Sistemas Informáticos
Curso 2005-2006

Estudio de un recolector de memoria basado en regiones

Laura Herraiz Sanz
Víctor Parra Serrano
Gabriel Salafranca Sánchez de Puerta

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

Facultad de Informática
Universidad Complutense de Madrid

Resumen del proyecto

El lenguaje Java utiliza una herramienta muy útil para el programador: el recolector de basura, que hace que no nos tengamos que preocupar de tareas como liberar memoria. El problema es que en un sistema de tiempo real no se puede detener el control del programa para un algoritmo tan costoso en tiempo como es el del colector de basura. Una alternativa es la propuesta por la especificación de tiempo real para Java; *The Real-Time Specification for Java (RTSJ),* que propone un modelo basado en regiones, agrupando objetos con tiempo de vida similar.

Nuestro proyecto se centra en el estudio del modelo de regiones *scoped* propuesto por RTSJ como alternativa al recolector de basura. Las regiones aparecen de forma explícita en el programa y se recogen de forma implícita. Dado que estas regiones se pueden anidar y que la vida de las mismas depende del flujo del programa, RTSJ define unas reglas de asignación para tratar las relaciones que se pueden producir entre objetos en diferentes regiones.

Estas reglas hacen que RTSJ presente un modelo de programación complicado y poco familiar para el programador, condiciones de carrera en la ejecución de los programas y una ejecución de programas no predecible en tiempo. Para intentar solucionarlos se han propuesto varios modelos alternativos.

En este proyecto analizaremos estos problemas proponiendo soluciones. Dado que se presentan varias soluciones, explicaremos las características de cada una de ellas y las ventajas y desventajas que tienen respecto al modelo original (e.d. RTSJ). También hemos desarrollado una aplicación que muestra una simulación del comportamiento de los modelos propuestos frente a distintas situaciones.

Summary

The Java language use a useful tool for the programmer: the garbage collector, and it does that we don't get worried about tasks like to free memory. The problem is that in a real-time system we can't stop the program control for an algorithm so costly in time like the garbage collector algorithm. The Real-Time Specification for Java (RTSJ) proposes a choice: a model based in regions, that it groups objects with similar lifetime.

Our project is centred on the study of the scoped memory region model, proposed by RTSJ like an alternative to the garbage collector. The regions appear explicitly and they are collected implicitly. Given that this regions can be nestted and their lifes depend on program flow, RTSJ define some assignament rules for the connections between objects in differents regions.

This rules make that RTSJ present an unifamiliar programming model that makes it difficult and tedious for the programmer, race carrier conditions on program execution and a non time-predictable execution programs. In order to solve these problems several alternative models are been seggested.

In this project we will analyze this problems and we will propose solutions. Given that we present some solutions, we will explain the characteristic of each of them and its advantages and disadvantages with the RTSJ model. Furthermore we are developed an application that it shows a simulation of the behavior of each propose model in differents situations.

Lista de palabras

Regiones de memoria

Memoria scoped

Colector (o recolector) de basura

Sistemas de tiempo real

Regla del único padre

Referencias prohibidas

Condiciones de carrera

Modelo que suprime la pila scope

Modelo que suprime la regla del único padre

ScopedCycledException

Los autores de este proyecto, Laura Herraiz, Víctor Parra y Gabriel Salafranca, autorizan mediante este texto a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, la memoria, el código, la documentación y/o el prototipo desarrollado.

Madrid, a de de 2006

Fdo. Laura Herraiz Fdo. Víctor Parra Fdo. Gabriel Salafranca

Agradecimientos

Queríamos expresar nuestro más profundo agradecimiento tanto a nuestra tutora, Mª Teresa Higuera Toledano, como a los profesores Manuel García Clavel y Fernando Rubio Diez por el gran esfuerzo que hicieron para que pudiésemos realizar la presentación días antes de lo que en un principio estaba establecido.

Índice

Capítulo 1
Objetivos

El objetivo principal de este proyecto es entender el comportamiento del modelo RTSJ, sus problemas y las alternativas presentadas y para ello hemos estudiado la especificación dada por Java y los dos modelos propuestos como alternativa. El desarrollo de nuestro trabajo ha ido siguiendo los siguientes pasos:

- Conocer y entender las reglas del capítulo de "Gestión de memoria" de la especificación de tiempo real para Java. También es importante que conociéramos el comportamiento de algunos métodos como `enter()` o `executeInArea()` de las distintas áreas de memoria.

- Estudiar y analizar los problemas que presenta el modelo de Java y posibles alternativas propuestas a dicho modelo. Para ello hemos utilizado dos artículos publicados que explican dos alternativas y las comparan con el modelo de la especificación de Java (Bibliografía, 4 y 5.

- Después de estudiar todo el material y comprender el funcionamiento y la estructura del modelo de regiones de Java, hemos empezado a pensar en realizar una aplicación que mostrase lo que habíamos estudiado.

- Entender en profundidad la concurrencia en Java tomamos como base el libro de Andy Wellings: "Concurrent and Real-Time Programming in Java", y en concreto el capítulo sobre gestión de memoria, que explica el

modelo de memoria utilizado por Java y propone varios ejemplos para su utilización.También hemos tenido que entender y dominar conceptos sobre monitores y threads en Java, para lo que hemos recurrido a distintos libros y materiales publicados en internet.

- Diseño de la aplicación consistente en una serie de clases que simulan el comportamiento de cada uno de los tres modelos. Para ello hemos necesitado hacer una clase "región", una clase "modelo" y una clase "tarea" para cada modelo que contiene la definición de los métodos necesarios, que simulan el comportamiento de los métodos reales de cada modelo.

Queremos hacer notar que al estar todo el material en inglés, para facilitarnos el trabajo, ha sido necesario ir construyendo nuestro propio material en español según íbamos avanzando, así de cada lectura que hemos hecho hemos elaborado informes o resúmenes que pudieran ayudar a los demás.

Capítulo 2
Introducción

Una de las aportaciones del entorno Java es la recolección automática de espacios no utilizados (*garbage collection* o recolección de basura). Dada la sobrecarga de tiempo y de memoria que las técnicas de recolección requieren hacen incompatible su empleo en sistemas de tiempo real. Por otra parte, la recolección implícita de los espacios de memoria no usados permite el desarrollo de programas robustos, ya que el programador se libera de esta tediosa tarea (*dispose* en Pascal, *free* en C), concentrando sus esfuerzos en el desarrollo de la aplicación. Un método intermedio y alternativo consiste en agrupar los objetos con un tiempo de vida similar en regiones de memoria que son recolectadas de forma implícita.

En este proyecto se trata de estudiar el comportamiento de las regiones de memoria propuestas por RTSJ, estudiando las leyes o reglas que lo gobiernan, en particular el anidamiento de regiones de vida limitada (regiones *scoped*) y la recolección implícita de las mismas. El modelo general de regiones que estudia nuestro proyecto plantea además tres tipos de regiones: *Inmortal, Heap* e *InmortalPhysical.*.

La problemática que pretende estudiar este proyecto se centra en las relaciones entre objetos ubicados en regiones con distintos tiempos de vida (no se puede crear una referencia desde un objeto X a otro Y cuyo tiempo de vida es potencialmente menor).

Dadas las ventajas que presenta la recolección automática de memoria, los problemas que plantea su uso en sistemas de tiempo real y el interés que la comunidad de tiempo real tiene en el uso del entorno Java en este tipo de sistemas, el estudio de la gestión de memoria en sistemas Java de tiempo real supone una línea de investigación relevante.

2.1 Java y los sistemas de tiempo real

Los sistemas de tiempo real tienen unas características muy concretas que hacen que no puedan demorarse ciertos cálculos. Por ejemplo, en un sistema que controla una central nuclear, en caso de que se produzca un escape y se tengan que activar los sistemas de emergencia, no puede retrasarse la solución del problema.

Estos sistemas se basan en diferentes tareas que tienen unas características como un tiempo de ejecución o tiempo de cómputo, un *deadline* o un punto máximo en el que debe haber sido ejecutada toda la tarea, un *jitter* o tiempo de error, tolerancia a que no se cumpla el deadline, una fecha de comienzo… Estas características serán en unos sistemas más estrictas (sistemas de tiempo real duros) o más suaves (sistemas de tiempo real blandos). Además si el comportamiento incorrecto del sistema puede provocar la pérdida de vidas o catástrofes similares entonces el sistema de tiempo real es de misión crítica.

2.2 Threads

Java es un sistema *multi-thread* (multi-hilo), ya que permite muchas actividades simultáneas en un programa. Los *threads* (hilos), son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar los threads incluidos en el lenguaje, son más fáciles de usar y más robustos que sus implementaciones en otros lenguajes como C o C++. Esto produce beneficios como por ejemplo un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo sobre el que se ejecuta, aún supera a los entornos de flujo único de programa (*single-threaded*) tanto en facilidad de desarrollo como en rendimiento.

Java es un lenguaje de programación que incorpora threads en el corazón del mismo lenguaje. Comúnmente, los threads son implementados a nivel de sistema, requiriendo una interfaz de programación específica separada del núcleo del lenguaje de programación. Java se presenta como ambos, como lenguaje y como sistema de tiempo de ejecución (*runtime*), siendo posible integrar threads dentro de ambos. El resultado final es que se pueden usar threads de Java como estándar en cualquier plataforma.

En Java, los threads comparten el mismo espacio de memoria. Incluso comparten gran parte del entorno de ejecución, de modo que la creación de nuevos threads es mucho más rápida que la creación de nuevos procesos. La ventaja que proporcionan los threads es la capacidad de tener más de un camino de ejecución en un mismo programa. Así, con un único proceso, ejecutándose una JVM (*Java Virtual Machine*), puede haber más de un thread, cada uno con su propio camino de ejecución.

Un proceso demonio es un proceso que debe ejecutarse continuamente en modo *background* (en segundo plano), y generalmente se diseña para responder a peticiones de otros procesos a través de la red. Los threads demonio también pueden llamarse servicios, porque se ejecutan, normalmente, con prioridad baja y proporcionan un servicio básico a un programa o programas cuando la actividad de la máquina es reducida. Un ejemplo de thread demonio que está ejecutándose continuamente es el colector de basura (*garbage collector*).

Java tiene un planificador (*Scheduler*), que decide que threads deben ejecutarse y cuáles deben encontrarse preparados para su ejecución. Hay dos características de los threads que el planificador tiene en cuenta en este proceso de decisión. Lo primero que tiene en cuenta ya que es lo más importante, es la prioridad del thread y después el indicador de demonio, que dice si un thread es demonio o no.

La regla básica del planificador es que si solamente hay threads demonio ejecutándose, la Máquina Virtual Java concluirá. Los nuevos threads heredan la prioridad y el indicador de demonio de los threads que los han creado. El planificador determina qué threads deberán ejecutarse comprobando la prioridad de todos los threads. Aquellos con prioridad más alta dispondrán del procesador antes de los que tienen prioridad más baja. El planificador puede seguir dos patrones, preventivo y no preventivo:

- Los planificadores preventivos proporcionan un segmento de tiempo a cada thread que se esté ejecutando en el sistema. El planificador decide cuál será el siguiente thread a ejecutarse y le da vida durante un periodo de tiempo fijo. Cuando finaliza ese periodo de tiempo se suspende y se lanza el siguiente thread en la lista de procesos.
- Los planificadores no preventivos, en cambio, deciden cuál es el thread que debe correr y lo ejecutan hasta que concluye. El thread tiene control total sobre el sistema mientras esté en ejecución. Un thread puede forzar al planificador para que comience la ejecución de otro thread que esté esperando.

Los threads presentan algunos problemas para tiempo real, por ejemplo que no se pueden especificar características como el deadline, el jitter... por eso para tiempo real Java utiliza unos hilos específicos, los threads de tiempo real.

2.3 Colector de basura (*Garbage Collector GC)*

Cualquier programa informático hace uso de una cierta cantidad de memoria de trabajo puesta a su disposición por el sistema operativo. Esta memoria tiene que ser gestionada por el propio programa para:

- Reservar espacios de memoria para su uso.
- Liberar espacios de memoria previamente reservados.
- Compactar espacios de memoria libres y consecutivos entre sí.
- Llevar cuenta de que espacios están libres y cuales no.

Generalmente, el programador dispone de una biblioteca de código que se encarga de estas tareas. No obstante, el propio programador es responsable de utilizar adecuadamente esta biblioteca. Esto tiene la ventaja de que se hace un uso eficiente de la memoria, es decir, los espacios de memoria quedan libres cuando ya no son necesarios. No obstante, este mecanismo explícito de gestión de memoria es propenso a errores. Por ejemplo, un programador puede olvidar liberar la memoria de manera que, tarde o temprano, no quede memoria disponible, abortando la ejecución del programa.

Como alternativa es necesaria una gestión implícita de memoria, donde el programador no es consciente de la reserva y liberación de memoria. Java, entre otros lenguajes utiliza un algoritmo colector de basura. En Java el programador no tiene que invocar a una subrutina para liberar memoria. La reserva de memoria también es más o menos automática sin la intervención del programador: se reserva memoria cada vez que el programador crea un objeto, pero éste no tiene que saber cuanta memoria se reserva ni cómo se hace esto.

Uno de los algoritmos típicos del recolector de basura usado por la máquina virtual Java es el colector de seguimiento, que sigue los objetos que son alcanzables por un conjunto de objetos raíz, para determinar qué objetos deberían permanecer en memoria y cuales no. Los objetos alcanzables sobreviven al colector de basura porque podrían ser utilizados más adelante en la ejecución del programa.

El uso del recolector de basura tiene la ventaja de que el programador no puede cometer errores y queda liberado de la tediosa tarea de gestionar la memoria. Pero tiene otras desventajas como que la memoria permanezca retenida durante más tiempo del estrictamente necesario y que el colector de basura tarde cierto tiempo en hacer su tarea, por lo que el programa siempre será más ineficiente.

Además, en un sistema de tiempo real crítico, no puede pararse ni un momento el control de los hechos del sistema y menos para un algoritmo tan costoso cómo es el del colector de basura de Java. Como alternativa la especificación de Java para tiempo real (e.d. RTSJ) introduce las regiones de memoria, presentando los siguientes tipos de regiones:

- *Heap Memory*: es el espacio de memoria que se colecta por el algoritmo de colector de basura. Solo puede haber una región de este tipo.
- *Scoped Memory*: los objetos de esta región tienen un tiempo de vida limitado y cuando pasa este tiempo se destruye junto con los objetos que contiene. Puede haber varias instancias de regiones Scoped. Las regiones scoped pueden ser:
    a. Scoped constante: que requiere que el tiempo de asignación sea directamente proporcional al tamaño del objeto que esta siendo asignado.
    b. Scoped variable: donde la asignación puede producirse en un tiempo variable.
- *Inmortal Memory*: los objetos almacenados existen hasta que la aplicación finaliza. Sólo puede haber una región de este tipo.
- *Inmortal Physical Memory*: los objetos de esta región hacen referencia a zonas de memoria física concretas. Puede haber varias instancias de regiones de este tipo.

2.4 Regiones de RTSJ

Como hemos dicho antes, en el sistema solo puede haber una instancia de la región Heap y otra de la región Inmortal, las cuales son compartidas por todos los threads del sistema. Sin embargo, para la región Scoped y para la región Inmortal Physical se pueden crear varias instancias.

Lo primero a lo que debemos hacer mención es a las relaciones posibles entre unas zonas de memoria y otras. Entre las regiones Heap e Inmortal puede haber relaciones, es decir: si un objeto A pertenece a Heap y un objeto B a Immortal puedo realizar asignaciones del tipo: A.campo := B o B.campo := A. Desde los objetos de Scoped también podemos hacer referencia a objetos de Heap, que aunque desaparecen se tratan estos elementos de otra forma, y a objetos de Inmortal.

El problema llega cuando se trabaja con las regiones Scoped. Dado que tienen distinta forma de recolección que la región heap y la inmortal ni siquiera tiene recolección, no podemos hacer que un campo de un objeto ubicado en la región Inmortal o en el Heap referencie a un objeto en una región Scoped, debido a que un objeto de Inmortal que va a permanecer siempre ahí hasta el final de la vida del sistema, o que un objeto de Heap, que tendrá un tiempo de vida indeterminado, apunte a información de Scoped, que en cualquier momento puede desaparecer, dejando un puntero colgando, es decir a NIL, y provocando una excepción irreparable que abortaría la ejecución del programa. No podemos permitir esto es un sistema que puede tener una importancia crítica.

Las asignaciones entre regiones de memoria se pueden resumir en la siguiente tabla:

Desde el área de memoria: ↓	Referencia a **Heap**	Referencia a **Inmortal**	Referencia a **Scoped**
Heap	Sí	Sí	No
Inmortal	Sí	Sí	No
Scoped	Sí	Sí	Sí, con condiciones

Por ultimo, el principal problema en el que se centra nuestro proyecto, son las relaciones entre objetos de la región de memoria Scoped, que explicaremos con más detalle.

2.5 Estructura de la memoria del proyecto

En este capítulo se ha dado una introducción sobre las bases teóricas en las que se encuadra este proyecto: sistemas de tiempo real, recolector de basura... El capítulo tres presenta el modelo RTSJ y analiza las reglas que tiene. En el capítulo cuatro comentamos los problemas de este modelo y presentamos unos modelos como alternativa para intentar solucionar esos problemas.

Una vez conocidas las ideas teóricas que rigen nuestro proyecto, en el capítulo cinco presentamos la simplificación realizada para desarrollar una simulación en que se representen las operaciones básicas que se realizan entre regiones de memoria. Presentaremos las ideas que inicialmente nos fueron surgiendo y que nos llevaron a crear el sistema de una determinada forma. También explicamos como queda la estructura del sistema a grandes rasgos para presentar a continuación el diseño del sistema.

En el siguiente capítulo explicaremos la implementación de la aplicación que hemos realizado y en el siguiente explicaremos una serie de pruebas que hemos realizado para comprobar el correcto funcionamiento de los modelos implementados.

Capítulo 3
Modelo RTSJ para la memoria Scoped.

La clase *MemoryArea* es la clase abstracta de la que heredan todas las clases de áreas de memoria: Heap, Inmortal, Scoped. La clase `ScopedMemory` es la clase abstracta de las clases que se utilizan para representar espacios de memoria con un tiempo de vida limitado. De ella heredan dos clases:

- `LTMemory`, que requiere que el tiempo de asignación sea directamente proporcional al tamaño del objeto que esta siendo asignado y
- `VTMemory`, donde la asignación puede producirse en un tiempo variable.

Ninguna de estas regiones está sujeta al colector de basura.

