Sistemas Informáticos
Curso 2005-2006

# *Recolector de memoria incremental para sistemas Java de tiempo real*

Diego Raboso Rivas

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

Facultad de Informática
Universidad Complutense de Madrid

# SUMARIO

# ÍNDICE DE FIGURAS

# RESUMEN DEL PROYECTO

En este proyecto se han estudiado los diferentes algoritmos de recolección de memoria, prestando especial atención a los algoritmos de recolección de memoria incrementales y su relación con los sistemas de tiempo real. Por sistema de tiempo real entendemos aquél que interacciona repetidamente con su entorno físico en un plazo de tiempo. Para su correcto funcionamiento, además de producir resultados correctos, los tiene que producir en un tiempo determinado. Es decir, no deben detenerse mientras se encuentran en ejecución, por ese motivo son importantes los recolectores de memoria incrementales en este tipo de sistemas, ya que para ejecutarse no necesitan que el sistema esté detenido. El objetivo del proyecto es modificar el recolector de memoria de una Máquina Virtual de Java para que trabaje de forma incremental. Para conseguirlo se deben introducir las Write Barriers. En la realización del proyecto se usó como punto de partida una aplicación existente, pensada para la enseñanza que realizaba la recolección de memoria de un heap de manera no incremental.

# ABSTRACT

In this project we have studied the different garbage collectors algorithms, lending special attention to the incrementales garbage collectors and its relation with the real-time systems. By real-time system we understand that one that interacts repeatedly with its physical surroundings in a term of time. For its correct operation, besides to produce correct results, it has them to produce in a certain time. That is to say, they do not have to stop while they are in execution, by that reason are important the incrementales garbage collectors in this type of systems, since to execute itself they do not need that the system is lengthy. The objective of the project is to modify the garbage collector of a Java Virtual Machine so that it works of incremental form. In order to obtain it they are due to introduce the Write Barriers. For the accomplishment of the project we used like departure point an existing application, thought for the education that made the garbage collection of no incremental way.

# LISTA DE PALABRAS CLAVE / KEYWORDS LIST

Recolector de Memoria.

Heap.

Write Barrier.

Sistemas de tiempo real.

Lectores – Escritores.

# LICENCIA

Diego Raboso Rivas autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, la memoria desarrollada en este proyecto.

Madrid, a 25 de Septiembre de 2006

Diego Raboso Rivas

# 1. INTRODUCCIÓN

## *1.1 Planteamiento del problema*

El objetivo de nuestro proyecto es realizar un recolector de memoria incremental sobre un heap, de forma que los procesos que utilizan el heap no tengan que detenerse cuando se realiza la recolección de memoria. De esta forma se solucionaría el problema de recolección de memoria en sistemas de tiempo real.

## *1.2 ¿Qué debe hacer el sistema?*

El sistema creado debe realizar la recolección de memoria de un heap, en concreto debe liberar la parte de memoria considerada "basura", es decir, la correspondiente a los objetos no utilizados. Esta liberación de memoria se hará de forma incremental, de manera que no sea necesario detener el resto de los procesos que necesiten acceder al heap mientras dura el proceso de recolección. Para conseguirlo, se partió de una aplicación ya existente pensada para la enseñanza. La aplicación consistía en un simulador del funcionamiento de un espacio de memoria en el que los objetos son representados como peces, los peces pueden ser de tres tipos: **Peces Rojos, Peces Azules y Peces Amarillos**, y están relacionados entre sí mediante tres tipos de relaciones: **Amistad, Comida y Aperitivo**. Además existen tres nodos raíces, uno para cada tipo de pez. La aplicación permite crear objetos sobre el espacio de memoria, asignar relaciones entre dichos objetos, así como realizar el proceso de recolección de memoria. La modificación realizada por nosotros consistió en sustituir este algoritmo de recolección de memoria por otro recolector de memoria incremental. Sobre este simulador se realizarán los casos de prueba necesarios para demostrar si es posible o no lo es introducir nuevas ideas y funcionalidades en recolectores de memoria reales. Al tratarse de un simulador pequeño, los resultados obtenidos tienen poca validez, sin embargo, pueden utilizarse como punto de partida para estudios más profundos.

## *1.3 ¿Cómo se ha desarrollado el proyecto?*

Tras un período previo de familiarización con la temática a abordar consistente en la lectura de artículos y reuniones con la profesora directora del proyecto, hubo que decidir hacia dónde se encaminaba el proyecto. Surgieron varias opciones, como la de trabajar directamente con el recolector de memoria de la Máquina Virtual de Java. Finalmente se optó por realizar un simulador del recolector de memoria de un heap y para ello se partió de una aplicación ya existente. El proceso seguido para el desarrollo del proyecto ha sido un *modelo iterativo e incremental*. Se tomó como punto de partida una aplicación diseñada para la docencia, y en cada paso se le fueron añadiendo nuevas funcionalidades. De este modo al final de cada paso, se obtenía una nueva versión del sistema más completa que la anterior. En total han sido 4 las versiones por las que ha pasado la aplicación. Pensamos que esta era la mejor forma de trabajar ya que los requisitos finales del sistema no estaban del todo definidos al principio del trabajo y se fueron definiendo durante las distintas reuniones mantenidas. Además, a medida que se iba desarrollando el proyecto, íbamos apuntando las modificaciones y los motivos por los que se realizaban las modificaciones. De este

modo en el momento de escribir la memoria disponíamos de un guión de todo el trabajo realizado.

## *1.4 ¿Por qué se eligió esta aplicación?*

Fueron varios los motivos que hicieron que se eligiera esta aplicación como punto de partida. En primer lugar nos proporcionaba implementados algunos elementos necesarios como son el heap, las relaciones entre los objetos… con lo que evitábamos tener que implementarlos nosotros. Además disponía de una interfaz gráfica que nos pareció muy interesante para exponer nuestro trabajo, ya que consideramos que de esta manera se entendería mejor nuestro algoritmo. Una vez que vimos que aparentemente la aplicación podía sernos útil como punto de partida, se procedió a estudiar el código para ver su estructura y estimar si sería costoso introducir modificaciones. Se comprobó que el código se podía entender ya que disponía de comentarios, por lo que no sería muy costoso realizar modificaciones.

# 2. ALGORITMOS DE RECOLECCIÓN DE MEMORIA

Se llama recolector de memoria al proceso que se encarga de detectar y liberar automáticamente los fragmentos de memoria que han dejado de ser referenciados por el programa en ejecución. [Knu69, Coh81, App91]. Este tipo de procesos son necesarios en lenguajes de programación en los que no existan instrucciones específicas para la liberación de memoria, cómo son “free” o “dispose”. Estos algoritmos son de dos tipos: **contadores de referencia** y **basados en traza**. La mayoría de los algoritmos de recolección de memoria basados en traza, por ejemplo el algoritmo de *mark and sweep*, constan de dos fases, una primera fase en la que se identifican los objetos que deben ser liberados y una segunda en la que se liberan los objetos que se consideraron “basura” en la fase anterior [Wil92].

En este capítulo se describen los principales algoritmos para la recolección de memoria, haciendo hincapié en las ventajas y desventajas que conlleva el uso de cada tipo de técnica.

## *2.1 Contador de Referencias*

La idea de este tipo de algoritmos es contar las veces que un objeto es referenciado, para ello, cada objeto tiene asociado un contador de referencias, que se incrementa cada vez que se crea una nueva referencia hacia dicho objeto, análogamente, el contador de referencias se decrementa cada vez que una referencia se eliminada. Cuando el contador de referencias del objeto es igual a cero, indica que dicho objeto no es referenciado, por lo que se libera [Wil92]. La variación en el número de referencias sucede por ejemplo cuando una referencia se asigna a una variable, cuando una referencia pasa como argumento a un método o cuando se devuelve una referencia desde un método.

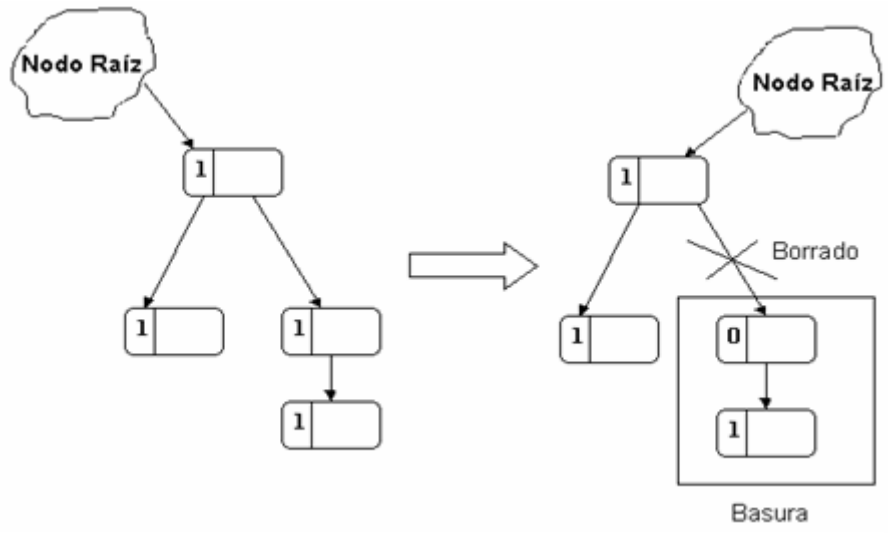



Figura 2.1: Algoritmo contador de referencias.

Ventajas:

La principal ventaja del algoritmo es la sencillez y que puede ejecutarse en pequeños intervalos de tiempos lo que hace que esta técnica sea buena para aplicaciones de tiempo real.

Inconvenientes:

Es necesario añadir un nuevo campo a cada objeto, el contador de referencias. El elevado número de actualizaciones de los contadores puede producir una sobrecarga. Otro problema que puede aparecer es el problema del overflow del contador, puede ocurrir que el número de referencias del objeto supere la capacidad del contador, si esto ocurre, el valor del contador estaría corrupto, no reflejaría el número real de referencias, es más, puede ser que el valor del contador sea cero, con lo que el objeto sería liberado y el sistema fallaría. Existen varias soluciones para este problema, la primera es obvia, ampliar el número de bits del contador de referencias. Otra posible solución es fijar el valor máximo del contador, de modo que una vez llegado a él no se pueda incrementar. Posteriormente cuando se hayan producido algunos decrementos del contador, se puede calcular el número exacto de referencias. Con esto se evita que el valor del contador debido al overflow pueda ser cero. [Ritz03].

El inconveniente principal de los algoritmos basados en contador de referencias es que no detecta ciclos, el contador de referencias de 2 objetos que se referencian mutuamente no será nunca cero, por lo que nunca serán liberados. A continuación se muestra un ejemplo de situación en la que existe un ciclo que no sería nunca liberado.

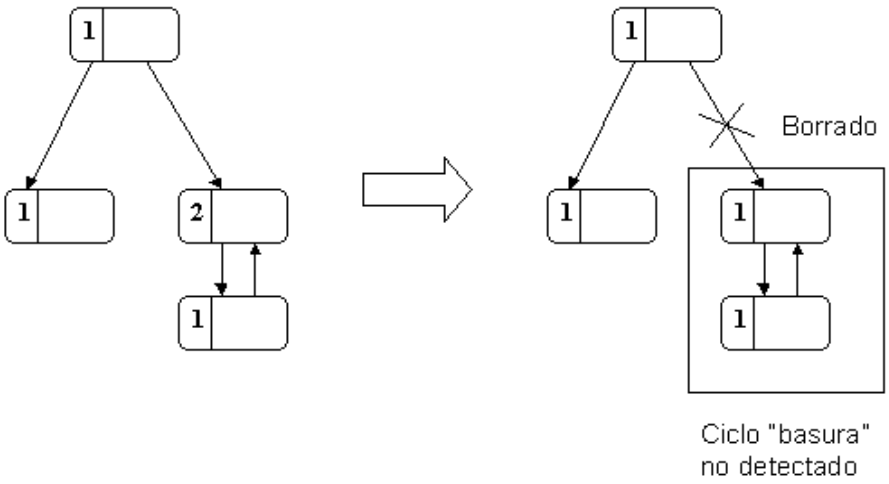



Figura 2.2: Ejemplo de ciclo “basura” no detectado por el algoritmo contador de referencias.

En la actualidad existen modificaciones de estos algoritmos que solucionan este problema. A continuación se muestra un algoritmo basado en contador de referencias que detecta y elimina los ciclos. Antes de explicar los pasos del algoritmo se dan unas definiciones previas.

- Nodo Candidato: Se considera que un nodo es candidato a provocar un ciclo cuando su contador es decrementado a un valor distinto de cero. Es fácil comprobar que este es el único caso en el que un nodo puede provocar ciclos no detectados, ya que si el contador se incrementa no hay problemas de ciclo y si se decrementa a cero tampoco, puesto que será eliminado por el recolector de memoria.
- Subgrafo A: Llamamos subgrafo A al subgrafo cuya raíz es el nodo A.
- Nodo interno al subgrafo A: Nodo que pertenece al subgrafo A.
- Nodo externo al subgrafo A: Nodo que no pertenece al subgrafo A.
- Referencias internas: Asignaciones entre nodos pertenecientes al subgrafo A.
- Referencias externas: Asignación a un nodo interno desde un nodo externo.

El algoritmo consta de tres pasos:

1. Eliminación de referencias internas: En el primer paso se actualizan los contadores de referencias de todos los nodos del subgrafo de manera que no se tengan en cuenta las referencias internas. El contador de referencias de cada nodo del subgrafo indicará el número de referencias externas.
2. Extensión de las referencias externas: Los nodos que tienen el contador de referencias distinto de cero son alcanzables desde el exterior, por lo tanto sus hijos también lo son. Para cada nodo con contador de referencias mayor que 1 que no haya sido visitado ya, se incrementa en uno el contador de sus hijos.
3. Eliminación de ciclos: Los nodos que han quedado con el contador de referencias igual a cero después de los 2 primeros pasos no son alcanzables desde el exterior y son eliminados. [PBK03].

Sin embargo, este algoritmo tiene un problema, complica la idea inicial del algoritmo contador de referencias ya que hay que hacer varios recorridos por el grafo. Por este motivo no se puede ejecutar en pequeños intervalos de tiempo y no podría usarse para sistemas de tiempo real.

## *2.2 Mark and Sweep*

El nombre Mark and Sweep corresponde a las dos fases de las que consta el recolector:

1. Primera fase (Mark): Se recorre el grafo de referencias marcando los objetos por los que se pasando, de esta manera se tienen marcados los objetos alcanzables.
2. Segunda fase (Sweep): Se liberan todos los objetos que no hayan sido marcados en la primera fase.

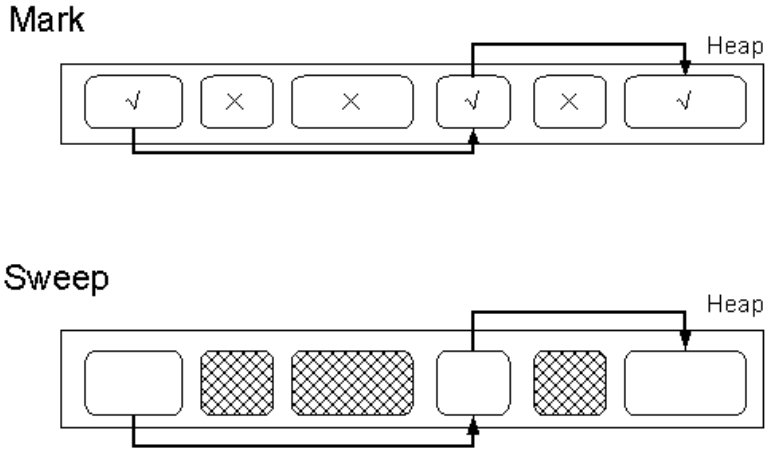



Figura 2.3: Algoritmo Mark and Sweep.

Ventajas:

La ventaja que aporta este tipo de algoritmos respecto a los basados en contador de referencia, es la solución al problema de los ciclos.

Inconvenientes:

No se puede utilizar esta técnica para sistemas en tiempo real, ya que mientras se están ejecutando las 2 fases del algoritmo no se debe modificar el heap. El segundo inconveniente es el alto coste de la recolección, ya que cada objeto es visitado dos veces por el algoritmo, una durante la fase de mark y otra durante la fase de sweep. Un tercer problema es que causa fragmentación de la memoria. Las técnicas basadas en compactación y en copia de objetos solucionan este problema [Ven00].

## *2.3 Algoritmos basados en Compactación*

Se trata de algoritmos parecidos al algoritmo Mark and Sweep, difieren en la segunda fase.

1. Primera fase (Mark): Igual que en el algoritmo de Mark and Sweep.
2. Segunda fase (Compacting): Una vez que han sido identificados los objetos "vivos", es decir, los que no se pueden liberar porque están siendo referenciados, éstos se mueven para conseguir que todos los objetos vivos estén de forma contigua en memoria, el resto de la memoria se considera como un único espacio libre.

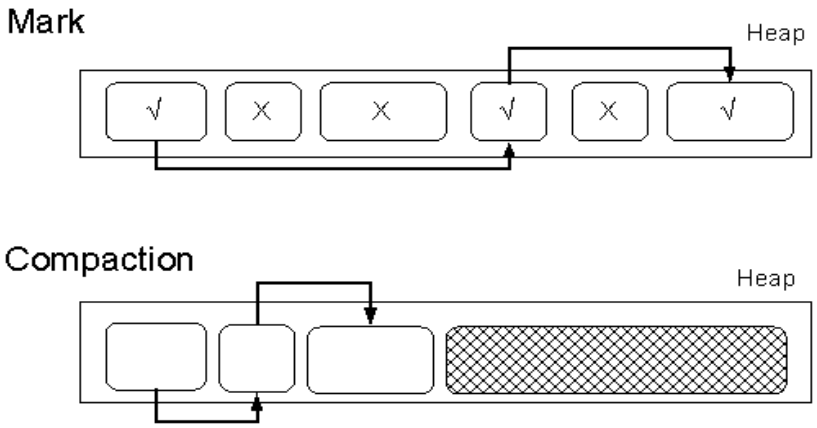



Figura 2.4: Algoritmo basado en compactación.

Ventajas:

La ventaja respecto a Mark and Sweep es que resuelve el problema la fragmentación.

Inconvenientes:

Al igual que los algoritmos de Mark and Sweep, los algoritmos basados en compactación no pueden ser utilizados para sistemas en tiempo real [Ven00].

## 2.4 Algoritmos basados en Copia

El heap se divide en dos espacios de memoria (espacio origen y espacio destino). Se recorre el grafo de referencias y se mueve los objetos vivos por los que se va pasando, del espacio origen al espacio destino. Una vez que el proceso ha terminado, el espacio origen puede ser borrado, ya que todo lo necesario se encuentra en el espacio destino. La siguiente vez que se ejecute el recolector, los espacios intercambian las funciones, el que actuaba como destino para a ser origen y viceversa.

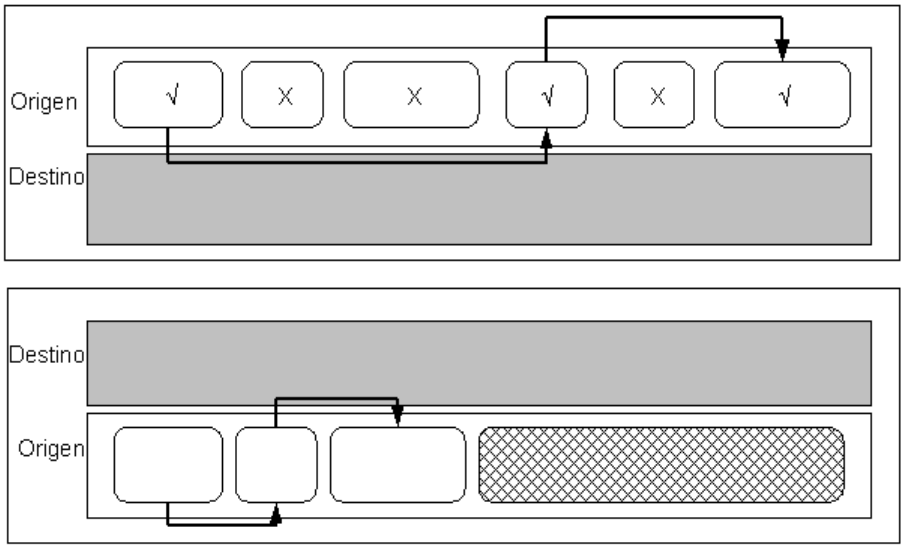



Figura 2.5: Algoritmo basado en copia.

Ventajas:

Resuelve el problema de la fragmentación. Resuelve el problema de visitar dos veces cada objeto que aparecía en los algoritmos de Mark and Sweep, los algoritmos basados en Copia son algoritmos más rápido, ya que sólo se pasa una vez por cada objeto vivo.

Inconvenientes:

Si una ventaja es el ahorro en el tiempo de ejecución, el precio que hay que pagar es un aumento en el tamaño de la memoria, es necesario usar dos espacios de memoria idénticos. Esto hace que para espacios de memoria muy grandes sea impracticable este tipo de algoritmo. No se puede utilizar esta técnica para sistemas en tiempo real ni para sistemas empotrados debido a que duplica las necesidades de memoria [Ven00].

## 2.5 Incrementales

En las técnicas vistas hasta ahora, era necesario detener el proceso que modifica las referencias, al que llamaremos mutador, antes de ejecutar el recolector. En los algoritmos incrementales, el mutador y el recolector se ejecutan simultáneamente. Los algoritmos basados en contador de referencias pueden considerarse como un tipo de

algoritmo incremental, aunque no suele usarse debido a sus muchos inconvenientes. También existen variaciones de cada una de las técnicas no incrementales vistas anteriormente que los transforman en incrementales.

## Algoritmo de los tres colores

Se trata de un algoritmo incremental basado en el coloreado de los objetos. Cada objeto está coloreado de 1 color, el significado del color es el siguiente:

- Negro: El objeto y todos sus hijos han sido visitados por el recolector.
- Gris: El objeto ha sido visitado, pero tiene algún hijo que no lo ha sido.
- Blanco: El objeto no ha sido visitado. Al final del algoritmo, los objetos coloreados de color blanco serán considerados basura y serán liberados.

El algoritmo consta de 4 pasos:

1. Se empieza con todos los objetos coloreados de blanco.
2. Se colorea el nodo raíz de gris.
3. Mientras haya nodos grises, se toma un nodo gris G
    - Se colorean el nodo G de negro
    - Se colorean los hijos del nodo G de gris
4. Al final, se eliminan los objetos coloreados de blanco.

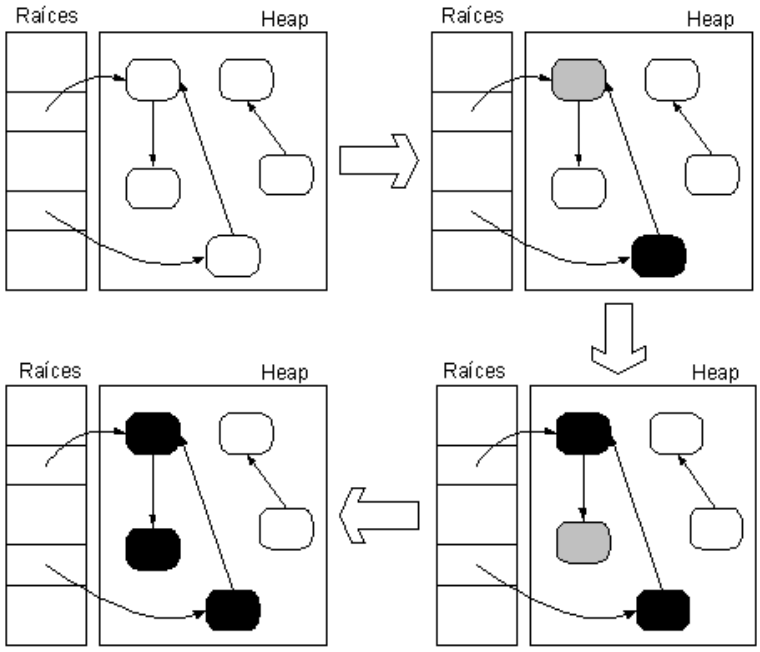



Figura 2.6: Algoritmo Incremental de los 3 colores.

Para que el algoritmo funcione correctamente, debe cumplirse el siguiente invariante:

**¡¡¡No puede haber referencias de objetos negros a objetos blancos!!!**

En los algoritmos incrementales existen 2 procesos en ejecución concurrente por lo que es necesaria una sincronización, esta sincronización se consigue mediante Write Barriers y Read Barriers:

- **Write Barrier**: Mecanismo que protege las operaciones de escritura.
- **Read Barrier**: Mecanismo que protege las operaciones de lectura.

**Ventajas e inconvenientes de algoritmos incrementales:**

Ventajas:

No es necesario que el mutador (es decir, la aplicación) se pare para que el recolector pueda ejecutarse. Este tipo de algoritmos es el usado para las aplicaciones en tiempo real.

Inconvenientes:

Al existir dos procesos ejecutándose a la vez, es necesaria una sincronización entre el mutador y el recolector [Ritz03].

## *2.6 Generacionales*

Este tipo de algoritmos están basados en la observación empírica de la mortalidad infantil de los objetos, muchos objetos mueren al poco de nacer o si no mueren al nacer tienen una vida relativamente larga. Una generación es un conjunto de objetos que fueron creados aproximadamente a la vez. Se clasifican los objetos en generaciones y se ejecuta el recolector con más frecuencia en las generaciones jóvenes que en las adultas.

# 3. SISTEMAS DE TIEMPO REAL

Un sistema en tiempo real es un sistema informático que interacciona repetidamente con su entorno físico y responde a los estímulos que recibe de dicho entorno en un plazo de tiempo determinado. Para que el funcionamiento del sistema sea correcto, no sólo deben producir resultados correctos sino que tienen que realizarlos en un tiempo determinado. Los sistemas de tiempo real suelen estar integrados en un sistema más general en el que se realizan funciones de control y/o monitorización: Sistemas Empotrados. Las necesidades de los programas de tiempo real se pueden solucionar con sistemas operativos en tiempo real, que ofrecen un marco sobre el que construir aplicaciones de programas en tiempo real [BDG05, BUW01].

## *3.1 Clasificación*

### 3.1.1 Sistemas Críticos o Duros

Se trata de sistemas en los que todas las acciones deben ocurrir dentro del plazo de tiempo especificado. La superación de las limitaciones de tiempo podría provocar consecuencias catastróficas. Requieren una validación de que el sistema siempre cumple las restricciones (se deben demostrar que se cumplen). El comportamiento temporal está determinado por el entorno. Ejemplos de este tipo de sistemas son: control de vuelo, control de procesos industriales, control de un reactor nuclear y gestión del motor de un coche.

### 3.1.2 Sistemas No Críticos o Blandos

Sistemas con restricciones de tiempo menos rigurosas en los que la superación de las limitaciones de tiempo no acarrea graves consecuencias. El comportamiento temporal está determinado por el computador. Se permite un comportamiento en sobrecarga degradado. Ejemplos de este tipo de sistemas son: planes de vuelo, transacciones on line, multimedia y chat IRC.

## *3.2 Características*

### 3.2.1 Determinismo temporal

Se debe comprobar que todas las tareas terminan dentro de su plazo. Para poder planificar el sistema es necesario que el comportamiento del sistema sea predecible. Existen tres tipos de tareas según su esquema de activación:

- **Periódicas**: Tareas que se ejecutan regularmente, con un período conocido.
- **Aperiódicas**: Tareas que se ejecutan de forma irregular en respuesta a un evento del entorno o del sistema.
- **Esporádica**: Tareas que se ejecutan en instantes de tiempo aleatorios.

### 3.2.2 Interacción con dispositivos físicos

Los mecanismos de entrada-salida son dependientes del dispositivo. Los manejadores de dispositivos forman parte del software de la aplicación, no están bajo el control del sistema operativo.

### 3.2.3 Complejidad

Aunque los sistemas de tiempo real pueden llegar a tener millones de líneas de código, la variedad de sus funciones hace que sean complejos incluso los sistemas pequeños. Por otra parte, dados los continuos cambios del mundo real, los sistemas están continuamente sometidos a modificaciones y mejoras.

### 3.2.4 Concurrencia

Los dispositivos físicos funcionan al mismo tiempo por lo que los componentes software que los controlan se ejecutan concurrentemente. Se necesita suficiente velocidad de proceso para simular paralelismo con un solo procesador. Los programas concurrentes son poco claros y difíciles de depurar.

### 3.2.5 Seguridad y fiabilidad

El sistema debe seguir en funcionamiento en todo momento a pesar de errores de software o de hardware, proporcionando una buena calidad de servicio, ya que una degradación del servicio en un sistema crítico puede provocar pérdidas de vidas humanas, pérdidas económicas, daños medioambientales... Si lo hace, el sistema debe de fallar de manera que cuando ocurra el fallo, se preserve la mayor parte de los datos y capacidades del sistema en la máxima medida posible. El interfaz debe diseñarse de forma que minimice la posibilidad de un error humano.

### 3.2.6 Eficiencia

Los sistemas deben ser eficientes en:

- **Implementación**: debido a los requisitos temporales.
- **Capacidad**: el espacio físico debe ser reducido.
- **Costes económicos**: Viabilidad del producto final.

## *3.3 Recolector de memoria en Sistemas de tiempo real*

Como se ha visto, existen varios tipos de sistemas de tiempo real según su criticidad, atendiendo a esta clasificación se estudiará la conveniencia del uso de algoritmos recolectores de memoria en este tipo de sistemas. Para los sistemas de tiempo real no críticos, en los que una parada en la ejecución de los procesos del sistema no tuviera graves consecuencias, se acepta el uso de algoritmos recolectores de memoria con detenciones del sistema predecibles, es decir, medidas y previstas con antelación. Por otro lado, la ejecución de un recolector de memoria sobre el sistema, implica que los procesos de dicho sistema deben detener su ejecución mientras el recolector se esté ejecutando. Por esta razón no se pueden utilizar algoritmos de recolección de memoria sobre este tipo de sistemas. La solución pasa por encontrar recolectores de memoria que puedan ejecutarse sin que el sistema esté detenido, este tipo de recolectores son los algoritmos de recolección de memoria incrementales, que se estudiaron en el apartado 1.5. Los algoritmos de recolección incrementales pueden usarse sin problemas en sistemas de tiempo real.

En cambio en los sistemas de tiempo real críticos o duros, no ocurre lo mismo, ya que si el sistema falla y/o detiene su ejecución, las consecuencias pueden llegar a

ser catastróficas. Una de las características de este tipo de sistemas es que no se debe detener su ejecución bajo ningún concepto, deben estar en funcionamiento en todo momento. En resumen podemos afirmar que los sistemas de tiempo real no críticos toleran paradas por el recolector de memoria, mientras que los críticos no deben verse afectados por el mismo. Este proyecto se encuadra en los sistemas de tiempo real no críticos.

# 4. REFERENCIAS EXTERNAS AL HEAP

En este apartado se plantean los problemas originados por la inclusión de referencias externas al heap provenientes de regiones. Empezamos definiendo qué es una región, el concepto de región es similar al de heap con la diferencia que las regiones no están provistas de algoritmos de recolección de memoria, esto es debido a que en ellas se ejecutan tareas críticas, cuya ejecución no puede ser detenida en ningún momento.

Sin embargo se permite que la región entera se libere cuando deja de ser usada por el programa. En este caso la región se comporta como un objeto y se podría ejecutar cualquiera de los recolectores de memoria conocidos tomando como nodos las regiones en vez de los objetos de un heap.

Hasta ahora hemos visto que los objetos del heap podían estar referenciados desde otros objetos del mismo heap o desde la pila de referencias, existe también otro tipo de referencias, son las referencias externas al heap provenientes de las regiones. No se permiten sin embargo referencias desde el heap hacia las regiones.

