

Memoria de prácticas de Programación Concurrente

Curso 2006 -07

UNED
Centro Asociado de A Coruña

Víctor M. Álvarez Pérez
C.A. de A Coruña

Índice

1.-	Enunciado	3
2.-	Trabajo del alumno	3
2.1.-	Planteamiento general	3
2.2.-	Estructuras de datos	4
2.3.-	Problemas a evitar	5
2.4.-	Descripción de cada tipo de proceso	6
3.-	Codificación de la práctica	8
3.1.-	Taller de pintura con semáforos en PascalFC	9
	Código	10
3.2.-	Taller de pintura con monitores en PascalFC	22
	Código	23
3.3.-	Problemas relativos a la codificación en PascalFC	37
3.4.-	Taller de pintura con monitores en Java	38
	Código	40
	Clase TallerMonitores	40
	Clase Bedel	41
	Clase Profesor	42
	Clase Alumno	44
	Clase VariablesGlobales	50
	Clase RepositorioCuadros	50
	Clase Taller	52
	Clase ColaFIFO	63
	Clase Pizarra	65
	Clase Caballete	67
	Clase Peticion	68
	Clase Mesa	69
3.5.-	Taller de pintura con semáforos en Java	71
	Código	73
	Clase TallerSemaforos	73
	Clase VariablesGlobales	74
	Clase Profesor	75
	Clase Alumno	77
	Clase Bedel	85
	Clase Taller	87
	Clase SemaforoConCola	92
	Clase SemaforoGeneral	95
	Clase SemaforoBinario	96
	Clase Hilos	97
	Clase Overflow	98
	Clase Empty	99
4.-	Bibliografía	100

1.- Enunciado.

Escribir dos programas en pseudocódigo que simulen la situación descrita en el problema; uno utilizando monitores y otro semáforos. Utilizar tres tipos de proceso: Alumno, Profesor y Bedel. A continuación programar en un lenguaje de programación distinto (preferiblemente PascalFC, Java o Ada) cada una de las soluciones descritas en pseudocódigo.

En un taller de pintura se realizan copias de obras clásicas. La primera vez el profesor entra en la sala y propone cinco títulos de cuadros famosos, de entre una lista de posibilidades, para que algunos de los alumnos ralicen una copia de memoria. Hay diez alumnos en la sala del taller. Para pintar un cuadro hace falta, además del título, un caballete, de los cuales hay cinco en la sala, pintura y un pincel. Los alumnos pueden llamar a un bedel para que traiga un pincel o un bote de pintura, que deposita en una mesa antes de volverse a marchar. En la mesa sólo hay sitio para un pincel o un bote. El bedel no puede traer más objetos hasta que la mesa quede libre. Cuando un alumno termina de pintar un cuadro deja el caballete libre, avisa al profesor para que proponga otro cuadro, sale al pasillo un rato y después vuelve al taller.

2.- Trabajo del alumno.

2.1.- Planteamiento general.

Tal como yo lo entiendo tenemos que manejar cuatro recursos: (a) pizarra para escribir los títulos de cuadros, (b) caballetes, (c) panel de peticiones al bedel –que pueden ser de tipo bote o pincel-, y (d) mesa para recoger los pedidos –que siguen pudiendo ser botes o pinceles. Con este planteamiento y teniendo en cuenta que deben existir tres tipos de procesos (Profesor, Alumno y Bedel) parece que corresponde bastante bien con varios problemas de tipo Productor/Consumidor: (a) respecto a los títulos, el profesor se encarga de generarlos y cada alumno de consumirlos ; (b) respecto a los caballetes un mismo alumno se encarga de retirar uno de los existentes (se comporta como un consumidor de los mismos) y de devolverlo (producirlo) cuando ha acabado; (c) el panel de peticiones (o también el oído del bedel) sirve para que cada alumno ponga una en cuanto haya sitio (productor) y el bedel se encarga de atenderlas (consumidor) cuando toque; (d) en la mesa el bedel pone (produce) utensilios de pintura, mientras que cada alumno retira (consume) uno cuando le toque. En resumen:

- i.- Proceso Profesor: Produce títulos.
- ii.- Proceso Alumno: Consume títulos, caballetes y utensilios de pintura (de la mesa). Produce peticiones.
- iii.- Proceso Bedel: Consume peticiones. Produce utensilios de pintura.

Respecto al tipo de recursos utilizados, tanto el pincel como el bote son ilimitados (puede haber en el aula una cantidad de ambos determinada solamente por el número de alumnos), mientras que los caballetes y títulos disponibles en cada momento son limitados (hay como máximo cinco caballetes que se pueden utilizar a la vez y también como máximo cinco títulos que pueden ser pintados a la vez). Existe también un recurso implícito del que hablaremos más adelante: el acceso a la pantalla para emitir mensajes.

2.2.- Estructuras de datos.

Como el tratamiento de estos recursos se puede hacer de manera similar por corresponder todos ellos a una estrategia de Productor/Consumidor, las estructuras de datos que podemos utilizar para manejar estos recursos pueden ser arrays (matrices unidimensionales) en los que vamos introduciendo o eliminando elementos. Las estrategias de acceso a estos arrays pueden ser diversas: acceso directo mediante un índice, colas con o sin prioridad, colas circulares con o sin prioridad, etc. En general nos ha parecido más adecuado implementar el acceso a los recursos de una manera homogénea y hemos escogido colas circulares con prioridad (salvo en uno de los programas en PascalFC, que hemos preferido acceso directo mediante un índice para uno de los recursos, por probar el comportamiento). De hecho nos hemos decidido por colas FIFO puesto que parece razonable que se vayan consumiendo los recursos en el mismo orden en que se van produciendo (aunque esto no es imprescindible) y por ser una de las estructuras empleadas con más profusión en la asignatura (debido a su facilidad de implementación, uso y aprovechamiento de la memoria del sistema).

Tenemos así cuatro colas FIFO: pizarra (máximo de cinco elementos), caballetes disponibles (máximo de cinco elementos), panel de peticiones (máximo de una petición simultánea) y mesa (máximo de un objeto simultáneamente). Insisto en que el hecho de emplear las mismas estructuras corresponde a hacer homogéneo su uso y facilitar la programación (por ejemplo al unificar las clases en java), aunque lo razonable para mesa y panel de peticiones sería una variable del tipo correspondiente en vez de un array de dichas variables (con tamaño unidad).

Tenemos por tanto los siguientes elementos globales al programa (en el sentido de que deben ser conocidos por todos los tipos de proceso):

- i.- Número de alumnos, número de títulos en la pizarra, número máximo de títulos que puede proponer el profesro, número de caballetes disponibles, número de peticiones simultáneas en el oído del bedel y en la mesa (todas ellas serán constantes de tipo entero).
- ii.- Pizarra, almacén de caballetes, panel de peticiones y mesa (todos ellos arrays).
- iii.- Los índices y tamaños correspondientes a cada uno de los anteriores arrays (índices y tamaño de tipo entero).

En el array pizarra podemos poner un identificador numérico del título (array de entero). En el array caballetes disponibles podemos poner un valor booleano (disponible o no disponible) o incluso podemos prescindir totalmente del array aunque no de sus índices y tamaño, puesto que el hecho de consumirse un caballete hace que ya no esté disponible en el array (no sea accesible); hemos escogido la segunda posibilidad por ahorro de memoria: no tenemos un array físico en memoria pero sí implícito puesto que utilizaremos los índices de frente y cola así como su tamaño. Los arrays mesa y tablón de peticiones hemos preferido mantenerlos puesto que el programa podría ampliarse de forma que hubiese más bedeles atendiendo peticiones (o un mismo bedel más eficiente) e incluso más mesas (o una mesa con más capacidad). El tipo de elemento de los arrays mesa y tablón de peticiones puede ser: un identificador numérico (0 ó 1, por ejemplo), un boolean (true bote y false bote, o lo mismo con pincel), un tipo de dato específico, una cadena de caracteres, etc. Hemos preferido crear un tipo de dato específico en PascalFC (que evita errores con valores incorrectos) y usar cadenas de caracteres en Java por la facilidad de uso y por no aumentar la cantidad de clases definidas (aunque perdemos en fiabilidad por poderse colar en algún sitio una cadena incorrecta).

De la misma manera, tanto los semáforos como los monitores empleados tendrán que ser comunes a todos los procesos, por lo que se deben definir como elementos globales al programa.

2.3.- Problemas a evitar.

El acceso a cada uno de los cuatro recursos debe ser en *exclusión mutua* (para mantener la consistencia de los datos almacenados) y esperando previamente a poder acceder a los mismos. La espera no puede ser *espera ocupada* sino que cada proceso que quiera acceder a un recurso ocupado debe bloquearse hasta que se le notifique que ya puede acceder al mismo (o que puede evaluar nuevamente el acceso al recurso). Con esto se evita el bajo rendimiento del sistema (en cuanto a concurrencia y capacidad de proceso) al no tener que estar evaluando de manera continuada una condición sino solamente en los momentos en que es requerido para ello.

Para evitar los *interbloqueos* no puede ocurrir que dos procesos adquieran recursos limitados (finitos) y los retengan en espera de que el otro libere el que les hace falta, sin poder avanzar ninguno de los dos. En nuestro caso los recursos que pueden presentar este problema son los caballetes y los títulos de la pizarra. Para evitar los interbloqueos en estos recursos no podemos dejar que cada proceso coja alegremente uno de los dos y se ponga a esperar por el otro. En lugar de ello impondremos un orden de acceso a los dos recursos: nos ha parecido razonable considerar que antes de retener un título de la pizarra se posea un caballete, puesto que en el momento que se tenga el título se puede empezar a pintar inmediatamente (sin tener que trasladar antes el caballete hasta el lugar de trabajo). Sin embargo también sería válida la estrategia de reservar primero un título y obtener después un caballete.

Se debe asegurar también que no haya un proceso o grupo de procesos que no consigan progresar en su ejecución por no ser planificados (*inanición*). No vamos a utilizar prioridades en la ejecución de ningún proceso, lo que hace que el tratamiento por parte del planificador sea paritario. Además el hecho de que en el programa exista un número de procesos finito (aunque pueda ser grande) y que cada proceso requiera una cantidad finita y fija de elementos de cada recurso asegura que todos ellos serán planificados para ejecución en un tiempo finito (si se asegura que no hay interbloqueos). En PascalFC los monitores se proporcionan con colas FIFO para bloquearse en la entrada a los mismos (lo cual asegura que todos los procesos se planificarán en un tiempo finito: el primero en bloquearse antes que el segundo, etc). Sin embargo los semáforos de PascalFC no incluyen un tratamiento FIFO a sus colas de bloqueo, sino que se asegura que se desbloqueará uno de los procesos de la cola al hacer una llamada a signal (esto no es un problema por lo antes comentado: número finito de procesos que requieren un número finito y fijo de elemento de cada recurso). En Java usaremos los semáforos propuestos en el libro de Palma et al, añadiéndoles una cola con tratamiento FIFO, sin embargo no implementaremos colas FIFO en los monitores (por comodidad en el diseño y legibilidad). Esto último tampoco es problemático por lo antes comentado.

No vamos a imponer un *orden fijo* a la hora de adquirir los recursos ilimitados, de hecho escogeremos aleatoriamente si cada proceso empieza por pedir un bote o un pincel. Sin embargo si que nos quedaremos a la espera del elemento que se ha solicitado: si se ha pedido un pincel se consulta inmediatamente (o después de ‘dormirse’ el proceso un breve intervalo de tiempo) la mesa para recogerlo. La intención es modelar de forma lo más realista posible el comportamiento de un grupo de alumnos en un aula: un alumno hace la petición y si ya hay un utensilio disponible lo coge, aunque ese en particular no le correspondiese a él (el esperar por su propia petición llevaría a un disminución de la concurrencia).

Ya hemos dicho que los recursos limitados es necesario obtenerlos (y liberarlos) en un orden fijo, pero pueden pedirse en cualquier orden intercalados con los ilimitados, siempre que se mantenga el orden entre los finitos. Es decir, si hemos decidido que la política sea pedir antes los caballetes que reservar los títulos, es posible que un alumno pida aleatoriamente bote, pincel o caballete (primero caballete, después pincel, después bote, o primero bote, después caballete, después pincel, etc) siempre que a continuación reserve un

título de la pizarra (y no antes de un caballete). Esto que parece tan obvio expresado así no es lo que pensamos inicialmente, sino que decidimos usar como política de acceso a los recursos: primero escoger entre bote o pincel (pidiendo a continuación el que falta de los dos) y después pedir caballete antes de reservar título. Ahora somos conscientes de que la mejor política es la explicada en primer lugar en este párrafo y que no es más difícil de implementar en la práctica que la segunda (es solamente hacer un pequeño cambio en el código del proceso Alumno para que incluya en la aleatoriedad al recurso caballete). Hemos optado por modificar la codificación inicial y adoptar la primera estrategia de las explicadas: primero pedir aleatoriamente escogiendo entre tres de los recursos –lo cual incluye a uno de los recursos limitados- y a continuación solicitar el último. De esta manera los resultados observados se corresponden con lo exigido por la especificación y no solamente de forma aproximada (como ocurría con la otra política de acceso).

Respecto a los otros problemas mencionados en el guión de prácticas, hemos realizado pruebas que aseguran que no hay errores de sincronización ni de intercambio de información, ni tampoco errores de simulación. Al menos hasta donde hemos podido revisar tanto en relación con el comportamiento de los programas como en relación a su código. Para asegurar esto hemos mantenido los programas funcionando varias horas (algunos toda la noche) y revisado después los mensajes emitidos en varios grupos escogidos aleatoriamente de 50 a 100 líneas.

2.4.- Descripción de cada tipo de proceso.

En base a lo explicado hasta el momento e independientemente de la primitiva de concurrencia empleada, cada proceso debe hacer los siguientes:

Proceso Profesor:

- (a) Indicar el trabajo inicial (a poder ser sin que un alumno o bedel lo interrumpa)
- (b) Repetir indefinidamente: (i) esperar a que se le pida un nuevo título
(ii) escribir el nuevo título en la pizarra.

El párrafo (a) se debería ejecutar en exclusión mutua, o por lo menos el proceso Alumno debería sincronizarse con la finalización de esta parte del Profesor. En el párrafo (b) el Profesor se debe ejecutar concurrentemente con el Alumno de forma que el primero espere a que el segundo le notifique que ya puede escribir un nuevo cuadro (apartado i); a continuación debe acceder a la pizarra (apartado ii) en exclusión mutua para escribir un nuevo título y notificar que ya hay un título en la misma.

Proceso Bedel:

- Repetir indefinidamente:
- (a) esperar a que se le haga una petición
 - (b) atender la petición (retirla del tablón)
 - (c) ir al almacén por el objeto
 - (d) esperar a que haya sitio en la mesa
 - (e) poner el objeto en la mesa

El párrafo (a) se debe ejecutar concurrentemente con el Alumno, de forma que el Bedel reciba noticia de que hay una nueva petición; el párrafo (b) se debe ejecutar en exclusión mutua puesto que accede al recurso tablón de pedidos. El párrafo (c) se puede simular simplemente pasando el proceso al estado dormido (se ausenta del aula). El párrafo (d) se debe ejecutar concurrentemente con los procesos de tipo Alumno, al esperar el Bedel a ser notificado de que hay un sitio en la mesa donde poner el objeto pedido; para poner el

objeto en la mesa –párrafo (e)- debe acceder a la misma en exclusion mutua con los otros procesos y al acabar notificar concurrentemente que hay un nuevo objeto en la mesa.

Proceso Alumno:

- (a) Esperar a que el profesor ponga el trabajo del día
- (b) Repetir indefinidamente: (i) generar un orden de petición de recursos aleatorio
- (ii) pedir el primer recurso (bote, pincel o caballete)
- (iii) recoger el primer recurso (bote, pincel o caballete)
- (iv) pedir el segundo recurso (bote, pincel o caballete)
- (v) recoger el segundo recurso (bote, pincel o caballete)
- (vi) pedir el tercer recurso (bote, pincel o caballete)
- (vii) recoger el tercer recurso (bote, pincel o caballete)
- (viii) reservar un título de la pizarra
- (ix) hacer copia del cuadro
- (x) devolver el caballete
- (xi) notificar el consumo de título (al profesor)
- (xii) descansar en el pasillo

El párrafo (a) debe esperar a que el profesor acabe: es decir, o finaliza la exclusión mutua, o el profesor indica que ya se puede empezar a trabajar. A continuación, los párrafos (ii), (iv) y (vi) se deben ejecutar concurrentemente con los otros procesos de tipo Alumno y con el Bedel en el caso de que el recurso sea bote o pincel. Los procesos (iii), (v) y (vii) se deben ejecutar en exclusión mutua con otros Alumno y la notificación de que se puede acceder a la mesa (y de que se puede poner otro objeto) en concurrencia con el Bedel. El párrafo (viii) se debe hacer en exclusión mutua con los otros procesos Alumno y concurrentemente con el Profesor (no es necesario sincronización con otros Alumno puesto que no se elimina por el momento el título de la pizarra). Los párrafos (ix) y (xii) se pueden simular haciendo entrar al proceso en el estado dormido durante un tiempo para que desaparezca de la escena. En el párrafo (x) se debe devolver el caballete concurrentemente con otros alumnos, pero se debe asegurar la exclusión mutua en el acceso al almacén. El párrafo (xi) se debe ejecutar en exclusión mutua con otros procesos Alumno (puesto que es el momento en el que se borra el título de la pizarra) y concurrentemente con el proceso Profesor, puesto que está esperando para escribir un nuevo título.

Todo esto se explicará de una forma más estructurada empleando PascalFC (en lugar de pseudocódigo). Se ha preferido usar PascalFC por su similitud con el pseudocódigo empleado en el libro de texto de la asignatura y de otras (como Ingeniería del Software); porque la simplicidad de la sintaxis y estructura del lenguaje añade poca complejidad al programa expresado en pseudocódigo puro; porque el hecho de poder compilarlo permite detectar errores tanto de diseño como de sintaxis y porque el profesor de la asignatura permite su uso como alternativa al pseudocódigo puro.

Un problema que hemos detectado –particularmente en PascalFC- es el de la exclusión mutua en el acceso a la pantalla del ordenador. Hemos visto que muchos mensajes se veían interrumpidos por otros antes de sacar todo el texto por pantalla. Particularmente aquellos mensajes que eran resultado de una llamada a write() o writeln() con parámetros: la función no se ejecuta de forma atómica, puesto que cuando consulta el valor del parámetro (por

ejemplo el número del alumno o el título del cuadro) después de haber sacado el texto anterior al mismo, el proceso que llama a `write()` o `writeln()` es susceptible de ser interrumpido por planificarse otro para ejecución, que a su vez puede sacar por pantalla un mensaje que consecuentemente aparecerá (si no es interrumpido él también) antes de que se complete el que ha sido interrumpido.

Debido a este problema hemos considerado el monitor del ordenador (preferimos emplear el término pantalla porque monitor se puede confundir con la primitiva de programación concurrente) como un recurso implícito, y aunque no hemos diseñado una estrategia de acceso al mismo similar a la de los otros recursos, sí que hemos procurado que éste (el acceso) se haga en exclusión mutua: tanto aprovechando el acceso a otros recursos para emitir mensajes, como obteniendo de forma explícita la exclusión mutua. Para ello hemos puesto las llamadas a función de salida por pantalla entre sus correspondientes `wait(mutex)`-`signal(mutex)` al emplear semáforos, o bien en el caso de usar monitores hemos incluido estos mensajes dentro del correspondiente monitor que asegura la exclusión mutua.

Finalmente hemos implementado la misma estrategia en la codificación en Java para evitarnos estos problemas desde el principio, aunque es poco probable que se dé la misma situación que en PascalFC. Sospechamos que las cadenas que se sacan por pantalla se elaboran de una sola vez antes de visualizarse, lo cual a efectos del observador es como si el acceso a la pantalla se hiciese de forma atómica.

3.- Codificación de la práctica.

En vista de que el guión de prácticas exigía la codificación de la práctica en pseudocódigo y que el equipo docente permitía usar PascalFC como pseudocódigo a la vez que como lenguaje de programación concurrente, se ha preferido codificar la práctica en PascalFC (tanto para semáforos como para monitores) y aprovechar el esfuerzo ahorrado para hacer lo mismo en Java. Así presentamos la práctica (tanto para semáforos como para monitores) codificada en ambos lenguajes. En todos los casos se ha tratado de reutilizar la mayor cantidad posible de código, particularmente en lo referente a Java. También se ha tratado de documentar al máximo todo el código escrito, para que cada programa sea lo más inteligible posible. En Java se ha procurado documentar el código de acuerdo con las exigencias de javadoc.

Para realizar la práctica se ha utilizado el compilador de PascalFC proporcionado por el profesor de la asignatura corriendo en un Windows 98 emulado mediante VirtualPC en un ordenador con MacOSX (iBook G4 con procesador Motorola-IBM PPC). La razón es que el compilador proporcionado para Linux no corría en MacOSX ni siquiera tratando de recompilar las fuentes. La parte correspondiente a Java se ha realizado usando el entorno BlueJ recomendado por el profesorado de Programación III, debido a su facilidad de uso y a la poca experiencia del alumno en dicho lenguaje de programación. El entorno BlueJ se encuentra disponible para Linux, MacOSX y Windows en bluej.org. De todas formas se ha hecho la comprobación del correcto funcionamiento de los programas utilizando la máquina virtual de Java desde el terminal ya que este último proporciona herramientas más versátiles para consultar los mensajes emitidos por pantalla.

Es de señalar que el entorno BlueJ proporciona herramientas para la visualización de relaciones entre clases que facilitan mucho la fase de diseño de la aplicación, así como la fase de codificación y depuración. En cuanto a la documentación interna de la aplicación, el entorno proporciona herramientas para visualizar la misma consistentes con la API de Java y fáciles de usar: en la ventana de visualización del código de cada clase se puede accionar un selector para cambiar a modo Javadoc.

