones que había anteriormente, ya que la creación y destrucción de referencias no se hace de manera interactiva. Sin embargo, el usuario sí que puede mover los peces.

La opción *Cargar Código*, una vez finalizada la ejecución, tendrá el siguiente aspecto:

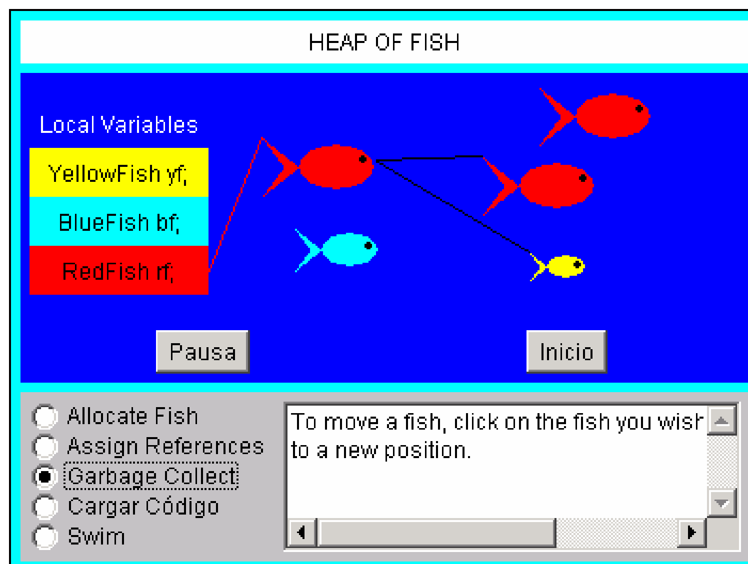


Como puede observarse, la región *1* aparece remarcada en verde, ya que la ejecución en dicha región ha sido finalizada. Si hacemos clic en este cuadrado, la región será destruida.

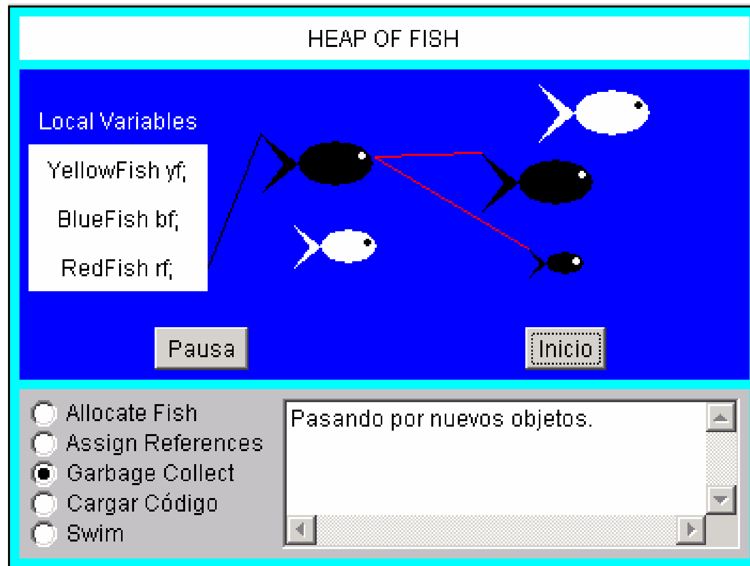
Para hacer una prueba del garbage collector, vamos a utilizar el siguiente programa:

```
crr p1
crr p2
craz p3
crr p4
cram p5
am p1,p2
asr p1,r3
am p1,p4
sn p1,p5
```

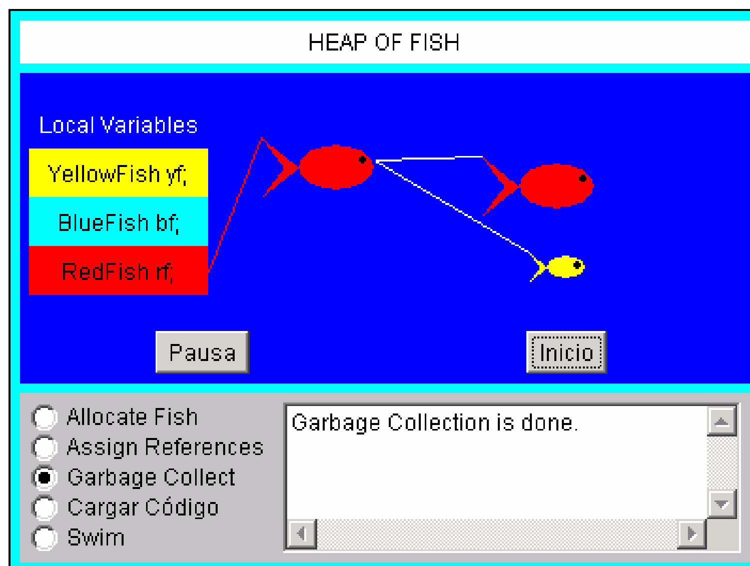
Cargamos este programa en el heap y nos vamos a la opción *Garbage Collect*, apareciendo lo siguiente:



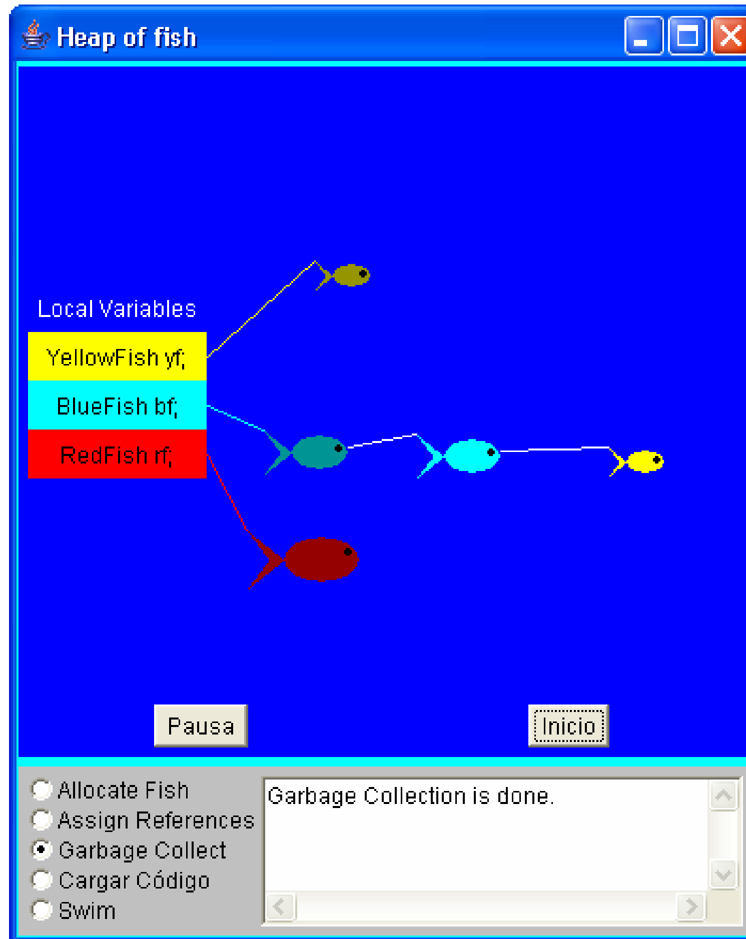
Aquí aparecen los peces con la misma representación que en la opción *Assign References*. Si pulsamos el botón *Inicio*, el garbage collector ejecutará el tipo de recolección que corresponda, major collection o minor collection. En ambos tipos de recolección se muestran todos los objetos contenidos en el heap, pero en la minor collection no se recorren los de la generación antigua. En el panel *Garbage Collector* nunca se muestran los objetos situados en regiones. El proceso se va mostrando coloreando los peces del color correspondiente y el elemento (variable o pez) donde se sitúa el garbage collector en cada momento aparece mostrado de color magenta.



Esto es el aspecto del garbage collector justo antes de empezar a liberar la memoria correspondiente a estos objetos. Ya que hay dos objetos coloreados de blanco y además están situados en la generación nueva (lo que hace que no dependa de si estamos antes minor collection o major collection), van a ser liberados, quedando así:



En el caso en que haya objetos que promocionen a la generación antigua, estos aparecerán mostrados en un color más oscuro, tanto en el panel *Garbage Collect* como en el panel *Assign References*.



Como se puede observar, los peces que están más cerca de las variables están coloreados en un color más oscuro que los demás. Esto se debe a que estos objetos están situados en la generación antigua.

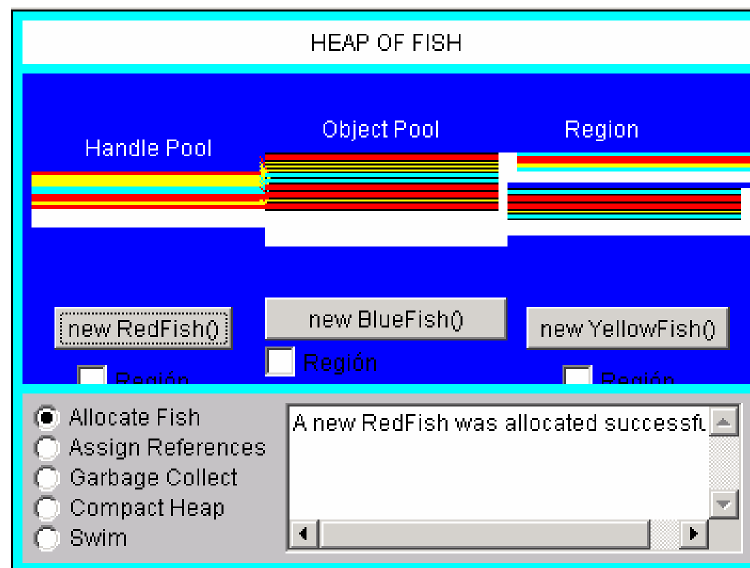
5.4.- Evolución del prototipo

5.4.1.- Primera versión

En una primera implementación, nuestro sistema dispone de una sola región. Esta región siempre existe, ya que se crea al inicio de la aplicación y su tamaño es de 8 unidades para los manejadores y de 25 unidades para los propios objetos

Para implementar esta región se declara una nueva clase llamada *Region*, que es hija de la clase *GCHeap*. La dificultad introducida consiste en que ahora, al recolectar la basura en el heap hay que chequear las referencias desde la región al heap antes de recolectar la basura. Esto implica que además de extender la clase *GCHeap* hay que añadir un atributo nuevo, que es una tabla de punteros. Esta tabla tiene el mismo tamaño que la zona de objetos de la región y es un array que contiene un 0 si el puntero situado en esa posición de la región no apunta a ningún objeto del heap y la dirección del objeto del heap en caso contrario. Como en la región no hay recolección de basura, no es posible recolectar en el heap aquellos objetos cuyas referencias desde la región procedan desde objetos muertos de la región, pero dichos objetos serán liberados cuando se vacíe toda la región.

Para realizar esta implementación ha sido necesario modificar numerosos puntos de la aplicación. Uno de ellos que ha sido modificado es la interfaz, porque tiene que permitir crear nuevos objetos en la región y mostrarlos. Esta modificación se ha realizado en la parte de la interfaz referida a *AllocateFish* y ha consistido en añadir un espacio para representar tanto los manejadores como los objetos de la región y objetos de tipo *CheckBox* para especificar que un objeto se debe colocar en la región. En la figura se ilustra como se ha hecho este cambio:



En la figura casi no se aprecia, pero hay una línea en la región, del mismo tono azul que el fondo del applet que divide la zona de manejadores y la zona de objetos. También se pueden observar cuadros de confirmación etiquetados con *Región* debajo de los botones para crear peces. Para cada color, si el cuadro correspondiente está activado el pez se crea en la región y si no se crea en el heap.

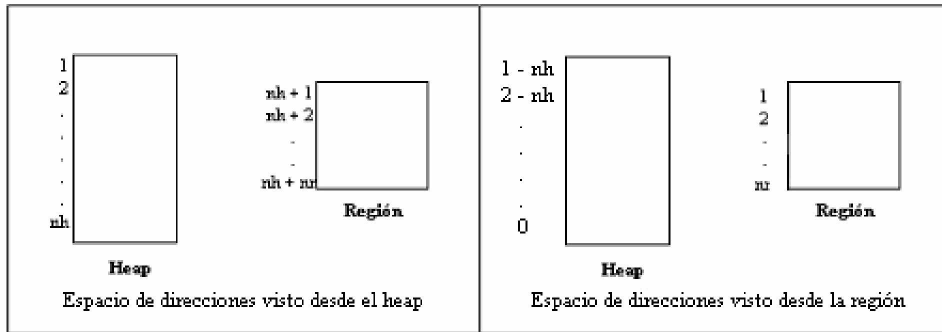
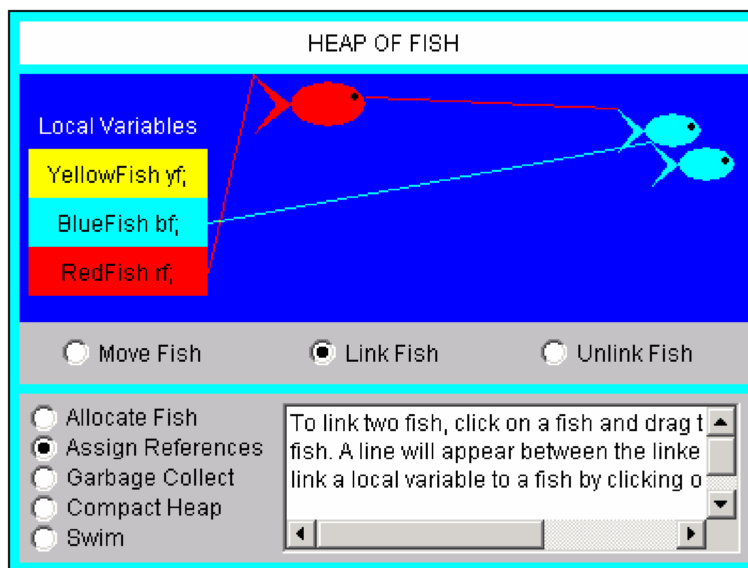


Fig. 14: Direccionamiento utilizado en la primera versión

En cuanto a la asignación de referencias, es necesario tener un tipo de asignación que permita hacer referencias desde el heap hacia la región (en la primera versión no estaba aún contemplado que esto no está permitido) y viceversa y en este caso consiste en una asignación que nos da la dirección relativa dentro del Heap o región y con un desplazamiento positivo en el caso del heap y negativo en caso de la región igual al tamaño del Heap. Por ejemplo, supongamos que estamos trabajando con un heap de tamaño 40. Si un objeto en el heap situado en la dirección 3 referencia a un objeto del heap situado en la posición 23, la referencia contenida en dicho objeto será 23, pero si este mismo objeto referencia a un objeto de la región situado en la posición 7 de la misma, la referencia contenida será 47 = 7(dirección dentro de la región) + 40 (tamaño del heap). Si este objeto, situado en la posición 7 de la región, referencia al objeto contenido en la posición 12 de la región contendrá la referencia 12, ya que estamos dentro del mismo sistema de almacenamiento, pero si referencia al objeto contenido en la posición 23 del heap, la referencia contenida será –17 = 23(dirección dentro del heap) – 40 (tamaño del heap).



Esto es un ejemplo de un pez rojo perteneciente al heap apuntando a un pez azul de la región. Dado que tanto el pez rojo como el azul son los primeros en la dirección de

memoria donde están alojados, la variable *rf* contendrá el valor 1 y el atributo *myLunch* del pez rojo contendrá el valor *nh + 1*, siendo *th* el tamaño de almacenamiento de la zona de objetos en el heap. El segundo pez azul ocupa la segunda posición en el heap, por tanto la variable *bf* contendrá *nh + 2*, ya que este objeto está asociado al segundo manejador de la región.

El código origen es muy poco escalable, y por tanto esta modificación supuso muchas dificultades y hubo que modificar código en muchos puntos. Por ejemplo, cada vez que se establecía un puntero entre dos peces había que comprobar si el pez origen y destino pertenecían al Heap o la región y realizar acciones distintas para cada una de las combinaciones posibles. Otro de los problemas presentados ha sido la necesidad de distinguir cuando una dirección se refería a la región y cuando se refería al Heap y se ha controlado mediante las variables booleanas *punteroDesdeRegion* y *punteroARegion* y cuando se realizado la inserción en memoria de la dirección ya se aplica el criterio anteriormente indicado.

El funcionamiento del garbage collector en este caso es el siguiente:

- En primer lugar chequea las 3 variables locales existentes, detecta y marca los objetos que están referenciados, tanto directamente como indirectamente desde ellas.
- Por último, chequea la tabla de referencias de la región y, por cada valor que encuentra distinto de 0, detecta y marca también los objetos referenciados tanto directamente como indirectamente teniendo en cuenta las direcciones de memoria contenidas en esos objetos.

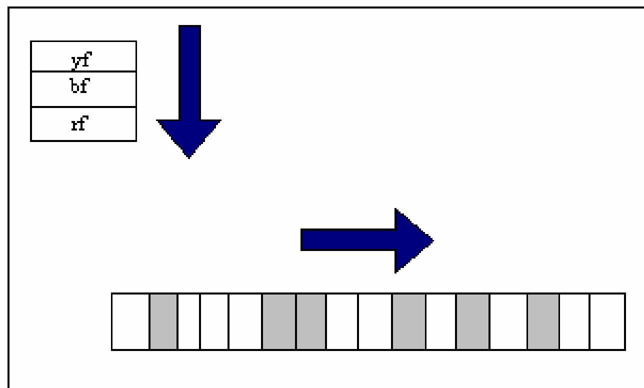


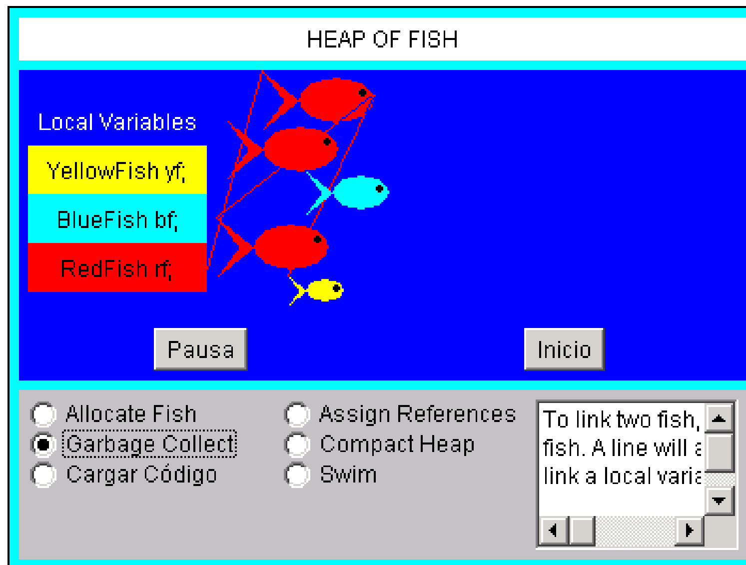
Fig. 15: Recorrido del garbage collector en la primera versión

Es decir, sabiendo que las posiciones del array sombreadas contienen algún valor del heap, el chequeo se hace en este orden.

5.4.2.- Segunda versión

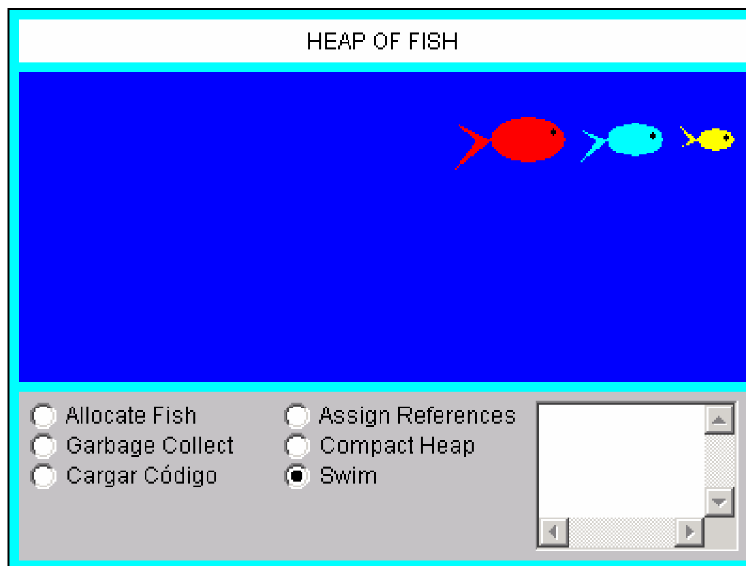
En la primera versión, la creación de objetos y asignación de referencias se hacía de manera interactiva. Esto se aleja del caso real, porque en la realidad, la aplicación y el garbage collector se ejecutan concurrentemente. Por tanto, necesitamos un modelo que nos permita ver la aplicación de esta manera. El modelo pensado más parecido al real es aquel que ejecute un programa y el garbage collector concurrentemente. Por ello, en esta versión, las acciones de crear objetos y asignar referencias se especifica en un archivo y esas acciones son ejecutadas. Esta versión es independiente de la anterior y, por tanto el único espacio de almacenamiento es el heap. Para que las instrucciones sean ejecutadas, tienen que estar en memoria, por lo que se ha añadido una nueva clase, la clase *MemoriaDeInstrucciones*, que es una cola de instrucciones de la que si el primer elemento corresponde a la creación de un objeto es leída, extraída y ejecutada por el panel *Allocate Fish* y si corresponde a la asignación o anulación de referencias, es ejecutada por el panel *Assign References*.

Otra modificación realizada fue la transformación del panel del garbage collector para permitir una ejecución ininterrumpida del mismo. Ahora, en lugar de tener el usuario que ir pasando de un paso al siguiente manualmente, lo hace el programa por sí solo.

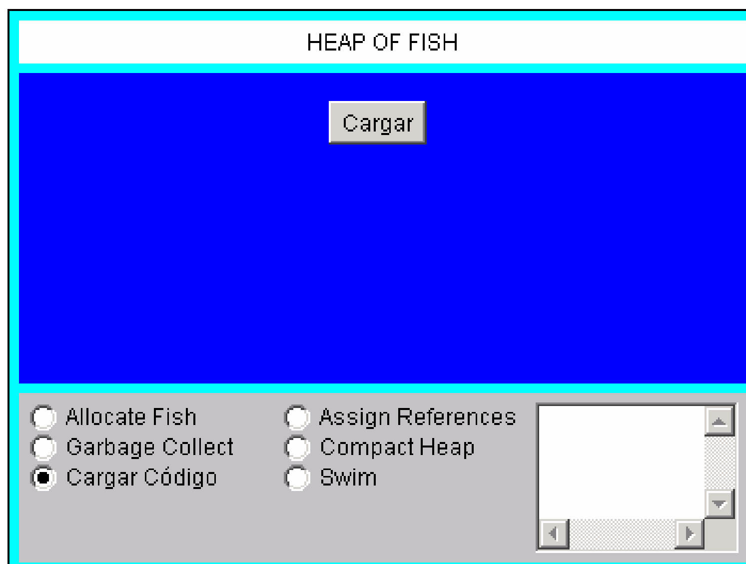


Para empezar a ejecutar el garbage collector hay que pulsar el botón Inicio. Esto se puede hacer en cualquier momento, ya que se ejecuta en un hilo independiente. En cuanto se pulse este botón, el garbage collector empezará a ejecutarse.

En esta versión la aplicación tiene el aspecto que aparece en la figura:



Obsérvese que ahora, además de las funciones habituales aparece una nueva función que es la de cargar código. Ahí es donde se puede seleccionar el archivo que va a ser ejecutado. Hacemos clic en el botón correspondiente y saldrá la siguiente pantalla:

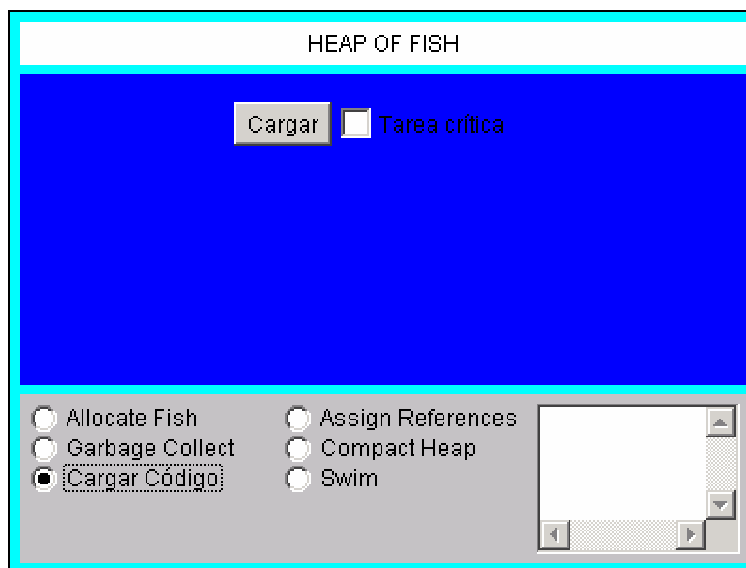


En esta versión hay, por lo tanto, 4 hilos en ejecución: uno para el panel *Allocate Fish*, otro para el panel *Asign References*, otro para la carga de archivos y el último para el garbage collector. Se podría decir que los 3 primeros hilos están sincronizados por la memoria de instrucciones y que el hilo de la carga de archivos produce los elementos (instrucciones) que van a consumir los otros 2 hilos. El hilo de carga de archivos ejecuta libremente y los otros 2 consumen espera activa mientras esperan a recibir una instrucción que corresponda ser ejecutada por ellos.

5.4.3.- Tercera versión

El siguiente paso es integrar las dos últimas partes, es decir el sistema basado en regiones y el sistema para lanzar instrucciones a ejecución. En esta integración cabe destacar la incorporación de un motor de ejecución y la distinción entre tarea normal y tarea crítica a la hora de ejecutar un programa.