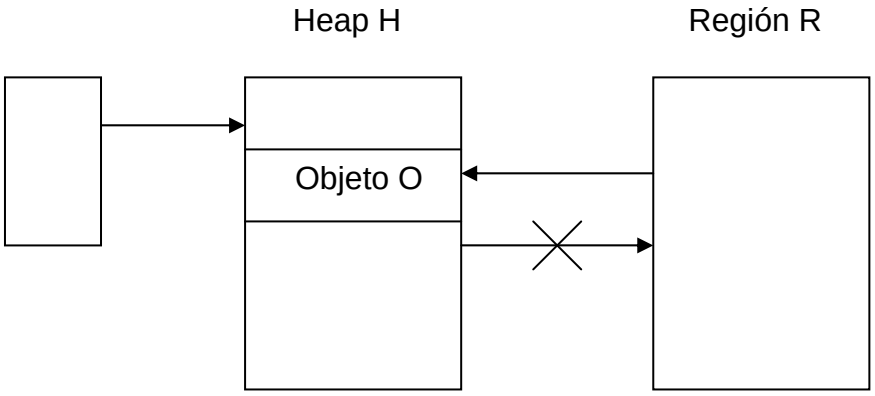



Figura 4.1: Esquema de referencias posibles entre el heap y la región.

## *4.1 Ejemplo de situación problemática*

Cuando una región se libera, las referencias que iban desde la región que va a ser liberada al heap son modificadas, esta situación puede originar problemas de "basura flotante" no recolectada. Imaginemos una situación inicial en la que un objeto O del heap H es referenciado desde la región R. Si en un momento determinado la región R es liberada: ¿Qué pasará entonces con las referencias desde R hasta H? ¿Qué sucederá con O si únicamente era referenciado desde R?

Puede suceder que a pesar de que R haya sido ya liberada, O no tenga conocimiento de este hecho, por lo que seguirá creyendo que existe una referencia externa hacia él. Dicho objeto no será liberado cuando se ejecute el recolector de memoria y permanecerá en el heap como "basura flotante".

## *4.2 Solución propuesta*

Una posible solución al problema es crear una tabla en la que ir guardando información sobre las referencias externas. La tabla se actualiza cuando se crea una nueva referencia, cuando se modifica o elimina una referencia ya existente y cuando se libera una región. Al añadir esta nueva estructura de datos, es necesario que cuando el recolector de memoria visita un objeto, se consulte la tabla de referencias externas para saber si dicho objeto tiene referencias externas o no. Para el correcto mantenimiento de la tabla hay que tener en cuenta que no se puede acceder a la tabla si está siendo modificada ya que esto podría producir acceso a datos erróneos. Como ya se ha explicado en otro apartado de esta memoria, este tipo de problemas se solucionan añadiendo una Write Barrier.

Con la solución propuesta se soluciona el problema expuesto, pero a cambio se produce una sobrecarga del recolector de memoria, al que se le da más trabajo del que tenía anteriormente.

# 5. DISEÑO DEL SISTEMA

Hasta ahora se ha visto el tipo de aplicación se quería construir y los pasos seguidos hasta conseguirlo. Pasamos a continuación a detallar el diseño del sistema.

## *5.1 Motivación del diseño*

Como cualquier sistema informático, se pretendía que el sistema realizado fuera *reutilizable y escalable*, de forma que soportara futuras ampliaciones y modificaciones. Sin embargo esto no siempre se ha conseguido debido a los errores de diseño que hemos detectado en la aplicación de partida. En la aplicación de partida existía un único paquete al que pertenecían todas las clases. La aplicación de partida estaba programada pensando en clases y no pensando en interfases.
Otras características del diseño de nuestro sistema son:

- Intenta garantizar la existencia de una única instancia de algunas clases (*Patrón Singleton*) [Pre01, Som01].
- En lugar de utilizar varias copias de objetos, estos objetos se comparten, de esta forma se mantiene la coherencia en todo el sistema. (*Patrón Peso Ligero*) [Pre01, Som01].

## *5.2 Diseño del Sistema*

El cambio principal en el diseño fue la agrupación de clases con funcionalidades relacionadas en paquetes. A continuación se explica en profundidad cada paquete y sus principales clases. Los nombres de los métodos y de las clases se muestran otro tipo de letra, para que destaquen y no polucionen el resto del texto. Al final del capítulo se muestra el diagrama de clases del proyecto.

### a) Paquete principal

En este paquete se encuentra la clase *HeapOfFish*, que es la que comienza la ejecución del applet, también se incluyen en este paquete las clases *HeapOfFishCanvases* y *HeapOfFishControlPanel* que contienen los paneles principales sobre los que se establecen los demás componentes.

### b) Paquete heap

En este paquete se incluyen los objetos relacionados con el manejo del heap, tanto la estructura propia del heap como los objetos que se crearán en él. La clase principal es *GCHeap* que implementa la estructura de datos sobre la que se crearán los objetos. Esta clase contiene también las operaciones de acceso y modificación del heap. Además se incluyen en este paquete la clase que implementan los objetos que se crean en el heap: *ObjectHandle,* y las de los distintos tipos de peces. Estas clases son: *BigRedFishIcon, MediumBlueFishIcon y LittleYellowFishIcon.*

### c) Paquete creación de objetos

A este paquete pertenecen las clases que intervienen en el proceso de creación de objetos sobre el heap. La clase principal de este paquete es: *AllocateFishPanel*. Esta clase contiene los métodos para crear peces de cada uno de los 3 tipos (rojos, azules y amarillos). También se incluye en este paquete la clase *PecesIniciales*, se trata de una nueva creada por nosotros que permite la creación de nuevos objetos de forma automática sin necesidad de pulsar ningún botón.

### d) Paquete creación de relaciones

En este paquete se recogen las clases que intervienen en el proceso de creación y borrado de relaciones entre objetos. Algunas clases de este paquete son: *AssignReferencesCanvas, AssignReferencesPanel, LinkFishCanvas, UnlinkFishCanvas y MoveFishCanvas*. La clase *AssignReferencesCanvas* permite elegir la acción a realizar entre las tres acciones posibles en esta pantalla (crear relación, deshacer relación y mover pez de posición). La clase *LinkFishCanvas* contiene los métodos para crear relaciones entre dos objetos. La clase *UnlinkFishCanvas* contiene los métodos para deshacer relaciones creadas. La clase *MoveFishCanvas* contiene los métodos para mover un pez de posición. También se incluye en este paquete la clase *ReferenciasIniciales*, se trata de una nueva clase creada por nosotros que realiza la creación de relaciones entre objetos de forma automática sin necesidad de pulsar ningún botón.

### e) Paquete recolección de memoria

A este paquete pertenecen las clases que intervienen en el proceso de recolección de memoria. Algunas clases de este paquete son: *GarbageCollectCanvas y GarbageCollectPanel*. La clase *GarbageCollectCanvas* es una de las clases más importantes del proyecto ya que es aquí donde se encuentra el método *nextGCStep()*, que implementa el algoritmo de recolección de memoria, y los métodos que implementan los recorridos sobre el grafo de relaciones, por ejemplo *traverseNextFishNodeProfundidad() y traverseNextFishNodeAnchura()*, este último método para el recorrido en anchura fue añadido por nosotros. También se incluyen en este paquete las la clase *Lector*, una nueva clase creada por nosotros que implementa un lector que accede al grafo de relaciones mientras el recolector de memoria se está ejecutando.

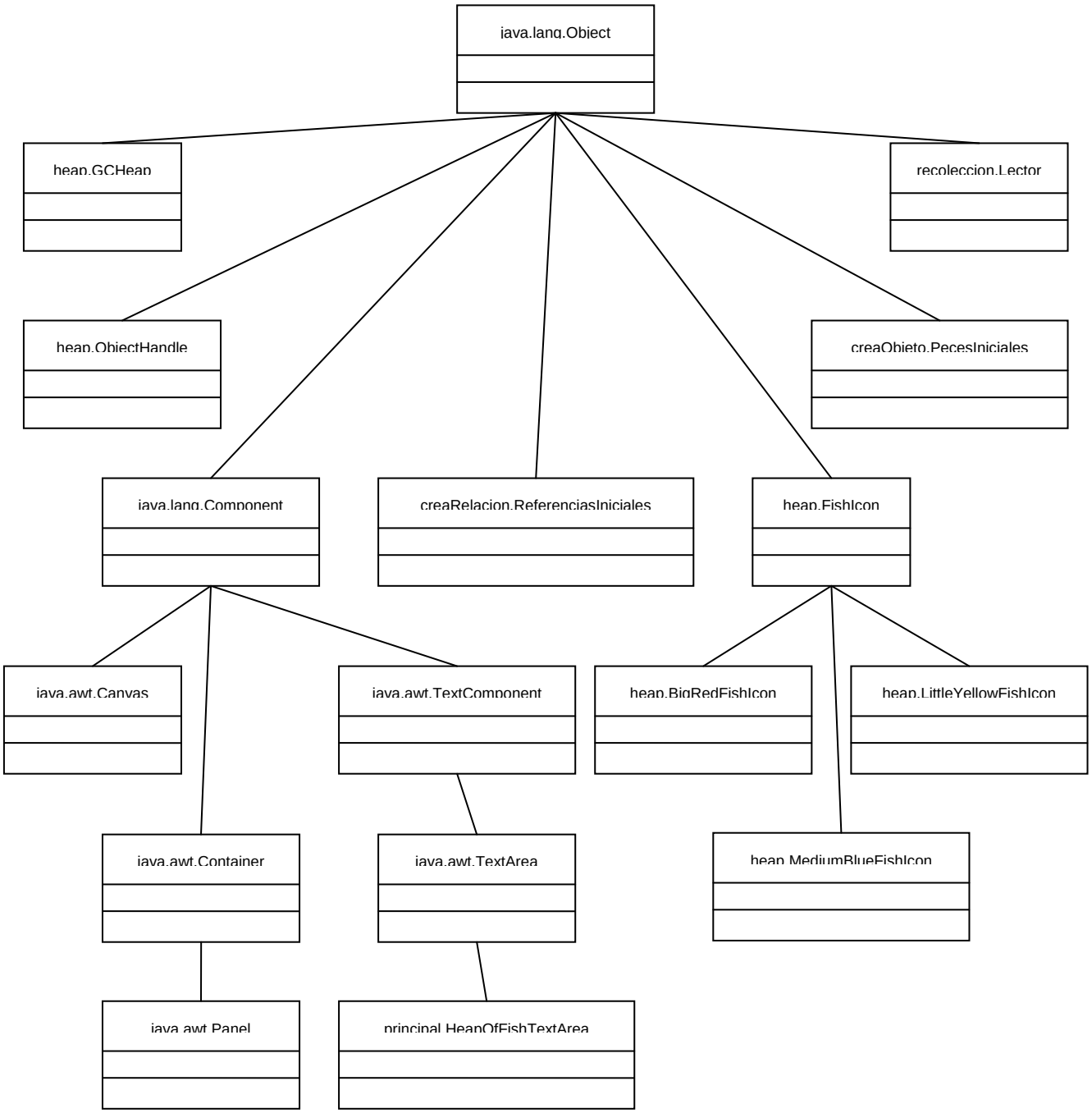
### f) Paquete compactación del heap

En este paquete se incluyen las clases que intervienen en el proceso de compactación del la memoria. Algunas clases de este paquete son: *CompactHeapCanvas y CompactHeapPanel*.

**Jerarquía de clases:**

- clase java.lang.Object
  - clase java.awt.Component (interface)
    - clase java.awt.Canvas (interface)
      - clase creaRelacion.AssignReferencesCanvas
        - clase creaRelacion.LinkFishCanvas
        - clase creaRelacion.MoveFishCanvas
        - clase creaRelacion.UnlinkFishCanvas
      - clase creaObjeto.BlueFishButtonCanvas
      - clase compactacion.CompactHeapCanvas
      - clase recoleccion.GarbageCollectCanvas (interface)
      - clase heap.PoolsCanvas
      - clase creaObjeto.RedFishButtonCanvas
      - clase principal.SwimmingFishCanvas (interface)
      - clase creaObjeto.YellowFishButtonCanvas
    - clase java.awt.Container
      - clase java.awt.Panel (interface)
        - clase creaObjeto.AllocateFishButtonPanel
        - clase creaObjeto.AllocateFishPanel (interface)
        - clase java.applet.Applet
          - clase principal.HeapOfFish
        - clase creaRelacion.AssignReferencesCanvases
        - clase creaRelacion.AssignReferencesCheckboxPanel
        - clase creaRelacion.AssignReferencesPanel (interface)
        - clase creaObjeto.BlueFishButtonPanel
        - clase principal.ColoredLabel
        - clase compactacion.CompactHeapButtonPanel
        - clase compactacion.CompactHeapPanel
        - clase recoleccion.GarbageCollectButtonPanel
        - clase recoleccion.GarbageCollectCheckboxGroup
        - clase recoleccion.GarbageCollectPanel
        - clase principal.HeapOfFishCanvases
        - clase principal.HeapOfFishControlPanel
        - clase principal.HeapOfFishModeCheckboxPanel
        - clase creaObjeto.RedFishButtonPanel
        - clase creaObjeto.YellowFishButtonPanel
    - clase java.awt.TextComponent (interface)
      - clase java.awt.TextArea
        - clase principal.HeapOfFishTextArea
  - clase heap.FishIcon
    - clase heap.BigRedFishIcon
    - clase heap.LittleYellowFishIcon
    - clase heap.MediumBlueFishIcon
  - clase heap.GCHeap
  - clase principal.HeapOfFishStrings
  - clase recoleccion.Lector (interface)
  - clase heap.LocalVariables
  - clase heap.ObjectHandle
  - clase heap.Objeto_Indice
  - clase creaObjeto.PecesIniciales
  - clase creaRelacion.ReferenciasIniciales (interface)

**Diagrama de clases:**



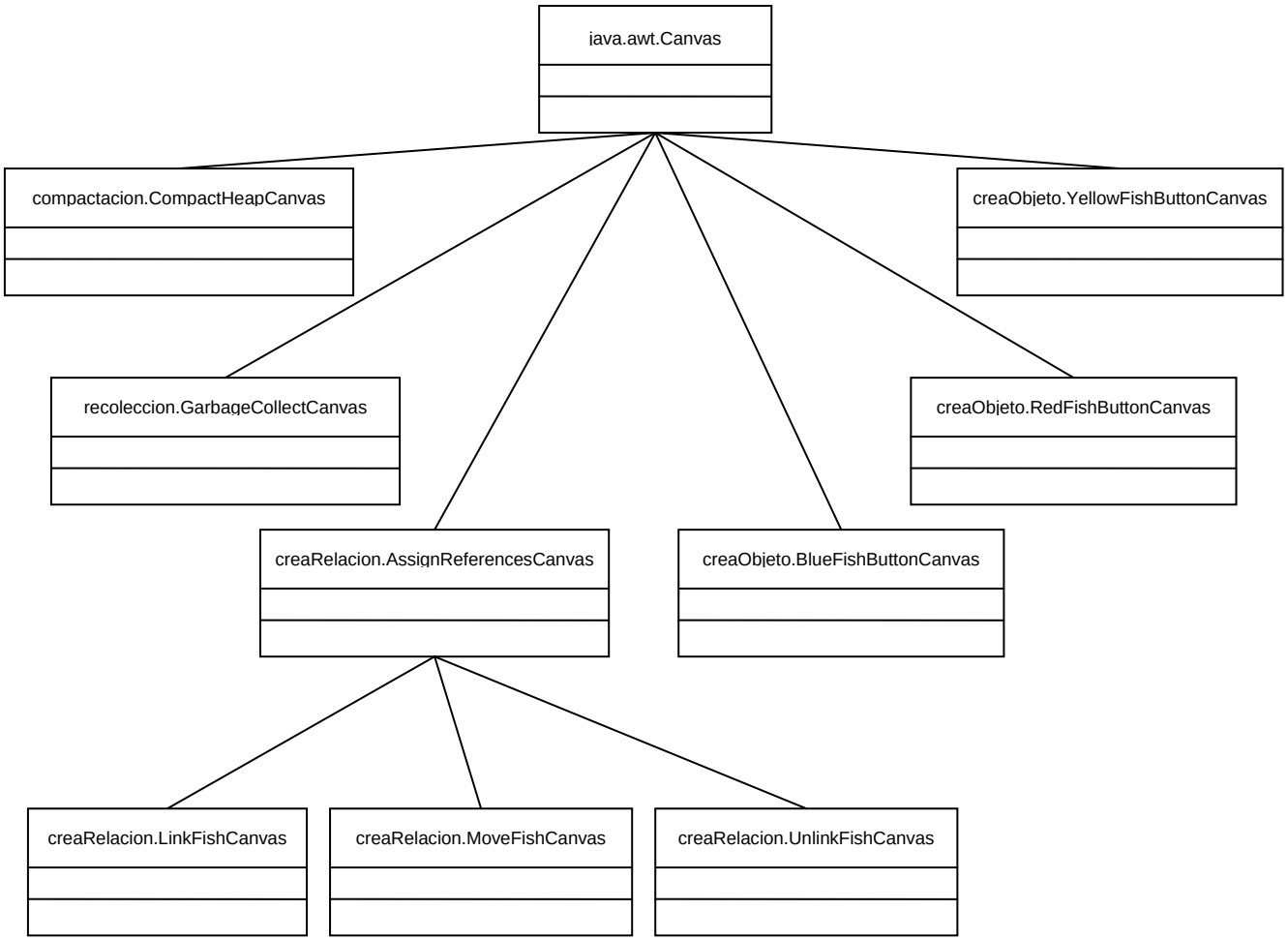

java.awt.Canvas
compactacion.CompactHeapCanvas
creaObjeto.YellowFishButtonCanvas
recoleccion.GarbageCollectCanvas
creaObjeto.RedFishButtonCanvas
creaRelacion.AssignReferencesCanvas
creaObjeto.BlueFishButtonCanvas
creaRelacion.LinkFishCanvas
creaRelacion.MoveFishCanvas
creaRelacion.UnlinkFishCanvas


Figura 5.1: Diagrama de clases.

# 6. IMPLEMENTACIÓN

En el apartado correspondiente a la implementación se describe en detalle el algoritmo recolector de memoria de partida y el recolector de memoria incremental realizado por nosotros. Se muestra un pseudocódigo y un ejemplo para un caso concreto para cada uno de ellos. También se detallan los transformaciones necesarias para transformar el algoritmo de partida en incremental. En la parte final del capítulo se indican las nuevas funcionalidades aportadas por nosotros al sistema y que no tienen que ver directamente con el algoritmo incremental.

## *6.1 Algoritmo de recolección de memoria de partida*

A continuación se detalla el algoritmo de recolección de memoria de partida. En el apartado 2 se dio una explicación sobre los principales tipos de recolectores de memoria existentes, el algoritmo usado en la aplicación de partida para la recolección de memoria es un algoritmo parecido al **"Algoritmo de los 3 colores"** visto en el apartado 2.5 "Algoritmos de recolección de memoria incrementales", pero a diferencia de este, el utilizado por la aplicación no era incremental, es decir, mientras se ejecutaba el recolector no se podía modificar el heap. La manera en la que este algoritmo recorre el grafo de relaciones de peces es la siguiente, se realiza un recorrido en profundidad empezando por la raíz amarilla, a continuación se sigue por la azul y por último se visita la roja. El orden de visita de los hijos de cada pez es como a continuación se detalla, primero pasa por las relaciones de Amistad, a continuación por las de Comida y por último pasa por las relaciones de Aperitivo.

La implementación del algoritmo se realiza en la clase *GarbageCollectCanvas,* y consiste en lo siguiente, el algoritmo va pasando por una serie de estados dependiendo del nodo que esté visitando en ese momento. Los posibles estados del algoritmo son: *detenido, visitando raíz amarilla, visitando peces desde raíz amarilla, visitando raíz azul, visitando peces desde raíz azul, visitando raíz roja, visitando peces desde raíz roja y borrando peces no alcanzados*. Dependiendo del estado se realizarán una serie de acciones diferentes. El método que implementa el algoritmo es el *nextGCStep()* de la clase *GarbageCollectCanvas* y cuyo funcionamiento describe el siguiente pseudocódigo:

Si el algoritmo está en estado:

**- *detenido***: el algoritmo no ha comenzado. Todos los peces están coloreados de blanco.
Pasar al estado *visitando raíz amarilla.*

**- *visitando raíz amarilla***: el algoritmo marca como visitada la raíz amarilla y pasa al estado *visitando peces desde raíz amarilla.*

**- *visitando peces desde raíz amarilla***: se recorren todos los peces del árbol cuya raíz es la raíz amarilla, cuando ha terminado de recorrer todos los nodos pasa al estado *visitando raíz azul.*

**- *visitando raíz azul:*** el algoritmo marca como visitada la raíz azul y pasa al estado *visitando peces desde raíz azul.*

**- *visitando peces desde raíz azul***: se recorren todos los peces del árbol cuya raíz es la raíz azul, cuando ha terminado de recorrer todos los nodos pasa al estado *visitando raíz roja.*

- ***visitando raíz roja**:* el algoritmo marca como visitada la raíz roja y pasa al estado *visitando peces desde raíz roja.*

- ***visitando peces desde raíz roja**:* se recorren todos los peces del árbol cuya raíz es la raíz roja, cuando ha terminado de recorrer todos los nodos pasa al estado *borrando peces no alcanzados.*

- ***borrando peces no alcanzados:*** se borran todos los peces que no han sido alcanzados, es decir los coloreados como blancos y se pasa al estado detenido.

Se debe aclarar qué entendemos por "recorrer un árbol cuya raíz es el nodo A", cuando en el pseudocódigo se dice que se "recorre un árbol cuya raíz es el nodo A", quiere decir que se visitan los nodos del árbol con raíz en A que no han sido ya visitados, coloreando cada nodo según corresponda, es decir, se colorea el nodo a visitar de color gris y cuando todos sus hijos se han visitado, es decir, están coloreados de gris, dicho nodo se colorea de negro. Este recorrido se realiza en profundidad. Otra de las características del algoritmo es que no se ejecuta de forma continua, sino que requería ir pulsando un botón, cada vez que se pulsaba el botón se visitaba un nuevo pez.

## *6.2 Implementación del algoritmo incremental*

Una vez que conocemos el algoritmo de partida y conocemos los distintos tipos de algoritmos de recolección de memoria podemos afirmar que nuestro algoritmo de recolección de memoria está basado en el **"Algoritmo de los 3 colores incremental"**. Una de las características que distingue nuestro algoritmo del algoritmo de partida es que nuestro algoritmo permite el recorrido tanto en profundidad como en anchura. El orden de visita de las raíces es el mismo de la aplicación de partida, primero se visita la raíz amarilla, luego la azul y por último la roja. Tampoco se modifica el orden de visita de los hijos de cada pez, primero pasa por las relaciones de Amistad, a continuación por las de Comida y por último pasa por las relaciones de Aperitivo. Además nuestro algoritmo se ejecuta de forma continua sin necesidad de tener que ir pulsando un botón. Pero la principal aportación de nuestro algoritmo y el motivo de este proyecto es que al mismo tiempo que se ejecuta el recolector de memoria se pueden realizar modificaciones en el heap (crear nuevos objetos, crear relaciones entre objetos…), es decir, se ejecuta concurrentemente con la aplicación.

A continuación se explican los pasos seguidos para realizar las modificaciones necesarias para pasar del algoritmo de partida al algoritmo incremental. El primer paso que se dio para convertir el algoritmo en un algoritmo incremental fue añadir una nueva estructura de datos, en la que se almacenaran los objetos que son destino de las relaciones creadas una vez que ha empezado la ejecución del algoritmo recolector y que tienen como nodo origen un nodo ya visitado por dicho recolector, es decir, un nodo coloreado de negro o de gris. Una vez terminado el recorrido del grafo, que se hace de la misma manera que lo hace el algoritmo de partida, nuestro algoritmo incremental pasaría a continuación a recorrer la nueva estructura, visitando así las nuevas relaciones creadas. Se modificó el método *nextGCStep()* de la clase *GarbageCollectCanvas* para añadir el recorrido de la nueva estructura como un estado más. A continuación se muestra el pseudocódigo del nuevo algoritmo, que mantiene los primeros estados igual que el algoritmo de partida, la diferencia está en el nuevo estado *pasando por nuevos nodos.*

Si el algoritmo está en estado:

**...**

***- pasando por nuevos nodos:*** se recorren los nodos que se han ido añadiendo a la nueva estructura de datos y se pasa al estado *borrando peces no alcanzados.*

***...***

La nueva estructura de datos consiste un nuevo atributo llamado *listaNuevosObjetos*, de tipo vector, que se añade a la clase *GCHeap*. Pero no es suficiente con ir añadiendo nodos a esta estructura, se debe controlar en todo momento que al crearse las nuevas relaciones se cumpla el invariante: **No puede haber nodos negros apuntando a nodos blancos**. Si el invariante no se cumple, el nodo blanco se pinta de color gris. Para el manejo de la nueva estructura se modificó la clase *LinkFishCanvas*, que implementa la creación de relaciones, se añadieron las modificaciones necesarias en los métodos que crean relaciones para que cuando se crea una relación desde un nodo negro o gris una vez que el recolector está en ejecución, se añada el nodo destino de la relación a *listaNuevosObjetos*. Además, como ya se ha dicho, se debe tener en cuenta el problema del invariante.

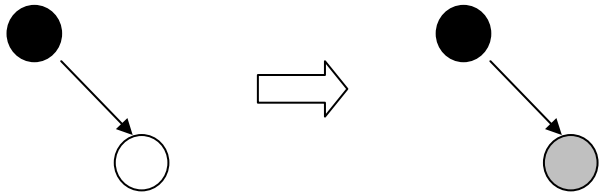


Figura 6.1: Solución al problema del incumplimiento del invariante.

A continuación se muestra un ejemplo en el que se explica la importancia de esta nueva estructura. Imaginemos que el recolector de memoria se ejecuta sobre un grafo sencillo de sólo dos nodos a los que llamaremos Nodo1 y Nodo2, además Nodo2 es hijo de Nodo1. En un momento de la ejecución del algoritmo de recolección, éste se encuentra visitando Nodo2 tras haber visitado anteriormente Nodo1. Si en este momento se añadiera una relación desde Nodo1 hasta otro nodo nuevo, al que llamaremos Nodo3, el algoritmo debe visitar Nodo3, ya que es un nodo alcanzable. La implementación de nuestro algoritmo almacenaría Nodo3 en la nueva estructura *listaNuevosObjetos* y posteriormente recorrería los nodos existentes en dicha estructura de datos.

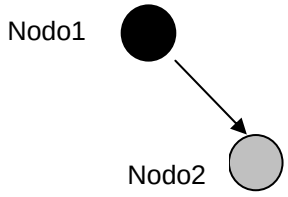



Figura 6.2: Momento en el que el recolector visita Nodo2

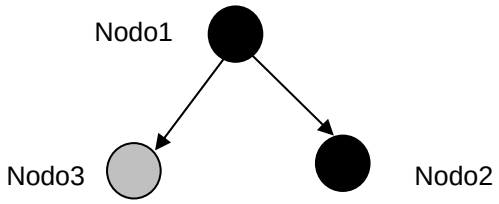



Figura 6.3: Se crea nueva relación de Nodo1 a Nodo3, Nodo3 se añade ala estructura de nuevas relaciones

En este punto teníamos una aplicación en la que, se podían añadir nuevas referencias a la vez que el algoritmo recolector se ejecutaba, se podía considerar que habíamos conseguido un simulador de un algoritmo recolector incremental, pero queríamos que fuera lo más parecido posible a un recolector incremental real. El siguiente paso consistió en sustituir la forma de ejecución del algoritmo. Hasta ahora la ejecución del algoritmo no se realizaba de forma continua, sino exigía pulsar un botón, cada vez que se pulsaba dicho botón, el algoritmo recolector visitaba un nuevo objeto.
A partir de ahora el algoritmo se ejecutaría de forma continua al pulsar un botón "*Comienzo*". También hay posibilidad de detener el algoritmo pulsando otro botón "*Pausa*". Pero además su ejecución se realizaría de la forma más independiente posible de los métodos que creaban relaciones. Se crearon dos hilos que se ejecutarían a la vez, en uno de los hilos se ejecutaría el recolector de forma continua, y en el otro se ejecutaría la creación de referencias. Para implementar los hilos se hizo mediante la implementación de la interfaz *Runnable* por parte de *AssignReferencesPanel* y *GarbageCollectCanvas*, las clases encargadas de la asignación de referencias y del recolector de memoria respectivamente. En ambas clases se añade un atributo de tipo *thread* y se rellena el método *run(),* que es el método que se ejecuta cuando comienza el hilo. Para lograr la ejecución continua del algoritmo, se modifica la llamada al método *nextGCStep()* de la clase *GarbageCollectCanvas*. Hasta ahora se hacía una llamada a dicho método cada vez que se pulsa el botón "*Paso*", a partir de ahora, se añade un bucle al método *run()* y se llama a *nextGCStep()* desde dentro de ese bucle. Estos dos hilos comparten variables, como por ejemplo el grafo de relaciones, por lo que es necesario una sincronización entre ambos, ésta se consigue mediante las **Write Barriers**, que se explicaron en el apartado 2.5 "Algoritmos de Recolección de Memoria Incrementales". La implementación de nuestras **Write Barriers** se realiza mediante monitores, una funcionalidad de los threads de Java que consiste en declarar todos los métodos que acceden a la variable compartida como sincronizados (*synchronized*). De esta forma se evita que un método declarado como sincronizado pueda acceder a una variable que está siendo modificada por otro método sincronizado.

En cuanto al aspecto visual, en la clase *GarbageCollectPanel*, se suprime el botón "*Paso*", que era el que había que ir pulsando para hacer avanzar el recolector y se añaden dos nuevos botones, un botón "*Comienzo*", que ejecuta el algoritmo de forma continua y un el botón "*Pausa*", cuya función es detener el algoritmo si está en ejecución, y reanudar el algoritmo si está detenido.

Para completar el algoritmo de recolección lo siguiente que se hizo fue añadir un nuevo tipo de recorrido al algoritmo recolector, además del usado hasta ahora que consistía en un recorrido en profundidad, se añadió un recorrido en anchura. Los dos recorridos podrían usarse posteriormente para realizar estudios comparativos. A continuación se muestra un ejemplo de orden en el que se visitan los nodos de un grafo en ambos recorridos:

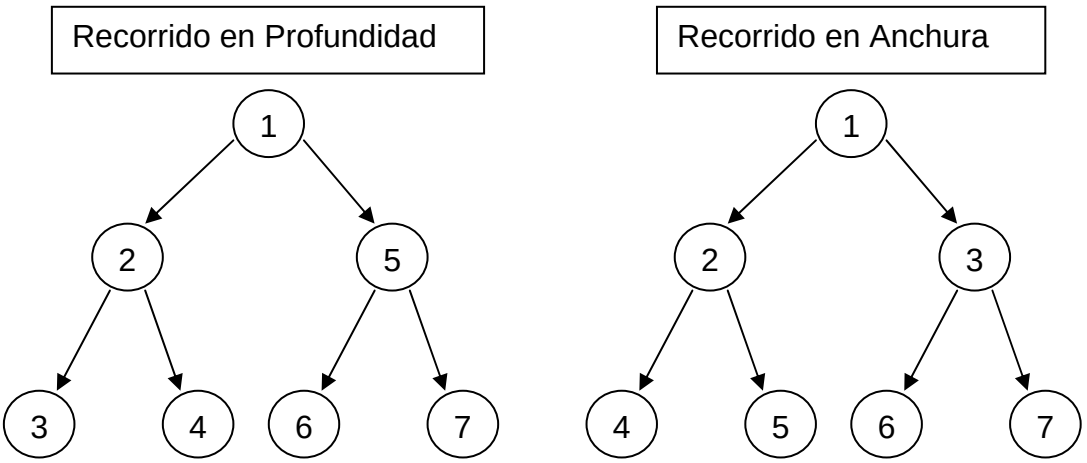



Figura 6.4: Orden en que se visitan los nodos de un grafo en los recorridos en profundidad y anchura.

El recorrido en anchura lo realiza un nuevo método *traverseNextFishNodeAnchura(ObjectHandle oh),* que se añadió a la clase *GarbageCollectCanvas*, cuya función es recorrer el grafo en anchura a partir del objeto *oh*. Para implementar el recorrido se usa un nuevo vector llamado *listaPendientes*. El método devuelve verdadero cuando ha visitado todos los hijos de *oh* y falso en caso contrario. A continuación se muestra el pseudocódigo del recorrido en anchura.

- Si *oh* tiene una relación de Amistad, se colorea de gris el pez destino de dicha relación y se añade a *listaPendientes*.
- Si *oh* tiene una relación de Comida, se colorea de gris el pez destino de dicha relación y se añade a *listaPendientes*.
- Si *oh* tiene una relación de Aperitivo, se colorea de gris el pez destino de dicha relación y se añade a *listaPendientes*.
- Una vez que ha visitado todos los hijos, colorea oh de negro, calcula cual es el siguiente objeto a visitar (el siguiente a *oh* en *listaPendiente*) y devuelve verdadero.