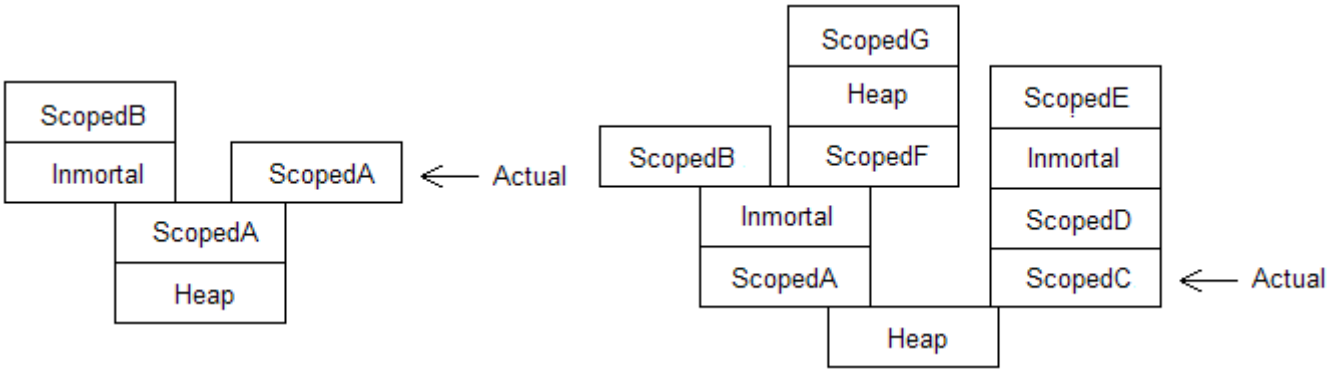
El mecanismo que se usa para asociar una memoria scoped a un thread es colocar el área como uno de los parámetros que se le pasa en el constructor del thread o ejecutar el método `enter()` de `ScopedMemory` desde el thread de tiempo real que se quiera asociar a dicho área de memoria scoped. Los objetos creados mientras está asociada una región `ScopedMemory` a un thread se crearán en dicho área. Una región de memoria scoped es válida mientras algún thread de tiempo real tenga acceso a ella. Cuando la última referencia a la región de memoria sea eliminada saliendo del thread o saliendo de su método `enter()`, entonces se ejecutarán los finalizadores para todos los objetos del área de memoria y ésta será colectada.

El mecanismo de recolección de la memoria scoped es distinto de la recolección del heap y las variables locales, y las regiones de memoria inmortal no tienen mecanismo de recolección porque los objetos que están en ellas viven mientras dura la aplicación. Por este motivo los objetos que están en un área de memoria inmortal o en el heap, no pueden referenciar a objetos que están en un área de memoria scoped, porque podrían dejar punteros colgantes y provocar excepciones. Entre regiones de memoria Scoped si puede haber referencias pero tienen que cumplir una serie de condiciones.

Un puntero colgante es una referencia a una dirección de memoria en la que antes había un objeto y por algún motivo esta memoria se ha liberado pero la referencia sigue existiendo, por lo que ahora el puntero queda apuntando a un objeto que se ha recolectado.

3.1 Anidamiento de regiones scoped

Si en una determinada pila se cambia el área de memoria actual a una que está por debajo de ella, mediante el método `executeInArea(),` y luego desde ese área se entra en otras regiones de memoria (respetando siempre las reglas de asignación y la regla del único padre), ya no hay una estructura de pila, si no que tiene forma de árbol o más bien de cactus. Y si la llamada a `executeInArea()` se hace con una región de memoria heap o de inmortal, entonces se crea una nueva pila con la región primordial. De esta forma se podrían producir situaciones en la pila como la de las figuras:



3.2 Regla del único padre y reglas de asignación

Cada vez que se apila una región de memoria scoped en la pila scope, se necesita comprobar que se cumple la regla del único padre, que obliga a que cada región de memoria scoped no tenga más de un padre. El padre de una región de memoria scoped para una pila que crece es:

- Si la región de memoria no está actualmente en ninguna pila, entonces no tiene padre.
- Si la región de memoria es la primera región scoped que hay en cualquier pila (la más inferior), entonces su padre es la región llamada "*scope primordial*".
- Para cualquier otra región scoped, su padre es la primera memoria scoped que esté por debajo de ella en la pila.

La regla del único padre afecta a todas las regiones de memoria scoped excepto a la región primordial que representa al heap o a la memoria inmortal. Los efectos de esta regla son que una vez que se le ha asignado un padre a una región scoped, ninguna de las operaciones descritas anteriormente puede cambiar ese padre y el orden impuesto por la primera asignación de padres de una serie de memorias scoped anidadas es el único orden permitido hasta que el control deje esa región de memoria scoped. Así que si un thread intenta entrar en una región de memoria scoped solo puede hacerlo en el orden de anidamiento establecido.

La pila scope funciona como un árbol en el que la raíz es el scope primordial (heap o inmortal) y cada uno de los nodos se corresponde con cada región de memoria scoped que esté actualmente en la pila de cualquier thread. Cada región de memoria scoped *"mem"* tiene una referencia a su padre *mem.padre*, que puede tomar distintos valores:

- otra área de memoria scoped,
- "noPadre" si no tiene padre o
- la región scoped primordial.

En cada apilado y desapilado que se realiza a la pila de un thread se tiene que verificar la regla del único padre. Si se quiere apilar una región de memoria *mem*, se tendría que realizar lo siguiente:

```
Precondición:
       mem.padre está fijado al padre correcto
       t.scopeStack es la pila scope del thread actual

si mem es una región scoped entonces
       padre = findFirstScope (t.scopeStack)
       si mem.padre = noPadre entonces
              mem.padre = padre
       si mem.padre != padre entonces
              lanzar excepción ScopedCycleException
       si no
              t.scopeStack.apila(mem)
              mem.contadorReferencias ++
```

La función "`findFirstScope (pila)`" encuentra la primera región de memoria de la pila que sea una región scoped empezando la pila desde arriba. Si no encuentra ninguna región scoped, devuelve la scoped primordial.

Cada vez que se desapila una región de memoria *mem* se realiza lo siguiente:

```
mem = t.scopeSatck.desapila()
si mem es una región scoped entonces
       mem.contadorReferencias –-
       si mem.contadorReferencias == 0 entonces
              mem.padre = noPadre
```

Junto con la regla del único padre hay unas reglas de asignación entre regiones de memoria scoped que se tienen que cumplir para evitar que se produzcan punteros colgantes.

- Una referencia desde un objeto de una región de memoria scoped hacia otro objeto en la misma región está permitida.
- Una referencia desde un objeto de una región de memoria scoped hacia otro objeto de otra región de memoria scoped que esté por debajo de la primera región en la pila está permitida.
- Una referencia desde un objeto de una región de memoria scoped hacia otro objeto de otra región de memoria scoped que esté por encima de la primera región en la pila está prohibida.

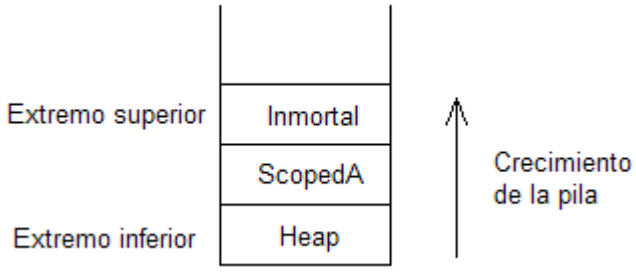
3.3 Contador de referencias y pila scope

Cada región de memoria scoped tiene un contador de referencias que lleva la cuenta del número de referencias externas a dicha área de memoria. Este contador se incrementa cuando se entra en una nueva memoria scoped a través del método `enter()`, cuando se crea un nuevo thread de tiempo real que usa esa región de memoria scoped o cuando se entra en una región scoped interior. El contador de un área de memoria scoped se decrementa cuando se vuelve del método `enter()`, cuando se sale del thread de tiempo real o cuando una región interna a dicha scoped vuelve de su método `enter()`.

Cuando el contador de referencias pasa de uno a cero, se ejecuta el método `finalize()` para cada objeto que haya en esa región de memoria y ésta se vacía. Esta región de memoria no puede utilizarse hasta que la finalización se haya completado.

Todos los threads de tiempo real tienen una pila asociada a ellos. Cuando se crea un nuevo thread de tiempo real se le puede pasar en su constructor un área de memoria, que se utiliza como base en la pila que tiene asociada. Si se ha utilizado un constructor que no utiliza un área de memoria como parámetro, entonces el área que se utiliza como base de la pila es el área desde donde se creó dicho thread de tiempo real.

Vamos a utilizar para las explicaciones una pila según el siguiente esquema:



Hay cuatro operaciones que afectan al crecimiento de la pila:

- El método `enter()` de objetos de la clase `MemoryArea`.

Para una región de memoria *`mem`*, que ejecuta el método `enter` `mem.enter(logic)` se produce lo siguiente:

```
Si cuando entre mem violase la regla del único padre
entonces
      lanzar la excepción ScopedCycleException
apilar mem en la pila del thread actual
ejecutar el método run() de logic, logic.run()
desapilar mem de la pila
```

- La construcción de un nuevo thread de tiempo real.

Si en el área de memoria *`mem`* se crea un nuevo thread de tiempo real con un área de memoria inicial *`imem`* pasaría lo siguiente:

```
Si mem es heap o inmortal entonces
      crear una nueva pila scope que contenga mem
si no
      iniciar una nueva pila scope que contenga la
      actual pila scope
para cada área de memoria scope en la nueva pila
      incrementar el contador de referencias
si imem es distinta del actual contexto de asignación
entonces
      apilar imem en la nueva pila
            (esto puede lanzar la excepción
            ScopedCycleException)
ejecutar el nuevo thread con su nueva pila
cuando el thread termina
      desapilar cada área de memoria que haya sido
      apilada por el thread
      para cada área de memoria que hay en la pila
            decrementar el contador de referencias
liberar la nueva pila
```

- El método `executeInArea()` de objetos de la clase `MemoryArea`.

Para una región de memoria *`mem`*, que ejecuta el método `executeInArea`, `mem.executeInArea(logic)` se produce lo siguiente:

```
Si mem es una instancia de heap o inmortal entonces
      crear una nueva pila que sólo contenga a mem
      hacer que la nueva pila sea la pila del thread
      actual

si no //mem es scoped
      si mem está en la pila del thread actual
      entonces
            empezar una nueva pila que contenga a mem y
            a todas las regiones que haya debajo de mem
            en la pila
            hacer que la nueva pila sea la pila del
            thread actual
      si no
           excepción InaccessibleAreaException
ejecutar logic.run()
restaurar la pila previa para el thread actual
descartar la nueva pila
```

- Los métodos `newInstance ()` o `newArray ()` para objetos de la clase `MemoryArea`.

Para una región de memoria *`mem`*, que ejecuta `newInstance()`, `mem.newInstance()` o `newArray()`, *`mem.newArray()`* pasa exactamente igual que en el apartado anterior, solo que en lugar de ejecutar el método `run` de `logic`, `logic.run()`, se ejecuta el constructor del objeto.

3.4 Declaración, constructores y métodos

La declaración de la clase abstracta `ScopedMemory` es la siguiente:

```
public abstract class ScopedMemory extends MemoryArea
```

Tiene cuatro constructores. Utilizan como parámetros el tamaño inicial de la región o un estimador de la clase `SizeEstimator` cuando no se sabe con seguridad la cantidad de memoria que se necesita. También se puede utilizar otro parámetro, un objeto que implemente la interfaz `Runnable` que utilizará este área de memoria como su área de memoria inicial.

Como la clase `ScopedMemory` es abstracta no se pueden crear instancias de ella, así los constructores y los demás métodos los heredan las dos subclases que tiene: `VTMemory` y `LTMemory.` Los métodos de la clase `ScopedMemory` que más nos interesan son los siguientes:

- `enter()`:

```
public void enter() throws ScopedCycleException

public void enter (java.lang.Runnable logic) throws
ScopedCycleException
```

Ambos métodos asocian este área de memoria al actual thread de tiempo real durante la duración de su método `run()`, que es disparado por este método. El primero utiliza el objeto `Runnable` que se pasó en el constructor del área de memoria y el segundo utiliza el objeto `Runnable` que se le pasa como paramétro en este metodo `enter()`. Durante el periodo de tiempo que dure la ejecucion del método `run()` todos los objetos que se creen, lo harán desde esta zona de memoria, hasta que se salga del metodo `enter()`. Ambos métodos lanzarían una excepcion en tiempo de ejecucion si el metodo es llamado desde un thread distinto de `RealTimeThread` o `NoHeapRealTimeThread`. También se lanzaría una excepción si no se cumple la regla del único padre.

- `executeInArea()` :

```
public void executeInArea(java.lang.Runnable logic)
throws InaccessibleAreaException
```

Este método se hereda de la clase `MemoryArea`. Ejecuta el método `run()` del objeto `Runnable` pasado como parámetro usando este área de memoria como el actual contexto de ubicación del objeto `Runnable`. En un área de memoria scoped, este método se comporta como si se hubiera movido el contexto de asignación de la pila scope.

- `getReferenceCount()` :

```
public int getReferenceCount()
```

Devuelve un entero: el contador de referencias, que lleva una cuenta del número de threads que pueden tener acceso a esta región scope.

- `newArray()` y `newInstance()`:

```
public java.lang.Object newArray (java.lang.Class
type, int number)
throws IllegalAccessException,
InstantiationException

public java.lang.Object newInstance (java.lang.Class
type)
throws IllegalAccessException,
InstantiationException

public java.lang.Object newInstance
(java.lang.reflect.Constructor c,
java.lang.Object[] args)
throws IllegalAccessException,
InstantiationException
```

El método `newArray()` crea un array con un número de elementos "`number`" del tipo "`type`" y lo asigna en la región de memoria actual. Los métodos `newInstance()` crean una instancia de un objeto y lo asignan a la región de memoria actual. Se puede pasar como parámetro el tipo del objeto o un constructor con un array de argumentos. Se pueden producir excepciones porque la clase o el inicializador no sean inaccesibles o porque la clase del objeto especificado no pueda ser instanciada por ser abstracta, por ser un array o porque su constructor lance una excepción.

Capítulo 4
Alternativas al modelo RTSJ

4.1 Problemas del modelo RTSJ

El modelo RTSJ tiene varios problemas, entre ellos se encuentran:

- Un modelo de programación poco familiar que hace complicada la tarea del programador.
- Condiciones de carrera en la ejecución de los programas.
- Una ejecución de programas no predecible en tiempo.

Las reglas de asignación y la regla del único padre crean distintas situaciones como por ejemplo tener regiones huérfanas, que el padre de una región cambie a lo largo de la vida de ésta…lo que provoca un modelo de programación poco familiar para el programador. Es difícil decidir cuando usar la memoria scoped del modelo RTSJ y una vez decidido eso, es difícil usarla correctamente, ya que tiene muchas reglas y muchos casos en los que se pueden lanzar excepciones.

Se pueden crear condiciones de carrera, como en la siguiente situación:

- Se crean dos regiones de memoria scoped A y B.
- Se crean dos threads de tiempo real H_1 y H_2.
- H_1 entra en las regiones en el orden A, B.
- H_2 entra en las regiones en el orden B, A.

Cuando se ejecute el programa, se puede provocar distintos comportamientos, dependiendo de las condiciones de carrera:

- Si H_1 entra en A y B antes de que H_2 entre en B, H_2 viola la regla del único padre ya que B se encontraría con dos padres: A y el padre que le correspondería al ser entrada por H_2.

Pila scope de H1 Pila scope de H2

H2 viola la regla del único padre al entrar en B

- Si ocurre al revés, que H_2 entre en B y A antes de que H_1 entre A, pasaría lo mismo, solo que sería H_1 quien violase la regla del único padre.

Pila scope de H2 Pila scope de H1

H1 viola la regla del único padre al entrar en A

Si H_1 y H_2 entran respectivamente en A y B y permanecen un tiempo en ellas, al cabo de ese tiempo puede ocurrir que:

- Cuando H_1 intente entrar en B violará la regla del único padre.

Pila scope de H2 Pila scope de H1

Al intentar entrar H1 en B viola la regla del único padre

- Cuando H_2 intente entrar en A violará la regla del único padre.

Pila scope de H1 Pila scope de H2

Al intentar entrar H2 en B viola la regla del único padre

Estas cuatro situaciones provocarían que se lanzase la excepción `ScopedCycleException()`, y el programador se encontraría con 4 posibles errores en tiempo de ejecución lo que haría muy complicado repasar el código para corregirlo. Además con el mismo código, hay otras cuatro situaciones en las que el programa sí funcionaría correctamente:

- H_1 entra en A y B, sale de ambas y luego H_2 entra en B y A.
- H_2 entra en B y A, sale de ambas y luego H_1 entra en A y B.
- H_1 entra en A y B, y sale de B antes de que H_2 entre en ella, y luego H_1 sale de A, antes de que H_2 intente entrar en ella.
- H_2 entra en B y A, y sale de A antes de que H_1 entre en ella, y luego H_2 sale de B, antes de que H_1 intente entrar en ella.

Todo esto además, provoca distintas relaciones de padre-hijo: cuando se está ejecutando el thread H_1, A es padre de B, pero cuando se ejecuta H_2, B es padre de A, y por las reglas de asignación, habrá asignaciones que cuando se ejecute H_1 estarán permitidas, pero cuando se ejecute H_2 no y viceversa.

Esto hace complicada la tarea del programador y además provoca indeterminismo, en este caso, la ejecución de un mismo programa puede tener diferentes resultados. También se puede producir indeterminismo debido a la pila de regiones scoped. Hemos mencionado antes las cuatro operaciones que afectan al crecimiento de la pila, por tanto cada vez que se produce alguna, se tienen que ejecutar las instrucciones asociadas a cada una de ellas, lo que provoca una gran sobrecarga y además impredecible en tiempo porque depende del estado de la pila en cada momento.

Para intentar resolver estos vamos a estudiar varios modelos que hacen algún cambio respecto al modelo original.