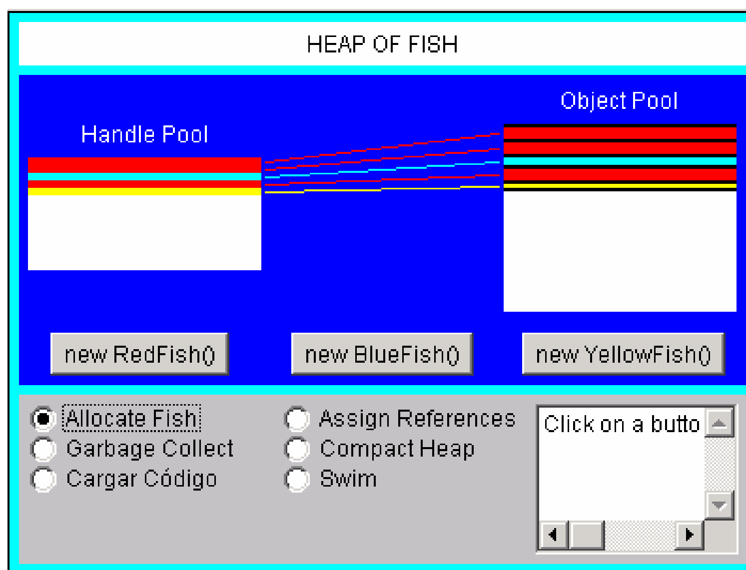
Como se había indicado anteriormente, las tareas críticas van a ser ejecutadas en la región. Por tanto es necesario diferenciar las acciones producidas cuando la tarea es normal y cuando la tarea es crítica. Esto se hace definiendo la interfaz *Tarea* con las implementaciones *TareaNormal* y *TareaCritica* , cada una con su código correspondiente. El código aquí puesto es el exclusivo de cada tipo de tarea y es invocado cuando es necesario que sea ejecutado. Una de ellas es chequear si se desasigna un objeto del Heap que es apuntado desde uno de la región. Cuando esto sucede en una tarea normal es necesario actualizar la tabla de punteros, pero si esto sucede en una tarea crítica no se hace nada, ya que no hay realizar ninguna acción que retrase la ejecución de la tarea en sí y cuando acabe su ejecución se liberará la memoria de la región y se reseteará la tabla de referencias y por tanto se liberará la memoria del Heap apuntada exclusivamente por la región.



Como se puede observar en la figura ahora tenemos la opción *Tarea crítica*. Si está este cuadro activado la tarea a ejecutar será crítica, será ejecutada en la región y no será interrumpida por el garbage collector y si está desactivado, la tarea será ejecutada en el heap y puede ser interrumpida por el garbage collector.

En esta versión el garbage collector utilizado es el mismo que el de la segunda versión y, por tanto no hace ningún tipo de chequeo en las referencias que se establezcan desde la región al heap pero, sin embargo, permite la ejecución de tareas críticas sin ser interrumpidas por el garbage collector.

Solamente se muestran los objetos creados en el heap. Si los objetos son creados en la región (tarea crítica) no se muestran en esta versión. Al igual que aquí, los objetos tampoco son mostrados en la opción *Assign References* ni en el garbage collector, ya que la región no tiene recolector de basura y, por tanto no se puede mostrar su funcionamiento.



5.4.3.1.- Sistema de direccionamiento

Anteriormente estábamos utilizando un sistema de direccionamiento en el que en el heap las referencias a la región se representaban sumando el tamaño del espacio para almacenar objetos en el heap al valor propio en la región. Y en el caso de que el direccionamiento en la región sea al heap se restaba esta cantidad. Ahora el sistema de direccionamiento es global y sirve tanto para el heap como para la región. Se hace de la siguiente manera:

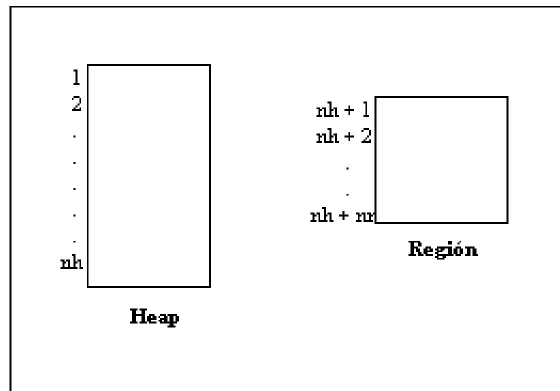


Fig. 16: Sistema de direccionamiento en la tercera versión

Es decir el espacio de direcciones [1,nh] corresponde al heap y el espacio [nh + 1, nh + nr] correspondiente a la región, indistintamente desde donde se haga la referencia. Es decir, si por ejemplo el objeto que ocupa la posición 1 en la región tiene una referencia al objeto que está en la posición 10, el primero contendrá el valor nh + 10, no simplemente 10 como en la versión anterior. Si este último objeto tiene a su vez una referencia al objeto en la posición 4 del heap, contendrá el valor 4.

5.4.3.2.- Incorporación de un motor de ejecución

En la versión anterior las partes del applet *Allocate Fish* y *Asign References* eran las que se encargaban de leer las instrucciones de la memoria de instrucciones y de ejecutarlas. Esto tiene como inconveniente que es más difícil tener información sobre modos de ejecución. Por está razón se ha creado el motor de ejecución.

El motor de ejecución es una clase que lee las instrucciones desde memoria y manda ejecutar cada instrucción a la parte del applet correspondiente. También contiene información sobre si la tarea que se está ejecutando es o no crítica.

Con esta nueva implementación quitamos un hilo, ya que las instrucciones a ejecutar en *Allocate Fish* y *Asign References* lo hacen en el mismo hilo. Tenemos, por tanto, un hilo para la carga, otro para la ejecución y otro para el garbage collector.

5.5.- Pruebas realizadas

En esta aplicación, al igual que en la mayoría de ellas, es necesario realizar diversas pruebas para verificar diversos puntos que debe cumplir la aplicación. Estos puntos son:

- Comprobación de que los objetos apuntados exclusivamente desde la región son chequeados por el garbage collector. Cuando esta opción fue incorporada por primera vez, aún no se creaban objetos y se establecían referencias desde un archivo, sino que se hacía de manera interactiva, por lo tanto para hacer esta comprobación basta con crear un objeto en la región que apunte a un árbol de objetos donde todos estén situados en el heap y, a la vez no estén apuntados desde ningún otro sitio. Es importante que la profundidad del árbol sea mayor que 1, porque así se chequea que el algoritmo examina todo el árbol y no solamente los objetos adyacentes al situado en la región. Cuando se mostraban los objetos creados en la región, se pudieron realizar pruebas cargando los archivos. La prueba que se realizó fue esta:
  En primer lugar se lanzó a ejecución en el heap este código:

```
craz p6
craz p7
craz p8
cram p9
am p6,p7
asr p6,r2
am p7,p8
cm p6,p9
```

  Posteriormente se lanzó a ejecución en la región el siguiente código:

```
craz p1
asr p1,r2
ami #51,#1
```

  Dado que la referencia r2 (*bf*) se sustituye, los objetos creados en primer lugar son exclusivamente apuntados desde objetos situados en la región. Por lo tanto, estamos en un buen momento para ejecutar el garbage collector y se comprobará que el objeto de la región no es recorrido por el garbage collector, pero si lo es su descendencia presente en el heap.

- Comprobación de que en la memoria de instrucciones se almacena correctamente la secuencia de instrucciones leída del archivo. Para hacer esta prueba hay que introducir código adicional. Este código contiene la creación de un iterador sobre la memoria de instrucciones que recorre la misma y nos va dando las instrucciones una a una. Dado que el número de operandos varía según la instrucción de que se trate, es necesario hacer un bucle que recorra los operandos uno a uno. El código debe mostrar por pantalla el código de operación y el valor de cada uno de los operandos.

- Comprobación de que el archivo se ejecuta correctamente. Para hacer esto basta con cargar el archivo e ir observando en los paneles *Allocate Fish* y *Assign References* si se ejecuta cada una de las instrucciones contenida en el archivo y en el mismo orden que están escritas. Para hacer una buena prueba hay que hacer creación de peces,

asignaciones, cambios y anulaciones de referencias. Para hacer esta prueba se utilizó este código:

```
crr p1
crr p2
craz p3
crr p4
cram p5
asr p5,r1
am p1,p2
asr p1,r3
am p1,p4
sn p1,p5
nr r1
```

- Comprobación de la distinción entre tareas críticas y no críticas. Dado que cuando esta característica fue añadida las tareas críticas eran ejecutadas en la región y no había representación de la región, la única comprobación que se podía hacer era ejecutar el mismo programa como tarea crítica y como tarea no crítica y comprobar que en la tarea no crítica se iba mostrando la ejecución del programa y en la tarea crítica no. No se hizo ninguna otra comprobación hasta que las regiones estaban representadas, momento donde era posible hacer mejores pruebas.

- Comprobación de que se chequean las referencias de objetos desde la generación antigua a la generación nueva en la minor collection. Inicialmente, la zona de objetos del heap estaba dividida de manera que un tercio correspondía a la generación nueva y dos tercios a la generación antigua y el espacio libre consistía en un único bloque. Por lo tanto, en el momento en que la generación antigua se llenaba de objetos, los objetos creados a partir de entonces eran alojados en la generación nueva. Para hacer esta comprobación basta con llenar la generación nueva de objetos, momento a partir del cual empezamos a crear objetos en la generación antigua. Se ponen instrucciones para establecer referencias desde un objeto de la generación antigua a un árbol de objetos de la generación nueva y se ejecuta el garbage collector de manera que haga una minor collection. De esta manera se observa si realmente se realiza ese chequeo.

- Comprobación de que en la minor collection el garbage collector detiene la exploración de objetos cuando se encuentra un objeto alojado en la generación antigua. Cuando se hizo esta prueba, al igual que en el caso anterior, la zona de objetos de la generación nueva era un tercio del total y la memoria era inicialmente un único bloque libre, por lo que cuando se llena la generación nueva de objetos, los objetos que se creen a partir de ese momento, se alojarán en la generación antigua. Por tanto, para hacer esta comprobación hay que crear un programa que cree objetos en ambas generaciones y tenga referencias desde la generación nueva a la generación antigua, ejecutar el garbage collector de manera que haga una minor collection y observar que efectivamente la exploración se detiene en el objeto cuyos hijos están todos en la generación antigua.

- Comprobación de que los objetos se crean correctamente en las regiones. Ahora, dado que se muestra el contenido de las regiones, para comprobar esto basta con ejecutar el código en alguna de las regiones y observar en el panel *Allocate Fish* como se van creando los objetos. Es importante que el tamaño de la región creada sea mayor o igual al espacio que ocupan los objetos.

- Comprobación de que la asignación de direcciones en las regiones se hace correctamente. Para hacer esta comprobación hay que poner en el constructor de la región un mensaje a la consola con la dirección de comienzo y creando varias regiones se puede observar si las direcciones se asignan correctamente.

- Comprobación de que la ejecución en las regiones finaliza y las regiones se destruyen correctamente. Para hacer esta comprobación basta con lanzar una ejecución en alguna región y comprobar que cuando finaliza (al pasar un tiempo), el cuadrado se remarca en verde y al hacer clic se destruye, desapareciendo todos los objetos que se habían creado en ella.

- Comprobación de que los objetos promocionan correctamente. Para hacer esta comprobación hay que crear un programa donde haya objetos vivos que se mantengan al menos hasta que lleguen a promocionar, reducir el número de veces sobrevividas para promocionar (en nuestro caso se ha reducido a 3) y ejecutar sucesivamente el garbage collector hasta que se alcance el número de veces para promocionar (solamente afecta a las minor collection). Cuando los objetos promocionen se podrá observar en el panel Allocate Fish que aparecen más abajo y cuando se ejecute la próxima minor collection no se explorarán estos objetos. Una vez implementada la promoción de objetos podemos volver a comprobar el chequeo de objetos nuevos apuntados desde objetos antiguos lanzando a ejecución justo después de la promoción un programa que cree objetos nuevos y asigne referencias desde los antiguos.

- Comprobación de que cuando se hace una asignación donde el padre está coloreado de negro y el hijo está blanco se colorea el hijo de gris y justo antes de liberar el garbage se chequea la descendencia del hijo. Para hacer esta comprobación hay que aprovechar la concurrencia entre el garbage collector y la aplicación y lanzar el garbage collector antes de que la aplicación finalice su ejecución. Se observará que, en el momento que la aplicación haga la asignación, en el garbage collector se coloreará el nodo hijo de color gris y, justo antes de recolectar el garbage, se hace una exploración desde los nodos coloreados de gris.

```
cram p1
asr p1,r1
cram p2
am p1,p2
cram p3
am p2,p3
crr p4
asr p4,r3
crr p5
am p4,p5
sn p5,p1
```

Se podría utilizar este código para probar esto, puesto que en este código hay varias asignaciones. Si el garbage collector termina de pasar por un objeto antes de que el hijo haya sido asignado, se observará que se colorea el hijo de gris y al final se chequea su descendencia.

- Comprobación de que las nuevas instrucciones *amdi*, *cmdi* y *sndi* actúan de manera correcta y que cuando estas instrucciones establecen referencias de alguna región al heap se chequean correctamente los objetos del heap que son apuntados y que cuando se ejecute en otra región creada posteriormente un mismo código que contenga alguna de estas 3 instrucciones (no debe contener *ami*, *cmi* ni *sni* y el contenido del heap debe haberse mantenido intacto) las referencias establecidas son las mismas. Para hacer esta prueba, primero hay que ejecutar en el heap el siguiente código:

```
crr p2
craz p3
cram p4
cram p5
cm p3,p5
craz p6
cm p2,p6
craz p7
am p6,p7
```

  Después de lanzar a ejecución este código en el heap hay que lanzar en una de las regiones el siguiente código:

```
crr p1
asr p1,r3
amdi p1,#1
cmdi p1,#2
sndi p1,#3
```

  A continuación, se lanza a ejecución el garbage collector y se observa que ninguno de los objetos del heap son recolectados porque están apuntados desde el objeto *p1*, situado en la región. Ejecutando reiteradamente el garbage collector, se detectó un error de ejecución. Cuando se ejecutaba la major collection, no se chequeaban los punteros de la región al heap y los objetos eran recolectados. Esto se debía a que, en el caso de que la variable *rf* no apuntase a un objeto del heap, nos saltábamos los estados del garbage collector correspondientes al chequeo de objetos del heap apuntados desde la región cuando esos estados deben ser recorridos en toda recolección, ya sea minor collection o major collection y cualquiera que sea el contenido de las variables locales. Para corregirlo, simplemente se cambió el nombre del estado al que se pasaba cuando entrábamos en la rama *else*.

```
if (localVars.redFish != 0 && localVars.redFish < gcHeap.getObjectPoolSize())
........................
else
{
   currentGCState = empezandoConLosObjetosApuntadosDesdeRegion;
.........................
```

Tan pronto como se cambió el valor asignado a la variable *currentGCState* , el garbage collector funcionó correctamente.

Para probar que al cambiar de región todo sigue funcionando igual, después de haber probado esto y antes de que los objetos del heap lleguen a la generación antigua hay que lanzar este mismo código en otra región distinta e inmediatamente después, se destruye esta región. Es importante lanzarlo antes de destruir esta región para que la asignación de direcciones para la región no sea la misma. Después de haber hecho esto, se ejecuta el garbage collector y se observa que los objetos del heap siguen sin ser recolectados, ya que están apuntados desde el objeto p1 de esta nueva región.

- Comprobación de que cuando una tarea no crítica ejecuta en una región y se elimina una referencia al heap, se elimina de la tabla de referencias y no se chequea. Para esto tenemos que lanzar a ejecución en el heap el siguiente código:

```
crr p8
craz p9
cram p4
cram p5
cm p9,p5
craz p6
cm p8,p6
craz p7
am p6,p7
```

Después de lanzar este código a ejecución en el heap, hay que lanzar a ejecución este código en la región:

```
asr p1,r3
sndi p1,#3
amdi p1,#1
cmdi p1,#2
cram p2
cram p3
am p2,p3
sn p1,p2
```

Mientras este código está en ejecución hay que lanzar a ejecución el garbage collector, procurando que el chequeo de las regiones se haga antes que acabe la ejecución el programa y se verá como se chequean los peces *p4*, *p8* y *p9* situados

en el heap, ya que están apuntados desde el objeto *p1*, situado en la región, pero cuando llegamos al final de la ejecución, p4 deja de estar apuntado por *p1* porque *p1* pasa a apuntan a *p2*. Si el garbage collector vuelve a ejecutar, *p4* será recolectado. Para que *p4* acabe siendo recolectado, la tarea ejecutada en la región no debe ser crítica. Si probamos a ejecutar una crítica, al finalizar la ejecución, p4 no será recolectado porque las tareas críticas insertan referencias en la lista, pero no las borran.

5.6.- Ajuste de parámetros

En esta aplicación hay varios parámetros que tienen que ser ajustados, estos parámetros son la frecuencia con la que se ejecuta una major collection, el tamaño de la generación nueva y generación antigua y el número de minor collection a los que debe sobrevivir un objeto para pasar a la generación antigua.

- En cuanto a la frecuencia con la que ejecuta una major collection, se ha decidido que una major collection debe ejecutar 1 vez por cada 5 veces que ejecuta un minor collection, es decir, primero ejecutan 5 minor collection, luego una major collection y luego se vuelve a empezar: 5 minor collection, una major collection y así sucesivamente. Para hacer esto no hay más que fijar el valor de la constante *cicloRecoleccion* a 6 (5 + 1), situada en la clase *GarbageCollectCanvas*.
- El tamaño de la generación nueva es 2/3 del heap y el tamaño de la generación antigua es de 1/3 del heap. La generación nueva debe ser mayor porque los objetos, en su mayoría, son de vida corta y pocos llegarán a pasar a la generación antigua. Esto se hace fijando el valor del atributo *frontera* de la clase *GCHeap*.
- En cuanto al número de minor collection a los que debe sobrevivir un objeto para que llegue a promocionar, es de 10 minor collection; es decir, todo objeto de la generación nueva que sobreviva a 10 minor collection será trasladado a la generación antigua. Para hacer esto se fija el valor de la constante *vecesSobrevividasParaPromocionar*, situada en la clase *MinorCollection*.

Apéndice I.- Instrucciones utilizadas para representar las acciones típicas de la aplicación ejemplo

Como se había comentado anteriormente, tuvo lugar la implementación de un subprograma capaz de lanzar a ejecución una secuencia de instrucciones de creación de peces y asignación de referencias, en lugar de introducirlo el usuario interactivamente. Para ello, en lugar de utilizar lenguaje Java, se utiliza el siguiente código:

Instrucciones que se usarán en nuestro código

crr	p1	Crea un pez rojo en el heap, etiquetado con p1
craz	p1	Crea un pez azul en el heap, etiquetado con p1
cram	p1	Crea un pez amarillo en el heap, etiquetado con p1
cm	p1,p2	Establece una referencia de p1 a p2 indicando que el pez p2 es la comida del pez p1
am	p1,p2	Establece una referencia de p1 a p2 indicando que el pez p2 es amigo del pez p1
sn	p1,p2	Establece una referencia de p1 a p2 indicando que el pez p2 es snack del pez p1
ncm	p1	Pone la referencia de tipo comida de p1 a nula
nam	p1	Pone la referencia de tipo amigo de p1 a nula
nsn	p1	Pone la referencia de tipo snack de p1 a nula
asr	p1,r1	Asigna el pez p1 a la referencia p1
nr	r1	Pone la referencia 1 (blue fish) a nula
cmi	#1,#2	Establece una referencia del pez que está en la dirección de memoria 1 al pez que está en la dirección de memoria 2 indicando que el segundo es la comida del primero
ami	#1,#2	Establece una referencia del pez que está en la dirección de memoria 1 al pez que está en la dirección de memoria 2 indicando que el segundo es el amigo del primero
sni	#1,#2	Establece una referencia del pez que está en la dirección de memoria 1 al pez que está en la dirección de memoria 2 indicando que el segundo es el snack del primero
cmdi	p1,#2	Establece una referencia del pez p1 al pez que está en la dirección de memoria 2 indicando que el segundo es la comida del primero
amdi	p1,#2	Establece una referencia del pez p1 al pez que está en la dirección de memoria 2 indicando que el segundo es el amigo del primero
sndi	p1,#2	Establece una referencia del pez p1 al pez que está en la dirección de memoria 2 indicando que el segundo es el snack del primero

Estas instrucciones están contenidas en un archivo cualquiera y se permite que el usuario seleccione el archivo.

Apéndice II.- Código modificado sobre el prototipo original.

Como comentabámos en el punto las clases que se han modificado sobre el original son las siguientes:

a) *AllocateFishPanel*.
b) *AssignReferencesCanvas*.
c) *GarbageCollectCanvas*.
d) *GCHeap*.
e) *HeapOfFish*.
f) *HeapOfFishCanvases*.
g) *HeapOfFishCheckModePanel*.
h) *HeapOfFishStrings*.
i) *LinkFishCanvas*.
j) *ObjectHandle*.
k) *PoolsCanvas*.
l) *FishIcon*.

Además se han creado estas clases:

m) *EspacioRegiones*.
n) *Instruccion*.
o) *ListaReferencias*.
p) *MemoriaDeInstrucciones*
q) *MotorDeEjecucion* e *HiloDeEjecucion*.
r) *PanelCargarCodigo*.
s) *PanelBotonesCargarArchivo*.
t) *PecesIniciales*.
u) *ReferenciasIniciales*.
v) *Region*.
w) *Recoleccion, RecoleccionAbstracta*, *MajorCollection* y *MinorCollection*.
x) *Tarea*, *TareaNormal* y *TareaCritica*.

La modificación de las primeras e implementación de las segundas se ha hecho de la siguiente manera:

a) Clase *AllocateFishPanel*:

Se han añadido referencias a los objetos de las clases *MemoriaDeInstrucciones*, *EspacioRegiones*, y *ReferenciasIniciales* , así como los métodos *crearPezRojo*, *crearPezAzul* y *crearPezAmarillo* y se han suprimido todos los demás métodos.

```
class AllocateFishPanel extends Panel{

    GCHeap gcHeap;
    Region region;
    EspacioRegiones espRegiones;
    MemoriaDeInstrucciones mi;
    HeapOfFishTextArea controlPanelTextArea;


    ReferenciasIniciales refIni;

    PoolsCanvas poolsCanvas;


AllocateFishPanel(GCHeap heap, MemoriaDeInstrucciones memIns, HeapOfFishTextArea ta,
EspacioRegiones regiones) {

        gcHeap = heap;
        region = null;
        espRegiones = regiones;
        controlPanelTextArea = ta;
        mi = memIns;

        setBackground(Color.blue);
        setLayout(new BorderLayout());

        poolsCanvas = new PoolsCanvas(gcHeap,espRegiones);


        add("Center", poolsCanvas);

    }


public synchronized void crearPezRojo(Instruccion ins,PecesIniciales pi,MemoriaDeInstrucciones mem)
    {
      int direccion = pi.nuevoPezRojo();
      mem.ponReferencia(ins.dameOperando(0),direccion);
      poolsCanvas.repaint();
    }

    public synchronized void crearPezAzul(Instruccion ins,PecesIniciales pi,MemoriaDeInstrucciones
mem)
    {
      int direccion = pi.nuevoPezAzul();
      mem.ponReferencia(ins.dameOperando(0),direccion);
      poolsCanvas.repaint();
    }

    public synchronized void crearPezAmarillo(Instruccion ins,PecesIniciales pi,MemoriaDeInstrucciones
mem)
    {
      int direccion = pi.nuevoPezAmarillo();
      mem.ponReferencia(ins.dameOperando(0),direccion);
      poolsCanvas.repaint();
    }
}
```

Se ha eliminado del constructor la instrucción:

```
add("South", new AllocateFishButtonPanel());
```

Esta instrucción ha sido eliminada porque la clase *AllocateFishButtonPanel* se compone de un panel que contiene unos botones para crear los peces y ahora no hay que crear peces por orden del usuario ya que ahora se especifica mediante un archivo. Por otra parte se han definido métodos para crear peces de cada color y se ha añadido una referencia al objeto de la clase *EspacioRegiones*, que se ocupa de gestionar las regiones.

b) Clase *AssignReferencesCanvas*:

Se han añadido atributos para poder pintar también los peces de las regiones:

```
....
protected EspacioRegiones espRegiones;
protected Region region;
....
```

por otra parte se quieren pintar los peces de la generación antigua de un color más oscuro, hay que chequear si alguna de las 3 variables apunta a objetos de la región y además hay que recorrer los peces de las regiones para poder pintarlos. Por lo tanto el método paint se ha transformado.

```
public void paint(Graphics g) {

    Font font = g.getFont();
    FontMetrics fm = getFontMetrics(font);

    int localVarsStringWidth = fm.stringWidth(HeapOfFishStrings.localVariables);
    int redFishStringWidth = fm.stringWidth(HeapOfFishStrings.redFishLocalVar);
    int blueFishStringWidth = fm.stringWidth(HeapOfFishStrings.blueFishLocalVar);
    int yellowFishStringWidth = fm.stringWidth(HeapOfFishStrings.yellowFishLocalVar);

    localVarRectWidth = localVarsStringWidth;
    if (redFishStringWidth > localVarRectWidth) {
      localVarRectWidth = redFishStringWidth;
    }
    if (blueFishStringWidth > localVarRectWidth) {
      localVarRectWidth = blueFishStringWidth;
    }
    if (yellowFishStringWidth > localVarRectWidth) {
      localVarRectWidth = yellowFishStringWidth;
    }

    localVarRectWidth += 2 * localVarStringMargin;
    xFishAreaStart = localVarRectWidth + (2 * localVarStringMargin);
```

```
localVarRectHeight = fm.getAscent() + fm.getDescent() + 2 * localVarStringMargin;

Dimension dim = size();
int yLocalVarsAreaStart = (dim.height - (4 * localVarRectHeight)) / 2;
if (yLocalVarsAreaStart < 0) {
   yLocalVarsAreaStart = 0;
}

// draw "Local Variables"
int xStart = ((localVarRectWidth - localVarsStringWidth) / 2) + localVarStringMargin;
int yStart = yLocalVarsAreaStart + localVarStringMargin + fm.getAscent();
g.setColor(Color.white);
g.drawString(HeapOfFishStrings.localVariables, xStart, yStart);

// draw "YellowFish yf" on a yellow rectangle
xStart = localVarStringMargin;
yStart = yLocalVarsAreaStart + localVarRectHeight;
g.setColor(Color.yellow);
g.fillRect(xStart, yStart, localVarRectWidth, localVarRectHeight);
xLocalVarRectStart = xStart;
yYellowFishLocalVarStart = yStart;

xStart = ((localVarRectWidth - yellowFishStringWidth) / 2) + localVarStringMargin;
yStart += localVarStringMargin + fm.getAscent();
g.setColor(Color.black);
g.drawString(HeapOfFishStrings.yellowFishLocalVar, xStart, yStart);

// draw "BlueFish bf" on a cyan rectangle
xStart = localVarStringMargin;
yStart = yLocalVarsAreaStart + (2 * localVarRectHeight);
g.setColor(Color.cyan);
g.fillRect(xStart, yStart, localVarRectWidth, localVarRectHeight);
yBlueFishLocalVarStart = yStart;

xStart = ((localVarRectWidth - blueFishStringWidth) / 2) + localVarStringMargin;
yStart += localVarStringMargin + fm.getAscent();
g.setColor(Color.black);
g.drawString(HeapOfFishStrings.blueFishLocalVar, xStart, yStart);

// draw "RedFish rf" on a red rectangle
xStart = localVarStringMargin;
yStart = yLocalVarsAreaStart + (3 * localVarRectHeight);
g.setColor(Color.red);
g.fillRect(xStart, yStart, localVarRectWidth, localVarRectHeight);
yRedFishLocalVarStart = yStart;

xStart = ((localVarRectWidth - redFishStringWidth) / 2) + localVarStringMargin;
yStart += localVarStringMargin + fm.getAscent();
g.setColor(Color.black);
g.drawString(HeapOfFishStrings.redFishLocalVar, xStart, yStart);

if (localVars.yellowFish != 0) {
   g.setColor(Color.blue);
   g.setXORMode(Color.yellow);
   ObjectHandle yf;
   if(localVars.yellowFish < gcHeap.getObjectPoolSize())
     yf = gcHeap.getObjectHandle(localVars.yellowFish);
   else
   {
     region = espRegiones.dameRegionQueContengaLaDireccion(localVars.yellowFish);
```

```
    yf = region.getObjectHandle(localVars.yellowFish);
  }
  int xLineStart = xLocalVarRectStart + localVarRectWidth;
  int yLineStart = yYellowFishLocalVarStart + (localVarRectHeight / 2);
  g.drawLine(xLineStart, yLineStart, yf.fish.getFishPosition().x, yf.fish.getFishPosition().y);
  if (localVars.yellowLineStart == null) {
    localVars.yellowLineStart = new Point(0, 0);
    localVars.yellowLineEnd = new Point(0, 0);
  }
  localVars.yellowLineStart.x = xLineStart;
  localVars.yellowLineStart.y = yLineStart;
  localVars.yellowLineEnd.x = yf.fish.getFishPosition().x;
  localVars.yellowLineEnd.y = yf.fish.getFishPosition().y;
  g.setPaintMode();
}

if (localVars.blueFish != 0) {
  g.setColor(Color.blue);
  g.setXORMode(Color.cyan);
  ObjectHandle bf;
  if(localVars.blueFish < gcHeap.getObjectPoolSize())
    bf = gcHeap.getObjectHandle(localVars.blueFish);
  else
  {
    region = espRegiones.dameRegionQueContengaLaDireccion(localVars.blueFish);
    bf = region.getObjectHandle(localVars.blueFish);
  }
  int xLineStart = xLocalVarRectStart + localVarRectWidth;
  int yLineStart = yBlueFishLocalVarStart + (localVarRectHeight / 2);
  g.drawLine(xLineStart, yLineStart, bf.fish.getFishPosition().x, bf.fish.getFishPosition().y);
  if (localVars.blueLineStart == null) {
    localVars.blueLineStart = new Point(0, 0);
    localVars.blueLineEnd = new Point(0, 0);
  }
  localVars.blueLineStart.x = xLineStart;
  localVars.blueLineStart.y = yLineStart;
  localVars.blueLineEnd.x = bf.fish.getFishPosition().x;
  localVars.blueLineEnd.y = bf.fish.getFishPosition().y;
  g.setPaintMode();
}

if (localVars.redFish != 0) {
  g.setColor(Color.blue);
  g.setXORMode(Color.red);
  ObjectHandle rf;
  if(localVars.redFish < gcHeap.getObjectPoolSize())
    rf = gcHeap.getObjectHandle(localVars.redFish);
  else
  {
    region = espRegiones.dameRegionQueContengaLaDireccion(localVars.redFish);
    rf = region.getObjectHandle(localVars.redFish);
  }
  int xLineStart = xLocalVarRectStart + localVarRectWidth;
  int yLineStart = yRedFishLocalVarStart + (localVarRectHeight / 2);
  g.drawLine(xLineStart, yLineStart, rf.fish.getFishPosition().x, rf.fish.getFishPosition().y);
  if (localVars.redLineStart == null) {
    localVars.redLineStart = new Point(0, 0);
    localVars.redLineEnd = new Point(0, 0);
  }
  localVars.redLineStart.x = xLineStart;
```

```
            localVars.redLineStart.y = yLineStart;
            localVars.redLineEnd.x = rf.fish.getFishPosition().x;
            localVars.redLineEnd.y = rf.fish.getFishPosition().y;
            g.setPaintMode();
        }

        // Figure out how many slots will be in each of three columns of slots where
        // new fish are put.
        int columnCount = 3;
        int slotsPerColumn = (gcHeap.getHandlePoolSize() + espRegiones.getHandlePoolSize()) /
columnCount;
        if (gcHeap.getHandlePoolSize() % columnCount > 0) {
            ++slotsPerColumn;
        }
        int fishAreaWidth = dim.width - xFishAreaStart;
        int slotWidth = fishAreaWidth / columnCount;
        int slotHeight = dim.height / slotsPerColumn;
        int pecesHeap = gcHeap.getHandlePoolSize();

        for (int i = 0; i < gcHeap.getHandlePoolSize(); ++i) {
            ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
            if (!oh.free) {

                FishIcon fishIcon = oh.fish;
                if (!fishIcon.getFishHasBeenInitiallyPositioned()) {
                    int column = i / slotsPerColumn;
                    int row = i % slotsPerColumn;
                    int xFishPosition = (int) ((double) (slotWidth - fishIcon.getFishWidth()) * Math.random());
                    if (xFishPosition < 0) {
                        xFishPosition = 0;
                    }
                    xFishPosition += xFishAreaStart + (column * slotWidth);

                    int yFishPosition = (slotHeight - fishIcon.getFishHeight()) / 2;
                    if (yFishPosition < 0) {
                        yFishPosition = 0;
                    }
                    yFishPosition += row * slotHeight;
                    fishIcon.moveFish(xFishPosition, yFishPosition);
                }

                if(oh.objectPos < gcHeap.dameFrontera())
                  fishIcon.paint(g);
                else
                  fishIcon.pintarPezViejo(g);

                // Draw any lines connecting fish.
                g.setColor(Color.blue);
                g.setXORMode(fishIcon.getFishColor());
                Point fishNose = fishIcon.getFishNosePosition();

                oh.gotFriend = false;
                oh.myFriendLineStart = null;
                oh.myFriendLineEnd = null;

                oh.gotLunch = false;
                oh.myLunchLineStart = null;
                oh.myLunchLineEnd = null;

                oh.gotSnack = false;
```

```
            oh.mySnackLineStart = null;
            oh.mySnackLineEnd = null;

            int myFriendIndex = gcHeap.getObjectPool(oh.objectPos);

            if (myFriendIndex != 0) {
               ObjectHandle myFriend;
                 myFriend= gcHeap.getObjectHandle(myFriendIndex);
               g.drawLine(fishNose.x, fishNose.y, myFriend.fish.getFishPosition().x,
myFriend.fish.getFishPosition().y);
               oh.gotFriend = true;
               oh.myFriendLineStart = new Point(fishNose.x, fishNose.y);
               oh.myFriendLineEnd = new Point(myFriend.fish.getFishPosition().x,
myFriend.fish.getFishPosition().y);
            }

            if (fishIcon.getFishColor() == Color.yellow) {
               g.setPaintMode();
               continue;
            }

            int myLunchIndex = gcHeap.getObjectPool(oh.objectPos + 1);

            if (myLunchIndex != 0) {
               ObjectHandle myLunch;
                 myLunch = gcHeap.getObjectHandle(myLunchIndex);
               g.drawLine(fishNose.x, fishNose.y, myLunch.fish.getFishPosition().x,
myLunch.fish.getFishPosition().y);
               oh.gotLunch = true;
               oh.myLunchLineStart = new Point(fishNose.x, fishNose.y);
               oh.myLunchLineEnd = new Point(myLunch.fish.getFishPosition().x,
myLunch.fish.getFishPosition().y);
            }

            if (fishIcon.getFishColor() == Color.cyan) {
               g.setPaintMode();
               continue;
            }

            int mySnackIndex = gcHeap.getObjectPool(oh.objectPos + 2);

            if (mySnackIndex != 0) {
               ObjectHandle mySnack;
                 mySnack = gcHeap.getObjectHandle(mySnackIndex);
               g.drawLine(fishNose.x, fishNose.y, mySnack.fish.getFishPosition().x,
mySnack.fish.getFishPosition().y);
               oh.gotSnack = true;
               oh.mySnackLineStart = new Point(fishNose.x, fishNose.y);
               oh.mySnackLineEnd = new Point(mySnack.fish.getFishPosition().x,
mySnack.fish.getFishPosition().y);
            }

            g.setPaintMode();
         }
      }

         //Pintamos los peces de las regiones
      Region[] regs = espRegiones.dameRegionesOcupadas();
      int [] numReg = espRegiones.dameNumerosRegionesOcupadas();
      for(int j=0;j<regs.length;j++)
```

```
{
  region = regs[j];

   for (int i = 0; i < region.getHandlePoolSize(); ++i) {
   ObjectHandle oh = region.dameManejador(i + 1);
   if (!oh.free) {

       FishIcon fishIcon = oh.fish;
       if (!fishIcon.getFishHasBeenInitiallyPositioned()) {
           int column = (i + pecesHeap) / slotsPerColumn;
           int row = (i + pecesHeap) % slotsPerColumn;
           int xFishPosition = (int) ((double) (slotWidth - fishIcon.getFishWidth()) * Math.random());
           if (xFishPosition < 0) {
               xFishPosition = 0;
           }
           xFishPosition += xFishAreaStart + (column * slotWidth);
           if(xFishPosition > dim.width - xFishAreaStart)
             xFishPosition = dim.width - xFishAreaStart - 10;
           int yFishPosition = (slotHeight - fishIcon.getFishHeight()) / 2;
           if (yFishPosition < 0) {
               yFishPosition = 0;
           }
           yFishPosition += row * slotHeight;
           fishIcon.moveFish(xFishPosition, yFishPosition);
       }
       fishIcon.paint(g);



       Dimension tamCuerpo = fishIcon.ovalDim;
       int posCuerpoX = fishIcon.getFishPosition().x + 6 * tamCuerpo.width/10;
       int posCuerpoY = fishIcon.getFishPosition().y + tamCuerpo.height/3;
       int tamCuerpoX = 6 * tamCuerpo.width / 10;
       int tamCuerpoY = 7 * tamCuerpo.height / 10;

       g.setColor(Color.black);
       g.fillRect(posCuerpoX, posCuerpoY, tamCuerpoX, tamCuerpoY);

       Font fuente = g.getFont();
       fuente = new Font(fuente.getName(), fuente.getStyle(), 10);

       g.setColor(fishIcon.getFishColor());
       g.setFont(fuente);

       g.drawString(Integer.toString(numReg[j]), posCuerpoX, posCuerpoY);


       // Draw any lines connecting fish.
       g.setColor(Color.blue);
       g.setXORMode(fishIcon.getFishColor());
       Point fishNose = fishIcon.getFishNosePosition();

       oh.gotFriend = false;
       oh.myFriendLineStart = null;
       oh.myFriendLineEnd = null;

       oh.gotLunch = false;
       oh.myLunchLineStart = null;
       oh.myLunchLineEnd = null;
```

```
            oh.gotSnack = false;
            oh.mySnackLineStart = null;
            oh.mySnackLineEnd = null;

            int myFriendIndex = region.getObjectPool(oh.objectPos);

            if (myFriendIndex != 0) {
               ObjectHandle myFriend;
               if(myFriendIndex <= gcHeap.getObjectPoolSize())
                 myFriend = gcHeap.getObjectHandle(myFriendIndex);
               else
                 myFriend = region.getObjectHandle(myFriendIndex);
               g.drawLine(fishNose.x, fishNose.y, myFriend.fish.getFishPosition().x,
myFriend.fish.getFishPosition().y);
               oh.gotFriend = true;
               oh.myFriendLineStart = new Point(fishNose.x, fishNose.y);
               oh.myFriendLineEnd = new Point(myFriend.fish.getFishPosition().x,
myFriend.fish.getFishPosition().y);
            }

            if (fishIcon.getFishColor() == Color.yellow) {
               g.setPaintMode();
               continue;
            }

            int myLunchIndex = region.getObjectPool(oh.objectPos + 1);

            if (myLunchIndex != 0) {
               ObjectHandle myLunch;

               if(myLunchIndex <= gcHeap.getObjectPoolSize())
                 myLunch = gcHeap.getObjectHandle(myLunchIndex);
               else
                 myLunch = region.getObjectHandle(myLunchIndex);
               g.drawLine(fishNose.x, fishNose.y, myLunch.fish.getFishPosition().x,
myLunch.fish.getFishPosition().y);
               oh.gotLunch = true;
               oh.myLunchLineStart = new Point(fishNose.x, fishNose.y);
               oh.myLunchLineEnd = new Point(myLunch.fish.getFishPosition().x,
myLunch.fish.getFishPosition().y);
            }

            if (fishIcon.getFishColor() == Color.cyan) {
               g.setPaintMode();
               continue;
            }

            int mySnackIndex = region.getObjectPool(oh.objectPos + 2);

            if (mySnackIndex != 0) {
               ObjectHandle mySnack;
               if(mySnackIndex <= gcHeap.getObjectPoolSize())
                 mySnack = gcHeap.getObjectHandle(mySnackIndex);
               else
                 mySnack = region.getObjectHandle(mySnackIndex);
               g.drawLine(fishNose.x, fishNose.y, mySnack.fish.getFishPosition().x,
mySnack.fish.getFishPosition().y);
               oh.gotSnack = true;
               oh.mySnackLineStart = new Point(fishNose.x, fishNose.y);
```

```
                oh.mySnackLineEnd = new Point(mySnack.fish.getFishPosition().x,
mySnack.fish.getFishPosition().y);
             }

             g.setPaintMode();
          }
          }
       }
    }
```

c) Clase *GarbageCollectCanvas* :

Se han añadido atributos para permitir los dos tipos de recolección y para conseguir que la ejecución del garbage collector sea continuada, sin tener el usuario que indicar cuando hay que pasar de una etapa a la siguiente. El atributo *listaRecoleccion* es un array que contiene la secuencia de recolecciones que se ejecutan, ya sean major collection o minor collection. El atributo *recoleccion* apunta a un componente del array y es la recolección actual. Esta variable se utiliza por comodidad pero podríamos utilizar también *listaRecoleccion[indiceRecoleccion]*. El atributo *indiceRecoleccion* marca la posición actual del array para ejecutar la siguiente recolección. La constante *cicloRecoleccion* es la que especifica el tamaño del array *listaRecoleccion*, que siempre es un numero más que las minor collection ejecutadas entre dos major collection.

```
public class GarbageCollectCanvas extends Canvas implements Runnable{

    .....
     private Recoleccion recoleccion;
    private Recoleccion listaRecoleccion[];
    private int indiceRecoleccion;

    private final int cicloRecoleccion = 6;

   .....

    private boolean parar = false;

    private Thread thread;

    .....
```

El constructor debe inicializar todos los objetos de manera que se puedan ejecutar tanto minor collection como major collection y se pueda alternar entre ellas de manera automática.

```
....
GarbageCollectCanvas(GCHeap heap, EspacioRegiones er, ReferenciasIniciales locVars,
HeapOfFishTextArea ta) {
        setBackground(Color.blue);
        gcHeap = heap;
        localVars = locVars.dameVariablesLocales();
        controlPanelTextArea = ta;
        recoleccion = new MinorCollection(this,heap,er,locVars,ta);
        listaRecoleccion = new Recoleccion[cicloRecoleccion];
        for(int i=0;i < cicloRecoleccion - 1;i++)
```

```
        listaRecoleccion[i] = recoleccion;
      listaRecoleccion[cicloRecoleccion - 1] = new MajorCollection(this,heap,er,localVars,ta);
      indiceRecoleccion = -1;
  }
....
```

Por otra parte hay algunos atributos que necesitan accesores y/o mutadores para poder ser usados por *MinorCollection* , *MajorCollection* y otras clases.

```
.....
  public Thread dameThread(){
    return thread;
  }

  public boolean dameParar(){
    return parar;
  }

  public void setYellowFishLocalVarLineColor(Color yflvlc)
  {
    yellowFishLocalVarLineColor = yflvlc;
  }

  public void setBlueFishLocalVarLineColor(Color bflvlc)
  {
    blueFishLocalVarLineColor = bflvlc;
  }

  public void setRedFishLocalVarLineColor(Color rflvlc)
  {
    redFishLocalVarLineColor = rflvlc;
  }
  ....
```

En el método *paint* se debe controlar que si una variable apunta a un objeto de una región no hay que pintar nada, ya que en el panel *Garbage Collector* solamente se pintan los objetos del heap; por otra parte si el garbage collector no está ejecutando hay que pintar los peces de la generación antigua de un color más oscuro y además, las variables de estado del garbage collector ahora se encuentran en la recolección correspondiente, por tanto se debe acceder a ellas utilizando los accesores de la interface *Recoleccion*.

```
public void paint(Graphics g) {

      Font font = g.getFont();
      FontMetrics fm = getFontMetrics(font);

      int localVarsStringWidth = fm.stringWidth(HeapOfFishStrings.localVariables);
      int redFishStringWidth = fm.stringWidth(HeapOfFishStrings.redFishLocalVar);
      int blueFishStringWidth = fm.stringWidth(HeapOfFishStrings.blueFishLocalVar);
      int yellowFishStringWidth = fm.stringWidth(HeapOfFishStrings.yellowFishLocalVar);

      localVarRectWidth = localVarsStringWidth;
      if (redFishStringWidth > localVarRectWidth) {
          localVarRectWidth = redFishStringWidth;
```

```
}
if (blueFishStringWidth > localVarRectWidth) {
    localVarRectWidth = blueFishStringWidth;
}
if (yellowFishStringWidth > localVarRectWidth) {
    localVarRectWidth = yellowFishStringWidth;
}

localVarRectWidth += 2 * localVarStringMargin;
xFishAreaStart = localVarRectWidth + (2 * localVarStringMargin);

localVarRectHeight = fm.getAscent() + fm.getDescent() + 2 * localVarStringMargin;

Dimension dim = size();
int yLocalVarsAreaStart = (dim.height - (4 * localVarRectHeight)) / 2;
if (yLocalVarsAreaStart < 0) {
    yLocalVarsAreaStart = 0;
}

// draw "Local Variables"
int xStart = ((localVarRectWidth - localVarsStringWidth) / 2) + localVarStringMargin;
int yStart = yLocalVarsAreaStart + localVarStringMargin + fm.getAscent();
g.setColor(Color.white);
g.drawString(HeapOfFishStrings.localVariables, xStart, yStart);

// draw "YellowFish yf" on a yellow rectangle
xStart = localVarStringMargin;
yStart = yLocalVarsAreaStart + localVarRectHeight;
if (recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorHasNotStarted() ||
    recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorIsDone()) {
    g.setColor(Color.yellow);
}
else if (recoleccion.getYellowFishLocalVarIsCurrentGCMarkNode()) {
    g.setColor(currentGCMarkNodeColor);
}
else {
    g.setColor(Color.white);
}
g.fillRect(xStart, yStart, localVarRectWidth, localVarRectHeight);
xLocalVarRectStart = xStart;
yYellowFishLocalVarStart = yStart;

xStart = ((localVarRectWidth - yellowFishStringWidth) / 2) + localVarStringMargin;
yStart += localVarStringMargin + fm.getAscent();
g.setColor(Color.black);
g.drawString(HeapOfFishStrings.yellowFishLocalVar, xStart, yStart);

// draw "BlueFish bf" on a cyan rectangle
xStart = localVarStringMargin;
yStart = yLocalVarsAreaStart + (2 * localVarRectHeight);
if (recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorHasNotStarted() ||
    recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorIsDone()) {
    g.setColor(Color.cyan);
}
else if (recoleccion.getBlueFishLocalVarIsCurrentGCMarkNode()) {
    g.setColor(currentGCMarkNodeColor);
}
else {
    g.setColor(Color.white);
}
```

```
g.fillRect(xStart, yStart, localVarRectWidth, localVarRectHeight);
yBlueFishLocalVarStart = yStart;

xStart = ((localVarRectWidth - blueFishStringWidth) / 2) + localVarStringMargin;
yStart += localVarStringMargin + fm.getAscent();
g.setColor(Color.black);
g.drawString(HeapOfFishStrings.blueFishLocalVar, xStart, yStart);

// draw "RedFish rf" on a red rectangle
xStart = localVarStringMargin;
yStart = yLocalVarsAreaStart + (3 * localVarRectHeight);
if (recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorHasNotStarted() ||
   recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorIsDone()) {
   g.setColor(Color.red);
}
else if (recoleccion.getRedFishLocalVarIsCurrentGCMarkNode()) {
   g.setColor(currentGCMarkNodeColor);
}
else {
   g.setColor(Color.white);
}
g.fillRect(xStart, yStart, localVarRectWidth, localVarRectHeight);
yRedFishLocalVarStart = yStart;

xStart = ((localVarRectWidth - redFishStringWidth) / 2) + localVarStringMargin;
yStart += localVarStringMargin + fm.getAscent();
g.setColor(Color.black);
g.drawString(HeapOfFishStrings.redFishLocalVar, xStart, yStart);

if (localVars.yellowFish != 0 && localVars.yellowFish < gcHeap.getObjectPoolSize()) {
   g.setColor(Color.blue);
   if (recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorHasNotStarted() ||
      recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorIsDone()) {
      g.setXORMode(Color.yellow);
   }
   else {
      g.setXORMode(yellowFishLocalVarLineColor);
   }
   ObjectHandle yf = gcHeap.getObjectHandle(localVars.yellowFish);
   int xLineStart = xLocalVarRectStart + localVarRectWidth;
   int yLineStart = yYellowFishLocalVarStart + (localVarRectHeight / 2);
   g.drawLine(xLineStart, yLineStart, yf.fish.getFishPosition().x, yf.fish.getFishPosition().y);
   if (localVars.yellowLineStart == null) {
      localVars.yellowLineStart = new Point(0, 0);
      localVars.yellowLineEnd = new Point(0, 0);
   }
   localVars.yellowLineStart.x = xLineStart;
   localVars.yellowLineStart.y = yLineStart;
   localVars.yellowLineEnd.x = yf.fish.getFishPosition().x;
   localVars.yellowLineEnd.y = yf.fish.getFishPosition().y;
   g.setPaintMode();
}

if (localVars.blueFish != 0 && localVars.blueFish < gcHeap.getObjectPoolSize()) {
   g.setColor(Color.blue);
   if (recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorHasNotStarted() ||
      recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorIsDone()) {
      g.setXORMode(Color.cyan);
   }
   else {
```

```
      g.setXORMode(blueFishLocalVarLineColor);
    }
    ObjectHandle bf = gcHeap.getObjectHandle(localVars.blueFish);
    int xLineStart = xLocalVarRectStart + localVarRectWidth;
    int yLineStart = yBlueFishLocalVarStart + (localVarRectHeight / 2);
    g.drawLine(xLineStart, yLineStart, bf.fish.getFishPosition().x, bf.fish.getFishPosition().y);
    if (localVars.blueLineStart == null) {
      localVars.blueLineStart = new Point(0, 0);
      localVars.blueLineEnd = new Point(0, 0);
    }
    localVars.blueLineStart.x = xLineStart;
    localVars.blueLineStart.y = yLineStart;
    localVars.blueLineEnd.x = bf.fish.getFishPosition().x;
    localVars.blueLineEnd.y = bf.fish.getFishPosition().y;
    g.setPaintMode();
  }

  if (localVars.redFish != 0 && localVars.redFish < gcHeap.getObjectPoolSize()) {
    g.setColor(Color.blue);
    if (recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorHasNotStarted() ||
      recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorIsDone()) {
      g.setXORMode(Color.red);
    }
    else {
      g.setXORMode(redFishLocalVarLineColor);
    }
    ObjectHandle rf = gcHeap.getObjectHandle(localVars.redFish);
    int xLineStart = xLocalVarRectStart + localVarRectWidth;
    int yLineStart = yRedFishLocalVarStart + (localVarRectHeight / 2);
    g.drawLine(xLineStart, yLineStart, rf.fish.getFishPosition().x, rf.fish.getFishPosition().y);
    if (localVars.redLineStart == null) {
      localVars.redLineStart = new Point(0, 0);
      localVars.redLineEnd = new Point(0, 0);
    }
    localVars.redLineStart.x = xLineStart;
    localVars.redLineStart.y = yLineStart;
    localVars.redLineEnd.x = rf.fish.getFishPosition().x;
    localVars.redLineEnd.y = rf.fish.getFishPosition().y;
    g.setPaintMode();
  }

  // Figure out how many slots will be in each of three columns of slots where
  // new fish are put.
  int columnCount = 3;
  int slotsPerColumn = gcHeap.getHandlePoolSize() / columnCount;
  if (gcHeap.getHandlePoolSize() % columnCount > 0) {
    ++slotsPerColumn;
  }
  int fishAreaWidth = dim.width - xFishAreaStart;
  int slotWidth = fishAreaWidth / columnCount;
  int slotHeight = dim.height / slotsPerColumn;

  for (int i = 0; i < gcHeap.getHandlePoolSize(); ++i) {
    ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
    if (!oh.free) {

      FishIcon fishIcon = oh.fish;
      if (!fishIcon.getFishHasBeenInitiallyPositioned()) {
        int column = i / slotsPerColumn;
        int row = i % slotsPerColumn;
```

```
        int xFishPosition = (int) ((double) (slotWidth - fishIcon.getFishWidth()) * Math.random());
        if (xFishPosition < 0) {
            xFishPosition = 0;
        }
        xFishPosition += xFishAreaStart + (column * slotWidth);
        int yFishPosition = (slotHeight - fishIcon.getFishHeight()) / 2;
        if (yFishPosition < 0) {
            yFishPosition = 0;
        }
        yFishPosition += row * slotHeight;
        fishIcon.moveFish(xFishPosition, yFishPosition);
    }
    if (recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorHasNotStarted() ||
        recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorIsDone()) {
  if(oh.objectPos < gcHeap.dameFrontera())
        fishIcon.paint(g);
      else
        fishIcon.pintarPezViejo(g);
    }
    else {
        if (recoleccion.getFishAreBeingMarked() && recoleccion.getCurrentFishBeingMarked() ==
i + 1) {
            fishIcon.paintWithSpecialColors(g, currentGCMarkNodeColor, Color.black);
        }
        else {
            Color eyeColor = Color.black;
            if (oh.myColor == Color.black) {
                eyeColor = Color.white;
            }
            fishIcon.paintWithSpecialColors(g, oh.myColor, eyeColor);
        }
    }

    // Draw any lines connecting fish.
    g.setColor(Color.blue);
    g.setXORMode(fishIcon.getFishColor());
    Point fishNose = fishIcon.getFishNosePosition();

    oh.gotFriend = false;
    oh.myFriendLineStart = null;
    oh.myFriendLineEnd = null;

    oh.gotLunch = false;
    oh.myLunchLineStart = null;
    oh.myLunchLineEnd = null;

    oh.gotSnack = false;
    oh.mySnackLineStart = null;
    oh.mySnackLineEnd = null;

    int myFriendIndex = gcHeap.getObjectPool(oh.objectPos);

    if (myFriendIndex != 0) {
      if (recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorHasNotStarted() ||
          recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorIsDone()) {
            g.setPaintMode();
            g.setColor(Color.blue);
            if(oh.myFriendLineColor == null)
              oh.myFriendLineColor = Color.white;
            g.setXORMode(oh.myFriendLineColor);
```

```
      }
      ObjectHandle myFriend = gcHeap.getObjectHandle(myFriendIndex);
      g.drawLine(fishNose.x, fishNose.y, myFriend.fish.getFishPosition().x,
myFriend.fish.getFishPosition().y);
      oh.gotFriend = true;
      oh.myFriendLineStart = new Point(fishNose.x, fishNose.y);
      oh.myFriendLineEnd = new Point(myFriend.fish.getFishPosition().x,
myFriend.fish.getFishPosition().y);
    }

    if (fishIcon.getFishColor() == Color.yellow) {
      g.setPaintMode();
      continue;
    }

    int myLunchIndex = gcHeap.getObjectPool(oh.objectPos + 1);

    if (myLunchIndex != 0) {
     if (recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorHasNotStarted() ||
        recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorIsDone()) {
        g.setPaintMode();
        g.setColor(Color.blue);
        if(oh.myLunchLineColor == null)
          oh.myLunchLineColor = Color.white;
        g.setXORMode(oh.myLunchLineColor);
      }
      ObjectHandle myLunch = gcHeap.getObjectHandle(myLunchIndex);
      g.drawLine(fishNose.x, fishNose.y, myLunch.fish.getFishPosition().x,
myLunch.fish.getFishPosition().y);
      oh.gotLunch = true;
      oh.myLunchLineStart = new Point(fishNose.x, fishNose.y);
      oh.myLunchLineEnd = new Point(myLunch.fish.getFishPosition().x,
myLunch.fish.getFishPosition().y);
    }

    if (fishIcon.getFishColor() == Color.cyan) {
      g.setPaintMode();
      continue;
    }

    int mySnackIndex = gcHeap.getObjectPool(oh.objectPos + 2);

    if (mySnackIndex != 0) {
     if (recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorHasNotStarted() ||
        recoleccion.getCurrentGCState() == recoleccion.getGarbageCollectorIsDone()) {
        g.setPaintMode();
        g.setColor(Color.blue);
        if(oh.mySnackLineColor == null)
          oh.mySnackLineColor = Color.white;
        g.setXORMode(oh.mySnackLineColor);
      }
      ObjectHandle mySnack = gcHeap.getObjectHandle(mySnackIndex);
      g.drawLine(fishNose.x, fishNose.y, mySnack.fish.getFishPosition().x,
mySnack.fish.getFishPosition().y);
      oh.gotSnack = true;
      oh.mySnackLineStart = new Point(fishNose.x, fishNose.y);
      oh.mySnackLineEnd = new Point(mySnack.fish.getFishPosition().x,
mySnack.fish.getFishPosition().y);
    }
```

```
            g.setPaintMode();
          }
        }
    }
```