En lo relativo a la parte gráfica, es necesario disponer de un mecanismo para poder elegir el tipo de recorrido deseado, la elección se realiza mediante *checkbox*. Se añade una nueva clase llamada *GarbageCheckBoxGroup*, que contiene los dos *checkbox*.

**Ejemplo de ejecución**

Para comprender mejor el algoritmo incremental se muestra a continuación la ejecución de nuestro algoritmo para un ejemplo sencillo y un recorrido en anchura, relacionándolo con el pseudocódigo del algoritmo recolector incremental visto anteriormente. Se parte del siguiente grafo de relaciones inicial, en el que hemos dado nombre a los peces para ayudar en las explicaciones.

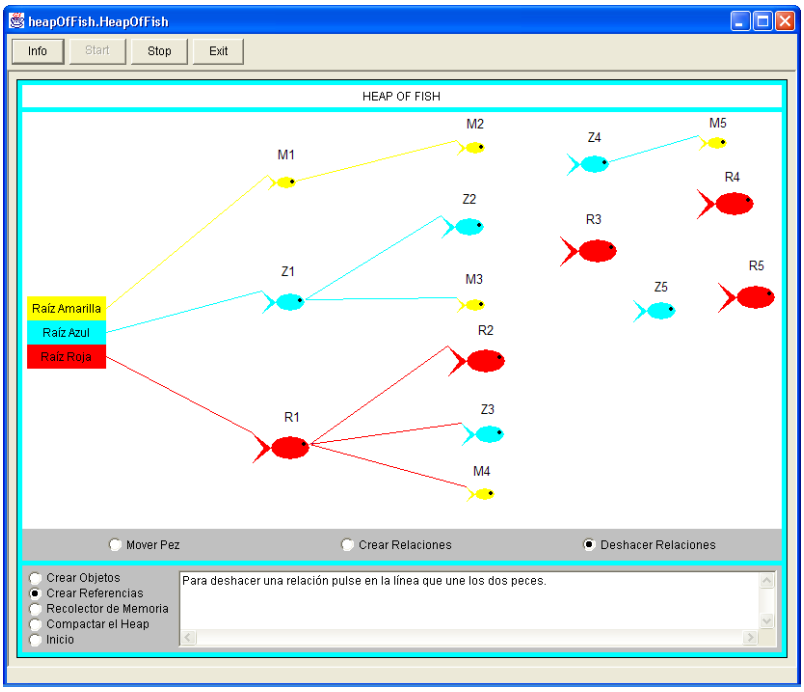



Figura 6.5: Situación inicial del grafo de relaciones.

Inicialmente todos los peces comienzan coloreados de blanco (estado *detenido*). El algoritmo comienza visitando la raíz amarilla, la marca como visitada y colorea de gris su hijo *M1* (estado *visitando raíz amarilla*). En el siguiente paso visita *M1* (estado *visitando peces desde raíz amarilla*), colorea los hijos de *M1* de gris (el hijo de *M1* es *M2*) y colorea *M1* de negro. A continuación visita *M2*, como no tiene hijos, lo colorea de negro y con esto termina el recorrido de los peces que cuelgan de la raíz amarilla y empieza el recorrido de los peces que cuelgan de la raíz azul. Visita la raíz azul (estado *visitando raíz azul*), la marca como visitada y colorea *Z1* de gris. A continuación visita *Z1* (estado *visitando peces desde raíz azul*) y colorea de gris *Z2* y *M3* y de negro *Z1*. Visita *Z2*, como no tiene hijos lo colorea de negro y pasa a visitar a su hermano *M3*. Como *M3* tampoco tiene hijos lo colorea de negro y termina el recorrido de los peces que cuelgan de la raíz azul. En este momento del recorrido se añade una nueva relación desde *Z2* a *Z4*. *Z2* estaba coloreado de negro puesto que ya había sido visitado y *Z4* está coloreado de blanco. Esta nueva relación incumple el invariante que no permite que haya relaciones desde nodos negros a nodos blancos, para solucionarlo, se colorea *Z4* de gris. Además se añade *Z4* y su hijo *M5* a *listaNuevosObjetos*, para ser recorridos posteriormente. Al mismo tiempo que se añadía la nueva relación el recolector continúa ejecutándose, visita la raíz roja (estado *visitando raíz roja*), la marca como visitada y colorea *R1* de gris. En el siguiente paso visita *R1* (estado *visitando peces desde raíz amarilla*), colorea de gris *R2*, *Z3*, y *M4* y de negro *R1*. Visita *R2* y lo colorea de negro. Visita *Z3* y lo colorea de negro. Visita *M4* y lo colorea de negro. Con esto termina el recorrido de los peces que cuelgan de la raíz roja, pero no ha terminado la ejecución del recolector ya que a continuación recorre *listaNuevosObjetos* para visitar las relaciones que se han creado cuando el recolector ya había comenzado a ejecutarse (estado *pasando por nuevos nodos*). En el recorrido de las nuevas relaciones, se visita *Z4* (recordemos que estaba coloreado de gris para hacer cumplir el invariante), colorea su hijo *M5* de gris y *Z4* de negro. Visita el siguiente nodo en *listaNuevosObjetos* que es *Z4*, lo colorea de negro y termina el recorrido de los nuevos objetos. En el siguiente paso se eliminan los nodos que han quedado coloreados de blanco, ya que no son alcanzables desde las raíces (estado *borrando peces no alcanzados*). En el ejemplo los peces que borra son *R3*, *R4*, *R5* y *Z5*. El grafo resultante tras la ejecución del algoritmo recolector es el siguiente.

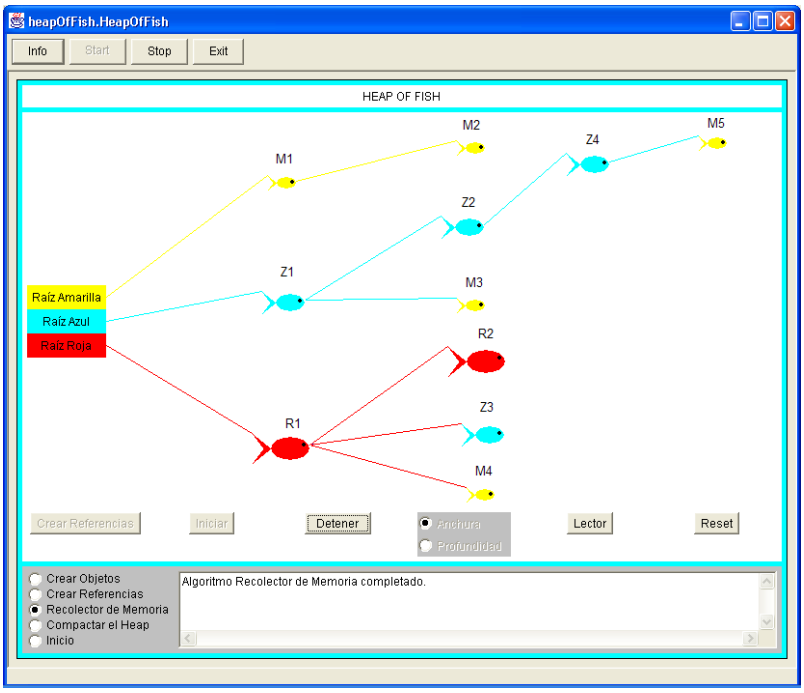



Figura 6.6: Situación final del grafo de relaciones.

## *6.3 Otras funcionalidades añadidas*

En este apartado se muestran nuevas funcionalidades desarrolladas por nosotros que no tienen que ver directamente con el algoritmo de recolección de memoria, pero que se consideraron de interés para estudiar alguna de las características de los recolectores de memoria reales.

### 6.3.1 Problema de los lectores y escritores

Cuando el recolector de memoria de un heap está en ejecución, puede ocurrir que otro programa necesite acceder a una variable contenida en dicho heap. En este apartado se explica la sincronización existente entre el programa accesor y el algoritmo recolector, conocido como el **problema de los lectores y escritores**. En los recolectores de memoria actuales, la sincronización entre el lector y el escritor se realiza mediante **Read Barriers**, esto implica que ningún programa puede acceder a los datos de un heap mientras sobre dicho heap se está ejecutando el recolector de memoria. Si algún programa intentara este acceso, se produciría un error de acceso a memoria. Esto supone un grave problema si el programa accesor se trata de una tarea crítica. Como ya sabemos, la idea de tarea crítica es una tarea que no puede detener su ejecución bajo ningún concepto. Sin embargo, por lo explicado anteriormente, si sobre un heap se está ejecutando el recolector de memoria y la tarea crítica necesita acceder a una variable de dicho heap, debe esperar a que el recolector finalice.

Se pretende estudiar si es posible sustituir las **Read Barriers** por **Write Barriers**, de forma que se permita la lectura de objetos del heap aunque esté en ejecución su algoritmo recolector, siempre y cuando no se esté modificando el objeto al que se quiera acceder. Aunque la mejora propuesta fuera viable, continuaría sin poderse realizar escrituras de objetos del heap si se está ejecutando el recolector y en caso de intentarlo, se produciría un error de escritura. A pesar de que el programa

accesor continuaría deteniéndose al intentar leer un objeto que está siendo modificado por el recolector y que sigue sin poderse realizar escrituras, la mejora propuesta es un avance, puesto que permite realizar la mayoría de las lecturas. Mediante nuestra aplicación se ha intentado demostrar que la mejora propuesta es factible y se ha visto que sí lo es. Para este estudio se añadió a nuestra aplicación una nueva clase llamada *Lector*, que contiene un método *lector()* cuya función es realizar un recorrido en anchura sobre el grafo de relaciones al mismo tiempo que se ejecuta el recolector de memoria. Para facilitar la demostración, el lector marcará cada nodo al que accede, la marca que se usa es colorear el ojo de los peces de color verde. Podría pensarse que colorear el ojo es una escritura, pero no se considera como tal porque el color del ojo es una variable que no afecta al recolector. Pero sí que es una forma visual de demostrar que se ha accedido al pez.

Si se ejecuta el recolector de memoria y a continuación se ejecuta el *Lector*, se puede comprobar que a la vez que se ejecuta el recolector de memoria, los ojos de los peces por los que va pasando el *Lector*, se van coloreando de verde, esto demuestra que la lectura y la recolección se ejecutan simultáneamente sin problemas. La clase *Lector* implementa el interfaz *Runnable*, para que el método *lector()* se ejecute sobre otro hilo independiente. Es necesaria una sincronización con el algoritmo recolector, esto se consigue con las **Write Barriers** que se han implementado declarando como sincronizados los métodos que acceden a la variable compartida, en este caso el grafo de relaciones. En lo relativo a la parte gráfica, se añade un nuevo botón llamado "*Lector*" al panel de la clase *GarbageCollectPanel*, al pulsar dicho botón empieza la ejecución del hilo lector.

### 6.3.2 Creación automática de relaciones

El último paso en la implementación de nuestra aplicación consistió en añadir la posibilidad de que las relaciones entre los peces fueran creadas de forma automática. Se pensó que añadir todas las referencias a mano podía ser una tarea incómoda, por lo que se decidió que el usuario pudiera crear referencias de forma automática pulsando un botón. Estas relaciones creadas no son elegidas por el usuario directamente, el usuario elige entre una batería de ejemplos predefinidos. La creación manual y automática no son incompatibles, es decir pueden crearse sobre el mismo grafo relaciones de las dos maneras posibles.
Con esta nueva opción se han conseguido 3 objetivos:

- **Realismo:** Al no intervenir el usuario en el programa que modifica el grafo de relaciones se da una situación más real, en la que 2 programas, en este caso 2 hilos, interactúan.
- **Facilidad para las pruebas:** Si se quiere ejecutar varias veces el recolector sobre el mismo grafo no es necesario crear todas las referencias a mano.
- **Sincronización:** Se comprueba la correcta sincronización entre mutador y accesor gracias al correcto funcionamiento de las **Write Barriers**.

Para conseguirlo, se añadió una nueva clase *ReferenciasIniciales*, que se ejecuta sobre un hilo independiente, para ello implementa la interfaz *Runnable* y contiene los métodos encargados de crear las referencias de forma automática. Estos métodos deben asegurar que en todo momento se cumple el invariante de los algoritmos incrementales: **No puede haber relaciones desde un objeto negro hacia uno blanco**. En esta clase hay métodos para tratar *listaNuevosObjetos* parecidos a los de la clase *LinkFishCanvas*, mencionados en el apartado 6.3. Otros métodos importantes de la clase, son:

- *void creaReferencia(int destino, int origen):* Crea una referencia desde el objeto cuya posición en el heap es “origen” hasta el objeto situado en la posición “destino”.
- *void creaReferenciaDeRaiz(int origen, int destino):* Crea una referencia desde una raíz al objeto cuya posición en el heap es “destino”. La codificación de las raíces es la siguiente: si origen es 1, se crea referencia desde la raíz amarilla, si origen es 2, se crea desde la raíz azul, por último, si origen es 3, se crea desde la raíz roja.

Para poder realizar asignaciones de forma automática es necesario que existan objetos que relacionar, para ello se creó una nueva clase *PecesIniciales*, que contiene los métodos necesarios para crear nuevos peces de cada uno de los 3 tipos posibles (rojos, azules y amarillos) y los añade al heap, esta creación de peces se hace de forma masiva, es decir se crean todos los peces a la vez, al contrario que las asignaciones, que se hacen de una en una. Los métodos principales de esta clase son:

- *void nuevoPezRojo():* Crea un nuevo pez rojo y lo añade al heap, tiene el mismo efecto que si pulsamos el botón “CrearPezRojo” desde la pantalla de Creación de Objetos.
- *void nuevoPezAzul():* Crea un nuevo pez azul y lo añade al heap, tiene el mismo efecto que si pulsamos el botón “CrearPezAzul” desde la pantalla de Creación de Objetos.
- *void nuevoPezAmarillo():* Crea un nuevo pez rojo y lo añade al heap, tiene el mismo efecto que si pulsamos el botón “CrearPezAmarillo” desde la pantalla de Creación de Objetos.

En el aspecto gráfico, se añade al panel de la clase *GarbageCollectPanel* un botón llamado “*CrearReferencias*”, al pulsar el nuevo botón se muestra un mensaje de diálogo, desde el que se puede elegir el grafo de relaciones que se desea crear automáticamente de entre uno de los ejemplos predefinidos, una vez elegido el grafo, se ejecuta el hilo que crea las relaciones.

# 7. MANUAL DE USUARIO

## *7.1 Requisitos del sistema*

Nuestro proyecto es un Applet de Java. Como todos los Applet de Java, para ejecutarlo es necesario un navegador que sea capaz de reconocerlo y ejecutarlo. Netscape y Microsoft Internet Explorer son dos exploradores con los que se pueden ejecutar los Applets de Java. Además los Applets de Java se ejecutan en la denominada "Máquina Virtual de Java", por lo que otro requisito es disponer de una Máquina Virtual de Java. Puede conseguir todos los requisitos del sistema en los siguientes enlaces:

Netscape:
http://www.netscape.com/

Microsoft Internet Explorer:
http://www.microsoft.com/

Máquina virtual de java:
http://java.sun.com/

## *7.2 Instalación*

El sistema no necesita instalación. Para ejecutar el sistema se ejecutará el archivo HeapOfFish.html que abre una venta del navegador en la que se encuentra el Applet de nuestra aplicación.

## *7.3 Uso de la aplicación*

### 7.3.1 Pantalla principal

Al ejecutar la aplicación se accede a la pantalla principal, desde la que se puede acceder a las otras cuatro:

1. Creación de objetos.
2. Asignación de referencias.
3. Recolección de memoria.
4. Compactación del heap.

Para navegar entre las distintas pantallas, se debe pulsar con el ratón en el checkbox correspondiente a la pantalla deseada. El checkbox se encuentra en la esquina inferior izquierda de la pantalla. A su derecha hay un cuadro de texto en el que se mostrarán mensajes de ayuda.

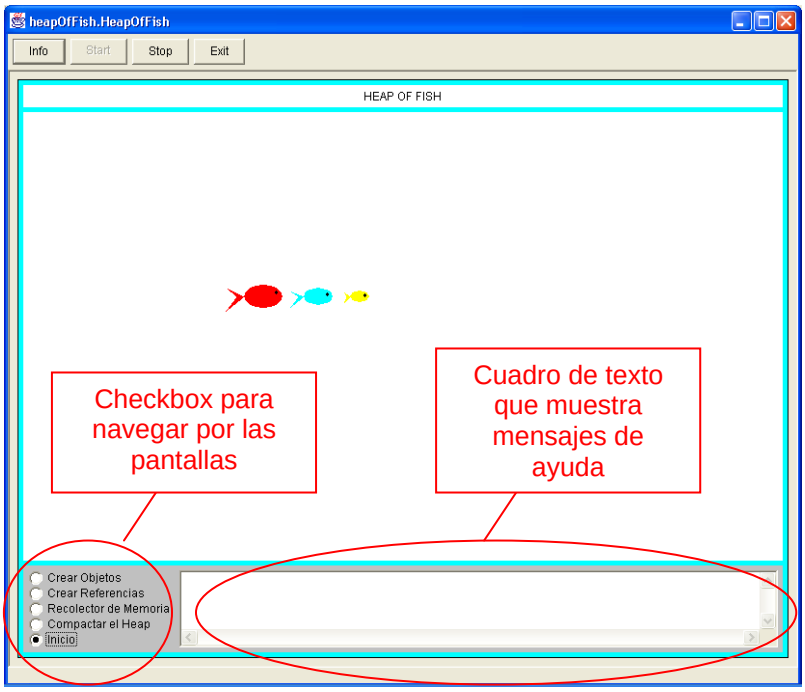



Figura 7.1: Pantalla Inicial.

### 7.3.1 Pantalla de creación de objetos

Desde esta pantalla se crean los peces en el heap. Como ya sabemos se pueden crear peces de 3 tipos: Peces Rojos, Peces Azules y Peces Amarillos.

- Para crear un Pez Rojo, se debe pulsar con el ratón en el botón “CrearPezRojo”.
- Para crear un Pez Azul, se debe pulsar con el ratón en el botón “CrearPezAzul”.
- Para crear un Pez Amarillo, se debe pulsar con el ratón en el botón “CrearPezAmarillo”.

A medida que los objetos se van creando, van apareciendo en el heap, también se muestra el manejador de objetos. En el manejador, todos los objetos tienen el mismo tamaño, en el heap no es así, los peces rojos son los objetos más grandes, ocupan 3 posiciones, los peces azules ocupan 2 posiciones y por último los peces amarillos ocupan una posición. Esta diferencia de tamaño es debido al número de referencias que cada objeto puede tener. Una vez que el heap está lleno no se pueden crear nuevos objetos.

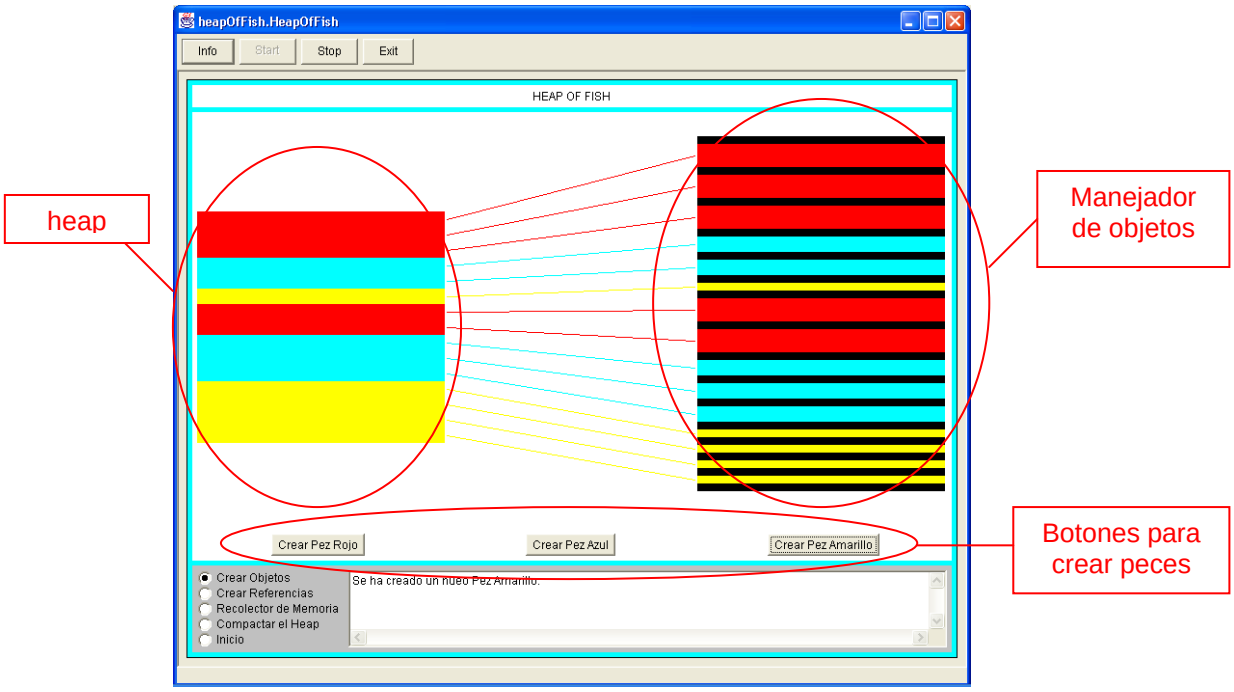



Figura 7.2: Pantalla de Creación de Objetos.

### 7.3.3 Pantalla de Asignación de Referencias

Desde esta pantalla se pueden realizar 3 acciones:

1. Mover Pez
2. Crear relación
3. Deshacer relación

Para elegir la acción a realizar se debe pulsar con el ratón en el checkbox correspondiente a la acción deseada. Este checkbox se encuentra encima del cuadro de texto de ayuda.

- **Mover Peces**

Pasos a seguir para mover un pez de posición:

- Pulsar el checkbox correspondiente a la acción “Mover Pez”.
- Pulsar y mantener pulsado encima del pez que se desea mover y mantener pulsado
- Mover el cursor del ratón hasta la nueva posición deseada.
- Dejar de pulsar el botón del ratón.

- **Crear Asignación**

Antes de ver cómo se crean relaciones, se muestran los diferentes tipos de relaciones existentes. Existen tres tipos de relaciones entre los peces: Amistad, Comida y Aperitivo, cada pez puede tener una relación de Amistad con otro pez del mismo color, un pez rojo puede tener una relación de Comida con un pez azul, a su vez un pez azul puede tener la misma relación de Comida con un pez amarillo. La relación de Aperitivo sólo existe entre peces rojos y amarillos. Los tipos de relación existentes se resumen en las siguientes figuras:

| Origen | Destino | Tipo de Relación |
|---|---|---|
| Rojo | Rojo | Amistad |
| Rojo | Azul | Comida |
| Rojo | Amarillo | Aperitivo |
| Azul | Azul | Amistad |
| Azul | Amarillo | Comida |
| Amarillo | Amarillo | Amistad |

Figura 7.3: Tabla de los Tipos de Relaciones entre los Peces.

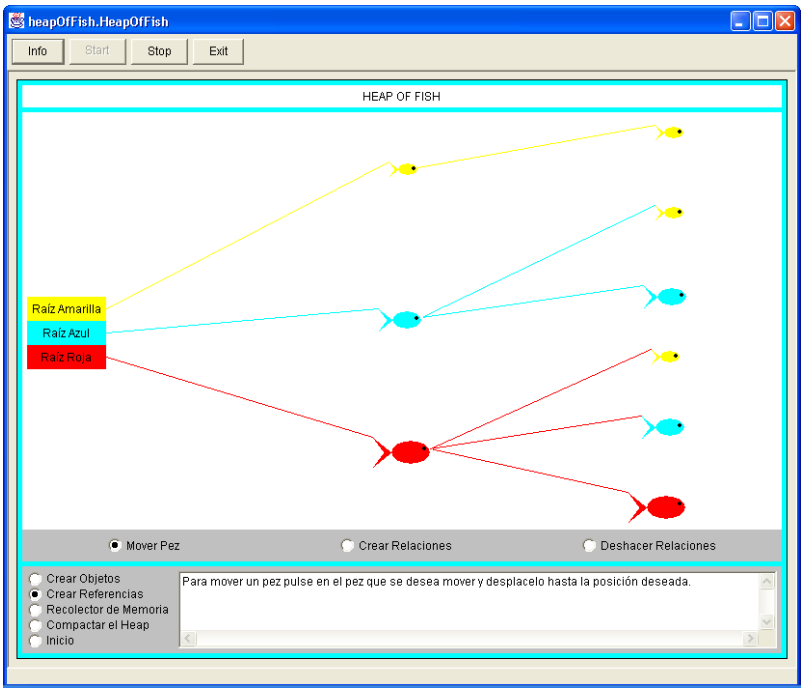



Figura 7.4: Tipos de Relaciones entre los Peces.

Además de los peces, existen tres raíces una correspondiente a cada tipo de pez, se pueden crear asignaciones desde cada raíz a objetos que sean de su mismo color. Ahora sí, vemos los pasos a seguir para crear una relación entre dos peces o entre una raíz y un pez:

- Pulsar el checkbox correspondiente a la acción “Crear Relación”.
- Pulsar y mantener pulsado encima del pez o raíz origen de la relación.
- Mover el cursor del ratón hasta el pez destino de la relación.
- Dejar de pulsar el botón del ratón. Una vez que se ha creado la relación se dibuja una línea que une el pez o la raíz origen con el pez destino.

- **Deshacer Asignación**

Pasos a seguir para deshacer una relación entre dos peces o entre una raíz y un pez:

- Pulsar el checkbox correspondiente a la acción “Deshacer Relación”.
- Pulsar en la línea que representa la relación. Una vez que se ha deshecho la relación se borra la línea.

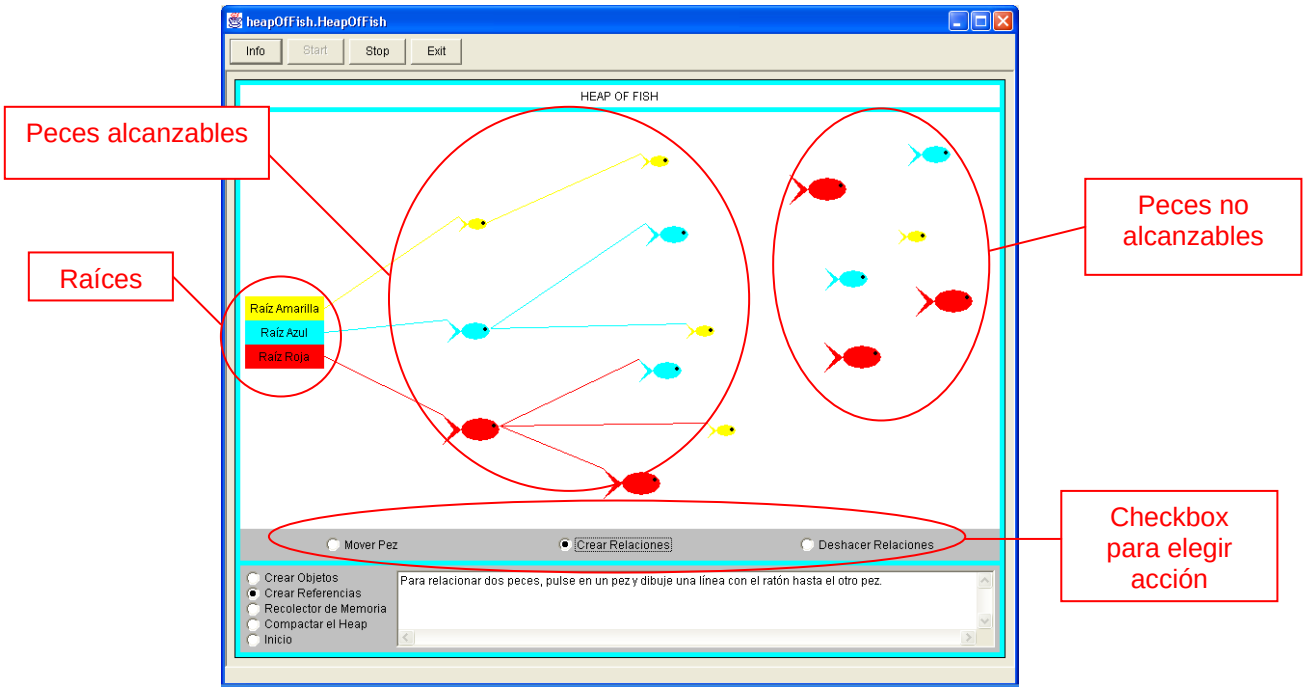



Figura 7.5: Pantalla de Asignación de Referencias.

### 7.3.4 Pantalla de Recolección de Memoria

Esta es la pantalla desde la que se ejecuta el recolector de memoria, la creación automática de relaciones y el lector que recorre el grafo a la vez que el recolector.

- **Creación automática de relaciones**

Para comenzar la creación automática de relaciones, pulsar el botón "Crear Referencias". Al pulsar dicho botón aparece un cuadro de diálogo para elegir el ejemplo de grafo a crear entre una batería de ejemplos predefinida. Una vez elegido el ejemplo, comienzan a crearse relaciones de forma automática. Se debe tener en cuenta que la creación automática de relaciones se debe hacer antes de la ejecución del recolector, ya que durante la ejecución del recolector el botón "Crear Referencias" se mantiene desactivado.

- **Ejecución del recolector de memoria**

Para ejecutar el algoritmo de recolección de memoria, en primer lugar se debe elegir el tipo de recorrido que se desea realizar sobre el grafo de relaciones. Para ello se debe pulsar el checkbox correspondiente. Si no se elige ningún recorrido, por defecto se realizará un recorrido en anchura. A continuación pulsar el botón "Iniciar".

- **Ejecución del lector**

Para ejecutar el lector, pulsar el botón "Lector". Se debe tener en cuenta que para poder ejecutar el lector debe estar en ejecución el recolector de memoria, ya que antes del comienzo de la ejecución del recolector, el botón "Lector" se mantiene desactivado.

- **Botones detener y reset**

En cualquier momento se puede detener la ejecución del recolector, de la creación automática de relaciones y del lector pulsando el botón "Detener". Al pulsar dicho botón, pasa a denominarse "Reanudar". Para volver a la ejecución del recolector, de la creación automática de relaciones y del lector en el punto donde se encontraba, pulsar

el botón “Reanudar”. Cuando se pulsa dicho botón vuelve a denominarse “Detener”. Una vez que ha terminado la ejecución del recolector de memoria, se debe preparar el grafo de relaciones para una nueva ejecución. Para resetear el grafo, pulsar el botón “Reset”. Al pulsar dicho botón, se vuelve a colorear todos los peces de blanco y se eliminan las relaciones existentes de la ejecución anterior.