Los documentos que constituyen esta práctica son: este documento (la memoria de prácticas), los dos archivos fuente codificados en PascalFC (uno para monitores y otro para

semáforos) junto con sus correspondientes programas compilados para Windows, los dos proyectos BlueJ en Java (semáforos y monitores) listos para funcionar bajo el entorno (una vez instalado el mismo) tanto en Windows como en otro SO distinto y dos carpetas conteniendo únicamente las clases sin compilar de ambos proyectos en Java.

3.1.- Taller de pintura con Semáforos en PascalFC.

Para codificar esta parte de la práctica hemos empleado los semáforos que incluye PascalFC como primitivas de sincronización. No nos hemos dado cuenta hasta haber acabado esta práctica de que las colas asociadas a los semáforos en PascalFC no implementan ningún tipo de prioridad. Sin embargo esto no afecta (al menos no lo parece) al comportamiento ni al rendimiento del programa.

Hemos utilizado los semáforos de tres formas distintas: (a) para asegurar la exclusion mutua, (b) para indicar cuándo se puede acceder a un recurso para producir un elemento, y (c) para indicar cuándo se puede acceder a un recurso para consumir un elemento. Al primer tipo pertenece el semáforo binario ‘mutex’ (se inicializa a uno). Al segundo tipo pertenecen todos los semáforos cuyo nombre acaba en ‘-Vacio’: cuadrosVacio, peticionVacia, mesaVacia, caballetVacio; en ellos se bloquean los procesos productores. Se inicializan todos a su valor máximo (nPeticiones o nCaballetes) indicando disponibilidad, salvo cuadrosVacio que se inicializa a 0 porque es el profesor quien debe escribir antes en la pizarra y a continuación indicar que ya hay títulos disponibles mediante las correspondientes llamadas a signal(). Al tercer tipo (tipo c) corresponden los semáforos cuyo nombre acaba en ‘-Lleno’: cuadrosLleno, peticionLleno, mesaLleno, caballetLleno y en ellos se bloquen los procesos consumidores. Se inicializan todos a 0 puesto que inicialmente no hay ningún recurso disponible para consumir.

Se ha procurado ser lo más claro posible en el código y documentar el mismo lo más posible con comentarios.

```
program TallerPintura;
const
    nMaxCuadros = 50;          (*N maximo de cuaros que se pueden proponer*)
    nCuadros = 5;              (*N de cuadros propuestos simultaneamente*)
    nAlumnos = 15;             (*N de alumnos en la clase*)
    nPeticones = 1;            (*N de peticiones simultaneas al bedel*)
    nCaballetes = 5;           (*N de caballetes en la clase*)
    nRecursos = 3;             (*N de recursos que se pueden pedir en cualquier
                                orden*)

type
    peticion = (bote, pincel);  (*Tipo de peticiones posibles*)
    tipoOrden = record          (*Tipo para el orden de peticion*)
        o: array[1..nRecursos] of integer;
    end;

var
    cuadros: array[1..nCuadros] of integer;    (*Pizarra para apuntar los cuadros*)
    frenteCuadros, colaCuadros: integer;        (*Punteros para manejar la pizarra*)

    (*Indicadores de posesion de cuadro, bote, pincel y caballete para cada alumno*)
    tengoCuadro, tengoBote, tengoPincel, tengoCaballete:
        array[1..nAlumnos] of boolean;        (*Hay que inicializar a 'false'*)

    (*Indicador de que caballete es el que tiene cada alumno*)
    queCaballeteTengo:                        (*No es necesario inicializarlo, el*)
        array[1..nAlumnos] of integer;        (*acceso se protege con semaforos*)

    (*Indicador de disponibilidad de cada caballete*)
    caballeteLibre: array[1..nCaballetes] of boolean;    (*Se inicializa a 'true'*)

    (*Indicadores de peticion hecha y peticion atendida*)
    peticionBedel, mesa: peticion;            (*No es necesario inicializarlos, el
                                                acceso esta protegido por semaforos*)

    (*Semaforos*)
    mutex: semaphore;                        (*Semaforo binario para exclusion mutua*)
    cuadrosVacio, cuadrosLleno,              (*Semaforos para accesos a la pizarra*)
    peticionVacio, peticionLleno,            (*Semaforos para peticiones al Bedel*)
    mesaVacia, mesaLlena:                   (*Semaforos para recoger de la mesa*)
        semaphore;                          (*Inic: Vacio a nRecurso, Lleno a 0*)
    caballetVacio, caballetLleno:            (*Semaforos para acceder al caballete*)
        semaphore;                          (*Inic: Vacio a NCab, Lleno a 0*)

    (*Proceso para escribir por pantalla el titulo del cuadro que se propone (profesor)
    o que se va a pintar (alumno).

    Necesita saber si es el profesor (true/false) o el alumno; en el primer caso
```

prescinde del parametro alumno y en el segundo necesita el identificador (entero) del alumno. Necesita tambien el numero identificador (entero) del cuadro. Hace una conversion interna del identificador para que no se salga de los que dispone en la lista.

NOTA: Por haber escogido como funcion de dispersion la funcion modulo, pueden aparecer muchas colisiones. Es decir, a pesar de que la funcion reciba distintos enteros en 'numCuadro', el resto de la division entre 20 puede ser el mismo, por lo que la cadena de salida (el título del cuadro) será la misma. Con otra eleccion de la funcion de dispersion o del tratamiento del repositorio de titulos no tendríamos este comportamiento. Se deja como esta porque no afecta a la especificacion de la practica.

```
*)
procedure escribeCuadro(profesor: boolean; alumno: integer; numCuadro: integer);
var
    nCuad: integer;
begin
    if(profesor) then
        write('Profesor: He anotado en la pizarra ')
    else
        write('Alumno ', alumno:2, ': Escojo para pintar ');

    nCuad := (numCuadro mod 20)+1;  (*El '+1' ajusta mod con numCuadro*)
    case nCuad of
        1: writeln("La rendicion de Breda");
        2: writeln("La balsa de la Medusa");
        3: writeln("Los girasoles");
        4: writeln("Los fusilamientos del 3 de Mayo");
        5: writeln("El Gernika");
        6: writeln("El jardin de las delicias");
        7: writeln("Las ilusiones");
        8: writeln("Las senoritas de Avignon");
        9: writeln("El gran pino");
        10: writeln("Los emigrantes");
        11: writeln("El gran masturbador");
        12: writeln("El matrimonio Arnolfini");
        13: writeln("Los jugadores de cartas");
        14: writeln("La joven de la perla");
        15: writeln("La catedral de Rouen");
        16: writeln("La estacion de Saint Lazare");
        17: writeln("El almuerzo de los remeros");
        18: writeln("El grito");
        19: writeln("El beso");
        20: writeln("Las meninas");
    end
end;
end;
```

(*Proceso que se encarga producir cuadros bajo demanda.

En primer lugar inicializa la 'pizarra' con una cantidad nCuadros de titulos de cuadros (obtenida aleatoriamente de una lista de nMaxCuadros), actualiza el

semaforo que indica que la pizarra contiene cuadros (por ello no se inicializa en el cuerpo principal del programa) e indica que cuadros ha escrito en la pizarra. Se comprueba antes que no haya titulos repetidos.

A continuacion espera a que se le indique que hay un hueco en la pizarra y genera aleatoriamente otro titulo que escribe en la pizarra. Finalmente indica que la pizarra contiene un titulo mas y vuelve a esperar a que haya un nuevo hueco.

La pizarra se trata como una cola FIFO de tamano nCuadros e implementada mediante un array circular, para ello cuenta con dos indicadores:

frenteCuadros y colaCuadros; el segundo lo actualiza el productor (profesor) y el primero lo actualiza el consumidor (obtenerCuadro).

```
*)
process profesor;
var
    i, j: integer;
    seRepite: boolean;
    numeroCuadro: integer;
begin
    writeln('Profesor: Buenos dias, os voy a anotar en la pizarra el trabajo de hoy');
    wait(mutex);
    for i:=1 to nCuadros do                (*Inicializa pizarra*)
    begin
        repeat                            (*Genera y comprueba que no se repita*)
            seRepite := false;
            numeroCuadro := (random(nMaxCuadros-1)) + 1;  (*Genera titulo*)
            for j:= 1 to (colaCuadros-1) do
            begin
                if(cuadros[j] = numeroCuadro) then (*Comprueba repetidos*)
                    seRepite := true;
            end;
        until(not seRepite);
        cuadros[colaCuadros] := numeroCuadro;
        colaCuadros := (colaCuadros mod nCuadros) + 1;
    end;

    for i:=1 to nCuadros do                (*Indica nuevo cuadro por cada uno escrito*)
        signal(cuadrosLleno);

    for i:=1 to nCuadros do                (*Saca por pantalla los titulos escritos*)
        escribeCuadro(true, 0, cuadros[i]);
        writeln('Profesor: Ya teneis los ', nCuadros:2, ' primeros cuadros en la
pizarra');
        signal(mutex);

    repeat
        wait(cuadrosVacio);                (*Espera a que se le indique que escriba otro*)
        wait(mutex);
        repeat                            (*Genera y comprueba que no se repita*)
            seRepite := false;
            numeroCuadro := (random(nMaxCuadros-1)) + 1;  (*Genera titulo*)
```

```
        for j:=frenteCuadros to (colaCuadros-1) do
        begin
            if(cuadros[j] = numeroCuadro) then (*Comprueba repetidos*)
                seRepite := true;
            end;
        until(not seRepite);

        cuadros[colaCuadros] := numeroCuadro;
        colaCuadros := (colaCuadros mod nCuadros) + 1;
        escribeCuadro(true, 0, numeroCuadro);
        signal(mutex);
        signal(cuadrosLleno); (*Senala que hay nuevo cuadro en la pizarra*)
    forever;
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de esperar y obtener un titulo para copiar. Es decir consume los cuadros que produce profesor.

Para ello espera a que el profesor indique que ya hay un nuevo titulo, accede en exclusiva a la pizarra y se queda con el cuadro, actualiza el indicador de cola de la pizarra (cola FIFO), actualiza el indicador de posesion de un cuadro, indica que cuadro ha cogido y libera la exclusion mutua.

No indica que hay un espacio vacio en la pizarra, puesto que hay que esperar a que el alumno acabe de copiar el cuadro para pedirle al profesor que proponga otro.

*)

```
procedure obtenerCuadro(alumno:integer);
```

```
var
```

```
    cuadro: integer;
```

```
begin
```

```
    wait(cuadrosLleno);                (*Espera a que en la pizarra haya un cuadro*)
```

```
    wait(mutex);                      (*Acceso en exclusion mutua*)
```

```
    cuadro := cuadros[frenteCuadros]; (*Obtiene cuadro*)
```

```
    frenteCuadros := (frenteCuadros mod nCuadros) +1;(*Actualiza cola y posesion*)
```

```
    tengoCuadro[alumno] := true;
```

```
    escribeCuadro(false, alumno, cuadro);
```

```
    signal(mutex);                    (*Libera exclusion mutua*)
```

```
    (*No hacemos 'signal(cuadrosVacio)' porque lo haremos cuando notifiemos  
    al profesor que escriba otro cuadro.*)
```

```
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de avisar al profesor para que proponga un nuevo cuadro.

Esta es la parte que complementa a obtenerCuadro, puesto que activa el semaforo cuadrosVacio. De la misma manera, lo hace en exclusion mutua.

*)

```
procedure avisarProfesor(alumno:integer);
```

```
begin
```

```
wait(mutex);                (*Obtiene exclusion mutua*)
tengoCuadro[alumno] := false; (*Actualiza posesion*)
writeln('Alumno ', alumno:2, ': Aviso al profesor que he acabado');
signal(mutex);              (*Libera exclusion mutua*)
signal(cuadrosVacio);       (*Libera un espacio en la pizarra*)
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de producir peticiones que consumira el proceso Bedel. Las peticiones pueden ser 'bote' o 'pincel', para ello se emplea un parametro 'pedido' de tipo peticion que indica al bedel que utensilio se precisa.

Se produce una peticion cuando el Bedel no tiene ninguna pendiente (semaforo binario) y se accede al 'oido' del Bedel en exclusion mutua. Al acabar se le indica al Bedel que hay una peticion.

*)

```
procedure pedirBedel(pedido:peticion);
begin
    wait(peticionVacio);      (*Espera a que no haya peticiones pendientes*)
    wait(mutex);             (*Obtiene exclusion mutua*)
    peticionBedel := pedido;
    signal(mutex);           (*Libera exclusion mutua*)
    signal(peticionLleno);    (*Informa de que hay una peticion pendiente*)
end;
```

(*Este procedimiento enmascara al procedimiento pedirBedel, para que sea mas clara la lectura del programa principal.

Llama a pedirBedel con el valor 'bote' en el argumento.

*)

```
procedure pedirBote;
begin
    pedirBedel(bote);
end;
```

(*Este procedimiento enmascara al procedimiento pedirBedel, para que sea mas clara la lectura del programa principal.

Llama a pedirBedel con el valor 'pincel' en el argumento.

*)

```
procedure pedirPincel;
begin
    pedirBedel(pincel);
end;
```

(*El proceso bedel se encarga de recibir peticiones y aportar utensilios de pintura. Nos ha parecido mas logico implementarlo como un productor de utiles y un consumidor de peticiones.

Primero espera a que haya una peticion, cuando la hay obtiene la peticion en

exclusion mutua y en funcion de que se le pide asi indica que va a recoger; informa de que se pueden emitir mas peticiones. En el almacen se demora un poco (se ha preferido utilizar sleep en vez de una espera activa mediante un bucle for) y a continuacion espera a que la mesa quede vacia, momento en el que coloca el pedido (en exclusion mutua) e indica que ha puesto; informa de que la mesa ya tiene uno de los utensilios pedidos.

```
*)
process bedel;
var
    pedido: petition;
    espera: integer;
begin
    repeat
        wait(petitionLleno);
        wait(mutex);
        pedido := petitionBedel;
        if(pedido = bote) then
            begin
                writeln('Bedel: Me han pedido un bote, voy al almacen a buscarlo');
                espera := 3; (*Tardara mas en volver del almacen, el bote pesa mas*)
            end
        else
            begin
                writeln('Bedel: Me han pedido un pincel, voy al almacen a buscarlo');
                espera := 2; (*Tardara menos en volver, el pincel pesa menos*)
            end;
        signal(mutex);
        signal(petitionVacio);

        sleep(espera);          (*No esta disponible mientras va al almacen*)

        wait(mesaVacia);
        wait(mutex);
        mesa := pedido;
        if(pedido = bote) then
            writeln('Bedel: He puesto en la mesa un bote')
        else
            writeln('Bedel: He puesto en la mesa un pincel');
        signal(mutex);
        signal(mesaLlena);

    forever;
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de obtener el bote. Es decir es un consumidor de utiles de tipo 'bote'.

Mientras no obtenga lo que quiere (okPetition), hace lo siguiente: Espera a que haya algo en la mesa, accede a la mesa en exclusion mutua y comprueba si es lo que estaba esperando (bote), si no es asi libera la exclusion, indica que la

mesa sigue llena y se va a dar la lata a otro alumno un rato. Si hay un bote en la mesa, actualiza la posesion y el indicador de exito (se ha preferido usar dos variables distintas por claridad), lo expresa por pantalla, vacia la mesa y sale del bucle.

*)

```
procedure obtenerBote(alumno:integer);
```

```
var
```

```
    okPetition: boolean;
```

```
begin
```

```
    okPetition := false;
```

```
    repeat
```

```
    wait(mesaLlena);
```

```
    wait(mutex);
```

```
    if(mesa = bote) then
```

```
        begin
```

```
            tengoBote[alumno]:=true;
```

```
            okPetition := true;
```

```
            writeln('Alumno ', alumno:2,': He conseguido el bote de pintura');
```

```
            signal(mutex);
```

```
            signal(mesaVacía);
```

```
        end
```

```
    else
```

```
        begin
```

```
            signal(mutex);
```

```
            signal(mesaLlena);
```

```
            sleep(2);      (*Se va a dar la lata un tiempo primo con
```

```
obtenerPincel*)
```

```
        end
```

```
    until(okPetition);
```

```
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de obtener el pincel. Es decir es un consumidor de utiles de tipo 'pincel'.

Mientras no obtenga lo que quiere (okPetition), hace lo siguiente: Espera a que haya algo en la mesa, accede a la mesa en exclusion mutua y comprueba si es lo que estaba esperando (pincel), si no es así libera la exclusion, indica que la mesa sigue llena y se va a dar la lata a otro alumno un rato. Si hay un bote en la mesa, actualiza la posesion y el indicador de exito (se ha preferido usar dos variables distintas por claridad), lo expresa por pantalla, vacia la mesa y sale del bucle.

*)

```
procedure obtenerPincel(alumno:integer);
```

```
var
```

```
    okPetition: boolean;
```

```
begin
```

```
    okPetition := false;
```

```
    repeat
```

```
wait(mesaLlena);
wait(mutex);
if(mesa = pincel) then
    begin
        tengoPincel[alumno] := true;
        okPetición := true;
        writeln('Alumno ', alumno:2, ': He conseguido el pincel');
        signal(mutex);
        signal(mesaVacía);
    end
else
    begin
        signal(mutex);
        signal(mesaLlena);
        sleep(3);      (*Se va a dar la lata un tiempo primo con obtenerBote*)
    end
until(okPetición);
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de consumir caballetes.

El recurso caballetes se ha implementado como un array de boolean que hay que recorrer para encontrar el caballete vacío (se asegura que existe mediante un semaforo que controla el acceso). Se podría implementar como una cola FIFO (o de otro tipo), pero se ha preferido usar una implementación distinta (aunque sea menos eficiente para tamaños grandes de nCaballetes).

Se espera a que haya un caballete vacío (esta asegurado por los semaforos) y se busca en el array cual es (en exclusión mutua), se actualiza la ocupación y la posesión, se indica por pantalla y se informa de que hay un caballete ocupado más.

```
*)
procedure obtenerCaballete(alumno:integer);
var
    i: integer;
begin
    wait(caballeteVacío);
    wait(mutex);
    i:=1;
    while (not caballeteLibre[i]) do
        i := (i mod nCaballetes)+1;  (*mod da un valor entre 0 y nCaballetes-1*)
    caballeteLibre[i] := false;
    tengoCaballete[alumno] := true;
    queCaballeteTengo[alumno] := i;
    writeln('Alumno ', alumno:2, ': He conseguido el caballete');
    signal(mutex);
    signal(caballeteLleno);
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de producir caballetes.

Lo dicho en obtenerCaballete respecto al recurso caballetes es valido tambien aqui, asi que no lo repetimos.

Se accede al array de caballetes en exclusion mutua, se indica que el caballete poseido vuelve a estar libre, se actualiza posesion, se indica por pantalla y se libera la exclusion mutua informando de que hay un nuevo caballete desocupado.

```
*)
procedure devolverCaballete(alumno:integer);
var
    i: integer;
begin
    wait(caballetLleno);
    wait(mutex);
    i := queCaballeteTengo[alumno];    (*No es necesario actualizar
queCaballeteTengo*)
    caballeteLibre[i] := true;
    tengoCaballete[alumno] := false;
    writeln('Alumno ', alumno:2, ': Devuelvo el caballete');
    signal(mutex);
    signal(caballetVacio);
end;
```

(*Funcion para generar un orden aleatorio entre los recursos que no tienen por por que pedirse en un orden fijo.

Cada recurso se numera de 0 a nRecursos y se almacena el identificador del mismo en una variable de tipo tipoOrden, que enmascara un array de enteros. Se comprueba que no se repiten idendificadores de recurso en la petición.