Se han creado varios métodos para realizar la ejecución a través de hilos:

```
public void seguidoGC() {
      indiceRecoleccion = (indiceRecoleccion + 1) % cicloRecoleccion;
      recoleccion = listaRecoleccion[indiceRecoleccion];
      recoleccion.resetGCState();
      while(!parar && (recoleccion.getCurrentGCState()!=recoleccion.getGarbageCollectorIsDone())){
        recoleccion.nextGCStep();
        repaint();
        try {
            thread.sleep(1000);
        }  catch (InterruptedException e) {}
     }
     recoleccion.resetGCState();
    }

    public void pausa() {
      parar=!parar;
    }

    public void start() {
       // if (thread.isAlive()) {
            thread = new Thread(this);
            thread.start();
       // }
    }

    public void stop() {
        if (thread != null) {
            thread.stop();
            thread = null;
        }
    }

    public void run() {
      seguidoGC();
    }
```

y, por último, la implementación del método *resetGC* ha sido trasladado a las clases que implementan la recolección:

```
public void resetGCState(){
      recoleccion.resetGCState();
}
```

d) Clase *GCHeap*.

La clase *GCHeap* ha sido transformada para permitir que el heap esté dividido en generaciones y que haya promoción de una generación a la siguiente. Para conseguir esto, se ha movido el código original a la clase *EspacioAlmacenamiento* y ésta se declarado como hija suya, añadiendo el atributo *frontera*, que marca el límite entre las dos generaciones y el método *slideObjectDown*, que implementa la promoción de un objeto de la generación nueva a la generación antigua:

```
public class GCHeap extends EspacioAlmacenamiento{


private int frontera;

  GCHeap(int hndlPoolSize, int objPoolSize) {

    super(hndlPoolSize, objPoolSize);
    frontera = 2*objPoolSize/3;
    setObjectPool(0,formMemBlockHeader(true, frontera));
    setObjectPool(frontera,formMemBlockHeader(true, getObjectPoolSize() - frontera));
  }

  public int dameFrontera()
  {
   return frontera;
  }

  // Returns true if an object was slid
  public boolean slideObjectDown(ObjectHandle oh) {

    boolean retVal = false;


    int i = frontera;

      int header = getObjectPool(i);
      boolean free = false;
      if (getMemBlockFree(header)) {
        free = true;
      }
      int length = getMemBlockLength(header);



      while (!free) {

        // This memory block is not free, so continue by looking at the next memory
        // block. Add length to the current index into the constant pool to get the
        // index of the next memory block header.
        i += length;
        header = getObjectPool(i);
        free = false;
        if (getMemBlockFree(header)) {
          free = true;
        }
        length = getMemBlockLength(header);
      }
```

```
        // We have found a free block. First, concatenate any other free blocks that may
        // be contiguous to this one.
        while ((i + length) < getObjectPoolSize() && getMemBlockFree(getObjectPool(i + length))) {

            // The next memory block is also free, so concatenate with the previous
            // one. This is done by extending the size of the previous memory block
            // to include the next block.


            length += getMemBlockLength(getObjectPool(i + length));
            setObjectPool(i,formMemBlockHeader(true, length));
        }

        // See if an object exists at the next location, if not break out of the loop
        // and return false. There are no other objects that can be slid.


        int sliderLength = getMemBlockLength(getObjectPool(oh.objectPos - 1));

        for (int j = 0; j < sliderLength - 1; ++j) {
            setObjectPool(i + j + 1, getObjectPool(oh.objectPos + j));
        }

        setObjectPool(i, formMemBlockHeader(false, sliderLength));
        setObjectPool(oh.objectPos - 1, formMemBlockHeader(true, sliderLength));

        if(length > sliderLength)
          setObjectPool(i + sliderLength,formMemBlockHeader(true, length - sliderLength));

        oh.objectPos = i + 1;

        retVal = true;



    return retVal;
  }


}
```

e) Clase *HeapOfFish*.

En la clase *HeapOfFish* se han añadido los objetos de las clases *EspacioRegiones*, *MemoriaDeInstrucciones*, *MotorDeEjecucion* y *ReferenciasIniciales*:

```
public class HeapOfFish extends Applet {

  ....

  private EspacioRegiones espRegiones = new EspacioRegiones(gcHeap.getObjectPoolSize());
  private MemoriaDeInstrucciones mi = new MemoriaDeInstrucciones(15);
  private LocalVariables localVars = new LocalVariables();
```

```
   private MotorDeEjecucion motor = new MotorDeEjecucion(mi,gcHeap);
   private ReferenciasIniciales refIni = new ReferenciasIniciales(gcHeap,localVars);
```

Además de crear estos objetos, hay que pasar sus referencias a los diferentes objetos; por lo tanto, se pasan al objeto de la clase *HeapOfFishCanvases* y luego se van distribuyendo:

```
....
private HeapOfFishCanvases canvases = new HeapOfFishCanvases(gcHeap, mi, refIni,
controlPanel.getTextArea(), espRegiones, motor);
....
```

Por otra parte, al haberse introducido la opción de *Cargar Código*, se ha tenido que modificar el método *action*:

```
public boolean action(Event evt, Object arg) {
       if (evt.target instanceof Checkbox) {


          Checkbox cb = (Checkbox) evt.target;
          String cbname = cb.getLabel();
          if (cbname.equals(HeapOfFishStrings.allocateFish)) {
             if (!currentHeapOfFishMode.equals(HeapOfFishStrings.allocateFish)) {
                controlPanel.getTextArea().setText(HeapOfFishStrings.allocateFishInstructions);
                canvases.setMode(HeapOfFishStrings.allocateFish);
             }
          }
          else if (cbname.equals(HeapOfFishStrings.assignReferences)) {
             if (!currentHeapOfFishMode.equals(HeapOfFishStrings.assignReferences)) {
                canvases.setMode(HeapOfFishStrings.assignReferences);
             }
          }
          else if (cbname.equals(HeapOfFishStrings.garbageCollect)) {
             if (!currentHeapOfFishMode.equals(HeapOfFishStrings.garbageCollect)) {
                canvases.setMode(HeapOfFishStrings.garbageCollect);
             }
          }
          else if (cbname.equals(HeapOfFishStrings.compactHeap)) {
             if (!currentHeapOfFishMode.equals(HeapOfFishStrings.compactHeap)) {
                controlPanel.getTextArea().setText(HeapOfFishStrings.compactHeapInstructions);
                canvases.setMode(HeapOfFishStrings.compactHeap);
             }
          }
          else if (cbname.equals(HeapOfFishStrings.swim)) {
             if (!currentHeapOfFishMode.equals(HeapOfFishStrings.swim)) {
                controlPanel.getTextArea().setText("");
                canvases.setMode(HeapOfFishStrings.swim);
             }
          }
          else if (cbname.equals(HeapOfFishStrings.cargarCodigo)) {
             if (!currentHeapOfFishMode.equals(HeapOfFishStrings.cargarCodigo)) {
                controlPanel.getTextArea().setText("");
                canvases.setMode(HeapOfFishStrings.cargarCodigo);
```

```
            }
        }
        currentHeapOfFishMode = cbname;
        canvases.repaint();
    }
    return true;
}
```

f) Clase *HeapOfFishCanvases*.

En la clase *HeapOfFishCanvases* ha sido necesario mandar las referencias a estos objetos nuevos a cada uno de los paneles que componen el applet. Para ello, se modifican los constructores en cada uno de ellos y se invoca al *new* con estos parámetros. Por otra parte, ahora disponemos de la opción *Cargar Codigo*, por lo que hay que crear un objeto de la clase *PanelCargarCodigo*.

```
class HeapOfFishCanvases extends Panel {

    private CardLayout cl = new CardLayout();
    private SwimmingFishCanvas swimmers = new SwimmingFishCanvas();
    private String currentMode = HeapOfFishStrings.swim;
    private AllocateFishPanel allocateFishPanel;
    private AssignReferencesPanel assignRefsPanel;
    private GarbageCollectPanel garbageCollectPanel;
    private PanelCargarCodigo panelCargarCodigo;

    HeapOfFishCanvases(GCHeap gcHeap, MemoriaDeInstrucciones memIns, ReferenciasIniciales
localVars, HeapOfFishTextArea ta, EspacioRegiones espRegiones, MotorDeEjecucion me) {

        assignRefsPanel = new AssignReferencesPanel(gcHeap, memIns, localVars, ta, espRegiones,me);
        allocateFishPanel = new AllocateFishPanel(gcHeap, memIns, ta, espRegiones);
        garbageCollectPanel = new GarbageCollectPanel(gcHeap, espRegiones, localVars, ta);
        panelCargarCodigo = new
PanelCargarCodigo(gcHeap,memIns,localVars.dameVariablesLocales(),ta,me,espRegiones);
        setLayout(cl);
        add(HeapOfFishStrings.allocateFish, allocateFishPanel);
        add(HeapOfFishStrings.assignReferences, assignRefsPanel);
        add(HeapOfFishStrings.garbageCollect, garbageCollectPanel);
        add(HeapOfFishStrings.compactHeap, new CompactHeapPanel(gcHeap, ta));
        add(HeapOfFishStrings.cargarCodigo,panelCargarCodigo);
        add(HeapOfFishStrings.swim, swimmers);
        me.ponAllocateFishPanel(allocateFishPanel);
        swimmers.start();
        cl.show(this, HeapOfFishStrings.swim);
    }

……
```

g) Clase *HeapOfFishModeCheckBoxPanel*.

En la clase *HeapOfFishModeCheckBoxPanel* se ha añadido la opción *Cargar Código* y se ha eliminado la opción *Compact Heap*. Esta clase ha quedado así:

```
class HeapOfFishModeCheckboxPanel extends Panel {

   private CheckboxGroup cbg = new CheckboxGroup();

   HeapOfFishModeCheckboxPanel() {

      setBackground(Color.lightGray);

      setLayout(new GridLayout(5, 1));
      add(new Checkbox(HeapOfFishStrings.allocateFish, cbg, false));
      add(new Checkbox(HeapOfFishStrings.assignReferences, cbg, false));
      add(new Checkbox(HeapOfFishStrings.garbageCollect, cbg, false));
      //add(new Checkbox(HeapOfFishStrings.compactHeap, cbg, false));
      add(new Checkbox(HeapOfFishStrings.cargarCodigo, cbg, false));
      add(new Checkbox(HeapOfFishStrings.swim, cbg, true));
   }

   public CheckboxGroup getModeCheckboxGroup() {
      return cbg;
   }
}
```

h) Clase *HeapOfFishStrings*.

La clase *HeapOfFishStrings* solamente contiene atributos estáticos, de los cuales han sido sido añadidos los siguientes:

```
public final static String cargarCodigo = "Cargar Código";
public final static String pasandoNuevosObjetos = "Pasando por nuevos objetos.";
   public final static String recorriendoObjetosApuntadosDesdeRegiones = "Recorriendo objetos 
apuntados desde regiones";
```

i) Clase *LinkFishCanvas*.

En la clase *LinkFishCanvas* se han añadido métodos para ejecutar toda instrucción que tenga relación con las referencias entre objetos o entre variables y objetos. Estos métodos son:

- *asignarReferenciasEntrePecesIndirecto*, que implementa las instrucciones *am*, *cm* y *sn*. Este método lee de los operandos los alias de las referencias utilizados, con ellos va a la tabla contenida en el objeto de la clase *MemoriaDeInstrucciones*, lee las direcciones y con ellas invoca al método *asignarReferenciasEntrePeces*.

- *asignarReferenciasEntrePecesInmediato*, que implementa las instrucciones *ami*, *cmi* y *sni*. Este método invoca al método *asignarReferenciasEntrePeces* con las direcciones contenidas en los operandos, pues son inmediatas.
- *asignarReferenciasEntrePecesDestinoInmediato*, que implementa las instrucciones *amdi*, *cmdi* y *sndi*. En estas instrucciones tan solo el destino es un operando inmediato, por tanto lee la dirección destino y el alias del origen, con el alias del origen obtiene la dirección origen y con ambas direcciones se invoca al método *asignarReferenciasEntrePeces*.
- *anularReferenciasPeces*. Este método, además de implementar las instrucciones *nam*, *ncm* y *nsn*, comprueba si la referencia que se anula es de un objeto de la generación antigua a uno de la generación nueva y se elimina de la lista una instancia si es así. También comprueba si la referencia que se anula es de un objeto de la región a un objeto del heap y en ese caso se invoca al método *chequearReferencias* de la interface *Tarea*, que elimina una instancia de la lista en el caso en que la tarea no sea crítica.
- *asignarPecesAReferencias*. Este método implementa la instrucción *asr*.
- *anularReferencias*. Este método implementa la instrucción *nr*.

Se ha implementado un método privado cuyo nombre es *asignarReferenciasEntrePeces*, que se ocupa de hacer todo tipo de asignación de referencias entre peces a partir de sus direcciones origen y destino y además se ocupa de averiguar a que objeto estaba apuntando el objeto origen antes de lanzar a ejecución esta asignación y, realiza las siguientes acciones:
- Si el origen es un objeto de la generación antigua y el destino anterior era un objeto de la generación nueva, se elimina una instancia de la dirección destino anterior en la lista.
- Si el origen es un objeto de la generación antigua y el destino actual es un objeto de la generación nueva, se inserta una instancia de la dirección destino actual en la lista.
- Si el origen es un objeto de la región, se invoca al método *chequearReferencias* de la interface Tarea, para que inserte en la lista una instancia de la dirección destino actual si el destino actual está en el heap y borre una instancia de la dirección destino anterior si el destino anterior estaba en el heap y la tarea no es crítica.
- Si el origen es un objeto que ha sido coloreado de negro por el garbage colector, comprueba si el destino está en blanco, y si es así, colorea el destino de color gris.

Este método solamente obtiene la región al leer el operando origen, ya que si está en el heap no puede haber una referencia a una región y si está en una región el destino no puede estar en otra región; por lo tanto, no vamos a necesitar acceder a una región a la que no hayamos accedido en el origen, en el caso en que hayamos accedido a una región.

Además de esto, los métodos *mouseUp*, *mouseDown* y *mouseDrag* han sido sustituidos por los de la clase *MoveFishCanvas* y se ha añadido código para que se puedan mover también los objetos situados en la región.

```
public class LinkFishCanvas extends AssignReferencesCanvas
{

   private boolean iconClicked = false;
```

```
   private boolean yellowLocalVarClicked = false;
   private boolean blueLocalVarClicked = false;
   private boolean redLocalVarClicked = false;

   private Point posOfMouseInsideIconWhenFirstPressed = new Point(0, 0);
   private int objectIndexOfFishIconThatWasClicked;

   private boolean dragging = false;
   private Point currentMouseDragPosition = new Point(0, 0);
   private boolean mouseIsOverAnIconThatCanBeDroppedUpon = false;
   private int objectIndexOfIconThatCanBeDroppedUpon;

   private MemoriaDeInstrucciones mi;
   private ListaReferencias lista;




   LinkFishCanvas(GCHeap heap, MemoriaDeInstrucciones memIns, ReferenciasIniciales locVars,
HeapOfFishTextArea ta, EspacioRegiones regiones) {
       gcHeap = heap;
       region = null;
       espRegiones = regiones;
       localVars = locVars.dameVariablesLocales();
       controlPanelTextArea = ta;
       mi = memIns;
       lista = locVars.dameListaObjetosNuevos();

   }



   private void asignarReferenciasEntrePeces(Tarea t, int dir1, int dir2nueva, int offset)
   {
     ObjectHandle oh1,oh2;
     Point p1,p2;
     ListaReferencias listaR = null;

     int dir2antigua = -1;




     if(dir1 > gcHeap.getObjectPoolSize())
     {

       region = espRegiones.dameRegionQueContengaLaDireccion(dir1);
       oh1 = region.getObjectHandle(dir1);
       dir2antigua = region.getObjectPool(oh1.objectPos + offset);
       region.setObjectPool(oh1.objectPos + offset, dir2nueva);
       listaR = espRegiones.dameListaReferencias(region);
     }
     else
     {

       oh1 = gcHeap.getObjectHandle(dir1);
       dir2antigua = gcHeap.getObjectPool(oh1.objectPos + offset);
       gcHeap.setObjectPool(oh1.objectPos + offset, dir2nueva);
     }
```

```
if((oh1.gotFriend && offset == 0) ||
  (oh1.gotLunch && offset == 1) ||
  (oh1.gotSnack && offset == 2))
{

     if(dir2antigua <= gcHeap.getObjectPoolSize())
     {
       oh2 = gcHeap.getObjectHandle(dir2antigua);
       if(oh1.objectPos > gcHeap.dameFrontera() && oh2.objectPos < gcHeap.dameFrontera())
         lista.borrar(dir2antigua);
     }
     else
       oh2 = region.getObjectHandle(dir2antigua);



}
else
  dir2antigua = gcHeap.getObjectPoolSize() + 1;
if(dir2nueva <= gcHeap.getObjectPoolSize())
{
  oh2 = gcHeap.getObjectHandle(dir2nueva);
  if(oh1.objectPos > gcHeap.dameFrontera() && oh2.objectPos < gcHeap.dameFrontera())
    lista.insertar(dir2nueva);

}
else
  oh2 = region.getObjectHandle(dir2nueva);
//oh2 = gcHeap.getObjectHandle(dir2nueva);
if ((oh1.myColor == Color.black)
          &&((oh2.myColor==Color.white))){
         oh2.myColor=Color.gray;
         oh1.myFriendLineColor=Color.black;
         oh1.myLunchLineColor=Color.black;
         oh1.mySnackLineColor=Color.black;
         oh2.myFriendLineColor=Color.white;
         oh2.myLunchLineColor=Color.white;
         oh2.mySnackLineColor=Color.white;
        }
        if ((oh1.myColor == Color.gray)
          &&((oh2.myColor==Color.white))){

         oh2.myFriendLineColor=Color.white;
         oh2.myLunchLineColor=Color.white;
         oh2.mySnackLineColor=Color.white;
        }




repaint();
t.chequearReferencias(dir1,dir2antigua,dir2nueva,offset,gcHeap,listaR);
}

public synchronized void asignarReferenciasEntrePecesIndirecto(Instruccion ins,Tarea t,int offset,
MemoriaDeInstrucciones mem)
{
```

```
    int dir1,dir2;
    dir1 = mem.dameReferencia(ins.dameOperando(0));
    dir2 = ins.dameOperando(1);
    dir2 = mem.dameReferencia(dir2);
    if(dir1 == 0 || dir2 == 0)
      throw new IllegalStateException();
    asignarReferenciasEntrePeces(t,dir1,dir2,offset);
  }

  public synchronized void asignarReferenciasEntrePecesInmediato(Instruccion ins,Tarea t,int offset)
  {
    asignarReferenciasEntrePeces(t,ins.dameOperando(0),ins.dameOperando(1),offset);
  }

  public synchronized void asignarReferenciasEntrePecesDestinoInmediato(Instruccion ins,Tarea t,int
offset, MemoriaDeInstrucciones mem)
  {
    int dir1,dir2;
    dir1 = mem.dameReferencia(ins.dameOperando(0));
    dir2 = ins.dameOperando(1);
    if(dir1 == 0 || dir2 == 0)
      throw new IllegalStateException();
    asignarReferenciasEntrePeces(t,dir1,dir2,offset);
  }

  public synchronized void anularReferenciasPeces(Instruccion ins, Tarea t, int offset,
MemoriaDeInstrucciones mem)
  {
    int dir1,dir2;
    ObjectHandle oh1,oh2;
    ListaReferencias listaR;


    dir1 = mem.dameReferencia(ins.dameOperando(0));
    listaR = null;
    if(dir1 > gcHeap.getObjectPoolSize())
    {
      region = espRegiones.dameRegionQueContengaLaDireccion(dir1);
      oh1 = region.getObjectHandle(dir1);
      dir2 = region.getObjectPool(oh1.objectPos + offset);
      region.setObjectPool(oh1.objectPos + offset, 0);
      listaR = espRegiones.dameListaReferencias(region);
    }
    else
    {
      oh1 = gcHeap.getObjectHandle(dir1);
      dir2 = gcHeap.getObjectPool(oh1.objectPos + offset);
      gcHeap.setObjectPool(oh1.objectPos + offset, 0);
    }
    if(dir2 <= gcHeap.getObjectPoolSize())
    {
       oh2 = gcHeap.getObjectHandle(dir2);
       if(oh1.objectPos > gcHeap.dameFrontera() && oh2.objectPos < gcHeap.dameFrontera())
         lista.borrar(dir2);
    }
    else
       oh2 = region.getObjectHandle(dir2 - gcHeap.getObjectPoolSize());
```

```
  switch(offset)
  {
    case 0: oh1.gotFriend = false;
            break;
    case 1: oh1.gotLunch = false;
            break;
    case 2: oh1.gotSnack = false;
            break;
  }
  repaint();
  t.chequearReferencias(dir1,dir2,gcHeap.getObjectPoolSize() + 1,offset,gcHeap,listaR);
}

 public synchronized void asignarPecesAReferencias(Instruccion ins, MemoriaDeInstrucciones mem)
{
  int direccion = ins.dameOperando(0);
  direccion = mem.dameReferencia(direccion);
  ObjectHandle oh1;



  if(direccion <= gcHeap.getObjectPoolSize())
    oh1 = gcHeap.getObjectHandle(direccion);
  else
  {
    region = espRegiones.dameRegionQueContengaLaDireccion(direccion);
    oh1 = region.getObjectHandle(direccion);
  }

  switch(ins.dameOperando(1))
  {
      case 1:
              localVars.yellowFish = direccion;
              break;

      case 2:
              localVars.blueFish = direccion;
              break;

      case 3:
              localVars.redFish = direccion;
              break;
  }






  oh1.myFriendLineColor=Color.black;
  oh1.myLunchLineColor=Color.black;
  oh1.mySnackLineColor=Color.black;
  repaint();
}

public synchronized void anularReferencias(Instruccion ins)
{
```

```
  switch(ins.dameOperando(0))
  {
    case 1:
      localVars.yellowFish = 0;
      break;

    case 2:
      localVars.blueFish = 0;
      break;

    case 3:
      localVars.redFish = 0;
      break;
  }

  repaint();
}

public boolean mouseDown(Event evt, int x, int y) {

   // Find out if the mouse went down inside an icon's hot area. Count down from the
   // top of the heap list so that fish that are drawn later will be found first. This
   // is because if two fish are overlapping, the one later in the array will be
   // drawn second and appear to be on top. The top fish will be found first by this
   // for loop.
   for (int i = gcHeap.getHandlePoolSize() - 1; i >= 0; --i) {
       ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
       if (!oh.free) {
           Point o = oh.fish.getFishPosition();
           if (x >= o.x && x < o.x + oh.fish.getFishWidth() && y >= o.y
               && y < o.y + oh.fish.getFishHeight()) {

               iconClicked = true;
               objectIndexOfFishIconThatWasClicked = i + 1;
               posOfMouseInsideIconWhenFirstPressed.x = x - o.x;
               posOfMouseInsideIconWhenFirstPressed.y = y - o.y;
               //fishObjectThatWasClicked = oh;
               break;
           }
       }
   }

   Region[] regiones = espRegiones.dameRegionesOcupadas();
   for(int j=0;!iconClicked && j < regiones.length;j++)
     for (int i = regiones[j].getHandlePoolSize() - 1; i >= 0; --i) {
       ObjectHandle oh = regiones[j].dameManejador(i + 1);
       if (!oh.free) {
           Point o = oh.fish.getFishPosition();
           if (x >= o.x && x < o.x + oh.fish.getFishWidth() && y >= o.y
               && y < o.y + oh.fish.getFishHeight()) {

               iconClicked = true;
               objectIndexOfFishIconThatWasClicked = i + 1 + regiones[j].dameDireccionInicio();
               posOfMouseInsideIconWhenFirstPressed.x = x - o.x;
               posOfMouseInsideIconWhenFirstPressed.y = y - o.y;
               //fishObjectThatWasClicked = oh;
               break;
           }
       }
```

```
        }
        return true;
    }

    public boolean mouseUp(Event evt, int x, int y) {

        if (iconClicked == false) {
            return true;
        }

        iconClicked = false;
        FishIcon fishIconThatWasClicked;
        Color colorOfClickedFish;


        if(objectIndexOfFishIconThatWasClicked < gcHeap.getObjectPoolSize())
        {
          fishIconThatWasClicked = gcHeap.getObjectHandle(objectIndexOfFishIconThatWasClicked).fish;
          colorOfClickedFish =
gcHeap.getObjectHandle(objectIndexOfFishIconThatWasClicked).fish.getFishColor();
        }
        else
        {
          region =
espRegiones.dameRegionQueContengaLaDireccion(objectIndexOfFishIconThatWasClicked);
          fishIconThatWasClicked = region.getObjectHandle(objectIndexOfFishIconThatWasClicked).fish;
          colorOfClickedFish =
region.getObjectHandle(objectIndexOfFishIconThatWasClicked).fish.getFishColor();
        }

        if (dragging) {
            dragging = false;
            // Clear old drag icon.
            Graphics g = getGraphics();
            g.setColor(Color.blue);
            g.setXORMode(colorOfClickedFish);
            fishIconThatWasClicked.drawFishOutline(g, currentMouseDragPosition.x,
                currentMouseDragPosition.y);
            fishIconThatWasClicked.moveFish(currentMouseDragPosition.x,
                currentMouseDragPosition.y);
            repaint();
        }

        return true;
    }

    public boolean mouseDrag(Event evt, int x, int y) {

        if (!iconClicked) {
            return true;
        }

        FishIcon fishIconThatWasClicked;
        Color colorOfClickedFish;


        if(objectIndexOfFishIconThatWasClicked < gcHeap.getObjectPoolSize())
        {
          fishIconThatWasClicked = gcHeap.getObjectHandle(objectIndexOfFishIconThatWasClicked).fish;
```

```
        colorOfClickedFish =
gcHeap.getObjectHandle(objectIndexOfFishIconThatWasClicked).fish.getFishColor();
      }
      else
      {
        region =
espRegiones.dameRegionQueContengaLaDireccion(objectIndexOfFishIconThatWasClicked);
        fishIconThatWasClicked = region.getObjectHandle(objectIndexOfFishIconThatWasClicked).fish;
        colorOfClickedFish =
region.getObjectHandle(objectIndexOfFishIconThatWasClicked).fish.getFishColor();
      }

      // Don't start dragging unless the mouse has moved a threshold number of
      // pixels in x or y.
      if (!dragging) {
         int thresholdPixels = 5;
         Point iconOrigin = fishIconThatWasClicked.getFishPosition();
         int xOriginalClick = iconOrigin.x + posOfMouseInsideIconWhenFirstPressed.x;
         int yOriginalClick = iconOrigin.y + posOfMouseInsideIconWhenFirstPressed.y;
         int xDifference = x - xOriginalClick;
         if (xDifference < 0) {
            xDifference = 0 - xDifference;
         }
         int yDifference = y - yOriginalClick;
         if (yDifference < 0) {
            yDifference = 0 - yDifference;
         }
         if (xDifference < thresholdPixels && yDifference < thresholdPixels) {
            return true;
         }
      }

      Graphics g = getGraphics();
      g.setColor(Color.blue);
      g.setXORMode(colorOfClickedFish);

      if (!dragging) {
         dragging = true;
      }
      else {
         // Clear old drag icon.
         fishIconThatWasClicked.drawFishOutline(g, currentMouseDragPosition.x,
            currentMouseDragPosition.y);
      }

      int xNew = x - posOfMouseInsideIconWhenFirstPressed.x;
      int yNew = y - posOfMouseInsideIconWhenFirstPressed.y;

      // Don't let any of the icon go off of either the left
      // or the right side of the Component.
      if (xNew < xFishAreaStart) {
         xNew = xFishAreaStart;
      }
      else if (xNew + fishIconThatWasClicked.getFishWidth() - 1 > size().width) {
         xNew = size().width - fishIconThatWasClicked.getFishWidth() - 1;
      }

      // Don't let any of the icon go off of either the top
      // or the bottom side of the Component.
      if (yNew < 0) {
```

```
            yNew = 0;
        }
        else if (yNew + fishIconThatWasClicked.getFishHeight() - 1 > size().height) {

            yNew = size().height - fishIconThatWasClicked.getFishHeight() - 1;
        }

        // Then draw the new outline around the icon
        fishIconThatWasClicked.drawFishOutline(g, xNew, yNew);
        currentMouseDragPosition.x = xNew;
        currentMouseDragPosition.y = yNew;

        g.setPaintMode();
        return true;
    }



}
```