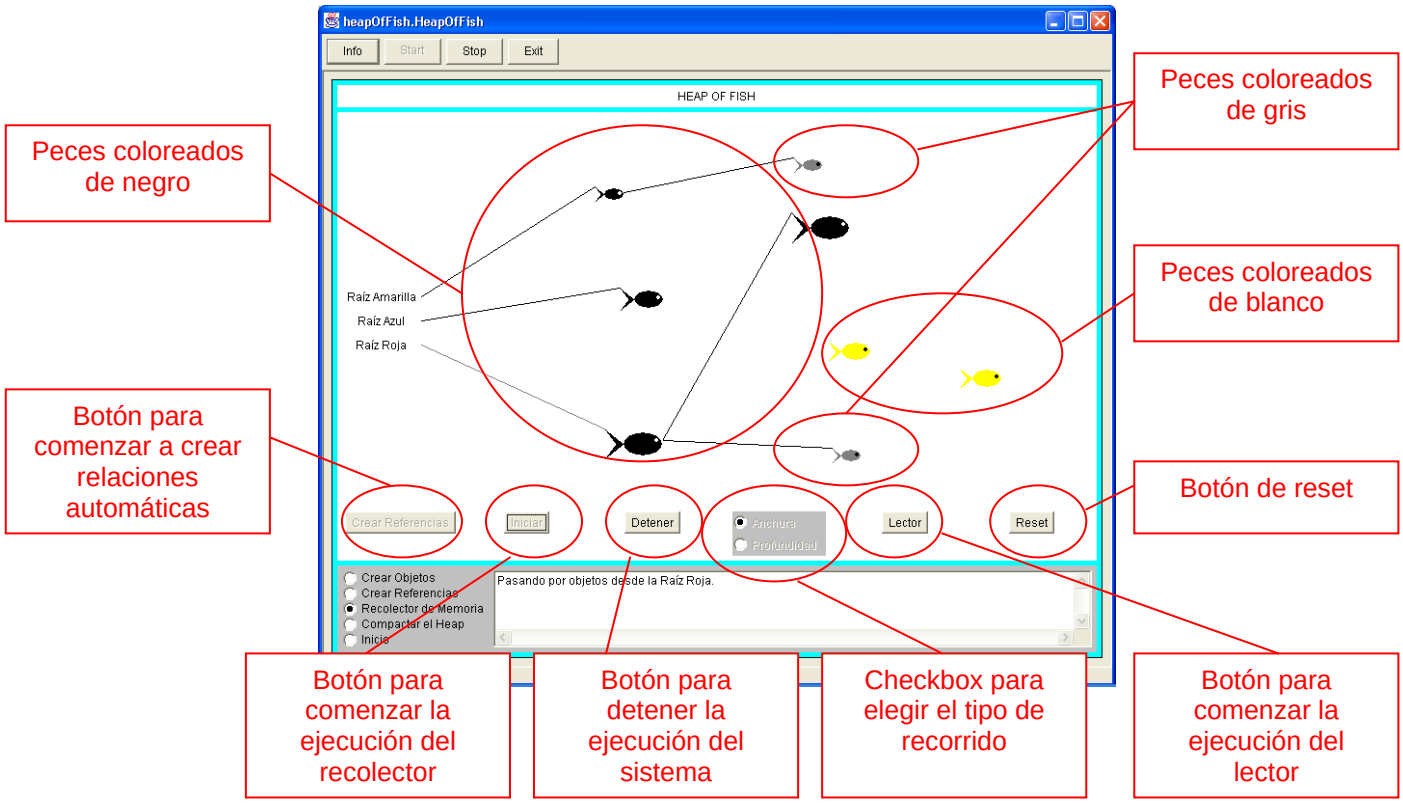



Figura 7.6: Pantalla de Recolección de Memoria.

### 7.3.5 Pantalla de Compactación del heap

Debido a que existen objetos de diferentes tamaños y para evitar el fenómeno de la fragmentación de memoria, existe la posibilidad de compactar el heap desde esta pantalla. Para compactar el heap se debe ir pulsando con el ratón en el botón “Compactar”.

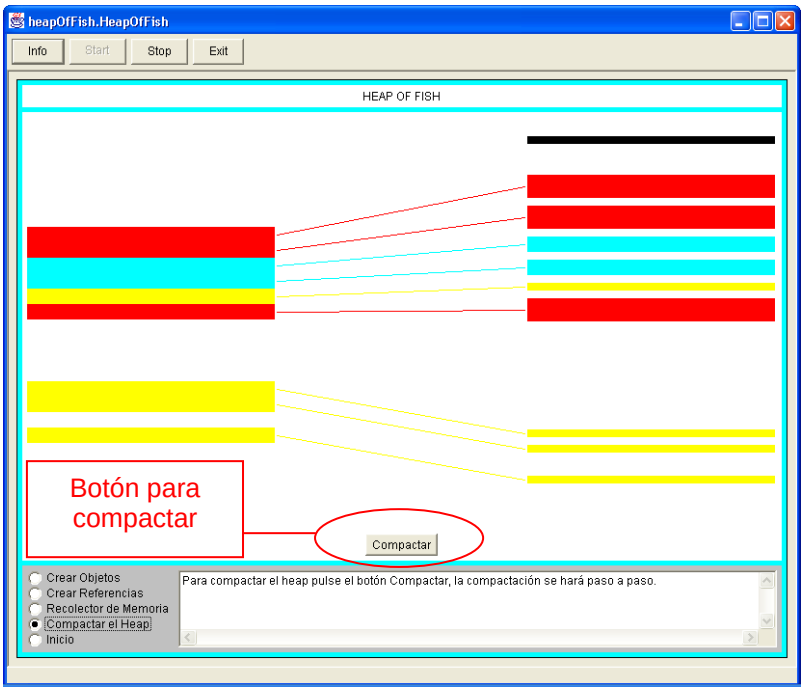



Figura 7.7: Pantalla de Compactación del Heap.

# 8. PRUEBAS REALIZADAS

Para comprobar el correcto funcionamiento del sistema, es necesario realizar un conjunto de pruebas. Se realizaron dos tipos de pruebas, pruebas unitarias y pruebas de bloque. A continuación se detalla cada una de ellas.

## *8.1 Pruebas unitarias*

Las pruebas unitarias son una forma de comprobar el correcto funcionamiento de cada funcionalidad del sistema de forma independiente. [Pre01, Som01]. Estas pruebas las realiza el programador. En nuestro proyecto, para cada funcionalidad añadida se realizaban las pruebas unitarias correspondientes. Al ser muchas las pruebas unitarias realizadas, no se añaden a este documento, ya que lo harían muy extenso.

## *8.2 Pruebas de bloque*

Las pruebas de bloque también llamadas pruebas de integración son una forma de probar el correcto funcionamiento de varias componentes del sistema. [Pre01, Som01]. En este apartado se realizarán también las pruebas del correcto funcionamiento del sistema completo.

- **Correcto funcionamiento de las Write Barrier**

Comprueba que se realiza correctamente la creación de una nueva relación desde un pez que está siendo visitado por el recolector de memoria.

Prueba realizada:
Se parte de un grafo de relaciones cualquiera, en un momento determinado se añade una nueva relación al pez que está siendo visitado en ese momento por el recolector de memoria. Se comprueba que no se incumple el invariante y que el recolector continúa su ejecución correctamente. En la siguiente imagen se muestra el estado del grafo antes de crear la nueva relación:

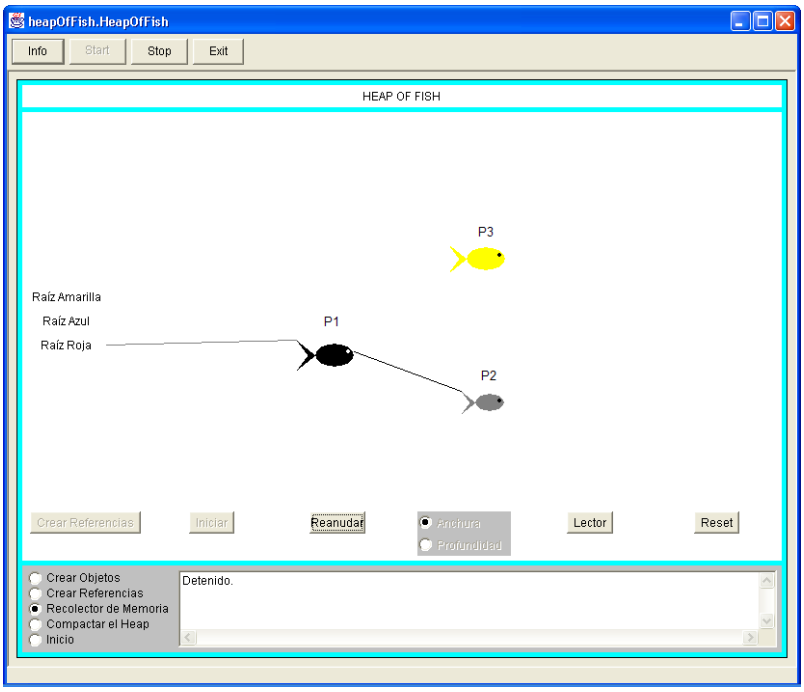



Figura 8.1: Prueba Write Barrier 1.

En este momento, en el que el recolector está visitando el nodo *P1* se añade una relación desde *P1* a *P3*.

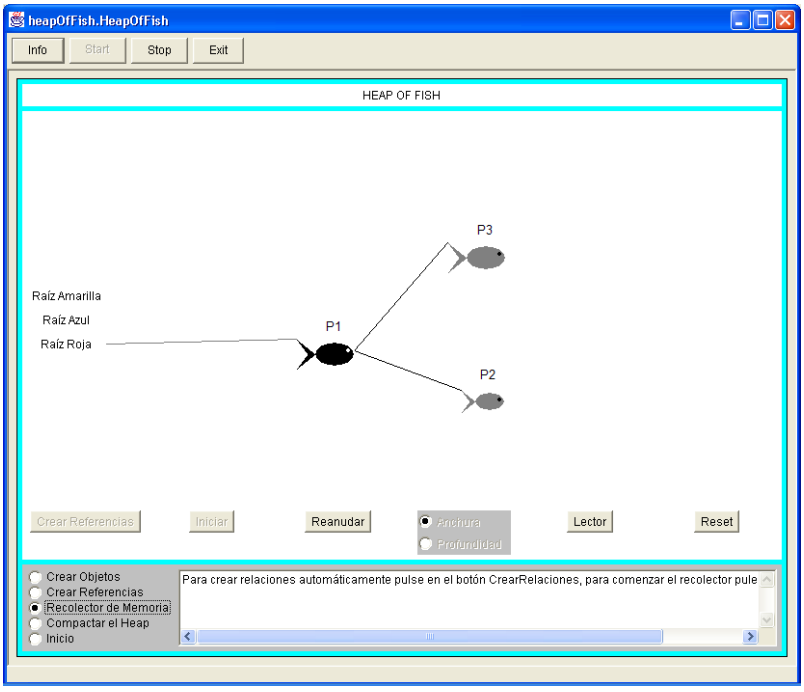



Figura 8.2: Prueba Write Barrier 2.

Se comprueba que efectivamente se cumple el invariante, ya que P3 se ha coloreado de gris. A continuación el recolector continúa su ejecución correctamente.

- **Correcto funcionamiento de la elección del tipo de recorrido**

Comprueba que el recolector realiza los 2 tipos de recorridos distintos, se realiza una ejecución para cada tipo de recorrido y se comprueba el orden de visita de los peces que realiza cada recorrido.

Prueba realizada:

Se parte del siguiente grafo de relaciones como punto de partida.

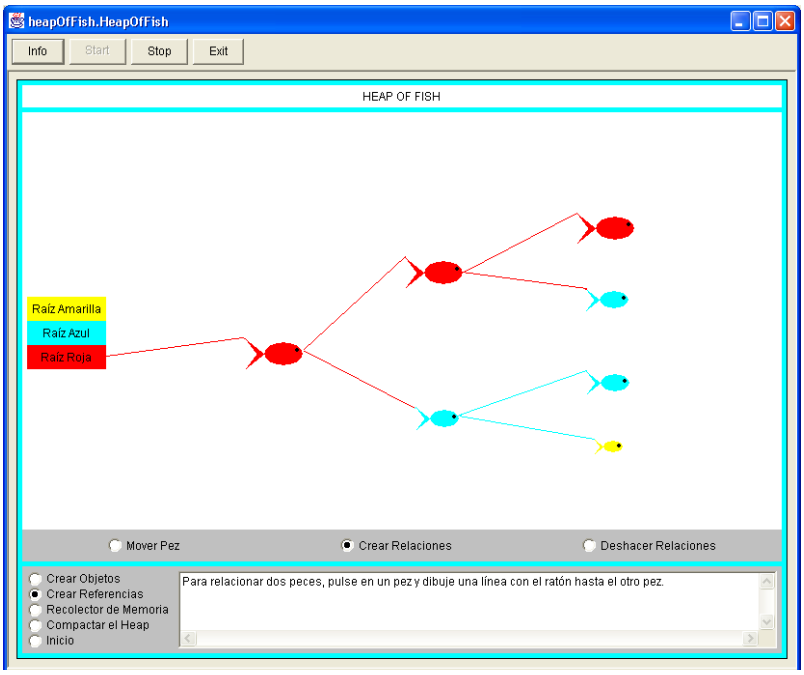



Figura 8.3: Prueba Tipo de Recorrido 1.

Se realiza una ejecución del algoritmo de recolección con recorrido en profundidad sobre el grafo anterior. Para ello, se elige el recorrido en profundidad y se pulsa el botón “Iniciar”. El orden en que este recorrido visita cada nodo es el siguiente.

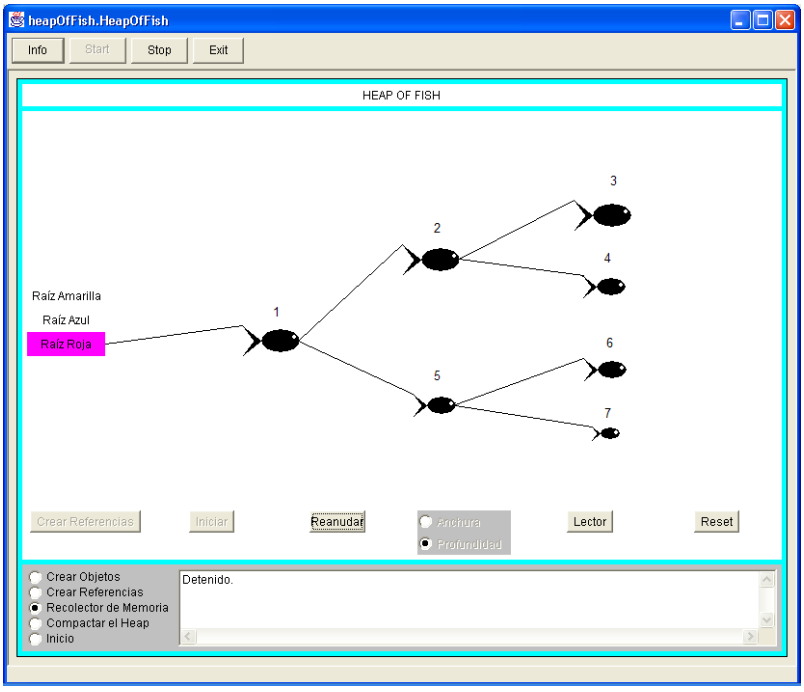



Figura 8.4: Prueba Tipo de Recorrido 2.

Se realiza una ejecución del algoritmo de recolección con recorrido en anchura sobre el grafo anterior. Para ello, se elige el recorrido en anchura y se pulsa el botón “Iniciar”. El orden en que este recorrido visita cada nodo es el siguiente.

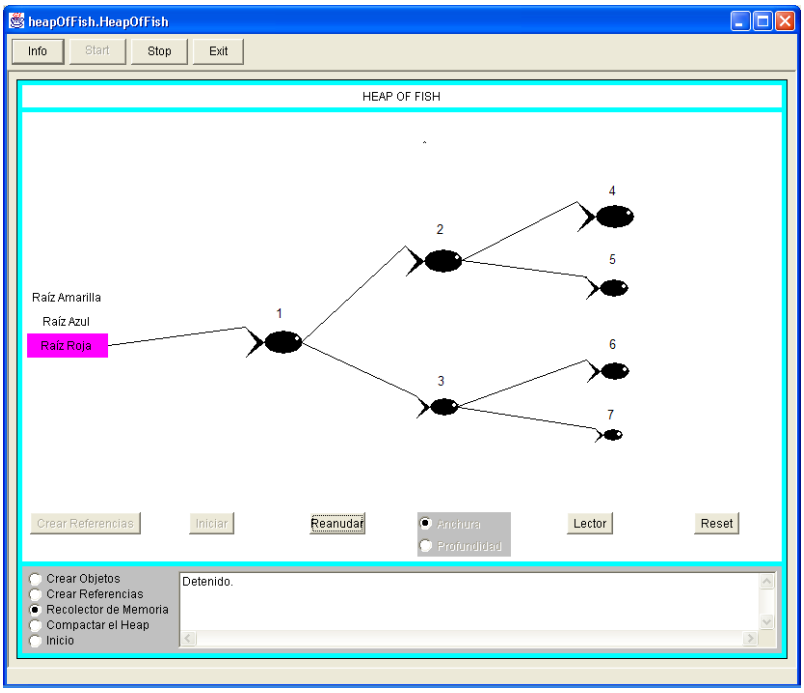



Figura 8.5: Prueba Tipo de Recorrido 3.

- **Correcto funcionamiento del botón de reset**

Comprueba que una vez que ha terminado de ejecutarse el recolector de memoria, al pulsar el botón “Reset” se prepara el heap para una nueva ejecución del recolector.

Prueba realizada:

Se parte de un grafo de relaciones cualquiera, se ejecuta el recolector de memoria pulsando el botón “Iniciar”. Una vez que el recolector ha terminado, se pulsa el botón “Reset” y se comprueba que el grafo ha quedado preparado para una nueva ejecución del recolector, es decir se han coloreado todos los nodos de blanco y se han eliminado las relaciones correspondientes a la ejecución anterior. En la siguiente imagen se muestra el estado del grafo tras la ejecución del recolector de memoria.

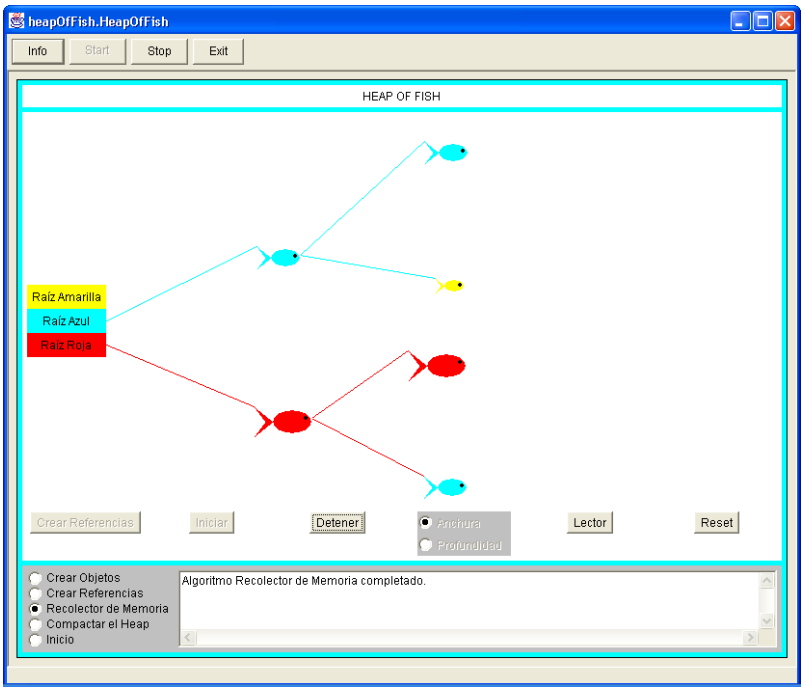



Figura 8.6: Prueba Botón Reset 1.

A continuación se muestra el estado del grafo tras haber pulsado el botón “Reset”.

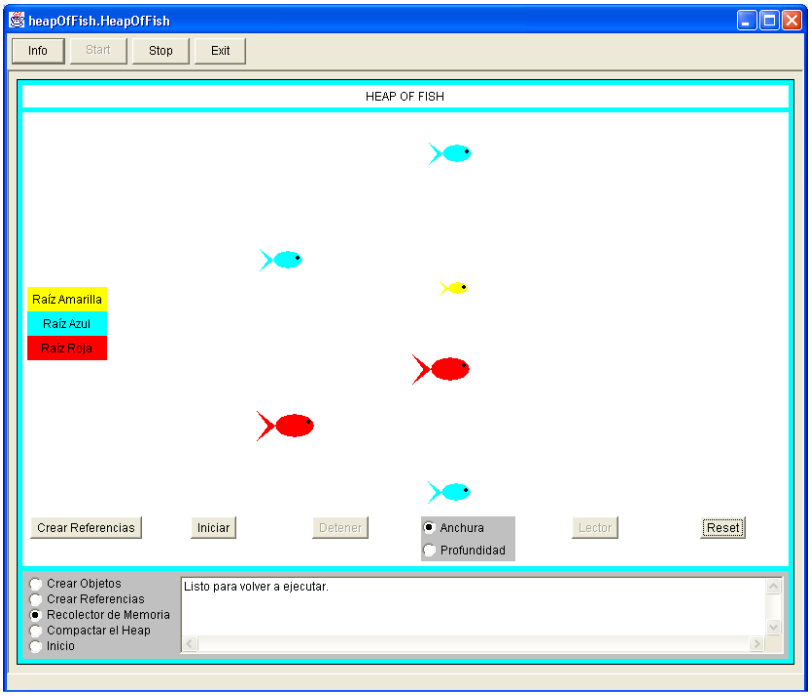



Figura 8.7: Prueba Botón Reset 2.

- **Correcto funcionamiento de la creación de relaciones automáticas**

Comprueba que se crean correctamente las relaciones de forma automática a la vez que se ejecuta el recolector de memoria.

Prueba realizada:

Se pulsa el botón “Crear Referencias” para empezar a crear relaciones de forma automática. En la imagen se muestra el estado del grafo antes de crear las relaciones automáticas.

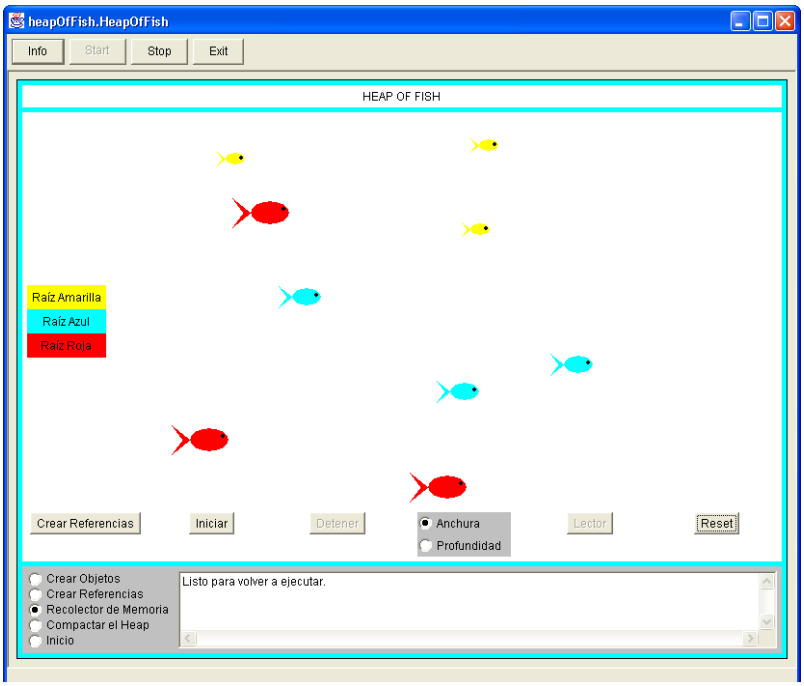



Figura 8.8: Prueba Relaciones Automáticas 1.

Al pulsar el botón “Crear Referencias”, aparece el cuadro de diálogo para introducir el ejemplo de grafo a crear.

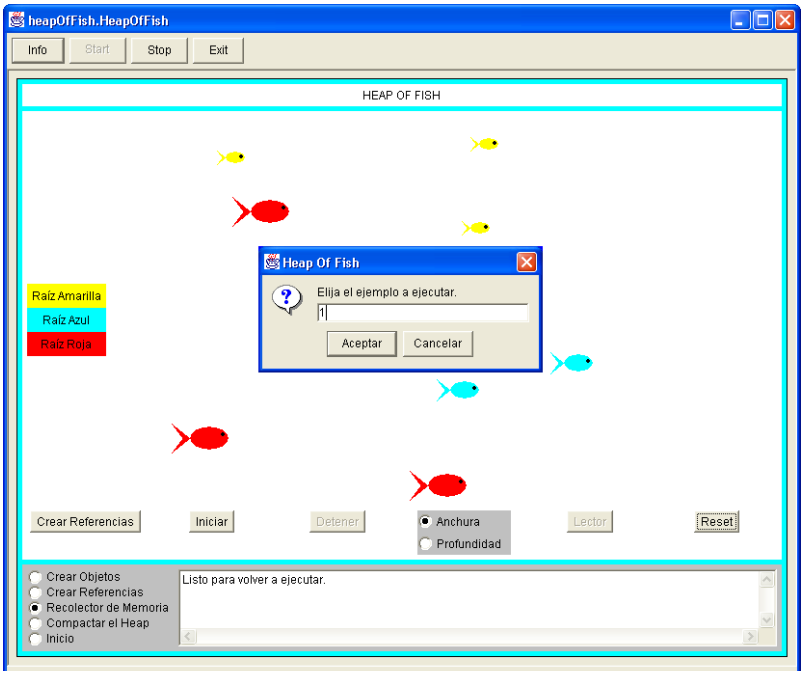



Figura 8.9: Prueba Relaciones Automáticas 2.

Una vez elegido el ejemplo, empiezan a crearse relaciones de forma automática.

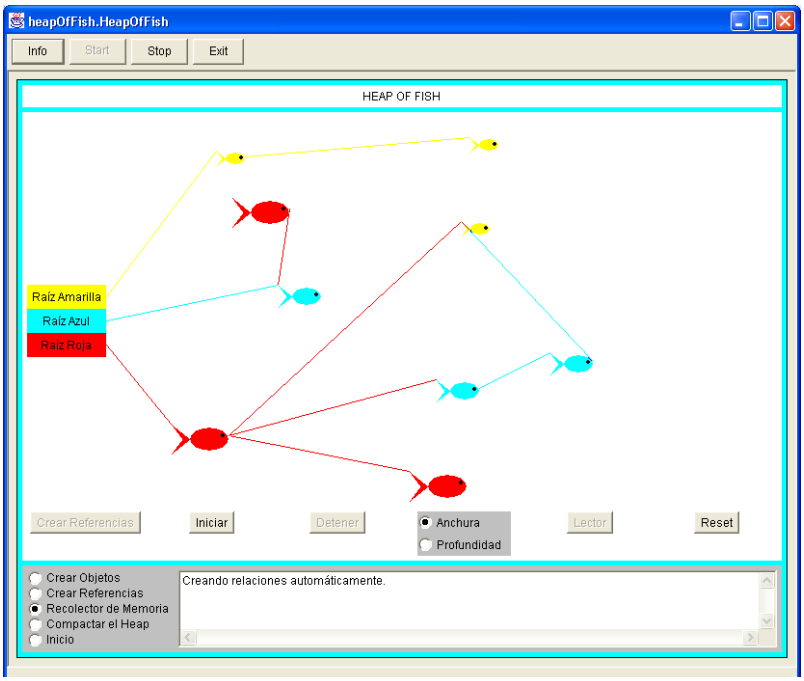



Figura 8.10: Prueba Relaciones Automáticas 3.

- **Correcto funcionamiento del lector**

Comprueba que el lector recorre el grafo de relaciones coloreando los ojos de los peces de color verde a la vez que se ejecuta el recolector de memoria.

Prueba realizada:

Se parte de un grafo de relaciones cualquiera, se ejecuta el recolector de memoria pulsando el botón “Iniciar” y a continuación se ejecuta el lector pulsando el botón “Lector”. En la imagen se muestra el grafo inicial.

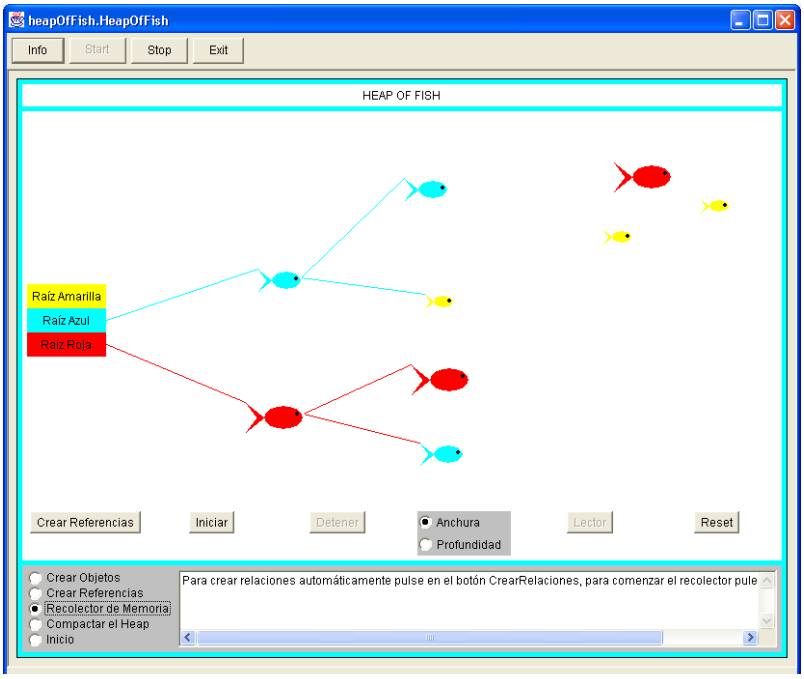



Figura 8.11: Prueba Lector 1.

A continuación comienza la ejecución del recolector de memoria y posteriormente comienza la ejecución del lector, en un momento determinado, la situación del grafo es la siguiente.

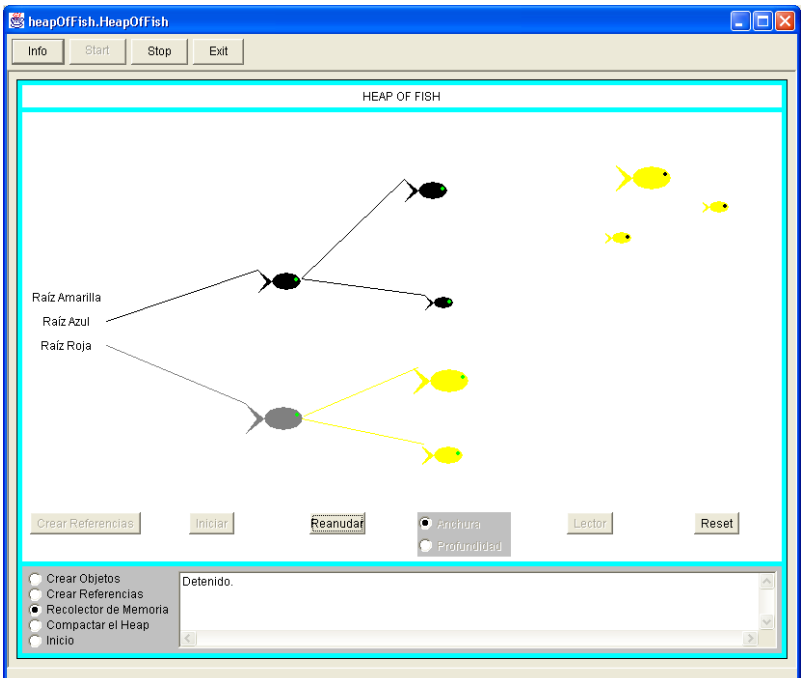



Figura 8.12: Prueba Lector 2.

Se observa que todos los peces alcanzables desde las raíces tienen su ojo coloreado de verde. Esto indica que han sido visitados por el lector. A su vez y de manera independiente se ha ido ejecutando el recolector de memoria, que ha ido recorriendo los peces y coloreándolos de gris y negro.

- **Correcto funcionamiento de los botones "Detener" y "Reanudar"**

Comprueba que al pulsar el botón detener, se detiene la ejecución del recolector de memoria, del lector y la creación de relaciones y que una vez detenidos, al pulsar el botón reanudar vuelven a ejecutarse en el punto en el que se encontraban.

Prueba realizada:

Se parte de un grafo inicial cualquiera. Se pulsa el botón "Crear Referencias" para empezar a crear relaciones de forma automática, después se ejecuta el recolector de memoria pulsando el botón "Iniciar" y a continuación se ejecuta el lector pulsando el botón "Lector". Cuando están los 3 hilos en funcionamiento, se pulsa el botón "Detener" y se comprueba que efectivamente el recolector y el lector no visitan nuevos nodos ni se crean nuevas relaciones.

Estando los 3 hilos detenidos, volver pulsar el mismo botón, que ahora se llama "Reanudar" y comprobar que recolector y lector siguen su recorrido por el grafo desde el lugar en el que se quedaron y siguen creándose nuevas relaciones.

La imagen muestra el grafo inicial.