4.2 Suprimir la pila scope

Para eliminar los ciclos que se pueden producir, como los que hemos visto en el apartado anterior, este modelo propone cambiar la regla del único padre. Define que "**el padre de una región de memoria scoped es el área de memoria**

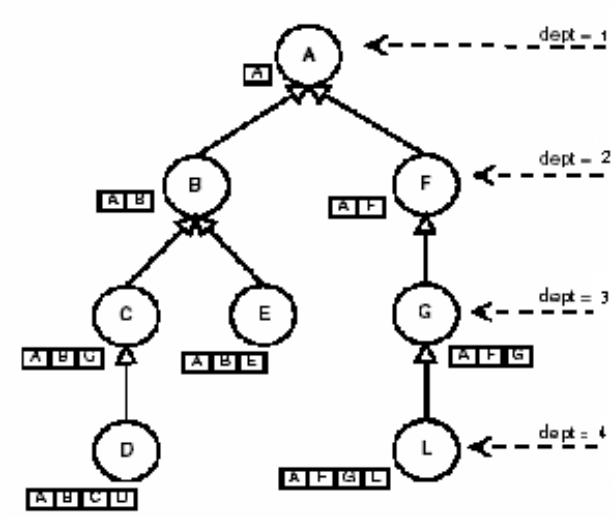
en la cual el objeto que representa el área de memoria scoped es asignado" (Bibliografía.4)

De esta forma, el padre de una región se asigna cuando se crea la región y no cambia a lo largo de su vida. Así, situaciones como las del ejemplo anterior que provocan ciclos no podrían producirse, ya que en el ejemplo ambas regiones fueron creadas en el heap, por tanto el heap es el padre de ambas y los punteros entre una y otra no estarían permitidos. Estas situaciones de referencias prohibidas son conocidas antes de la ejecución del programa y por eso no pueden provocar ciclos.

Consideremos otro ejemplo: tenemos dos regiones de memoria A y B; A ha sido creada en el interior del heap y B en el interior de A, por tanto A es el padre de B. Consideremos también dos threads de tiempo real H_1 y H_2; supongamos que H_1 entra en A y H_2 entra en B. Si luego H_1 quisiera entrar en B o H_2 quisiera entrar en A, la regla del único padre no sería violada, a diferencia de lo que ocurre en el modelo RTSJ.

En lo referente a la excepción `ScopedCycleException()` tenemos la siguiente situación: la pila scope asociada al thread H_1 incluye A y B. La pila scope asociada a H_2 incluye sólo A. Entonces, incluso si H_2 hubiera entrado en B antes que en A, las referencias de objetos de A, a objetos de B son punteros peligrosos y se pueden quedar colgados, por eso no están permitidos. Las referencias de objetos de B a objetos de A si están permitidas. Si se crea una referencia de un objeto de B a un objeto de A, esta situación es estable independientemente del thread de tiempo real que haga la referencia.

Con este modelo se puede usar el nombre de las regiones para comprobar las asignaciones ilegales, se puede nombrar como A, a la región de memoria scoped A, AB a la región de memoria B que se creó en A, ABC, ABD… a las regiones de memoria C y D, que se han creado en B y así sucesivamente. De esta forma se puede eliminar la pila scope.

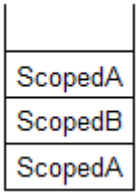


Este modelo tiene las ventajas de que la regla del único padre se convierte en trivial, no se producen ciclos, y la relación de parentesco no cambia durante la vida de la región por lo que no se tiene que chequear todos los ancestros de una región cada vez que se entra en una región nueva. Además al eliminar la pila scope los chequeos de asignaciones ilegales se hacen por nombre, lo que simplifica las operaciones de chequeo en cada asignación y las operaciones que afectan al crecimiento de la pila, con lo cual se reduce la sobrecarga y los programas son predecibles en tiempo.

Pero también presenta sus inconvenientes, este modelo pierde poder de expresividad con respecto al modelo RTSJ, ya que no permite referencias entre dos regiones AB y AC, creadas dentro de una misma región A, a diferencia del modelo RTSJ, que si las permite. Solo estarían permitidas las referencias desde objetos de la región AB o AC a la región A.

4.3 Utilizar la regla del único padre de forma local

Hemos pensado que una forma de evitar las condiciones de carrera en el modelo RTSJ, con lo que se volviese a ganar en expresividad, fuese un nuevo modelo en el que se modifique la regla del único padre para que solo se tenga que verificar localmente, es decir en una misma pila. Así en un hilo no se puede entrar en A, luego en B y otra vez en A puesto que como hemos visto se podría hacer una referencia de A a B, de B a A y A quedarse con un puntero colgante al recolectar B.



T1

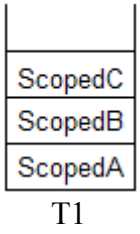
En cambio si que se podría hacer que un hilo T1, entrase en B y en A y otro hilo T2 en C y en A. Así A tendría dos padres, por T1 B y por T2 C, por T1 se podría hacer una referencia desde A a B, y en T2 se podría hacer otra desde A hasta C. Así no hay ningún problema, y cuando en T1 y en T2 se salgan de A, el reference count de A bajará de 2 a 0 y A podrá ser recolectada.

T1 T2

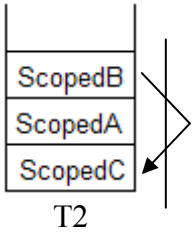
El problema que se encuentra este modelo es que una vez que en un hilo se va a entrar en una región, no se puede entrar en otra región que este por debajo suya (un antecesor) en cualquier otro hilo. Por ejemplo si T1 entra en A y luego en B, creando una referencia de B a A y T2 entra en B y decide entrar en A, T2 no puede hacer una referencia de A a B, puesto que si la hiciese, según en el orden que fuésemos a salir, A o B se quedarían con punteros colgantes.

Otro posible ejemplo:

Si en T1 entro en A, luego en B, y luego en C.



En T2 una vez que entro en C y luego en A y en B, me tengo que asegurar que al hacer referencias en T2 por ejemplo desde B no puede ser hacia un elemento que en la pila de T1 este por encima de B, es decir C.

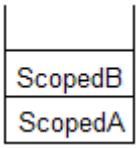


Pues si lo hiciese puede darse el caso de que este puntero en T2 se quede colgando, siempre que recolecte C antes de B (salir en T1 de C y salir en T2 de todas las regiones).

Así que para solucionar el problema de los punteros colgantes, necesitaría información global a todos los hilos para que me indicase si en un hilo, en el momento de hacer una referencia a otra región, esa región podría ser una hija (por encima) en algún otro hilo.

Esta información consiste en que cada vez que se entre en una región, marcar a todos los padres para que sepan que la tienen como hija, y que en un futuro no pueden referenciar a esta hija en otro hilo.

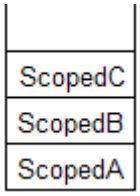
Así para el ejemplo anterior de T1:
En el momento de la figura tendríamos la tabla:



T1

	A	B
A		X
B		

Y al entrar T1 en C:



T1

	A	B	C
A		X	X
B			X
C			

Así el pseudocodidgo que habría que añadir a la hora de comprobar si una asignación es ilegal entre dos regiones de tipo Scoped es:

```
boolean asignaciónIlegal (RegionOrigen, RegionDestino) {
      Devolver (   indice(RegionOrigen)  >  indice(RegionDestino)  o
                   tablaMarcada(regionOrigen,regionDestino)   )
}
```

La ventaja de este modelo es que elimina parcialmente la regla del único padre. Ahora en un hilo se puede entrar en una región sin tener que preocuparse de si se ha entrado en todas las anteriores con el mismo orden que otro hilo.

Las desventajas son que aunque en un principio si se pensaba, este modelo no soluciona las condiciones de carrera, sigue sin ser determinista, ya que según entren en unas regiones en un orden un hilo u otro puede no lanzarse excepción, o lanzar un hilo u otro la excepción.

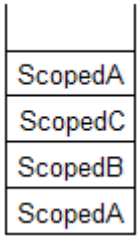
Y además de que no se ha superado el problema de condiciones de carrera, no se mejora la expresividad, pues aunque si es verdad que se puede entrar en regiones con mayor facilidad que en RTSJ (mejor modelo de programación) se le restringe aún más al programador el poder hacer referencias.

Así en un programa en las que se entran repetidas veces en las regiones y se hacen pocas referencias entre ellas, este modelo es mucho más eficiente que el RTSJ. En cambio el modelo RTSJ es más indicado si lo que se va a hacer es muchas referencias en pocas regiones.

4.4 Suprimir la regla del único padre

Otra posible solución para eliminar las excepciones lanzadas a causa de los ciclos y el comportamiento no determinista de los sistemas de tiempo real consiste en utilizar una generalización del modelo anterior, eliminando totalmente la regla del único padre, permitiendo los ciclos entre regiones de memoria scoped. Para evitar los punteros colgantes entre objetos del interior del heap o de inmortal, se siguen utilizando las reglas de asignación del modelo RTSJ. Por tanto no habrá excepciones por ciclos entre objetos de regiones scoped ni entre objetos de heap o inmortal. (Bibliografía, 4 y 5)

Con este modelo puede darse el caso del dibujo, en el que la tarea T1 ha entrado dos veces en la región A por lo que podría hacer una referencia la región A a la región C (está por debajo de la pila) y de la región C a la región A (también está por debajo).



T1

Para utilizar esta solución se tiene que modificar el uso del contador de referencias de una región scoped. Antes, simplemente era un contador que se incrementaba y se decrementaba en situaciones que tenían que ver con la creación o la destrucción de una región memoria scoped. En esta aproximación se introduce una nueva estructura de datos para cada región de memoria scoped, que consistirá en una lista que tenga en cuenta todas las regiones scoped que deben ser recolectadas antes que ella, es decir todas sus regiones internas. Así los ciclos pueden ser permitidos porque son conocidos en cualquier instante y el espacio para almacenar esta estructura es mínimo.

Usando una lista de regiones internas, conocemos todos los ciclos entre regiones scoped en el sistema en un instante determinado y así podemos detectar las regiones scoped que componen un ciclo que no está siendo usado por ningún thread y en este caso puede ser recolectado.

Permitiendo ciclos entre regiones scoped en la pila scope (eliminando la regla del único padre), las reglas de asignación y las instrucciones asociadas a cada operación que afecta al crecimiento de la pila, pueden por sí mismas evitar la creación de posibles punteros colgantes. Por lo tanto, se puede eliminar la excepción `ScopedCycleException()`, porque ahora los ciclos entre regiones están permitidos.

Este modelo tiene la ventaja de que al permitir los ciclos se simplifica y además ayuda al programador, ya que ahora no tiene que tener en cuenta las condiciones de carrera. Esto tiene una gran ventaja que hace que la expresividad sea mayor que en el modelo RTSJ y que en el modelo en el que se elimina la pila scope. Consideremos un ejemplo en el que dos regiones B y C han sido creadas en el interior de la región A. Consideremos además que un thread de tiempo real ha entrado en las regiones con el siguiente orden: A, B, C y A. Ahora estarían permitidas las referencias desde objetos en la región de memoria B a objetos que estén en C y referencias desde objetos de C a objetos en B.

Esto, evidentemente, presenta el problema de que se aumenta mucho la complejidad y la sobrecarga del colector de basura, ya que tiene que preocuparse ahora, además, de controlar que ciclos están siendo usados por algún thread y cuales no, para poder recolectar las regiones de memoria que aunque tengan ciclos, no estén siendo usadas por ningún thread.

Este modelo lo empezamos a implementar, pero como todavía es un modelo en estudio, decidimos centrarnos en otros modelos más consolidados como puede ser el de RTSJ o el modelo que no utiliza la pila, y conseguir que funcionasen correctamente.

De todas formas, este modelo a la hora de implementar solo se diferencia del RTSJ en el que cada región lleva implícita una tabla, en el que se marcan las regiones internas (por encima de la pila) que hay que recolectar antes de recolectarla a ella misma por que podrían hacerle una referencia. Este

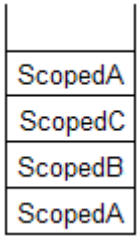
mantenimiento de la tabla cada vez hace que aumente considerablemente la sobrecarga.

El otro cambio que hay que hacer es en el momento de decidir si se puede recolectar una región Scoped, ya no vale comprobar que únicamente su reference count baje de uno a cero, sino que cada vez que se sale de una región además de decrementarse el referente count y desapilar la región de la pila, se tiene que ejecutar el siguiente algoritmo.

CodigoRegion (i):

```
Si la region i tiene alguna marca
        Para cada region j marcada
                Si referenceCount (j)=0 entonces
                        QuitarMarca(i,j);
                        CodigoRegion(j);
                fsi;
        finPara
        Si (referenceCount(i) =0 y  marcas(i)=0) entonces
                 recolecto región  i
        fsi;
fsi;
finCodigoRegion
```

EJEMPLO 1



T1

Su tabla en este momento es:

Ref.Counter		A	B	C
2	A		X	X
1	B	X		X
1	C	X		

Y se empieza a salir de regiones (A,C,B,A):

CodigoRegion A

Ref.Counter		A	B	C
1	A		X	X
1	B	X		X
1	C	X		

CodigoRegion C

Ref.Counter		A	B	C
1	A		X	X
1	B	X		X
0	C	X		

CodigoRegion B

Ref.Counter		A	B	C
1	A		X	X
0	B	X		‘X’
0	C	X		

->CodigoRegion C

Ref.Counter		A	B	C
1	A		X	X
0	B	X		
0	C	X		

CodigoRegion A

Ref.Counter		A	B	C
0	A		‘X’	X
0	B	X		
0	C	X		

->CodigoRegión B

Ref.Counter		A	B	C
0	A			X
0	B	‘X’		
0	C	X		

->CodigoRegion A (Recolecto B)

Ref.Counter		A	B	C
0	A			‘X’
0	B			
0	C	X		

->CodigoRegión C (Recolecto B,Recolecto A)

Ref.Counter		A	B	C
0	A			
0	B			
0	C	‘X’		

->CodigoRegion A (Recolecto B, Recolecto A, Recolecto C)

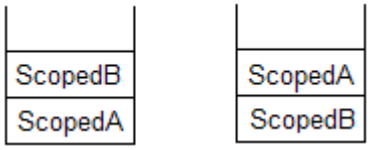
Ref.Counter		A	B	C
0	A			
0	B			
0	C			

Nada.

->Recolecto C. ->Recolecto A.-> Recolecto B.

EJEMPLO 2

En este ejemplo se usan dos hilos:



Así hay cuatro formas de salir de las regiones:

1ª FORMA

B(T2) A A B (T1)

CodigoRegion B

Ref.Counter		A	B
2	A		X
1	B	X	

CodigoRegion A

Ref.Counter		A	B
1	A		X
1	B	X	

CodigoRegion A

Ref.Counter		A	B
0	A		X
1	B	X	

CodigoRegion B

Ref.Counter		A	B
0	A		X
0	B	‘X’	

->CodigoRegion A (RecolectarB)

Ref.Counter		A	B
0	A		'X'
0	B		

->CodigoRegion B (Recolectar B,Recolectar A)

Ref.Counter		A	B
0	A		
0	B		

->RecolectarB ->Recolectar A

2ª FORMA

B(T2) A B(T1) A

Codigo B

Ref.Counter		A	B
2	A		X
1	B	X	

Codigo A

Ref.Counter		A	B
1	A		X
1	B	X	

Codigo B

Ref.Counter		A	B
1	A		X
0	B	X	

Codigo A

Ref.Counter		A	B
0	A		
0	B	‘X’	

->CodigoRegion B (RecolectarA)

Ref.Counter		A	B
0	A		‘X’
0	B	X	

->CodigoRegion A (Recolectar A,Recolectar B)

Ref.Counter		A	B
0	A		
0	B		

Nada.

->RecolectarA ->Recolectar B

3ª FORMA

A(T1) B A(T2) B

CodigoRegion A

Ref.Counter		A	B
1	A		X
2	B	X	

CodigoRegion B

Ref.Counter		A	B
1	A		X
1	B	X	

CodigoRegion A

Ref.Counter		A	B
0	A		X
1	B	X	

CodigoRegion B

Ref.Counter		A	B
0	A		X
0	B	‘X’	

->CodigoRegion A (RecolectarB)

Ref.Counter		A	B
0	A		‘X’
0	B		

->CodigoRegion B (Recolectar B,Recolectar A)

Ref.Counter		A	B
0	A		
0	B		

Nada.

->RecolectarB ->Recolectar A

4ª FORMA

A(T1) B B A(T2)

CodigoRegion A

Ref.Counter		A	B
1	A		X
2	B	X	

CodigoRegion B

Ref.Counter		A	B
1	A		X
1	B	X	

CodigoRegion B

Ref.Counter		A	B
1	A		X
0	B	X	

CodigoRegion A

Ref.Counter		A	B
0	A		
0	B	‘X’	

->CodigoRegion B (RecolectarA)

Ref.Counter		A	B
0	A		‘X’
0	B	X	

->CodigoRegion A (Recolectar A,Recolectar B)

Ref.Counter		A	B
0	A		
0	B		

Nada.

->RecolectarA ->Recolectar B

4.5 Comparativa

Cada modelo estudiado tiene diferente comportamiento respecto a diferentes problemas y situaciones como son las condiciones de carrera, la

simplificación del modelo de programación, el tiempo de sobrecarga, el determinismo en tiempo y el poder de expresividad. Comparando los dos modelos propuestos con el modelo RTSJ podemos obtener la siguiente tabla sobre el comportamiento de cada modelo:

	Condiciones de carrera	Modelo de programación	Sobrecarga	Determinismo en tiempo	Poder de expresividad
Sin pila	Mejor	Mejor	Mejor	Mejor	Peor
Regla único padre local	Igual	Mejor	Igual	Igual	Peor
Sin regla del único padre	Mejor	Mejor	Peor	Igual	Mejor

Excepto en la solución que utiliza la regla del único padre de forma local, en las otras se eliminan las condiciones de carrera, además puesto que en una los ciclos entre regiones son imposibles y en la otra están permitidos, se elimina también la excepción `ScopedCycleException()`.

La solución basada en la eliminación de la pila y la redefinición de la regla del único padre presenta las siguientes ventajas:

- La regla del único padre se vuelve trivial, lo que simplifica la definición de la memoria scoped.
- No es necesario chequear todos los ancestros de una región de memoria cada vez que un thread de tiempo real cambia su contexto de asignación.
- Los chequeos de asignaciones ilegales están basados en el nombre, lo que reduce la sobrecarga y hace que los programas sean predecibles en el tiempo.

La solución basada en la utilización de la regla del único padre local presenta las siguientes ventajas:

- Se elimina la regla del único padre cuando existen varios hilos.
- Se mejora el modelo de programación al no tener que preocuparnos por la excepción de tipo `ScopedCycleException` entre varios hilos.

La solución basada en la eliminación de la regla del único padre presenta también diferentes ventajas:

- Eliminar esta regla simplifica la definición de la memoria scoped.
- No es necesario chequear todos los ancestros de una región de memoria cada vez que un thread de tiempo real cambia su contexto de asignación.
- Incrementa el poder de expresividad porque permite ciclos que hacen que el programador tenga más flexibilidad al usar la memoria scoped.

También decir que este es un campo de investigación que para nada esta cerrado. Lo ideal seria un modelo en el que se superasen las condiciones de carrera, se mejorase el modelo de programación eliminando la regla del único padre, la sobrecarga en tiempo de ejecución fuese mínima, totalmente predecible con operaciones de tiempo constante y la expresividad se viese mejorada pudiendo hacer asignaciones entre regiones a nuestro antojo.