```
*)
procedure generaOrdenPeticion(var orden: tipoOrden);
var
    i, j: integer;
    seRepite: boolean;
begin
    for i:=1 to nRecursos do
        begin
            repeat
                seRepite := false;
                orden.o[i] := random(nRecursos-1);(*Produce entero positivo: 0, 1 o
2*)
                for j:=1 to i-1 do
                    begin
                        if(orden.o[j] = orden.o[i]) then
                            seRepite := true;
                        end;
                    until(not seRepite);
                end;
            end;
        end;
end;
```

(*El proceso tipoAlumno se encarga de obtener primero los utiles de pintura (pincel, bote o caballete en un orden aleatorio), una vez que los tiene se encarga de obtener el titulo del cuadro (puesto que es uno de los recursos limitados y estos no se pueden pedir a la vez en cualquier orden para evitar los interbloqueos). Nos ha parecido lógico que se tenga antes el caballete que el titulo, sin embargo lo contrario también es válido siempre que se imponga un orden en el acceso a los recursos limitados. A continuación informa de que esta haciendo (pintando) y desaparece de la escena mientras pinta (esta concentrado en la pintura), cuando acaba lo indica por pantalla, devuelve caballete y avisa al profesor, se va al pasillo un rato a descansar y vuelve a desaparecer de la escena un rato, cuando ha descansado vuelve al trabajo, empezando de nuevo.

No se devuelve pincel ni bote puesto que se consideran inservible el primero y agotado el segundo; son recursos ilimitados.

Se sacan los mensajes por pantalla en exclusion mutua porque se ha observado que algunos se ven interrumpidos por mensajes de otros procesos (fundamentalmente el profesor), lo que dificulta la lectura.

*)

```
process type tipoAlumno(alumno:integer);
```

```
var
```

```
    i: integer;
```

```
    util: integer;
```

```
    ordenPeticion: tipoOrden;
```

```
begin
```

```
    wait(cuadrosLleno);          (*Espera a que el profesor empiece la clase*)
```

```
    signal(cuadrosLleno);        (*Deja el semáforo como estaba*)
```

```
    repeat
```

```
        generaOrdenPeticion(ordenPeticion);
```

```
        for i:=1 to nRecursos do
```

```
            begin
```

```
                util := ordenPeticion.o[i]+1;          (*1: bote, 2: pincel, 3: caballete*)
```

```
                case util of
```

```
                    1:
```

```
                        begin
```

```
                            pedirBote;
```

```
                            obtenerBote(alumno);
```

```
                            sleep(3);          (*Lleva el bote a su puesto*)
```

```
                        end;
```

```
                    2:
```

```
                        begin
```

```
                            pedirPincel;
```

```
                            obtenerPincel(alumno);
```

```
                            sleep(2);          (*Lleva el pincel a su puesto*)
```

```
                        end;
```

```
                    3:
```

```
begin
    sleep(2);      (*Se desplaza al almacén*)
    obtenerCaballete(alumno);
    sleep(3);      (*Vuelve cargado con el
caballete*)
end;
end;

obtenerCuadro(alumno);

wait(mutex);
writeln('Alumno ', alumno:2, ': Empiezo a pintar el cuadro');
signal(mutex);

sleep(12);        (*Concentrado mientras pinta, no habla*)

wait(mutex);
writeln('Alumno ', alumno:2, ': He acabado de pintar el cuadro');
signal(mutex);

sleep(3);         (*Ida al almacén de caballetes*)

devolverCaballete(alumno);

sleep(2);         (*Vuelta del almacén y desplazamiento hasta el
profesor*)

avisarProfesor(alumno);

sleep(3);         (*Vuelta de avisar al profesor*)

wait(mutex);
writeln('Alumno ', alumno:2, ': Me voy al pasillo a descansar');
signal(mutex);

sleep(7);         (*Desaparece mientras descansa*)

wait(mutex);
writeln('Alumno ', alumno:2, ': Vuelvo al trabajo');
signal(mutex);
forever;
end;

var
    k: integer;
    alumno: array[1..nAlumnos] of tipoAlumno; (*Array de procesos alumno*)
begin
    initial(mutex, 1);          (*Exclusion mutua, inicializa a abierto*)
    initial(cuadrosVacio, 0);   (*Pizarra con huecos, inicializa sin titulos*)
```

```
initial(cuadrosLleno, 0); (*Pizarra con titulos, inicializa sin titulos*)
initial(peticionVacio, nPeticones); (*Peticones bedel, una disponible*)
initial(peticionLleno, 0); (*Peticones bedel, no hay peticones*)
initial(mesaVacia, nPeticones); (*Espacio en la mesa, uno disponible*)
initial(mesaLlena, 0); (*Espacio en la mesa, no hay nada*)
initial(caballetVacio, nCaballetes); (*Caballetes disp, nCab disponibles*)
initial(caballetLleno, 0); (*Caballetes disp, ninguno ocupado*)

for k:=1 to nAlumnos do (*Inicializa posesion de utensilios para*)
begin (*todos los alumnos. No tienen nada.*)
    tengoCuadro[k] := false;
    tengoBote[k] := false;
    tengoPincel[k] := false;
    tengoCaballote[k] := false;
end;

for k:=1 to nCaballetes do (*Inicializa disponibilidad de
caballetes.*) (*todos disponibles.*)
begin
    caballoteLibre[k] := true;
end;

frenteCuadros := 1; (*Inicializa punteros a la cola circular*)
colaCuadros := 1;

cobegin
    profesor;
    bedel;
    for k:=1 to nAlumnos do
    begin
        alumno[k](k);
    end;
coend;
end. (*TallerPintura*)
```

3.2.- Taller de pintura con Monitores en PascalFC.

En la codificación de esta parte hemos empleado los monitores proporcionados por PascalFC como primitivas de sincronización. Somos conscientes de que el manejo de las colas del monitor se hace siguiendo una estrategia FIFO y que la salida del monitor se hace siguiendo una política DU (desbloqueo y espera urgente en una cola de cortesía). Inicialmente optamos por codificar cuatro monitores independientes –uno para cada recurso-, pero nos encontramos con que la salida por pantalla mediante `write()` o `writeln()` con parámetros (y la impresión con ‘`escribeCuadro`’ todavía más) aparecía en ocasiones interrumpida por otros mensajes. Elaboramos entonces un nuevo monitor que se encargase de gestionar únicamente la exclusión mutua para los mensajes en pantalla, tratando a ésta como un recurso explícito. Sin embargo se hacía difícil sincronizar de esta manera los mensajes emitidos con las acciones llevadas a cabo (aparecía un ligero desfase que afectaba en ocasiones al orden de la información); también ocurría que era difícil mantener algunos mensajes fuera de ciertos monitores, obteniéndose un código bastante chapucero. Se ha optado por sacrificar en cierta medida la reusabilidad de los monitores para hacer un código algo más coherente y que mantenga implícitamente la exclusión mutua en el acceso al recurso limitado pantalla.

El monitor que hemos implementado se denomina taller y exporta funciones de dos tipos: (a) funciones que producen y (b) funciones que consumen. Salvo en el caso de la pizarra, que es necesario que cada alumno reserve un título pero no lo consuma hasta que finalmente lo borra de la pizarra cuando ha acabado de copiarlo. Pertenecen al primer tipo las funciones ‘`escribirPizarra`’, ‘`devolverCaballete`’, ‘`hacerPedido`’ y ‘`ponerMesa`’. Pertenecen al segundo tipo las funciones ‘`pedirCaballete`’, ‘`atenderPedido`’, ‘`quitarBoteMesa`’ y ‘`quitarPincelMesa`’. La función consumidora de títulos de la pizarra se ha dividido en dos partes ‘`leePizarra`’, que reserva un título y lo pone fuera del alcance de los otros alumnos y ‘`borraPizarra`’, que hace efectivo el consumo de un título y notifica al profesor.

Internamente el monitor dispone de cuatro arrays de donde se consume y en donde se deposita. Estos arrays se implementan como colas circulares con estrategia de acceso FIFO. Como particularidad debemos señalar que el array de caballetes no es necesario implementarlo físicamente puesto que si un caballete está en él significa que está disponible (un sólo valor). Sólo se declaran y usan los índices que lo recorren, así como el grado de ocupación en cada momento, siendo así la cola un ‘array virtual’.

Para acceder a estos arrays disponemos de dos variables de condición por array: (a) una para bloquearse esperando que el array no siga vacío y otra (b) para bloquearse esperando que el array deje de estar lleno. Como particularidad debemos señalar que el array mesa dispone de dos variables de condición adicionales para bloquearse esperando bien por un pincel, bien por un bote (una vez que ya se estuvo esperando a que la mesa tuviese algún objeto).

Se ha procurado ser lo más claro posible en el código y documentar el mismo lo más posible con comentarios.

```
program TallerPintura;
const
    nMaxCuadros = 50;          (*N maximo de cuaros que se pueden proponer*)
    nCuadros = 5;              (*N de cuadros propuestos simultaneamente*)
    nAlumnos = 15;             (*N de alumnos en la clase*)
    nPeticones = 1;            (*N de peticiones simultaneas al bedel*)
    nCaballetes = 5;           (*N de caballetes en la clase*)
    (*nRecursos = 3;           N de recursos que se pueden pedir en cualquier
                                orden*)(*Eliminado por excederse la tabla de
                                simbolos del compilador*)

type
    tipoPeticon = (bote, pincel);    (*Tipo de peticiones posibles*)
    tipoTitulo = record               (*Enmascara el array para pasarlo al monitor*)
        t: array[1..nCuadros] of integer;
    end;
    (*tipoOrden = record             (Tipo para el orden de peticion)
        o: array[1..nRecursos] of integer;
    end;*)(*Eliminado por excederse la tabla del compilador*)
```

(*Proceso para escribir por pantalla el titulo del cuadro que se propone (profesor)
o que se va a pintar (alumno).

Necesita saber si es el profesor (true/false) o el alumno; en el primer caso prescinde del parametro alumno y en el segundo necesita el identificador (entero) del mismo. Necesita tambien el identificador (entero) del cuadro. Hace una conversion interna del identificador para que no se salga de los que dispone en la lista, de forma que siempre se pueda ampliar o reducir sin afectar al funcionamiento del taller.

NOTA: Por haber escogido como funcion de dispersion la funcion modulo, pueden aparecer muchas colisiones. Es decir, a pesar de que la funcion reciba distintos enteros en 'numCuadro', el resto de la division entre 20 puede ser el mismo, por lo que la cadena de salida (el título del cuadro) será la misma.

Con otra eleccion de la funcion de dispersion o del tratamiento del repositorio de titulos no tendríamos este comportamiento. Se deja como esta porque no afecta a la especificacion de la practica.

```
*)
procedure escribeCuadro(profesor: boolean; alumno: integer; numCuadro: integer);
var
    nCuad: integer;
begin
    if(profesor) then
        write('Profesor: He anotado en la pizarra ')
    else
        write('Alumno ', alumno:2, ': Escojo para pintar ');

    nCuad := (numCuadro mod 20)+1;  (*El '+1' ajusta mod con numCuadro*)
    case nCuad of
        1: writeln("La rendicion de Breda");
```

```
2: writeln("La balsa de la Medusa");
3: writeln("Los girasoles");
4: writeln("Los fusilamientos del 3 de Mayo");
5: writeln("El Gernika");
6: writeln("El jardin de las delicias");
7: writeln("Las ilusiones");
8: writeln("Las señoritas de Avignon");
9: writeln("El gran pino");
10: writeln("Los emigrantes");
11: writeln("El gran masturbador");
12: writeln("El matrimonio Arnolfini");
13: writeln("Los jugadores de cartas");
14: writeln("La joven de la perla");
15: writeln("La catedral de Rouen");
16: writeln("La estacion de Saint Lazare");
17: writeln("El almuerzo de los remeros");
18: writeln("El grito");
19: writeln("El beso");
20: writeln("Las meninas");

end
end;
```

(*Monitor que se encarga de la gestion de la pizarra, caballetes, pedidos y entregas. Internamente funciona como si fuesen cuatro monitores casi independientes puesto que exporta las funciones que exportarian cada uno de ellos por separado; sin embargo al ser un unico monitor evita cierto problemas ligados a la exclusion mutua.

Inicialmente optamos por codificar cuatro monitores independientes, pero nos encontramos con que la impresion en pantalla mediante 'write' o 'writeln' con parametros (y la impresion con 'escribeCuadro' todavia mas) aparecia en ocasiones interrumpida por otros mensajes. Elaboramos entonces un nuevo monitor que se encargase de gestionar unicamente la exclusion mutua para los mensajes en pantalla. Sin embargo se hacia dificil sincronizar de esta manera los mensajes emitidos con las acciones llevadas a cabo (aparecia un ligero desfase que afectaba en ocasiones al orden de la informacion); tambien ocurría que algunos mensajes era difícil mantenerlos fuera de ciertos monitores, por lo que se obtenia un codigo bastante chapucero. Se ha optado por sacrificar en cierta medida la reusabilidad de los monitores para hacer un codigo algo mas coherente y que mantenga implicitamente la exclusion mutua en el acceso al recurso limitado 'pantalla'.

Cada uno de los 'pseudo-monitores' es una variante del productor/consumidor mostrado en el libro, con modificaciones que lo adaptan a la idiosincrasia del recurso manejado.

*)

```
monitor taller;
export (*Funciones exportadas:*)
    escribirPizarra, leerPizarra, borrarPizarra, (*Gestionan la pizarra*)
    devolverCaballote, pedirCaballote, (*Gestionan el uso de los caballetes*)
```

```

hacerPedido, atenderPedido,      (*Gestionan los pedidos*)
ponerMesa, quitarBoteMesa, quitarPincelMesa; (*Gestionan la mesa*)

var

(*Variables internas para manejar la pizarra*)
tamPizarra, frentePizarra, colaPizarra: integer;      (*Punteros cola FIFO*)
pizarra: array[1..nCuadros] of integer;              (*Cola FIFO de títulos*)
pizarraNoLlena, pizarraNoVacía: condition;           (*Condiciones de bloqueo*)

(*Variables internas para manejar los caballetes*)
(*Se supone implícitamente que hay tamCaballetes sin usar, por lo que no es
necesario tener la cola en memoria, solamente sus punteros*)
tamCaballete, frenteCaballete, colaCaballete: integer; (*Punteros cola FIFO*)
cabNoLleno, cabNoVacio: condition;                   (*Condiciones de bloqueo*)

(*Variables internas para manejar los pedidos*)
tamPedido, frentePedido, colaPedido: integer;         (*Punteros cola FIFO*)
pedido: array[1..nPetición] of tipoPetición;         (*Cola FIFO de pedidos*)
pedidoNoLleno, pedidoNoVacio: condition;             (*Condiciones de bloqueo*)

(*Variables internas para manejar la mesa*)
tamMesa, frenteMesa, colaMesa: integer;              (*Punteros cola FIFO*)
mesa: array[1..nPetición] of tipoPetición;           (*Cola FIFO de utensilios*)
mesaNoLlena, mesaNoVacía, mesaSiBote,               (*Condiciones de bloqueo*)
mesaSiPincel: condition;

i: integer;                                           (*Índice de uso general dentro del monitor*)

(*Procedimiento exportado que escribe 'nEscribir' títulos en la cola FIFO pizarra.

Inicialmente será llamado con el número máximo de títulos que admite la pizarra
para inicializarla, en las siguientes llamadas se introducirán títulos de uno
en uno a medida que vaya habiendo sitio. Hace uso de un parámetro de tipo
'tipoTítulo' que enmascara el array y facilita la comprensión.
Cuando el proceso entra en el monitor llamando a esta función comprueba que
hay sitio suficiente para escribir todos los títulos del array, si no es así
se bloquea en la cola asociada a la condición 'pizarraNoLlena' (cuando sea
desbloqueado volverá a evaluar el espacio libre) dejando paso a otro proceso.
Cuando por fin pueda escribir en el array lo hará usando un bucle for sin ceder
el monitor hasta que este finalice. Una vez finalizado tratará de desbloquear
a algún proceso esperando en la cola asociada a 'pizarraNoVacía', bloqueándose
a sí mismo en la cola de cortesía en caso afirmativo antes de salir
definitivamente del monitor (estrategia DU).
*)
procedure escribirPizarra(título: tipoTítulo; nEscribir: integer);
begin
    while(tamPizarra + nEscribir > nCuadros) do
        delay(pizarraNoLlena);      (*Se bloquea si no hay espacio*)
    for i:=1 to nEscribir do
        begin

```

```
        pizarra[colaPizarra] := titulo.t[i];
        colaPizarra := (colaPizarra mod nCuadros) + 1;
        tamPizarra := tamPizarra + 1;
        escribeCuadro(true, 0, titulo.t[i]);    (*Informa del titulo generado*)
    end;
    resume(pizarraNoVacia);                    (*Desbloquea consumidor al finalizar*)
end;
```

(*Procedimiento exportado que lee un titulo de la pizarra y lo consume o deja en funcion del parametro 'reservar'. Este parametro es necesario para que cada alumno espere a que haya algun titulo en la pizarra y no se lance a pedir recursos sin que haya titulos disponibles al principio.

Si el procedimiento se quiere usar en su faceta de consumidor, se ejecutara la parte 'else', comprobandose primero que haya titulos en la pizarra (en caso negativo se bloqueara el proceso hasta que los haya). A continuacion se reserva un cuadro y se actualiza el puntero 'frente' (observese que no se actualiza el tamaño ni se llama a 'resume' puesto que notificaremos el consumo del titulo cuando hayamos finalizado el cuadro). Se informa tambien del titulo escogido y de que se empieza a pintar. A continuacion se sale del monitor permitiendo la entrada a un nuevo proceso o uno que ya estuviese en la cola de cortesia (no se desbloquea nada).

Si se quiere usar en su faceta de 'observador', se comprueba primero que haya algun titulo en la pizarra: si lo hay se sale del monitor sin hacer nada y si no lo hay se espera en la cola 'pizarraNoVacia' hasta desbloquearse, momento en el que se desbloquea a su vez al siguiente proceso de la lista y se sale sin hacer nada mas.

```
*)
procedure leerPizarra(alumno: integer; reservar: boolean);
begin
    if(not reservar) then
    begin
        if(tamPizarra = 0) then                (*Consulta pizarra pero no*)
        begin
            delay(pizarraNoVacia);             (*consume ningun titulo y*)
            resume(pizarraNoVacia);            (*desbloquea en cadena*)
        end;
    end
    else
    begin
        if(tamPizarra = 0) then
            delay(pizarraNoVacia);              (*Se bloquea si no hay titulos*)
        i := pizarra[frentePizarra];
        frentePizarra := (frentePizarra mod nCuadros) + 1;
        escribeCuadro(false, alumno, i);        (*Informa del titulo escogido*)
        writeln('Alumno ', alumno:2, ': Tengo todo, empiezo a pintar el cuadro');
    end;
end;
```

(*Procedimiento exportado que hace efectivo el consumo de un titulo.

Al entrar de nuevo en el monitor se informa de la finalizacion del cuadro, se actualiza la cantidad de titulos en la pizarra (uno menos) y se trata de desbloquear algun proceso que este esperando a que la pizarra no este llena. Cuando finalmente se sale de la cola de cortesia (estrategia DU), se informa de que se va al pasillo (no se hace el descanso en el monitor porque sino se bloquearia todo el taller mientras un alumno descansa).

```
*)
procedure borrarPizarra(alumno: integer);
begin
    writeln('Alumno ', alumno:2, ': Aviso al profesor que he acabado');
    tamPizarra := tamPizarra - 1;
    resume(pizarraNoLlena);                (*Desbloquea productor*)
    writeln('Alumno ', alumno:2, ': Me voy al pasillo a descansar');
end;
```

(*Procedimiento exportado que produce un caballete libre.

Se bloquea si hay demasiados caballetes libres, actualiza punteros y desbloquea algun proceso esperando por caballete. Realmente no seria necesario comprobar si hay sitio para caballetes libres, ya que si se devuelve es porque previamente se consumio (siempre que los procesos respeten las reglas de juego) y necesariamente tiene que haber al menos un sitio.

Se deja la comprobacion por simetria y por facilitar la comprension del procedimiento al ser parecido a otros.

```
*)
procedure devolverCaballete(alumno: integer);
begin
    writeln('Alumno ', alumno:2, ': He acabado de pintar el cuadro');
    if(tamCaballete = nCaballetes) then
        delay(cabNoLleno);                (*Se bloquea si no hay espacio*)
    colaCaballete := (colaCaballete mod nCaballetes) + 1;
    tamCaballete := tamCaballete + 1;
    writeln('Alumno ', alumno:2, ': Devuelvo el caballete');
    resume(cabNoVacio);                  (*Desbloquea consumidor*)
end;
```

(*Procedimiento exportado que consume un caballete libre.

Se bloquea si no hay caballetes disponibles hasta que es desbloqueado por otro proceso que aporte uno, actualiza los punteros y la cantidad de caballetes, informa del exito y trata de desbloquear algun proceso que este esperando a que el sitio de los caballetes no este lleno.

Al igual que en el procedimiento anterior esta ultima sentencia no deberia ser necesaria, se deja por coherencia con otros recursos y para facilitar la comprension.

```
*)
procedure pedirCaballete(alumno: integer);
begin
    if(tamCaballete = 0) then
```

```
        delay(cabNoVacio);                                (*Se bloquea si no hay caballetes*)
        frenteCaballete := (frenteCaballete mod nCaballetes) + 1;
        tamCaballete := tamCaballete - 1;
        writeln('Alumno ', alumno:2, ': He conseguido el caballete');
        resume(cabNoLleno);                                (*Desbloquea productor*)
    end;
```

(*Procedimiento exportado que produce pedidos para ser atendidos por el bedel.

Lleva un parametro 'primeraVez' que indica si es la primera vez que el proceso llama a este procedimiento, si no es asi se indica que el alumno vuelve al taller dispuesto a hacer una nueva copia.

Este procedimiento es similar a 'devolverCaballete' con la salvedad de que se emplea una cola FIFO implementada realmente con un array de 'tipoPetición' puesto que ahora se pueden introducir dos tipos de elementos (y no un unico tipo como en el procedimiento mencionado).

```
*)
procedure hacerPedido(util: tipoPetición; alumno: integer; primeraVez: boolean);
begin
    if(not primeraVez) then
        writeln('Alumno ', alumno:2, ': Vuelvo al trabajo');

        if(tamPedido = nPetición) then
            delay(pedidoNoLleno);                                (*Se bloquea si no hay espacio*)
            pedido[colaPedido] := util;
            colaPedido := (colaPedido mod nPetición) + 1;
            tamPedido := tamPedido + 1;
            resume(pedidoNoVacio);                                (*Desbloquea consumidor*)
        end;
end;
```

(*Funcion exportada que consume pedidos, usado por el bedel cuando presta atencion a uno nuevo.

Similar a 'pedirCaballete' con la salvedad de usar un array real para almacenar los dos tipos de pedidos y emitir mensajes distintos.

```
*)
procedure atenderPedido(var util: tipoPetición);
begin
    if(tamPedido = 0) then
        delay(pedidoNoVacio);                                (*Se bloquea si no hay pedidos*)
        util := pedido[frentePedido];
        frentePedido := (frentePedido mod nPetición) + 1;
        tamPedido := tamPedido - 1;
        if(util = bote) then
            begin
                writeln('Bedel: Me han pedido un bote, voy al almacen a buscarlo');
            end
        else
            begin
                writeln('Bedel: Me han pedido un pincel, voy al almacen a buscarlo');
            end
        end
    end;
end;
```

```
end;  
resume(pedidoNoLleno);          (*Desbloquea productor*)  
end;  
  
(*Procedimiento exportado que produce utensilios en respuesta a los pedidos,  
depositandolos en una mesa (representada por un array de 'tipoPetición').  
  