j) Clase *ObjectHandle*.

En la clase *ObjectHandle* se ha añadido un atributo para contabilizar el número de veces que un objeto sobrevive al garbage collector. Este atributo es *vecesSobrevividasAlGarbageCollector*:

```
public class ObjectHandle {

....

public int vecesSobrevividasAlGarbageCollector;
}
```

k) Clase *PoolsCanvas*.

En la clase *PoolsCanvas* se ha añadido una referencia al objeto de la clase *EspacioRegiones* y se ha añadido código para pintar tanto los manejadores como los objetos que se creen en las regiones.

```
public class PoolsCanvas extends Canvas {

    private GCHeap gcHeap;
    private EspacioRegiones espRegiones;
    private final int poolImageInsets = 5;

    PoolsCanvas(GCHeap heap, EspacioRegiones eRegiones) {
        gcHeap = heap;
        espRegiones = eRegiones;
    }
```

```
    public void paint(Graphics g) {

        // First calculate the positions of the goodies on the canvas based on the width
        // and height of the canvas.
        Dimension dim = getSize();

        //dim.height = dim.height/2;

        // Divide width into three equal portions. The left portion will hold the handle pool.
        // The right portion will hold the object pool. The middle portion will have arrows
        // that go from valid handles to their respective objects.
        int xHandlePoolPortion = 0;
        int xArrowPortion = dim.width / 3;
        int xObjectPoolPortion = 2 * xArrowPortion;


        Font font = getFont();
        FontMetrics fm = getFontMetrics(font);

        int labelHeight = fm.getAscent() + fm.getDescent() + (2 * poolImageInsets);

        int heightAvailableForPools = (dim.height/2)  - labelHeight - poolImageInsets;
        int objectPoolIntsCount = gcHeap.getObjectPoolSize();
        int handlePoolIntsCount = gcHeap.getHandlePoolSize() * 2;

        int xCuadrante = dim.width / 2;
        int yCuadrante = dim.height/4;

        int maxIntsCount = objectPoolIntsCount;
        if (maxIntsCount < handlePoolIntsCount) {
            maxIntsCount = handlePoolIntsCount;
        }

        int yPixelsPerInt = heightAvailableForPools / maxIntsCount;

        int handlePoolHeight = handlePoolIntsCount * yPixelsPerInt;
        int objectPoolHeight = objectPoolIntsCount * yPixelsPerInt;

        int poolsWidth = xArrowPortion - poolImageInsets;

        int xTextStart = poolsWidth - fm.stringWidth(HeapOfFishStrings.handlePool);
        if (xTextStart < 0) {
            xTextStart = 0;
        }
        xTextStart /= 2;

        int yStart = ((dim.height/2) - handlePoolHeight - labelHeight - (2 * poolImageInsets)) / 2;
        if (yStart < 0) {
            yStart = 0;
        }
        g.setColor(Color.white);
        g.drawString(HeapOfFishStrings.handlePool, poolImageInsets + xTextStart, poolImageInsets +
yStart + fm.getAscent());
        int yHandlePoolRect = yStart + labelHeight;
        g.fillRect(xHandlePoolPortion + poolImageInsets, yHandlePoolRect, poolsWidth,
handlePoolHeight);
```

```
        xTextStart = poolsWidth - fm.stringWidth(HeapOfFishStrings.objectPool);
        if (xTextStart < 0) {
            xTextStart = 0;
        }
        xTextStart /= 2;

        yStart = ((dim.height/2) - objectPoolHeight - labelHeight - (2 * poolImageInsets)) / 2;
        if (yStart < 0) {
            yStart = 0;
        }

        //g.setColor(Color.white);
        g.drawString(HeapOfFishStrings.objectPool, xObjectPoolPortion + xTextStart, poolImageInsets +
yStart + fm.getAscent());
        int yObjectPoolRect = yStart + labelHeight;
        //g.setColor(Color.white);
        g.fillRect(xObjectPoolPortion, yObjectPoolRect, poolsWidth, objectPoolHeight);

        // Draw the headers in the object pool
        g.setColor(Color.black);
        int i = 0;
        while (i < objectPoolIntsCount) {

            for (int j = 0; j < yPixelsPerInt; ++j) {
                int yLinePos = yObjectPoolRect + (i * yPixelsPerInt) + j;
                g.drawLine(xObjectPoolPortion, yLinePos, xObjectPoolPortion + poolsWidth - 1, yLinePos);
            }
            int header = gcHeap.getObjectPool(i);
            int length = gcHeap.getMemBlockLength(header);
            if (length <= 0) { // In case object pool gets corrupted, don't hang up.
                break;
            }
            i += length;
        }

        for (i = 0; i < gcHeap.getHandlePoolSize(); ++i) {
            ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
            if (!oh.free) {

                Color color = Color.red;
                int objectSizeInInts = 3;
                if (oh.fish.getFishColor() == Color.cyan) {
                    color = Color.cyan;
                    objectSizeInInts = 2;
                }
                else if (oh.fish.getFishColor() == Color.yellow) {
                    color = Color.yellow;
                    objectSizeInInts = 1;
                }
                g.setColor(color);

                // Draw bar across handle pool
                for (int j = 0; j < yPixelsPerInt * 2; ++j) {
                    int yLinePos = yHandlePoolRect + (i * yPixelsPerInt * 2) + j;
                    g.drawLine(xHandlePoolPortion + poolImageInsets, yLinePos, xHandlePoolPortion +
poolImageInsets + poolsWidth - 1, yLinePos);
                }

                // Draw colored bars to represent object in the object pool
                for (int j = 0; j < yPixelsPerInt * objectSizeInInts; ++j) {
```

```
                int yLinePos = yObjectPoolRect + (oh.objectPos * yPixelsPerInt) + j;
                g.drawLine(xObjectPoolPortion, yLinePos, xObjectPoolPortion + poolsWidth - 1,
yLinePos);
            }

            // Draw a line from the handle to the object to show that the handle
            // points to the object.
            int yArrowStart = yHandlePoolRect + (i * yPixelsPerInt * 2) + yPixelsPerInt;
            int yArrowEnd = yObjectPoolRect + (oh.objectPos * yPixelsPerInt) + ((yPixelsPerInt *
objectSizeInInts) / 2);
            g.drawLine(xHandlePoolPortion + poolImageInsets + poolsWidth + 2, yArrowStart,
                xObjectPoolPortion - 3, yArrowEnd);
        }
    }

    Region actual;

    for(int nRegion = 1;nRegion <= 4;nRegion++)
    {
      actual = espRegiones.dameRegion(nRegion);
      if(actual != null)
      {
        xHandlePoolPortion = 0 + (nRegion + 1)%2 * (xCuadrante + 8);
        xArrowPortion = xCuadrante / 3;
        xObjectPoolPortion = 2 * xArrowPortion + (nRegion + 1)%2 * xCuadrante;


        //heightAvailableForPools = (dim.height/4)  - labelHeight - poolImageInsets;
        objectPoolIntsCount = actual.getObjectPoolSize();
        handlePoolIntsCount = actual.getHandlePoolSize() * 2;

        maxIntsCount = objectPoolIntsCount;
        if (maxIntsCount < handlePoolIntsCount) {
          maxIntsCount = handlePoolIntsCount;
        }

        yPixelsPerInt = heightAvailableForPools /  (2 * maxIntsCount);

        handlePoolHeight = handlePoolIntsCount * yPixelsPerInt;
        objectPoolHeight = objectPoolIntsCount * yPixelsPerInt;

        poolsWidth = xArrowPortion - poolImageInsets;

        xTextStart = poolsWidth - fm.stringWidth(HeapOfFishStrings.handlePool) + (nRegion + 1)%2 *
(xCuadrante + 8);
        if (xTextStart < 0) {
          xTextStart = 0;
        }
        xTextStart = xTextStart/2 - (nRegion%2 - 1) * xCuadrante/2;

        yStart = ((dim.height/4) * (3 + nRegion/3) - handlePoolHeight - labelHeight - (2 *
poolImageInsets)) / 1;
        if (yStart < yCuadrante * (2 + nRegion/3)) {
          yStart = yCuadrante * (2 + nRegion/3);
        }
        g.setColor(Color.white);
        g.drawString(HeapOfFishStrings.handlePool, poolImageInsets + xTextStart, poolImageInsets +
yStart + fm.getAscent());
        yHandlePoolRect = yStart + labelHeight;
```

```
        g.fillRect(xHandlePoolPortion + poolImageInsets, yHandlePoolRect, poolsWidth,
handlePoolHeight);

        xTextStart = poolsWidth - fm.stringWidth(HeapOfFishStrings.objectPool);
        if (xTextStart < 0) {
          xTextStart = 0;
        }
        xTextStart /= 2;

        yStart = ((dim.height/4) * (3 + nRegion/3) - objectPoolHeight - labelHeight - (2 *
poolImageInsets)) / 2;
        if (yStart < yCuadrante * (2 + nRegion/3)) {
          yStart = yCuadrante * (2 + nRegion/3);
        }

        //g.setColor(Color.white);
        g.drawString(HeapOfFishStrings.objectPool, xObjectPoolPortion + xTextStart, poolImageInsets
+ yStart + fm.getAscent());
        yObjectPoolRect = yStart + labelHeight;
        //g.setColor(Color.white);
        g.fillRect(xObjectPoolPortion, yObjectPoolRect, poolsWidth, objectPoolHeight);

        // Draw the headers in the object pool
        g.setColor(Color.black);
        i = 0;
        while (i < objectPoolIntsCount) {

          for (int j = 0; j < yPixelsPerInt; ++j) {
            int yLinePos = yObjectPoolRect + (i * yPixelsPerInt) + j;
            g.drawLine(xObjectPoolPortion, yLinePos, xObjectPoolPortion + poolsWidth - 1, yLinePos);
          }
          int header = actual.getObjectPool(i);
          int length = actual.getMemBlockLength(header);
          if (length <= 0) { // In case object pool gets corrupted, don't hang up.
            break;
          }
          i += length;
        }

        for (i = 0; i < actual.getHandlePoolSize(); ++i) {
          ObjectHandle oh = actual.dameManejador(i + 1);
          if (!oh.free) {

            Color color = Color.red;
            int objectSizeInInts = 3;
            if (oh.fish.getFishColor() == Color.cyan) {
              color = Color.cyan;
              objectSizeInInts = 2;
            }
            else if (oh.fish.getFishColor() == Color.yellow) {
              color = Color.yellow;
              objectSizeInInts = 1;
            }
            g.setColor(color);

            // Draw bar across handle pool
            for (int j = 0; j < yPixelsPerInt * 2; ++j) {
              int yLinePos = yHandlePoolRect + (i * yPixelsPerInt * 2) + j;
              g.drawLine(xHandlePoolPortion + poolImageInsets, yLinePos, xHandlePoolPortion +
poolImageInsets + poolsWidth - 1, yLinePos);
```

```
            }

            // Draw colored bars to represent object in the object pool
            for (int j = 0; j < yPixelsPerInt * objectSizeInInts; ++j) {
              int yLinePos = yObjectPoolRect + (oh.objectPos * yPixelsPerInt) + j;
              g.drawLine(xObjectPoolPortion, yLinePos, xObjectPoolPortion + poolsWidth - 1, yLinePos);
            }

            // Draw a line from the handle to the object to show that the handle
            // points to the object.
            int yArrowStart = yHandlePoolRect + (i * yPixelsPerInt * 2) + yPixelsPerInt;
            int yArrowEnd = yObjectPoolRect + (oh.objectPos * yPixelsPerInt) + ((yPixelsPerInt *
objectSizeInInts) / 2);
            g.drawLine(xHandlePoolPortion + poolImageInsets + poolsWidth + 2, yArrowStart,
                    xObjectPoolPortion - 3, yArrowEnd);
          }
        }
      }
    }
  }
}
```

l) Clase *FishIcon*:

En esta clase se ha creado el atributo *colorPezViejo* que contiene el color del que se va a colorear el pez cuando se pinte en el caso en que esté en la generación antigua.
Su declaración es la siguiente:

```
abstract class FishIcon {
….
  private Color colorPezViejo = null;
….
```

para generarlo tenemos un método privado llamado *generarColorPezViejo* que se invoca en caso que necesitemos acceder a él y tenga el valor *null*.

```
….
  private void generarColorPezViejo()
  {
    Color col = getFishColor();
    int r,g,b;
    r = 149 * col.getRed() / 255;
    g = 149 * col.getGreen() / 255;
    b = 149 * col.getBlue() / 255;
    colorPezViejo = new Color(r,g,b);
  }
….
```

Finalmente, necesitamos los métodos *dameColorPezViejo* para obtener el color y *pintarPezViejo* para pintarlo.

```
.....

public Color dameColorPezViejo()
   {
     if(colorPezViejo == null)
       generarColorPezViejo();
     return colorPezViejo;
   }

   public void paint(Graphics g) {
      paintWithSpecialColors(g, fishColor, Color.black);
   }

   public void pintarPezViejo(Graphics g)
   {
     if(colorPezViejo == null)
       generarColorPezViejo();
     paintWithSpecialColors(g, colorPezViejo, Color.black);
   }
....
```

m) clase *EspacioRegiones* :

Se ocupa de crear y destruir las regiones con el número especificado, por medio de los métodos *crearNuevaRegion* y *destruirRegion*, acceder a las regiones indicadas, ya sea por el número de región, con el método *dameRegion* o por la dirección del objeto contenido en la región, con el método *dameRegionQueContengaLaDireccion*. Además de las propias regiones, también contiene las listas de las referencias existentes de cada una de las regiones al heap. Es posible obtener las regiones que están ocupadas con el método *dameRegionesOcupadas*, sus correspondientes listas con el método *dameListaReferencias* o los numeros de dichas regiones, con el método *dameNumerosRegionesOcupadas*.

```
public class EspacioRegiones {

  private Region[] regiones;
  private ListaReferencias[] lista;
  private int regionesOcupadas;
  private int finHeap;

  public EspacioRegiones(int fh) {
    regiones = new Region[4];
    lista = new ListaReferencias[4];
    regionesOcupadas = 0;
    finHeap = fh;
    for(int i=0;i<4;i++)
    {
      regiones[i] = null;
      lista[i] = new ListaReferencias();
    }
  }

  public boolean crearNuevaRegion(int num, int handlePoolSize, int objectPoolSize)
```

```
  {
    int indUltRegion, dirIni;
    if(regiones[num - 1] != null)
      return false;
    else
    {
      indUltRegion = ultimaRegion();
      if(indUltRegion == -1)
        dirIni = finHeap;
      else
        dirIni = regiones[indUltRegion].dameDireccionInicio() +
regiones[indUltRegion].getObjectPoolSize();
      regiones[num - 1] = new Region(handlePoolSize,objectPoolSize,dirIni);
      regionesOcupadas++;
      return true;
    }
  }

  public boolean destruirRegion(int num)
  {
    if(regiones[num -1] == null)
      return false;
    else
    {
      regiones[num - 1] = null;
      lista[num - 1].limpiarLista();
      regionesOcupadas--;
      return true;
    }
  }

  public Region dameRegion(int num)
  {
    return regiones[num - 1];
  }

  public Region dameRegionQueContengaLaDireccion(int dir)
  {
    Region res = null;
    for(int i=0;res == null && i<4;i++)
      if(regiones[i] != null && dir >= regiones[i].dameDireccionInicio() &&
         dir <= regiones[i].dameDireccionInicio() + regiones[i].getObjectPoolSize())
           res = regiones[i];
    return res;
  }

  public ListaReferencias[] dameListaReferencias()
  {
    ListaReferencias res[] = new ListaReferencias[regionesOcupadas];
    int j = 0;
    for(int i=0;i<4;i++)
      if(regiones[i] != null)
        res[j++] = lista[i];
    return res;
  }

  public ListaReferencias dameListaReferencias(Region rg)
  {
    ListaReferencias res = null;
    for(int i=0;i<4;i++)
```

```
      if(regiones[i] == rg)
        res = lista[i];
    return res;
  }

  public Region[] dameRegionesOcupadas()
  {
    Region[] res = new Region[regionesOcupadas];
    int j = 0;
    for(int i=0;i<4;i++)
      if(regiones[i] != null)
        res[j++] = regiones[i];
    return res;
  }

public int[] dameNumerosRegionesOcupadas()
  {
    int[] res = new int[regionesOcupadas];
    int j = 0;
    for(int  i=0;i<4;i++)
      if(regiones[i] != null)
        res[j++] = i + 1;
    return res;
  }

  public int regionesOcupadas()
  {
    return regionesOcupadas;
  }

  private int ultimaRegion()
  {
    int res = -1;
    int dirMax = 0;
    for(int i=0;i<4;i++)
      if(regiones[i] != null)
      {
        if(regiones[i].dameDireccionInicio() > dirMax)
        {
          dirMax = regiones[i].dameDireccionInicio();
          res = i;
        }
      }
    return res;
  }

  public int getHandlePoolSize()
  {
    int res = 0;
    Region[] regsOcup = dameRegionesOcupadas();
    for(int i = 0;i<regionesOcupadas;i++)
      res += regsOcup[i].getHandlePoolSize();
    return res;
  }
}
```

n) Clase *Instruccion*:

Dado que las instrucción se componen de código de operación y operandos esto es lo que representa esta clase. Al haber instrucción de distinto número de operandos, que en este caso es 1 ó 2, pero no habría dificultad alguna en introducir instrucciones que contengan más de 2 operandos, la representación de los operandos se hace con un array, cuyo tamaño se especifica en construcción junto con el código de operación, ya que una vez conocida la instrucción, ya que se conoce el número de operandos. Una vez creada una instrucción, se pueden establecer cada uno de los operandos con el método *ponOperando*.

```
public class Instruccion {

  public static final int crr = 0;
  public static final int craz = 1;
  public static final int cram = 2;
  public static final int am = 3;
  public static final int cm = 4;
  public static final int sn = 5;
  public static final int nam = 6;
  public static final int ncm = 7;
  public static final int nsn = 8;
  public static final int asr = 9;
  public static final int nr = 10;
  public static final int ami = 11;
  public static final int cmi = 12;
  public static final int sni = 13;
  public static final int amdi = 14;
  public static final int cmdi = 15;
  public static final int sndi = 16;

  private int codigo;
  private int operandos[];

  public Instruccion(int codigoIns,int noperandos) {
    codigo = codigoIns;
    operandos = new int[noperandos];
  }

  public void ponOperando(int indice,int op)
  {
    operandos[indice] = op;
  }

  public int dameOperando(int indice)
  {
    return operandos[indice];
  }

  public int dameNOperandos()
  {
    return operandos.length;
  }

  public int dameCodigo()
  {
    return codigo;
```

```
  }
}
```

o) Clase *ListaReferencias*:

La lista de referencias utilizada es una lista de direcciones de memoria, donde se conoce el número de repeticiones de cada dirección. El método *insertar* incrementa en una unidad el número de repeticiones de la dirección especificada, por otra parte el método *borrar* elimina una repetición de la dirección especificada, pudiendo eliminar todas las repeticiones de esa dirección con el método *borrarTodos*. Para acceder a los elementos de esta lista, se itera sobre ellos, a través de los métodos *reset*, *siguiente* y *haySiguiente*. El método privado *busca*, se ocupa de devolver un puntero al nodo anterior donde se encuentra el elemento especificado, o *null* si el elemento está en primera posición o no existe.

```
public class ListaReferencias {

  private Nodo primero;
  private Nodo actual;

  public ListaReferencias() {
    limpiarLista();
  }

  public void insertar(int dato)
  {
    if(primero == null)
      primero = new Nodo(dato,null);
    else if(primero.dato == dato)
      primero.cantidad++;
    else
    {
      Nodo aux = busca(dato);
      if(aux != null)
        aux.siguiente.cantidad++;
      else
        primero = new Nodo(dato,primero);
    }

  }

  public boolean borrar(int dato)
  {
    Nodo aux = primero;
    if(primero == null)
      return false;
    else if(primero.dato == dato)
    {
      primero.cantidad--;
      if(primero.cantidad == 0)
      {
        primero = primero.siguiente;
```

```
      if(actual == null || actual.dato == dato)
        actual = primero;
    }
    return true;
  }
  else
  {
    aux = busca(dato);
    if(aux == null)
      return false;
    else
    {
      aux.siguiente.cantidad--;
      if(aux.siguiente.cantidad == 0)
      {
        aux.siguiente = aux.siguiente.siguiente;
        if(actual != null && actual.dato == dato)
          actual = actual.siguiente;
      }
      return true;
    }
  }
}

public boolean borrarTodos(int dato)
{
  Nodo aux = primero;
  if(primero == null)
    return false;
  else if(primero.dato == dato)
  {
      primero = primero.siguiente;
      if(actual != null && actual.dato == dato)
        actual = primero;
      return true;
  }
  else
  {
    aux = busca(dato);
    if(aux == null)
      return false;
    else
    {
        aux.siguiente = aux.siguiente.siguiente;
        if(actual != null && actual.dato == dato)
          actual = actual.siguiente;
        return true;
    }
  }
}

public boolean pertenece(int dato)
{
  return primero != null && primero.dato == dato || busca(dato) != null;
}

public int siguiente() throws NoSuchElementException
{
  if(actual == null)
    throw new NoSuchElementException();
```

```
    int res = actual.dato;
    actual = actual.siguiente;
    return res;
  }

  public boolean haySiguiente()
  {
    return actual != null;
  }

  public void reset()
  {
    actual = primero;
  }

  public void limpiarLista()
  {
    primero = null;
    actual = null;
  }

  private Nodo busca(int dato)
  {
    if(primero == null)
      return null;
    else
    {
      boolean encontrado = primero.siguiente != null && primero.siguiente.dato == dato;
      Nodo aux;
      for(aux = primero;!encontrado &&  aux.siguiente != null;encontrado = aux.siguiente != null &&
aux.siguiente.dato == dato)
        aux = aux.siguiente;
      if(encontrado)
        return aux;
      else
        return null;
    }
  }

  private class Nodo
  {
    public int dato;
    public int cantidad;
    public Nodo siguiente;

    public Nodo(int dt,Nodo sg)
    {
      dato = dt;
      siguiente = sg;
      cantidad = 1;
    }
  }
}
```

p) Clase MemoriaDeInstrucciones.