Capítulo 5
Especificación y diseño

5.1 Primeras ideas

- Se han presentado tres **modelos** de trabajo: el modelo RTSJ, el modelo sin pila y el modelo sin regla padre. Cada uno de estos modelos podría tener un tratamiento independiente ya que las funciones y métodos que se incluirán en cada uno son bien diferentes.
- La unidad de trabajo que nos interesa son las **Regiones**. Dependiendo del modelo de trabajo, se hace un tratamiento diferente de ellas. Por ejemplo, en el modelo sin pila, el nombre de la región dependerá de su padre y eso es distinto de lo que ocurre con otros modelos.
- En toda la parte teórica se habla de **threads**. Los threads necesitan de las regiones para almacenar el valor de sus variables, necesitan un espacio de memoria. Nosotros no vamos a utilizar threads como tal, es decir, no vamos a utilizar objetos de la clase Thread, sino que vamos a utilizar **Tareas** que van a simular el comportamiento de los threads. Cada tarea que se realice, necesita un espacio de memoria para contener a sus variables y en caso de tener más de una tarea, también deberán pelearse por saber quien llega antes si lo hacen de forma simultánea.
- Como cada modelo tiene diferentes reglas y diferentes algoritmos que lo rigen, cada modelo deberá tener una definición diferente de las regiones y las tareas. Es decir, cada modelo tendrá una clase que implemente las regiones de memoria de ese modelo y otra clase que implemente las tareas (y las operaciones que estas realizan sobre las regiones) del modelo.

Nos planteamos el sistema como lo siguiente:

- Un modelo tiene regiones y tareas.
- Las tareas se crean y cuando entramos en ellas (al ejecutar su método `enter()`) es el momento en el que necesitamos una región que nos proporcione un lugar donde ejecutarse.
- Las regiones por tanto son espacios de memoria (es lo que en la teoría llamábamos regiones Scoped por ejemplo) en los que se producen operaciones y se crean variables en su interior.

A nosotros no nos interesa las operaciones que se realicen dentro de cada región, es decir, no nos interesa el cuerpo de las tareas, lo que realmente nos interesa es saber las relaciones que se producen entre las distintas regiones. Por ello, no vamos a hacer un sistema con operaciones particulares o con fragmentos de código, ni siquiera vamos a utilizar Thread en realidad, sino que vamos a suponer que existen programas que realizarían dichas operaciones y nosotros vamos a implementar realmente las operaciones entre regiones, y todo lo que motiva este proyecto: quien es el padre de una región de memoria concreta y ver si se vulneran o no las reglas de cada modelo.

5.2 Estructura

Tenemos principalmente tres tipos de objetos: modelos, tareas y regiones.

Modelos

No son una estructura de datos como tal, en el sólo se tendrán que incluir operaciones relativas al funcionamiento general del sistema para un modelo concreto. Esto se ve de forma más clara mediante un ejemplo: En el modelo RTSJ tenemos una pila de regiones por tarea. El modelo se encargará de observando estas pilas, obtener información acerca de por ejemplo si una referencia es prohibida entre dos regiones o si podemos recolectar una región. En cada modelo se tendrá sus particularidades dependiendo de las reglas que lo rijan.

Tareas

Las tareas son las que nos van a proporcionar concurrencia. Una tarea va a intentar acceder a una región de memoria mediante el método *enter()*, saldrá de esa región de memoria mediante *exit()*, se encargará de darnos información de si existen asignaciones ilegales o de si se viola la regla del único padre. Las tareas, junto con las regiones, van a ser la parte fundamental de este sistema.

Regiones

Las regiones de memoria (Heap, Inmortal y Scoped) las vemos como estructuras de datos que contendienen información relativa a su estado (dependiendo del modelo que se use, porque cada modelo precisará de una información diferente). Esta información puede ser su nombre, su región padre, el número de referencias que se hacen a esta región desde alguna tarea…

A modo de resumen en la siguiente tabla se muestra como quedaría la estructura del sistema:

	RTSJ	Sin Regla Padre	Sin Pila
1 modelo	ModeloRTSJ	ModeloSinPadre	ModeloSinPila
1 tipo de tarea	TareaRTSJ	TareaSinPadre	TareaSinPila
1 tipo de región	RegiónRTSJ	RegiónSinPadre	RegiónSinPila

5.3 Conclusiones

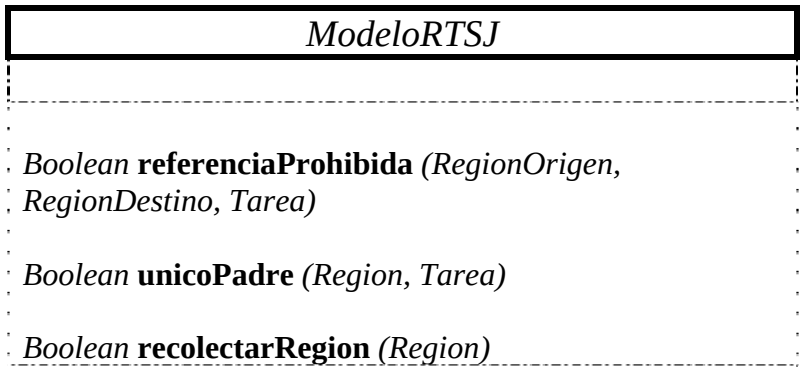
Hemos intentado que el problema quedara reflejado de forma clara y sencilla, y creemos que con esta simulación que evita usar complicaciones como el uso de Thread reales y que considera modelos con tareas y regiones, puede ser la forma de hacer de manera fácil y descriptiva el modelado del problema. Por tanto, vamos a proceder a describir ahora el diseño, en el que iremos conociendo más en profundidad las relaciones entre dichas clases.

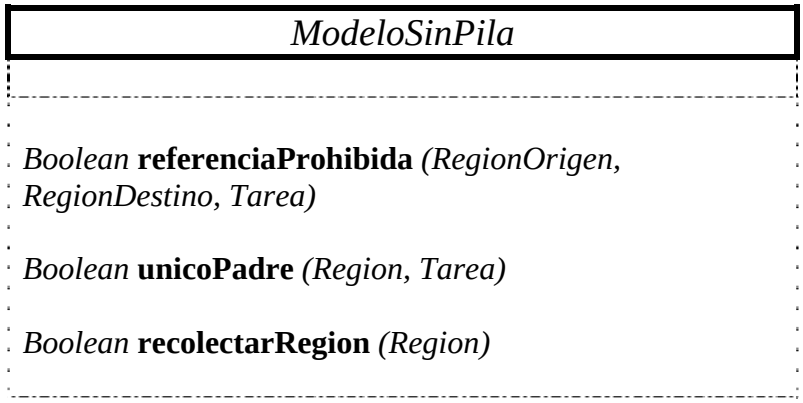
5.4 Diseño

Una vez que teníamos clara la especificación era el momento de pasar al diseño. Teníamos que abstraernos de todo, e incluir solamente las clases y métodos que nos interesaban para nuestro estudio, dejando de lado detalles que no nos importan como pueden ser el tamaño de las regiones, el tipo de los hilos o el número de bytes consumidos en esa región. Así cada modelo contendría los tres métodos que nos interesaban:

- Saber si es posible realizar una referencia entre dos regiones en una determinada tarea.
- Conocer si en una determinada tarea, una región puede violar la regla del único padre.
- Determinar si una región puede ser recolectada.

Estas son las tres cuestiones básicas de nuestro proyecto, y son propios de cada modelo.





Ahora teníamos que diseñar las restantes clases que ayudasen a dar respuestas a cada módulo. Lo propio que nos interesa de cada Región es su **nombre** único, que sirve para identificarla entre todas las regiones y su lista de **referencias** a otras Regiones. Esta lista contendrá las regiones desde las que se pueden crear referencias a la propia región, y que deben ser por lo tanto recolectadas antes que ella. También nos interesan otros parámetros ya dependientes del modelo:

- Para la *RegionRTSJ* nos interesan:

El **referenceCount** que es el número de tareas que tiene activa dicha región. Es decir, la cantidad de tareas que han entrado en esa región y aun no han salido.
La **regionPadre**, que es la región en la que se encontraba el hilo antes de entrar en la nueva region..

Una vez que tenemos definidos los atributos de dicha clase. Necesitamos métodos para trabajar con ellos:
añadeReferencia (RegionRTSJ r) añade una nueva región r a su lista de regiones referenciadas.
quitarReferencia (RegionRTSJ r) elimina la referencia a la región r de su lista.
contineReferencia (RegionRTSJ r) indica si existe dicha referencia a la región r.

También necesitamos una constructora **RegionRTSJ** (String nombre) que crea una nueva región vacía cuyo identificador es nombre.

Además necesitamos accesores para dichos atributos. Como son:
daNombre(), **daPadre**(), **daReferenceCount**(), **daReferencias**(), **esScoped**().

Y mutadores como **ponPadre**() y **ponReferenceCount**()

RegionRTSJ
nombre *String* **referencias** *Lista* **referenceCount** *Int* **padre** *RegionRTSJ*
RegionRTSJ *(nombre)* **añadeReferencia** *(Region)* **quitarReferencia** *(Region)* *Boolean* **contieneReferencia** *(Region)* **ponReferenceCount** *(Int)* *Int* **daReferenceCount** () *Region* **daPadre**() *String* **daNombre**() **ponPadre** (*Region*) *Lista* **daReferencias** ()

- Para la *RegionSinPila:*

Además de los ya mencionados **nombre** y lista de **referencias** son necesarios los atributos:
taskCounter: que indica la cantidad de hilos que tienen a la región como región actual en la que están trabajando.
childCounter: almacena el numero de hijos activos que tiene la región.
profundidad: es el nivel o profundidad de la región en el árbol de regiones
display: la lista con los nombres de los antecesores de la región.

Y métodos para trabajar con ellos:
añadeReferencia (RegionRTSJ r) añade una nueva región r a su lista de referencias.
quitarReferencia (RegionRTSJ r) elimina la referencia r de su lista.
contineReferencia (RegionRTSJ r) indica si existe dicha referencia a la región r.

También necesitamos una constructora **RegionSinPila**(String nombre, RegionSinPila padre) que crea una nueva región vacía cuyo identificador es nombre y cuya región padre es el parámetro padre.

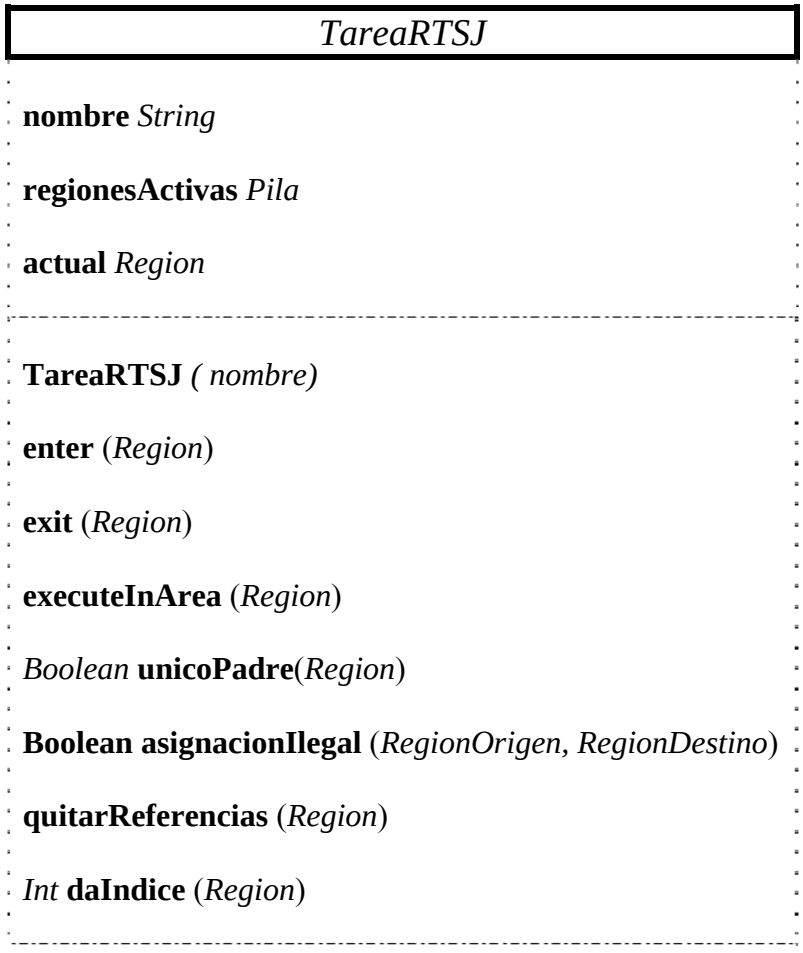
Como accesotes tenemos:
daNombre(),**daPadre**(),**daTaskCounter**(), **daChildCounter**(), **daReferencias**(), **daProfundidad**()**, daDisplay**()**.** Y mutadores como **ponTaskCounter**(Int t) y **ponChildCounter**(Int c).

- Para la clase *TareaRTSJ:*

Sólo necesitamos conocer el **nombre** identificativo de la Tarea, la lista de **regiones activas** y la región **actual** en la que ha entrado actualmente.

Como métodos usamos:
enter(RegionRTSJ) hace que, si no hay problemas, la Tarea entre en una región.
exit(RegionRTSJ) la tarea sale de una región.
executeInArea(RegionRTSJ) una determinada región activa se vuelva la actual.
asignacionIlegal (RegionRTSJ,RegionRTSJ) determina si se puede hacer una referencia entre dos regiones
daIndice(RegionRTSJ) devuelve el índice de una determinada región en su lista de regiones activas.
quitarReferencias(RegionRTSJ) recorre la lista de regiones quitando a una determinada región de la lista de las referencias.
unicoPadre(RegionRTSJ) determina si al entrar en una región voy a violar la regla del único padre.

También tenemos una constructora: **TareaRTSJ**(Nombre) que crea una nueva Tarea con identificador Nombre. Y métodos que sirven para visualizar información como **dameActual**() muestra por pantalla cual es la región actual de la Tarea.



- Para *TareaSinPila*:

Además del **nombre,** del **identificador** (número entero, que va a ser el índice de la lista de actuales) y **del número de Tareas** totales propios de cada Tarea, tenemos como atributos el **árbol** de Regiones, la lista de punteros **actuales** de todas las tareas, y la **Raíz** o región "primordial".Todos ellos compartidos por todas las tareas.

Como métodos:

enter(RegionRTSJ) hace que, si no hay problemas, la Tarea entre en una región.
exit(RegionRTSJ) la tarea sale de una región.
executeInArea(RegionRTSJ) una determinada región activa se vuelva la actual.
asignacionIlegal (RegionRTSJ,RegionRTSJ) determina si se puede dar hacer una referencia entre dos regiones
quitarReferencias (RegionRTSJ) recorre el árbol de regiones quitando a una determinada región de la lista de las referencias.

Como accesoras:

daIdentificador () devuelve el entero que identifica a la tarea.
dameActual() devuelve la región actual de la tarea.
daArbolRegiones() devuelve el árbol entero.

Como mutadoras:

ponRaiz (RegionRTSJ) pone a una región como raíz.
ponActual() pone a una región como la actual.

También tenemos la constructora: **TareaSinPila**(Nombre) que crea una nueva Tarea cuyo nombre va a ser el parámetro. Y el método **nuevoArbol**(),que se tiene que ejecutar una vez al principio, al crear el modelo, para crear un árbol vació y una lista de actuales vacía.

Y métodos que sirven para visualizar información como

dameActual() muestra por pantalla cual es la región actual de la Tarea.
regionesEnArbol() imprime por pantalla todas las regiones del árbol.

TareaSinPila
nombre *String*
identificador *Int*
<*static*> **arbolRegiones** A*rbol*
<*static*>**numeroTareas** *Int*
<*static*> **actuales** *Lista*
<*static*> **raiz** *Region*

TareaSinPila (*nombre***)**

enter(*Region*)

exit(*Region*)

executeInArea(*Region*)

Int **daIdentificador**()

Boolean **asignacionIlegal** (*RegionOrigen, RegionDestino*)

quitarReferencias (*Region*)

Region **dameActual** ()

daArbolRegiones()

<static> **nuevoArbol**()

ponActual(Region)

<static> **ponRaiz**(Region)

Region **dameActual**()

regionesEnPila()

PSEUDOCODIGOS DE LOS MÉTODOS *

(*)Solo incluimos los pseudocodigos de los métodos que no son triviales.