Similar a 'devolverCaballote' y 'hacerPedido', selecciona el mensaje y el tipo  
de proceso que tratara de desbloquear en funcion del tipo de utensilio  
depositado en la mesa. Si en la mesa se deposita un bote, trata de desbloquear  
al (o a los si nPeticiónes>1) proceso que esta esperando un la cola asociada  
a 'mesaSiBote'; si por el contrario se deposita un pincel, hace lo mismo  
en la cola asociada a 'mesaSiPincel'. Finalmente tratara de desbloquear algun  
proceso que este esperando a que la mesa tenga algun utensilio independientemente  
del tipo.  
)  
procedure ponerMesa(util: tipoPetición);  
begin  
    if(tamMesa = nPeticiónes) then  
        delay(mesaNoLlena);          (*Se bloquea si no hay espacio en la mesa*)  
        mesa[colaMesa] := util;  
        colaMesa := (colaMesa mod nPeticiónes) + 1;  
        tamMesa := tamMesa + 1;  
        if(util = bote) then  
            begin  
                writeln('Bedel: He puesto en la mesa un bote');  
                resume(mesaSiBote);          (*Desbloq consum esperando por bote*)  
            end  
        else  
            begin  
                writeln('Bedel: He puesto en la mesa un pincel');  
                resume(mesaSiPincel);          (*Desbloq consum esperando por pincel*)  
            end  
        end;  
        resume(mesaNoVacía);          (*Desbloquea consumidor*)  
    end;  
end;  
  
(*Procedimiento exportado que consume utensilios de tipo 'bote'.  
  
Primero se bloquea en la cola asociada a 'mesaNoVacía' en caso de que no haya  
ningun utensilio de pintura, en cuanto hay un utensilio comprueba que sea del  
tipo que esta esperando (si no es asi se bloquea en la cola 'mesaSiBote'), en  
cuyo caso lo consume, informa y antes de salir trata de desbloquear un proceso  
esperando a que la mesa no este ocupada.  
)  
procedure quitarBoteMesa(alumno: integer);  
begin  
    if(tamMesa = 0) then  
        delay(mesaNoVacía);          (*Se bloquea si no hay nada en la mesa*)  
    if(mesa[frenteMesa] <> bote) then  
        delay(mesaSiBote);          (*Se bloquea si no hay un bote*)  
        frenteMesa := (frenteMesa mod nPeticiónes) + 1;
```

```
tamMesa := tamMesa - 1;
writeln('Alumno ', alumno:2,': He conseguido el bote de pintura');
resume(mesaNoLlena);          (*Desbloquea productor*)
end;
```

(*Procedimiento exportado que consume utensilios de tipo 'pincel'.

Primero se bloquea en la cola asociada a 'mesaNoVacía' en caso de que no haya ningún utensilio de pintura, en cuanto hay un utensilio comprueba que sea del tipo que está esperando (si no es así se bloquea en la cola 'mesaSiPincel'), en cuyo caso lo consume, informa y antes de salir trata de desbloquear un proceso esperando a que la mesa no esté ocupada.

*)

```
procedure quitarPincelMesa(alumno: integer);
begin
    if(tamMesa = 0) then
        delay(mesaNoVacía);          (*Se bloquea si no hay nada en la mesa*)
    if(mesa[frenteMesa] <> pincel) then
        delay(mesaSiPincel);          (*Se bloquea si no hay un pincel*)
    frenteMesa := (frenteMesa mod nPeticiónes) + 1;
    tamMesa := tamMesa - 1;
    writeln('Alumno ', alumno:2,': He conseguido el pincel');
    resume(mesaNoLlena);          (*Desbloquea productor*)
end;
```

(*Cuerpo principal, inicialización de las variables internas del monitor*)

```
begin
    tamPizarra := 0; frentePizarra := 1; colaPizarra := 1;
    tamCaballote := nCaballotes; frenteCaballotes := 1; colaCaballotes := 1;
    tamPedido := 0; frentePedido := 1; colaPedido := 1;
    tamMesa := 0; frenteMesa := 1; colaMesa := 1;
end;  (*Del monitor taller*)
```

(*Proceso que se encarga producir cuadros bajo demanda.

En primer lugar inicializa la 'pizarra' con una cantidad nCuadros de títulos de cuadros (obtenida aleatoriamente de una lista de nMaxCuadros) e indica que cuadros ha escrito. Inicialmente comprueba que no repite ningún título en la pizarra.

A continuación se bloquea esperando que haya un nuevo hueco en la pizarra y genera aleatoriamente otro título que escribe en la pizarra. Esta parte se repite indefinidamente. No hace comprobación de repetidos, si lo quisiésemos implementar tendríamos que modificar el monitor.

*)

```
process profesor;
var
    i, j: integer;
    seRepite: boolean;
    numeroCuadro: tipoTitulo;
begin
```

```
writeln('Profesor: Buenos dias, os voy a anotar en la pizarra el trabajo de hoy');
for i:=1 to nCuadros do
begin
    repeat
        seRepite := false;
        numeroCuadro.t[i] := (random(nMaxCuadro)) + 1; (*Genera titulo*)
        for j:=1 to i-1 do
        begin
            if(numeroCuadro.t[j] = numeroCuadro.t[i]) then
                seRepite := true; (*Comprueba repetidos*)
        end;
    until(not seRepite);
end;

taller.escribirPizarra(numeroCuadro, nCuadros);

repeat
    numeroCuadro.t[1] := (random(nMaxCuadros-1)) + 1; (*Genera titulo*)
    taller.escribirPizarra(numeroCuadro, 1); (*Escribe titulos de uno en uno*)
forever;
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de obtener un titulo para copiar. Es decir consume los cuadros que produce profesor.

Enmascara la llamada a taller.leerPizarra para facilitar la lectura del procedimiento alumno. El parametro 'reserva' es true para indicar que se va a consumir ese titulo.

```
*)
procedure obtenerCuadro(alumno:integer);
begin
    taller.leerPizarra(alumno, true);
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de consultar si existen titulos en la pizarra.

Al igual que el anterior enmascara la llamada a taller.leerPizarra para facilitar la lectura del proceso alumno. El parametro 'reserva' es false para indicar que todavia no se va a consumir ningun titulo.

```
*)
procedure consultarPizarra(alumno: integer);
begin
    taller.leerPizarra(alumno, false);
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de avisar al profesor para que proponga un nuevo cuadro. Hace efectivo el consumo del cuadro que se reservo previamente con taller.leerPizarra.

Esta es la parte que complementa a obtenerCuadro.

```
*)  
procedure avisarProfesor(alumno:integer);  
begin  
    taller.borrarPizarra(alumno);                (*Libera un espacio en la pizarra*)  
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de producir peticiones que consumira el proceso Bedel. Las peticiones pueden ser 'bote' o 'pincel', para ello se emplea un parametro 'pedido' de tipo tipoPeticion que indica al bedel que utensilio se precisa.

Se produce una peticion cuando el Bedel no tiene ninguna pendiente (monitor con un hueco) y se accede al 'oido' del Bedel en exclusion mutua, si el Bedel esta ocupado atendiendo otra peticion, el alumno se bloquea esperando a que el Bedel acabe.

El parametro primeraVez indica si es o no la primera vez que se pide un utensilio, si no es asi de debe indicar que el alumno vuelve al trabajo.

```
*)  
procedure pedirBedel(pedido: tipoPeticion; alumno: integer; primeraVez: boolean);  
begin  
    taller.hacerPedidos(pedido, alumno, primeraVez);  
end;
```

(*El proceso bedel se encarga de recibir peticiones y aportar utensilios de pintura. Nos ha parecido mas logico implementarlo como un productor de utiles y un consumidor de peticiones.

Primero espera a que se haga una peticion (taller.atender), cuando la obtiene evalua que se le pide y actualiza 'espera' para tardar mas o menos en funcion del utensilio, se demora en el almacen (antes ha dejado el monitor, permitiendo que se haga una nueva peticion aunque todavia no se haya puesto 'a la escucha') y a continuacion espera a que la mesa quede vacia, momento en el que coloca el pedido.

La demora en el almacen se ha preferido implementarla mediante sleep() en vez de hacer uso de un bucle for para una espera activa.

```
*)  
process bedel;  
var  
    utensilio: tipoPeticion;  
begin  
    repeat  
        taller.atenderPedido(utensilio);  
        if(utensilio = bote) then  
            begin  
                sleep(3); (*Tardara mas en volver del almacen, el bote pesa mas*)  
            end  
        else  
            begin  
                sleep(2); (*Tardara menos en volver, el pincel pesa menos*)  
            end  
    until false;  
end;
```

```
        end;

        taller.ponerMesa(utensilio);
    forever;
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de obtener el bote. Es decir es un consumidor de utiles de tipo 'bote'. Enmascara al procedimiento taller.quitarBoteMesa a fin de mejorar la legibilidad de alumno.

Se bloqueara a la entrada del monitor hasta que algun proceso del interior le deje paso, en cuyo caso comprobara si obtiene un bote. Si no obtiene un bote se bloqueara hasta que algun productor lo despierte diciendole que hay en la mesa el utensilio que espera. Saldra del monitor con el bote cuando se desbloquee de la cola de cortesia. En tal caso anuncia que tiene el bote.

*)

```
procedure obtenerBote(alumno:integer);
begin
    taller.quitarBoteMesa(alumno);
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de obtener el pincel. Es decir es un consumidor de utiles de tipo 'pincel'. Enmascara al procedimiento taller.quitarPincelMesa a fin de mejorar la legibilidad de alumno.

Se bloqueara a la entrada del monitor hasta que algun proceso del interior le deje paso, en cuyo caso comprobara si obtiene un pincel. Si no obtiene un pincel se bloqueara hasta que algun productor lo despierte diciendole que hay en la mesa el utensilio que espera. Saldra del monitor con el pincel cuando s desbloquee de la cola de cortesia. En tal caso anuncia que tiene el pincel.

*)

```
procedure obtenerPincel(alumno:integer);
begin
    taller.quitarPincelMesa(alumno);
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de consumir caballetes. Por conservar el estilo enmascara a taller.pedirCaballete.

Si no hay caballetes disponibles se bloqueara a la entrada del monitor hasta que algun proceso del interior le deje paso, en tal caso consume un caballete y tratara de desbloquear un proceso en la cola de los que esperan a devolver un caballete (esperando a continuacion en la cola de cortesia). Como no habra ningun proceso en esta cola, se desbloqueara de la cola de cortesia (antes devque entre otro proceso) y simplemente saldra del monitor.

*)

```
procedure obtenerCaballete(alumno:integer);
begin
    taller.pedirCaballete(alumno);
end;
```

(*Este procedimiento es la parte del proceso Alumno que se encarga de producir caballetes. Por conservar el estilo enmascara a taller.devolverCaballete.

Como el procedimiento trata de devolver un caballete que ha consumido antes, no se bloqueara esperando un hueco en el recurso, por lo que lo devuelve y trata de despertar a un proceso que este esperando por un caballete libre (poniendose a su vez en la cola de cortesia). Saldrá de la cola de cortesia antes de que un nuevo proceso pueda entrar en el monitor.

```
*)
procedure devolverCaballete(alumno:integer);
begin
    taller.devolverCaballete(alumno);
end;
```

(*****Eliminado por excederse la tabla de simbolos del compilador. Se incluye el mismo codigo en el cuerpo del proceso Alumno*****)

(*Funcion para generar un orden aleatorio entre los recursos que no tienen por por que pedirse en un orden fijo.

Cada recurso se numera de 0 a nRecursos y se almacena el identificador del mismo en una variable de tipo tipoOrden, que enmascara un array de enteros. Se comprueba que no se repiten idendificadores de recurso en la petición.

```
*)
(*procedure generaOrdenPeticion(var orden: tipoOrden);
var
    i, j: integer;
    seRepite: boolean;
begin
    for i:=1 to nRecursos do
        begin
            repeat
                seRepite := false;
                orden.o[i] := random(nRecursos-1);(Produce un entero positivo: 0, 1 o 2)
                for j:=1 to i-1 do
                    begin
                        if(orden.o[j] = orden.o[i]) then
                            seRepite := true;
                    end;
                until(not seRepite);
            end;
        end;
    end;*)
```

(*****)

(*El proceso actividadAlumno se encarga de obtener primero los utiles de pintura (pincel o bote en un orden aleatorio), una vez que tiene pincel y bote se encarga de obtener caballete y titulo del cuadro (puesto que son recursos limitados y no es justo que se posean antes de tener los otros utiles, mientras otros alumnos puedan estar esperando, pretendemos tambien evitar los interbloques). Parece logico que se tenga antes el caballete que el titulo. Informa de que esta haciendo (pintando) y desaparece de la escena mientras pinta (esta

concentrado en la pintura), cuando acaba lo indica por pantalla, devuelve caballete y avisa al profesor, se va al pasillo un rato a descansar y vuelve a desaparecer de la escena un rato.

No se devuelve pincel ni bote puesto que se consideran inservible el primero y agotado el segundo; son recursos ilimitados.

```
*)
procedure actividadAlumno(alumno:integer; primeraVez: boolean);
var
    i, util: integer;
    (*ordenPeticion: tipoOrden;          (Se sustituye por el siguiente array por
                                         tener que disminuir la tabla de simbolos del compilador*)
    orden: array[1..3] of integer;
    seRepite: boolean;
begin
    (*generarOrdenPeticion(ordenPeticion);    (Se elimina por tener que reducir la
                                              tabla de simbolos del compilador*)
    (*Algoritmo para generar un orden aleatorio entre los
    recursos que no tienen por que pedirse en un orden fijo.
    Cada recurso se numera de 0 a nRecursos y se almacena el
    identificador del mismo en una variable de tipo tipoOrden,
    que enmascara un array de enteros.
    Se comprueba que no se repiten idendificadores de recurso en la
    petición.
    *)
    for i:=1 to 3 do          (*En vez de 3 deberiamos usar nProcesos, pero*)
    begin                    (*no cabe en la tabla de simbolos del compilador*)
        repeat              (*Lo mismo en la llamada a random()*)
            seRepite := false;
            orden[i] := random(3-1);(*Produce un entero positivo: 0, 1 o 2*)
            for util:=1 to i-1 do
            begin
                if(orden[util] = orden[i]) then
                    seRepite := true;
            end;
        until(not seRepite);
    end;

    for i:=1 to 3 do          (*Igual que más arriba deberiamos usar nProcesos*)
    begin
        util := orden[i]+1;    (*1: bote, 2: pincel, 3: caballete*)
        case util of
            1:
                begin
                    pedirBedel(bote, alumno, primeraVez);
                    obtenerBote(alumno);
                    sleep(3);    (*Lleva el bote a su puesto*)
                end;
        end;
```

```
2:
    begin
        pedirBedel(pincel, alumno, primeraVez);
        obtenerPincel(alumno);
        sleep(2);      (*Lleva el pincel a su puesto*)
    end;
3:
    begin
        sleep(2);      (*Se desplaza al almacen de caballetes*)
        obtenerCaballete(alumno);
        sleep(3);      (*Vuelve cargado con el caballete*)
    end;
end;
if((util = 1) or (util = 2)) then      (*Necesario para el mensaje de*)
    primeraVez := false;              (*volver al trabajo*)
end;

obtenerCuadro(alumno);

sleep(12);      (*Concentrado mientras pinta, no habla*)

devolverCaballete(alumno);

sleep(2);      (*Vuelta del almacen y desplazamiento hasta el profesor*)

avisarProfesor(alumno);

sleep(7);      (*Desaparece mientras descansa*)
end;

(*El proceso tipoAlumno consulta primero la pizarra para evitar que se hagan
peticiones antes de que el profesor indique el trabajo (evitando asi el barullo
de muchos alumnos llamando al bedel que puedan impedir hablar al profesor); a
cotinuacion se realiza la actividad del alumno con el parametro primeraVez a
true y una vez acabada se entra en un bucle infinito que llama a la actividad
del alumno con el parametro primeraVez a false.
*)
process type tipoAlumno(alumno:integer);
begin
    consultarPizarra(alumno);(*Comprueba que haya algun cuadro antes de empezar*)

    actividadAlumno(alumno, true);

    repeat
        actividadAlumno(alumno, false);
    forever;
end;

(*Cuerpo principal de programa, se encarga de iniciar los procesos*)
var
```

```
k: integer;
alumno: array[1..nAlumnos] of tipoAlumno; (*Array de procesos alumno*)
begin
  cobegin
    profesor;
    bedel;
    for k:=1 to nAlumnos do
      begin
        alumno[k](k);
      end;
    coend;
end. (*TallerPintura*)
```

3.3.- Problemas relativos a la codificación con PascalFC.

Nos hemos encontrado con varias limitaciones relativas al uso de este lenguaje de programación tan sencillo:

(a) No se pueden utilizar expresiones en los índices de los arrays, tanto en relación con la declaración de los mismos como en el acceso a los elementos. Es decir es ilegal la siguiente declaración:

```
var mesa: array[0..nPeticiones-1] of integer
```

y también tratar de acceder a un elemento de la siguiente manera:

```
pedido := mesa[i-1]
```

Esto supuso algunos problemas al principio de la codificación por estar acostumbrado a otra forma más flexible de trabajar con los arrays, obligando a hacer algunas ‘chapuzas’ con los índices para poder utilizarlos con operadores como ‘mod’ y funciones como ‘random()’.

(b) No se pueden pasar arrays como parámetros de una función o procedimiento, lo que nos ha obligado a encapsularlos de una forma poco natural dentro de definiciones de nuevos tipos.

(c) Las funciones de salida por monitor `write()` y `writeln()` se pueden ver interrumpidas cuando otros procesos ejecutan también salidas por pantalla –particularmente cuando en la llamada se incluyen parámetros distintos de cadenas-. Esto es una pega importante puesto que se supone que PascalFC se usará para proyectos que implementen concurrencia y esta particularidad obliga a encapsular las llamadas a estas funciones dentro de un monitor o entre semáforos que aseguren la exclusión mutua. A veces resulta poco natural encapsular de esta manera las llamadas a `write()`, resultando un código algo farragoso (en el caso de los monitores) y ciertamente algo más complejo en el caso de los semáforos, consideramos esto una limitación importante del lenguaje. En favor de esta característica hay que decir que obliga a hacerse consciente de la concurrencia en todo momento.

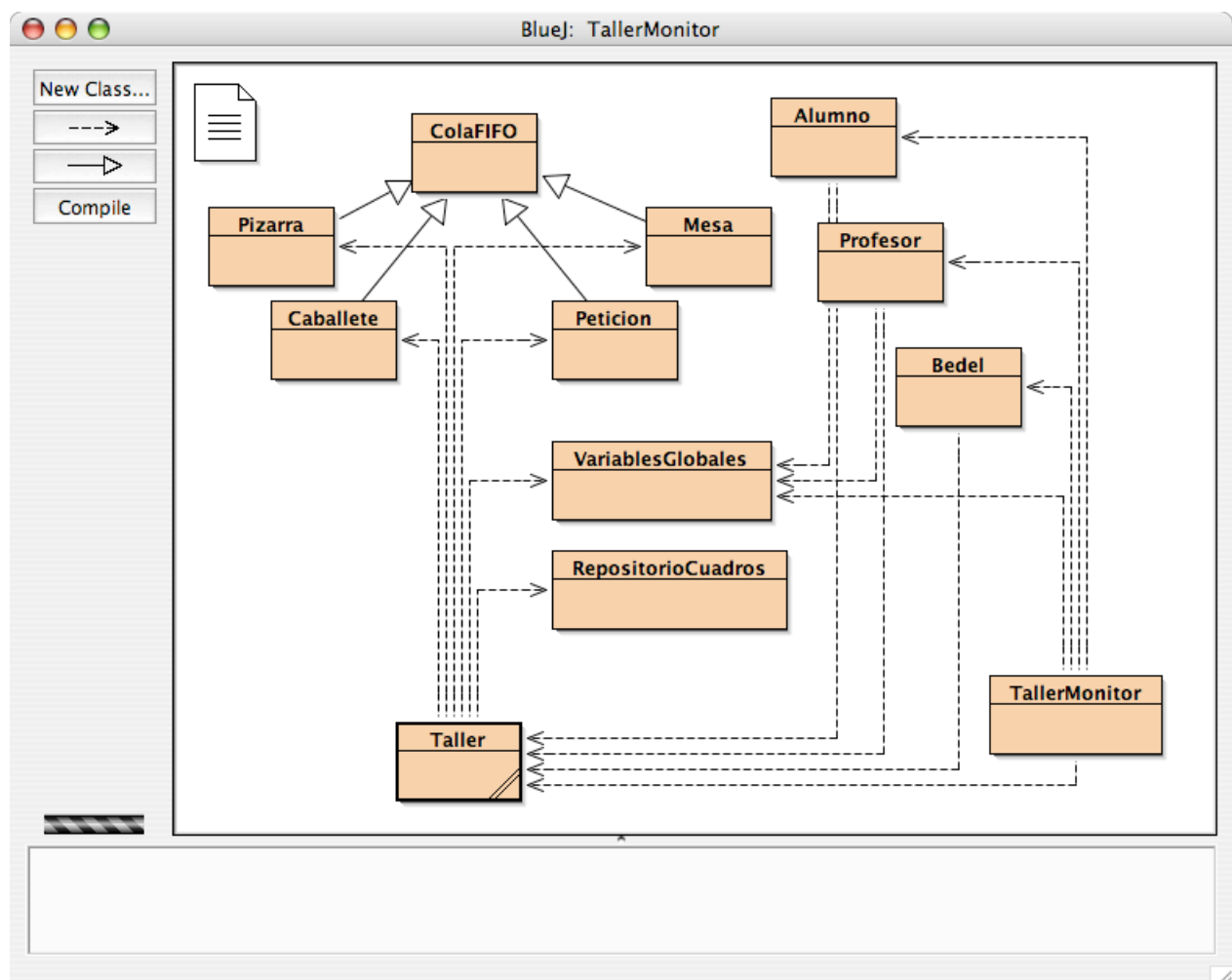
(d) El número máximo de funciones exportadas por un monitor no puede exceder de diez. Aunque esto no ha sido un problema en nuestras práctica (exportamos exactamente diez) sí es una limitación importante a la hora de realizar proyectos de mayor envergadura.