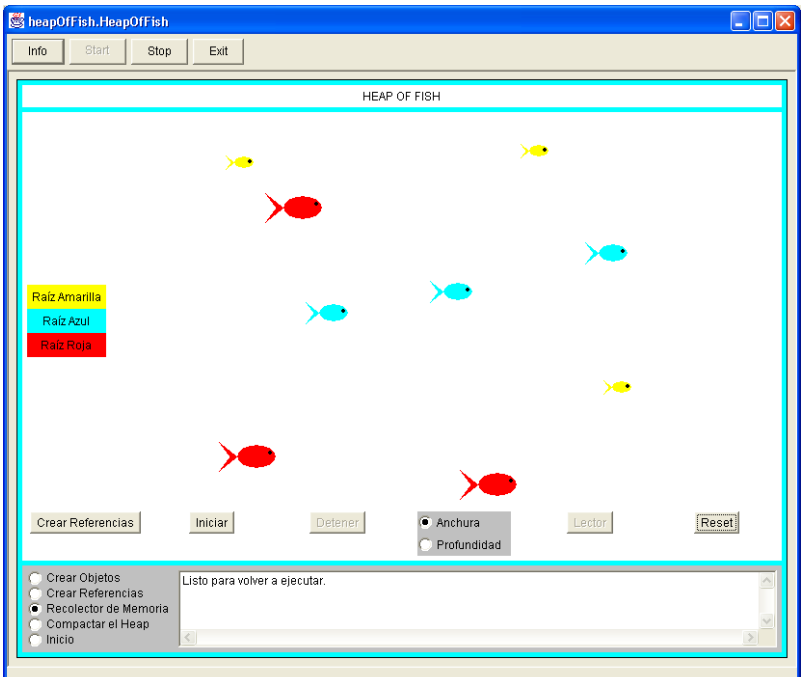



Figura 8.13: Prueba Detener y Reanudar 1.

Comienzan a crearse las relaciones automáticas, comienza la ejecución del recolector de memoria y la del lector. Se pulsa el botón detener.

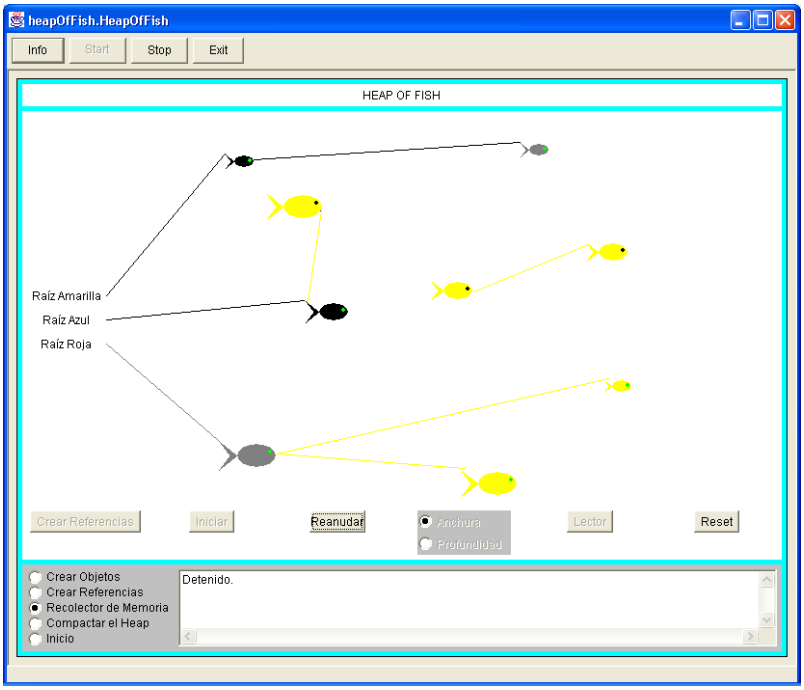



Figura 8.14: Prueba Detener y Reanudar 2.

Se comprueba que han dejado de crearse nuevas relaciones, y que ni el recolector de memoria ni el lector continúan su recorrido sobre el grafo. A continuación se vuelve a reanudar la ejecución.

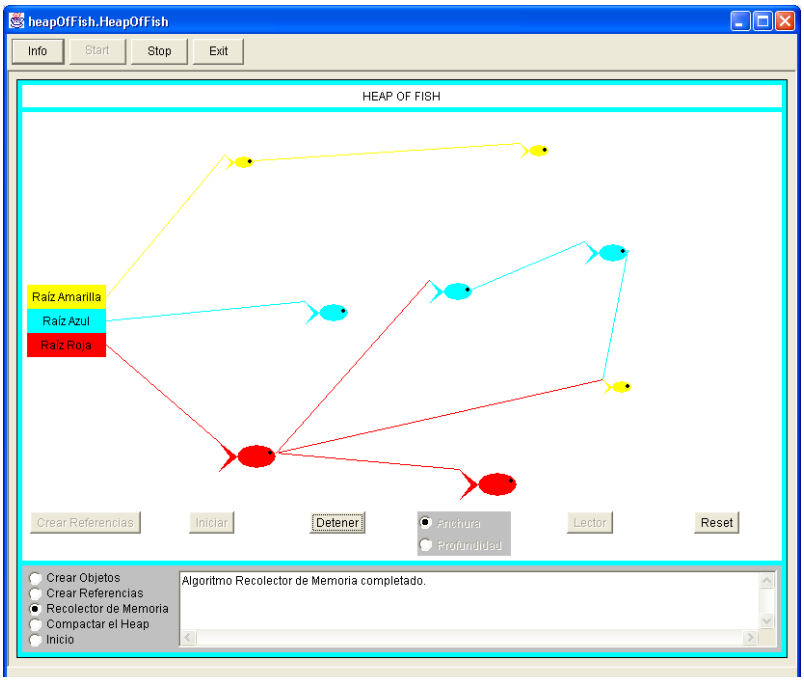



Figura 8.15: Prueba Detener y Reanudar 3.

Se comprueba que el recolector de memoria y el lector siguen su ejecución en el punto donde se detuvieron y además vuelve la creación de nuevas relaciones.

- **Correcto funcionamiento del sistema completo para un ejemplo**

Prueba general que comprueba la correcta ejecución de todas las funcionalidades del sistema. A continuación se muestra el grafo inicial.

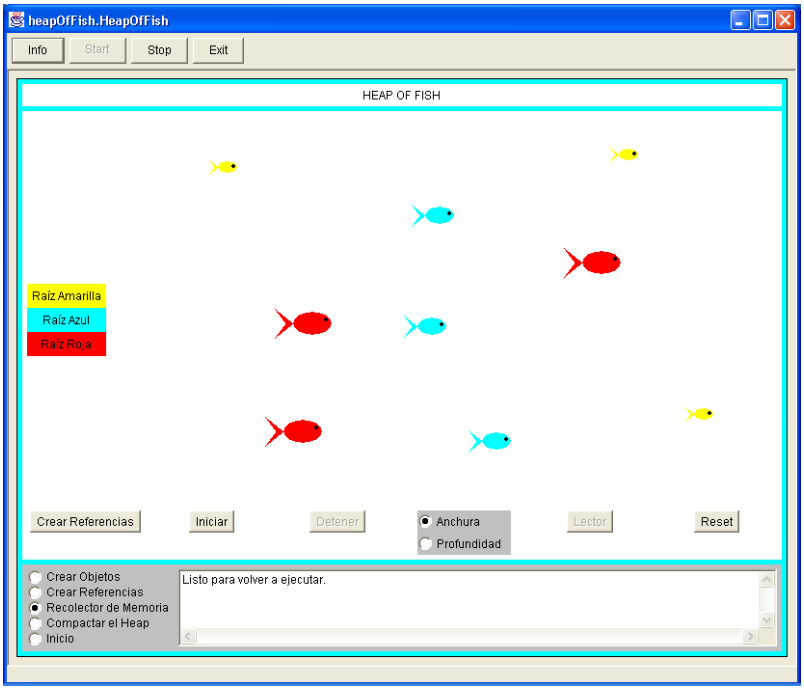



Figura 8.16: Prueba General 1.

En primer lugar se ejecuta la creación automática de relaciones para crear un grafo complejo en el que existen ciclos. A continuación se ejecuta el algoritmo recolector de memoria y por último el lector. Con esto conseguimos que a la vez que se crean

nuevas relaciones, el recolector recorre el grafo coloreando los peces de gris y negro y a su vez el lector recorre de manera independiente el mismo grafo coloreando los ojos de los peces que visita de verde. Se observa que los 3 procesos se ejecutan simultáneamente y sin problemas a pesar de que comparten el grafo de relaciones, esto vuelve a poner de manifiesto el correcto funcionamiento de las Write Barrier. Además el resultado final es el esperado, se ha creado un grafo de relaciones en el que todos los nodos alcanzables desde las raíces han sido visitados por un lado por el lector que ha coloreado sus ojos de verde y por otro lado por el recolector de memoria. Finalmente el recolector de memoria borra los peces que no han sido alcanzados.
El grafo tras la ejecución es el siguiente.

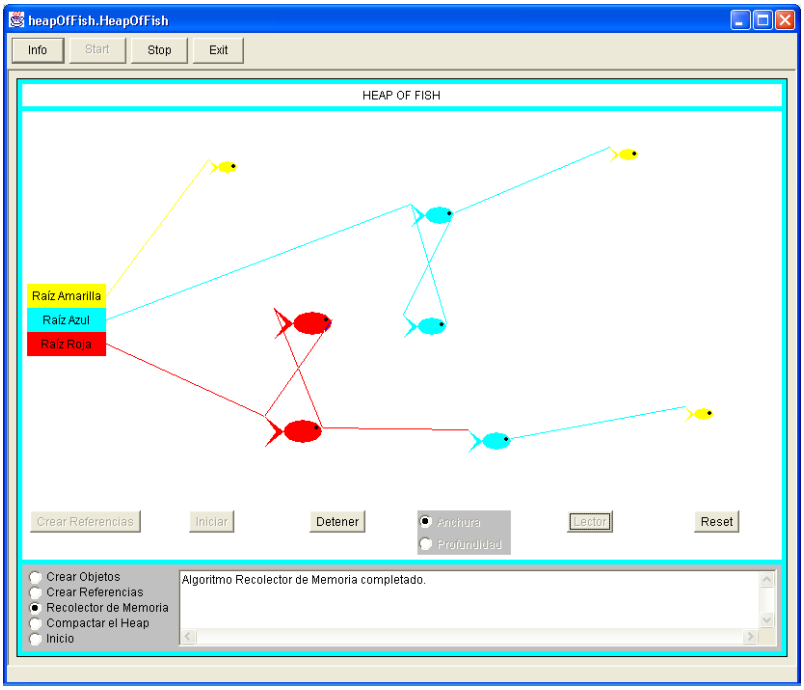



Figura 8.17: Prueba General 2.

# 9. TRABAJO FUTURO

Según se ha ido avanzando en el trabajo, han surgido nuevas ideas de mejora y posibles ampliaciones para versiones posteriores. En este apartado se indican las nuevas mejoras y ampliaciones del sistema, y de forma general, como podrían llevarse a cabo.

## *9.1 Modificación en la creación de objetos y relaciones*

Ésta ampliación del sistema tiene como objetivo facilitar el uso de la aplicación a usuarios acostumbrados a trabajar introduciendo comandos por pantalla. Se propone definir e implementar un pequeño lenguaje en el que el usuario, mediante la introducción de órdenes, pueda realizar las mismas acciones que realiza mediante ratón, por ejemplo crear los diferentes tipos de peces, crear y borrar las relaciones entre peces...
Un ejemplo de posible lenguaje sería:

Para la creación de objetos
> CR Crea pez rojo
> CZ Crea pez azul
> CM Crea pez amarillo

Para relacionar objetos con las raíces
> Ri Para relacionar la raíz roja con el objeto número i
> Zi Para relacionar la raíz azul con el objeto número i
> Ai Para relacionar la raíz amarilla con el objeto número i

Para relacionar objetos entre sí
> Oij Para relacionar el pez número i con el pez número j

Para llevar a cabo esta modificación bastaría con añadir las clases necesarias para el tratamiento de las órdenes. En primer lugar se necesita un cuadro de diálogo en el que introducir las órdenes y por otro lado es necesario un tratamiento de cada orden introducida. Una vez que la orden ha sido detectada se debe hacer una llamada al método correspondiente de creación de objetos o creación de relaciones. Estos métodos serán parecidos a los ya existentes en las clases *AllocateFishPanel, LinkFishCanvas, PecesIniciales* y *ReferenciasIniciales.*

## *9.2 Inclusión de una región*

Una de las funcionalidades que se deseaba añadir a nuestro sistema y que al final no se ha podido incorporar es la inclusión de una región. Se pretendía añadir un heap parecido al existente sobre el que no se ejecuta el recolector de memoria, los objetos de la nueva región podrían crea referencias internas sobre otros objetos de la región y referencias externas sobre el heap, no así el heap, que sólo podría crear relaciones internas. El recolector de memoria del heap debe controlar además de las relaciones internas del heap, las relaciones que provienen de la región antes de liberar los objetos. Esta ampliación es interesante, ya que se introduce el concepto de región y se podría estudiar de forma práctica la relación existente entre regiones y heap que ya ha

sido estudiada de forma teórica en el apartado 4 dedicado a las “Referencias Externas al heap”.

Para llevar a cabo esta modificación, es necesario duplicar la clase *GCHeap*, para la implementación de la nueva región. En la pantalla de creación de objetos, aparecería la nueva región al lado del heap y se podría elegir si el objeto que se desea crear se crea en la región o en el heap.

## *9.2 Transformación del algoritmo a generacional*

La última modificación que se propone es transformar el algoritmo de recolección de memoria en un algoritmo de recolección de memoria generacional. El funcionamiento de los algoritmos de recolección de memoria generacionales se explica en el apartado 2.6.

Para lograr estas modificaciones se debe permitir diferenciar la generación a la que pertenece cada objeto del heap, por ejemplo mediante un nuevo atributo *generacion* de tipo entero en la clase *ObjectHandle*, este atributo se incrementará cada vez que el recolector de memoria visite dicho objeto. También se necesita un contador para tener constancia del número de veces que el recolector de memoria se ha ejecutado, este nuevo atributo *nVeces* de tipo entero se añade a la clase *GarbageCollectCanvas*. En esta misma clase se debe modificar el algoritmo de recolección de modo que los objetos con *generacion* pequeño sean visitados con más frecuencia y a medida que la *generacion* es mayor, el recorrido sobre estos objetos es más infrecuente. Por ejemplo se puede definir que los objetos con *generacion* menor que 5 sean visitados todas las ejecuciones del algoritmo recolector. Los objetos con *generacion* entre 5 y 10 sean visitados una vez de cada tres de las que se ejecuta el recolector, es decir cuando *nVeces* múltiplo de 3. Y que los objetos con *generacion* mayor de 10 sean visitados cada 5 veces.

# Apéndice 1: Código de las Clases Principales

En este apéndice se muestra y explica el código de las clases principales del programa, haciendo especial hincapié en las clases modificadas por nosotros, se hace una diferenciación entre las clases de la aplicación de partida y las clases modificadas mostrando estas modificaciones con un sombreado especial. En cada clase se han numerado las líneas de código para poder referenciarlas en las explicaciones. Además para explicar con más detalle algunos métodos se describe un pseudocódigo.

## *1. AllocateFishPanel*

Clase que contiene los métodos necesarios para la creación de objetos en el heap.

Atributos más importantes:
El atributo *gcHeap* de tipo *GCHep* (declarado en la línea 4), es el heap que contendrá los objetos creados.
El atributo *controlPanelTextArea* de tipo *HeapOfFishTextArea* (declarado en la línea 6) es el cuadro de texto en el que aparecen los mensajes de ayuda.
El atributo *pecesIni* de la clase *PecesIniciales* (declarado en la línea 8) es el encargado de crear los peces de forma automática.

Métodos:
El único método de esta clase es el método *public boolean action(Event evt, Object arg)* que se ejecuta cuando se pulsa sobre la pantalla. El funcionamiento del método es el siguiente:

- Se comprueba si sobre lo que se ha pulsado es de tipo botón, *evt.target instanceof Button* (línea 26), si es así se distingue sobre cual de los 3 botones se ha pulsado (crearPezRojo, crearPezAzul o crearPezAmarillo).
- Si se ha pulsado sobre el botón *crearPezRojo* se crea un pez rojo (línea 30).
- Si se ha pulsado sobre el botón *crearPezAzul* se crea un pez azul (línea 49).
- Si se ha pulsado sobre el botón *crearPezAmarillo* se crea un pez amarillo (línea 67).

En la transformación del algoritmo de recolección de memoria de partida al algoritmo incremental se modificó la creación de los peces para que se crearan coloreados de blanco (líneas 41,59 y 76 para cada uno de los tipos de peces).

```
1  class AllocateFishPanel extends Panel implements Runnable{
2
3    //Heap que contendrá los objetos creados
4    GCHeap gcHeap;
5    //Cuadro de texto
6    HeapOfFishTextArea controlPanelTextArea;
7    //Atributo de la clase PecesIniciales que crea los peces al principio de la aplicación
8    PecesIniciales pecesIni;
9    PoolsCanvas poolsCanvas;
10
11   //Inicialización de la pantalla de creación de objetos
12   AllocateFishPanel(GCHeap heap, HeapOfFishTextArea ta) {
13     gcHeap = heap;
14     controlPanelTextArea = ta;
15     setBackground(Color.blue);
16     setLayout(new BorderLayout());
```

```
17        poolsCanvas = new PoolsCanvas(gcHeap);
18        add("South", new AllocateFishButtonPanel());
19        add("Center", poolsCanvas);
20        pecesIni = new PecesIniciales(gcHeap);
21        start();
22      }
23      //Método  que se ejecuta al pulsar sobre la pantalla
24      public boolean action(Event evt, Object arg) {
25        //Si se ha pulsado sobre algo de tipo Botón se diferencia entre los diferentes botones
26        if (evt.target instanceof Button) {
27          String bname = (String) arg;
28          Dimension canvasDim = poolsCanvas.size();
29          //Si se ha pulsado sobre el botón newRedFish se crea un pez rojo
30          if (bname.equals(HeapOfFishStrings.newRedFish)) {
31            FishIcon fish = new BigRedFishIcon(false);
32            int newFish = gcHeap.allocateObject(12, fish);
33            if (newFish > 0) {
34              ObjectHandle oh = gcHeap.getObjectHandle(newFish);
35              gcHeap.setObjectPool(oh.objectPos, 0);
36              gcHeap.setObjectPool(oh.objectPos + 1, 0);
37              gcHeap.setObjectPool(oh.objectPos + 2, 0);
38              controlPanelTextArea.setText(HeapOfFishStrings.newRedFishAllocated);
39              //Se inicializa el color que se usa en el algoritmo de recolección de memoria de
40              //los tres colores a blanco
41              oh.myColor=Color.white;
42           }
43            else {
44              controlPanelTextArea.setText(HeapOfFishStrings.newRedFishNotAllocated);
45            }
46            poolsCanvas.repaint();
47          }
48          //Si se ha pulsado sobre el botón newBlueFish se crea un pez azul
49          else if (bname.equals(HeapOfFishStrings.newBlueFish)) {
50            FishIcon fish = new MediumBlueFishIcon(false);
51            int newFish = gcHeap.allocateObject(8, fish);
52            if (newFish > 0) {
53              ObjectHandle oh = gcHeap.getObjectHandle(newFish);
54              gcHeap.setObjectPool(oh.objectPos, 0);
55              gcHeap.setObjectPool(oh.objectPos + 1, 0);
56              controlPanelTextArea.setText(HeapOfFishStrings.newBlueFishAllocated);
57              //Se inicializa el color que se usa en el algoritmo de recolección de memoria de
58              //los tres colores a blanco
59              oh.myColor=Color.white;
60           }
61            else {
62              controlPanelTextArea.setText(HeapOfFishStrings.newBlueFishNotAllocated);
63            }
64            poolsCanvas.repaint();
65          }
66          //Si se ha pulsado sobre el botón newYellowFish se crea un pez amarillo
67          else if (bname.equals(HeapOfFishStrings.newYellowFish)) {
68            FishIcon fish = new LittleYellowFishIcon(false);
69            int newFish = gcHeap.allocateObject(4, fish);
70            if (newFish > 0) {
71              ObjectHandle oh = gcHeap.getObjectHandle(newFish);
72              gcHeap.setObjectPool(oh.objectPos, 0);
73              controlPanelTextArea.setText(HeapOfFishStrings.newYellowFishAllocated);
74              //Se inicializa el color que se usa en el algoritmo de recolección de memoria de
75              //los tres colores a blanco
76              oh.myColor=Color.white;
```

```
77            }
78            else {
79              controlPanelTextArea.setText(HeapOfFishStrings.newYellowFishNotAllocated);
80            }
81            poolsCanvas.repaint();
82          }
83        }
84        return true;
85      }
86    }
```

## 2. GarbageCollectButtonPanel

Clase en la que se crean los botones de la pantalla desde la que se ejecuta el recolector y se añaden al panel. Se sustituyen los botones que había en la aplicación de partida que eran *b_start* y *b_reset* por los siguientes:

Botón *b_Inicio*, que se pulsará para iniciar el algoritmo recolector (se declara en la línea 6 y se añade al panel en la línea 26).
Botón *b_Pausa*, que se pulsará para detener o reanudar el recolector (se declara en la línea 4 y se añade al panel en la línea 31).
Botón *b_reset*, que se pulsará para eliminar las relaciones entre peces y preparar el grafo para una nueva ejecución del recolector (se declara en la línea 13 y se añade al panel en la línea 44).
Botón *b_Lector*, que se pulsará para iniciar el lector (se declara en la línea 8 y se añade al panel en la línea 40).
Botón *b_ref*, que se pulsará para crear automáticamente relaciones entre peces (se declara en la línea 10 y se añade al panel en la línea 22).
El atributo *gcdbg* es el grupo de checkbox en el que se puede elegir el tipo de recorrido que realiza el recolector. (se declara en la línea 16 y se añade al panel en la línea 35).

```
1  class GarbageCollectButtonPanel extends Panel {
2
3   //Botón que se pulsará para detener o reaunudar el recolector
4   Button b_Pausa = new Button(HeapOfFishStrings.pausa);
5   //Botón que se pulsará para iniciar el algoritmo recolector
6   Button b_Inicio = new Button(HeapOfFishStrings.inicio);
7   //Botón que se pulsará para iniciar el lector
8   Button b_Lector = new Button(HeapOfFishStrings.lector);
9   //Botón que se pulsará para crear automáticamente relaciones entre peces
10  Button b_ref = new Button(HeapOfFishStrings.refIni);
11
12  //Botón que se pulsara para eliminar todas las relaciones entre peces
13  Button b_reset = new Button(HeapOfFishStrings.reset);
14  //Grupo de check box en el que se puede elegir el tipo de recorrido que realiza el
15  //recolector: anchura o profundidad
16  GarbageCollectCheckboxGroup gcdbg = new GarbageCollectCheckboxGroup();
17
18     //Se añaden los botones y el grupo de checkbox al panel
19     GarbageCollectButtonPanel() {
20        setLayout(new GridLayout(1, 6));
21        Panel p = new Panel();
22        p.add(b_ref);
23        add(p);
24
25        p = new Panel();
26        p.add(b_Inicio);
27        add(p);
28
29        p = new Panel();
30        b_Pausa.enable(false);
31        p.add(b_Pausa);
32        add(p);
33
34        p=new Panel();
35        p.add(gcdbg);
36        add(p);
37
```

```
38          p = new Panel();
39          b_Lector.enable(false);
40          p.add(b_Lector);
41          add(p);
42
43          p = new Panel();
44          p.add(b_reset);
45          add(p);
46      }
47  }
```

## *3. GarbageCollectCanvas*

Esta es una de las clases más importantes del programa, ya que contiene el método que implementa el algoritmo recolector de memoria incremental y los que recorren el grafo de relaciones.

Atributos más importantes
El atributo *gcHeap* de tipo *GCHep* (declarado en la línea 2), es el heap que contendrá los objetos creados.
El atributo *localVars* de tipo *LocalVariables* (declarado en la línea 3) implementa las raíces.
El atributo *controlPanelTextArea* de tipo *HeapOfFishTextArea* (declarado en la línea 6) es el cuadro de texto en el que aparecen los mensajes de ayuda.
Desde la línea 18 hasta la 38 se declara la codificación de los estados posibles del algoritmo recolector de memoria.
El atributo *parar* de tipo booleano (línea 42) indica si el recolector está parado o en ejecución.
El atributo *thread* de tipo *Thread* (línea 43) es el hilo que ejecutará el recolector de memoria.
El atributo *currentGCState* de tipo entero (línea 45) codifica el estado en el que se encuentra el recolector, inicialmente el estado será "detenido"

Métodos más importantes
El método *nextGCStep()* (línea 91) es el método que implementa el recolector de memoria, el recolector va pasando por una serie de estados y en cada estado debe hacer una serie de acciones. A continuación se muestra el pseudocódigo:
Si el estado del algoritmo es igual a:

**- *detenido***: el algoritmo no ha comenzado.
Pasar al estado *visitando raíz amarilla.*
(línea 92)

**- *visitando raíz amarilla***: el algoritmo marca como visitada la raíz amarilla y pasa al estado *visitando peces desde raíz amarilla.*
(línea 101)

**- *visitando peces desde raíz amarilla***: se recorren todos los peces del árbol cuya raíz es la raíz amarilla, cuando ha terminado de recorrer todos los nodos pasa al estado *visitando raíz azul.*
(línea 120)

**- *visitando raíz azul:*** el algoritmo marca como visitada la raíz azul y pasa al estado *visitando peces desde raíz azul.*
(línea 144)

**- *visitando peces desde raíz azul***: se recorren todos los peces del árbol cuya raíz es la raíz azul, cuando ha terminado de recorrer todos los nodos pasa al estado *visitando raíz roja.*
(línea 163)

**- *visitando raíz roja***: el algoritmo marca como visitada la raíz roja y pasa al estado *visitando peces desde raíz roja.*
(línea 186)

***- visitando peces desde raíz roja**:* se recorren todos los peces del árbol cuya raíz es la raíz roja, cuando ha terminado de recorrer todos los nodos pasa al estado *pasando por nuevos nodos.*
(línea 204)

***- pasando por nuevos nodos:*** se recorren los nodos que se han ido añadiendo a la nueva estructura de datos y se pasa al estado *borrando peces no alcanzados.*
(línea 226)

***- borrando peces no alcanzados:*** se borran todos los peces que no han sido alcanzados, es decir los coloreados como blancos y se pasa al estado detenido.
(línea 248)

El método *traverseNextFishNode(ObjectHandle oh)* (línea 277) hace una llamada al recorrido en profundidad (*traverseNextFishNodeProfundidad(ObjectHandle oh)*) o al recorrido en anchura (*traverseNextFishNodeProfundidad (ObjectHandle oh)*) según corresponda.
El método booleano *traverseNextFishNodeProfundidad(ObjectHandle oh)* (línea 288) recorre en profundidad el grafo, tomando como raíz *oh*. Devuelve verdadero cuando ha terminado el recorrido y falso en caso contrario.
El método booleano *traverseNextFishNodeAnchura(ObjectHandle oh)* (línea 356) recorre en anchura el grafo, tomando como raíz *oh*. Devuelve verdadero cuando ha terminado el recorrido y falso en caso contrario. A continuación se muestra el pseudocódigo del recorrido en anchura.
- Si *oh* tiene una relación de Amistad, se colorea de gris el pez destino de dicha relación y se añade a *listaPendientes* (línea 361).
- Si *oh* tiene una relación de Comida, se colorea de gris el pez destino de dicha relación y se añade a *listaPendientes* (línea 377).
- Si *oh* tiene una relación de Aperitivo, se colorea de gris el pez destino de dicha relación y se añade a *listaPendientes* (línea 406).
- Una vez que ha visitado todos los hijos, colorea oh de negro, calcula cual es el siguiente objeto a visitar (el siguiente a *oh* en *listaPendiente*) y devuelve verdadero (línea 361).