ModeloRTSJ

public boolean referenciaProhibida(RegionRTSJ ro, RegionRTSJ rd, TareaRTSJ t)
{
Si en la tarea t es una asignación ilegal hacer una referencia de ro a rd
devolver true;

sino añadir a la región rd una referencia de ro;
devolver false;
}

```
public boolean unicoPadre(RegionRTSJ r, TareaRTSJ t) {
    devolver si en la región t r puede tener un único padre;
}

public boolean recolectarRegion(RegionRTSJ r) {
    Si el referencecounter de r  >0 entonces
        informar que hay mas tareas con esa región activa
        devolver false

    Si el numero de referencias de r >0 entonces
        recorrer la lista de referencias de r informando de las que hay que
        recolectar antes
        devolver false

    sino devolver true;
}
```

TareaRTSJ

```
public TareaRTSJ(String n) {
    nombre = n;
    incremento numero de Tareas
    inicializo la lista de regiones Activas a una lista vacía;
 }


public boolean unicoPadre(RegionRTSJ r) {
    devolver  (r no tiene padre o el padre que ya tiene  no coincide con el
    actual,(su futuro padre))
}


public boolean asignacionIlegal(RegionRTSJ ro, RegionRTSJ rd) {
     devolver (ro no es el actual o
              indice(ro)<=indice(rd) o
              ro no es Scoped y rd es escoped)
}

public int daIndice(RegionRTSJ r){
    devolver el numero de antecesores de r hasta llegar a la base
}
```

public void enter(RegionRTSJ r) {
    si por entrar la region r se viola la regla del unico padre
        Lanzar ScopedException
    sino
        añado r a la pila
        a r le pongo de padre el actual
        actualizo actual a r
        incremento el reference counter de r
}

public void exit(RegionRTSJ r) {
    si r es el actual
        decremento el reference counter de r
        elimino r del arbol de regiones
        quitar las referencias de r
        actualizo actual al padre de r
}

public void executeInArea(RegionRTSJ r) {
    si r pertenece
        cambio el puntero actual a la region r
}

public void quitarReferencias(RegionRTSJ r) {
    recorrer lista de regiones activas
        si la region i contiene una referencia a r
            eliminar referencia
}

ModeloSinPila

public boolean referenciaProhibida(RegionRTSJ ro, RegionRTSJ rd, TareaRTSJ t)
{
    Si en la tarea t es una asignación ilegal hacer una referencia de ro a rd
        devolver true;

    sino   añadir a la región rd una referencia de ro;
            devolver false;
}

public boolean unicoPadre(RegionRTSJ r, TareaRTSJ t) {
    devolver true;
}

public boolean recolectarRegion(RegionRTSJ r) {

Si taskcounter de r >0 o childcounter de r >0 entonces
informar del error
devolver false

Si el numero de referencias de r >0 entonces
recorrer la lista de referencias de r informando de las que hay que recolectar antes
devolver false

sino quitar las referencias de r
eliminar r del arbol
devolver true;

}

TareaSinPila

public TareaSinPila(String n) {

nombre = n;
identificador = numeroTareas;
incremento numeroTareas;
añado a la lista de actuales un puntero a la raíz
incremento el taskcounter de la raíz

}

public static void nuevoArbol() {

inicializo árbol a un árbol vacío;
inicializo la lista de actual a una lista vacía;

}

public void enter(RegionSinPila r) {

decrementar task counter del actual
actualizar puntero actual
incrementar task counter del actual

}

public void exit(RegionSinPila r) {

decrementar task counter del actual
actualizar actual al padre de r
incrementer task counter del nuevo actual

}

```
public void executeInArea(RegionSinPila r) {
    si r pertenece a la lista de regiones activas
            decremento el task counter del actual
            cambio el puntero actual a la region r
            incremento el task counter del nuevo actual
}

public boolean asignacionIlegal(RegionSinPila ro, RegionSinPila rd) {
    devolver( ro no es la actual o
    X.display[Y.profundidad]<>Y.display[Y.pronfundidad])
}
```

Capítulo 6
Implementación

Implementación de los modelos propuestos

Una vez que ya teníamos hecho el diseño se empezó con la implementación. Se empezó haciendo el modelo RTSJ muy básico, para sólo un hilo y para regiones Scoped que entraban y salían en la pila haciendo referencias entre ellas. Solo se tenía una pila y se trabajaba siempre con la cima. Esta fue la versión 1.0 y fue básica para posteriores trabajos.

- PROBLEMA EXIT

El primer problema en la implementación de RTSJ que nos encontramos fue que teníamos la idea de que después de que un hilo hubiese ejecutado su código dentro de una región, (pudiendo entrar en otras regiones) tenia que decrementar su referencecount y desapilarse. ¿Pero como indicábamos que queríamos terminar el trabajo en una región? Para ello incluimos la sentencia exit y la añadimos en el diseño. Su función consiste en hacer las sentencias propias de finalización del método enter, es decir, que el hilo salga de la región, y que no hay que confundir con la función de recolectar la región por que no haya ningún hilo usándola. Siempre primero se sale de una región en un hilo y luego si se puede, se recolecta.

Una vez que disponíamos de la estructura necesaria para entrar y salir de regiones añadimos la posibilidad de hacer referencias entre ellas y poder recolectarlas si no hay ningún hilo usándolas. (Versión 1.5).

- PROBLEMA EXECUTE IN AREA

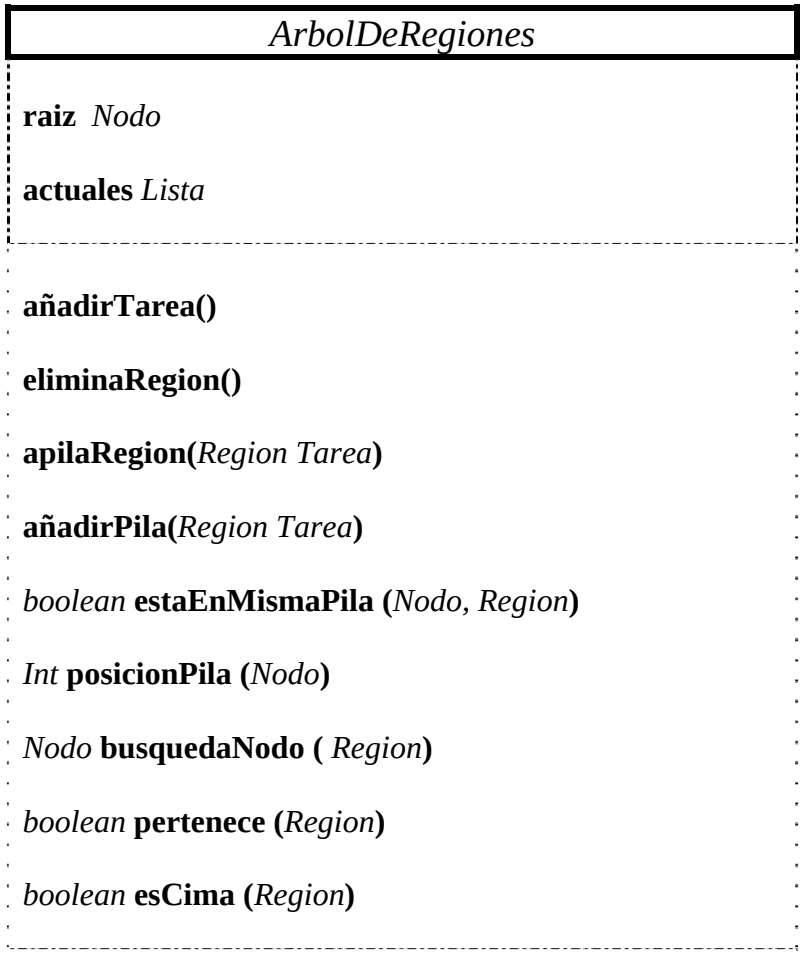
  En el libro de Andy Wellings se dice textualmente:

"El uso de executeInArea y luego entrar en una nueva memory area, produce como resultado que una pila simple no sea adecuada para guardar la traza de las scopes activas. En su lugar se tiene que usar un cactus o árbol."

Así, para introducir la instrucción execueteInArea tuvimos que desarrollar la estructura de datos cactus, haciéndose la clase auxiliar **Nodo** con atributos la propia región, el índice dentro de la pila y dos punteros a dos nodos, el siguiente nodo de la misma pila, y la siguiente pila y la clase **ArbolRegiones** en principio solo con el atributo raíz que era de la clase Nodo.

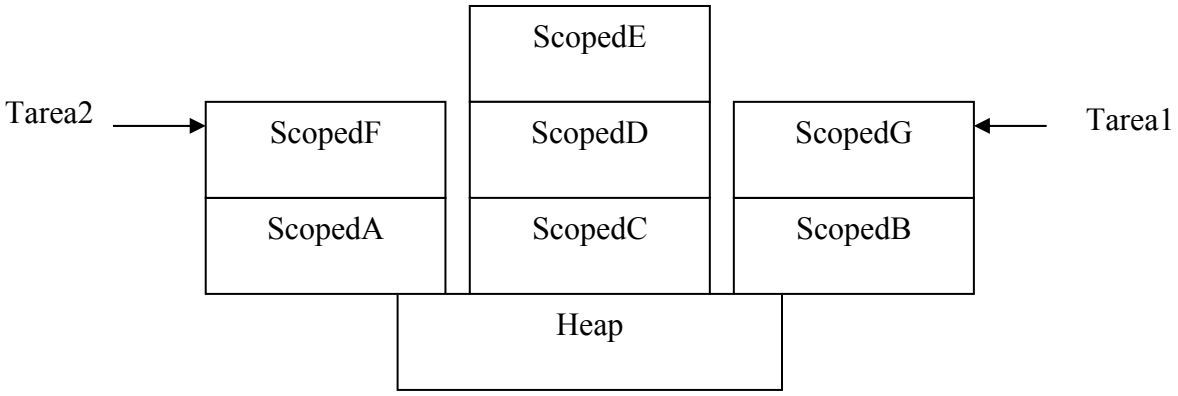
Empezamos con un árbol de Regiones simple, que solo constase de una sola pila. Fuimos refinando el modelo e introdujimos la instrucción executeInArea y la posibilidad de tener más de un hilo, por lo que también añadimos como atributo la lista de punteros de las regiones actuales de cada tarea.

Nodo
region *RegionRTSJ* **indicePila** *Int* **siguienteRegionPila** *Nodo* **siguienteRegion** *Nodo*
Nodo*(Region reg)* *Region* ***dameRegion****()* ***ponNodoPila****(Nodo r)* ***ponOtraPila****(Nodo r)* *Nodo* ***daSiguienteRegionPila****()* *Nodo* ***daSiguienteRegion****()* *Int* ***dameIndiceEnPila****()* ***ponIndiceEnPila****(Int id)*



Fue bastante tediosa la implementación del cactus, sobre todo actualizar el cactus a la hora de ir eliminando regiones (que no son cimas). A esta nueva versión la llamamos 2.0.

Aquí nos encontramos con el mayor problema, por que como bien aparece en el libro de Andy Wellings necesito saber las regiones actualmente activas, para hacer un executeInArea, (para hacer una asignación seguiría valiendo comprobar que la región esta por debajo) pero nos dimos cuenta de que el cactus no da esa información, pues ya las regiones activas no son solo las regiones que están por debajo del puntero hasta llegar a la base. Así puedo tener situaciones como la del dibujo en la que no se cuáles son para las tareas 1 y 2 sus regiones activas. ¿Podría hacer un executeInArea de Tarea1 o Tarea2 a E? No se si la región E esta activa para la Tarea 1, para la Tarea 2 o para las dos. Igualmente pasa con las regiones D y C.



Así decidimos volver a que cada Tarea trabaje con su lista (sería una pila si no se introdujese el executeInArea) de sus regiones activas con un puntero a la región en la que actualmente esta. Si decide hacer un executeInArea se comprueba que la región pertenezca a la lista y se cambia su puntero actual a dicha región. A partir de ahí puede volver a entrar en regiones que serán insertadas en la lista, y borradas cuando se salgan de ellas. El puntero actual se actualiza cada vez que se entra en una región, se hace un executeInArea o se sale de una región, en este caso actualizándose el puntero actual al padre de la región de la que se ha salido. Esta fue la versión 3.0.

- PROBLEMA HEAP-IMMORTAL

Decidimos solo trabajar con regiones de tipo Scoped, pues las relaciones entre ellas son las que realmente nos interesan. En el modelo SinPila tanto Heap como Inmortal no intervienen, pues nunca van a ser el padre de ninguna región Scoped. Y también para el modelo RTSJ resulta más claro y más intuitivo no añadirlas, además que su estudio no nos interesa a nosotros.

De todas formas, aunque su inclusión hace más lioso el modelo, sólo habría que hacer los siguientes cambios:

Meter en RegionRTSJ un atributo **tipo** (“Heap”,”Immortal”o”Scoped”).
Según este tipo tener en cuenta en TareaRTSJ y en ModeloRTSJ:

- No se puede hacer referencias de Heap o Inmortal a una región Scoped
- No se recolecta la región Inmortal, ni la región Heap (sí sus variables por el garbage collector)
- Al salir de una región, el puntero actual no va al padre de la región, sino al siguiente de la pila, pues sino se saltarían las regiones Heap e Inmortal

Con esta nueva versión, las secuencias consistían básicamente en:

Crear el modelo.
Crear distintas Tareas
Crear regiones
Repetir en paralelo con otras tareas:
    En esa tarea entrar en regiones
    Trabajar en esa región creando referencias de esas regiones a otras
    Salir de esas regiones
    Preguntar si se pueden recolectar

En paralelo también trabajamos el modeloSinPila, pero este nos dio menos problemas que el modeloRTSJ, pues es más sencillo, más claro y las estructuras de datos más simples. No ha tenido casi modificaciones desde la primera versión. Como en el modeloRTSJ primero nos centramos en un solo hilo. Creamos el árbol y su puntero actual a una región del árbol (Versión 1.3). Añadimos la posibilidad de crear referencias entre regiones y recolectarlas si no tienen hijos ni son las regiones actuales de ningún hilo (Versión 1.5). Cuando funcionaba perfectamente, incluimos la posibilidad de ser varias tareas las que usan el árbol con lo que incluimos la lista de punteros actuales y sus métodos correspondientes para actualizar la lista (Versión 2.0).

Una vez que ya estaban los dos modelos sin fallos (Versión 3.0) se integraron con la parte grafica y de ficheros. Así antes tenía que tener una clase prueba con el método Main donde se metía los comandos, y el resultado salía impreso por pantalla, pero ahora todo es mucho más fácil y cómodo, pues disponemos de botones y de información por pantalla .

CONCLUSION DE LA IMPLEMENTACIÓN

Tenemos que decir que el haber entendido el problema y el disponer de un diseño claro, con los pseudocodigos de los algoritmos nos hizo mucho mas fácil la parte de implementación. No queríamos que fuese engorrosa ni difícil de entender, sino que fuese muy simple y parecida a los pseudocodigos de los algoritmos. Queríamos un prototipo de pocas líneas con las ideas importantes abstraídas sobre las regiones de memoria y no perdernos con cosas no esenciales. Así no hemos tenido problemas por las ideas subyacentes de los modelos, sino con cosas más propias de interpretaciones humanas.

Implementación de la interfaz gráfica

1- Motivación

En una primera versión de nuestro proyecto no se incluía ningún tipo de interfaz gráfica que facilitara la visualización del proyecto. Lejos de ser una parte fundamental de nuestro proyecto, gracias a una interfaz gráfica podemos obtener

una visión más global de lo que ocurre con cada una de nuestras pruebas, además de facilitar la tarea de depuración del programa y ser más vistoso para un posible usuario.

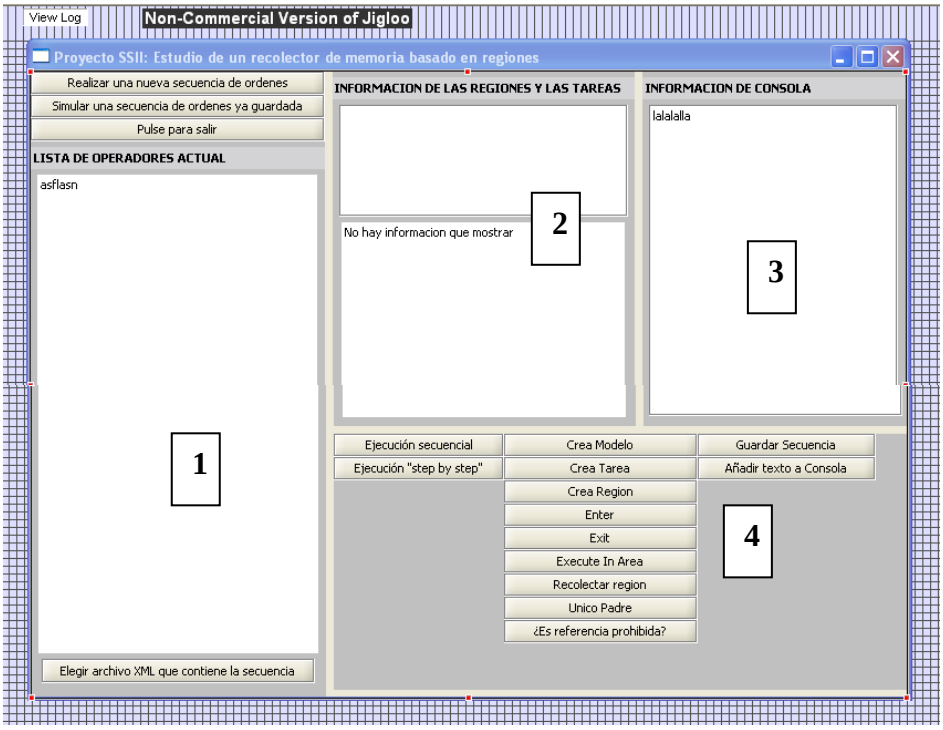
Inicialmente, todas nuestras salidas, toda la información que el usuario obtenía de la simulación era en línea de comandos, y alguno de nosotros incluso no entendíamos muy bien lo que aquella información quería decir.

Es por ello, que decidimos incluir una interfaz gráfica en paralelo al desarrollo del sistema ya que lo creíamos necesario. En esta interfaz, además, era necesario que hubiera una posible entrada/salida de datos con ficheros de algún tipo, y como explicaremos en el siguiente punto (con título "Implementación de entrada/salida con XML") esto se hizo mediante archivos XML. De esta forma, nuestro sistema podía ser probado sin tener que crear una clase para cada una de las pruebas que hiciésemos y nos mostraba unos datos que podrían tener persistencia.

2- Desarrollo

Para facilitar la creación de una interfaz gráfica y no usar librerías no incluidas en las distribución de Java 1.5 decidimos apoyarnos en la herramienta JIgloo para Eclipse. Esta herramienta, que tiene un editor de GUI's bastante estable, utiliza las librerías de Java y es de código libre.

Voy a comenzar a explicar como fue el proceso de desarrollo de la interfaz y como realizamos la pantalla principal. Para ello vamos a descubrir la pantalla ya finalizada y vamos a ir analizándola paso a paso. Aquí tenemos una captura completa de dicha pantalla:



Espacios necesarios

Necesitábamos 4 espacios diferentes para expresar toda la información necesaria para el sistema. Estos espacios o ventanas están destinados para lo siguiente:

- Ventana de secuencia (marcada con un 1 en la figura anterior): Es la ventana que contiene la información acerca de la secuencia de operaciones que se está realizando y es aquella en la que incluiremos los botones referidos a la carga de la secuencia o al comienzo de una secuencia desde el principio.
- Ventana de datos (2): En esta ventana se muestran los datos relativos a las regiones y a las tareas que actualmente existen en el modelo.
- Ventana de consola (3): Mostramos la información que mostraríamos por consola de comandos al realizar cada operación.
- Ventana de comandos (4): Es la ventana que contiene los botones para manejar la ejecución y para crear diferentes secuencias y guardarlas.

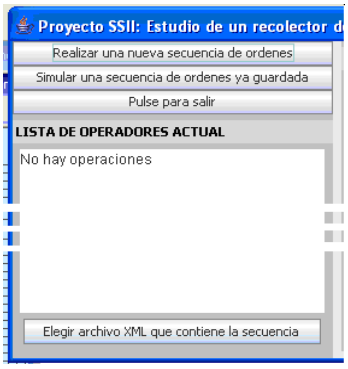
Vamos a estudiar una por una estas ventanas.