La clase *MemoriaDeInstrucciones* además de almacenar las instrucciones que esperan ser ejecutadas almacena la dirección de memoria a la que se refieren las instrucciones *am*, *cm*, *sn*, *asr*, *nam*, *ncm* y *nsn*, generada al crear un objeto mediante *crr*, *craz* o *cram* e identificada en el operando *pX*. Por tanto, contiene un atributo *referencias* que contiene en cada posición del array la dirección a la que hace referencia.

```
public class MemoriaDeInstrucciones extends LinkedList{
  private int[] referencias;

  public MemoriaDeInstrucciones(int tam) {
    super();
    referencias = new int[tam];
  }

  public void ponReferencia(int indice,int valor)
  {
    referencias[indice] = valor;
  }

  public int dameReferencia(int indice)
  {
    return referencias[indice];
  }

}
```

q) Clases *MotorDeEjecucion* e *HiloDeEjecucion*:

La clase *MotorDeEjecucion* contiene un puntero a los objetos de las clases *AllocateFishPanel* y *LinkFishCanvas* , porque en esos paneles es donde realmente se va a hacer la ejecución de nuestro código. Se han implementado mutadores para fijar estos atributos porque la creación de estos paneles es posterior a la creación de este objeto. Esta clase, al ser hija de la clase *Thread*, es un hilo, cuyo método *run* se ocupa de leer las instrucciones de la memoria de instrucciones y lanzarlas a ejecución en el panel correspondiente. En el caso de las instrucciones correspondientes a crear peces, lo hace en el objeto de la clase *AllocateFishPanel*, pero dado que tiene efecto también en la clase *LinkFishCanvas*, invoca su método *repaint* cada vez que ejecuta esta instrucción. Esta clase se ocupa de ejecutar en el heap y su ejecución es infinita, pero es necesario implementar los métodos *continuar* y *finalizar* porque van a ser sobreescritos en el hijo. Este propio objeto también se ocupa de crear objetos de la clase hija cada vez que haya que se lanzar una ejecución en una región. Esto se hace con el método *lanzarAEjecucion*.

```
public class MotorDeEjecucion extends Thread{
  private LinkFishCanvas lfc;
  private AllocateFishPanel afp;
  private Tarea tarea;
  protected MemoriaDeInstrucciones mi;
  private EspacioAlmacenamiento espacio;
  private static final Tarea tc = new TareaCritica();
  private static final Tarea tn = new TareaNormal();
```

```
  protected ArrayList hilosEnEjecucion;
  protected PecesIniciales pi;

  public MotorDeEjecucion(MemoriaDeInstrucciones memIns, GCHeap heap) {
    mi = memIns;
    espacio = heap;
    pi = new PecesIniciales(heap);
    tarea = tn;
    hilosEnEjecucion = new ArrayList();
  }

  protected MotorDeEjecucion(LinkFishCanvas l, AllocateFishPanel a, Tarea t, MemoriaDeInstrucciones
memoria, EspacioAlmacenamiento e)
  {
    lfc = l;
    afp = a;
    tarea = t;
    mi = memoria;
    espacio = e;
  }

  public void ponLinkFishCanvas(LinkFishCanvas canvas)
  {
    lfc = canvas;
  }

  public void ponAllocateFishPanel(AllocateFishPanel canvas)
  {
    afp = canvas;
  }

  public void lanzarAEjecucion(MemoriaDeInstrucciones memoria,EspacioAlmacenamiento esp, boolean
critica, PanelCargarCodigo pcc, int region)
  {
    Tarea t;
    int prioridad;
    if(critica)
    {
      t = tc;
      prioridad = 9;
    }
    else
    {
      t = tn;
      prioridad = Thread.NORM_PRIORITY;
    }
    Thread hilo = new HiloDeEjecucion(this,lfc,afp,t,memoria,esp,pcc,region);
    hilo.setPriority(prioridad);
    hilosEnEjecucion.add(hilo);
    hilo.start();
  }

  protected boolean continuar()
  {
    return true;
  }

  protected void finalizar()
  {
```

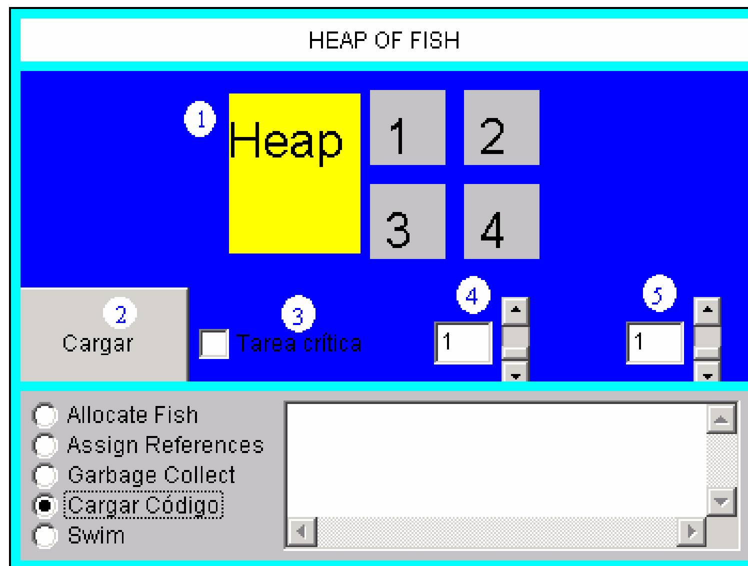
```
}


public void run()
{
  Instruccion ins;
  while(continuar())
  {
    if(!mi.isEmpty())
    {
      ins = (Instruccion)mi.removeFirst();
      switch(ins.dameCodigo())
      {
        case Instruccion.crr:afp.crearPezRojo(ins,pi,mi);
                         lfc.repaint();
                         break;
        case Instruccion.craz:afp.crearPezAzul(ins,pi,mi);
                          lfc.repaint();
                          break;
        case Instruccion.cram:afp.crearPezAmarillo(ins,pi,mi);
                          lfc.repaint();
                          break;
        case Instruccion.am:lfc.asignarReferenciasEntrePecesIndirecto(ins,tarea,0,mi);
                        break;
        case Instruccion.cm:lfc.asignarReferenciasEntrePecesIndirecto(ins,tarea,1,mi);
                        break;
        case Instruccion.sn:lfc.asignarReferenciasEntrePecesIndirecto(ins,tarea,2,mi);
                        break;
        case Instruccion.nam:lfc.anularReferenciasPeces(ins,tarea,0,mi);
                         break;
        case Instruccion.ncm:lfc.anularReferenciasPeces(ins,tarea,1,mi);
                         break;
        case Instruccion.nsn:lfc.anularReferenciasPeces(ins,tarea,2,mi);
                         break;
        case Instruccion.asr:lfc.asignarPecesAReferencias(ins,mi);
                         break;
        case Instruccion.nr:lfc.anularReferencias(ins);
                        break;
        case Instruccion.ami:lfc.asignarReferenciasEntrePecesInmediato(ins,tarea,0);
                         break;
        case Instruccion.cmi:lfc.asignarReferenciasEntrePecesInmediato(ins,tarea,1);
                         break;
        case Instruccion.sni:lfc.asignarReferenciasEntrePecesInmediato(ins,tarea,2);
                         break;
        case Instruccion.amdi:lfc.asignarReferenciasEntrePecesDestinoInmediato(ins,tarea,0,mi);
                          break;
        case Instruccion.cmdi:lfc.asignarReferenciasEntrePecesDestinoInmediato(ins,tarea,1,mi);
                          break;
        case Instruccion.sndi:lfc.asignarReferenciasEntrePecesDestinoInmediato(ins,tarea,2,mi);
                          break;
      }
    }
    try {
        sleep(3000);
      }  catch (InterruptedException e) {}
  }
  finalizar();
 }
}
```

Esta clase tiene una clase hija conocida como *HiloDeEjecucion* que se ocupa de ejecutar las instrucciones lanzadas a ejecución en una región. Tiene referencias a los objetos de las clases *MotorDeEjecucion* y *PanelCargarCodigo* para notificar cuando la ejecución llega a su final. Su implementación es así:

```
public class HiloDeEjecucion extends MotorDeEjecucion{

  private MotorDeEjecucion motor;
  private boolean arrancado;
  private int region;
  private PanelCargarCodigo pcc;

  public HiloDeEjecucion(MotorDeEjecucion me,LinkFishCanvas l, AllocateFishPanel a, Tarea t,
MemoriaDeInstrucciones memoria, EspacioAlmacenamiento e, PanelCargarCodigo p, int reg) {
    super(l,a,t,memoria,e);
    pi = new PecesIniciales(e);
    motor = me;
    arrancado = false;
    region = reg;
    pcc = p;
  }

  protected boolean continuar()
  {
    if(mi.isEmpty())
    {
      return !arrancado;
    }
    else
    {
      arrancado = true;
      return true;
    }
  }

  protected void finalizar()
  {
    pcc.finDeEjecucion(region);
    motor.hilosEnEjecucion.remove(this);
  }
}
```

r) Clase *PanelCargarCodigo*:

Como habíamos comentado anteriormente, el aspecto de este panel es el siguiente:



La zona del área *1* que muestra el heap está representada por el atributo *panelHeap* y cada una de las regiones está representada por cada uno de los elementos del array *panelesRegiones*. El resto del panel está contenido en otro panel de la clase *PanelBotonesCargarArchivo*, pero algunos objetos están también referenciados desde este panel como el *3*, representado por *critica* y el *4* y *5*, estando representados los campos de texto por *numManejadores* y *tamEspObjetos* y las flechas para modificarlos por *modificarManejadores* y *modificarObjetos*, respectivamente.

El método action se ocupa de realizar la acción correspondiente cuando se pulsa el botón *Cargar*, que en principio es saber si la ejecución va a tener lugar en el heap o en alguna región y en este último caso en que region va a ser. Esto se hace leyendo la variable *regionSeleccionada*, que es *null* cuando es en el heap y apunta a la región en que se va a ejecutar si es en una región. En el caso en que se trate de una región, también hay que leer los tamaños de manejadores y objetos y si la tarea es crítica. Luego se ejecuta un diálogo para pedir el archivo que se quiere cargar y si se pulsa *Aceptar* (los métodos *getDirectory* y *getFile* no devuelven *null*) se lanza a ejecución un hilo de la clase *HiloCarga* que se ocupa de leer el archivo y cargarlo en la memoria de instrucciones. Si la ejecución es en el heap, esta memoria es la creada en el objeto de la clase *HeapOfFish*, pero si es en una región, hay que crear una memoria nueva. En el caso de las regiones, además hay que crear la región invocando al método *crearRegion* de la clase *EspacioRegiones*, invocar al método *lanzarAEjecucion* de la clase *MotorDeEjecucion* para crear un hilo de la clase *HiloDeEjecucion* que se encargue de ejecutar el código y marcar su panel en rojo, invocando al método ocupar de la clase *PanelRegion*.

Los métodos *modificandoManejadores* y *modificandoObjetos* son invocados desde un objeto de una clase anónima que implementa la interface *AdjustmentListener* , que hace de oyente de las flechas, y modifican el tamaño de las zonas de manejadores y objetos (*4* y *5*) cuando se pulsan las flechas.

Los objetos de las clases internas *PanelHeap* y *PanelRegiones* implementan su propio oyente, que consiste en marcar el panel como seleccionado, es decir de color magenta y modificar el contenido de la variable *regionSeleccionada*.

```
public class PanelCargarCodigo extends Panel{

   private GCHeap gcHeap;
   private EspacioRegiones espRegiones;
   private LocalVariables localVars;


   private MemoriaDeInstrucciones mi;
   private MotorDeEjecucion motor;

   private PanelRegion[] panelesRegiones;
   private Panel contenedorRegiones;
   private Panel panelCentral;
   private Panel panelHeap;
   private PanelRegion regionSeleccionada;

   private TextField numManejadores;
   private TextField tamEspObjetos;

   private Scrollbar modificarManejadores;
   private Scrollbar modificarObjetos;

   private Checkbox critica;

   private ArrayList hilosEnEjecucion;

   private int flag;

   PanelCargarCodigo(GCHeap heap,MemoriaDeInstrucciones memIns, LocalVariables locVars,
HeapOfFishTextArea ta, MotorDeEjecucion me, EspacioRegiones er) {

      setBackground(Color.blue);

      critica = new Checkbox("Tarea crítica",false);
      panelCentral = new Panel();
      panelHeap = new PanelHeap();
      contenedorRegiones = new Panel();
      contenedorRegiones.setLayout(new GridLayout(2,2,10,10));
      panelesRegiones = new PanelRegion[4];
      gcHeap = heap;
      espRegiones = er;



      numManejadores = new TextField("1");
      tamEspObjetos = new TextField("1");
      modificarManejadores = new Scrollbar(Scrollbar.VERTICAL,100,0,0,100);
      modificarObjetos = new Scrollbar(Scrollbar.VERTICAL,100,0,0,100);
      Panel pm = new Panel();
      Panel po = new Panel();

      for(int i=0;i<4;i++)
      {
        panelesRegiones[i] = new PanelRegion(i+1, espRegiones.dameRegion(i+1) != null);
        contenedorRegiones.add(panelesRegiones[i]);
      }
```

```
        mi = memIns;
        motor = me;
        localVars = locVars;
        regionSeleccionada = null;

        pm.add(numManejadores);
        pm.add(modificarManejadores);
        po.add(tamEspObjetos);
        po.add(modificarObjetos);
        modificarManejadores.addAdjustmentListener(new AdjustmentListener(){public void
adjustmentValueChanged(AdjustmentEvent e){modificandoManejadores(e);}});
        modificarObjetos.addAdjustmentListener(new AdjustmentListener(){public void
adjustmentValueChanged(AdjustmentEvent e){modificandoObjetos(e);}});
        panelCentral.add("West",panelHeap);
        panelCentral.add("East",contenedorRegiones);
        //add("North", ccCanvas);
        add("Center",panelCentral);
        add("South", new PanelBotonesCargarArchivo(critica,pm,po));
        hilosEnEjecucion = new ArrayList();
    }


    public boolean action(Event evt, Object arg) {
      if (evt.target instanceof Button) {
           String bname = (String) arg;
           if (bname.equals(HeapOfFishStrings.cargar)) {
             if(regionSeleccionada == null)
             {
               FileDialog fd = new FileDialog(Frame.getFrames()[0]);
               fd.setMode(FileDialog.LOAD);
               fd.show();
               String directorio = fd.getDirectory();
               String archivo = fd.getFile();
               try{

                 if(directorio != null && archivo != null)
                 {
                   FileReader fr = new FileReader(directorio + archivo);
                   Thread hilo = new HiloCarga(mi,gcHeap,fr);
                   hilosEnEjecucion.add(hilo);
                   hilo.start();
                 }
               }catch(FileNotFoundException fnfe)
               {
                 JOptionPane.showMessageDialog(null,"El archivo no
existe","Error",JOptionPane.ERROR_MESSAGE);
               }
             }
             else
             {
               try{
                 int hndPoolSize = Integer.parseInt(numManejadores.getText());
                 int objPoolSize = Integer.parseInt(tamEspObjetos.getText());
                 int numRegion = regionSeleccionada.numeroRegion();
                 String directorio, archivo;
                 FileDialog fd = new FileDialog(Frame.getFrames()[0]);
                 fd.setMode(FileDialog.LOAD);
                 fd.show();
```

```
              directorio = fd.getDirectory();
              archivo = fd.getFile();
              if(directorio != null && archivo != null)
              {
                espRegiones.crearNuevaRegion(numRegion,hndPoolSize,objPoolSize);
                Region rg = espRegiones.dameRegion(numRegion);
                MemoriaDeInstrucciones m = new MemoriaDeInstrucciones(40);
                FileReader fr = new FileReader(directorio + archivo);
                Thread hilo = new HiloCarga(m,rg,fr);
                hilo.start();
                regionSeleccionada.ocupar();
                regionSeleccionada = null;
                motor.lanzarAEjecucion(m,rg,critica.getState(),this,numRegion);
                hilosEnEjecucion.add(hilo);
              }
            }catch(NumberFormatException e)
            {
              JOptionPane.showMessageDialog(this,"Formato de tamaño de región
incorrecto","Error",JOptionPane.ERROR_MESSAGE);
            }
            catch(FileNotFoundException fnfe)
            {
              JOptionPane.showMessageDialog(null,"El archivo no
existe","Error",JOptionPane.ERROR_MESSAGE);
            }
          }
        }
    }
    return true;
  }

  private void modificandoManejadores(AdjustmentEvent e)
  {
    int numero;
    try{
      numero = Integer.parseInt(numManejadores.getText());
      switch(e.getAdjustmentType())
      {
        case AdjustmentEvent.UNIT_DECREMENT: numero++;
          break;
        case AdjustmentEvent.UNIT_INCREMENT:if(numero >1)numero--;
          break;
      }
    }catch(NumberFormatException nfe)
    {
      numero = 1;
    }
    numManejadores.setText(Integer.toString(numero));
  }

  private void modificandoObjetos(AdjustmentEvent e)
  {
    int numero;
    try{
      numero = Integer.parseInt(tamEspObjetos.getText());
      switch(e.getAdjustmentType())
      {
        case AdjustmentEvent.UNIT_DECREMENT: numero++;
          break;
        case AdjustmentEvent.UNIT_INCREMENT:if(numero >1)numero--;
```

```
        break;
      }
    }catch(NumberFormatException nfe)
    {
      numero = 1;
    }
    tamEspObjetos.setText(Integer.toString(numero));
  }

  public void finDeEjecucion(int region)
  {
    panelesRegiones[region - 1].setBackground(Color.green);
  }

  private void cargarArchivo(FileReader archivo, MemoriaDeInstrucciones mem) throws
FileNotFoundException,IOException
  {
    BufferedReader br = new BufferedReader(archivo);
    String linea;
    String cod;
    String ins;
    String[] etiquetaOperando;
    int espacio;
    int codIns = -1;
    int nOpIns = -1;
    Instruccion insNueva;
    etiquetaOperando = new String[3];
    while(br.ready())
    {
      linea = br.readLine();
      espacio = linea.indexOf(" ");
      cod = linea.substring(0,espacio);
      ins = linea.substring(espacio);
      switch(cod.charAt(0))
      {
        case 'c': if(cod.charAt(1) == 'r')
                  {
                    if(cod.charAt(2) == 'r')
                      codIns = Instruccion.crr;
                    else switch(cod.charAt(3))
                    {
                      case 'm': codIns = Instruccion.cram;
                                break;
                      case 'z': codIns = Instruccion.craz;
                                break;
                    }
                    nOpIns = 1;
                    etiquetaOperando[0] = "p";
                  }
                  else if(cod.charAt(1) == 'm')
                  {
                    if(cod.length() > 2 && cod.charAt(2) == 'i')
                    {
                      codIns = Instruccion.cmi;
                      etiquetaOperando[0] = "#";
                      etiquetaOperando[1] = "#";
                    }
                    else if(cod.length() > 3 && cod.charAt(2) == 'd' && cod.charAt(3) == 'i')
                    {
                      codIns = Instruccion.cmdi;
```

```
            etiquetaOperando[0] = "p";
            etiquetaOperando[1] = "#";
          }
          else
          {
            codIns = Instruccion.cm;
            etiquetaOperando[0] = "p";
            etiquetaOperando[1] = "p";
          }

          nOpIns = 2;
        }
        break;
case 'a': if(cod.charAt(1) == 'm')
        {
          if(cod.length() > 2 && cod.charAt(2) == 'i')
          {
            codIns = Instruccion.ami;
            etiquetaOperando[0] = "#";
            etiquetaOperando[1] = "#";
          }
          else if(cod.length() > 3 && cod.charAt(2) == 'd' && cod.charAt(3) == 'i')
          {
            codIns = Instruccion.amdi;
            etiquetaOperando[0] = "p";
            etiquetaOperando[1] = "#";
          }
          else
          {
            codIns = Instruccion.am;
            etiquetaOperando[0] = "p";
            etiquetaOperando[1] = "p";
          }
          nOpIns = 2;
        }
        else if(cod.charAt(1) == 's' && cod.charAt(2) == 'r')
        {
          codIns = Instruccion.asr;
          etiquetaOperando[0] = "p";
          etiquetaOperando[1] = "r";
          nOpIns = 2;
        }
        break;
case 's': if(cod.charAt(1) == 'n')
        {
          if(cod.length() > 2 && cod.charAt(2) == 'i')
          {
            codIns = Instruccion.sni;
            etiquetaOperando[0] = "#";
            etiquetaOperando[1] = "#";
          }
          else if(cod.length() > 3 && cod.charAt(2) == 'd' && cod.charAt(3) == 'i')
          {
            codIns = Instruccion.sndi;
            etiquetaOperando[0] = "p";
            etiquetaOperando[1] = "#";
          }
          else
          {
            codIns = Instruccion.sn;
```

```
                etiquetaOperando[0] = "p";
                etiquetaOperando[1] = "p";
               }

               nOpIns = 2;
              }
              break;
      case 'n': switch(cod.charAt(1))
              {
               case 'a': codIns = Instruccion.nam;
                    etiquetaOperando[0] = "p";
                    break;
               case 'c': codIns = Instruccion.ncm;
                    etiquetaOperando[0] = "p";
                    break;
               case 's': codIns = Instruccion.nsn;
                    etiquetaOperando[0] = "p";
                    break;
               case 'r': codIns = Instruccion.nr;
                    etiquetaOperando[0] = "r";
                    break;
              }
              nOpIns = 1;

              break;
     }
     insNueva = new Instruccion(codIns,nOpIns);
     int indOp = 0;
     int indComa;
     for(int i=0;i<nOpIns;i++)
     {
      indOp = ins.indexOf(etiquetaOperando[i],indOp);
      indOp++;
      indComa = ins.indexOf(",",indOp);
      if(indComa == -1)
        indComa = ins.length();

      insNueva.ponOperando(i,Integer.parseInt(ins.substring(indOp,indComa)));
     }
     mem.add(insNueva);

   }
  }

  private class HiloCarga extends Thread
  {
   private MemoriaDeInstrucciones memi;
   private EspacioAlmacenamiento esp;
   private FileReader archivo;

   public HiloCarga(MemoriaDeInstrucciones memoria, EspacioAlmacenamiento espacio, FileReader
fr)
   {
     memi = memoria;
     esp = espacio;
     archivo = fr;
   }

   public void run()
   {
```

```
        try{

           cargarArchivo(archivo,memi);

        }catch(FileNotFoundException fne)
        {
          JOptionPane.showMessageDialog(null,"El archivo no
existe","Error",JOptionPane.ERROR_MESSAGE);
        }catch(IOException ioe)
        {
          JOptionPane.showMessageDialog(null,"IOException");
        }
        hilosEnEjecucion.remove(this);

      }


    }

    private class PanelRegion extends Panel implements MouseListener
    {
      private int numRegion;
      private boolean ocupada;
      private Label etiqueta;

      public PanelRegion(int nr, boolean ocup)
      {
        super();
        numRegion = nr;
        ocupada = ocup;
        etiqueta = new Label();
        etiqueta.setFont(new Font("Arial",Font.PLAIN,26));
        etiqueta.setText(Integer.toString(numRegion));
        add("Center",etiqueta);
        addMouseListener(this);
        etiqueta.addMouseListener(this);
        if(ocupada)
          setBackground(Color.red);
        else
          setBackground(Color.lightGray);
      }

      public int numeroRegion()
      {
        return numRegion;
      }

      public void ocupar()
      {
        ocupada = true;
        setBackground(Color.red);
        etiqueta.setBackground(Color.red);
      }

      public void desocupar()
      {
        ocupada = false;
        setBackground(Color.lightGray);
        etiqueta.setBackground(Color.lightGray);
```

```
}

public Dimension minimumSize() {
  return new Dimension(40, 40);
}

public Dimension preferredSize() {
  return new Dimension(40, 40);
}

public void mouseClicked(MouseEvent e)
{

  if(!ocupada)
  {
    setBackground(Color.magenta);
    if(regionSeleccionada != null)
      regionSeleccionada.setBackground(Color.lightGray);
    else
      panelHeap.setBackground(Color.yellow);
    regionSeleccionada = this;
    critica.setEnabled(true);
    numManejadores.setEnabled(true);
    tamEspObjetos.setEnabled(true);
    modificarManejadores.setEnabled(true);
    modificarObjetos.setEnabled(true);
  }
  else if(getBackground().equals(Color.green))
  {
    int iniRegion, finRegion;
    iniRegion = espRegiones.dameRegion(numRegion).dameDireccionInicio();
    finRegion = iniRegion + espRegiones.dameRegion(numRegion).getObjectPoolSize();
    if(localVars.yellowFish > iniRegion && localVars.yellowFish <= finRegion)
      localVars.yellowFish = 0;
    if(localVars.blueFish > iniRegion && localVars.blueFish <= finRegion)
      localVars.blueFish = 0;
    if(localVars.redFish > iniRegion && localVars.redFish <= finRegion)
      localVars.redFish = 0;
    desocupar();
    espRegiones.destruirRegion(numRegion);
  }
}

public void mousePressed(MouseEvent e)
{

}

public void mouseReleased(MouseEvent e)
{

}

public void mouseEntered(MouseEvent e)
{

}

public void mouseExited(MouseEvent e)
{
```

```
  }
}

private class PanelHeap extends Panel implements MouseListener
{
  private Label etiqueta;

  public PanelHeap()
  {
    super();
    etiqueta = new Label();
    etiqueta.setFont(new Font("Arial",Font.PLAIN,26));
    etiqueta.setText("Heap");
    add("Center",etiqueta);
    addMouseListener(this);
    etiqueta.addMouseListener(this);
    setBackground(Color.yellow);
  }


  public Dimension minimumSize() {
    return new Dimension(70, 85);
  }

  public Dimension preferredSize() {
    return new Dimension(70, 85);
  }

  public void mouseClicked(MouseEvent e)
  {
      setBackground(Color.magenta);
      if(regionSeleccionada != null)
        regionSeleccionada.setBackground(Color.lightGray);
      regionSeleccionada = null;
      critica.setEnabled(false);
      numManejadores.setEnabled(false);
      tamEspObjetos.setEnabled(false);
      modificarManejadores.setEnabled(false);
      modificarObjetos.setEnabled(false);
  }

  public void mousePressed(MouseEvent e)
  {

  }

  public void mouseReleased(MouseEvent e)
  {

  }

  public void mouseEntered(MouseEvent e)
  {

  }

  public void mouseExited(MouseEvent e)
  {
```

```
      }
    }
}
```

s) Clase *PanelBotonesCargarArchivo*.