(d) La tabla de símbolos del compilador es demasiado limitada. En particular nos hemos visto obligados a reducir bastante las declaraciones en el apartado de monitores puesto que el compilador se negaba a finalizar el trabajo por excederse en varias ocasiones su tabla de símbolos. Si en el estilo de codificación se incluyen llamadas a funciones que encapsulan a otras (con vistas a mejorar la legibilidad y con intención de aprovecharlas en programación orientada a objetos), rápidamente nos encontramos con una cantidad de identificadores que superan el tamaño de la antedicha tabla. Esto obliga en ocasiones a sacrificar la legibilidad en favor de un código más corto pero más difícil de entender y por supuesto mantener. Consideramos que esto es también una limitación importante del lenguaje.

3.4.- Taller de pintura con Monitores en Java.

Para la realización de esta parte de la práctica hemos utilizado la misma estructura empleada en pseudocódigo (PascalFC). De hecho implementamos la solución en pseudocódigo pensando en aprovechar la mayor cantidad posible de código en la codificación en Java.

Hemos implementado el monitor taller (el mismo que en el apartado de PascalFC) de acuerdo con las ideas expuestas en el libro de Palma et al. Nos ha parecido más natural diseñar el monitor usando objetos compartidos, tal como recomienda el libro de la asignatura y otros consultados (particularmente el Tutor de Java de Sun). Así hemos declarado como objetos lo que en PascalFC declaramos como variables condición del monitor: pizarraNoLlena, pizarraNoVacia, caballetesNoLleno, caballetesNoVacio, pedidoNoLleno, pedidoNoVacio, mesaNoLlena, mesaNoVacia, mesaSiBote, mesaSiPincel. Estos son ahora objetos en los que se bloquearán los hilos cuando otro haya iniciado la ejecución del bloque sincronizado o bien ellos mismos hagan una llamada a wait(). Los de tipo NoLleno son objetos que sirven para bloquear a los hilos que una vez dentro del monitor se encuentran con que el recurso está lleno y necesitan poner un nuevo elemento (esperan por tanto a que no esté lleno). Los de tipo NoVacio son los que sirven para bloquear a los hilos que se encuentran con que el recurso está vacío y pretenden extraer un elemento (esperarán por tanto a que el recurso no esté vacío).



No se ha implementado una política expresa de tratamiento de los hilos bloqueados en una cola del monitor, por lo que es la propia máquina virtual de Java la que se encarga de

desbloquear a los hilos correspondientes cuando se hace una llamada a `resume()` dentro de un bloque sincronizado. Según expresa el Tutor de Java Sun –y también sugiere Palma et al– cuando se ejecuta un `resume()` se asegura que se despertará uno de los hilos que esperan en la cola correspondiente. Sin embargo no hay ningún compromiso sobre qué hilo se escogerá, ni se garantiza que se desbloqueará uno en particular.

Respecto a la representación de los recursos, se han implementado como colas FIFO al igual que se hizo en PascalFC. Sin embargo se ha diseñado una única clase –ColaFIFO– a partir de la que heredan las otras cuatro y a la vez particularizan los métodos empleados por cada una de ellas. Esta estrategia permite reutilizar el código en otros proyectos. En particular ha servido para reutilizar estas clases y crear alguna nueva en el proyecto Java correspondiente a los semáforos.

En la figura se puede observar la estructura de clases en un diagrama de relaciones (pantalla principal del proyecto en el entorno BlueJ). Los cuatro recursos son las clases Pizarra, Caballete, Peticion y Mesa, que heredan de ColaFIFO (son colas FIFO). A estos recursos se accede a través del monitor –clase Taller–, que a su vez tiene métodos que son usados por las clases Profesor, Alumno y Bedel (las líneas punteadas indican que un método es usado por la clase de donde parte la flecha). La clase TallerMonitor inicializa las clases necesarias y pone en marcha los hilos. Hemos puesto un par de clases más que ayudan a clarificar el código: VariablesGlobales, donde están las constantes usadas por las otras clases y RepositorioCuadros, donde se guardan los títulos y se proporciona un método para convertir los enteros aportados por el profesor o los alumnos en cadenas listas para imprimir por pantalla.

A continuación se listan las clases sin más comentarios que la documentación propia de la clase –preparada para ser publicadas con JavaDoc.

Clase TallerMonitor.

```
/**
 * Clase principal que contiene el método main.
 * Se encarga de crear los objetos taller, profesor, alumno y bedel,
 * pasándoles a estos tres últimos el monitor taller que debe ser común a
 * los tres.
 * Se encarga también de transformar a profesor, bedel y alumno en hilos e
 * iniciar su ejecución concurrente.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class TallerMonitor
{
    private static int nAlumnos;    //No de alumnos en la clase

    public static void main(String[] args)
    {
        nAlumnos = new VariablesGlobales().nAlumnos;

        //Crea el monitor
        Taller taller = new Taller();

        //Crea los objetos profesor, bedel y alumnos[]
        Profesor profesor = new Profesor(taller);
        Bedel bedel = new Bedel(taller);
        Alumno[] alumnos = new Alumno[nAlumnos];

        //Inicializa cada uno de los alumnos[]
        //Los alumnos se numeran de 1 a nAlumnos, aunque internamente
        //el array tenga índices entre 0 y nAlumnos-1.
        for(int i=0; i<nAlumnos; i++)
            alumnos[i] = new Alumno(i+1, taller);

        //Crea los hilos asociados a profesor, bedel y alumnos[]
        Thread hiloProfesor = new Thread(profesor);
        Thread hiloBedel = new Thread(bedel);
        Thread[] hilosAlumnos = new Thread[nAlumnos];

        //Crea cada uno de los hilos asociados a alumnos[]
        for(int i=0; i<nAlumnos; i++)
            hilosAlumnos[i] = new Thread(alumnos[i]);

        //Inicia la ejecución de los hilos
        hiloProfesor.start();
        hiloBedel.start();

        for(int i=0; i<nAlumnos; i++)
            hilosAlumnos[i].start();
    }
}
```

```
}  
}
```

Clase Bedel.

```
/**  
 * Clase que modela el comportamiento del bedel. Implementa Runnable  
 * para poder ejecutarlo como un hilo.  
 *  
 * @author Víctor M. Álvarez Pérez  
 * @version Junio de 2007  
 */  
  
public class Bedel implements Runnable  
{  
    private Taller taller;    //Monitor con todas las funciones exportadas  
  
    /**  
     * Constructor de la clase Bedel.  
     *  
     * @param taller Es el monitor que debe ser común los hilos Profesor,  
     *             Bedel y Alumno y las colas FIFO pizarra, petición,  
     *             mesa y caballete.  
     */  
    public Bedel(Taller taller)  
    {  
        this.taller = taller;  
    }  
  
    /**  
     * Método de donde parte el hilo cuando se inicia con start().  
     * Al iniciarse el método entra en un bucle infinito en el que en primer  
     * lugar se obtiene un nuevo pedido mediante la función atenderPedido()  
     * exportada por el monitor taller, se valora si el utensilio obtenido es un  
     * bote o un pincel y se actualiza 'espera' en función de ello.  
     * A continuación entra en el estado 'dormido' durante la cantidad de ms  
     * especificada en 'espera', simulando la ida y vuelta al almacén. Cuando  
     * está de vuelta pone en la mesa el útil demandado, entra en el estado  
     * 'dormido' un rato, simulando que todavía no ha consultado el tablón de  
     * pedidos y a continuación vuelve a iniciar el bucle.  
     * No es necesario preocuparse por la sincronización ni la exclusión mutua  
     * porque ya se encarga de ello el monitor. Esto hace que el código sea más  
     * legible y se eviten errores de codificación (si el monitor está bien hecho).  
     */  
    public void run()  
    {  
        String utensilio = new String();  
        int espera;  
        while(true)
```

```
{
    utensilio = taller.atenderPedido();
    if(utensilio.equals("bote"))
        espera = 600; //Tarda más en volver con el bote, pesa más.
    else
        espera = 400; //Tarda menos en volver con el pincel, pesa menos.

    hacerTiempo(espera); //Mientras va y vuelve al almacén.

    taller.ponerMesa(utensilio);
    hacerTiempo(100); //Tarda en darse la vuelta para consultar
} //el tablón de pedidos.
}

/**
 * Método privado para introducir intervalos de espera entre las acciones
 * del bedel; sirve para enmascarar el bloque try-catch y mejorar la
 * legibilidad.
 * Hace una llamada a sleep() (dentro de try-catch) para que el hilo pase
 * al estado 'dormido' y el intervalo de tiempo no sea mediante espera
 * activa.
 *
 * @param espera Tiempo en ms que debe dormir el hilo.
 */
private void hacerTiempo(int espera)
{
    try{Thread.sleep(espera);}
    catch(InterruptedException e) {}
}
}
```

Clase Profesor.

```
/**
 * Clase que modela el comportamiento del profesor. Implementa Runnable
 * para poder ejecutarlo como un hilo.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
import java.util.Random;

public class Profesor implements Runnable
{
    private int nMaxCuadros; //No máximo de cuadros en el repositorio
    private int nCuadros; //No máximo de cuadros en la pizarra
    private Random generaAzar; //Para escoger aleatoriamente un título

    private Taller taller; //Referencia al monitor taller
}
```

```
/**
 * Constructor de la clase Profesor.
 *
 * @param taller Es el monitor que debe ser común los hilos Profesor,
 *             Bedel y Alumno y las colas FIFO pizarra, petición,
 *             mesa y caballete.
 */
public Profesor(Taller taller)
{
    nCuadros = new VariablesGlobales().nCuadros;
    nMaxCuadros = new VariablesGlobales().nMaxCuadros;
    generaAzar = new Random();

    this.taller = taller;
}

/**
 * Método de donde parte el hilo cuando se inicia con start().
 * Saca por pantalla las indicaciones del principio de la clase, genera
 * una cantidad nCuadros de títulos y los escribe en la pizarra. A
 * continuación entra en un bucle infinito que hace lo siguiente: se
 * demora un poco en volver a la actividad (estado 'dormido'), genera un
 * nuevo título y lo escribe en la pizarra, volviendo a iniciar las
 * mismas acciones del bucle.
 * No es necesario preocuparse por la sincronización ni la exclusión mutua
 * porque ya se encarga de ello el monitor. Esto hace que el código sea
 * más legible y se eviten errores de codificación (si el monitor está
 * bien hecho).
 */
public void run()
{
    System.out.println("Profesor: Buenos dias, os voy a anotar en la pizarra el trabajo de
hoy");
    int[] numeroCuadro = new int[nCuadros];
    for(int i=0; i<nCuadros; i++)
        numeroCuadro[i] = generaAzar.nextInt(nMaxCuadros);

    taller.escribirPizarra(numeroCuadro, nCuadros);

    while(true)
    {
        hacerTiempo(400);    //Se demora un poco en volver a entrar

        numeroCuadro[0] = generaAzar.nextInt(nMaxCuadros);
        taller.escribirPizarra(numeroCuadro, 1);
    }
}

/**
 * Método privado para introducir intervalos de espera entre las acciones
```

```
* del profesor; sirve para enmascarar el bloque try-catch y mejorar la
* legibilidad.
* Hace una llamada a sleep() (dentro de try-catch) para que el hilo pase
* al estado 'dormido' y el intervalo de tiempo no sea mediante espera
* activa.
*
* @param espera Tiempo en ms que debe dormir el hilo.
*/
private void hacerTiempo(int espera)
{
    try {Thread.sleep(espera);}
    catch (InterruptedException e) {}
}
}
```

Clase Alumno.

```
/**
 * Clase que modela el comportamiento del alumno. Implementa Runnable
 * para poder ejecutarlo como un hilo.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
import java.util.Random;

public class Alumno implements Runnable
{
    private int alumno;      //Identificador del alumno en la clase
    private Taller taller;   //Monitor con todas las funciones exportadas
    private Random generaAzar; //Para escoger aleatoriamente bote o pincel

    /**
     * Constructor de la clase Alumno.
     *
     * @param alumno Identificador del alumno en la clase
     * @param taller Es el monitor que debe ser común los hilos Profesor,
     *             Bedel y Alumno y las colas FIFO pizarra, petición,
     *             mesa y caballete.
     */
    public Alumno(int alumno, Taller taller)
    {
        this.alumno = alumno;
        this.taller = taller;
        generaAzar = new Random();
    }

    /**
     * Método de donde parte el hilo cuando se inicia con start().
     */
}
```

```
* Consulta la pizarra para no agobiar al bedel mientras no haya ningún
* título en la misma, llama al método privado actividadAlumno() (que
* codifica las acciones del alumno) con el parámetro primeraVez a true
* y a continuación entra en un bucle infinito donde hace la misma
* llamada a actividadAlumno() pero con primeraVez a false (para que
* indique que vuelve al trabajo antes de empezar).
*/
public void run()
{
    consultarPizarra();

    actividadAlumno(true);

    while(true)
        actividadAlumno(false);
}

/**
 * Método privado que codifica las acciones del alumno.
 * Se hacen llamadas a funciones que a su vez enmascaran las funciones
 * exportadas por el monitor, para mejorar la legibilidad de este método;
 * se introducen retardos (mediante llamadas a sleep()) para simular lo mejor
 * posible la actividad en un aula.
 * Primero se escoje (aleatoriamente) qué tipo de útil de entre botes y
 * pinceles vamos a pedir por primera vez, se piden y obtienen ambos
 * (notificándolo por pantalla) y a continuación se piden el caballete
 * y el título (por este orden). Cuando ya se tiene todo, el alumno
 * desaparece de la escena durante un rato mientras está pintando. Cuando
 * acaba lo indica por pantalla, devuelve el caballete y avisa al profesor.
 * A continuación se va al pasillo un rato y desaparece de la escena
 * mientras descansa.
 * Se empieza por pedir bote o pincel en primer lugar ya que son
 * recursos ilimitados y no sería justo que un alumno acaparase primero
 * un caballete o un título (recursos limitados) sin tener los útiles de
 * pintura, mientras otros alumnos con pincel y bote tuviesen que esperar
 * para poder empezar a pintar (en el Bedel se crea una buena cola al
 * principio). También parece lógico hacer acopio del caballete antes que
 * del título, aunque también se podría implementar una estrategia en la
 * que se escogiese aleatoriamente entre los dos.
 * No se devuelve pincel ni bote puesto que se consideran inservible el
 * primero y agotado el segundo; son recursos ilimitados. La devolución
 * del caballete y el aviso al profesor se efectúan en el mismo orden en
 * que se obtuvo caballete y título para facilitar la concurrencia.
 * Los mensajes que emite el alumno se imprimen en exclusión mútua para
 * evitar la posible (aunque improbable en java) interrupción por parte
 * de otros hilos que también quieran acceder a la pantalla. Esto puede
 * ocurrir en las llamadas a println que incluyen variables. De esta forma
 * se trata implícitamente a la pantalla como un recurso limitado al que
 * se accede mediante el semáforo mutex. No es necesario otro semáforo
 * binario para controlar el acceso a la pantalla, aunque se podría
```

```
* incluir por completitud (de tratamiento de los recursos).
*
* @param primeraVez Parámetro para controlar si es la primera vez que se
*     ejecuta el método, se pasa directamente al método
*     pedirBedel.
*/
private void actividadAlumno(boolean primeraVez)
{
    int nRecursos = new VariablesGlobales().nRecursos;

    int[] ordenPeticion = generarOrdenPeticion(nRecursos); //Produce 0, 1 ó 2

    for(int i=0; i<nRecursos; i++)
    {
        switch(ordenPeticion[i])
        {
            case 0: //Si es un bote
                pedirBedel("bote", primeraVez);
                hacerTiempo(100); //Se distrae un momento
                obtenerBote();
                hacerTiempo(200); //Tarda en depositar el bote en su lugar de trabajo
                break;
            case 1: //Si es un pincel
                {
                    pedirBedel("pincel", primeraVez);
                    hacerTiempo(100); //Se distrae un momento
                    obtenerPincel();
                    hacerTiempo(200); //Tarda en depositar el pincel en su lugar de trabajo
                    break;
                }
            case 2: //Si es un caballete
                {
                    hacerTiempo(300); //Se demora un poco en ir al almacén de caballetes
                    obtenerCaballete();
                    hacerTiempo(500); //Tarda en volver cargado con el caballete
                    break;
                }
        }
    }

    obtenerCuadro();

    hacerTiempo(1500); //Le lleva cierto tiempo pintar el cuadro.

    hacerTiempo(300); //Va hasta el almacén para devolver el caballete.
    devolverCaballete();
    hacerTiempo(300); //Tarda en volver del almacén.

    avisarProfesor();
}
```

```
        hacerTiempo(700); //Descansa un buen rato.
    }

    /**
     * Método privado para generar un orden aleatorio entre los recursos que no
     * tienen por qué pedirse en un orden fijo.
     *
     * Cada recurso se numera de 0 a nRecursos-1 y se almacena el identificador
     * del mismo en un array. A continuación se comprueba que el identificador
     * generado no está ya en el array; si es así se vuelve a generar un número
     * aleatorio y se vuelve a comprobar hasta que no esté repetido. Se recorre
     * todo el array haciendo lo mismo y finalmente se devuelve dicho array.
     *
     * @param nRecursos El número de recursos que se van a solicitar aleatoriamente.
     * @return Un array conteniendo los identificadores (enteros) de recurso que
     *         se van solicitar.
     */
    private int[] generarOrdenPeticion(int nRecursos)
    {
        boolean seRepite;
        int[] orden = new int[nRecursos];
        for(int i=0; i<nRecursos; i++)
        {
            do
            {
                seRepite = false;
                orden[i] = generaAzar.nextInt(nRecursos); //Genera int entre 0 y nRecursos-1
                for(int j=0; j<i; j++) //Comprueba que no se repite
                {
                    if(orden[j] == orden[i])
                        seRepite = true;
                }
            } while (seRepite);
        }
        return orden;
    }

    /**
     * Método privado para esperar a que haya algún título en la pizarra
     * antes de que todos los alumnos empiecen a agobiar al bedel. Sirve
     * para sincronizar el inicio de las acciones de los alumnos con el
     * profesor: así no se ve interrumpido por un griterío mientras da
     * las instrucciones previas al comienzo de la clase.
     * Hace una llamada a la función leerPizarra() exportada por el monitor
     * taller. La llamada a la función exportada se hace con el identificador del
     * alumno -necesario para los mensajes por pantalla- y el indicador de
     * reserva a false -para que sólo se consulte la pizarra y no se reserve
     * ningún título-.
     */
    private void consultarPizarra()
```

```
{
    taller.leerPizarra(alumno, false);
}

/**
 * Método privado para solicitar al bedel un utensilio de pintura (bote
 * o pincel), sirve para enmascarar la llamada a la función hacerPedido()
 * exportada por el monitor taller. La llamada se hace con el tipo de pedido,
 * el identificador del alumno y un indicador de si es o no la primera vez.
 *
 * @param pedido Tipo de utensilio que se le pide al bedel (bote o pincel).
 * @param primeraVez Indicador de si ya se ha pedido algún otro utensilio
 * antes, hace que se añada un mensaje de vuelta al
 * trabajo si no es la primera vez.
 */
private void pedirBedel(String pedido, boolean primeraVez)
{
    taller.hacerPedido(pedido, alumno, primeraVez);
}

/**
 * Método privado para obtener un bote de la mesa, sirve para enmascarar
 * la llamada a la función quitarBoteMesa() exportada por el monitor taller.
 * La llamada a la función exportada se hace pasándole el identificador de
 * alumno, necesario para los mensajes por pantalla.
 */
private void obtenerBote()
{
    taller.quitarBoteMesa(alumno);
}

/**
 * Método privado para obtener un pincel de la mesa, sirve para enmascarar
 * la llamada a la función quitarPincelMesa() exportada por el monitor taller.
 * La llamada a la función exportada se hace pasándole el identificador de
 * alumno, necesario para los mensajes por pantalla.
 */
private void obtenerPincel()
{
    taller.quitarPincelMesa(alumno);
}

/**
 * Método privado para obtener un caballete del armario donde se guardan;
 * sirve para enmascarar la llamada a la función pedirCaballete() exportada
 * por el monitor taller. La llamada a la función exportada se hace pasándole
 * el identificador de alumno, necesario para los mensajes por pantalla.
 */
private void obtenerCaballete()
{

```

```
taller.pedirCaballete(alumno);
}

/**
 * Método privado para obtener un título de la pizarra donde están escritos;
 * sirve para enmascarar la llamada a la función leerPizarra() exportada por
 * el monitor taller. La llamada a la función exportada se hace pasándole el
 * identificador de alumno -necesario para los mensajes por pantalla- y el
 * indicador de reserva a true -para que se haga efectiva la reserva de un
 * título, impidiendo que otros alumnos puedan reservar el mismo-.
 */
private void obtenerCuadro()
{
    taller.leerPizarra(alumno, true);
}

/**
 * Método privado para devolver un caballete al armario donde se guardan;
 * sirve para enmascarar la llamada a la función devolverCaballete() exportada
 * por el monitor taller. La llamada a la función exportada se hace pasándole
 * el identifiador de alumno, necesario para los mensajes por pantalla.
 */
private void devolverCaballete()
{
    taller.devolverCaballete(alumno);
}

/**
 * Método privado para avisar al profesor de que se ha acabado de pintar un
 * cuadro y puede escribir un nuevo título en la pizarra; sirve para enmascarar
 * la llamada a la función borrarPizarra() exportada por el monitor taller. La
 * llamada a la función exportada se hace pasándole el identificador de alumno,
 * necesario para los mensajes por pantalla.
 */
private void avisarProfesor()
{
    taller.borrarPizarra(alumno);
}

/**
 * Método privado para introducir intervalos de espera entre las acciones
 * del alumno; sirve para enmascarar el bloque try-catch y mejorar la
 * legibilidad.
 * Hace una llamada a sleep() (dentro de try-catch) para que el hilo pase
 * al estado 'dormido' y el intervalo de tiempo no sea mediante espera
 * activa.
 *
 * @param espera Tiempo en ms que debe dormir el hilo.
 */
private void hacerTiempo(int espera)
```

```
{
    try{Thread.sleep(espera);}
    catch(InterruptedException w) {}
}
}
```