Ventana de secuencia

Para la realización de la ventana de secuencia hemos tenido que incluir (como elementos más significativos): 4 botones con sus respectivos eventos y manejadores de eventos, y un área de texto. En cada uno de los botones la función manejadora realiza lo siguiente:

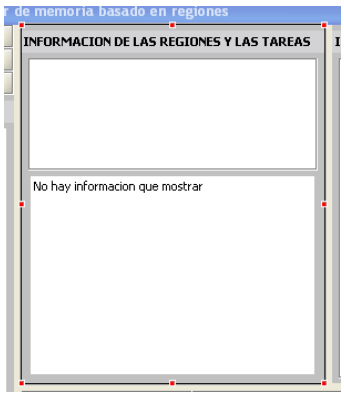
- *botonNuevaSecuencia*: muestra cada uno de los botones útiles para crear una nueva secuencia y actualiza todas las variables inicializándolas.
- *botonCargarSecuencia*: muestra cada uno de los botones necesarios para cargar una nueva secuencia y actualiza las variables inicializándolas.
- *botonSalir*: Sale del sistema con la llamada a *System.exit(0);*
- *botonEleccionSecuencia*: Crea un menú emergente el cual nos permite elegir un archivo XML que contenga una secuencia previamente guardada. Actualiza la ventana de forma que una vez cargado se vean reflejados todos los cambios.

En cuanto al área de texto, llamada *textoOperaciones*, es donde voy a mostrar cada una de las operaciones tanto cargadas como generadas y colocaré un puntero cono forma de flecha indicando la instrucción actual.



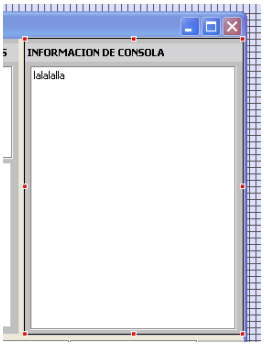
Ventana de datos

En esta ventana, en la que muestro la información relativa a las variables del modelo en curso, tan solo tengo una lista y un área de texto. La lista con nombre *listaVariables* muestra cada una de las regiones y tareas con su nombre, y según seleccionemos cada nombre, nos aparecerá su información el área de texto inferior llamado *textoVariables*. Esta área de texto mostrará diferente información según sea una región o una tarea y también según el modelo en que nos encontremos, ya que cada modelo nos habla de una concepción diferente de región o de tarea. La información que se muestra en *textoVariables* se recibe mediante el string proporcionado por la función *daInfo()* de cada clase referente a región y cada clase referente a tareas.



Ventana de consola

La ventana de consola es la más sencilla de todas. Sólo incluye un campo de texto llamado *textoDatosConsola* y en él se incluyen todas las informaciones que hemos considerado relevantes al hacer operaciones, es decir: mensajes aclaratorios acerca de la operación realizada, avisos de que algo no está funcionando correctamente…



Ventana de comandos

En esta ventana, proporcionamos botones para realizar cualquier operación que necesitamos para manejar modelos con regiones y tareas. Realmente, cada botón corresponde a aplicar sin más, un método de las tareas o de las regiones. Aunque en el manual de uso se puede observar más gráficamente como usar este interfaz, aquí se incluye un resumen de la tarea de cada botón:

- Ejecución secuencial (*botonEjecutar*): ejecuta sin parar una por una cada instrucción cargada. Después de cada paso, te informa de que se ha dado el paso.
- Ejecución "step by step" (*botonStep*): Realiza de una en una las operaciones cargadas de un archivo. En cada paso, podemos observar las variaciones producidas en las tareas y en las regiones. Es el más cómodo de utilizar.
- Crea Modelo (*botonCreaModelo*): Según el modelo que queramos crear, inicializa una variable de la clase llamada *variableEnUso*. Pone además el número del modelo mediante la variable *nModelo*.
- Crea Tarea (*botonCreaTarea*): Según el modelo en uso actual, crea una tarea del mismo tipo, con un nombre introducido por el usuario. Hace un case con *nModelo* para decidir que tipo de tarea se crea: TareaRTSJ, TareaSinPila o TareaSinPadre.
- Crea Región (*botonCreaRegion*): Funciona de forma similar a Crea Tarea sólo que para regiones.
- Enter (*botonEnter*): Realiza el método *enter()* de la tarea introducida por el usuario en la región también introducida. Estas entradas al sistema se proporcionan mediante cómodos JOptionPane.
- Exit (*botonExit*): Realiza el método exit() de la tarea introducida con la region introducida.
- Execute In Area (*botonExecuteInArea*): Realiza el método executeInArea() de la tarea introducida en la región introducida. Estos tres últimos tratadores de eventos son similares y llaman a métodos que son básicos para el funcionamiento del sistema como se puede observar. Son principalmente en los que nos centraremos para hacer pruebas.
- Recolectar Región (*botonRecolectarRegion*): Llama al método *recolectarRegion()* del modelo que actualmente se esté usando.

- Único Padre (*botonUnicoPadre*): Llama al método *unicoPadre()* del modelo que se esta usando. Para estos dos últimos métodos, como en los anteriores, la información necesaria: una región y una tarea, las pide por pantalla.
- ¿Es referencia prohibida? (*botonRefProhibida*): Realiza el método *referenciaProhibida()* del modelo en uso.
- Guardar secuencia (*botonGuardarSec*): Es el botón que llama a la función que guarda la secuencia actual que hemos generado. Se explica más a fondo en la parte del desarrollo del tratamiento de XML.

Este ha sido sobretodo el trabajo realizado en cuanto a implementación de la parte gráfica. Los detalles en cuanto a como se hicieron los diseños de la pantalla no los hemos incluido y tampoco hemos explicado nada referente a la vistosidad, por entender que esto se aleja bastante del tema del proyecto.

Cómo últimas incorporaciones a la implementación y tras realizar bastantes pruebas, decidimos incluir dos nuevos botones:

- Añadir texto a Consola (*botonAnadirTexto*): Para permitir hacer anotaciones que se mantengan en futuras cargas de cada prueba, incluimos este botón. Simplemente añade un texto a la consola de comandos.
- Continuar secuencia (*botonContinuar*): Una vez cargada una secuencia, puede que queramos continuarla de otra forma y para ello establecemos este vínculo entre ambas partes de la interfaz gráfica. Una vez que se han ejecutado todas las operaciones de una secuencia cargada, nos aparecerá un botón en el que pinchándolo se nos permite continuar esta secuencia como queramos.

Implementación de entrada/salida con XML

1- Motivación

Para completar la interfaz gráfica, pensamos que era necesario de alguna manera almacenar todos los ejemplos que nos pudieran parecer interesantes, ya que al tratarse principalmente de un proyecto de investigación, podría ser bueno recuperarlos sin necesidad de volver a hacerlos y recordar como se hizo. Es por ellos que vimos necesario incluir algún tipo de persistencia. La persistencia que nos pareció más sencilla y a la vez más clara por sus características, fue realizar archivos XML. Es por ello, que usamos alguno de los conocimientos adquiridos en otras asignaturas o cursos, para relacionar el trabajo realizado hasta ese momento con una generación y carga de archivos XML.

2- Desarrollo

Implementación de la lectura de ficheros XML

La lectura de ficheros XML se realiza mediante una clase llamada *ManejadorDeDocumentos.java*. Para cada palabra encerrada entre <> coge su

contenido y lo incluye en una lista de operaciones. Lee cada operación de este archivo y cuando termina devolverá el contenido de esta lista de operaciones a la clase principal del interfaz.

Sigue el siguiente algoritmo por tanto el sistema de lectura de XML:

Abre el fichero XML
Analiza la marca de comienzo de documento
Mientras no encuentre la marca de fin de fichero
Para cada palabra reservada (palabra entre <>) realiza operaciones diferentes según la palabra encontrada:
- secuenciaOperaciones: es la marca de comienzo de datos e inicializa las variables necesarias.
- modelo: Asigna el modelo concreto para este XML
- operación: Guarda la anterior operación si no es la primera e inicializa una estructura Operación para guardar la siguiente.
- acción: Asigna el valor de esta etiqueta como nombre a Operación.
- variable: Asigna el valor de esta etiqueta como valor de una de las variables de Operación.
fMientras
Guarda el último registro
Devuelve el control a la clase principal.

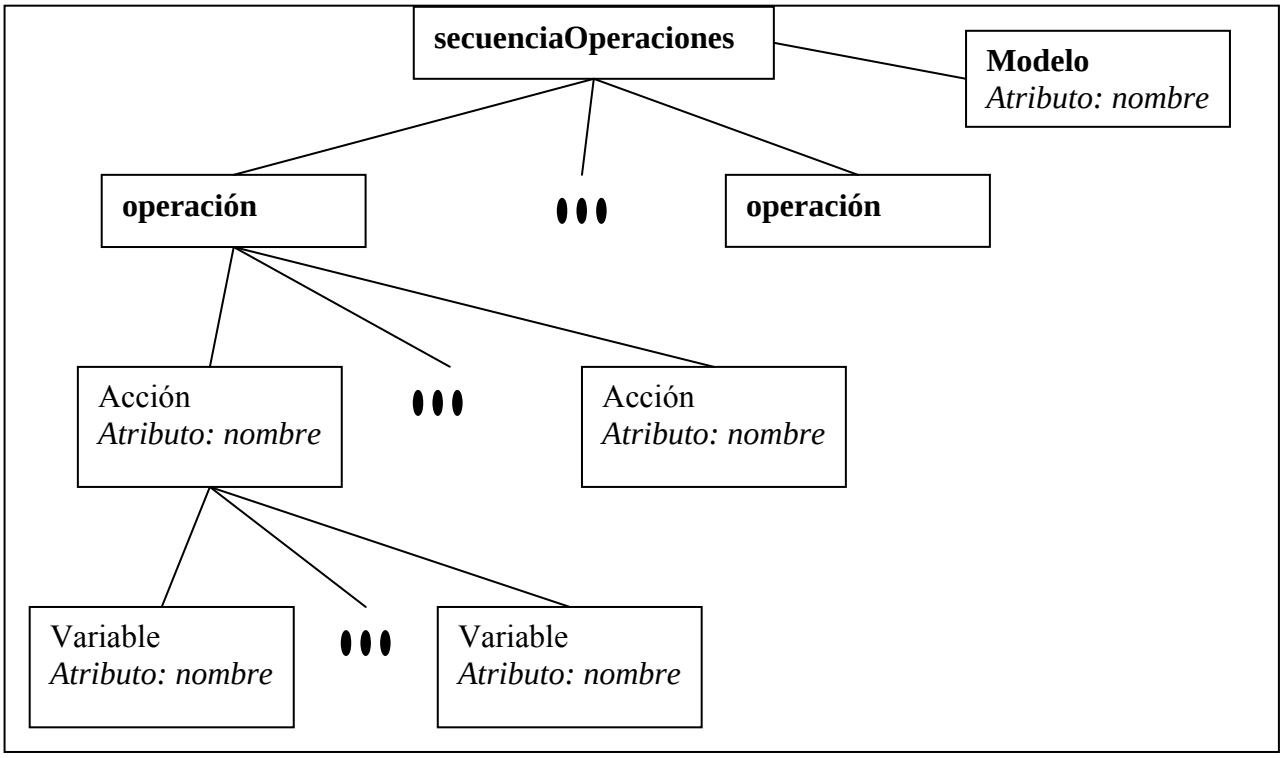
¿Qué es una lista de operaciones?
Hemos definido una clase llamada Operación.java que contienen un nombre de la operación (los nombres de operaciones posibles están incluidos en el manual de uso) y una lista de nombres de variables para dicha operación. De esta forma, es posibles fácilmente leer las distintas operaciones que se pueden dar para un modelo concreto.

Implementación de la escritura de ficheros XML

Para guardar un documento hago algunas operaciones algo más complicadas. La idea es crear un objeto de tipo Document en que cada nodo va a ser una de las etiquetas y cada atributo de cada nodo será la información que lleva asociado el nodo.

Este árbol que tenemos que crear en esta estructura Document es el siguiente:

Una vez definido el árbol ya simplemente llamamos a una función que convierte este objeto de tipo Document en un fichero XML y lo guardamos.



Estructura de paquetes definida

Tras todo lo visto en esta sección podemos ver ahora como queda la estructura de paquetes y clases del proyecto. Hemos decidido poner cada clase de un tipo en un paquete y hacer herencia que nos iba a facilitar en muchas operaciones de la parte gráfica.

La herencia realizada es muy simple. La clase Modelo es una clase padre de ModeloRTSJ, ModeloSinPila y ModeloSinReglaPadre. Esta clase tan sólo contiene un atributo llamado *tipo* que dirá el tipo del modelo y una estructura de la clase Texto llamada *infoConsola*. Esta estructura de datos Texto servirá para modificar el texto presente en la consola de comandos.

Las clases RegionMemoria y Tarea son abstractas y contienen simplemente el nombre y la variable para modificar la información de consola. Estas clases son abstractas ya que necesitan que se defina el método *daInfo()* que es propio de cada región o de cada tarea, y que sirve para mostrar la información propia del elemento en cuestión.

Por tanto, podemos hablar de tres paquetes principales que son los paquetes “tareas”, “regiones” y “modelos”.

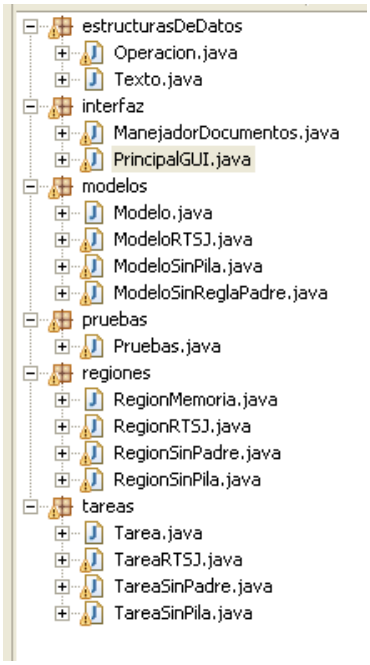
El paquete “pruebas” contiene las pruebas realizadas cuando no existía la interfaz gráfica que nos permite realizar pruebas automáticamente leyendo de un

archivo XML. Se puede observar al ver la clase Pruebas.java lo costoso que sería realizar una clase por cada ejemplo planteado.

En el paquete "interfaz" encontraremos la clase que ayuda a capturar la información de los XML y la clase principal de la interfaz gráfica. Es desde esta última clase desde donde lanzamos el sistema.

Por último, en el paquete "estructurasDeDatos" encontramos las 2 estructuras de datos que nos han sido necesarias para la realización del proyecto:

- Operación: Es la estructura que contiene un nombre de una operación y sus variables, útil para cargar y guardar el fichero XML.
- Texto: Una estructura en la que modificamos el valor de la información presentada por pantalla en la consola de comandos de nuestra interfaz.



Capítulo 7
Pruebas realizadas

Prueba 1

Descripción:

Vamos a demostrar con esta prueba que RTSJ tiene el fallo de generar condiciones de carrera. Como hemos visto en la teoría propuesta, existen hasta 4 posibles formas de que nos de una misma excepción, con lo que evidentemente además de generarse condiciones de carrera, es muy difícil depurar que es lo que realmente ha hecho el sistema. En nuestra implementación hemos tenido presente estos errores y podemos provocar que nuestro modelo muestre este fallo del modelo RTSJ.

Para ello, vamos a generar en archivos XML cada una de las 4 formas posibles de generar la excepción y también las 4 formas posibles de acceder correctamente sin que salte la excepción ScopedCycledException. Para ello nos basamos en el siguiente ejemplo ya propuesto en su momento al hablar de los problemas del modelo RTSJ:

Se pueden crear condiciones de carrera, como en la siguiente situación:
Se crean dos regiones de memoria scoped A y B.
Se crean dos hilos de tiempo real H1 y H2.
H1 entra en las regiones en el orden A, B.
H2 entra en las regiones en el orden B, A.

Cuando se ejecute el programa, se puede provocar distintos comportamientos, dependiendo de las condiciones de carrera:
• Si H1 entra en A y B antes de que H2 entre en B, H2 viola la regla del padre único ya que B se encontraría con dos padres: A y el padre que le correspondería al ser entrada por H2.
• Si ocurre al revés, que H2 entre en B y A antes de que H1 entre A, pasaría lo mismo.
Si H1 y H2 entran respectivamente en A y B y permanecen un tiempo en ellas, al cabo de ese tiempo puede ocurrir que:
• Cuando H1 intente entrar en B violará la regla del padre único.
• Cuando H2 intente entrar en A violará la regla del padre único.
Estas cuatro situaciones provocarían que se lanzase la excepción ScopedCycleException(), y el programador se encontraría con 4 posibles errores en tiempo de ejecución lo que haría muy complicado repasar el código para corregirlo. Además con el mismo código, hay otras cuatro situaciones en las que el programa sí funcionaría correctamente:
• H1 entra en A y B, sale de ambas y luego H2 entra en B y A.
• H2 entra en B y A, sale de ambas y luego H1 entra en A y B.
• H1 entra en A y B, y sale de B antes de que H2 entre en ella, y luego H1 sale de A, antes de que H2 intente entrar en ella.
• H2 entra en B y A, y sale de A antes de que H1 entre en ella, y luego H2 sale de B, antes de que H1 intente entrar en ella.

Desarrollo:
Los archivos que hemos generado son los siguientes:
PruebaRTSJ1.xml
Contiene el siguiente caso:
Tarea 1 entra en la región A
Tarea 1 entra en la región B
Tarea 2 entra en la región B
Salta una excepción de tipo ScopedCycledException

PruebaRTSJ2.xml
Contiene el siguiente caso:
Tarea 2 entra en la región B
Tarea 2 entra en la región A
Tarea 1 entra en la región A
Salta una excepción de tipo ScopedCycledException
Es un caso similar al anterior en realidad.

PruebaRTSJ3.xml
Contiene el siguiente caso:
Tarea 1 entra en la región A
Tarea 2 entra en la región B
Tarea 1 entra en la región B
Salta una excepción de tipo ScopedCycledException

PruebaRTSJ4.xml
Contiene el siguiente caso:
Tarea 1 entra en la región A
Tarea 2 entra en la región B
Tarea 2 entra en la región A
Salta una excepción de tipo ScopedCycledException
Es un caso paralelo al anterior.
Hasta aquí los casos que el modelo RTSJ nos "limita" a la hora de programar y los 4 siguientes sí son aquellos casos permitidos.