Este panel está situado en el panel *PanelCargarCodigo* y contiene el botón, el cuadro de confirmación y los tamaños de zona de manejadores y zona de objetos de las regiones.

```
public class PanelBotonesCargarArchivo extends Panel{

  public PanelBotonesCargarArchivo(Checkbox caja, Panel listaManejadores, Panel listaObjetos) {
    Panel p = new Panel();
    p.setLayout(new GridLayout(1,4,5,5));
    p.add(new Button(HeapOfFishStrings.cargar));
    p.add(caja);
    p.add(listaManejadores);
    p.add(listaObjetos);
    add(p);
  }

}
```

t) Clase *PecesIniciales*.

La clase *PecesIniciales* contiene métodos para crear peces de las tres clases y devuelven la dirección del manejador que se crea asociado al objeto. Contiene una referencia al espacio de almacenamiento donde se crea el pez (heap o región).

```
public class PecesIniciales {

  private EspacioAlmacenamiento gcHeap;

  public PecesIniciales(EspacioAlmacenamiento gc) {
    gcHeap = gc;
  }

  public int nuevoPezRojo(){
    FishIcon fish = new BigRedFishIcon(false);
     int newFish = gcHeap.allocateObject(12, fish);
     if (newFish > 0) {
         ObjectHandle oh = gcHeap.getObjectHandle(newFish);
         gcHeap.setObjectPool(oh.objectPos, 0);
         gcHeap.setObjectPool(oh.objectPos + 1, 0);
         gcHeap.setObjectPool(oh.objectPos + 2, 0);
         oh.myColor=Color.white;

     }
```

```
    return newFish;
  }

  public int nuevoPezAzul(){
    FishIcon fish = new MediumBlueFishIcon(false);
    int newFish = gcHeap.allocateObject(8, fish);
    if (newFish > 0) {
        ObjectHandle oh = gcHeap.getObjectHandle(newFish);
        gcHeap.setObjectPool(oh.objectPos, 0);
        gcHeap.setObjectPool(oh.objectPos + 1, 0);
        oh.myColor=Color.white;

    }
    return newFish;
  }

  public int nuevoPezAmarillo(){
    FishIcon fish = new LittleYellowFishIcon(false);
    int newFish = gcHeap.allocateObject(4, fish);
    if (newFish > 0) {
        ObjectHandle oh = gcHeap.getObjectHandle(newFish);
        gcHeap.setObjectPool(oh.objectPos, 0);
        oh.myColor=Color.white;

    }
    return newFish;
  }
}
```

u) Interface *Recoleccion* y clases *RecoleccionAbstracta*, *MajorCollection* y *MinorCollection*

La interface *Recoleccion* representa cada uno de los tipos de recoleccion y, por lo tanto contiene todos los métodos que son invocados por el garbage collector.

```
public interface Recoleccion {

  int getCurrentGCState();
  int getGarbageCollectorHasNotStarted();
  int getGarbageCollectorIsDone();
  void resetGCState();
  void nextGCStep();
  boolean getYellowFishLocalVarIsCurrentGCMarkNode();
  boolean getBlueFishLocalVarIsCurrentGCMarkNode();
  boolean getRedFishLocalVarIsCurrentGCMarkNode();
  boolean getFishAreBeingMarked();
  int getCurrentFishBeingMarked();

}
```

Esta interface está implementada por la clase *RecoleccionAbstracta*.

```
public abstract class RecoleccionAbstracta implements Recoleccion{

    protected GarbageCollectCanvas canvas;

    protected GCHeap gcHeap;
    protected EspacioRegiones espRegiones;
    protected LocalVariables localVars;
    protected HeapOfFishTextArea controlPanelTextArea;

    protected int currentGCState = 0;

    protected boolean fishAreBeingMarked;
    protected int currentFishBeingMarked;

    protected boolean yellowFishLocalVarIsCurrentGCMarkNode;
    protected boolean blueFishLocalVarIsCurrentGCMarkNode;
    protected boolean redFishLocalVarIsCurrentGCMarkNode;

    protected ListaReferencias[] punterosRegionesHeap;
    protected ListaReferencias visitados;

    protected int punteroRegionHeapActual;
    protected int indiceRegion;


    public RecoleccionAbstracta(GarbageCollectCanvas gcc, GCHeap heap, EspacioRegiones er,
LocalVariables lv, HeapOfFishTextArea ta) {
      canvas = gcc;
      gcHeap = heap;
      espRegiones = er;
      localVars = lv;
      controlPanelTextArea = ta;
      visitados = new ListaReferencias();
    }

    public int getCurrentGCState()
    {
      return currentGCState;
    }

    public boolean getFishAreBeingMarked(){
      return fishAreBeingMarked;
    }

    public int getCurrentFishBeingMarked(){
      return currentFishBeingMarked;
    }

    public boolean getYellowFishLocalVarIsCurrentGCMarkNode()
    {
      return yellowFishLocalVarIsCurrentGCMarkNode;
    }

    public boolean getBlueFishLocalVarIsCurrentGCMarkNode()
    {
      return blueFishLocalVarIsCurrentGCMarkNode;
    }
```

```
  public boolean getRedFishLocalVarIsCurrentGCMarkNode()
  {
    return redFishLocalVarIsCurrentGCMarkNode;
  }

  public void resetGCState() {

    for (int i = 0; i < gcHeap.getHandlePoolSize(); ++i) {
        ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
        if (!oh.free) {
            oh.myColor = Color.white;
            oh.previousNodeInGCTraversalIsAFish = false;
            oh.previousFishInGCTraversal = 0;
            oh.myFriendLineColor = Color.white;
            oh.myLunchLineColor = Color.white;
            oh.mySnackLineColor = Color.white;
        }
    }
    currentGCState = getGarbageCollectorHasNotStarted();
    fishAreBeingMarked = false;
    yellowFishLocalVarIsCurrentGCMarkNode = false;
    blueFishLocalVarIsCurrentGCMarkNode = false;
    redFishLocalVarIsCurrentGCMarkNode = false;
    canvas.setYellowFishLocalVarLineColor(Color.white);
    canvas.setBlueFishLocalVarLineColor(Color.white);
    canvas.setRedFishLocalVarLineColor(Color.white);
    //controlPanelTextArea.setText(HeapOfFishStrings.garbageCollectInstructions);


  }


}
```

Esta clase es padre de las clases *MajorCollection* y *MinorCollection*

La clase *MajorCollection* contiene el método *nextGCHeap* que es el método que había originalmente en la clase *GarbageCollectCanvas*, añadiendo que al empezar a chequear una variable, además de asegurarnos de que es no nula, debemos asegurarnos que el objeto al que apunta está en el heap y no en una región. Además se eso se han añadido estados para chequear los punteros desde las regiones al heap, contenidos en cada una de las listas del array de la clase padre (*RecoleccionAbstracta*) etiquetado como *punterosRegionesHeap*. Por último se ha añadido el estado *recorriendoDescendenciaDesdeGrises*, que se ocupa de explorar y colorear la descendencia de los objetos que hayan quedado coloreados de gris al final de la exploración. Los métodos *transverseNextFishNode* y *traverseBackFromGrayLine* se han mantenido igual que los originales en la clase *GarbageCollectCanvas*.

```
public class MajorCollection extends RecoleccionAbstracta{
```

```
 // State variables for the garbage collector
 private final int garbageCollectorHasNotStarted = 0;

 private final int startingAtYellowLocalVariableRoot = 1;
 private final int traversingFromYellowLocalVariableRoot = 2;
 private final int doneWithYellowLocalVariableRoot = 3;

 private final int startingAtBlueLocalVariableRoot = 4;
 private final int traversingFromBlueLocalVariableRoot = 5;
 private final int doneWithBlueLocalVariableRoot = 6;

 private final int startingAtRedLocalVariableRoot = 7;
 private final int traversingFromRedLocalVariableRoot = 8;
 private final int doneWithRedLocalVariableRoot = 9;

 private final int readyToSweepUnmarkedFish = 10;
 private final int doneSweepingUnmarkedFish = 11;




 private final int empezandoConLosObjetosApuntadosDesdeRegion = 14;
 private final int recorriendoDesdeObjetosApuntadosDesdeRegion = 15;
 private final int siguienteObjetoApuntadoDesdeRegion = 16;
 private final int finObjetosApuntadosDesdeRegion = 17;

 private final int recorriendoDescendenciaDeObjetosGrises = 18;

 private final int garbageCollectorIsDone = 19;


 public MajorCollection(GarbageCollectCanvas gcc, GCHeap heap, EspacioRegiones espRegiones,
LocalVariables lv, HeapOfFishTextArea ta) {
   super(gcc,heap,espRegiones,lv,ta);
 }

 public int getGarbageCollectorHasNotStarted()
 {
   return garbageCollectorHasNotStarted;
 }

 public int getGarbageCollectorIsDone()
 {
   return garbageCollectorIsDone;
 }


public synchronized void nextGCStep() {
  //repaint();
  switch (currentGCState) {

  case garbageCollectorHasNotStarted:
      yellowFishLocalVarIsCurrentGCMarkNode = true;
      currentGCState = startingAtYellowLocalVariableRoot;
      controlPanelTextArea.setText(HeapOfFishStrings.traversingYellowRoot);
      break;

  case startingAtYellowLocalVariableRoot:
```

```
        yellowFishLocalVarIsCurrentGCMarkNode = false;
        if (localVars.yellowFish != 0 && localVars.yellowFish < gcHeap.getObjectPoolSize()) {

            ObjectHandle oh = gcHeap.getObjectHandle(localVars.yellowFish);
            yellowFishLocalVarIsCurrentGCMarkNode = false;
            oh.myColor = Color.gray;
            canvas.setYellowFishLocalVarLineColor(Color.gray);
            currentFishBeingMarked = localVars.yellowFish;
            fishAreBeingMarked = true;
            currentGCState = traversingFromYellowLocalVariableRoot;
        }
        else {
            blueFishLocalVarIsCurrentGCMarkNode = true;
            currentGCState = startingAtBlueLocalVariableRoot;
            controlPanelTextArea.setText(HeapOfFishStrings.traversingBlueRoot);
        }
        break;

    case traversingFromYellowLocalVariableRoot:
        ObjectHandle oh2 = gcHeap.getObjectHandle(currentFishBeingMarked);
        boolean doneWithThisTree = traverseNextFishNode(oh2);
        if (doneWithThisTree) {
            ObjectHandle oh = gcHeap.getObjectHandle(localVars.yellowFish);
            canvas.setYellowFishLocalVarLineColor(Color.black);
            oh.myColor = Color.black;
            fishAreBeingMarked = false;
            yellowFishLocalVarIsCurrentGCMarkNode = true;
            currentGCState = doneWithYellowLocalVariableRoot;
            controlPanelTextArea.setText(HeapOfFishStrings.doneWithYellowRoot);
        }
        break;

    case doneWithYellowLocalVariableRoot:

        yellowFishLocalVarIsCurrentGCMarkNode = false;
        blueFishLocalVarIsCurrentGCMarkNode = true;
        currentGCState = startingAtBlueLocalVariableRoot;
        controlPanelTextArea.setText(HeapOfFishStrings.traversingBlueRoot);
        break;

    case startingAtBlueLocalVariableRoot:

        blueFishLocalVarIsCurrentGCMarkNode = false;
        if (localVars.blueFish != 0 && localVars.blueFish < gcHeap.getObjectPoolSize()) {

            ObjectHandle oh = gcHeap.getObjectHandle(localVars.blueFish);
            blueFishLocalVarIsCurrentGCMarkNode = false;
            oh.myColor = Color.gray;
            canvas.setBlueFishLocalVarLineColor(Color.gray);
            currentFishBeingMarked = localVars.blueFish;
            fishAreBeingMarked = true;
            currentGCState = traversingFromBlueLocalVariableRoot;
        }
        else {
            redFishLocalVarIsCurrentGCMarkNode = true;
            currentGCState = startingAtRedLocalVariableRoot;
            controlPanelTextArea.setText(HeapOfFishStrings.traversingRedRoot);
        }
        break;
```

```
case traversingFromBlueLocalVariableRoot:
   ObjectHandle oh3 = gcHeap.getObjectHandle(currentFishBeingMarked);
   doneWithThisTree = traverseNextFishNode(oh3);
   if (doneWithThisTree) {
      ObjectHandle oh = gcHeap.getObjectHandle(localVars.blueFish);
      canvas.setBlueFishLocalVarLineColor(Color.black);
      oh.myColor = Color.black;
      fishAreBeingMarked = false;
      blueFishLocalVarIsCurrentGCMarkNode = true;
      currentGCState = doneWithBlueLocalVariableRoot;
      controlPanelTextArea.setText(HeapOfFishStrings.doneWithBlueRoot);
   }
   break;

case doneWithBlueLocalVariableRoot:

   blueFishLocalVarIsCurrentGCMarkNode = false;
   redFishLocalVarIsCurrentGCMarkNode = true;
   currentGCState = startingAtRedLocalVariableRoot;
   controlPanelTextArea.setText(HeapOfFishStrings.traversingRedRoot);
   break;

case startingAtRedLocalVariableRoot:

   redFishLocalVarIsCurrentGCMarkNode = false;
   if (localVars.redFish != 0 && localVars.redFish < gcHeap.getObjectPoolSize()) {

      ObjectHandle oh = gcHeap.getObjectHandle(localVars.redFish);
      redFishLocalVarIsCurrentGCMarkNode = false;
      oh.myColor = Color.gray;
      canvas.setRedFishLocalVarLineColor(Color.gray);
      currentFishBeingMarked = localVars.redFish;
      fishAreBeingMarked = true;
      currentGCState = traversingFromRedLocalVariableRoot;
   }
   else {
      currentGCState = empezandoConLosObjetosApuntadosDesdeRegion;
      controlPanelTextArea.setText(HeapOfFishStrings.recorriendoObjetosApuntadosDesdeRegiones);
   }
   break;

case traversingFromRedLocalVariableRoot:
   ObjectHandle oh4 = gcHeap.getObjectHandle(currentFishBeingMarked);
   doneWithThisTree = traverseNextFishNode(oh4);
   if (doneWithThisTree) {
      ObjectHandle oh = gcHeap.getObjectHandle(localVars.redFish);
      canvas.setRedFishLocalVarLineColor(Color.black);
      oh.myColor = Color.black;
      fishAreBeingMarked = false;
      redFishLocalVarIsCurrentGCMarkNode = true;
      currentGCState = doneWithRedLocalVariableRoot;
      controlPanelTextArea.setText(HeapOfFishStrings.doneWithRedRoot);
   }
   break;

case doneWithRedLocalVariableRoot:

   redFishLocalVarIsCurrentGCMarkNode = false;
   currentGCState = empezandoConLosObjetosApuntadosDesdeRegion;
```

```
        controlPanelTextArea.setText(HeapOfFishStrings.recorriendoObjetosApuntadosDesdeRegiones);
        break;



      case empezandoConLosObjetosApuntadosDesdeRegion:

            punterosRegionesHeap = espRegiones.dameListaReferencias();
            visitados.limpiarLista();
            punteroRegionHeapActual = -1;
            indiceRegion = 0;
            for(int i=0;i<punterosRegionesHeap.length;i++)
              punterosRegionesHeap[i].reset();
            currentGCState = siguienteObjetoApuntadoDesdeRegion;
            break;

      case recorriendoDesdeObjetosApuntadosDesdeRegion:
             ObjectHandle oh7 = gcHeap.getObjectHandle(currentFishBeingMarked);
             doneWithThisTree = traverseNextFishNode(oh7);
             if (doneWithThisTree) {
                 ObjectHandle oh = gcHeap.getObjectHandle(punteroRegionHeapActual);
                 oh.myColor = Color.black;
                 fishAreBeingMarked = false;
                 currentGCState = siguienteObjetoApuntadoDesdeRegion;
             }
            break;

      case siguienteObjetoApuntadoDesdeRegion:

          if(punteroRegionHeapActual != -1)
            visitados.insertar(punteroRegionHeapActual);
          do{
            if(indiceRegion < punterosRegionesHeap.length)
            {
              if(punterosRegionesHeap[indiceRegion].haySiguiente())
              {
                punteroRegionHeapActual = punterosRegionesHeap[indiceRegion].siguiente();
                currentFishBeingMarked = punteroRegionHeapActual;
                ObjectHandle oh8 = gcHeap.getObjectHandle(currentFishBeingMarked);
                oh8.myColor = Color.gray;
                fishAreBeingMarked = true;
                currentGCState = recorriendoDesdeObjetosApuntadosDesdeRegion;
              }
              else
                indiceRegion++;
            }
            else
              currentGCState = finObjetosApuntadosDesdeRegion;
           }while(currentGCState != finObjetosApuntadosDesdeRegion && indiceRegion != 0 &&
visitados.pertenece(punteroRegionHeapActual));
            break;

      case finObjetosApuntadosDesdeRegion:

            currentGCState = recorriendoDescendenciaDeObjetosGrises;
            controlPanelTextArea.setText(HeapOfFishStrings.readyToSweepUnmarkedFish);
          break;

      case recorriendoDescendenciaDeObjetosGrises:
```

```
  for (int i = 0; i < gcHeap.getHandlePoolSize(); ++i) {
        ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
        if (!oh.free && oh.myColor == Color.gray)
          {
            ObjectHandle oh9 = oh;
            fishAreBeingMarked = true;
            currentFishBeingMarked = i + 1;
            while(traverseNextFishNode(oh9))
              oh9 = gcHeap.getObjectHandle(currentFishBeingMarked);
            fishAreBeingMarked = false;
          }
        }
        currentGCState = readyToSweepUnmarkedFish;
        controlPanelTextArea.setText(HeapOfFishStrings.readyToSweepUnmarkedFish);
        break;

 case readyToSweepUnmarkedFish:

    int objectsFreedCount = 0;
    for (int i = 0; i < gcHeap.getHandlePoolSize(); ++i) {
        ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
        if (!oh.free) {
          if (oh.myColor == Color.white) {
            gcHeap.freeObject(i + 1);
            ++objectsFreedCount;
          }
        }
    }
    currentGCState = doneSweepingUnmarkedFish;
    String doneSweepingText = HeapOfFishStrings.sweptFish0 + objectsFreedCount
        + HeapOfFishStrings.sweptFish1;
    controlPanelTextArea.setText(doneSweepingText);
    break;

 case doneSweepingUnmarkedFish:
    currentGCState = garbageCollectorIsDone;
    controlPanelTextArea.setText(HeapOfFishStrings.garbageCollectionDone);
    break;

 case garbageCollectorIsDone:
 default:
    break;
 }
}

 // Returns true if done with this tree.
 private boolean traverseNextFishNode(ObjectHandle oh) {
 //ObjectHandle oh = gcHeap.getObjectHandle(currentFishBeingMarked);

   int myFriendIndex = gcHeap.getObjectPool(oh.objectPos);

   if ((myFriendIndex != 0) && (oh.myFriendLineColor == Color.white)) {
    oh.myFriendLineColor = Color.gray;
    ObjectHandle myFriend = gcHeap.getObjectHandle(myFriendIndex);
    myFriend.previousNodeInGCTraversalIsAFish = true;
    myFriend.previousFishInGCTraversal = currentFishBeingMarked;
    if (myFriend.myColor == Color.white) {
      myFriend.myColor = Color.gray;
    }
    currentFishBeingMarked = myFriendIndex;
```

```
    return false;
  }
  else if (oh.fish.getFishColor() == Color.yellow) {
    if (oh.previousNodeInGCTraversalIsAFish) {
      traverseBackFromGrayLine(oh.previousFishInGCTraversal);
      return false;
    }
    return true;
  }

  int myLunchIndex = gcHeap.getObjectPool(oh.objectPos + 1);

  if ((myLunchIndex != 0) && (oh.myLunchLineColor == Color.white)) {
    oh.myLunchLineColor = Color.gray;
    ObjectHandle myLunch = gcHeap.getObjectHandle(myLunchIndex);
    myLunch.previousNodeInGCTraversalIsAFish = true;
    myLunch.previousFishInGCTraversal = currentFishBeingMarked;
    if (myLunch.myColor == Color.white) {
      myLunch.myColor = Color.gray;
    }
    currentFishBeingMarked = myLunchIndex;
    return false;
  }
  else if (oh.fish.getFishColor() == Color.cyan) {
    if (oh.previousNodeInGCTraversalIsAFish) {
      traverseBackFromGrayLine(oh.previousFishInGCTraversal);
      return false;
    }
    return true;
  }

  int mySnackIndex = gcHeap.getObjectPool(oh.objectPos + 2);

  if ((mySnackIndex != 0) && (oh.mySnackLineColor == Color.white)) {
    oh.mySnackLineColor = Color.gray;
    ObjectHandle mySnack = gcHeap.getObjectHandle(mySnackIndex);
    mySnack.previousNodeInGCTraversalIsAFish = true;
    mySnack.previousFishInGCTraversal = currentFishBeingMarked;
    if (mySnack.myColor == Color.white) {
      mySnack.myColor = Color.gray;
    }
    currentFishBeingMarked = mySnackIndex;
    return false;
  }
  else if (oh.previousNodeInGCTraversalIsAFish) {
    traverseBackFromGrayLine(oh.previousFishInGCTraversal);
    return false;
  }

  return true;
}

private void traverseBackFromGrayLine(int fishObjectHandle) {

  ObjectHandle oh = gcHeap.getObjectHandle(fishObjectHandle);

  int myFriendIndex = gcHeap.getObjectPool(oh.objectPos);

  if ((myFriendIndex != 0) && (oh.myFriendLineColor == Color.gray)) {
    ObjectHandle myFriend = gcHeap.getObjectHandle(myFriendIndex);
```

```
      myFriend.previousNodeInGCTraversalIsAFish = false;
      myFriend.myColor = Color.black;
      oh.myFriendLineColor = Color.black;
      currentFishBeingMarked = fishObjectHandle;
      return;
    }

    if (oh.fish.getFishColor() == Color.yellow) {
      return; // exception condition
    }

    int myLunchIndex = gcHeap.getObjectPool(oh.objectPos + 1);

    if ((myLunchIndex != 0) && (oh.myLunchLineColor == Color.gray)) {
      ObjectHandle myLunch = gcHeap.getObjectHandle(myLunchIndex);
      myLunch.previousNodeInGCTraversalIsAFish = false;
      myLunch.myColor = Color.black;
      oh.myLunchLineColor = Color.black;
      currentFishBeingMarked = fishObjectHandle;
      return;
    }

    if (oh.fish.getFishColor() == Color.cyan) {
      return; // exception condition
    }

    int mySnackIndex = gcHeap.getObjectPool(oh.objectPos + 2);

    if ((mySnackIndex != 0) && (oh.mySnackLineColor == Color.gray)) {
      ObjectHandle mySnack = gcHeap.getObjectHandle(mySnackIndex);
      mySnack.previousNodeInGCTraversalIsAFish = false;
      mySnack.myColor = Color.black;
      oh.mySnackLineColor = Color.black;
      currentFishBeingMarked = fishObjectHandle;
      return;
    }
  }


}
```