Clase VariablesGlobales.

```
/**
 * Clase para guardar las constantes globales que van a compartir las
 * otras clases. Se ha puesto para clarificar la clase principal.
 *
 * @author Víctor M. Álvarez Pérez
 * @date Junio de 2007
 */
public class VariablesGlobales
{
    public static int nMaxCuadros = 50; //No máximo de cuadros que puede poner el
profesor
    public static int nCuadros = 5;    //No máximo de cuadros que caben en la pizarra
    public static int nAlumnos = 15;   //No de alumnos en la clase
    public static int nPeticones = 1;   //No de peticiones que pueden ser atendidas por el
bedel o puestas en la mesa
    public static int nCaballetes = 5;  //No de caballetes que caben en el almacén
    public static int nRecursos = 3;    //No de recursos que se pueden pedir en cualquier
orden

    public VariablesGlobales()
    {

    }
}
```

Clase RepositorioCuadros.

```
/**
 * Modela el libro donde el profesor consulta las obras que pondrá a
 * sus alumnos. Tanto el profesor como los alumnos conocen el número
 * del cuadro y consultan en esta clase el título del mismo.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class RepositorioCuadros
{
    private static int nRepositorio = 20; //No de obras en el libro
    private static String[] repositorio; //Libro propiamente dicho
}
```

```
/**
 * Constructor de la clase RepositorioCuadros, inicializa el
 * libro asociando un título con un índice en el array.
 */
public RepositorioCuadros()
{
    repositorio = inicializaRepositorio();
}

/**
 * Inicializa el array que contiene los títulos de los cuadros.
 * Es de uso privado a la clase.
 *
 * @return Un array de cadenas, cada una conteniendo un título.
 */
private String[] inicializaRepositorio()
{
    String temp[] = { "\"La rendicion de Breda\"",
        "\"La balsa de la Medusa\"",
        "\"Los girasoles\"",
        "\"Los fusilamientos del 3 de Mayo\"",
        "\"El Gernika\"",
        "\"El jardin de las delicias\"",
        "\"Las ilusiones\"",
        "\"Las señoritas de Avignon\"",
        "\"El gran pino\"",
        "\"Los emigrantes\"",
        "\"El gran masturbador\"",
        "\"El matrimonio Arnolfini\"",
        "\"Los jugadores de cartas\"",
        "\"La joven de la perla\"",
        "\"La catedral de Rouen\"",
        "\"La estacion de Saint Lazare\"",
        "\"El almuerzo de los remeros\"",
        "\"El grito\"",
        "\"El beso\"",
        "\"Las meninas\""};

    return temp;
}

/**
 * Obtiene el título del cuadro correspondiente a un índice entero.
 * Independientemente del entero que se facilite, reduce el índice a
 * uno de los de la lista empleando el operador módulo.
 *
 * * NOTA: Por haber escogido como función de dispersion la función módulo,
 * pueden aparecer muchas colisiones. Es decir, a pesar de que la función
 * reciba distintos enteros en 'numCuadro', el resto de la división entre
 * 20 puede ser el mismo, por lo que la cadena de salida (el título del
```

```
* cuadro) será la misma.
* Con otra elección de la función de dispersion o del tratamiento del
* repositorio de titulos no tendríamos este comportamiento. Se deja como
* está porque no afecta a la especificación de la practica.
*
* @param numCuadro El número del título dado por el profesor o recibido
*                 por el alumno.
* @return Una cadena conteniendo el título del cuadro.
*/
public static String dameCuadro(int numCuadro)
{
    int nc = numCuadro % nRepositorio;

    return repositorio[nc];
}
```

Clase Taller.

```
/**
 * Monitor que se encarga de la gestión de la pizarra, caballetes, pedidos
 * y entregas. Internamente funciona como si fuesen cuatro monitores casi
 * independientes puesto que exporta las funciones que exportarían cada
 * uno de ellos por separado; sin embargo al ser un único monitor evita
 * ciertos problemas ligados a la exclusión mutua.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class Taller
{
    //Constantes del Taller de Pintura
    private int nCuadros;
    private int nPeticones;
    private int nCaballetes;

    //Almacenes de datos
    private Pizarra pizarra;
    private Caballete caballetes;
    private Peticion pedido;
    private Mesa mesa;

    //Variables de condición de las colas
    //..para la pizarra
    private Object pizarraNoLlena = new Object();
    private Object pizarraNoVacía = new Object();
    //..para el almacen de caballetes
    private Object caballetesNoLleno = new Object();
    private Object caballetesNoVacio = new Object();
}
```

```
//..para el tablón de pedidos
private Object pedidoNoLleno = new Object();
private Object pedidoNoVacio = new Object();
//..para la mesa
private Object mesaNoLlena = new Object();
private Object mesaNoVacía = new Object();
private Object mesaSiBote = new Object();
private Object mesaSiPincel = new Object();

/**
 * Constructor de la clase Taller.
 * Obtiene constantes de la clase VariablesGlobales e inicia las
 * colas pizarra, caballetes, pedido y mesa (usando las constantes
 * obtenidas antes).
 */
public Taller()
{
    nCuadros = new VariablesGlobales().nCuadros;
    nPeticones = new VariablesGlobales().nPeticones;
    nCaballetes = new VariablesGlobales().nCaballetes;

    pizarra = new Pizarra(nCuadros);
    caballetes = new Caballete(nCaballetes);
    pedido = new Peticion(nPeticones);
    mesa = new Mesa(nPeticones);
}

/**
 * Método exportado por el monitor que escribe 'nEscribir' títulos en la cola
 * FIFO pizarra.
 *
 * Inicialmente será llamado con el número máximo de títulos que admite
 * la pizarra (para inicializarla), en las siguientes llamadas se
 * introducirán títulos de uno en uno a medida que vaya habiendo sitio.
 * Cuando el hilo entra en el monitor llamando a esta función comprueba
 * que hay sitio suficiente para poner todos los títulos del array, si
 * no es así se bloquea en la cola asociada a la condición
 * 'pizarraNoLlena' (cuando sea desbloqueado volverá a evaluar el
 * espacio libre) dejando paso a otro hilo. Cuando por fin pueda
 * escribir en el array, lo hará usando un bucle for sin ceder el monitor
 * hasta que éste (el bucle) finalice. Una vez finalizado sale de la exclusión
 * mutua (sale del monitor -estrategia Desbloquear y Continuar, DC-) y se
 * vuelve a poner en la cola del monitor a la espera de volver a ser
 * seleccionado para entrar; en cuanto entre en el monitor tratará de
 * desbloquear a algún otro hilo que esté esperando en la cola asociada a
 * 'pizarraNoVacía' y sale del monitor (estrategia DC).
 *
 * @param titulo[] Array de enteros que representa los títulos que se quiere
 * escribir en la pizarra.
 * @param nEscribir Número de elementos del array que deben ser escritos en la
```

```
*          pizarra (empezando por el que tienen índice 0).
*/
public void escribirPizarra(int[] titulo, int nEscribir)
{
    synchronized (pizarraNoLlena)
    {
        while(!pizarra.sePuedePoner(nEscribir))
        {
            try {pizarraNoLlena.wait();}
            catch(InterruptedException e) {}
        }
        for(int i=0; i<nEscribir; i++)
        {
            pizarra.ponerTitulo(titulo[i]);
            escribeCuadro(true, -1, titulo[i]);
        }
    }
    synchronized (pizarraNoVacía)
    {
        pizarraNoVacía.notify();
    }
}

/**
 * Método exportado por el monitor, que lee y consume un título de la pizarra
 * en función del parámetro 'reservar'. Este parámetro es necesario para que
 * cada alumno espere a que haya algún título en la pizarra y no se lance a
 * pedir recursos sin que haya títulos disponibles al principio.
 *
 * Si se quiere usar el método en su faceta de consumidor se ejecutará el
 * bloque 'else', comprobándose primero que haya títulos en la pizarra (en
 * caso negativo se bloqueará el hilo hasta que los haya). A continuación se
 * reserva un cuadro (se actualiza el puntero frente pero no el tamaño de la
 * cola pizarra), se informa del título escogido y de que se empieza a pintar
 * y se sale del monitor permitiendo la entrada a un nuevo hilo. Obsérvese que
 * no se notifica (notify()) que se ha consumido un título puesto que se hará
 * cuando hayamos finalizado el cuadro.
 * Si se quiere usar la faceta de 'observador', se comprueba primero que haya
 * algún título en la pizarra: si lo hay se sale del monitor sin hacer nada y
 * si no lo hay se espera en la cola asociada a 'pizarraNoVacía' hasta
 * desbloquearse, momento en el que se hace una llamada a notify() para
 * desbloquear otro hilo que esté esperando (puede ser un desbloqueo en
 * cadena si todos están en su faceta de observadores) y se sale del monitor
 * sin hacer nada más.
 *
 * @param alumno Identificador del alumno en la clase, necesario para la
 * salida de mensajes por pantalla.
 * @param reservar Indica si lo que queremos es reservar un título sin
 * consumirlo (valor true) o bien observar simplemente que
 * exista algún título en la pizarra (valor false).
```

```
*/
public void leerPizarra(int alumno, boolean reservar)
{
    if(!reservar)    //Si sólo se quiere consultar que hay títulos
    {
        synchronized (pizarraNoVacia)
        {
            if(pizarra.estaVacio())
            {
                try {pizarraNoVacia.wait();} //Espera si está vacía
                catch(InterruptedException e) {}

                pizarraNoVacia.notify();    //Desbloquea en cadena
            }
        }
    }
    else    //Si se quiere reservar un título en propiedad
    {
        synchronized (pizarraNoVacia)
        {
            while(pizarra.estaVacio())
            {
                try {pizarraNoVacia.wait();} //Espera si está vacía
                catch(InterruptedException e) {}
            }
            int l = pizarra.reservarTitulo();
            escribeCuadro(false, alumno, l);
            System.out.println("Alumno " + alumno + ": Tengo todo, empiezo a pintar el
cuadro");
        }
        //No hace efectivo el borrado del título, puesto que enseguida
        //se escribiría otro en su lugar.
        //Espera a borrarlo al momento en que haya finalizado el cuadro
        //y se lo vaya a notificar al profesor.
    }
}

/**
 * Método exportado que hace efectivo el consumo de un título.
 *
 * Al entrar de nuevo en el monitor (recuérdese que este método complementa
 * a leerPizarra en su faceta de consumidor) se informa de la finalización
 * del cuadro, se actualiza la cantidad de títulos haciendo una llamada al
 * método borrarTitulo() de la cola pizarra (que únicamente llama al método
 * disminuirTamanho() heredado de ColaFIFO) y trata de desbloquear algún
 * proceso que esté esperando a que la pizarra no esté llena (operación
 * notify() sobre el objeto pizarraNoLlena). Finalmente el método sale del
 * monitor (estrategia desbloquear y continuar -DC-).
 *
 * @param alumno Identificador del alumno en la clase, necesario para la
```

```
*          salida de mensajes por pantalla.
*/
public void borrarPizarra(int alumno)
{
    synchronized (pizarraNoLlena)
    {
        System.out.println("Alumno " + alumno + ": Aviso al profesor y me voy al pasillo
a descansar");
        pizarra.borrarTitulo(); //Hace efectivo el borrado
        pizarraNoLlena.notify(); //Desbloquea un productor
    }
}

/**
 * Método exportado que produce un caballete libre.
 *
 * Se bloquea esperando por el acceso al monitor, en cuanto consigue entrar
 * comprueba si hay demasiados caballetes libres (en cuyo caso se vuelve a
 * bloquear a la entrada del monitor mediante una llamada a wait() -a la vuelta
 * de wait() se vuelve a comprobar la condición dentro del bucle while-), si
 * no los hay actualiza punteros mediante la llamada a ponerCaballete() de la
 * cola caballetes, informa de que ha acabado y sale de la exclusión mutua
 * (sale del monitor, estrategia DC). A continuación se queda a la espera
 * del monitor nuevamente y cuando vuelve a entrar trata de desbloquear algún
 * hilo que esté esperando en el objeto caballetesNoVacio (llamada a notify()).
 *
 * Realmente no sería necesario comprobar si hay sitio para caballetes libres,
 * ya que si se devuelve es porque previamente se consumió un caballete (siempre
 * que los hilos respeten las reglas del juego) y necesariamente tiene que
 * haber al menos un sitio. Se mantiene la comprobación por simetría y por
 * facilitar la comprensión del método al ser parecido a otros del tipo
 * productor/consumidor.
 *
 * @param alumno Identificador numérico del alumno en la clase.
 */
public void devolverCaballete(int alumno)
{
    synchronized (caballetesNoLleno)
    {
        //System.out.println("Alumno " + alumno + ": He acabado de pintar el cuadro");
        while(caballetes.estaLleno())
        {
            try {caballetesNoLleno.wait();}
            catch(InterruptedException e) {}
        }
        caballetes.ponerCaballete();
        System.out.println("Alumno " + alumno + ": He acabado de pintar el cuadro,
devuelvo el caballete");
    }
    synchronized (caballetesNoVacio)
```

```
{
    caballetesNoVacio.notify();
}
}

/**
 * Método exportado que consume un caballete libre.
 *
 * Espera a entrar en el monitor, cuando lo consigue hace la comprobación
 * del bucle while: si la cola de los caballetes está vacía (no se puede
 * conseguir uno) entra en el while y se bloquea nuevamente a la entrada del
 * monitor mediante una llamada a wait(). Cada vez que sea desbloqueado por
 * otro hilo volverá a hacer la comprobación hasta que haya un caballete y
 * pueda salir del while (sin salir del monitor). A continuación obtiene un
 * caballete y lo indica por pantalla, cediendo el monitor. Se vuelve a poner
 * en la cola del monitor para finalmente notificar a algún hilo que esté
 * esperando en el objeto caballetesNoLleno que ya hay al menos un caballete.
 * Finalmente sale del monitor.
 *
 * @param alumno Identificador numérico del alumno en la clase.
 */
public void pedirCaballete(int alumno)
{
    synchronized (caballetesNoVacio)
    {
        while(caballetes.estaVacio())
        {
            try {caballetesNoVacio.wait();}
            catch(InterruptedException e) {}
        }
        caballetes.quitarCaballete();
        System.out.println("Alumno " + alumno + ": He conseguido el caballete");
    }
    synchronized(caballetesNoLleno)
    {
        caballetesNoLleno.notify();
    }
}

/**
 * Método exportado que produce pedidos para ser atendidos por el bedel.
 *
 * Al entrar al monitor se comprueba mediante el parámetro 'primeraVez'
 * si no es o si es la primera vez que se hace uso del método (en cuyo caso
 * se indica o no por pantalla que se vuelve al trabajo), a continuación se
 * comprueba la condición del bucle while para ver si se puede poner un nuevo
 * pedido (si no es así se bloquea nuevamente a la entrada del monitor
 * mediante una llamada al procedimiento wait() del objeto pedidoNoLleno,
 * volviendo a comprobar la condición del while cada vez que es desbloqueado).
 * En caso afirmativo se pone el pedido en la cola 'pedido' mediante la
```

```
* llamada al método ponerPedido() y se sale del monitor. Se vuelve a poner
* en la cola del monitor y una vez que entra trata de desbloquear un hilo
* que esté esperando en el objeto pedidoNoVacio, finalmente sale del monitor.
*
* @param util Cadena de caracteres que indica el tipo de utensilio que se va
*         a poner en la cola de pedidos. Puede ser 'bote' o 'pincel', pero
*         no se hace comprobación de la corrección de la cadena.
* @param alumno Identificador numérico del alumno en la clase.
* @param primeraVez Indicador de si es la primera vez (true) o no (false) que
*         se hace un pedido al bedel.
*/
public void hacerPedido(String util, int alumno, boolean primeraVez)
{
    synchronized (pedidoNoLleno)
    {
        if(!primeraVez)
            System.out.println("Alumno " + alumno + ": Vuelvo al trabajo");
        while(pedido.estaLleno())
        {
            try {pedidoNoLleno.wait();}
            catch(InterruptedException e) {}
        }
        pedido.ponerPedido(util);
    }
    synchronized (pedidoNoVacio)
    {
        pedidoNoVacio.notify();
    }
}

/**
 * Método exportado que consume pedidos, usado por el bedel cuando presta
 * atención a uno nuevo.
 *
 * Cuando gana el acceso al monitor lo hace en exclusión mutua con los hilos
 * que están esperando en el objeto pedidoNoVacio, espera a tener también la
 * exclusión mutua sobre el objeto mesaNoLlena y comprueba si se puede poner
 * un objeto en la mesa (no atiende pedidos mientras no se haya retirado el
 * anterior de la mesa): si la mesa está llena se bloquea en la entrada del
 * monitor y cuando se vuelve a desbloquear notifica que la mesa no está
 * llena -pero no consume-, la comprobación no se hace dentro de un bucle
 * while porque el bedel (que usa este método) es el único que pondrá útiles
 * en la mesa. Después de comprobar que ya no hay objetos en la mesa, pasa a
 * atender las peticiones: mientras no haya pedidos se bloqueará en el
 * objeto pedidoNoVacio mediante una llamada a wait(); en cuanto hay pedidos
 * obtiene uno de la cola, informa por pantalla de qué utensilio ha sacado y
 * sale del monitor. A continuación trata de entrar en el monitor para
 * notificar que hay disponible un hueco para hacer un pedido, devuelve
 * -con return()- el pedido que ha atendido y vuelve a salir del monitor.
 */
```

```
* @return Cadena que indica el tipo de pedido que ha atendido. Puede ser un
* 'bote' o un 'pincel', aunque no se hace comprobación de lo obtenido
* más que para así indicarlo por pantalla.
*/
public String atenderPedido()
{
    String p;

    synchronized(pedidoNoVacio)
    {
        synchronized(mesaNoLlena) //No atiende un nuevo pedido hasta
        {
            //que no haya espacio en la mesa.
            if(mesa.estaLleno()) //Si la mesa está llena (no es
            {
                //necesario reexaminar con while):
                try {mesaNoLlena.wait();} //1o espera a que haya un hueco
                catch(InterruptedException e) {}

                mesaNoLlena.notify(); //2o desbloquea espera por mesa
            } //Si la mesa tiene algún hueco,
        } //continúa normalmente.
        while(pedido.estaVacio())
        {
            try {pedidoNoVacio.wait();}
            catch(InterruptedException e) {}
        }
        p = pedido.quitarPedido();
        if(p.equals("bote"))
            System.out.println("Bedel: Me han pedido un bote, voy al almacen a
buscarlo");
        else
            System.out.println("Bedel: Me han pedido un pincel, voy al almacen a
buscarlo");
        }
        synchronized(pedidoNoLleno)
        {
            pedidoNoLleno.notify(); //Notifica que se puede hacer un nuevo pedido
            return p;
        }
    }
}

/**
 * Método exportado que produce utensilios en respuesta a los pedidos,
 * depositándolos en una mesa gestionada como cola FIFO.
 *
 * Similar a 'devolverCaballero()', selecciona el tipo de mensaje y el tipo
 * de hilo que tratará de desbloquear en función del tipo de utensilio
 * depositado en la mesa. Si en la mesa se deposita un bote, trata de
 * desbloquear al (o a los, si nPeticones>1) hilo que está esperando en el
 * objeto 'mesaSiBote'; si por el contrario se deposita un pincel, hace lo
 * mismo en el objeto 'mesaSiPincel'. Finalmente sale del monitor (estrategia
```

```
* DC) y trata de volver a entrar, momento en el que tratará de desbloquear
* algún hilo que esté esperando en el objeto 'mesaNoVacía' a que la mesa
* tenga algún utensilio independientemente del tipo.
*
* @param util Cadena que indica el tipo de utensilio que se va a poner en la
* mesa. El utensilio puede ser 'bote' o 'pintura', pero no se
* hace comprobación del mismo más que para distinguir entre los
* mensajes que se sacan por pantalla.
*/
public void ponerMesa(String util)
{
    synchronized (mesaNoLlena)
    {
        while(mesa.estaLleno())
        {
            try {mesaNoLlena.wait();}
            catch(InterruptedException e) {}
        }
        mesa.ponerMesa(util);
        if(mesa.utensilioEs("bote"))
        {
            synchronized (mesaSiBote)
            {
                System.out.println("Bedel: He puesto en la mesa un bote");
                mesaSiBote.notify();
            }
        }
        else
        {
            synchronized(mesaSiPincel)
            {
                System.out.println("Bedel: He puesto en la mesa un pincel");
                mesaSiPincel.notify();
            }
        }
    }
    synchronized (mesaNoVacía)
    {
        mesaNoVacía.notify();
    }
}

/**
 * Método exportado que consume utensilios de tipo 'bote'.
 *
 * Espera a la entrada del monitor a que haya algún objeto en la mesa (si no
 * es así se bloquea dentro de un bucle while a la espera de que sí los haya),
 * comprueba si el objeto es un 'bote' (si no es así sale del monitor
 * bloqueándose mediante wait() dentro de un bucle while a la espera de dicho
 * objeto), en cuanto hay un bote lo extrae de la cola mesa, indicándolo por
```

```
* pantalla y sale del monitor. Vuelve a entrar en el monitor para indicar
* a un hilo que esté esperando en el objeto mesaNoLlena que ya hay un hueco
* en la mesa y finalmente sale del monitor.
*
* @param alumno Identificador numérico del alumno en la clase.
*/
public void quitarBoteMesa(int alumno)
{
    synchronized(mesaNoVacía)
    {
        while(mesa.estaVacio())
        {
            try {mesaNoVacía.wait();} //Se bloquea si no hay nada en la mesa
            catch(InterruptedException e) {}
        }
        synchronized(mesaSiBote)
        {
            while(!mesa.utensilioEs("bote"))
            {
                try {mesaSiBote.wait();} //Se bloquea si no hay un bote en la mesa
                catch(InterruptedException e) {}
            }
            mesa.quitarMesa();
            System.out.println("Alumno " + alumno + ": He conseguido el bote de
pintura");
        }
    }
    synchronized(mesaNoLlena) //Trata de despertar a algún productor
    {
        //bloqueado en espera de que la mesa
        mesaNoLlena.notify(); //no esté llena.
    }
}

/**
* Método exportado que consume utensilios de tipo 'pincel'.
*
* Espera a la entrada del monitor a que haya algún objeto en la mesa (si no
* es así se bloquea dentro de un bucle while a la espera de que sí los haya),
* comprueba si el objeto es un 'pincel' (si no es así sale del monitor
* bloqueándose mediante wait() dentro de un bucle while a la espera de dicho
* objeto), en cuanto hay un pincel lo extrae de la cola mesa, indicándolo por
* pantalla y sale del monitor. Vuelve a entrar en el monitor para indicar
* a un hilo que esté esperando en el objeto mesaNoLlena que ya hay un hueco
* en la mesa y finalmente sale del monitor.
*
* @param alumno Identificador numérico del alumno en la clase.
*/
public void quitarPincelMesa(int alumno)
{
    synchronized(mesaNoVacía)
```

```
{
    while(mesa.estaVacio())
    {
        try {mesaNoVacia.wait();} //Se bloquea si no hay nada en la mesa
        catch(InterruptedException e) {}
    }
    synchronized(mesaSiPincel)
    {
        while(!mesa.utensilioEs("pincel"))
        {
            try {mesaSiPincel.wait();} //Se bloquea si no hay un botepincel en la mesa
            catch(InterruptedException e) {}
        }
        mesa.quitarMesa();
        System.out.println("Alumno " + alumno + ": He conseguido el pincel");
    }
}
synchronized(mesaNoLlena) //Trata de despertar a algún productor
{
    //bloqueado en espera de que la mesa
    mesaNoLlena.notify(); //no esté llena.
}
}

/**
 * Método para escribir por pantalla el título del cuadro que se propone
 * (profesor) o que se va a pintar (alumno).
 *
 * Necesita saber si es el profesor (true/false) o el alumno; en el primer
 * caso prescinde del parámetro alumno y en el segundo necesita el identificador
 * (entero) del mismo. Necesita también el identificador (entero) del cuadro.
 *
 * Empieza obteniendo la cadena del título mediante una llamada al método
 * estático dameCuadro() de la clase RepositorioCuadros; para ello el método
 * hace una conversión interna del identificador a fin de que no se pida una
 * cadena que no figura en el repositorio. A continuación se hace una llamada
 * a println con los parámetros 'alumno' y 'tempTitulo' (si es el alumno) o
 * con el parámetro 'tempTitulo' si es el profesor.
 *
 * @param profesor Indica si es el profesor (true) el hace la llamada o bien un
 * alumno (false).
 * @param alumno Identificador numérico del alumno en la clase.
 * @param numCuadro Identificador numérico del título del cuadro que se propone
 * (en caso de que sea el profesor) o que se va a pintar (en
 * caso de que sea el alumno).
 */
public void escribeCuadro(boolean profesor, int alumno, int numCuadro)
{
    String tempTitulo = new RepositorioCuadros().dameCuadro(numCuadro);
    if(profesor)
        System.out.println("Profesor: He anotado en la pizarra " + tempTitulo);
}
```

```
        else
            System.out.println("Alumno " + alumno + ": Escojo para pintar " + tempTitulo);
    }
}
```