El método *resetGCState()* (línea 424) colorea todos los peces de color blanco, preparando el grafo para una nueva ejecución.
El método *seguidoGC()* (línea 453) es el bucle principal del recolector, mientras no se haya pulsado el botón detener y el algoritmo no esté en estado detenido, se sigue ejecutando el bucle. En cada iteración del bucle se hace una llamada al método *nextGCStep()*.

```
1  public class GarbageCollectCanvas extends Canvas implements Runnable{
2    private GCHeap gcHeap;
3    private LocalVariables localVars;
4    private HeapOfFishTextArea controlPanelTextArea;;
5    private final int poolImageInsets = 5;
6    private final int localVarStringMargin = 5;
7    private int localVarRectWidth;
8    private int localVarRectHeight;
9    private int xLocalVarRectStart;
10   private int yYellowFishLocalVarStart;
11   private int yBlueFishLocalVarStart;
12   private int yRedFishLocalVarStart;
```

```
13
14      // Fish area is just to the left of the local variables.
15      private int xFishAreaStart;
16
17      // State variables for the garbage collector
18      private final int garbageCollectorHasNotStarted = 0;
19
20      private final int startingAtYellowLocalVariableRoot = 1;
21      private final int traversingFromYellowLocalVariableRoot = 2;
22      private final int doneWithYellowLocalVariableRoot = 3;
23
24      private final int startingAtBlueLocalVariableRoot = 4;
25      private final int traversingFromBlueLocalVariableRoot = 5;
26      private final int doneWithBlueLocalVariableRoot = 6;
27
28      private final int startingAtRedLocalVariableRoot = 7;
29      private final int traversingFromRedLocalVariableRoot = 8;
30      private final int doneWithRedLocalVariableRoot = 9;
31
32      private final int readyToSweepUnmarkedFish = 10;
33      private final int doneSweepingUnmarkedFish = 11;
34
35      private final int garbageCollectorIsDone = 12;
36
37      //Nuevo estado que codifica el paso por la lista de nuevos objetos
38      private final int pasarPorListaDeNuevos = 13;
39      //Indice para recorrer el vector de nuevos objetos
40      private int indiceNuevosObjetos;
41      //Indica si el recolector está parado o en ejecución
42      private boolean parar = false;
43      private Thread thread;
44
45      private int currentGCState = garbageCollectorHasNotStarted;
46
47      private boolean fishAreBeingMarked;
48      private int currentFishBeingMarked;
49
50      private boolean yellowFishLocalVarIsCurrentGCMarkNode;
51      private boolean blueFishLocalVarIsCurrentGCMarkNode;
52      private boolean redFishLocalVarIsCurrentGCMarkNode;
53
54      private Color yellowFishLocalVarLineColor;
55      private Color blueFishLocalVarLineColor;
56      private Color redFishLocalVarLineColor;
57
58      GarbageCollectCanvas(GCHeap heap, LocalVariables locVars, HeapOfFishTextArea ta) {
59        setBackground(Color.blue);
60        gcHeap = heap;
61        localVars = locVars;
62        controlPanelTextArea = ta;
63      }
64
65      //Métodos accesores y mutadores
66      public Thread dameThread(){
67        return thread;
68      }
69
70      public boolean dameParar(){
71        return parar;
72      }
```

```
73
74    public GCHeap dameGCHeap(){
75      return gcHeap;
76    }
77
78    public HeapOfFishTextArea damePanelTextArea(){
79      return controlPanelTextArea;
80    }
81
82    public LocalVariables dameLocalVars(){
83      return localVars;
84    }
85
86    public int dameEstado(){
87      return currentGCState;
88    }
89
90    //Pasos del algoritmo recolector
91    public synchronized void nextGCStep(){
92      switch (currentGCState) {
93        // Estado: No comenzado
94        case garbageCollectorHasNotStarted:
95          yellowFishLocalVarIsCurrentGCMarkNode = true;
96          currentGCState = startingAtYellowLocalVariableRoot;
97          controlPanelTextArea.setText(HeapOfFishStrings.traversingYellowRoot);
98          break;
99
100       //Pasando por la raiz amarila
101       case startingAtYellowLocalVariableRoot:
102         yellowFishLocalVarIsCurrentGCMarkNode = false;
103         if (localVars.yellowFish != 0) {
104           ObjectHandle oh = gcHeap.getObjectHandle(localVars.yellowFish);
105           yellowFishLocalVarIsCurrentGCMarkNode = false;
106           oh.myColor = Color.gray;
107           yellowFishLocalVarLineColor = Color.gray;
108           currentFishBeingMarked = localVars.yellowFish;
109           fishAreBeingMarked = true;
110           currentGCState = traversingFromYellowLocalVariableRoot;
111         }
112         else {
113           blueFishLocalVarIsCurrentGCMarkNode = true;
114           currentGCState = startingAtBlueLocalVariableRoot;
115           controlPanelTextArea.setText(HeapOfFishStrings.traversingBlueRoot);
116         }
117         break;
118
119       //Pasando por objetos desde la raiz amarila
120       case traversingFromYellowLocalVariableRoot:
121         ObjectHandle oh2 = gcHeap.getObjectHandle(currentFishBeingMarked);
122         boolean doneWithThisTree;
123         doneWithThisTree = traverseNextFishNode(oh2);
124         if (doneWithThisTree) {
125           ObjectHandle oh = gcHeap.getObjectHandle(localVars.yellowFish);
126           yellowFishLocalVarLineColor = Color.black;
127           oh.myColor = Color.black;
128           fishAreBeingMarked = false;
129           yellowFishLocalVarIsCurrentGCMarkNode = true;
130           currentGCState = doneWithYellowLocalVariableRoot;
131           controlPanelTextArea.setText(HeapOfFishStrings.doneWithYellowRoot);
132         }
```

```
133           break;
134
135         //Terminado el paso por objetos desde la raiz amarila
136         case doneWithYellowLocalVariableRoot:
137           yellowFishLocalVarIsCurrentGCMarkNode = false;
138           blueFishLocalVarIsCurrentGCMarkNode = true;
139           currentGCState = startingAtBlueLocalVariableRoot;
140           controlPanelTextArea.setText(HeapOfFishStrings.traversingBlueRoot);
141           break;
142
143         //Pasando por la raiz azul
144         case startingAtBlueLocalVariableRoot:
145           blueFishLocalVarIsCurrentGCMarkNode = false;
146           if (localVars.blueFish != 0) {
147             ObjectHandle oh = gcHeap.getObjectHandle(localVars.blueFish);
148             blueFishLocalVarIsCurrentGCMarkNode = false;
149             oh.myColor = Color.gray;
150             blueFishLocalVarLineColor = Color.gray;
151             currentFishBeingMarked = localVars.blueFish;
152             fishAreBeingMarked = true;
153             currentGCState = traversingFromBlueLocalVariableRoot;
154           }
155           else {
156             redFishLocalVarIsCurrentGCMarkNode = true;
157             currentGCState = startingAtRedLocalVariableRoot;
158             controlPanelTextArea.setText(HeapOfFishStrings.traversingRedRoot);
159           }
160           break;
161
162         //Pasando por objetos desde la raiz azul
163         case traversingFromBlueLocalVariableRoot:
164           ObjectHandle oh3 = gcHeap.getObjectHandle(currentFishBeingMarked);
165           doneWithThisTree = traverseNextFishNode(oh3);
166           if (doneWithThisTree) {
167             ObjectHandle oh = gcHeap.getObjectHandle(localVars.blueFish);
168             blueFishLocalVarLineColor = Color.black;
169             oh.myColor = Color.black;
170             fishAreBeingMarked = false;
171             blueFishLocalVarIsCurrentGCMarkNode = true;
172             currentGCState = doneWithBlueLocalVariableRoot;
173             controlPanelTextArea.setText(HeapOfFishStrings.doneWithBlueRoot);
174           }
175           break;
176
177         //Terminado el paso por objetos desde la raiz azul
178         case doneWithBlueLocalVariableRoot:
179           blueFishLocalVarIsCurrentGCMarkNode = false;
180           redFishLocalVarIsCurrentGCMarkNode = true;
181           currentGCState = startingAtRedLocalVariableRoot;
182           controlPanelTextArea.setText(HeapOfFishStrings.traversingRedRoot);
183           break;
184
185         //Pasando por la raiz roja
186         case startingAtRedLocalVariableRoot:
187           redFishLocalVarIsCurrentGCMarkNode = false;
188           if (localVars.redFish != 0) {
189             ObjectHandle oh = gcHeap.getObjectHandle(localVars.redFish);
190             redFishLocalVarIsCurrentGCMarkNode = false;
191             oh.myColor = Color.gray;
192             redFishLocalVarLineColor = Color.gray;
```

```
193             currentFishBeingMarked = localVars.redFish;
194             fishAreBeingMarked = true;
195             currentGCState = traversingFromRedLocalVariableRoot;
196           }
197           else {
198             currentGCState = pasarPorListaDeNuevos;
199             controlPanelTextArea.setText(HeapOfFishStrings.pasandoNuevosObjetos);
200           }
201           break;
202
203         //Pasando por objetos desde la raiz roja
204         case traversingFromRedLocalVariableRoot:
205           ObjectHandle oh4 = gcHeap.getObjectHandle(currentFishBeingMarked);
206           doneWithThisTree = traverseNextFishNode(oh4);
207           if (doneWithThisTree) {
208             ObjectHandle oh = gcHeap.getObjectHandle(localVars.redFish);
209             redFishLocalVarLineColor = Color.black;
210             oh.myColor = Color.black;
211             fishAreBeingMarked = false;
212             redFishLocalVarIsCurrentGCMarkNode = true;
213             currentGCState = doneWithRedLocalVariableRoot;
214             controlPanelTextArea.setText(HeapOfFishStrings.doneWithRedRoot);
215           }
216           break;
217
218         //Terminado el paso por objetos desde la raiz roja
219         case doneWithRedLocalVariableRoot:
220           redFishLocalVarIsCurrentGCMarkNode = false;
221           currentGCState = pasarPorListaDeNuevos;
222           controlPanelTextArea.setText(HeapOfFishStrings.pasandoNuevosObjetos);
223           break;
224
225         //Pasando por objetos nuevos
226         case pasarPorListaDeNuevos:
227           //Si la lista de nuevos objetos está vacía, se ha terminado este paso y se pasa al
228           //siguiente
229           if (gcHeap.listaNuevosObjetos.isEmpty()){
230             currentGCState = readyToSweepUnmarkedFish;
231             controlPanelTextArea.setText(HeapOfFishStrings.readyToSweepUnmarkedFish);
232             break;
233           }
234           //Visitamos el objeto que toca tratar
235           Objeto_Indice oi1 =
236       (Objeto_Indice)gcHeap.listaNuevosObjetos.elementAt(indiceNuevosObjetos);
237           currentFishBeingMarked=oi1.dameIndice();
238           //Se visitan los hijos del objeto a tratar
239           doneWithThisTree = traverseNextFishNode(oi1.dameOh());
240           //Cuando se han visitado todos los hijos, el objeto a tratar se colorea de negro
241           oi1.dameOh().myColor = Color.black;
242           traverseBackFromGrayLine(oi1.dameIndice());
243           //Se elimina el objeto tratado de la lista de nuevos objetos
244           gcHeap.listaNuevosObjetos.remove(indiceNuevosObjetos);
245           break;
246
247         //Preparado para borrar los objetos que no se han alcanzado
248         case readyToSweepUnmarkedFish:
249           int objectsFreedCount = 0;
250           //Se reccorre el grafo de relaciones y se borran los peces coloreados de blanco
251           for (int i = 0; i < gcHeap.getHandlePoolSize(); ++i) {
252             ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
```

```
253         if (!oh.free && oh.myColor == Color.white) {
254           gcHeap.freeObject(i + 1);
255           ++objectsFreedCount;
256         }
257       }
258       currentGCState = doneSweepingUnmarkedFish;
259       String doneSweepingText = HeapOfFishStrings.sweptFish0 + objectsFreedCount
260                           + HeapOfFishStrings.sweptFish1;
261       controlPanelTextArea.setText(doneSweepingText);
262       break;
263
264     //Proceso terminado
265     case doneSweepingUnmarkedFish:
266       currentGCState = garbageCollectorIsDone;
267       controlPanelTextArea.setText(HeapOfFishStrings.garbageCollectionDone);
268       break;
269
270     case garbageCollectorIsDone:
271     default:
272       break;
273   }
274  }
275
276  //Método que hace la llamada al recorrido en profundidad o en anchura según corresponda
277  private boolean traverseNextFishNode (ObjectHandle oh) {
278    boolean resultado;
279    if (gcHeap.AnchOProf==0){
280      resultado=traverseNextFishNodeProfundidad(oh);
281    }
282    else{
283      resultado=traverseNextFishNodeAnchura(oh);
284    }
285    return resultado;
286  }
287
288  // Método que recorre el grafo en profundidad
289  private synchronized boolean traverseNextFishNodeProfundidad(ObjectHandle oh) {
290    int myFriendIndex = gcHeap.getObjectPool(oh.objectPos);
291    if ((myFriendIndex != 0) && ((oh.myFriendLineColor == Color.white)||(oh.myFriendLineColor
292 == Color.gray))) {
293      oh.myFriendLineColor = Color.gray;
294      ObjectHandle myFriend = gcHeap.getObjectHandle(myFriendIndex);
295      myFriend.previousNodeInGCTraversalIsAFish = true;
296      myFriend.previousFishInGCTraversal = currentFishBeingMarked;
297      if (myFriend.myColor == Color.white) {
298        myFriend.myColor = Color.gray;
299      }
300      currentFishBeingMarked = myFriendIndex;
301      return false;
302    }
303    else if (oh.fish.getFishColor() == Color.yellow) {
304      if (oh.previousNodeInGCTraversalIsAFish) {
305        traverseBackFromGrayLine(oh.previousFishInGCTraversal);
306        return false;
307      }
308      return true;
309    }
310
311    int myLunchIndex = gcHeap.getObjectPool(oh.objectPos + 1);
312
```

```
313     if ((myLunchIndex != 0) && ((oh.myLunchLineColor == Color.white)||(oh.myLunchLineColor
314 == Color.gray))) {
315       oh.myLunchLineColor = Color.gray;
316       ObjectHandle myLunch = gcHeap.getObjectHandle(myLunchIndex);
317       myLunch.previousNodeInGCTraversalIsAFish = true;
318       myLunch.previousFishInGCTraversal = currentFishBeingMarked;
319       if (myLunch.myColor == Color.white) {
320         myLunch.myColor = Color.gray;
321       }
322       currentFishBeingMarked = myLunchIndex;
323       return false;
324     }
325     else if (oh.fish.getFishColor() == Color.cyan) {
326       if (oh.previousNodeInGCTraversalIsAFish) {
327         traverseBackFromGrayLine(oh.previousFishInGCTraversal);
328         return false;
329       }
330       return true;
331     }
332
333     int mySnackIndex = gcHeap.getObjectPool(oh.objectPos + 2);
334
335     if ((mySnackIndex != 0) && ((oh.mySnackLineColor == Color.white)||(oh.mySnackLineColor
336 == Color.gray))) {
337       oh.mySnackLineColor = Color.gray;
338       ObjectHandle mySnack = gcHeap.getObjectHandle(mySnackIndex);
339       mySnack.previousNodeInGCTraversalIsAFish = true;
340       mySnack.previousFishInGCTraversal = currentFishBeingMarked;
341       if (mySnack.myColor == Color.white) {
342         mySnack.myColor = Color.gray;
343       }
344       currentFishBeingMarked = mySnackIndex;
345       return false;
346     }
347     else if (oh.previousNodeInGCTraversalIsAFish) {
348       traverseBackFromGrayLine(oh.previousFishInGCTraversal);
349       return false;
350     }
351     return true;
352   }
353
354   //Método que recorre el grafo en anchura a partir del nodo oh, devuelve true cuando ha
355   //visitado todos los hijos de oh
356   private synchronized boolean traverseNextFishNodeAnchura(ObjectHandle oh) {
357     //A partir de la posición del objeto, se puede calcular la posición de los hijos
358     //Se calcula la posicion del objeto Amigo de oh
359     int myFriendIndex = gcHeap.getObjectPool(oh.objectPos);
360     //Si esa posición no es vacía, es decir, si tiene Amigo
361     if (oh.gotFriend) {
362       ObjectHandle myFriend = gcHeap.getObjectHandle(myFriendIndex);
363       Objeto_Indice oiFriend = new Objeto_Indice(myFriend,myFriendIndex);
364       //Si el objeto Amigo está coloreado de blanco, lo coloreamos de gris y lo añadimos
365       //a la lista de objetos por los que debe pasar el recorrido en anchura
366       if ((myFriend.myColor == Color.white)){
367         myFriend.myColor=Color.gray;
368         myFriend.myFriendLineColor=Color.gray;
369         myFriend.myLunchLineColor=Color.gray;
370         myFriend.mySnackLineColor=Color.gray;
371         gcHeap.listaPendientes.add(oiFriend);
372       }
```

```
373       }
374       //Se calcula la posicion del objeto Comida de oh
375       int myLunchIndex = gcHeap.getObjectPool(oh.objectPos+1);
376       //Si esa posición no es vacía, es decir, si tiene Comida
377       if (oh.gotLunch) {
378         ObjectHandle myLunch = gcHeap.getObjectHandle(myLunchIndex);
379         Objeto_Indice oiLunch = new Objeto_Indice(myLunch,myLunchIndex);
380         //Si el objeto Comida está coloreado de blanco, lo coloreamos de gris y lo
381         //añadimos a la lista de objetos por los que debe pasar el recorrido en anchura
382         if ((myLunch.myColor == Color.white)){
383           myLunch.myColor=Color.gray;
384           myLunch.myFriendLineColor=Color.gray;
385           myLunch.myLunchLineColor=Color.gray;
386           myLunch.mySnackLineColor=Color.gray;
387           gcHeap.listaPendientes.add(oiLunch);
388         }
389       }
390      //Se calcula la posicion del objeto Aperitivo de oh
391       int mySnackIndex = gcHeap.getObjectPool(oh.objectPos+2);
392       //Si esa posición no es vacía, es decir, si tiene Comida
393       if (oh.gotSnack) {
394         ObjectHandle mySnack = gcHeap.getObjectHandle(mySnackIndex);
395         Objeto_Indice oiSnack = new Objeto_Indice(mySnack,mySnackIndex);
396         //Si el objeto Aperitivo está coloreado de blanco, lo coloreamos de gris y lo
397         //añadimos a la lista de objetos por los que debe pasar el recorrido en anchura
398         if ((mySnack.myColor == Color.white)){
399           mySnack.myColor=Color.gray;
400           mySnack.myFriendLineColor=Color.gray;
401           mySnack.myLunchLineColor=Color.gray;
402           mySnack.mySnackLineColor=Color.gray;
403           gcHeap.listaPendientes.add(oiSnack);
404         }
405       }
406       oh.myColor=Color.black;
407       oh.myFriendLineColor=Color.black;
408       oh.myLunchLineColor=Color.black;
409       oh.mySnackLineColor=Color.black;
410       boolean hecho = false;
411       //Se actualiza el siguiente objeto a visitar
412       if(gcHeap.listaPendientes.isEmpty()) hecho = true;
413       else{
414         Objeto_Indice siguiente = (Objeto_Indice)gcHeap.listaPendientes.elementAt(0);
415         int indiceSig = siguiente.dameIndice();
416         currentFishBeingMarked=indiceSig;
417         gcHeap.listaPendientes.remove(siguiente);
418       }
419       return hecho;
420     }
421
422     //Metodo que colorea todos los objectos de color blanco, preparandolos para una
423     //futura ejecución
424     public void resetGCState() {
425       for (int i = 0; i < gcHeap.getHandlePoolSize(); ++i) {
426         //Se recorre el vector de objetos y si está ocupado se colorea de blanco el objeto y las
427         //líneas
428         ObjectHandle oh = gcHeap.getObjectHandle(i + 1);
429         if (!oh.free) {
430           oh.myColor = Color.white;
431           oh.previousNodeInGCTraversalIsAFish = false;
432           oh.previousFishInGCTraversal = 0;
```

```
433         oh.myFriendLineColor = Color.white;
434         oh.myLunchLineColor = Color.white;
435         oh.mySnackLineColor = Color.white;
436         oh.leido=false;
437       }
438     }
439     currentGCState = garbageCollectorHasNotStarted;
440     fishAreBeingMarked = false;
441     yellowFishLocalVarIsCurrentGCMarkNode = false;
442     blueFishLocalVarIsCurrentGCMarkNode = false;
443     redFishLocalVarIsCurrentGCMarkNode = false;
444     yellowFishLocalVarLineColor = Color.white;
445     blueFishLocalVarLineColor = Color.white;
446     redFishLocalVarLineColor = Color.white;
447     indiceNuevosObjetos=0;
448     gcHeap.listaNuevosObjetos.clear();
449     gcHeap.listaPendientes.clear();
450   }
451
452   //Bucle principal del recolector
453   public void seguidoGC() {
454     //Se preparan todos los objetos para la ejecución
455     resetGCState();
456     //Se ejecuta mientras no se pulse el botón detener o no se haya terminado
457     while(!parar && (currentGCState!=garbageCollectorIsDone)){
458       nextGCStep();
459       repaint();
460       try {
461         thread.sleep(1000);
462         }  catch (InterruptedException e) {}
463     }
464     resetGCState();
465   }
466
467   public void pausa() {
468     parar=!parar;
469   }
470
471   public void start() {
472     thread = new Thread(this);
473     thread.start();
474   }
475
476   public void run() {
477     seguidoGC();
478   }
479
480   //Metodo que borra todas las relaciones entre peces
481   public synchronized void borraReferencias(){
482     for(int i=0;i<gcHeap.dameObjectPool().length;i++){
483       gcHeap.setObjectPool(i,0);
484     }
485     localVars.yellowFish=0;
486     localVars.blueFish=0;
487     localVars.redFish=0;
488     repaint();
489   }
490 }
```

## *4. GarbageCheckBoxGroup*

Clase que contiene los checkbox para elegir entre búsqueda en profundidad y búsqueda en anchura. El checkbox *cb_Anchura* (línea 5) se pulsará para elegir el recorrido en anchura. El checkbox *cb_Profundidad* (línea 7) se pulsará para elegir el recorrido en profundidad.

```
1  public class GarbageCollectCheckboxGroup extends Panel{
2    //Grupo de checkbox
3    private CheckboxGroup cbg = new CheckboxGroup();
4    //Checkbox para elegir el recorrido en anchura
5    Checkbox cb_Anchura = new Checkbox(HeapOfFishStrings.anchura, cbg, true);
6    //Checkbox para elegir el recorrido en profundidad
7    Checkbox cb_Profundidad = new Checkbox(HeapOfFishStrings.profundidad, cbg,       false);
8
9  //Se añaden los check box al panel
10 GarbageCollectCheckboxGroup() {
11     setLayout(new GridLayout(2, 1));
12     Panel p = new Panel();
13      setBackground(Color.lightGray);
14     p.setLayout(new GridLayout());
15     p.add(cb_Anchura);
16     add(p);
17     p = new Panel();
18     p.setLayout(new GridLayout());
19     p.add(cb_Profundidad);
20     add(p);
21   }
22 }
```

## *5. GarbageCollectPanel*

Clase encargada de realizar las acciones correspondientes a los botones que se pulsan desde la pantalla de recolección de memoria.

Atributos
El atributo *gcCanvas* de tipo *GarbageCollectCanvas* (línea 3) es el algoritmo recolector de memoria.
El atributo *lector* de tipo *Lector* (línea 5) recorre el grafo a la vez que el recolector coloreando de verde los ojos de los peces que visita.
El atributo *refIni* de tipo *ReferenciasIniciales* (línea 7) crea las relaciones de forma automática entre peces.
El atributo *gcbp* de tipo *GarbageCollectButtonPanel* (línea 9) son los botones y checkbox de la pantalla de recolección.

Métodos
El método *boolean action(Event evt, Object arg)* (línea 27) controla los botones pulsados desde la pantalla de recolección de memoria y realiza las acciones correspondientes. A continuación se muestra el pseudocódigo.
- Si se ha pulsado un botón (*evt.target instanteof Button*) entonces vemos que botón se ha pulsado
- Si se ha pulsado el botón *inicio* (línea 69), comienza el recolector de memoria. Se activan el botón *pausa* y el *lector* y se desactivan el botón de *creación de relaciones automáticas* y el checkbox de *elección de tipo de recorrido*.
- Si se ha pulsado el botón *pausa* y el algoritmo estaba parado (línea 53), se reauda. Si se ha pulsado el botón *pausa* y el algoritmo no estaba parado (línea 39), se para. Los hilos que se paran o se reanudan son todos los que intervienen en el programa, es decir el recolector de memoria, el lector y la creación automática de relaciones. Además hay que capturar las excepciones que se lanzarían al intentar detener un hilo que no está en ejecución o al intentar reanudar un hilo que ya ha sido reanudado.
- Si se ha pulsado el botón *lector* (línea 88), se ejecuta el hilo lector y se inactiva el botón lector.
- Si se ha pulsado el botón de *creación de relaciones* (línea 97), se muestra un cuadro de diálogo en que pide introducir un número que codifica el ejemplo a ejecutar.
- Si se ha pulsado el botón *reset* (línea 109), se prepara la aplicación para una nueva ejecución.

- Si se ha pulsado un checkbox se comprueba que checkbox se ha pulsado
- Si se ha pulsado el checkbox de anchura (línea 125), se marca que se desea un recorrido en anchura. Análogamente para el recorrido en profundidad (línea 128).

```
1  class GarbageCollectPanel extends Panel {
2     //Algoritmo recolector de memoria
3     private GarbageCollectCanvas gcCanvas;
4     //Lector
5     private Lector lector;
6     //Atributo para crear las relaciones entre peces automáticamente
7     private ReferenciasIniciales refIni;
8     //Botones y grupo de checkbox
9     private GarbageCollectButtonPanel gcbp = new GarbageCollectButtonPanel();
10
11    GarbageCollectPanel(GCHeap heap, LocalVariables locVars, HeapOfFishTextArea ta) {
12       setBackground(Color.blue);
13       setLayout(new BorderLayout());
14       gcCanvas = new GarbageCollectCanvas(heap, locVars, ta);
```

```
15          lector = new Lector(heap, locVars, ta);
16          refIni = new ReferenciasIniciales(heap,gcCanvas,locVars);
17          add("South", gcbp);
18          add("Center", gcCanvas);
19       }
20
21       public void resetGCState() {
22          gcCanvas.resetGCState();
23       }
24
25       //Método que controla los botones pulsados desde la pantalla del Recolector
26       //y realiza las acciones correspondientes dependiendo del botón pulsado
27       public boolean action(Event evt, Object arg) {
28         if (evt.target instanceof Button) {
29          String bname = (String) arg;
30          //Botón pausa: si el algoritmo está parado, se reanuda y si no lo está lo para, además de
31          //detener la ejecución del recolector de memoria también detiene y reanuda a todos los
32          //hilos en ejecución, es decir al lector y la creación automática de relaciones. Es necesario
33          //capturar las excepciones que se lanzarían al intentar detener un thread que no está en
34          //ejecución o al intentar reanudar un thread que ya ha sido reanudado
35          if
36    (bname.equals(HeapOfFishStrings.pausa)||((bname.equals(HeapOfFishStrings.reanuda)))) {
37            gcCanvas.pausa();
38            //Si está en ejecución, se detienen
39            if(gcCanvas.dameParar() == true){
40              try{
41                gcCanvas.dameThread().suspend();
42              } catch (Exception e) {}
43              try{
44                refIni.dameHilo().suspend();
45              } catch (Exception e) {}
46              try{
47                lector.dameLector().suspend();
48              } catch (Exception e) {}
49              gcbp.b_Pausa.setLabel(HeapOfFishStrings.reanuda);
50              gcCanvas.damePanelTextArea().setText(HeapOfFishStrings.detenido);
51            }
52            //Si no está en ejecución, se reanudan
53            else{
54              try{
55                gcCanvas.dameThread().resume();
56              } catch (Exception e) {}
57              try{
58                refIni.dameHilo().resume();
59              } catch (Exception e) {}
60              try{
61                lector.dameLector().resume();
62              } catch (Exception e) {}
63              gcbp.b_Pausa.setLabel(HeapOfFishStrings.pausa);
64              gcCanvas.damePanelTextArea().setText(HeapOfFishStrings.reanudado);
65            }
66          }
67          //Botón inicio: Comienza el algoritmo recolector de memoria y activa o desactiva los
68          //botones
69          else if (bname.equals(HeapOfFishStrings.inicio)) {
70            gcCanvas.start();
71            //El botón pausa permanece inactivo hasta que empieza la ejecución del recolector
72            gcbp.b_Pausa.setEnabled(true);
73            //Se activa el botón lector
74            gcbp.b_Lector.setEnabled(true);
```

```
75          //Las referencias automáticas solo pueden empezar a ejecutarse antes de que comience
76          //la ejecución del recolector, cuando empieza el recolector, se desactiva el botón de
77          //referencias
78          gcbp.b_ref.setEnabled(false);
79          //Si el recolector se está ejecutando, no se puede cambiar el tipo de recorrido, por lo que
80          //se desactiva el checbox
81          gcbp.gcdbg.cb_Anchura.setEnabled(false);
82          gcbp.gcdbg.cb_Profundidad.setEnabled(false);
83          //Se desactiva el botón de inicio
84          gcbp.b_Inicio.setEnabled(false);
85        }
86        //Botón lector: Se ejecuta el thread lector, que lee el grafo de dependencias mientras el
87        //recolector se ejecuta
88        else if (bname.equals(HeapOfFishStrings.lector)) {
89          lector.start();
90          //Se inactiva el botón lector
91          gcbp.b_Lector.setEnabled(false);
92          //Se muestra un mensaje en el cuadro de texto informando del comienzo del lector
93          gcCanvas.damePanelTextArea().setText(HeapOfFishStrings.lector_alg);
94        }
95        //Botón crear referencias: Se muestra un mensaje de diálogo en el que pide introducir un
96        //número, en función de dicho número se ejecutan los diferentes ejemplos
97        else if (bname.equals(HeapOfFishStrings.refIni)) {
98          String aux=JOptionPane.showInputDialog(this,"Elija el ejemplo a ejecutar.","Heap Of
99  Fish",JOptionPane.QUESTION_MESSAGE);
100         if( aux!=null){
101           refIni.ponTipoEjemplo(Integer.parseInt(aux));
102         }
103         refIni.start();
104         gcbp.b_ref.setEnabled(false);
105         gcCanvas.damePanelTextArea().setText(HeapOfFishStrings.refIni_alg);
106       }
107       //Botón reset: Prepara la aplicación para que pueda volverse a ejecutar el algoritmo
108       //recolector, activa y desactiva los botones para dejarlos como al principio
109       else if (bname.equals(HeapOfFishStrings.reset)) {
110         gcCanvas.borraReferencias();
111         gcbp.b_Pausa.setEnabled(false);
112         gcbp.b_Lector.setEnabled(false);
113         gcbp.b_ref.setEnabled(true);
114         gcbp.gcdbg.cb_Anchura.setEnabled(true);
115         gcbp.gcdbg.cb_Profundidad.setEnabled(true);
116         gcbp.b_Inicio.setEnabled(true);
117         gcCanvas.damePanelTextArea().setText(HeapOfFishStrings.reset_alg);
118       }
119     }
120     //Checkbox para elegir entre recorrido en anchura y profundidad
121     if (evt.target instanceof Checkbox){
122       Checkbox cb = (Checkbox) evt.target;
123       String cbname = cb.getLabel();
124       //La anchura se codifica como 1 y la profundidad como 0
125       if (cbname.equals(HeapOfFishStrings.anchura)) {
126         gcCanvas.dameGCHeap().AnchOProf=1;
127       }
128       if (cbname.equals(HeapOfFishStrings.profundidad)) {
129         gcCanvas.dameGCHeap().AnchOProf=0;
130       }
131     }
132     return true;
133   }
134 }
```

## 6. GCHeap

Clase que implementa el heap y contiene todos los métodos necesarios para su mantenimiento.