La clase *MinorCollection* difiere de la clase *MajorCollection* en que hay que explorar solamente las variables si están referenciando a un objeto de la generación nueva del heap, hay que hacer un chequeo de las referencias de objetos de la generación antigua a objetos de la generación nueva, contenidas en lista, hay que ignorar las referencias procedentes de las regiones que se dirijan a objetos de la generación antigua, el método *transverseNextFishNode* se ha modificado para que la exploración se detenga además de al haber alcanzado un objeto sin hijos, al haber alcanzado un objeto que todos sus hijos sean de la generación antigua y en el estado *readyToSweepUnmarkedFish* solamente se recorren los objetos de la generación nueva, liberando la memoria de los que queden coloreados de blanco, incrementado el contador de veces sobrevividas en el caso en que sobrevivan, pasandolos a la generación antigua en caso de que este valor se iguale al de la constante *vecesSobrevividasParaPromocionar*, eliminando todas sus instancias en la lista e insertando una instancia de cada hijo suyo que permanezca en la generación nueva.

```
public class MinorCollection extends RecoleccionAbstracta{

  private ListaReferencias lista;

  private int pezInicialActual;

  private final int vecesSobrevividasParaPromocionar = 10;

  private final int garbageCollectorSinEmpezar = 0;

  private final int empezandoDesdeLaVariableLocalYellowFish = 1;
  private final int recorriendoDesdeLaVariableLocalYellowFish = 2;
  private final int finalizandoConLaVariableLocalYellowFish = 3;

  private final int empezandoDesdeLaVariableLocalBlueFish = 4;
  private final int recorriendoDesdeLaVariableLocalBlueFish = 5;
  private final int finalizandoConLaVariableLocalBlueFish = 6;

  private final int empezandoDesdeLaVariableLocalRedFish = 7;
  private final int recorriendoDesdeLaVariableLocalRedFish = 8;
  private final int finalizandoConLaVariableLocalRedFish = 9;

  private final int empezandoConLosObjetosNuevosApuntadosDesdeAntiguos = 10;
  private final int recorriendoDesdeObjetosNuevosApuntadosDesdeAntiguos = 11;
  private final int siguienteObjetoNuevoApuntadoDesdeAntiguos = 12;
  private final int finObjetosNuevosApuntadosDesdeAntiguos = 13;

  private final int empezandoConLosObjetosApuntadosDesdeRegion = 14;
  private final int recorriendoDesdeObjetosApuntadosDesdeRegion = 15;
  private final int siguienteObjetoApuntadoDesdeRegion = 16;
  private final int finObjetosApuntadosDesdeRegion = 17;

  private final int recorriendoDescendenciaDeObjetosGrises = 18;

  private final int readyToSweepUnmarkedFish = 19;
  private final int doneSweepingUnmarkedFish = 20;
  private final int garbageCollectorIsDone = 21;

  public MinorCollection(GarbageCollectCanvas gcc, GCHeap heap, EspacioRegiones espRegiones,
ReferenciasIniciales ri, HeapOfFishTextArea ta) {
     super(gcc,heap,espRegiones,ri.dameVariablesLocales(),ta);
     lista = ri.dameListaObjetosNuevos();
   }

  public int getGarbageCollectorHasNotStarted()
  {
    return garbageCollectorSinEmpezar;
  }

  public int getGarbageCollectorIsDone()
  {
    return garbageCollectorIsDone;
  }


  public synchronized void nextGCStep()
  {
    switch (currentGCState) {
```

```
case garbageCollectorSinEmpezar:
   yellowFishLocalVarIsCurrentGCMarkNode = true;
   currentGCState = empezandoDesdeLaVariableLocalYellowFish;
   controlPanelTextArea.setText(HeapOfFishStrings.traversingYellowRoot);
   break;

case empezandoDesdeLaVariableLocalYellowFish:

   yellowFishLocalVarIsCurrentGCMarkNode = false;
   if (localVars.yellowFish != 0 && localVars.yellowFish < gcHeap.getObjectPoolSize()) {

      ObjectHandle oh = gcHeap.getObjectHandle(localVars.yellowFish);
      if(oh.objectPos > gcHeap.dameFrontera())
      {
        blueFishLocalVarIsCurrentGCMarkNode = true;
        currentGCState = empezandoDesdeLaVariableLocalBlueFish;
        controlPanelTextArea.setText(HeapOfFishStrings.traversingBlueRoot);
      }
      else
      {
        yellowFishLocalVarIsCurrentGCMarkNode = false;
        oh.myColor = Color.gray;
        canvas.setYellowFishLocalVarLineColor(Color.gray);
        currentFishBeingMarked = localVars.yellowFish;
        fishAreBeingMarked = true;

        currentGCState = recorriendoDesdeLaVariableLocalYellowFish;
      }
   }
   else {
      blueFishLocalVarIsCurrentGCMarkNode = true;
      currentGCState = empezandoDesdeLaVariableLocalBlueFish;
      controlPanelTextArea.setText(HeapOfFishStrings.traversingBlueRoot);
   }
   break;

case recorriendoDesdeLaVariableLocalYellowFish:
   ObjectHandle oh2 = gcHeap.getObjectHandle(currentFishBeingMarked);
   boolean doneWithThisTree = traverseNextFishNode(oh2);
   if (doneWithThisTree) {
      ObjectHandle oh = gcHeap.getObjectHandle(localVars.yellowFish);
      canvas.setYellowFishLocalVarLineColor(Color.black);
      oh.myColor = Color.black;
      fishAreBeingMarked = false;
      yellowFishLocalVarIsCurrentGCMarkNode = true;
      currentGCState = finalizandoConLaVariableLocalYellowFish;
      controlPanelTextArea.setText(HeapOfFishStrings.doneWithYellowRoot);
   }
   break;

case finalizandoConLaVariableLocalYellowFish:

   yellowFishLocalVarIsCurrentGCMarkNode = false;
   blueFishLocalVarIsCurrentGCMarkNode = true;
   currentGCState = empezandoDesdeLaVariableLocalBlueFish;
   controlPanelTextArea.setText(HeapOfFishStrings.traversingBlueRoot);
   break;

case empezandoDesdeLaVariableLocalBlueFish:
```

```
    blueFishLocalVarIsCurrentGCMarkNode = false;
    if (localVars.blueFish != 0 && localVars.blueFish < gcHeap.getObjectPoolSize()) {

        ObjectHandle oh = gcHeap.getObjectHandle(localVars.blueFish);
        if(oh.objectPos > gcHeap.dameFrontera())
        {
          redFishLocalVarIsCurrentGCMarkNode = true;
          currentGCState = empezandoDesdeLaVariableLocalRedFish;
          controlPanelTextArea.setText(HeapOfFishStrings.traversingRedRoot);
        }
        else
        {
          blueFishLocalVarIsCurrentGCMarkNode = false;
          oh.myColor = Color.gray;
          canvas.setBlueFishLocalVarLineColor(Color.gray);
          currentFishBeingMarked = localVars.blueFish;
          fishAreBeingMarked = true;
          currentGCState = recorriendoDesdeLaVariableLocalBlueFish;
        }
    }
    else {
        redFishLocalVarIsCurrentGCMarkNode = true;
        currentGCState = empezandoDesdeLaVariableLocalRedFish;
        controlPanelTextArea.setText(HeapOfFishStrings.traversingRedRoot);
    }
    break;

case recorriendoDesdeLaVariableLocalBlueFish:
    ObjectHandle oh3 = gcHeap.getObjectHandle(currentFishBeingMarked);
    doneWithThisTree = traverseNextFishNode(oh3);
    if (doneWithThisTree) {
        ObjectHandle oh = gcHeap.getObjectHandle(localVars.blueFish);
        canvas.setBlueFishLocalVarLineColor(Color.black);
        oh.myColor = Color.black;
        fishAreBeingMarked = false;
        blueFishLocalVarIsCurrentGCMarkNode = true;
        currentGCState = finalizandoConLaVariableLocalBlueFish;
        controlPanelTextArea.setText(HeapOfFishStrings.doneWithBlueRoot);
    }
    break;

case finalizandoConLaVariableLocalBlueFish:

    blueFishLocalVarIsCurrentGCMarkNode = false;
    redFishLocalVarIsCurrentGCMarkNode = true;
    currentGCState = empezandoDesdeLaVariableLocalRedFish;
    controlPanelTextArea.setText(HeapOfFishStrings.traversingRedRoot);
    break;

case empezandoDesdeLaVariableLocalRedFish:

    redFishLocalVarIsCurrentGCMarkNode = false;
    if (localVars.redFish != 0 && localVars.redFish < gcHeap.getObjectPoolSize()) {

        ObjectHandle oh = gcHeap.getObjectHandle(localVars.redFish);
        if(oh.objectPos > gcHeap.dameFrontera())
        {
          currentGCState = empezandoConLosObjetosNuevosApuntadosDesdeAntiguos;
          controlPanelTextArea.setText(HeapOfFishStrings.pasandoNuevosObjetos);
        }
```

```
        else
        {
          redFishLocalVarIsCurrentGCMarkNode = false;
          oh.myColor = Color.gray;
          canvas.setRedFishLocalVarLineColor(Color.gray);
          currentFishBeingMarked = localVars.redFish;
          fishAreBeingMarked = true;
          currentGCState = recorriendoDesdeLaVariableLocalRedFish;
        }
    }
    else {
        currentGCState = empezandoConLosObjetosNuevosApuntadosDesdeAntiguos;
        controlPanelTextArea.setText(HeapOfFishStrings.pasandoNuevosObjetos);
    }
    break;

case recorriendoDesdeLaVariableLocalRedFish:
    ObjectHandle oh4 = gcHeap.getObjectHandle(currentFishBeingMarked);
    doneWithThisTree = traverseNextFishNode(oh4);
    if (doneWithThisTree) {
        ObjectHandle oh = gcHeap.getObjectHandle(localVars.redFish);
        canvas.setRedFishLocalVarLineColor(Color.black);
        oh.myColor = Color.black;
        fishAreBeingMarked = false;
        redFishLocalVarIsCurrentGCMarkNode = true;
        currentGCState = finalizandoConLaVariableLocalRedFish;
        controlPanelTextArea.setText(HeapOfFishStrings.doneWithRedRoot);
    }
    break;

case finalizandoConLaVariableLocalRedFish:

    redFishLocalVarIsCurrentGCMarkNode = false;
    currentGCState = empezandoConLosObjetosNuevosApuntadosDesdeAntiguos;
    controlPanelTextArea.setText(HeapOfFishStrings.pasandoNuevosObjetos);
    break;

case empezandoConLosObjetosNuevosApuntadosDesdeAntiguos:

        lista.reset();
        currentGCState = siguienteObjetoNuevoApuntadoDesdeAntiguos;
        break;

case recorriendoDesdeObjetosNuevosApuntadosDesdeAntiguos:
          ObjectHandle oh6 = gcHeap.getObjectHandle(currentFishBeingMarked);
          doneWithThisTree = traverseNextFishNode(oh6);
          if (doneWithThisTree) {
              ObjectHandle oh = gcHeap.getObjectHandle(pezInicialActual);
              oh.myColor = Color.black;
              fishAreBeingMarked = false;
              currentGCState = siguienteObjetoNuevoApuntadoDesdeAntiguos;
          }
        break;

case siguienteObjetoNuevoApuntadoDesdeAntiguos:


        if(lista.haySiguiente())
        {
          pezInicialActual = lista.siguiente();
```

```
          currentFishBeingMarked = pezInicialActual;
          ObjectHandle oh5 = gcHeap.getObjectHandle(currentFishBeingMarked);
          oh5.myColor = Color.gray;
          fishAreBeingMarked = true;
          currentGCState = recorriendoDesdeObjetosNuevosApuntadosDesdeAntiguos;
        }
        else
          currentGCState = finObjetosNuevosApuntadosDesdeAntiguos;
        break;

  case finObjetosNuevosApuntadosDesdeAntiguos:

        currentGCState = empezandoConLosObjetosApuntadosDesdeRegion;
        controlPanelTextArea.setText(HeapOfFishStrings.recorriendoObjetosApuntadosDesdeRegiones);
       break;

  case empezandoConLosObjetosApuntadosDesdeRegion:

            punterosRegionesHeap = espRegiones.dameListaReferencias();
            visitados.limpiarLista();
            punteroRegionHeapActual = -1;
            indiceRegion = 0;
            for(int i=0;i<punterosRegionesHeap.length;i++)
              punterosRegionesHeap[i].reset();
            currentGCState = siguienteObjetoApuntadoDesdeRegion;
            break;

  case recorriendoDesdeObjetosApuntadosDesdeRegion:
              ObjectHandle oh7 = gcHeap.getObjectHandle(currentFishBeingMarked);
              doneWithThisTree = traverseNextFishNode(oh7);
              if (doneWithThisTree) {
                  ObjectHandle oh = gcHeap.getObjectHandle(punteroRegionHeapActual);
                  oh.myColor = Color.black;
                  fishAreBeingMarked = false;
                  currentGCState = siguienteObjetoApuntadoDesdeRegion;
              }
            break;

  case siguienteObjetoApuntadoDesdeRegion:

        ObjectHandle oh8 = null;
        if(punteroRegionHeapActual != -1)
          visitados.insertar(punteroRegionHeapActual);
        do{
            if(indiceRegion < punterosRegionesHeap.length)
            {
              if(punterosRegionesHeap[indiceRegion].haySiguiente())
              {
                punteroRegionHeapActual = punterosRegionesHeap[indiceRegion].siguiente();
                oh8 = gcHeap.getObjectHandle(punteroRegionHeapActual);
                currentGCState = recorriendoDesdeObjetosApuntadosDesdeRegion;
              }
              else
                indiceRegion++;
            }
            else
              currentGCState = finObjetosApuntadosDesdeRegion;
         }while(currentGCState != finObjetosApuntadosDesdeRegion && (punteroRegionHeapActual ==
-1 ||
              gcHeap.getObjectHandle(punteroRegionHeapActual).objectPos > gcHeap.dameFrontera() ||
```

```
            (indiceRegion != 0 && visitados.pertenece(punteroRegionHeapActual))));
        if(currentGCState != finObjetosApuntadosDesdeRegion)
        {
          oh8.myColor = Color.gray;
          currentFishBeingMarked = punteroRegionHeapActual;
          fishAreBeingMarked = true;
        }
            break;

case finObjetosApuntadosDesdeRegion:

            currentGCState = recorriendoDescendenciaDeObjetosGrises;
            controlPanelTextArea.setText(HeapOfFishStrings.readyToSweepUnmarkedFish);
          break;

case recorriendoDescendenciaDeObjetosGrises:

  for (int i = 0; i < gcHeap.getHandlePoolSize(); ++i) {
    ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
    if(!oh.free && oh.objectPos < gcHeap.dameFrontera())
        if(oh.myColor == Color.gray)
        {
          ObjectHandle oh9 = oh;
          fishAreBeingMarked = true;
          currentFishBeingMarked = i + 1;
          while(traverseNextFishNode(oh9))
            oh9 = gcHeap.getObjectHandle(currentFishBeingMarked);
          fishAreBeingMarked = false;
        }
  }
  currentGCState = readyToSweepUnmarkedFish;
  controlPanelTextArea.setText(HeapOfFishStrings.readyToSweepUnmarkedFish);

  break;


case readyToSweepUnmarkedFish:

    int objectsFreedCount = 0;
    ArrayList objetosPromocionados = new ArrayList();
    for (int i = 0; i < gcHeap.getHandlePoolSize(); ++i) {
        ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
        if(!oh.free && oh.objectPos < gcHeap.dameFrontera())
          if (oh.myColor == Color.white) {
            gcHeap.freeObject(i + 1);
            ++objectsFreedCount;
          }
          else
          {
            oh.vecesSobrevividasAlGarbageCollector++;
            if(oh.vecesSobrevividasAlGarbageCollector == vecesSobrevividasParaPromocionar)
            {
              gcHeap.slideObjectDown(oh);
              lista.borrarTodos(i + 1);
              objetosPromocionados.add(oh);
            }


          }
    }
```

```
      Iterator it = objetosPromocionados.iterator();
      ObjectHandle oph, ohhijo;
      int dirHijo;
      while(it.hasNext())
      {
        oph = (ObjectHandle) it.next();
        dirHijo = gcHeap.getObjectPool(oph.objectPos);
        if(dirHijo != 0)
        {
          ohhijo = gcHeap.getObjectHandle(dirHijo);
          if(ohhijo.objectPos < gcHeap.dameFrontera())
            lista.insertar(dirHijo);
        }

        if(oph.fish.fishColor != Color.yellow)
        {
          dirHijo = gcHeap.getObjectPool(oph.objectPos + 1);
          if(dirHijo != 0)
          {
            ohhijo = gcHeap.getObjectHandle(dirHijo);
            if(ohhijo.objectPos < gcHeap.dameFrontera())
              lista.insertar(dirHijo);
          }
        }
        if(oph.fish.fishColor == Color.red)
        {
          dirHijo = gcHeap.getObjectPool(oph.objectPos + 2);
          if(dirHijo != 0)
          {
            ohhijo = gcHeap.getObjectHandle(dirHijo);
            if(ohhijo.objectPos < gcHeap.dameFrontera())
              lista.insertar(dirHijo);
          }
        }
      }
      currentGCState = doneSweepingUnmarkedFish;
      String doneSweepingText = HeapOfFishStrings.sweptFish0 + objectsFreedCount
          + HeapOfFishStrings.sweptFish1;
      controlPanelTextArea.setText(doneSweepingText);
      break;

    case doneSweepingUnmarkedFish:
      currentGCState = garbageCollectorIsDone;
      controlPanelTextArea.setText(HeapOfFishStrings.garbageCollectionDone);
      break;

    case garbageCollectorIsDone:
    default:
      break;
    }
  }

  // Returns true if done with this tree.
  private boolean traverseNextFishNode(ObjectHandle oh) {
  //ObjectHandle oh = gcHeap.getObjectHandle(currentFishBeingMarked);

    int myFriendIndex = gcHeap.getObjectPool(oh.objectPos);

    if ((myFriendIndex != 0) && (oh.myFriendLineColor == Color.white)) {
      oh.myFriendLineColor = Color.gray;
```

```
  ObjectHandle myFriend = gcHeap.getObjectHandle(myFriendIndex);
  if(myFriend.objectPos > gcHeap.dameFrontera())
  {
    if (oh.previousNodeInGCTraversalIsAFish) {
       traverseBackFromGrayLine(oh.previousFishInGCTraversal);
       return false;
    }
    else
      return true;
  }
  myFriend.previousNodeInGCTraversalIsAFish = true;
  myFriend.previousFishInGCTraversal = currentFishBeingMarked;
  if (myFriend.myColor == Color.white) {
    myFriend.myColor = Color.gray;
  }
  currentFishBeingMarked = myFriendIndex;
  return false;
}
else if (oh.fish.getFishColor() == Color.yellow) {
  if (oh.previousNodeInGCTraversalIsAFish) {
    traverseBackFromGrayLine(oh.previousFishInGCTraversal);
    return false;
  }
  return true;
}

int myLunchIndex = gcHeap.getObjectPool(oh.objectPos + 1);

if ((myLunchIndex != 0) && (oh.myLunchLineColor == Color.white)) {
  oh.myLunchLineColor = Color.gray;
  ObjectHandle myLunch = gcHeap.getObjectHandle(myLunchIndex);
  if(myLunch.objectPos > gcHeap.dameFrontera())
  {
    if (oh.previousNodeInGCTraversalIsAFish) {
       traverseBackFromGrayLine(oh.previousFishInGCTraversal);
       return false;
    }
    else
      return true;
  }
  myLunch.previousNodeInGCTraversalIsAFish = true;
  myLunch.previousFishInGCTraversal = currentFishBeingMarked;
  if (myLunch.myColor == Color.white) {
    myLunch.myColor = Color.gray;
  }
  currentFishBeingMarked = myLunchIndex;
  return false;
}
else if (oh.fish.getFishColor() == Color.cyan) {
  if (oh.previousNodeInGCTraversalIsAFish) {
    traverseBackFromGrayLine(oh.previousFishInGCTraversal);
    return false;
  }
  return true;
}

int mySnackIndex = gcHeap.getObjectPool(oh.objectPos + 2);

if ((mySnackIndex != 0) && (oh.mySnackLineColor == Color.white)) {
  oh.mySnackLineColor = Color.gray;
```

```
    ObjectHandle mySnack = gcHeap.getObjectHandle(mySnackIndex);
    if(mySnack.objectPos > gcHeap.dameFrontera())
    {
      if (oh.previousNodeInGCTraversalIsAFish) {
        traverseBackFromGrayLine(oh.previousFishInGCTraversal);
        return false;
      }
      else
        return true;
    }
    mySnack.previousNodeInGCTraversalIsAFish = true;
    mySnack.previousFishInGCTraversal = currentFishBeingMarked;
    if (mySnack.myColor == Color.white) {
      mySnack.myColor = Color.gray;
    }
    currentFishBeingMarked = mySnackIndex;
    return false;
  }
  else if (oh.previousNodeInGCTraversalIsAFish) {
    traverseBackFromGrayLine(oh.previousFishInGCTraversal);
    return false;
  }

  return true;
}

private void traverseBackFromGrayLine(int fishObjectHandle) {

  ObjectHandle oh = gcHeap.getObjectHandle(fishObjectHandle);

  int myFriendIndex = gcHeap.getObjectPool(oh.objectPos);

  if ((myFriendIndex != 0) && (oh.myFriendLineColor == Color.gray)) {
    ObjectHandle myFriend = gcHeap.getObjectHandle(myFriendIndex);
    myFriend.previousNodeInGCTraversalIsAFish = false;
    myFriend.myColor = Color.black;
    oh.myFriendLineColor = Color.black;
    currentFishBeingMarked = fishObjectHandle;
    return;
  }

  if (oh.fish.getFishColor() == Color.yellow) {
    return; // exception condition
  }

  int myLunchIndex = gcHeap.getObjectPool(oh.objectPos + 1);

  if ((myLunchIndex != 0) && (oh.myLunchLineColor == Color.gray)) {
    ObjectHandle myLunch = gcHeap.getObjectHandle(myLunchIndex);
    myLunch.previousNodeInGCTraversalIsAFish = false;
    myLunch.myColor = Color.black;
    oh.myLunchLineColor = Color.black;
    currentFishBeingMarked = fishObjectHandle;
    return;
  }

  if (oh.fish.getFishColor() == Color.cyan) {
    return; // exception condition
  }
```

```
    int mySnackIndex = gcHeap.getObjectPool(oh.objectPos + 2);

    if ((mySnackIndex != 0) && (oh.mySnackLineColor == Color.gray)) {
      ObjectHandle mySnack = gcHeap.getObjectHandle(mySnackIndex);
      mySnack.previousNodeInGCTraversalIsAFish = false;
      mySnack.myColor = Color.black;
      oh.mySnackLineColor = Color.black;
      currentFishBeingMarked = fishObjectHandle;
      return;
    }
  }



}
```

v) Clase *ReferenciasIniciales*.

La clase *ReferenciasIniciales*, además de contener las variables contiene una lista donde se almacenan las referencias desde objetos antiguos a objetos nuevos (*objetosNuevos*).

```
public class ReferenciasIniciales {

  private GCHeap gcHeap;
  private LocalVariables localVars;
  private ListaReferencias objetosNuevos;

  public ReferenciasIniciales(GCHeap gc, LocalVariables lv) {
    gcHeap = gc;
    localVars=lv;
    objetosNuevos = new ListaReferencias();
  }

  public LocalVariables dameVariablesLocales()
  {
    return localVars;
  }

  public ListaReferencias dameListaObjetosNuevos()
  {
    return objetosNuevos;
  }


}
```

w) Clase Region.

La clase *Region* se diferencia de la clase *GCHeap* en que no es necesario dividir la zona de objetos ni implementar promoción de los mismos y en que hay que establecer una dirección a partir de la cual se asignan las direcciones de memoria (*direccionInicio*). El

método *getObjectHandle* ha sido sobreescrito para poder acceder a los manejadores con la dirección general del sistema, en lugar de la dirección dentro de la región, dejando el método original como *dameManejador*.

```
public class Region extends EspacioAlmacenamiento{
  private int direccionInicio;

  public Region(int hndlPoolSize, int objPoolSize, int dirInicio) {
    super(hndlPoolSize, objPoolSize);
    direccionInicio = dirInicio;
  }

  public int allocateObject(int bytesNeeded, FishIcon fish) {
    int valDev = super.allocateObject(bytesNeeded,fish);
    if(valDev != 0)
    {
      valDev += direccionInicio;
    }
    return valDev;
  }

  public ObjectHandle getObjectHandle(int direccion)
  {
    return super.getObjectHandle(direccion - direccionInicio);
  }

  public ObjectHandle dameManejador(int direccion)
  {
    return super.getObjectHandle(direccion);
  }

  public int dameDireccionInicio()
  {
    return direccionInicio;
  }


}
```

x) Interface *Tarea* y clases *TareaNormal* y *TareaCritica*.

La interface *Tarea* representa los tipos de tarea.

```
public interface Tarea {

  public void chequearReferencias(int dir1,int dir2antigua,int dir2nueva,int offset,GCHeap
gcHeap,ListaReferencias listaR);

}
```

Está implementada por las clases *TareaNormal* y *TareaCritica* y su función es implementar las acciones descritas en el punto *3.2*. Para implementar estas acciones se tienen la dirección origen (*dir1*) y las direcciones destino anterior (*dir2antigua*) y actual (*dir2nueva*) y a partir de ellas se sabe si se establece, se anula o se sustituye una referencia procedente de una región y con destino al heap.

```
public class TareaNormal implements Tarea{

  public TareaNormal() {
  }


  public void chequearReferencias(int dir1,int dir2antigua,int dir2nueva,int offset,GCHeap
gcHeap,ListaReferencias listaR)
  {
    if(dir1 > gcHeap.getObjectPoolSize())
    {
      if(dir2antigua <= gcHeap.getObjectPoolSize())
        listaR.borrar(dir2antigua);

       if(dir2nueva <= gcHeap.getObjectPoolSize())
        listaR.insertar(dir2nueva);
    }

  }
}
```

```
public class TareaCritica implements Tarea{

  public TareaCritica() {
  }


  public void chequearReferencias(int dir1,int dir2antigua,int dir2nueva,int offset,GCHeap
gcHeap,ListaReferencias listaR)
  {
    if(dir1 > gcHeap.getObjectPoolSize() &&
      dir2nueva <= gcHeap.getObjectPoolSize())
        listaR.insertar(dir2nueva);
  }
}
```

Bibliografía

- Paul R. Wilson. *Uniprocessor Garbage Collection Techniques*. University of Texas.
- Deitel y Deitel. *Cómo programar en JAVA*. Editorial Prentice Hall.
- Eckel. *Piensa en Java*. Editorial Prentice Hall.
- Bill Venners. *Garbage collection*. Chapter 9 of Inside the Java Virtual Machine.
- Página de SUN(http://java.sun.com/docs/hotspot/gc1.4.2/). *Tuning Garbage Collection with the 1.4.2 Java Virtual Machine*
- Steven Holzner. *La biblia. JAVA 2*. Editorial ANAYA
- Clyde Law. *Garbage Collection in Regions*