Clase ColaFIFO.

```
/**
 * Cola FIFO implementada sobre un sobre un array de objetos, el manejo
 * del array se hace suponiéndolo circular.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class ColaFIFO
{
    /*Variables de instancia de la cola*/
    private int tam, frente, cola;
    private Object[] almacen;
    private int tamMax;

    /**
     * Constructor de la clase colaFIFO. Genera una cola circular
     * implementada sobre un array de objetos, con politica FIFO para la
     * entrada y salida de la misma.
     * Asigna un tamaño máximo a la cola e inicializa los punteros.
     *
     * @param tamMaximo El tamaño máximo que gestionará la cola.
     */
    public ColaFIFO(int tamMax)
    {
        this.tamMax = tamMax;
        tam = 0;
        frente = 0;
        cola = 0;
        almacen = new Object[tamMax];
    }

    /**
     * Método para poner en la cola un objeto.
     *
     * @param ob Objeto que se quiere poner en la cola.
     */
    public void poner(Object ob)
    {
        almacen[cola] = ob;
        cola = (cola + 1) % tamMax;
        tam = tam + 1;
    }
}
```

```
/**
 * Método para quitar un entero de la cola.
 *
 * @return Devuelve el objeto que está al frente.
 */
public Object quitar()
{
    Object temp = almacen[frente];
    frente = (frente + 1) % tamMax;
    tam = tam - 1;

    return temp;
}

/**
 * Método para consultar el tamaño de la cola.
 *
 * @return Devuelve el número de objetos en la cola.
 */
public int tamanho()
{
    return tam;
}

/**
 * Método para consultar si la cola está vacía.
 *
 * @return Verdadero (true) si la cola está vacía, falso (false) si la
 *        cola tiene algún elemento.
 */
public boolean estaVacio()
{
    return (tam == 0);
}

/**
 * Método para consultar si la cola está llena.
 *
 * @return Verdadero (true) si la cola está llena, falso (false) si en
 *        la cola hay sitio para algún otro elemento.
 */
public boolean estaLleno()
{
    return (tam == tamMax);
}

/**
 * Método para consultar si se puede poner en la cola una cierta
 * cantidad de elementos.
```

```
*
* @param nObjetos Cantidad de elementos que tenemos intención de
*         poner en la cola.
* @return Verdadero (true) si en la cola se puede poner una cantidad
*         nObjetos de elementos, falso (false) si no es posible.
*/
public boolean sePuedePoner(int nObjetos)
{
    return (tam + nObjetos <= tamMax);
}

/**
 * Método para consultar el objeto situado al frente de la cola.
 *
 * @return Devuelve el objeto situado al frente.
 */
public Object consultarFrente()
{
    return almacen[frente];
}

/**
 * Método para avanzar el frente de la cola en una unidad.
 */
public void avanzarFrente()
{
    frente = (frente + 1) % tamMax;
}

/**
 * Método para disminuir el tamaño efectivo de la cola en una unidad.
 */
public void disminuirTamanho()
{
    tam = tam - 1;
}
}
```

Clase Pizarra.

```
/**
 * Particulariza el uso de la colaFIFO a la idiosincrasia de la pizarra.
 *
 * Añade los métodos ponerTitulo, quitarTitulo, tieneTitulos(),
 * reservaTitulo() y borraTitulo().
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
```

```
public class Pizarra extends ColaFIFO
{
    /**
     * Constructor de la clase Pizarra.
     *
     * @param maxTitulos El máximo número de títulos simultáneamente
     *                  en la pizarra.
     */
    public Pizarra(int maxTitulos)
    {
        //Usa el constructor de la clase padre.
        super(maxTitulos);
    }

    /**
     * Pone un título en la cola, representado por un número entero.
     *
     * @param numTitulo Número que representa al título insertado.
     */
    public void ponerTitulo(int numTitulo)
    {
        poner(numTitulo);
    }

    /**
     * Quita un título de la cola, representado por un número entero.
     *
     * @return El número de título eliminado.
     */
    public int quitarTitulo()
    {
        return (Integer) quitar();
    }

    /**
     * Comprueba si existe algún escrito en la pizarra.
     *
     * @return Verdadero (true) si hay títulos, falso (false) si la pizarra
     *         está vacía.
     */
    public boolean tieneTitulos()
    {
        return !estaVacio();
    }

    /**
     * Reserva un título escrito en la pizarra, de forma que un nuevo
     * consumidor acceda al siguiente título, pero no actualiza el tamaño.
     * De esta forma es imposible escribir en el hueco consultado porque
     * todavía no se ha hecho efectivo.
     */
}
```

```
*
* @return El entero que representa al título reservado.
*/
public int reservarTitulo()
{
    int temp = (Integer) consultarFrente();
    avanzarFrente();

    return temp;
}

/**
 * Hace efectivo el hueco dejado al disminuir el tamaño ocupado por
 * los títulos en la pizarra.
 */
public void borrarTitulo()
{
    disminuirTamanho();
}
}
```

Clase Caballete.

```
/**
 * Particulariza el uso de la colaFIFO a la idiosincrasia del
 * almacén de caballetes.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class Caballete extends ColaFIFO
{
    /**
     * Constructor de la clase Caballete.
     *
     * @param maxCaballetes El máximo número de caballetes que se
     * pueden guardar en el almacén.
     */
    public Caballete(int maxCaballetes)
    {
        //Usa el constructor de la clase padre.
        super(maxCaballetes);

        //Inicialmente están todos los caballetes disponibles.
        for(int i=0; i<maxCaballetes; i++)
            ponerCaballete();
    }

    /**
```

```
* Método para devolver un caballete al almacén.
*/
public void ponerCaballete()
{
    poner(new Object());
}

/**
 * Método para obtener un caballete del almacén.
 */
public void quitarCaballete()
{
    quitar();
}
}
```

Clase Peticion.

```
/**
 * Particulariza el uso de la colaFIFO a la idiosincrasia de los pedidos.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class Peticion extends ColaFIFO
{
    /**
     * Constructor de la clase Peticion.
     *
     * @param maxPedidos El máximo número de pedidos simultáneos que puede
     *                  admitir el bedel.
     */
    public Peticion(int maxPedidos)
    {
        //Usa el constructor de la clase padre.
        super(maxPedidos);
    }

    /**
     * Pone un pedido en el tablón del bedel.
     *
     * @param String Útil de pintura que se pide, puede ser un bote
     *             o un pincel.
     */
    public void ponerPedido(String ped)
    {
        poner(ped);
    }
}
```

```
/**
 * Atiende a un pedido de los que están esperando en el tablón.
 *
 * @return El utensilio de pintura que se le ha pedido.
 */
public String quitarPedido()
{
    return (String) quitar();
}
}
```

Clase Mesa.

```
/**
 * Particulariza el uso de la colaFIFO a la idiosincrasia de la mesa.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class Mesa extends ColaFIFO
{
    /**
     * Constructor de la clase Mesa.
     *
     * @param maxMesa El máximo número de pedidos simultáneos que se
     *                pueden poner en la mesa.
     */
    public Mesa(int maxMesa)
    {
        //Usa el constructor de la clase padre.
        super(maxMesa);
    }

    /**
     * Pone un pedido en la mesa.
     *
     * @param String Útil de pintura que se sirve, puede ser un bote
     *                o un pincel.
     */
    public void ponerMesa(String ped)
    {
        poner(ped);
    }

    /**
     * Obtiene el primer útil de los que están depositados en la mesa
     *
     * @return El utensilio de pintura que se ha depositado.
     */
}
```

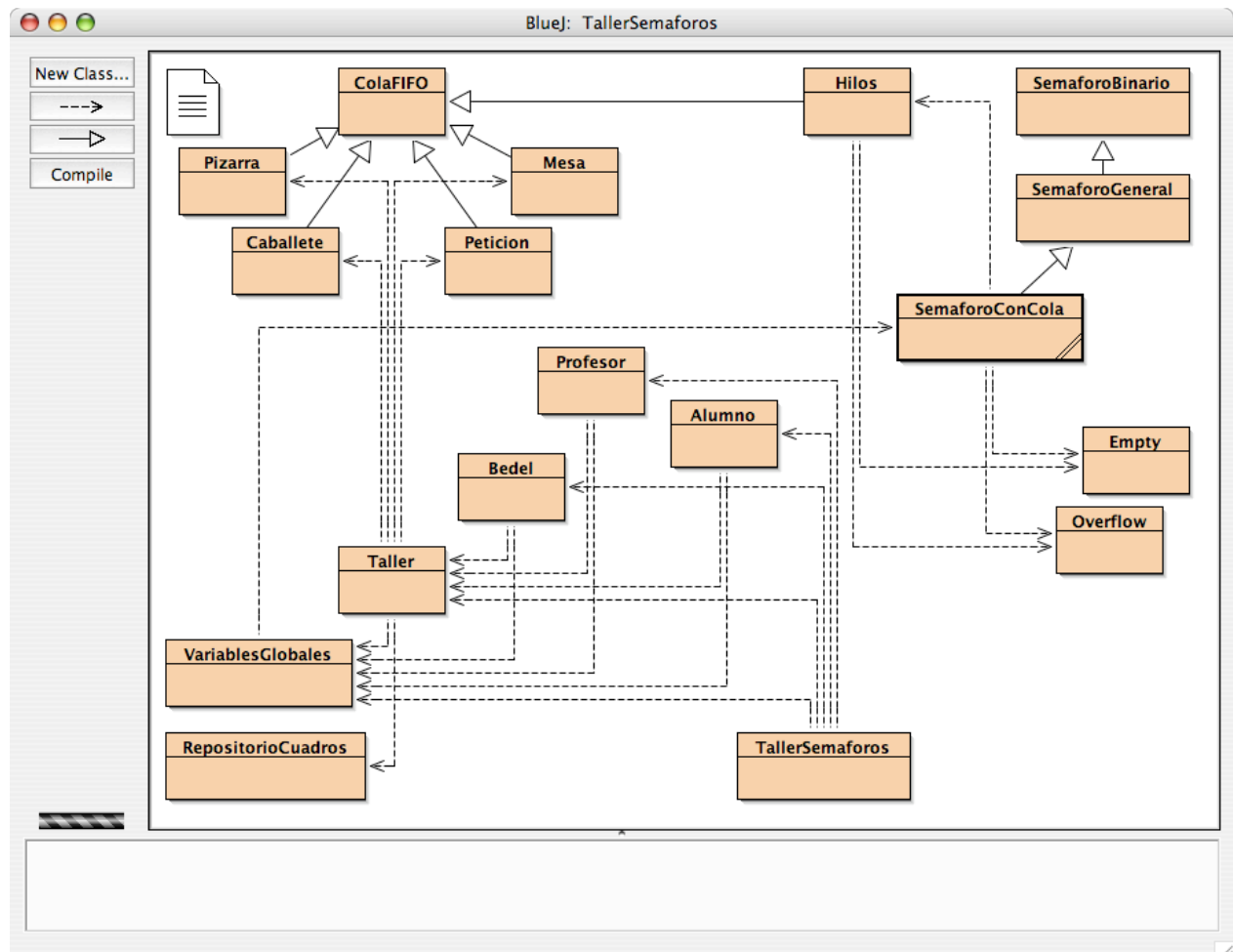
```
public String quitarMesa()
{
    return (String) quitar();
}

/**
 * Permite comprobar si lo que hay en primer lugar en la mesa es un bote
 * o un pincel sin necesidad de retirarlo.
 *
 * @param util La cadena que representa el utensilio deseado.
 * @return Verdadero (true) si el elemento que está al frente de la cola
 *         coincide con 'util', falso (false) en caso contrario.
 */
public boolean utensilioEs(String util)
{
    String temp = (String) consultarFrente();
    return temp.equals(util);
}
}
```

3.5.- Taller de pintura con Semáforos en Java.

Para realizar esta parte de la práctica hemos utilizado la misma estructura empleada en PascalFC, pero hemos reaprovechado la mayor cantidad posible de código de la parte de monitores en Java. Así los métodos de las clases Profesor, Alumno y Bedel llaman a otros métodos de la clase Taller, de forma que Taller se convierte en algo parecido a una interfase que proporciona métodos para el acceso a las colas de los recursos. Esto que complica algo más el diseño nos ha permitido reaprovechar una buena parte del código relativo a los monitores, con pequeñas modificaciones en algunas clases (en particular la clase Taller y algunos métodos de las clases Profesor, Alumno y Bedel).

Implementamos los semáforos en Java siguiendo las recomendaciones del libro de Palma et al. En realidad hemos copiado tal cual el código del libro. Pero al vernos en la necesidad de añadir las colas de bloqueo de los semáforos, creamos una nueva clase que hereda de SemaforoGeneral y que sobrescribe el método WAIT() heredado de SemaforoBinario. En nuestra clase hemos implementado la cola en la que se bloquean los hilos que hacen una llamada a WAIT() como una cola FIFO, para ello hemos aprovechado el trabajo realizado en el anterior proyecto (monitores) heredando de la clase ColaFIFO. La clase SemaforoConCola es una elaboración propia partiendo de las ideas y recomendaciones del libro de la asignatura.



La clase SemaforoConCola hace uso de excepciones para detectar los intentos de introducir un nuevo hilo cuando la cola está llena o extraer uno cuando está vacía. La clase ColaFIFO no implementa control sobre los límites de la cola, es responsabilidad de las clases que las utilizan o que heredan de ella el hacerlo. Para las excepciones mencionadas,

SemaforoConCola lanza Overflow o Empty según el caso, que imprimen un mensaje de error y finalizan la ejecución.

En la figura anterior se pueden observar las relaciones entre clases del proyecto (captura de la ventana principal del proyecto en el entorno BlueJ). La estructura básica es la misma que la del proyecto con monitores pero añadiéndole seis clases más relacionadas con los semáforos (las tres de semáforos propiamente dichas, la cola para el bloqueo de hilos y las dos clases que extienden Exception). Igual que en el anterior proyecto, la clase principal (donde se encuentra el método main()) es TallerSemaforos y se encarga de inicializar Profesor, Bedel, Alumno, Taller y VariablesGlobales, así como poner en marcha los hilos asociados con las tres primeras. La clase VariablesGlobales contiene las constantes que utilizan las otras clases así como la declaración de los semáforos que van a ser usadas por Profesor, Alumno y Bedel.

A continuación listamos las clases que componen el proyecto. Para no extendernos inútilmente no listamos algunas por ser exactamente iguales a las usadas en el anterior proyecto (monitores). Las clases que si listamos son: TallerSemaforos, VariablesGlobales, Profesor, Alumno, Bedel, Taller, SemaforoConCola, SemaforoGeneral, SemaforoBinario, Hilos, Overflow y Empty.

Clase TallerSemaforos

```
/**
 * Clase principal que contiene el método main.
 * Se encarga de crear los objetos taller, variables globales,
 * profesor, alumno y bedel, pasándoles a estos tres últimos tanto las
 * variables globales (lo cual incluye a los semáforos) como el objeto taller.
 * Se encarga también de transformar a profesor, bedel y alumno en hilos e
 * iniciar su ejecución concurrente.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class TallerSemaforos
{
    public static void main(String[] args)
    {
        VariablesGlobales vg = new VariablesGlobales();
        Taller taller = new Taller(vg);

        //Crea los objetos profesor, bedel y alumnos[]
        Profesor profesor = new Profesor(taller, vg);
        Bedel bedel = new Bedel(taller, vg);
        Alumno[] alumnos = new Alumno[vg.nAlumnos];

        //Inicializa cada uno de los alumnos[]
        //Los alumnos se numeran de 1 a nAlumnos, aunque internamente
        //el array tenga índices entre 0 y nAlumnos-1.
        for(int i=0; i<vg.nAlumnos; i++)
            alumnos[i] = new Alumno(i+1, taller, vg);

        //Crea los hilos asociados a profesor, bedel y alumnos[]
        Thread hiloProfesor = new Thread(profesor);
        Thread hiloBedel = new Thread(bedel);
        Thread[] hilosAlumnos = new Thread[vg.nAlumnos];

        //Crea cada uno de los hilos asociados a alumnos[]
        for(int i=0; i<vg.nAlumnos; i++)
            hilosAlumnos[i] = new Thread(alumnos[i]);

        //Inicia la ejecución de los hilos
        hiloProfesor.start();
        hiloBedel.start();

        for(int i=0; i<vg.nAlumnos; i++)
            hilosAlumnos[i].start();
    }
}
```

Clase VariablesGlobales

```
/**
 * Clase para guardar las constantes globales y los semáforos que
 * van a compartir las otras clases. Se ha puesto para clarificar
 * la clase principal.
 *
 * @author Víctor M. Álvarez Pérez
 * @date Junio de 2007
 */
public class VariablesGlobales
{
    public final int nMaxCuadros = 50; //No máximo de cuadros que puede poner el
profesor
    public final int nCuadros = 5;    //No máximo de cuadros que caben en la pizarra
    public final int nAlumnos = 15;   //No de alumnos en la clase
    public final int nPeticones = 1;   //No de peticiones que pueden ser atendidas o puestas
en la mesa
    public final int nCaballetes = 5;  //No de caballetes que caben el almacén
    public final int nRecursos = 3;    //No de recursos que se pueden pedir en cualquier
orden

    private int nMaxHilos;

    public SemaforoConCola mutex;
    public SemaforoConCola cuadrosVacio;
    public SemaforoConCola cuadrosLleno;
    public SemaforoConCola peticionVacio;
    public SemaforoConCola peticionLleno;
    public SemaforoConCola mesaVacía;
    public SemaforoConCola mesaLlena;
    public SemaforoConCola caballeteVacio;
    public SemaforoConCola caballeteLleno;

    public VariablesGlobales()
    {
        nMaxHilos = nAlumnos + 1 + 1; //nAlumnos, un profesor y un bedel como
máximo
        mutex = new SemaforoConCola(1, nMaxHilos); //Exclusión mutua, inicializa a
abierto
        cuadrosVacio = new SemaforoConCola(0, nMaxHilos); //Pizarra con huecos,
inicializa sin huecos
        cuadrosLleno = new SemaforoConCola(0, nMaxHilos); //Pizarra con títulos,
inicializa sin títulos
        peticionVacio = new SemaforoConCola(nPeticones, nMaxHilos); //Peticiones
Bedel, una disponible
        peticionLleno = new SemaforoConCola(0, nMaxHilos); //Peticions Bedel, no hay
peticiones
        mesaVacía = new SemaforoConCola(nPeticones, nMaxHilos); //Espacio en la
mesa, uno disponible
    }
}
```

```
        mesaLlena = new SemaforoConCola(0, nMaxHilos);    //Espacio en la mesa, no hay
nada
        caballeteVacio = new SemaforoConCola(nCaballetes, nMaxHilos); //Caballetes
disponibles, nCab disponibles
        caballeteLleno = new SemaforoConCola(0, nMaxHilos);    //Caballetes
disponibles, ninguno ocupado
    }
}
```