Atributos nuevos
Se añade el atributo *listaNuebosObjetos* de tipo *Vector* (línea 10), en este vector se almacenan los objetos destino de las relaciones creadas una vez que el algoritmo recolector ha comenzado su ejecución.
Se añade el atributo *listaNuebosObjetos* de tipo *Vector* (línea 14), este atributo se usa para el recorrido en anchura del grafo de relaciones.
El atributo *anchOProf* de tipo entero (línea 18) codifica el tipo de recorrido.

```
1   public class GCHeap {
2
3       private int handlePoolSize;
4       private int objectPoolSize;
5       protected ObjectHandle[] handlePool;
6       private int[] objectPool;
7
8       //En este vector se almacenan los objetos creados una vez que el recolector ha comenzado
9       //su ejecución
10      protected Vector listaNuevosObjetos;
11
12      //Vector en el que se almacenan los nodos pendientes, es necesario para el recorrido en
13      //anchura
14      protected Vector listaPendientes;
15
16      //Atributo que contiene el tipo de recorrido
17      //Codificación: AnchOProf = 0 en profundidad y AnchOProf = 1 en anchura
18      protected int AnchOProf=1;
19
20      GCHeap(int hndlPoolSize, int objPoolSize) {
21
22          handlePoolSize = hndlPoolSize;
23          objectPoolSize = objPoolSize;
24
25          objectPool = new int[objectPoolSize];
26          handlePool = new ObjectHandle[handlePoolSize];
27
28          // Initialize the object pool by putting a header at the zeroeth int location
29          // that indicates that the entire remainder of the object pool array is one
30          // big contiguous available memory block.
31          objectPool[0] = formMemBlockHeader(true, objectPoolSize);
32
33          for (int i = 0; i < handlePoolSize; ++i) {
34              handlePool[i] = new ObjectHandle();
35              handlePool[i].free = true;
36          }
37          listaNuevosObjetos = new Vector();
38          listaPendientes = new Vector();
39      }
40
41      // Returns number of handles in handlePool (2 ints each)
42      public int getHandlePoolSize() {
43          return handlePoolSize;
44      }
```

```
45
46      // Returns number of ints in object pool
47      public int getObjectPoolSize() {
48          return objectPoolSize;
49      }
50
51      public ObjectHandle getObjectHandle(int i) {
52          return handlePool[i - 1];   // Subtract one because zero is used to indicate null,
53                                      // so allocateHandle() adds 1 to the index.
54      }
55
56      public int getObjectPool(int i) {
57          return objectPool[i];
58      }
59
60      public void setObjectPool(int i, int value) {
61          objectPool[i] = value;
62      }
63
64      // formMemBlockHeader() makes an int that contains two pieces of information,
65      // the length of the memory block and whether the memory block is free. The int
66      // is the header for the block and is stored immediately before the block in
67      // the objectPool array. These headers form a kind of linked list of memory blocks
68      // because you can always jump to the next header by adding length to the index
69      // of the current blocks header to get index to the next block's header. The length
70      // variable is in units of ints; it tells how many ints long the memory block is,
71      // including the header int.
72      //
73      // The zeroeth bit of a memory block header is used to indicate freeness. If the bit
74      // is one, the memory block is free. Bits 1 through 31 are the memory block's length
75      // in number of ints.
76      private int formMemBlockHeader(boolean free, int length) {
77          int retVal = length << 1;
78          if (free) {
79              retVal |= 1;
80          }
81          return retVal;
82      }
83
84      // The length of a memory block includes the header int.
85      public int getMemBlockLength(int header) {
86          return header >> 1;
87      }
88
89      public boolean getMemBlockFree(int header) {
90          boolean retVal = false;
91          if ((header & 0x1) == 0x1) {
92              retVal = true;
93          }
94          return retVal;
95      }
96
97      // allocateObject() returns an int which is an index into the handlePool array. This is,
98      // in effect, a handle. Because a handle of zero indicates a null handle, the zeroeth int
99      // of the handlePool array is unused. The index returned by this function is a handle to
100     // the object.
101     //
102     //      bytesNeeded -- number of bytes needed by the object for which memory is being
103     //          allocated.
104     public int allocateObject(int bytesNeeded, FishIcon fish) {
```

```
105
106         // Initialize the return value to zero, which indicates not enough memory was
107         // available for the new object.
108         int retVal = 0;
109
110         // Because the objectPool is an array of ints, all objects are int aligned. Calculate
111         // the number of ints required by the new object based on the passed number of bytes
112         // required. This is done by adding 3 (sizeof(int) - 1) to the bytesNeeded and
113         // dividing by 4 (sizeof(int)). So one, two, three, and four byte objects will require
114         // one int. Five, six, seven, and eight byte objects will require two ints, etc...
115         int intsNeeded = (bytesNeeded + 3) / 4;
116
117         int i = 0;
118         while (i < objectPoolSize) {
119            int header = objectPool[i];
120            boolean free = false;
121            if (getMemBlockFree(header)) {
122               free = true;
123            }
124            int length = getMemBlockLength(header);
125
126            if (!free) {
127
128               // This memory block is not free, so continue by looking at the next memory
129               // block. Add length to the current index into the constant pool to get the
130               // index of the next memory block header.
131               i += length;
132               continue;
133            }
134
135            // We have found a free block. First, concatenate any other free blocks that may
136            // be contiguous to this one.
137            while ((i + length) < objectPoolSize && getMemBlockFree(objectPool[i + length])) {
138
139               // The next memory block is also free, so concatenate with the previous
140               // one. This is done by extending the size of the previous memory block
141               // to include the next block.
142               length += getMemBlockLength(objectPool[i + length]);
143               objectPool[i] = formMemBlockHeader(true, length);
144            }
145
146            // Now that we have a free block, check to see if it is big enough to hold
147            // the object for which the memory was requested.
148            if (length - 1 < intsNeeded) {
149               // This memory block is free, but not big enough for the fat object for
150               // which memory has been requested. So continue by looking at the next
151               // memory block. Add length to the current index into the constant pool
152               // to get the index of the next memory block header.
153               i += length;
154               continue;
155            }
156
157            // The current free block is big enough for our new object. First, check to see
158            // if there is more than enough memory in this block than that actually required
159            // by the new object.
160            int extraMem = length - intsNeeded - 1;
161            if (extraMem > 0) {
162
163               // The free block has more than enough memory, so we must split it into
164               // two blocks. The first block will be exactly the right size for our
```

```
165                 // new object. The second will contain whatever is leftover. This splitting
166                 // is accomplished by changing the length of the original header to exactly
167                 // equal whatever is required by the new object, then adding a second
168                 // header for the leftover memory.
169                 objectPool[i] = formMemBlockHeader(true, intsNeeded + 1);
170                 objectPool[i + intsNeeded + 1] = formMemBlockHeader(true, extraMem);
171             }
172
173             // At this point objectPool[i] is exactly the right size for the new object.
174             // All that we need to do now is allocate the handle to the memory. Add one
175             // to i before passing to allocHandle, because i currently points to the header.
176             // The object handle should point past the header at the start of the object
177             // itself.
178             retVal = allocateHandle(i + 1, fish);
179             if (retVal > 0) {
180                 objectPool[i] = formMemBlockHeader(false, intsNeeded + 1);
181             }
182             break;
183         }
184         return retVal;
185     }
186
187     // allocateHandle() returns an int which is an index into the handlePool array. This is,
188     // in effect, a handle. Because a handle of zero indicates a null handle, the zeroeth int
189     // of the handlePool array is unused.
190     public int allocateHandle(int objectHandle, FishIcon fishHandle) {
191         int retVal = 0;
192         for (int i = 0; i < handlePoolSize; ++i) {
193             if (handlePool[i].free) {
194                 handlePool[i].free = false;
195                 handlePool[i].objectPos = objectHandle;
196                 handlePool[i].fish = fishHandle;
197                 retVal = i + 1; // Add one because zero means failure.
198                 break;
199             }
200         }
201         return retVal;
202     }
203
204     public void freeObject(int handle) {
205         if (!handlePool[handle - 1].free) {
206             handlePool[handle - 1].free = true;
207             handlePool[handle - 1].fish = null;
208             int objectTableIndex = handlePool[handle - 1].objectPos;
209             int header = getObjectPool(objectTableIndex - 1);
210             int length = getMemBlockLength(header);
211             header = formMemBlockHeader(true, length);
212             setObjectPool(objectTableIndex - 1, header);
213         }
214     }
215
216     // Returns true if an object was slid
217     public boolean slideNextNonContiguousObjectDown() {
218
219         boolean retVal = false;
220
221         int i = 0;
222         while (i < objectPoolSize) {
223             int header = objectPool[i];
224             boolean free = false;
```

```
225             if (getMemBlockFree(header)) {
226                 free = true;
227             }
228             int length = getMemBlockLength(header);
229 
230             if (!free) {
231 
232                 // This memory block is not free, so continue by looking at the next memory
233                 // block. Add length to the current index into the constant pool to get the
234                 // index of the next memory block header.
235                 i += length;
236                 continue;
237             }
238 
239             // We have found a free block. First, concatenate any other free blocks that may
240             // be contiguous to this one.
241             while ((i + length) < objectPoolSize && getMemBlockFree(objectPool[i + length])) {
242 
243                 // The next memory block is also free, so concatenate with the previous
244                 // one. This is done by extending the size of the previous memory block
245                 // to include the next block.
246                 length += getMemBlockLength(objectPool[i + length]);
247                 objectPool[i] = formMemBlockHeader(true, length);
248             }
249 
250             // See if an object exists at the next location, if not break out of the loop
251             // and return false. There are no other objects that can be slid.
252             if (i + length >= objectPoolSize) {
253                 break;
254             }
255 
256             int sliderLength = getMemBlockLength(objectPool[i + length]);
257             for (int j = 1; j < sliderLength; ++j) {
258                 objectPool[i + j] = objectPool[i + length + j];
259             }
260 
261             objectPool[i] = formMemBlockHeader(false, sliderLength);
262             objectPool[i + sliderLength] = formMemBlockHeader(true, length);
263 
264             for (int j = 0; j < handlePoolSize; ++j) {
265                 if (!handlePool[j].free && handlePool[j].objectPos == i + length + 1) {
266                     handlePool[j].objectPos = i + 1;
267                     break;
268                 }
269             }
270 
271             retVal = true;
272             break;
273         }
274         return retVal;
275     }
276 }
```

## *7. HeapOfFish*

Clase principal del programa, la que contiene los métodos *init()* y *start()* del Applet (líneas14 y 23 respectivamente). También contiene el método *boolean action(Event evt, Object arg)* (línea 31) que detecta cuando se ha pulsado en el checkbox de navegación por las pantallas y realiza las acciones correspondientes para el cambio de una pantalla a otra.

```
1  public class HeapOfFish extends Applet {
2
3     //Heap
4     private GCHeap gcHeap = new GCHeap(15, 50);
5     //Objetos Raíz
6     private LocalVariables localVars = new LocalVariables();
7
8     private HeapOfFishControlPanel controlPanel = new HeapOfFishControlPanel();
9     private HeapOfFishCanvases canvases = new HeapOfFishCanvases(gcHeap, localVars,
10 controlPanel.getTextArea());
11    private String currentHeapOfFishMode = HeapOfFishStrings.swim;
12
13    //Inicialización del Applet
14    public void init() {
15       super.init();
16       setBackground(Color.cyan);
17       setLayout(new BorderLayout(5, 5));
18       add("North", new ColoredLabel("HEAP OF FISH", Label.CENTER, Color.white));
19       add("South", controlPanel);
20       add("Center", canvases);
21    }
22
23    public void start() {
24       canvases.start();
25    }
26
27    public Insets insets() {
28       return new Insets(5, 5, 5, 5);
29    }
30
31    //Método que detecta que se ha pulsado en el checkbox para cambiar de pantalla
32    public boolean action(Event evt, Object arg) {
33       if (evt.target instanceof Checkbox) {
34          Checkbox cb = (Checkbox) evt.target;
35          String cbname = cb.getLabel();
36          //Se cambia a la pantalla de creación de objetos
37          if (cbname.equals(HeapOfFishStrings.allocateFish)) {
38             if (!currentHeapOfFishMode.equals(HeapOfFishStrings.allocateFish)) {
39                controlPanel.getTextArea().setText(HeapOfFishStrings.allocateFishInstructions);
40                canvases.setMode(HeapOfFishStrings.allocateFish);
41             }
42          }
43          //Se cambia a la pantalla de creación de relaciones entre objetos
44          else if (cbname.equals(HeapOfFishStrings.assignReferences)) {
45             if (!currentHeapOfFishMode.equals(HeapOfFishStrings.assignReferences)) {
46                canvases.setMode(HeapOfFishStrings.assignReferences);
47             }
48          }
49          //Se cambia a la pantalla del algoritmo recolector de memoria
50          else if (cbname.equals(HeapOfFishStrings.garbageCollect)) {
```

```
51                  if (!currentHeapOfFishMode.equals(HeapOfFishStrings.garbageCollect)) {
52                     controlPanel.getTextArea().setText(HeapOfFishStrings.garbageCollectInstructions);
53                      canvases.setMode(HeapOfFishStrings.garbageCollect);
54                  }
55              }
56              //Se cambia a la pantalla de compactación del heap
57              else if (cbname.equals(HeapOfFishStrings.compactHeap)) {
58                  if (!currentHeapOfFishMode.equals(HeapOfFishStrings.compactHeap)) {
59                      controlPanel.getTextArea().setText(HeapOfFishStrings.compactHeapInstructions);
60                      canvases.setMode(HeapOfFishStrings.compactHeap);
61                  }
62              }
63              //Se cambia a la pantalla principal
64              else if (cbname.equals(HeapOfFishStrings.swim)) {
65                  if (!currentHeapOfFishMode.equals(HeapOfFishStrings.swim)) {
66                      controlPanel.getTextArea().setText("");
67                      canvases.setMode(HeapOfFishStrings.swim);
68                  }
69              }
70              currentHeapOfFishMode = cbname;
71              canvases.repaint();
72          }
73          return true;
74      }
75  }
```

## *8. Lector*

Clase que contiene el método que va accediendo a objetos del heap y colorea sus ojos de verde mientras el recolector de memoria se está ejecutando. El algoritmo del lector es parecido al del recolector. Para visitar los peces se utiliza un recorrido en anchura.

Atributos más importantes
Desde la línea 13 hasta la 31 se declara la codificación de los estados posibles del algoritmo lector.
El atributo *currentGCState* de tipo entero (línea 33) codifica el estado en el que se encuentra el recolector, inicialmente el estado será "detenido"

Métodos más importantes
El método *seguidoGC()* (línea 64) es el bucle principal del lector, mientras el algoritmo no esté en estado detenido, se sigue ejecutando el bucle. En cada iteración del bucle se hace una llamada al método *nextGCStep()*.
El método *resetGCState()* (línea 76) colorea todos los peces de color blanco, preparando el grafo para una nueva ejecución.
El método *nextGCStep()* (línea 86) es el método que implementa el lector, el algoritmo va pasando por una serie de estados y en cada estado debe hacer una serie de acciones. A continuación se muestra el pseudocódigo:
Si el estado del algoritmo es igual a:

**- *detenido***: el algoritmo no ha comenzado.
Pasar al estado *visitando raíz amarilla.*
(línea 89)

**- *visitando raíz amarilla***: el algoritmo marca como visitada la raíz amarilla y pasa al estado *visitando peces desde raíz amarilla.*
(línea 94)

**- *visitando peces desde raíz amarilla***: se recorren todos los peces del árbol cuya raíz es la raíz amarilla coloreando de verde los ojos de los peces que visita, cuando ha terminado de recorrer todos los nodos pasa al estado *visitando raíz azul.*
(línea 112)

**- *visitando raíz azul:*** el algoritmo marca como visitada la raíz azul y pasa al estado *visitando peces desde raíz azul.*
(línea 132)

**- *visitando peces desde raíz azul***: se recorren todos los peces del árbol cuya raíz es la raíz azul coloreando de verde los ojos de los peces que visita, cuando ha terminado de recorrer todos los nodos pasa al estado *visitando raíz roja.*
(línea 149)

**- *visitando raíz roja***: el algoritmo marca como visitada la raíz roja y pasa al estado *visitando peces desde raíz roja.*
(línea 167)

**- *visitando peces desde raíz roja***: se recorren todos los peces del árbol cuya raíz es la raíz roja coloreando de verde los ojos de los

peces que visita, cuando ha terminado de recorrer todos los nodos pasa al estado *pasando por nuevos nodos.*
(línea 182)

***- pasando por nuevos nodos:*** se recorren los nodos que se han ido añadiendo a la nueva estructura de datos y se pasa al estado *borrando peces no alcanzados.*
(línea 200)

***- borrando peces no alcanzados:*** se borran todos los peces que no han sido alcanzados, es decir los coloreados como blancos y se pasa al estado detenido.
(línea 212)

El método booleano *traverseNextFishNode(ObjectHandle oh)* (línea 223) recorre en anchura el grafo, tomando como raíz *oh*. Devuelve verdadero cuando ha terminado el recorrido y falso en caso contrario. A continuación se muestra el psudocódigo.
- Si *oh* tiene una relación de Amistad y el pez destino de dicha relación no ha sido visitado por el lector, se añade a *listaPendientes* (línea 228).
- Si *oh* tiene una relación de Comida, y el pez destino de dicha relación no ha sido visitado por el lector, se añade a *listaPendientes* (línea 239).
- Si *oh* tiene una relación de Aperitivo, y el pez destino de dicha relación no ha sido visitado por el lector, se añade a *listaPendientes* (línea 250).
- Una vez que ha visitado todos los hijos, colorea de verde el ojo de *oh*, calcula cual es el siguiente objeto a visitar (el siguiente a *oh* en *listaPendiente*) y devuelve verdadero (línea 258).

```
1  public class Lector implements Runnable{
2
3    //Recolector de meoria
4    private GarbageCollectCanvas gc;
5    //Hilo
6    private Thread lector;
7    //Vector donde se almacenan los objetos que debe visitar el recorrido en anchura
8    private Vector listaPendientes = new Vector();
9    //Indice para recorrer el vector de nuevos objetos
10   private int indiceNuevosObjetos=0;
11
12   // Estados del algoritmo lector
13   private final int garbageCollectorHasNotStarted = 0;
14
15   private final int startingAtYellowLocalVariableRoot = 1;
16   private final int traversingFromYellowLocalVariableRoot = 2;
17   private final int doneWithYellowLocalVariableRoot = 3;
18
19   private final int startingAtBlueLocalVariableRoot = 4;
20   private final int traversingFromBlueLocalVariableRoot = 5;
21   private final int doneWithBlueLocalVariableRoot = 6;
22
23   private final int startingAtRedLocalVariableRoot = 7;
24   private final int traversingFromRedLocalVariableRoot = 8;
25   private final int doneWithRedLocalVariableRoot = 9;
26
27   private final int readyToSweepUnmarkedFish = 10;
28   private final int doneSweepingUnmarkedFish = 11;
29
30   private final int garbageCollectorIsDone = 12;
```

```
31    private final int pasarPorListaDeNuevos = 13;
32
33    private int currentGCState = garbageCollectorHasNotStarted;
34
35    private boolean fishAreBeingMarked;
36    private int currentFishBeingMarked;
37
38    private boolean yellowFishLocalVarIsCurrentGCMarkNode;
39    private boolean blueFishLocalVarIsCurrentGCMarkNode;
40    private boolean redFishLocalVarIsCurrentGCMarkNode;
41
42    private Color yellowFishLocalVarLineColor;
43    private Color blueFishLocalVarLineColor;
44    private Color redFishLocalVarLineColor;
45
46    public Thread dameLector(){
47      return lector;
48    }
49
50    public Lector(GCHeap heap, LocalVariables locVars, HeapOfFishTextArea ta) {
51      gc = new GarbageCollectCanvas(heap, locVars, ta);
52    }
53
54    public void start() {
55      lector = new Thread(this);
56      lector.start();
57    }
58
59    public void run() {
60      seguidoGC();
61    }
62
63    //Bucle principal del algoritmo lector
64    public void seguidoGC() {
65      while(currentGCState!=garbageCollectorIsDone){
66        nextGCStep();
67        gc.repaint();
68        try {
69          lector.sleep(500);
70        }  catch (InterruptedException e) {}
71      }
72      resetGCState();
73    }
74
75    //Inicializa los objetos para futuras ejecuciones
76    public void resetGCState() {
77      currentGCState = garbageCollectorHasNotStarted;
78      fishAreBeingMarked = false;
79      yellowFishLocalVarIsCurrentGCMarkNode = false;
80      blueFishLocalVarIsCurrentGCMarkNode = false;
81      redFishLocalVarIsCurrentGCMarkNode = false;
82      listaPendientes.clear();
83    }
84
85    //Estados del algoritmo lector
86    public synchronized void nextGCStep(){
87      switch (currentGCState) {
88        //No comenzado
89        case garbageCollectorHasNotStarted:
90          yellowFishLocalVarIsCurrentGCMarkNode = true;
```

```
91          currentGCState = startingAtYellowLocalVariableRoot;
92          break;
93        //Visitando raíz amarilla
94        case startingAtYellowLocalVariableRoot:
95          yellowFishLocalVarIsCurrentGCMarkNode = false;
96          if (gc.dameLocalVars().yellowFish != 0) {
97            ObjectHandle oh = gc.dameGCHeap().getObjectHandle(gc.dameLocalVars().yellowFish);
98            yellowFishLocalVarIsCurrentGCMarkNode = false;
99            //Se marca como leido, el objeto que se está tratando, posteriormente a los objetos que
100           //cumplan leido=true, se les coloreará el ojo de verde
101           oh.leido=true;
102           currentFishBeingMarked = gc.dameLocalVars().yellowFish;
103           fishAreBeingMarked = true;
104           currentGCState = traversingFromYellowLocalVariableRoot;
105         }
106         else {
107           blueFishLocalVarIsCurrentGCMarkNode = true;
108           currentGCState = startingAtBlueLocalVariableRoot;
109         }
110         break;
111       //Visitando objetos que cuelgan de la raíz amarilla
112       case traversingFromYellowLocalVariableRoot:
113         ObjectHandle oh2 = gc.dameGCHeap().getObjectHandle(currentFishBeingMarked);
114         boolean doneWithThisTree;
115         doneWithThisTree = traverseNextFishNode(oh2);
116         if (doneWithThisTree) {
117           ObjectHandle oh = gc.dameGCHeap().getObjectHandle(gc.dameLocalVars().yellowFish);
118           //Se marca como leido, el objeto que se está tratando
119           oh.leido=true;
120           fishAreBeingMarked = false;
121           yellowFishLocalVarIsCurrentGCMarkNode = true;
122           currentGCState = doneWithYellowLocalVariableRoot;
123         }
124         break;
125       //Terminado el tratamiento de la raíz amarilla
126       case doneWithYellowLocalVariableRoot:
127         yellowFishLocalVarIsCurrentGCMarkNode = false;
128         blueFishLocalVarIsCurrentGCMarkNode = true;
129         currentGCState = startingAtBlueLocalVariableRoot;
130         break;
131       //Visitando raíz azul
132       case startingAtBlueLocalVariableRoot:
133         blueFishLocalVarIsCurrentGCMarkNode = false;
134         if (gc.dameLocalVars().blueFish != 0) {
135           ObjectHandle oh = gc.dameGCHeap().getObjectHandle(gc.dameLocalVars().blueFish);
136           blueFishLocalVarIsCurrentGCMarkNode = false;
137           //Se marca como leido, el objeto que se está tratando
138           oh.leido=true;
139           currentFishBeingMarked = gc.dameLocalVars().blueFish;
140           fishAreBeingMarked = true;
141           currentGCState = traversingFromBlueLocalVariableRoot;
142         }
143         else {
144           redFishLocalVarIsCurrentGCMarkNode = true;
145           currentGCState = startingAtRedLocalVariableRoot;
146         }
147         break;
148       //Visitando objetos que cuelgan de la raíz azul
149       case traversingFromBlueLocalVariableRoot:
150         ObjectHandle oh3 = gc.dameGCHeap().getObjectHandle(currentFishBeingMarked);
```

```
151         doneWithThisTree = traverseNextFishNode(oh3);
152         if (doneWithThisTree) {
153           ObjectHandle oh = gc.dameGCHeap().getObjectHandle(gc.dameLocalVars().blueFish);
154           oh.leido=true;
155           fishAreBeingMarked = false;
156           blueFishLocalVarIsCurrentGCMarkNode = true;
157           currentGCState = doneWithBlueLocalVariableRoot;
158         }
159         break;
160       //Terminado el tratamiento de la raíz azul
161       case doneWithBlueLocalVariableRoot:
162         blueFishLocalVarIsCurrentGCMarkNode = false;
163         redFishLocalVarIsCurrentGCMarkNode = true;
164         currentGCState = startingAtRedLocalVariableRoot;
165         break;
166        //Visitando raíz roja
167       case startingAtRedLocalVariableRoot:
168         redFishLocalVarIsCurrentGCMarkNode = false;
169         if (gc.dameLocalVars().redFish != 0) {
170           ObjectHandle oh = gc.dameGCHeap().getObjectHandle(gc.dameLocalVars().redFish);
171           oh.leido=true;
172           redFishLocalVarLineColor = Color.gray;
173           currentFishBeingMarked = gc.dameLocalVars().redFish;
174           fishAreBeingMarked = true;
175           currentGCState = traversingFromRedLocalVariableRoot;
176         }
177         else {
178           currentGCState = pasarPorListaDeNuevos;
179         }
180         break;
181       //Visitando objetos que cuelgan de la raíz roja
182       case traversingFromRedLocalVariableRoot:
183         ObjectHandle oh4 = gc.dameGCHeap().getObjectHandle(currentFishBeingMarked);
184         doneWithThisTree = traverseNextFishNode(oh4);
185         if (doneWithThisTree) {
186           ObjectHandle oh = gc.dameGCHeap().getObjectHandle(gc.dameLocalVars().redFish);
187           oh.leido=true;
188           fishAreBeingMarked = false;
189           redFishLocalVarIsCurrentGCMarkNode = true;
190           currentGCState = doneWithRedLocalVariableRoot;
191         }
192         break;
193       //Terminado el tratamiento de la raíz roja
194       case doneWithRedLocalVariableRoot:
195         redFishLocalVarIsCurrentGCMarkNode = false;
196         currentGCState = pasarPorListaDeNuevos;
197         break;
198       //Pasando por nuevos objetos, objetos creados una vez que el algoritmo de recolección ha
199       //comenzado
200       case pasarPorListaDeNuevos:
201         if (indiceNuevosObjetos==gc.dameGCHeap().listaNuevosObjetos.size()){
202           currentGCState = doneSweepingUnmarkedFish;
203           break;
204         }
205         Objeto_Indice oi1 =
206 (Objeto_Indice)gc.dameGCHeap().listaNuevosObjetos.elementAt(indiceNuevosObjetos);
207         currentFishBeingMarked=oi1.dameIndice();
208         doneWithThisTree = traverseNextFishNode(oi1.dameOh());
209         indiceNuevosObjetos++;
210         break;
```

```
211       //Preparado para borrar objetos de color blanco
212       case doneSweepingUnmarkedFish:
213         currentGCState = garbageCollectorIsDone;
214         break;
215       //Terminado
216       case garbageCollectorIsDone:
217       default:
218         break;
219     }
220   }
221
222   //Método que recorre el grafo en anchura
223   private synchronized boolean traverseNextFishNode(ObjectHandle oh) {
224     //A partir de la posición del objeto, se puede calcular la posición de los hijos
225     //Se calcula la posicion del objeto Amigo de oh
226     int myFriendIndex = gc.dameGCHeap().getObjectPool(oh.objectPos);
227     //Si esa posición no es vacía, es decir, si tiene Amigo
228     if (oh.gotFriend) {
229       ObjectHandle myFriend = gc.dameGCHeap().getObjectHandle(myFriendIndex);
230       Objeto_Indice oiFriend = new Objeto_Indice(myFriend,myFriendIndex);
231       //Si todavía no se ha leído se añade a la lista de objetos pendientes
232       if ((myFriend.leido == false)){
233         listaPendientes.add(oiFriend);
234       }
235     }
236     //A partir de la posición del objeto, se puede calcular la posición de los hijos
237     //Se calcula la posicion del objeto Comida de oh
238     int myLunchIndex = gc.dameGCHeap().getObjectPool(oh.objectPos+1);
239     if (oh.gotLunch) {
240       ObjectHandle myLunch = gc.dameGCHeap().getObjectHandle(myLunchIndex);
241       Objeto_Indice oiLunch = new Objeto_Indice(myLunch,myLunchIndex);
242       //Si todavía no se ha leído se añade a la lista de objetos pendientes
243       if ((myLunch.leido == false)){
244         listaPendientes.add(oiLunch);
245       }
246     }
247     //A partir de la posición del objeto, se puede calcular la posición de los hijos
248     //Se calcula la posicion del objeto Aperitivo de oh
249     int mySnackIndex = gc.dameGCHeap().getObjectPool(oh.objectPos+2);
250     if (oh.gotSnack) {
251       ObjectHandle mySnack = gc.dameGCHeap().getObjectHandle(mySnackIndex);
252       Objeto_Indice oiSnack = new Objeto_Indice(mySnack,mySnackIndex);
253       //Si todavía no se ha leído se añade a la lista de objetos pendientes
254       if ((mySnack.leido == false)){
255         listaPendientes.add(oiSnack);
256       }
257     }
258     oh.leido=true;
259     boolean hecho = false;
260     if(listaPendientes.isEmpty()) hecho = true;
261     else{
262       Objeto_Indice siguiente = (Objeto_Indice)listaPendientes.elementAt(0);
263       int indiceSig = siguiente.dameIndice();
264       currentFishBeingMarked=indiceSig;
265       listaPendientes.remove(siguiente);
266     }
267     return hecho;
268   }
```

## 9. LinkFishCanvas

Clase encargada de crear las relaciones entre los peces. Para comentar esta clase nos olvidaremos de la parte de código relacionada con el dibujo y coloreado de los peces.
En el método *boolean mouseUp(Event evt, int x, int y)* (línea 20) se detecta cuando se ha pulsado el botón del ratón y se comprueba si se ha pulsado sobre algún pez para crear una nueva relación. Para permitir la creación de nuevas relaciones una vez que el recolector ha comenzado su ejecución, se realizan algunas modificaciones en el código de partida que a continuación de detallan.
Si se detecta que se ha creado una relación desde una raíz hasta un pez, una vez que el recolector ya ha visitado dicha raíz (línea 96), el objeto destino de esta relación y sus hijos se añaden a la lista de nuevos objetos que visitará el recolector al final de su ejecución.
En caso de que la relación a crear no fuera de raíz a pez, sino entre dos peces, se distinguen varios casos:
- Si es una relación desde un nodo negro a uno blanco (línea 147), se debe hacer cumplir el invariante. El nodo blanco se colorea de gris y se añaden él y sus hijos a la lista de nodos pendientes.
- Si se crea una relación desde un nodo gris (línea 166), no es necesario modificar ningún color, ya que no incumple el invariante, únicamente se añade a la lista de nuevos objetos.
- Si se crea una relación desde un nodo blanco que se encuentra en la lista de nuevos objetos (línea 183), el nodo destino y sus hijos se añaden a la lista de nuevos objetos.