PruebaRTSJ5.xml
Contiene el siguiente caso:
Tarea 1 entra en A
Tarea 1 entra en B
Tarea 1 sale de B
Tarea 1 sale de A
Tarea 2 entra en B
Tarea 2 entra en A

PruebaRTSJ6.xml
Contiene el siguiente caso:
Tarea 2 entra en B
Tarea 2 entra en A
Tarea 2 sale de B
Tarea 2 sale de A
Tarea 1 entra en A
Tarea 1 entra en B

PruebaRTSJ7.xml
Contiene el siguiente caso:
Tarea 1 entra en A
Tarea 1 entra en B
Tarea 1 sale de B
Tarea 2 entra en B
Tarea 1 sale de A
Tarea 2 entra en A

PruebaRTSJ8.xml
Contiene el siguiente caso:
Tarea 2 entra en B
Tarea 2 entra en A
Tarea 2 sale de A
Tarea 1 entra en A
Tarea 2 sale de B
Tarea 1 entra en B

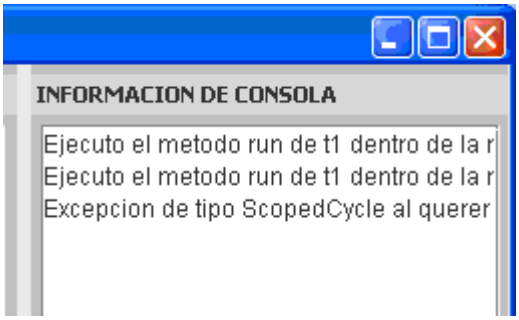
En cada uno de estos archivos se generaría el código XML correspondiente. Mostramos uno de estos códigos, concretamente el de PruebaRTSJ1.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<secuenciaOperaciones>
	<modelo name="rtsj"/>
	<operacion>
		<accion name="creaModelo"/>
		<variable name="rtsj"/>
	</operacion>
	<operacion>
		<accion name="creaTarea"/>
		<variable name="t1"/>
	</operacion>
	<operacion>
		<accion name="creaTarea"/>
		<variable name="t2"/>
	</operacion>
	<operacion>
		<accion name="creaRegion"/>
		<variable name="a"/>
	</operacion>
	<operacion>
		<accion name="creaRegion"/>
		<variable name="b"/></operacion>
	<operacion>
		<accion name="enter"/>
		<variable name="t1"/><variable name="a"/>
	</operacion>
	<operacion>
		<accion name="enter"/>
		<variable name="t1"/>
		<variable name="b"/>
	</operacion>
	<operacion>
		<accion name="enter"/>
		<variable name="t2"/>
		<variable name="b"/>
	</operacion>
</secuenciaOperaciones>
```

En los 4 primeros ejemplos (PruebaRTSJ del 1 al 4) los resultados han sido los siguientes:

En todos ellos hemos creado un modelo, tras ello hemos creado las regiones y las tareas y tras eso hemos realizado los métodos *enter()* específicos de cada ejemplo. Al llegar al punto indicado por el ejemplo propuesto por la teoría, en la consola de comandos nos muestra un mensaje informándonos de la excepción.

En el gráfico adjunto podemos ver como nos informa de esta excepción en nuestra simulación. Como advertimos, este error va a ser el mismo para cada posible forma de llegar al error. Esto es un inconveniente muy grande para cualquier persona que quiera depurar el código



En los 4 siguientes ejemplos, se puede realizar sin problema las operaciones en el orden indicado.

Conclusiones:

- El modelo RTSJ se comporta mal para 4 casos de los 8 posibles. Esto limita mucho al programador ya que muchas veces en programas concurrentes nos encontramos con situaciones parecidas y en este caso además de producir excepciones, el modelo RTSJ no proporciona una respuesta específica, sino genérica para todas, con lo que imposibilita la depuración. Es necesario que se busquen alternativas para un modelo más adecuado al punto de vista del programador que actúa como usuario.
- Por último, observamos que la implementación nos proporciona cada uno de las excepciones esperadas y funciona bien en los casos esperados. Por tanto, parece que el modelo actúa correctamente.

Prueba 2

Descripción:

En esta prueba vamos a demostrar que en el modelo RTSJ que hemos implementado se cumplen lo propuesto en la especificación del modelo en lo relativo a asignaciones ilegales.

La idea es la siguiente:

Vamos a provocar asignaciones ilegales. Para ello simplemente vamos a crear tres regiones y vamos a utilizar una tarea que en un orden establecido vaya entrando en ella. Observaremos como se comporta el sistema de detección de referencias ilegales. Tras ello realizamos un ExecuteInArea y tendremos un resultado similar.

Desarrollo:

Listado de operaciones:
creaModelo(rtsj)
creaTarea(t1)
creaRegion(a)

creaRegion(b)
creaRegion(c)
enter(t1 a)
enter(t1 b)
refProhibida(b a t1) ***1ª referencia***
refProhibida(a b t1) ***2ª referencia***
enter(t1 c)
refProhibida(c a t1) ***3ª referencia***
refProhibida(a c t1) ***4ª referencia***
executeInArea(t1 a)
refProhibida(a c t1) ***5ª referencia***
refProhibida(c a t1) ***6ª referencia***

Esta es la información mostrada por consola:

Ejecuto el metodo run de t1 dentro de la región a
Ejecuto el metodo run de t1 dentro de la región b
No
Si
Ejecuto el metodo run de t1 dentro de la región c
No
Si
Cambio el contexto actual a la region a
Si
Si

¿Qué significa esta información?
1ª Referencia→ No se trata de una referencia prohibida
2ª Referencia→ Sí se trata de una referencia prohibida
3ª Referencia→ No se trata de una referencia prohibida
4ª Referencia→ Sí se trata de una referencia prohibida
5ª Referencia→ Sí se trata de una referencia prohibida
6ª Referencia→ Sí se trata de una referencia prohibida
Estos resultados se explican en las conclusiones de la prueba.

La prueba se incluye en el archivo pruebaRTSJ9.xml

Conclusiones:
Para poder hacer una referencia desde una región origen o a una región destino se tienen que dar dos condiciones necesarias:
1º. Que la región origen sea la región actual de la Tarea.
El hilo tiene que tener a la región origen como región actual. Tiene que haber entrado en ella. Si esta condición no se da nos da igual la condición segunda.
Así la instrucción executeInArea nos puede ayudar para convertir regiones que están activas en regiones actuales.

2º Una vez que la región origen el la actual, sólo se permite crear referencias hacia regiones con un tiempo de vida menor. Así se consigue que no haya "punteros

colgantes" por la posibilidad de que la región destino sea recolectada antes que la región origen. Y esto se traduce en que solo se permiten referencias de una región hacia abajo de la pila, que son las regiones que seguro que tienen un tiempo de vida mayor de la que tiene ella.

Prueba 3

Descripción:

En esta prueba no tenemos por objetivo encontrar ningún fallo del modelo RTSJ sino más bien demostrar la capacidad del modelo para trabajar con varias tareas y varias regiones a la vez sin que se produzcan por ello errores, eso sí, andando con cuidado a la hora de realizar asignaciones ilegales y de crear condiciones de carrera. En dicha prueba, creamos un total de 8 regiones y dos tareas y vamos a realizar todas las operaciones propuestas para el modelo además de incluir información acerca de los datos obtenidos.

Desarrollo:

La secuencia de operaciones propuesta para esta prueba es la siguiente:
El modelo actual de trabajo es: rtsj
Listado de operaciones:
creaModelo(rtsj)
creaRegion(A)
creaRegion(B)
creaRegion(C)
creaRegion(D)
creaRegion(E)
creaRegion(Q)
creaRegion(X)
creaRegion(Heap)
creaTarea(T1)
creaTarea(T4)
enter(T1 A)
enter(T1 B)
enter(T1 Heap)
anadeTexto(?Puedo hacer una referencia de Heap1 a A en t1?)
refProhibida(Heap A T1)
enter(T1 C)
enter(T4 A)
enter(T4 C)
enter(T4 D)
anadeTexto(?Puedo hacer una referencia de C a A en t1?)
refProhibida(C A T1)
executeInArea(T1 A)
enter(T1 E)
enter(T1 Q)
anadeTexto(¿Puedo hacer una referencia de E a Q?)
ref(E Q T1));
anadeTexto(?Puedo hacer una referencia de Q a E?)

refProhibida(Q E T1)
anadeTexto(?Puedo hacer una referencia de Q a A?)
refProhibida(Q A T1)
anadeTexto(?Podria recolectar la region Q?(por las referencias))
recolectarRegion(Q)
exit(T1 Q)
anadeTexto(?Podria recolectar la region E?(por las referencias))
recolectarRegion(E)
exit(T1 E)
anadeTexto(Observa la region actual de T1)
executeInArea(T1 C)
anadeTexto(?Podria recolectar la region C?(por las referencias))
recolectarRegion(C)
exit(T1 C)
anadeTexto(Observa la region actual de T1)

La prueba se incluye en el archivo pruebaRTSJ10.xml

Conclusiones:
- El modelo RTSJ implementado se comporta adecuadamente: muestra las asignaciones ilegales de forma correcta, es capaz de trabajar con variadas regiones de memoria y tareas, es capaz de entrar en regiones, salir de ellas, realizar ExecuteInArea…

Prueba 4

Descripción:
La ventaja del modelo sin pila sobre el modelo RTSJ va a ser reflejada en esta prueba. El modelo sin pila resuelve el problema de condiciones de carrera que antes se provocaban en RTSJ. En esta prueba vamos a ver como dichas excepciones no aparecen.

Desarrollo:
No hemos realizado cada uno de los casos posibles sino tan sólo uno de ellos y vamos a observar que ya no se produce una ScopedCycledExcepction:

creaModelo(sinpila)
creaRegion(a)
creaRegion(b)
creaTarea(t1)
creaTarea(t2)
enter(t1 a)
enter(t1 b)
enter(t2 b)
enter(t2 a)

Estas operaciones se encuentran en el fichero pruebaSinPila1.xml

Conclusiones:

- El problema de la falta de expresividad queda de manifiesto con este ejemplo. Operaciones que podían ser realizadas anteriormente en el modelo RTSJ ahora es imposible realizarlas.
- Además este problema para expresar regiones hijas y hermanas se da muy frecuentemente a la hora de programar, en la que el programador no debe estar pendiente de que regiones está utilizando y si es posible o no ciertas operaciones.

Prueba 5

Descripción:
La principal pega que encontramos en el modelo sin pila es que no permite expresar operaciones que antes, en el modelo RTSJ sí era posible realizar. El ejemplo más claro es el de dos regiones creadas en el ámbito de una misma región A, que llamaríamos por sus nombres de display AB y AC. Al intentar realizar una referencia de una a otra, observamos que no es posible.
Vamos a ver como sería posible realizar esto.

Desarrollo:
Esta prueba va a seguir los siguientes pasos:

creaModelo(sinpila)
creaRegion(a)
creaRegion(b a)
creaRegion(c a)
creaTarea(t1)
enter(t1 a)
enter(t1 b)
enter(t1 c)
refProhibida(c b t1)

Lo que vemos es que al realizar la operación de referencia prohibida nos dice que está prohibida realizar dicha referencia, sin embargo esto no ocurría con el modelo RTSJ. Hemos realizado dos archivos uno para modelo sin pila y modelo rtsj y observamos las diferencias explicadas.
Esto demuestra claramente la falta de expresividad, y que este modelo también se aleja en este caso de la visión de un programador, que debería estar pendiente de las referencias realizadas.
Esta prueba la podemos encontrar en el archivo pruebaSinPIla2.xml y la prueba para comparar con el comportamiento en RTSj la encontramos en el fichero pruebaSinPila2vsRTSJ.xml.

Conclusiones:

- El problema de la falta de expresividad queda de manifiesto con este ejemplo. Operaciones que podían ser realizadas anteriormente en el modelo RTSJ ahora es imposible realizarlas.
- Además este problema para expresar regiones hijas y hermanas se da muy frecuentemente a la hora de programar, en la que el programador no debe estar pendiente de que regiones está utilizando y si es posible o no ciertas operaciones.

Prueba 6

Descripción:
Esta prueba es simplemente para ver como se comporta el modelo sin pila en caso de tener varias regiones y tareas. Es decir, probamos que realmente el modelo no produce excepciones y que es correcto. Se realizan varias operaciones y en base a ello damos por probado que el modelo funciona adecuadamente.

Desarrollo:
Estas son las operaciones que han sido realizadas:

creaModelo(sinpila)
creaRegion(c)
creaRegion(d c)
creaTarea(t2)
enter(t2 c)
enter(t2 d)
anadeTexto(Puedo hacer una referencia de D a C?)
refProhibida(d c t2)
anadeTexto(Puedo hacer una referencia de C a D?)
refProhibida(c d t2)
anadeTexto(Podria recolectar la region C?(Por las referencias))
recolectarRegion(c)
anadeTexto(Podria recolectar la region D?(Por las referencias))
recolectarRegion(d)
exit(t2 d)

Se puede encontrar esta información cargando el archivo PruebaSinPila3.xml

Conclusiones:

- Observamos que el modelo actúa de forma adecuada y entendemos que lo único de lo que podemos dudar es de su clara falta de expresividad.

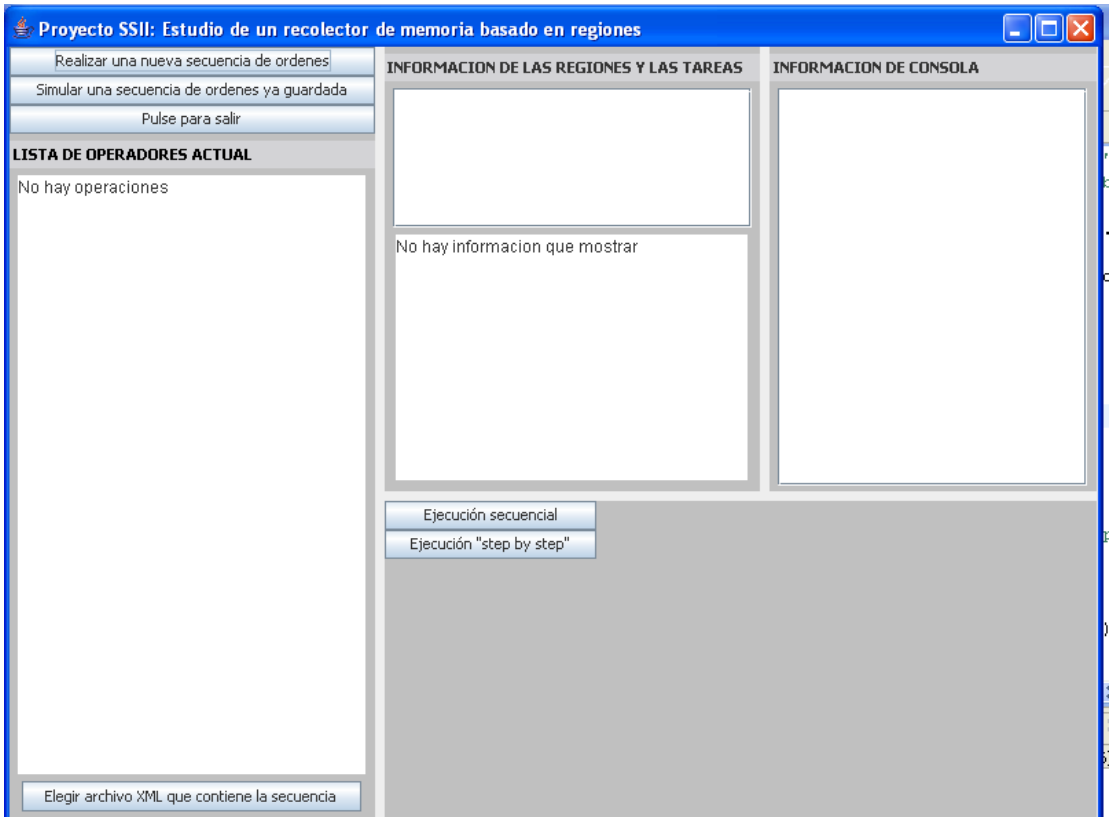
Anexos

A. Manual de uso

Documentación para la interfaz gráfica del proyecto

1- Inicio de la simulación

Para poner a funcionar el sistema tan solo tenemos que, una vez compiladas las clases del proyecto, ejecutar el método *main* de la clase PrincipalGUI.java. No hay que incluir ningún tipo de argumentos a esta llamada del método. Al realizar esta operación nos aparecerá la pantalla principal del sistema:



2- Funcionamiento del sistema.

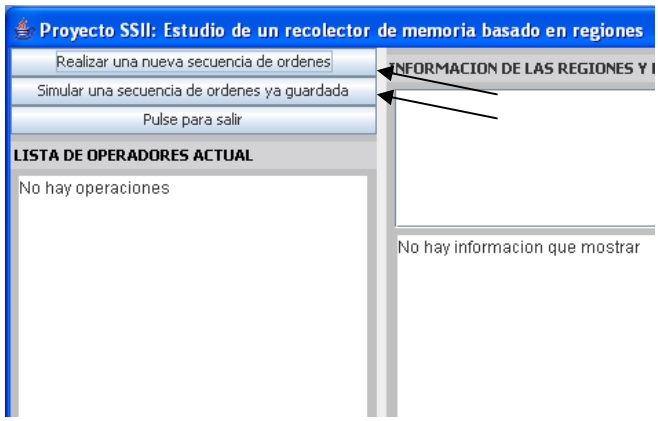
El interfaz gráfico nos permite diferentes opciones:

- Realizar una secuencia de ordenes manualmente, mediante el uso de unos botones que nos proporciona la interfaz. Con estas ordenes, una vez

acabado el proceso podemos guardar los resultados donde deseemos, par utilizarlos en otras ocasiones.

- Simular una secuencia de ordenes ya guardada, es decir, coger una secuencia que previamente hemos guardado.

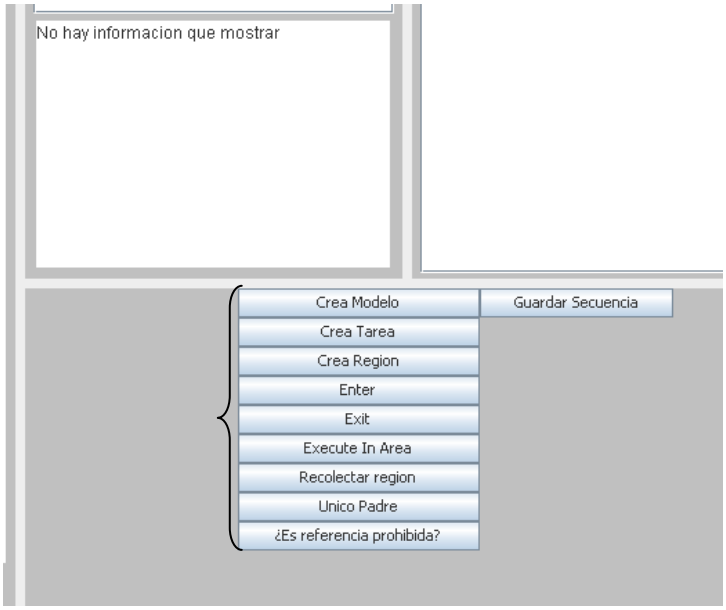
Esta es la primera decisión que tenemos que realizar y los botones para decidir entre una de ellas se encuentran en la parte superior izquierda.



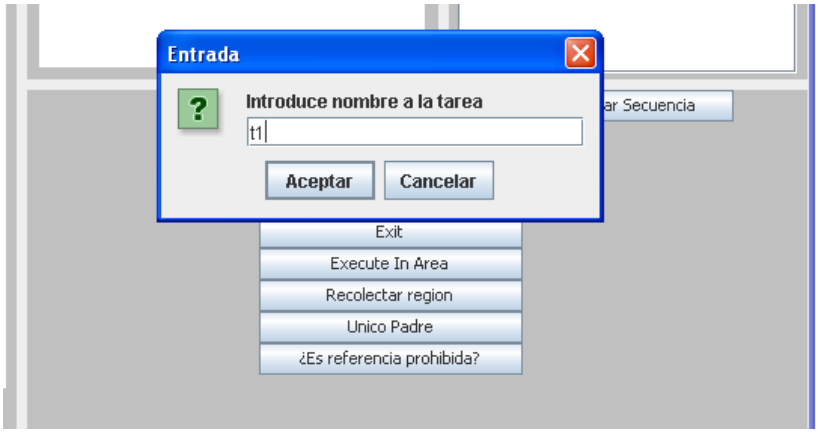
Cada vez que se pulsa un botón de estos dos, se reinicia el sistema. Al pulsar el botón con mensaje “Pulse para salir”, se cierra la interfaz y acaba la ejecución del programa.

Primera opción: Realizar una nueva secuencia de ordenes

Si nos decantamos por la primera opción, al pulsar el botón aparecerán una serie de botones para realizar operaciones sobre modelos, regiones y tareas.



El nombre que contiene el botón es bastante explicativo y al pulsar te aparecen ventanas emergentes que te solicitan la información necesaria para poder realizar la operación. Es bastante intuitivo. Veamos un ejemplo al solicitar crear una nueva tarea:



Alguna información importante:

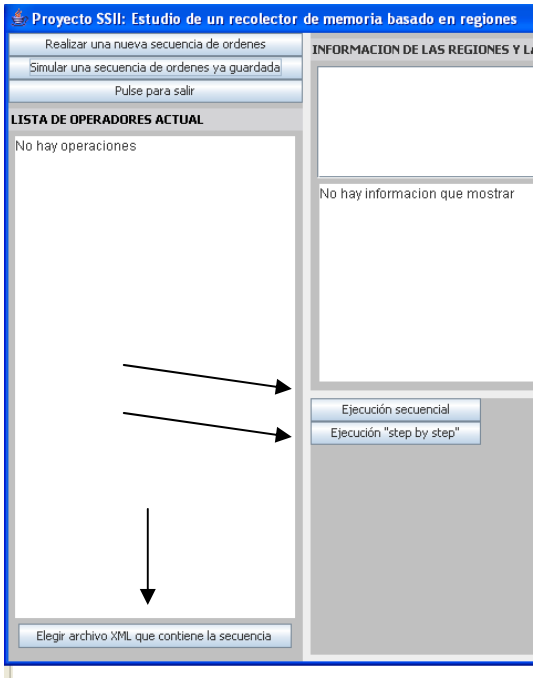
- Se entiende que antes de realizar cualquier operación sobre regiones y tareas, se debe elegir el modelo concreto que se va a usar.
- La mayor parte de los botones necesitan de tareas y regiones, así que si no existen tareas o regiones no podrá ser posible usar estos métodos. Toda la información relativa a los métodos se puede encontrar en el apartado de implementación.
- El botón guardarSecuencia te permite guardar en el lugar deseado la secuencia de operaciones, pero no se permite modificar una recientemente cargada.

Segunda opción: Simular una secuencia de ordenes ya guardada

Esta opción es para simular aquellas secuencias ya guardadas. De esta forma permite que generemos nuestros ejemplos más interesantes y los guardemos, para mostrarlos en cualquier momento. Esto se ha realizado mediante tratamiento de ficheros XML.

Al pulsar sobre esta segunda opción, obtenemos principalmente 3 botones distintos que no aparecerían en la otra opción. Son los botones de carga de secuencia, de ejecución secuencial y de ejecución paso a paso.

- El botón de carga de secuencia nos permite escoger el archivo a cargar.
- El botón de ejecución secuencial, ejecuta cada una de las ordenes de forma consecutiva de forma que solo podemos ver el resultado final.
- El botón de ejecución paso a paso, nos permite ejecutar paso a paso las órdenes, una por una, y de esta forma ver el contenido de cada región o de cada tarea en cada paso.



3- Librerías necesarias

Todas las librerías necesarias para el proyecto se encuentran en la distribución de la maquina virtual de Java 1.5

4- Acoplamiento con la simulación

Anteriormente, se usaban la consola de comandos para incluir información en el sistema. Para sustituir dicha salida se cambia todo aquello que deseemos sacar por pantalla (es decir, todo *System.out.println()* de la simulación), por la llamada al método "*infoConsola.ponInfoConsola(String s)*" de una variable global que contiene la información contenida en la consola. Por tanto, todo aquello que queramos mostrar, se muestra de esa forma.

5- Almacenaje en XML

La gestión de ficheros XML se hace de forma transparente para el usuario. El usuario no participa directamente en la creación de XML, o al menos eso cree, ya que en cada información introducida al sistema se proporciona lo necesario. No utilizamos más que las librerías que la distribución de Java nos proporciona para la creación y gestión de archivos XML y todo el tratamiento para su lectura se puede encontrar en la clase ManejadorDocumentos.java.

6- Comandos permitidos en el XML

Para guardar secuencias y de esta forma automatizar las pruebas, usamos archivos XML con una estructura.
La estructura del XML es la siguiente:

```
<secuenciaOperaciones>
    <modelo name="rtsj"/>
    <operacion>
        <accion name="metodo1"/>
        <variable name="variable1"/>
        ...(más variables)
    </operacion>
    ...(más operaciones)
</secuenciaOperaciones>
```

- Los campos posibles para el nombre en la etiqueta modelo son **rtsj**, **sinPila**, **sinPadre**.
- Los campos posibles para el nombre en las etiquetas accion son:
    - **creaModelo** → Usa tan solo una variable que será el *nombre*.
    - **creaRegion** → Usa tan solo una variable que será el *nombre*.
    - **creaTarea** → Usa tan solo una variable que será el *nombre*.
    - **referenciaProhibida** → Usa tres variables. La primera es el *nombre* de la **región** que queremos considerar como **origen**. La segunda es el *nombre* de la **región** que queremos considerar como **destino**. La tercera es el *nombre* de la **tarea** en donde vamos a hacer la comprobación.
    - **unicoPadre** → Usa dos variables. La primera es el *nombre* de la **región** que queremos ver si es el único padre. La segunda es el *nombre* de la **tarea**.
    - **recolectarRegion** → Usa dos variables. La primera es el *nombre* de la **región** a recolectar. La segunda es el *nombre* de la tarea en donde se recolecta.
    - **enter** → Usa tan solo una variable que es el *nombre* de la región.
    - **exit** → Usa tan solo una variable que es el *nombre* de la región.
    - **executeInArea** → Usa tan solo una variable que es el *nombre* de la región.
    - **incluyeTexto** → Tiene una variable que es un un string que contiene el *texto* que queramos incluir en la consola. Podemos incluir cualquier información adicional en la consola, de forma que quede totalmente explicado el funcionamiento de nuestro sistema.

Un ejemplo de archivo en XML que representa una secuencia es:

```
<secuenciaOperaciones>
    <modelo name="rtsj"/>
    <operacion>
        <accion name="creaRegion"/>
        <variable name="region1"/>
    </operacion>
    <operacion>
        <accion name="creaRegion"/>
        <variable name="region2"/>
```

```
        </operacion>
        <operacion>
                <accion name="creaTarea"/>
                <variable name="tarea1"/>
        </operacion>
</secuenciaOperaciones>
```

B. Glosario de términos

- **Constructor**

Es un método especial que tiene cada clase de Java, y se invoca cuando se crea un objeto de dicha clase. En este método están incluidas las operaciones necesarias para iniciar los atributos de la clase o cualquier otra operación que se tenga que realizar antes de utilizar el objeto por primera vez.

- **Finalizador**

El finalizador de una clase se ejecuta cuando el colector de basura ha determinado que un objeto de dicha clase ya no se está utilizando, antes de eliminar el objeto de la memoria. Se invoca el método definido en la clase Object, `finalize()`, que se puede sobrescribir en cualquier otra clase para incorporar el código necesario para la liberación de los recursos utilizados o para realizar algún tipo de procesado adicional en el momento en el que deje de utilizarse un objeto.

- **HeapMemory**

Literalmente heap significa montículo, pero hemos decidido no traducirlo cuando nos referimos a la memoria Heap. Es un objeto que permite, dentro de otras áreas de memoria, asignar objetos en el heap de Java.

- **IllegalAccessException**

Se lanza esta excepción cuando una aplicación intenta crear una instancia, fijar o pedir el valor de un campo o invocar un método, pero el método que se está ejecutando actualmente no tiene acceso a una definición específica de la clase, campo, método o constructor.

- **InaccessibleAreaException**

Se lanza esta excepción cuando el área de memoria especificada no es accesible por el actual contexto de asignación de la pila scope del thread actual.

- **InmortalMemory**

Es un recurso que comparten todos los threads. Los objetos asignados en este área viven hasta el final de la aplicación, no están sujetos al recolector de basura, aunque algunos algoritmos de GC requieren escanear la memoria inmortal. Un objeto de aquí solo puede referenciar a otro objeto del heap o de inmortal. Al contrario que los objetos del heap, los de inmortal siguen viviendo aunque no existan otras referencias hacia ellos.

- **InmortalPhysicalMemory**

Una instancia de esta clase permite que los objetos sean asignados en un rango de memoria física con atributos particulares determinados por su tipo de memoria. Este área de memoria tiene las mismas restricciones en cuanto a reglas de asignación que las áreas de `InmortalMemory` y puede ser usada en cualquier contexto en que se pueda usar `InmortalMemory`. Los objetos asignados en este área de memoria tienen un tiempo de vida mayor que la aplicación y tanto como los objetos asignados el `InmortalMemory`.

- **InstantiationException**

Se lanza esta excepción cuando una aplicación intenta crear una instancia o una clase usando el método `newInstance()`, pero el objeto de la clase especificada no ha podido ser instanciado porque es una interfaz o una clase abstracta.

- **LTMemory**

Hereda de la clase `ScopedMemory`. Representa un área de memoria asignada por la clase `RealtimeThread` o un grupo de threads de tiempo real, garantizado por el sistema que tiene una asignación en tiempo lineal. El área de memoria descrita por `LTMemory` no existe en el heap y no esta sujeta al recolector de basura. Por eso, es seguro usar un objeto de `LTMemory` como área de memoria asociada con un `NoHeapRealtimeThread` o entrar el área de memoria usando el método `ScopedMemory.enter()` dentro de un `NoHeapRealtimeThread`.

- **Máquina virtual Java**

La máquina virtual de Java (JVM o Java Virtual Machine), desarrollada por *Sun Microsystems*, es una máquina virtual que ejecuta el código resultante de

la compilación de un programa escrito mediante el lenguaje de programación Java, conocido también como bytecode de Java. Aunque compiladores de otros lenguajes pueden dar lugar a bytecode que se puede ejecutar sobre la JVM.

Un programa escrito en Java, debe pasar por un proceso de compilación. Si este término es generalmente usado para el mecanismo encargado de convertir código fuente a código nativo de la plataforma sobre la que pretende ejecutarse, en Java, la compilación produce un código intermedio (el bytecode) destinado a ser ejecutado por la máquina virtual de Java. Este bytecode toma la forma de un fichero con extensión ".class". Todo programa en Java está compuesto por una o varias clases, y cada clase estará en un archivo diferente, tanto antes como después de la compilación.

Para facilitar la distribución de grandes aplicaciones, las clases que lo forman pueden ser empaquetadas todas juntas, tras su compilación, en un único archivo (con extensión ".jar"). Este archivo puede entonces ser ejecutado por la JVM de dos maneras distintas. La forma original era mediante un proceso conocido como interpretación, por el cual se llevaba a cabo la emulación del conjunto de instrucciones de la JVM, y las instrucciones se ejecutan secuencialmente.

- **NoHeapRealTimeThread**

Un objeto de la clase `NoHeapRealTimeThread` es una especialización de la clase `RealTimeThread`. Debido a que un instancia de `NoHeapRealTimeThread` puede inmediatamente reemplazar cualquier ejecución del recolector de basura, la lógica contenida en su método `run()` no puede permitirse nunca asignar o referenciar cualquier objeto situado en el heap. A nivel de bytecode, es ilegal que aparezca una referencia a un objeto situado en el heap, en la pila de operaciones de un thread de la clase `NoHeapRealTimeThread` . Por esto, siempre es seguro para un thread de la clase `NoHeapRealTimeThread` interrumpir el recolector de basura en cualquier momento, sin tener que esperar a que termine el ciclo de recolección de basura.

Debido a estas restricciones, un objeto de la clase `NoHeapRealTimeThread`, debe estar situado en un área de memoria en la que la lógica del thread pueda tener acceso a las instancias de las variables, y de forma que los métodos de Java de la clase `Thread` (e.g., `enumerate()` y `join()`) se completen normalmente excepto donde la ejecución pueda causar violaciones de acceso. Los constructores de `NoHeapRealTimeThread` requieren una referencia a la clase `ScopedMemory` o a `InmortalMemory`. Cuando el thread es iniciado, cualquier ejecución se tiene que producir en el ámbito del área de memoria dada. Así, todas las asignaciones de memoria llevadas a cabo con el nuevo operador son tomadas en el área de memoria dada.

- **Predecible en tiempo**

Un sistema es predecible en tiempo cuando se puede garantizar su comportamiento en el tiempo requerido.

- **RealTimeThread**

La clase `RealTimeThread` hereda de la clase `Thread` y añade acceso a servicios de tiempo real como control de transferencia asíncrona, memoria fuera del heap, y servicios avanzados de planificación. Igual que la clase `Thread`, esta clase tiene dos formas de crear un thread de tiempo real utilizable:

- Crear una nueva clase que extienda `RealTimeThread` y sobrescriba el método `run()` con la lógica necesaria para el thread.
- Crear una instancia de `RealtimeThread` usando uno de los constructores con un parámetro lógico. Se pasa un objeto que implemente la interfaz `Runnable` cuyo método `run()` implementa el funcionamiento del thread.

- **Run**

Es el método que se ejecuta cuando un objeto `Runnable` es activado. El método `run()` es el "alma" de un thread. Es en este método donde se produce toda la actividad del thread. Este método es llamado después de que el thread empiece (llamando a su método `start()`).

- **Runnable**

Esta interfaz ha sido diseñada para proporcionar un protocolo común para los objetos de los que se quiera ejecutar código mientras estén activos. Por ejemplo, `Runnable` está implementada por la clase `Thread`. Estar activo quiere decir que un thread ha sido iniciado y todavía no ha sido parado.

- **Schedulable**

Es una interfaz que hereda de la interfaz `Runnable`. Está implementada por ejemplo por la clase `Thread`. Los manejadores y otros objetos pueden ser ejecutados por un `Scheduler` (planificador) si proporcionan un método `run()` y otros métodos propios de esta interfaz. El `Scheduler` usa esta información para crear un contexto conveniente para ejecutar el método `run()`.

- **ScopedCycleException**

Se lanza esta excepción cuando un objeto de la clase "`Schedulable`" intenta entrar, mediante el método `enter()`, una instancia de `ScopedMemory`, donde esta operación pudiera causar una violación de la regla del único padre.

- **ScopedMemory**

Literalmente scope significa ámbito, pero hemos decidido no traducirlo cuando nos referimos a la memoria Scoped. `ScopedMemory` es la clase abstracta con la que cuentan las representaciones de espacios de memoria con tiempo de vida limitado. Es valida el tiempo que los threads de tiempo real tengan acceso a ella. Se crea una referencia para cada accesor cuando cualquier thread de tiempo real es creado con el objeto `ScopedMemory` en este área de memoria, o cuando un thread de tiempo real ejecute su método `run()` para el área de memoria. Cuando la ultima referencia al objeto es eliminada, al salir del thread o del método `enter()`, los finalizadotes se ejecutan para todos los objeto del área de memoria y ésta es vaciada.

- **SizeEstimator**

Esta clase mantiene una estimación de la cantidad de memoria requerida para almacenar una serie de objetos.

- **VTMemory**

El tiempo de ejecución de una asignación de un área de `VTMemory` puede tomarse como una cantidad variable de tiempo. Sin embargo, desde que las áreas de `VTMemory` no están sujetas a la recolección de basura y los objetos de su interior pueden ser no movidos, estas áreas pueden ser usadas por instancias de `NoHeapRealtimeThread`.

C. Bibliografía

1. The Real-Time Specification for Java (RTSJ) obtenida de www.rtj.org.

2. Concurrent and Real-Time Programming in Java. Capítulo 8. Andy Wellings.

3. El artículo "Towards an Understanding of the Behavior of the Single Parent Rule in the RTSJ Scoped Memory Model". Mª Teresa Higuera Toledano.

4. El artículo "Towards a Better Memory Model for Real-time Java". Mª Teresa Higuera Toledano.

5. Apuntes de Programación Concurrente de Modesto Tomás Saavedra obtenidos de su página web: http://www.dlsi.ua.es/~abia/PC/material/programacion-concurrente-java.htm

6. Tutorial de Java obtenido en la página:
   http://proton.ucting.udg.mx/tutorial/java/Intro/tabla.html

7. Apuntes de la asignatura "Sistemas de tiempo real" obtenidos de http://dtm.unicauca.edu.co/esptelematica/sist-tiempo-real.html

8. Tutorial de Java obtenido de la página :
   http://www.programacion.com/java/tutorial/threads/

9. Apuntes de la asignatura "Ampliación de sistemas operativos" obtenidos en http://www.infor.uva.es/~jvegas/docencia/aso.html

10. Sobre los sistemas de tiempo real, de la página:
    http://www.geocities.com/txmetsb/sistemas-de-tiempo-real.htm

11. Sobre los recolectores de basura:
    http://es.wikipedia.org/wiki/Recolecci%C3%B3n_de_basura

12. Sobre la máquina virtual de Java:
    http://es.wikipedia.org/wiki/M%C3%A1quina_virtual_de_Java

13. Apuntes del Lenguaje Java, Especificación para tiempo real de Jose Manuel Pérez Lobato, obtenidos de la página:
    http://arcos.inf.uc3m.es/~ii_pstr/STRjava.pdf