Además se añadieron nuevos métodos:
El método *meteHijosEnListaNuevos(ObjectHandle oh)* (línea 239), se encarga de ir añadiendo los hijos no visitados de oh a la lista de nuevos objetos.
El método *boolean estaEnNuevos(ObjectHandle oh)* (línea 295), devuelve verdadero si el objeto oh se encuentra en la lista de nuevos objetos y falso en caso contrario.

```
1  public class LinkFishCanvas extends AssignReferencesCanvas {
2
3      private boolean iconClicked = false;
4      private boolean yellowLocalVarClicked = false;
5      private boolean blueLocalVarClicked = false;
6      private boolean redLocalVarClicked = false;
7      private Point posOfMouseInsideIconWhenFirstPressed = new Point(0, 0);
8      private int objectIndexOfFishIconThatWasClicked;
9      private boolean dragging = false;
10     private Point currentMouseDragPosition = new Point(0, 0);
11     private boolean mouseIsOverAnIconThatCanBeDroppedUpon = false;
12     private int objectIndexOfIconThatCanBeDroppedUpon;
13
14     LinkFishCanvas(GCHeap heap, LocalVariables locVars, HeapOfFishTextArea ta) {
15         gcHeap = heap;
16         localVars = locVars;
17         controlPanelTextArea = ta;
18     }
19
20     public synchronized boolean mouseUp(Event evt, int x, int y) {
21
22         if (!iconClicked && !yellowLocalVarClicked && !blueLocalVarClicked
23             && !redLocalVarClicked) {
24             return true;
25         }
```

```
26
27          if (!iconClicked) {
28
29              Color colorOfClickedLocalVar = Color.yellow;
30              if (blueLocalVarClicked) {
31                  colorOfClickedLocalVar = Color.cyan;
32              }
33              else if (redLocalVarClicked) {
34                  colorOfClickedLocalVar = Color.red;
35              }
36
37              if (dragging) {
38                  dragging = false;
39                  // Clear old line.
40                  Graphics g = getGraphics();
41                  g.setColor(Color.blue);
42                  g.setXORMode(colorOfClickedLocalVar);
43
44                  int xLineStart = xLocalVarRectStart + localVarRectWidth;
45                  int yLineStart = yYellowFishLocalVarStart + (localVarRectHeight / 2);
46                  if (blueLocalVarClicked) {
47                      yLineStart = yBlueFishLocalVarStart + (localVarRectHeight / 2);
48                  }
49                  else if (redLocalVarClicked) {
50                      yLineStart = yRedFishLocalVarStart + (localVarRectHeight / 2);
51                  }
52
53                  if (!mouseIsOverAnIconThatCanBeDroppedUpon) {
54
55                      // Clear old line
56                      g.drawLine(xLineStart, yLineStart, currentMouseDragPosition.x,
57                          currentMouseDragPosition.y);
58                  }
59                  else {
60                      ObjectHandle oh =
61  gcHeap.getObjectHandle(objectIndexOfIconThatCanBeDroppedUpon);
62
63                      // Clear the rectangle
64                      g.drawRect(oh.fish.getFishPosition().x, oh.fish.getFishPosition().y,
65  oh.fish.getFishWidth(),
66                          oh.fish.getFishHeight());
67
68                      // Clear the line between the nose of the from fish and the top of the tail of
69                      // of the to fish.
70                      g.drawLine(xLineStart, yLineStart, oh.fish.getFishPosition().x,
71                          oh.fish.getFishPosition().y);
72
73                      mouseIsOverAnIconThatCanBeDroppedUpon = false;
74
75                      Point o = oh.fish.getFishPosition();
76                      if (x >= o.x && x < o.x + oh.fish.getFishWidth() && y >= o.y
77                          && y < o.y + oh.fish.getFishHeight()) {
78
79                          if (oh.fish.getFishColor() == colorOfClickedLocalVar) {
80
81                              // Set the local variable to equal the dropped upon
82                              // fish object.
83                              if (yellowLocalVarClicked) {
84                                  localVars.yellowFish = objectIndexOfIconThatCanBeDroppedUpon;
85                              }
```

```
86                      else if (blueLocalVarClicked) {
87                         localVars.blueFish = objectIndexOfIconThatCanBeDroppedUpon;
88                      }
89                      else if (redLocalVarClicked) {
90                         localVars.redFish = objectIndexOfIconThatCanBeDroppedUpon;
91                      }
92
93                      //Si se crea una relación desde una raíz hasta un objeto una vez que el
94                      //recolector ya ha visitado dicha raíz, el objeto destino de esa relación se
95                      //añade a la lista de nuevos objetos, así como sus hijos
96                     if (((oh.myColor==Color.white))){
97                       oh.myColor=Color.gray;
98                       oh.myFriendLineColor=Color.gray;
99                       oh.myLunchLineColor=Color.gray;
100                      oh.mySnackLineColor=Color.gray;
101                      Objeto_Indice oi = new
102   Objeto_Indice(oh,objectIndexOfIconThatCanBeDroppedUpon);
103                      gcHeap.listaNuevosObjetos.add(oi);
104                      meteHijosEnListaNuevos(oh);
105                     }
106                     repaint();
107                  }
108               }
109            }
110         }
111
112         yellowLocalVarClicked = false;
113         blueLocalVarClicked = false;
114         redLocalVarClicked = false;
115
116         return true;
117      }
118
119      ObjectHandle fishObjectThatWasClicked =
120   gcHeap.getObjectHandle(objectIndexOfFishIconThatWasClicked);
121
122      FishIcon fishIconThatWasClicked = fishObjectThatWasClicked.fish;
123      Color colorOfClickedFish = fishObjectThatWasClicked.fish.getFishColor();
124
125      if (dragging) {
126         dragging = false;
127         // Clear old line.
128         Graphics g = getGraphics();
129         g.setColor(Color.blue);
130         g.setXORMode(colorOfClickedFish);
131
132         Point lineStart = fishIconThatWasClicked.getFishNosePosition();
133
134         if (!mouseIsOverAnIconThatCanBeDroppedUpon) {
135
136            // Clear old line
137            g.drawLine(lineStart.x, lineStart.y, currentMouseDragPosition.x,
138               currentMouseDragPosition.y);
139         }
140         else {
141            ObjectHandle oh =
142   gcHeap.getObjectHandle(objectIndexOfIconThatCanBeDroppedUpon);
143
144            // Se hace cumplir el invariante, si se crea una referencia desde un nodo negro a uno
145            //blanco, el nodo blanco se colorea de gris y se añade a la lista de nuevos objetos
```

```
146             //tanto él, como sus hijos
147             if ((fishObjectThatWasClicked.myColor == Color.black)
148                &&((oh.myColor==Color.white))){
149               oh.myColor=Color.gray;
150               fishObjectThatWasClicked.myFriendLineColor=Color.black;
151               fishObjectThatWasClicked.myLunchLineColor=Color.black;
152               fishObjectThatWasClicked.mySnackLineColor=Color.black;
153               oh.myFriendLineColor=Color.gray;
154               oh.myLunchLineColor=Color.gray;
155               oh.mySnackLineColor=Color.gray;
156               Objeto_Indice oi = new
157 Objeto_Indice(oh,objectIndexOfIconThatCanBeDroppedUpon);
158               gcHeap.listaNuevosObjetos.add(oi);
159               meteHijosEnListaNuevos(oh);
160             }
161             else
162
163               //Si se crea una referencia desde un nodo gris, no es necesario modificar ningún
164               //color, ya que no incumple el invariante, pero sí se añade a la lista de nuevos
165               //objetos él y sus hijos
166               if ((fishObjectThatWasClicked.myColor == Color.gray)
167                  &&((oh.myColor==Color.white)||(!estaEnNuevos(fishObjectThatWasClicked)))){
168
169                 oh.myFriendLineColor=Color.gray;
170                 oh.myLunchLineColor=Color.gray;
171                 oh.mySnackLineColor=Color.gray;
172
173                 Objeto_Indice oi = new
174 Objeto_Indice(oh,objectIndexOfIconThatCanBeDroppedUpon);
175                 gcHeap.listaNuevosObjetos.add(oi);
176                 meteHijosEnListaNuevos(oh);
177               }
178
179               else
180                 //Si se crea una referencia desde un nodo blanco que se encuentra en la lista de
181                 //nuevos objetos hacia otro nodo, el nodo destino y sus hijos se añaden a la lista
182                 //de nuevos objetos
183                 if ((estaEnNuevos(fishObjectThatWasClicked))&&!(estaEnNuevos(oh))){
184                   fishObjectThatWasClicked.myFriendLineColor=Color.black;
185                   fishObjectThatWasClicked.myLunchLineColor=Color.black;
186                   fishObjectThatWasClicked.mySnackLineColor=Color.black;
187                   oh.myFriendLineColor=Color.gray;
188                   oh.myLunchLineColor=Color.gray;
189                   oh.mySnackLineColor=Color.gray;
190
191               Objeto_Indice oi = new
192 Objeto_Indice(oh,objectIndexOfIconThatCanBeDroppedUpon);
193               gcHeap.listaNuevosObjetos.add(oi);
194               meteHijosEnListaNuevos(oh);
195               }
196
197             // Clear the rectangle
198             g.drawRect(oh.fish.getFishPosition().x, oh.fish.getFishPosition().y,
199 oh.fish.getFishWidth(),
200                oh.fish.getFishHeight());
201
202             // Clear the line between the nose of the from fish and the top of the tail of
203             // of the to fish.
204             g.drawLine(lineStart.x, lineStart.y, oh.fish.getFishPosition().x,
205                oh.fish.getFishPosition().y);
```

```
206
207                 mouseIsOverAnIconThatCanBeDroppedUpon = false;
208
209                 Point o = oh.fish.getFishPosition();
210                 if (x >= o.x && x < o.x + oh.fish.getFishWidth() && y >= o.y
211                     && y < o.y + oh.fish.getFishHeight()) {
212
213                     // If they dropped on the same fish they started from, don't link.
214                     if ((objectIndexOfIconThatCanBeDroppedUpon !=
215 objectIndexOfFishIconThatWasClicked)
216                         && fishCanLink(fishIconThatWasClicked, oh.fish)) {
217                         int offset = getInstanceVariableOffset(fishIconThatWasClicked, oh.fish);
218
219                         // Set the clicked upon fish variable to equal the dropped upon
220                         // fish object.
221                         gcHeap.setObjectPool(fishObjectThatWasClicked.objectPos + offset,
222 objectIndexOfIconThatCanBeDroppedUpon);
223                         repaint();
224                     }
225                 }
226             }
227         }
228
229         yellowLocalVarClicked = false;
230         blueLocalVarClicked = false;
231         redLocalVarClicked = false;
232         iconClicked = false;
233
234         return true;
235     }
236
237     //Método recursivo encargado de meter los hijos no visitados(blancos) del nodo oh en la lista
238     //de nuevos objetos
239     private void meteHijosEnListaNuevos(ObjectHandle oh){
240       //Si tiene Amigo de color blanco, actualiza el color de las líneas y llama al método
241       //recursivamente con el nodo amigo
242       if (oh.gotFriend){
243         int myFriendIndex = gcHeap.getObjectPool(oh.objectPos);
244         ObjectHandle myFriend = gcHeap.getObjectHandle(myFriendIndex);
245         if((myFriend.myColor==Color.white)){
246           Objeto_Indice oiFriend = new Objeto_Indice(myFriend,myFriendIndex);
247           oh.myFriendLineColor=Color.gray;
248           oh.myLunchLineColor=Color.gray;
249           oh.mySnackLineColor=Color.gray;
250           myFriend.myFriendLineColor=Color.black;
251           myFriend.myLunchLineColor=Color.black;
252           myFriend.mySnackLineColor=Color.black;
253           gcHeap.listaNuevosObjetos.add(oiFriend);
254           meteHijosEnListaNuevos(myFriend);
255         }
256       }
257       //Si tiene Comida de color blanco, actualiza el color de las líneas y llama al método
258       //recursivamente con el nodo Comida
259       if (oh.gotLunch){
260         int myLunchIndex = gcHeap.getObjectPool(oh.objectPos+1);
261         ObjectHandle myLunch = gcHeap.getObjectHandle(myLunchIndex);
262         if((myLunch.myColor==Color.white)){
263           Objeto_Indice oiLunch = new Objeto_Indice(myLunch,myLunchIndex);
264           oh.myFriendLineColor=Color.gray;
265           oh.myLunchLineColor=Color.gray;
```

```
266           oh.mySnackLineColor=Color.gray;
267           myLunch.myFriendLineColor=Color.black;
268           myLunch.myLunchLineColor=Color.black;
269           myLunch.mySnackLineColor=Color.black;
270           gcHeap.listaNuevosObjetos.add(oiLunch);
271           meteHijosEnListaNuevos(myLunch);
272         }
273       }
274       //Si tiene Aperitivo de color blanco, actualiza el color de las líneas y llama al método
275       //recursivamente con el nodo Aperitivo
276       if (oh.gotSnack){
277         int mySnackIndex = gcHeap.getObjectPool(oh.objectPos+2);
278         ObjectHandle mySnack = gcHeap.getObjectHandle(mySnackIndex);
279         if((mySnack.myColor==Color.white)){
280           Objeto_Indice oiSnack = new Objeto_Indice(mySnack,mySnackIndex);
281           oh.myFriendLineColor=Color.gray;
282           oh.myLunchLineColor=Color.gray;
283           oh.mySnackLineColor=Color.gray;
284           mySnack.myFriendLineColor=Color.black;
285           mySnack.myLunchLineColor=Color.black;
286           mySnack.mySnackLineColor=Color.black;
287           gcHeap.listaNuevosObjetos.add(oiSnack);
288           meteHijosEnListaNuevos(mySnack);
289         }
290       }
291     }
292
293   //Método que devuelve true si el objeto oh se encuentra en la lista de nuevos objetos y false
294   //en caso contrario
295   private boolean estaEnNuevos(ObjectHandle oh){
296     boolean estaEnNuevos = false;
297     for(int i=0; i<gcHeap.listaNuevosObjetos.size(); i++){
298       Objeto_Indice oi = (Objeto_Indice)gcHeap.listaNuevosObjetos.elementAt(i);
299       if (oi.dameOh()==oh){
300         estaEnNuevos = true;
301       }
302     }
303     return estaEnNuevos;
304   }
305 }
```

## *10. ObjetoIndice*

Clase que relaciona objetos con la posición que ocupan en el heap. El objetivo de esta clase es facilitar mucho el tratamiento de objetos del heap a la hora de hacer los recorridos.

```
1  public class Objeto_Indice {
2
3    //Posición del objeto oh en el heap
4    private int indice;
5    //Objeto del heap
6    private ObjectHandle oh;
7
8    public Objeto_Indice(ObjectHandle oh1, int i) {
9      oh = oh1;
10     indice=i;
11   }
12
13   public int dameIndice(){
14     return indice;
15   }
16
17   public ObjectHandle dameOh(){
18     return oh;
19   }
20
21   public void ponOh(ObjectHandle oh1){
22     oh=oh1;
23   }
24
25   public void ponIndice(int i){
26     indice=i;
27   }
28 }
```

## *11. PecesIniciales*

Clase encargada de crear los peces de forma automática.

Métodos

El método *pecesIniciales (GHeap gc)* (línea 7) crea el estado inicial del heap *gc*, al ejecutar la aplicación se llama a este método que crea 9 peces de diferentes colores. Con esto se consigue que se pueda realizar la creación de relaciones de forma automática, ya que si el heap estuviera vacío no se podría crear ninguna relación.

Los métodos void *nuevoPezRojo()* (línea 21), *nuevoPezAzul()* (línea 34), *nuevoPezAmarillo()* (línea 46) crean un pez de color rojo, azul y amarillo respectivamente. Para el correcto funcionamiento del recolector de memoria, los nuevos peces se crean coloreados de blanco (líneas 29, 31 y 52).

```
1  public class PecesIniciales {
2
3    private GCHeap gcHeap;
4
5    //Metodo que crea el estado inicial del heap, al ejecutar la aplicación se crean 9 peces,
6    //de cada color
7    public PecesIniciales(GCHeap gc) {
8      gcHeap = gc;
9      nuevoPezRojo();
10     nuevoPezRojo();
11     nuevoPezAzul();
12     nuevoPezAmarillo();
13     nuevoPezAmarillo();
14     nuevoPezAmarillo();
15     nuevoPezRojo();
16     nuevoPezAzul();
17     nuevoPezAzul();
18   }
19
20   //Metodo que crea un nuevo pez rojo
21   public void nuevoPezRojo(){
22     FishIcon fish = new BigRedFishIcon(false);
23      int newFish = gcHeap.allocateObject(12, fish);
24      if (newFish > 0) {
25          ObjectHandle oh = gcHeap.getObjectHandle(newFish);
26          gcHeap.setObjectPool(oh.objectPos, 0);
27          gcHeap.setObjectPool(oh.objectPos + 1, 0);
28          gcHeap.setObjectPool(oh.objectPos + 2, 0);
29          oh.myColor=Color.white;
30      }
31   }
32
33 //Metodo que crea un nuevo pez azul
34 public void nuevoPezAzul(){
35     FishIcon fish = new MediumBlueFishIcon(false);
36     int newFish = gcHeap.allocateObject(8, fish);
37     if (newFish > 0) {
38         ObjectHandle oh = gcHeap.getObjectHandle(newFish);
39         gcHeap.setObjectPool(oh.objectPos, 0);
40         gcHeap.setObjectPool(oh.objectPos + 1, 0);
41         oh.myColor=Color.white;
42     }
43   }
```

```
44
45      //Metodo que crea un nuevo pez amarillo
46      public void nuevoPezAmarillo(){
47        FishIcon fish = new LittleYellowFishIcon(false);
48        int newFish = gcHeap.allocateObject(4, fish);
49        if (newFish > 0) {
50            ObjectHandle oh = gcHeap.getObjectHandle(newFish);
51            gcHeap.setObjectPool(oh.objectPos, 0);
52            oh.myColor=Color.white;
53        }
54      }
55    }
```

## *12. ReferenciasIniciales*

Clase encargada de crear las relaciones entre peces de forma automática para un ejemplo predefinido.

Atributos
El atributo *thread* de tipo *Thread* es el hilo que irá creando las relaciones. (Se declara en línea 7).
Se puede elegir entre varios grafos ejemplos propuestos, el atributo *tipoEjemplo* codifica el ejemplo a crear, se toma por defecto el ejemplo 1. (Se declara en línea 9).
El atributo *tiempoEspera* es el tiempo de espera entre la creación de una relación y la siguiente. La velocidad a la que se van creando las relaciones. (Se declara en línea 11).

Métodos
Existen 2 métodos principales en esta clase, el método que crea una relación entre 2 peces y el método que crea una relación entre una raíz y un pez.

El método *creaReferencia (int indiceOrigen, int destino, int origen)* (línea 47) crea una relación desde el objeto cuya posición en el hea es *origen* hasta el objeto de posición *destino*. El significado de *indiceOrigen* lo explicaremos a través de un ejemplo. En el heap, los objetos rojos ocupan 3 posiciones porque tiene 3 tipos de relación, los azules 2 y los amarillos 1. Supongamos un heap como el siguiente, en el que hay 2 peces, uno rojo y uno azul.

Posición 1 del heap ->Pez Rojo, la posición 0 corresponde a Amistad
Posición 2 del heap ->Pez Rojo, la posición 1 corresponde a Comida
Posición 3 del heap ->Pez Rojo, la posición 2 corresponde a Aperitivo
Posición 4 del heap ->Pez Azul, la posición 0 corresponde a Amistad
Posición 5 del heap ->Pez Azul, la posición 1 corresponde a Comida

*indiceOrigen* es el lugar que ocupa en el heap la posición del tipo de relación a crear, por ejemplo *indOrigen* de una relación de Comida desde el pez azul sería 5 mientras que si la relación fuera de amistad, *indOrigen* sería 4. Para crear una relación entre el pez rojo y el azul, *origen* sería 1 (el primer objeto del heap es el rojo), *destino* es igual a 2 (el segundo objeto es el azul) y *indOrigen* es igual a 2 (lugar que ocupa el tipo de relación Comida del pez rojo en el heap). La llamada al método sería *creaReferencia(1,2,2).*

Al crear relaciones hay que controlar que se cumpla el invariente de los algoritmos incrementales: No puede haber relaciones de nodos negros a blancos. Si así fuera, el nodo blanco se colorearía de gris. Además el método se declara como sincronizado.

A continuación se describe un pseudocódigo del método:
- Si el nodo origen es negro y el destino blanco, se incumple el invariante, por lo que el nodo destino se colorea de gris, se crea la relación y se añade él y sus hijos a la lista de nuevos objetos. (línea 52).
- Sino, si el nodo origen es gris, no hace falta modificar ningún color, ya que no incumple el invariante, se crea la relación y se añade el destino y sus hijos a la lista de nuevos. (línea 69).
- Sino, si el origen está en la lista de nuevos y el destino no, se crea la relación y se añade el destino y sus hijos a la lista de nuevos. (línea 81).

El otro de los métodos importantes de esta clase es *creaReferenciaDeRaiz(int origen, int destino)* (línea 99), que crea una relación desde una raíz hacia un objeto destino. Las

raíces se codifican de la siguiente forma: *origen*=1, significa raíz amarilla, *origen*=2, raíz azul y *origen*=3, raíz roja. Este método también se declara como sincronizado.
Al crear la relación hay que tener en cuenta que si el origen es una raíz que ya ha sido visitada por el recolector, el objeto destino se debe colorear de gris y se añade él y sus hijos a la lista de nuevos objetos. (línea 113).

Otros métodos auxiliares de esta clase son *meteHijosEnListaNuevos(ObjectHandle oh)* (línea 134), que mete en la lista de nuevos, a los hijos de *oh* que no han sido visitados, es decir los coloreados de blanco; y *boolean estaEnNuevos(ObjectHandle oh)* (línea 182), que devuelve verdadero si el objeto *oh* esta en la lista de nuevos objetos y falso en caso contrario.

Por último se proponen dos ejemplos de creación de relaciones sobre un grafo, el *ejemplo1()* (línea 217) es un ejemplo sencillo mientras que el *ejemplo2()* (línea 279) es un ejemplo de grafo más complejo en el que existen ciclos.

```
1  public class ReferenciasIniciales implements Runnable{
2
3    private GCHeap gcHeap;
4    private GarbageCollectCanvas gc;
5    private LocalVariables localVars;
6    //Hilo que creará las relaciones
7    private Thread thread;
8    //Atributo que codifica el tipo de ejemplo a ejecutar, por defecto el ejemplo 1
9    int tipoEjemplo=1;
10   //Tiempo de espera entre la creación de una relación y otra
11   int tiempoEspera = 1500;
12
13   public ReferenciasIniciales(GCHeap heap,GarbageCollectCanvas gCollect, LocalVariables lv) {
14     gcHeap = heap;
15     gc = gCollect;
16     localVars=lv;
17   }
18
19   public Thread dameHilo(){
20     return thread;
21   }
22
23   public void ponTipoEjemplo(int i){
24     tipoEjemplo=i;
25   }
26
27   //Metodo que crea una referencia desde el objeto cuya posición en el heap es origen
28   //hasta el objeto de posición destino
29   //Para explicar el significado de indiceOrigen se realizará mediante un ejemplo
30   //En el heap, los objetos rojos ocupan 3 posiciones porque tienen 3 tipos de relación,
31   //los azules 2 y los amarillos 1. Supongamos un heap como el siguiente, en el que hay
32   //2 objetos,uno rojo y uno azul.
33
34 //1 Objeto Rojo posicion0 (Amistad)
35 //2 Objeto Rojo posicion1 (Comida)
36 //3 Objeto Rojo posicion2 (Aperitivo)
37 //4 Objeto Azul posicion0 (Amistad)
38 //5 Objeto Azul posicion1 (Comida)
39 //indiceOrigen es el lugar que ocupa la posición del tipo de relación en el heap, por
40 //ejemplo el indOrigen de una relación de Comida desde el pez azul sería 5 mientras
41 // que para una relación de amistad sería 4
42 //Si se quiere crear una relación entre el objeto rojo y el azul, origen=1 (el primer objeto
```

```
43  //del heap es el rojo), destino=2 (2º objeto) y indOrigen=2(lugar que ocupa el tipo de
44  //relación del objeto origen en el heap)
45  //La llamada al método sería: creaReferencia(5,1,2)
46
47   public synchronized void creaReferencia(int indiceOrigen, int destino, int origen){
48     gcHeap.setObjectPool(indiceOrigen, destino);
49     //Se calculan los objetos origen y destino
50     ObjectHandle nodoOrigen = gcHeap.getObjectHandle(origen);
51     ObjectHandle nodoDestino = gcHeap.getObjectHandle(destino);
52     //Si el origen es negro y el destino blanco, se incumpliría el invariante, por lo que el
53     //destino se colorea de gris y se añade él y sus hijos a la lista de nuevos objetos
54     if ((nodoOrigen.myColor == Color.black)
55         &&((nodoDestino.myColor==Color.white))){
56       nodoDestino.myColor=Color.gray;
57       nodoOrigen.myFriendLineColor=Color.black;
58       nodoOrigen.myLunchLineColor=Color.black;
59       nodoOrigen.mySnackLineColor=Color.black;
60       nodoDestino.myFriendLineColor=Color.gray;
61       nodoDestino.myLunchLineColor=Color.gray;
62       nodoDestino.mySnackLineColor=Color.gray;
63       Objeto_Indice oi = new Objeto_Indice(nodoDestino,destino);
64       gcHeap.listaNuevosObjetos.add(oi);
65       meteHijosEnListaNuevos(nodoDestino);
66     }
67     else
68       //Si el origen es gris, se añade el destino y sus hijos a la lista de nuevos
69       if ((nodoOrigen.myColor == Color.gray)
70           &&((nodoDestino.myColor==Color.white)||(!estaEnNuevos(nodoOrigen)))){
71         nodoDestino.myFriendLineColor=Color.gray;
72         nodoDestino.myLunchLineColor=Color.gray;
73         nodoDestino.mySnackLineColor=Color.gray;
74         Objeto_Indice oi = new Objeto_Indice(nodoDestino,destino);
75         gcHeap.listaNuevosObjetos.add(oi);
76         meteHijosEnListaNuevos(nodoDestino);
77       }
78       else
79         //Si el origen está en la lista de nuevos y el destino no, se añade este y sus hijos a la lista
80         //de nuevos
81         if ((estaEnNuevos(nodoOrigen))&&!(estaEnNuevos(nodoDestino))){
82           nodoOrigen.myFriendLineColor=Color.black;
83           nodoOrigen.myLunchLineColor=Color.black;
84           nodoOrigen.mySnackLineColor=Color.black;
85           nodoDestino.myFriendLineColor=Color.gray;
86           nodoDestino.myLunchLineColor=Color.gray;
87           nodoDestino.mySnackLineColor=Color.gray;
88           Objeto_Indice oi = new Objeto_Indice(nodoDestino,destino);
89           gcHeap.listaNuevosObjetos.add(oi);
90           meteHijosEnListaNuevos(nodoDestino);
91         }
92   }
93
94   //Metodo que crea una referencia desde una raiz hacia un objeto destino
95   //Las raices se codifican de la siguiente forma:
96   //Origen=1 -> raiz marilla
97   //Origen=2 -> raiz azul
98   //Origen=3 -> raiz roja
99   public synchronized void creaReferenciaDeRaiz(int origen, int destino){
100    ObjectHandle nodoDestino = gcHeap.getObjectHandle(destino);
101    //Raíz amarilla
102    if(origen==1){
```

```
103       localVars.yellowFish=destino;
104     }
105     //Raíz azul
106     else if(origen==2){
107       localVars.blueFish=destino;
108     }
109     //Raíz roja
110     else if(origen==3){
111       localVars.redFish=destino;
112     }
113     //Si se crea un objeto desde una raíz que ya ha sido visitada por el algoritmo recolector, el
114     //objeto destino se colorea de gris y se añade él y sus hijos a la lista de nuevos objetos
115     //Estado > 1 quiere decir que el algoritmo ya ha pasado por la fase de visita de la raíz roja
116     if (((origen==1)&&(gc.dameEstado()>1)&&(nodoDestino.myColor==Color.white))||
117         //Estado > 4 quiere decir que el algoritmo ya ha pasado por la fase de visita de la raíz azul
118        ((origen==2)&&(gc.dameEstado()>4)&&(nodoDestino.myColor==Color.white))||
119         //Estado > 7 quiere decir que el algoritmo ya ha pasado por la fase de visita de la raíz
120         //amarilla
121        ((origen==3)&&(gc.dameEstado()>7)&&(nodoDestino.myColor==Color.white))){
122       nodoDestino.myColor=Color.gray;
123       nodoDestino.myFriendLineColor=Color.gray;
124       nodoDestino.myLunchLineColor=Color.gray;
125       nodoDestino.mySnackLineColor=Color.gray;
126       Objeto_Indice oi = new Objeto_Indice(nodoDestino,destino);
127       gcHeap.listaNuevosObjetos.add(oi);
128       meteHijosEnListaNuevos(nodoDestino);
129     }
130   }
131
132   //Metodo recursivo que mete los hijos no visitados(blancos) del nodo en la lista de nuevos
133   //objetos
134   private void meteHijosEnListaNuevos(ObjectHandle oh){
135     if (oh.gotFriend){
136       int myFriendIndex = gcHeap.getObjectPool(oh.objectPos);
137       ObjectHandle myFriend = gcHeap.getObjectHandle(myFriendIndex);
138       if((myFriend.myColor==Color.white)){
139         Objeto_Indice oiFriend = new Objeto_Indice(myFriend,myFriendIndex);
140         oh.myFriendLineColor=Color.gray;
141         oh.myLunchLineColor=Color.gray;
142         oh.mySnackLineColor=Color.gray;
143         myFriend.myFriendLineColor=Color.black;
144         myFriend.myLunchLineColor=Color.black;
145         myFriend.mySnackLineColor=Color.black;
146         gcHeap.listaNuevosObjetos.add(oiFriend);
147         meteHijosEnListaNuevos(myFriend);
148       }
149     }
150     if (oh.gotLunch){
151       int myLunchIndex = gcHeap.getObjectPool(oh.objectPos+1);
152       ObjectHandle myLunch = gcHeap.getObjectHandle(myLunchIndex);
153       if((myLunch.myColor==Color.white)){
154         Objeto_Indice oiLunch = new Objeto_Indice(myLunch,myLunchIndex);
155         oh.myFriendLineColor=Color.gray;
156         oh.myLunchLineColor=Color.gray;
157         oh.mySnackLineColor=Color.gray;
158         myLunch.myFriendLineColor=Color.black;
159         myLunch.myLunchLineColor=Color.black;
160         myLunch.mySnackLineColor=Color.black;
161         gcHeap.listaNuevosObjetos.add(oiLunch);
162         meteHijosEnListaNuevos(myLunch);
```

```
      }
    }
    if (oh.gotSnack){
      int mySnackIndex = gcHeap.getObjectPool(oh.objectPos+2);
      ObjectHandle mySnack = gcHeap.getObjectHandle(mySnackIndex);
      if((mySnack.myColor==Color.white)){
        Objeto_Indice oiSnack = new Objeto_Indice(mySnack,mySnackIndex);
        oh.myFriendLineColor=Color.gray;
        oh.myLunchLineColor=Color.gray;
        oh.mySnackLineColor=Color.gray;
        mySnack.myFriendLineColor=Color.black;
        mySnack.myLunchLineColor=Color.black;
        mySnack.mySnackLineColor=Color.black;
        gcHeap.listaNuevosObjetos.add(oiSnack);
        meteHijosEnListaNuevos(mySnack);
      }
    }
  }

  //Devuelve true si el objeto oh está en el vetor listaNuevosObjetos y false en caso contrario
  private boolean estaEnNuevos(ObjectHandle oh){
    boolean estaEnNuevos = false;
    for(int i=0; i<gcHeap.listaNuevosObjetos.size(); i++){
      Objeto_Indice oi = (Objeto_Indice)gcHeap.listaNuevosObjetos.elementAt(i);
      if (oi.dameOh()==oh){
        estaEnNuevos = true;
      }
    }
    return estaEnNuevos;
  }

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

  public void run() {
    switch(tipoEjemplo){
      //Ejecución del ejemplo elegido
      case 1:{
        ejemplo1();
      }
      break;
      case 2:{
        ejemplo2();
      }
      break;
      default: {
        ejemplo1();
      }
    }
  }

//Ejemplo de ejecución automática de relaciones
  public void ejemplo1(){
    //Referencia de la raíz roja al objeto 1 del heap
    creaReferenciaDeRaiz(3,1);
    gc.repaint();
    try {
      thread.sleep(tiempoEspera);
```

```
223       } catch (InterruptedException e) {}
224       //Referencia de la raíz azul al objeto 8 del heap
225       creaReferenciaDeRaiz(2,8);
226       gc.repaint();
227       try {
228         thread.sleep(tiempoEspera);
229       } catch (InterruptedException e) {}
230       //Referencia de la raíz amarilla al objeto 5 del heap
231       creaReferenciaDeRaiz(1,5);
232       gc.repaint();
233       try {
234         thread.sleep(tiempoEspera);
235       } catch (InterruptedException e) {}
236       //Referencia del objeto 4 al 1
237       creaReferencia(3,4,1);
238       gc.repaint();
239       try {
240         thread.sleep(tiempoEspera);
241       } catch (InterruptedException e) {}
242       //Referencia del objeto 8 al 7
243       creaReferencia(19,8,7);
244       gc.repaint();
245       try {
246         thread.sleep(tiempoEspera);
247       } catch (InterruptedException e) {}
248       //Referencia del objeto 2 al 1
249       creaReferencia(1,2,1);
250       gc.repaint();
251       try {
252         thread.sleep(tiempoEspera);
253       } catch (InterruptedException e) {}
254       //Referencia del objeto 9 al 3
255       creaReferencia(9,9,3);
256       gc.repaint();
257       try {
258         thread.sleep(tiempoEspera);
259       } catch (InterruptedException e) {}
260       //Referencia del objeto 6 al 5
261       creaReferencia(14,6,5);
262       gc.repaint();
263       try {
264         thread.sleep(tiempoEspera);
265       } catch (InterruptedException e) {}
266       //Referencia del objeto 3 al 1
267       creaReferencia(2,3,1);
268       gc.repaint();
269       try {
270         thread.sleep(tiempoEspera);
271       } catch (InterruptedException e) {}
272       //Referencia del objeto 4 al 9
273       creaReferencia(26,4,9);
274       gc.repaint();
275     }
276
277     //Ejemplo para demostrar la correción del algoritmo cuando existen ciclos en el grafo de
278     //relaciones
279     public void ejemplo2(){
280       creaReferenciaDeRaiz(3,1);
281       gc.repaint();
282       try {
```

```
283         thread.sleep(tiempoEspera);
284       }  catch (InterruptedException e) {}
285       creaReferenciaDeRaiz(2,8);
286       gc.repaint();
287       try {
288         thread.sleep(tiempoEspera);
289       }  catch (InterruptedException e) {}
290       creaReferenciaDeRaiz(1,5);
291       gc.repaint();
292       try {
293         thread.sleep(tiempoEspera);
294       }  catch (InterruptedException e) {}
295       //Referencia del objeto 2 al 1
296       creaReferencia(1,2,1);
297       gc.repaint();
298       try {
299         thread.sleep(tiempoEspera);
300       }  catch (InterruptedException e) {}
301       //Referencia del objeto 1 al 2
302       creaReferencia(5,1,2);
303       gc.repaint();
304       try {
305         thread.sleep(tiempoEspera);
306       }  catch (InterruptedException e) {}
307       //Referencia del objeto 9 al 1
308       creaReferencia(2,9,1);
309       gc.repaint();
310       try {
311         thread.sleep(tiempoEspera);
312       }  catch (InterruptedException e) {}
313       //Referencia del objeto 4 al 9
314       creaReferencia(26,4,9);
315       gc.repaint();
316       try {
317         thread.sleep(tiempoEspera);
318       }  catch (InterruptedException e) {}
319       //Referencia del objeto 3 al 8
320       creaReferencia(22,3,8);
321       gc.repaint();
322       try {
323         thread.sleep(tiempoEspera);
324       }  catch (InterruptedException e) {}
325       //Referencia del objeto 6 al 8
326       creaReferencia(23,6,8);
327       gc.repaint();
328       try {
329         thread.sleep(tiempoEspera);
330       }  catch (InterruptedException e) {}
331       //Referencia del objeto 8 al 3
332       creaReferencia(10,8,3);
333       gc.repaint();
334     }
335   }
```

# BIBLIOGRAFÍA


[App91] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation*, Cambridge, Massachusetts, 1991.

[BDG05] Greg Bollella, Bertrand Delsart, Romain Guider, Christophe Lizzi, Frederic Parain. *Mackinac: Making HotSpot Real-Time.* Seattle,Washington. May 2005.

[BW01] Alan Burns and Andy Wellings, *Sistemas de tiempo real y lenguajes de programación.* Addison Wesley. Tercera Edición. 2001.

[Coh81] Jacques Cohen. *Garbage collection of linked data structures.* Computing Surveys. Septiembre 1981.

[JL96] Richard Jones y Rafael D Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons. Septiembre 1996.

[Knu69] Donald E.Knuth. *The Art of Computer Programming*. Volume1: Fundamental Algorithm. Addison Wesley, Reading, Massachusetts,1969.

[LH83] Henry Lieberman and Carl Hewitt. *A real time garbage collector based on the lifetimes of objects.* Junio, 1983.

[PBK03] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V.T. Rajan. *Efficient on-the-fly cycle collection*. Technion University, 2003.

[Pre01] R. Presuman. *Ingeniería del Software. Un enfoque práctico*. 5ª edición, Ed. McGraw-Hill, 2001.

[Ritz03] Tobias Ritzau.*Memory Efficient Hard Real-Time Garbage Collection*. Department of Computer and Information Science Linköping University. Sweden Linköping 2003.

[Som01] I. Sommerville. *Ingeniería del Software*. 6ª edición, Ed. Addison Wesley, 2001.

[Ven00] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill Companies. 2nd edición. Enero 2000.

[Wil92] Paul R. Wilson. *Uniprocessor Garbage Collection Techniques*. St. Malo, Francia. Septiembre 1992.

[Yua90] Taichi Yuasa. *Real-time garbage collection on general-purpose machines*. Journal of Software and Systems. 1990.