Clase Profesor

```
/**
 * Clase que modela el comportamiento del profesor. Implementa Runnable
 * para poder ejecutarlo como un hilo.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
import java.util.Random;

public class Profesor implements Runnable
{
    private Random generaAzar;    //Para escoger aleatoriamente un título

    private Taller taller;        //Para no acceder a todas las colas por separado
    private VariablesGlobales vg; //Para incluir las mismas variables en cada clase

    /**
     * Constructor de la clase Profesor.
     *
     * @param taller Apunta a los métodos que hacen de puente entre los
     *             hilos Profesor, Bedel y Alumno y las colas FIFO pizarra,
     *             petición, mesa y caballete.
     * @param vg Apunta a un objeto VariablesGlobales donde están las
     *           constantes compartidas por todas las clases y los semáforos
     *           que emplearán los hilos Profesor, Bedel y Alumno.
     */
    public Profesor(Taller taller, VariablesGlobales vg)
    {
        this.taller = taller;
        this.vg = vg;
        generaAzar = new Random();
    }

    /**
     * Método de donde parte el hilo cuando se inicia con start().
     * En este método no se enmascaran las llamadas a los métodos de taller
     * puesto que el código resultante es bastante legible y no hace necesario

```

```
* el esfuerzo de crear nuevos métodos privados.
* Saca por pantalla las indicaciones del principio de la clase, genera
* una cantidad nCuadros de títulos y los escribe en la pizarra (en
* exclusión mutua, no es necesario que espere ningún otro semáforo
* puesto que es el profesor quien da la salida para los otros hilos). A
* continuación indica que ya hay títulos en la pizarra haciendo 'nCuadros'
* signal sobre el semáforo cuadrosLleno (donde están esperando los
* alumnos) y entra en un bucle infinito que hace lo siguiente: se demora
* un poco en volver a la actividad (estado 'dormido'), espera a que haya
* un hueco en la pizarra para poner otro título, espera a obtener la
* exclusividad sobre la pizarra y a continuación genera un nuevo título
* y lo escribe. Finalmente libera la exclusión mutua e indica que hay
* un nuevo título en la pizarra, volviendo a iniciar el bucle.
*/
public void run()
{
    System.out.println("Profesor: Buenos dias, os voy a anotar en la pizarra el trabajo de
hoy");
    int[] numeroCuadro = new int[vg.nCuadros];

    for(int i=0; i<vg.nCuadros; i++)
        numeroCuadro[i] = generaAzar.nextInt(vg.nMaxCuadros);

    vg.mutex.WAIT();
    taller.escribirPizarra(numeroCuadro, vg.nCuadros);
    vg.mutex.SIGNAL();

    //Indica que ha puesto los nCuadros en la pizarra
    for(int i=0; i<vg.nCuadros; i++)
        vg.cuadrosLleno.SIGNAL();

    while(true)
    {
        hacerTiempo(400);    //Se demora un poco en volver a entrar

        vg.cuadrosVacio.WAIT();
        vg.mutex.WAIT();
        numeroCuadro[0] = generaAzar.nextInt(vg.nMaxCuadros);
        taller.escribirPizarra(numeroCuadro, 1);
        vg.mutex.SIGNAL();
        vg.cuadrosLleno.SIGNAL();
    }
}

/**
 * Método privado para introducir intervalos de espera entre las acciones
 * del profesor; sirve para enmascarar el bloque try-catch y mejorar la
 * legibilidad.
 * Hace una llamada a sleep() (dentro de try-catch) para que el hilo pase
 * al estado 'dormido' y el intervalo de tiempo no sea mediante espera
 */
```

```
* activa.  
*  
* @param espera Tiempo en ms que debe dormir el hilo.  
*/  
private void hacerTiempo(int espera)  
{  
    try {Thread.sleep(espera);}  
    catch (InterruptedException e) {}  
}  
}
```

Clase Alumno

```
/**  
 * Clase que modela el comportamiento del alumno. Implementa Runnable  
 * para poder ejecutarlo como un hilo.  
 *  
 * @author Víctor M. Álvarez Pérez  
 * @version Junio de 2007  
 */  
import java.util.Random;  
  
public class Alumno implements Runnable  
{  
    private int alumno;          //Identificador del alumno en la clase  
    private Taller taller;       //Para no acceder a todas las colas por separado  
    private VariablesGlobales vg; //Para no incluir las mismas variables en cada clase  
    private Random generaAzar;   //Para escoger aleatoriamente bote o pincel  
  
    /**  
     * Constructor de la clase Alumno.  
     *  
     * @param alumno Identificador del alumno en la clase  
     * @param taller Apunta a los métodos que hacen de puente entre los  
     *             hilos Profesor, Bedel y Alumno y las colas FIFO pizarra,  
     *             petición, mesa y caballete.  
     * @param vg Apunta a un objeto VariablesGlobales donde están las  
     *           constantes compartidas por todas las clases y los semáforos  
     *           que emplearán los hilos Profesor, Bedel y Alumno.  
     */  
    public Alumno(int alumno, Taller taller, VariablesGlobales vg)  
    {  
        this.alumno = alumno;  
        this.taller = taller;  
        this.vg = vg;  
        generaAzar = new Random();  
    }  
  
    /**
```

```
* Método de donde parte el hilo cuando se inicia con start().
* Consulta la pizarra para no agobiar al bedel mientras no haya ningún
* título en la misma, llama al método privado actividadAlumno() (que
* codifica las acciones del alumno) con el parámetro primeraVez a true
* y a continuación entra en un bucle infinito donde hace la misma
* llamada a actividadAlumno() pero con primeraVez a false (para que
* indique que vuelve al trabajo antes de empezar).
*/
public void run()
{
    consultarPizarra();    //Espera a que haya títulos en la pizarra

    while(true)
        actividadAlumno();
}

/**
* Método privado que codifica las acciones del alumno.
* Se hacen llamadas a funciones en las que están los semáforos, para
* mejorar la legibilidad de este método; se introducen retardos (mediante
* llamadas a sleep()) para simular lo mejor posible la actividad en un
* aula.
* Primero se escoje (aleatoriamente) qué tipo de útil de entre botes y
* pinceles vamos a pedir por primera vez, se piden y obtienen ambos
* (notificándolo por pantalla) y a continuación se piden el caballete
* y el título (por este orden). Cuando ya se tiene todo, el alumno
* desaparece de la escena durante un rato mientras está pintando. Cuando
* acaba lo indica por pantalla, devuelve el caballete y avisa al profesor.
* A continuación se va al pasillo un rato y desaparece de la escena
* mientras descansa.
* Se empieza por pedir bote o pincel en primer lugar ya que son
* recursos ilimitados y no sería justo que un alumno acaparase primero
* un caballete o un título (recursos limitados) sin tener los útiles de
* pintura, mientras otros alumnos con pincel y bote tuviesen que esperar
* para poder empezar a pintar (en el Bedel se crea una buena cola al
* principio). También parece lógico hacer acopio del caballete antes que
* del título, aunque también se podría implementar una estrategia en la
* que se escogiese aleatoriamente entre los dos.
* No se devuelve pincel ni bote puesto que se consideran inservible el
* primero y agotado el segundo; son recursos ilimitados. La devolución
* del caballete y el aviso al profesor se efectúan en el mismo orden en
* que se obtuvo caballete y título para facilitar la concurrencia.
* Los mensajes que emite el alumno se imprimen en exclusión mutua para
* evitar la posible (aunque improbable en java) interrupción por parte
* de otros hilos que también quieran acceder a la pantalla. Esto puede
* ocurrir en las llamadas a println que incluyen variables. De esta forma
* se trata implícitamente a la pantalla como un recurso limitado al que
* se accede mediante el semáforo mutex. No es necesario otro semáforo
* binario para controlar el acceso a la pantalla, aunque se podría
* incluir por completitud (de tratamiento de los recursos).
```

```
*
* @param primeraVez Parámetro para controlar si es la primera vez que se
*     ejecuta el método, se pasa directamente al método
*     pedirBedel.
*/
private void actividadAlumno()
{
    int[] ordenPeticion = generarOrdenPeticion(vg.nRecursos); //Produce 0, 1 ó 2

    for(int i=0; i<vg.nRecursos; i++)
    {
        switch(ordenPeticion[i])
        {
            case 0: //Si es un bote
                pedirBote();
                hacerTiempo(100); //Se distrae un momento
                obtenerBote();
                hacerTiempo(200); //Tarda en depositar el bote en su lugar de trabajo
                break;
            case 1: //Si es un pincel
                pedirPincel();
                hacerTiempo(100); //Se distrae un momento
                obtenerPincel();
                hacerTiempo(200); //Tarda en depositar el pincel en su lugar de trabajo
                break;
            case 2: //Si es un caballete
                hacerTiempo(400); //Se demora un poco en ir al almacén de caballetes
                obtenerCaballete();
                hacerTiempo(700); //Tarda en volver cargado con el caballete
                break;
        }
    }

    obtenerCuadro();

    hacerTiempo(1500); //Le lleva cierto tiempo pintar el cuadro.

    hacerTiempo(300); //Va hasta el almacén para devolver el caballete.
    devolverCaballete();
    hacerTiempo(300); //Tarda en volver del almacén.

    avisarProfesor();

    hacerTiempo(700); //Descansa un buen rato.

    volverAlTrabajo();
}

/**
 * Método privado para generar un orden aleatorio entre los recursos que no
```

```
* tienen por qué pedirse en un orden fijo.
*
* Cada recurso se numera de 0 a nRecursos-1 y se almacena el identificador
* del mismo en un array. A continuación se comprueba que el identificador
* generado no está ya en el array; si es así se vuelve a generar un número
* aleatorio y se vuelve a comprobar hasta que no esté repetido. Se recorre
* todo el array haciendo lo mismo y finalmente se devuelve dicho array.
*
* @param nRecursos El número de recursos que se van a solicitar aleatoriamente.
* @return Un array conteniendo los identificadores (enteros) de recurso que
*         se van solicitar.
*/
private int[] generarOrdenPetición(int nRecursos)
{
    boolean seRepite;
    int[] orden = new int[nRecursos];
    for(int i=0; i<nRecursos; i++)
    {
        do
        {
            seRepite = false;
            orden[i] = generaAzar.nextInt(nRecursos); //Genera int entre 0 y nRecursos-1
            for(int j=0; j<i; j++) //Comprueba que no se repite
            {
                if(orden[j] == orden[i])
                    seRepite = true;
            }
        } while (seRepite);
    }
    return orden;
}

/**
 * Método privado para esperar a que haya algún título en la pizarra
 * antes de que todos los alumnos empiecen a agobiar al bedel. Sirve
 * para sincronizar el inicio de las acciones de los alumnos con el
 * profesor: así no se ve interrumpido por un griterío mientras da
 * las instrucciones previas al comienzo de la clase.
 */
private void consultarPizarra()
{
    vg.cuadrosLleno.WAIT(); //Espera que haya al menos un título,
    vg.cuadrosLleno.SIGNAL(); //deja todo como estaba.
}

/**
 * Método privado para solicitar al bedel un utensilio de pintura (bote
 * o pincel), sirve para enmascarar los semáforos en actividadAlumno()
 * y mejorar la legibilidad.
 * Hace una llamada a taller.hacerPedido() con el tipo de pedido, el
```

```
* identificador del alumno y un indicador de si es o no la primera vez.
*
* @param pedido Tipo de utensilio que se le pide al bedel (bote o pincel).
* @param primeraVez Indicador de si ya se ha pedido algún otro utensilio
*     antes, hace que se añada un mensaje de vuelta al
*     trabajo si no es la primera vez.
*/
private void pedirBedel(String pedido, boolean primeraVez)
{
    vg.peticionVacio.WAIT();
    vg.mutex.WAIT();
    taller.hacerPedido(pedido, alumno, primeraVez);
    vg.mutex.SIGNAL();
    vg.peticionLleno.SIGNAL();
}

/**
 * Método privado que enmascara al procedimiento pedirBedel().
 *
 * Hace una llamada a pedirBedel() con el valor "bote" en pedido y el
 * valor 'true' en primeraVez (para que no salga por pantalla el mensaje
 * de volver al trabajo).
 */
private void pedirBote()
{
    pedirBedel("bote", true);
}

/**
 * Método privado que enmascara al procedimiento pedirBedel().
 *
 * Hace una llamada a pedirBedel() con el valor "pincel" en pedido y el
 * valor 'true' en primeraVez (para que no salga por pantalla el mensaje
 * de volver al trabajo).
 */
private void pedirPincel()
{
    pedirBedel("pincel", true);
}

/**
 * Método privado para obtener un bote de la mesa, sirve para enmascarar
 * los semáforos de actividadAlumno() y mejorar la legibilidad.
 * Mientras no se consiga el bote hace lo siguiente: espera a que la mesa
 * tenga algún objeto y pueda acceder a la misma, espera a la exclusión
 * mutua y trata de quitar un bote de la misma (llamada a quitarBoteMesa()).
 * Si tiene éxito (quitarBoteMesa() devuelve true) actualiza el indicador
 * de éxito, libera la exclusión mutua e indica que se puede poner otro
 * objeto en la mesa. Si no tiene éxito (no retira nada y quitarBoteMesa()
 * devuelve false) libera la exclusión mutua e indica que la mesa todavía
 * tiene un objeto.
```

```
*/
private void obtenerBote()
{
    boolean okPetición = false;
    do
    {
        vg.mesaLlena.WAIT();
        vg.mutex.WAIT();
        if(taller.quitarBoteMesa(alumno))
        {
            okPetición = true;
            vg.mutex.SIGNAL();
            vg.mesaVacía.SIGNAL();
        }
        else
        {
            vg.mutex.SIGNAL();
            vg.mesaLlena.SIGNAL();
        }
    } while(!okPetición);
}

/**
 * Método privado para obtener un pincel de la mesa, sirve para enmascarar
 * los semáforos de actividadAlumno() y mejorar la legibilidad.
 * Mientras no se consiga el pincel hace lo siguiente: espera a que la mesa
 * tenga algún objeto y pueda acceder a la misma, espera a la exclusión
 * mutua y trata de quitar un pincel de la misma (llamada a
 * quitarPincelMesa()). Si tiene éxito (quitarPincelMesa() devuelve true)
 * actualiza el indicador de éxito, libera la exclusión mutua e indica que
 * se puede poner otro objeto en la mesa. Si no tiene éxito (no retira
 * nada y quitarPincelMesa() devuelve false) libera la exclusión mutua e
 * indica que la mesa todavía tiene un objeto.
 */
private void obtenerPincel()
{
    boolean okPetición = false;
    do
    {
        vg.mesaLlena.WAIT();
        vg.mutex.WAIT();
        if(taller.quitarPincelMesa(alumno))
        {
            okPetición = true;
            vg.mutex.SIGNAL();
            vg.mesaVacía.SIGNAL();
        }
        else
        {
            vg.mutex.SIGNAL();

```

```
        vg.mesaLlena.SIGNAL();
    }
    } while(!okPeticion);
}

/**
 * Método privado para obtener un caballete del armario donde se guardan;
 * sirve para enmascarar los semáforos de actividadAlumno() y mejorar la
 * legibilidad.
 * Espera a que se le indique que hay un caballete disponible, a
 * continuación obtiene la exclusión mutua y solicita un caballete mediante
 * una llamada a pedirCaballete. Cuando por fin lo obtiene, libera la
 * exclusión mutua e indica que hay un caballete menos en el armario.
 */
private void obtenerCaballete()
{
    vg.caballeteVacio.WAIT();
    vg.mutex.WAIT();
    taller.pedirCaballete(alumno);
    vg.mutex.SIGNAL();
    vg.caballeteLleno.SIGNAL();
}

/**
 * Método privado para obtener un título de la pizarra donde están escritos;
 * sirve para enmascarar los semáforos de actividadAlumno() y mejorar la
 * legibilidad.
 * Espera a que se le indique que hay un título disponible (si lo hay no se
 * bloquea en el semáforo), espera por la exclusión mutua (si es necesario
 * se bloquea en mutex) y reserva un título de la pizarra llamando al
 * método leerPizarra() (no lo borra puesto que eso haría que inmediatamente
 * el profesor -que está esperando por un hueco en la pizarra- escribiese
 * un nuevo título). Cuando tiene el título reservado libera la exclusión
 * mutua pero no hace signal sobre cuadrosVacio puesto que lo hará en el
 * momento que le indique al profesor que ha acabado.
 */
private void obtenerCuadro()
{
    vg.cuadrosLleno.WAIT();
    vg.mutex.WAIT();
    taller.leerPizarra(alumno);
    vg.mutex.SIGNAL();
    //No hace cuadrosVacio.signal() puesto que enseguida
    //se escribiría otro en su lugar.
    //Espera a borrarlo al momento en que haya finalizado
    //el cuadro y se lo vaya a notificar al profesor
}

/**
 * Método privado para devolver un caballete al armario donde se guardan;
```

```
* sirve para enmascarar los semáforos de actividadAlumno() y mejorar la
* legibilidad.
* Espera a que se le indique que hay un sitio disponible en el armario
* (realmente no sería necesario, puesto que si se quiere devolver es
* porque ya se consumió antes y por tanto hay un sitio libre; se utiliza
* este semáforo por completitud y para mantener una buena comprensión del
* código), espera por la exclusión mutua y devuelve el caballete. Cuando
* ya está devuelto libera la exclusión mutua e indica que hay un caballete
* disponible para ser usado.
*/
private void devolverCaballete()
{
    vg.caballeteLleno.WAIT();
    vg.mutex.WAIT();
    taller.devolverCaballete(alumno);
    vg.mutex.SIGNAL();
    vg.caballeteVacio.SIGNAL();
}

/**
 * Método privado para avisar al profesor de que se ha acabado de pintar
 * un cuadro y puede escribir un nuevo título en la pizarra; sirve para
 * enmascarar los semáforos de actividadAlumno() y mejorar la legibilidad.
 * Espera a obtener la exclusión mutua y a continuación hace una llamada a
 * borrarPizarra() -que hace efectivo el borrado del título-. Cuando se ha
 * borrado el título, libera la exclusión mutua y avisa al profesor de que
 * hay un hueco en la pizarra para escribir un nuevo título (el profesor
 * está bloqueado en el semáforo cuadrosVacio).
 */
private void avisarProfesor()
{
    vg.mutex.WAIT();
    taller.borrarPizarra(alumno);
    vg.mutex.SIGNAL();
    vg.cuadrosVacio.SIGNAL();
}

/**
 * Método privado para sacar por pantalla el mensaje de vuelta al trabajo.
 *
 * Aunque en Java no es necesario, se pone println() entre wait() y signal()
 * para asegurar el acceso en exclusiva a la pantalla.
 */
private void volverAlTrabajo()
{
    vg.mutex.WAIT();
    System.out.println("Alumno " + alumno + ": Vuelvo al trabajo");
    vg.mutex.SIGNAL();
}
```

```
/**
 * Método privado para introducir intervalos de espera entre las acciones
 * del alumno; sirve para enmascarar el bloque try-catch y mejorar la
 * legibilidad.
 * Hace una llamada a sleep() (dentro de try-catch) para que el hilo pase
 * al estado 'dormido' y el intervalo de tiempo no sea mediante espera
 * activa.
 *
 * @param espera Tiempo en ms que debe dormir el hilo.
 */
private void hacerTiempo(int espera)
{
    try {Thread.sleep(espera);}
    catch (InterruptedException w) {}
}
}
```

Clase Bedel

```
/**
 * Clase que modela el comportamiento del bedel. Implementa Runnable
 * para poder ejecutarlo como un hilo.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class Bedel implements Runnable
{
    private Taller taller;      //Para no acceder a todas las colas por separado
    private VariablesGlobales vg; //Para no incluir las mismas variables en cada clase

    /**
     * Constructor de la clase Bedel.
     *
     * @param taller Apunta a los métodos que hacen de puente entre los
     *             hilos Profesor, Bedel y Alumno y las colas FIFO pizarra,
     *             petición, mesa y caballete.
     * @param vg Apunta a un objeto VariablesGlobales donde están las
     *           constantes compartidas por todas las clases y los semáforos
     *           que emplearán los hilos Profesor, Bedel y Alumno.
     */
    public Bedel(Taller taller, VariablesGlobales vg)
    {
        this.taller = taller;
        this.vg = vg;
    }

    /**
     * Método de donde parte el hilo cuando se inicia con start().
     */
}
```

```
* En este método no se enmascaran las llamadas a los métodos de taller
* puesto que el código resultante es bastante legible y no hace necesario
* el esfuerzo de crear nuevos métodos privados.
* Al iniciarse el método entra en un bucle infinito en el que en primer
* lugar espera a que haya una nueva petición, a continuación espera por
* la exclusión mutua y obtiene el pedido que está en primer lugar el la
* cola, actualiza la variable espera en función de si ha obtenido un bote
* o un pincel y finalmente libera la exclusión mutua e indica que se puede
* hacer otro pedido.
* A continuación entra en el estado 'dormido' durante la cantidad de ms
* especificada en 'espera', simulando la ida y vuelta al almacén. Cuando
* está de vuelta espera a que no haya ningún objeto en la mesa y obtener
* la exclusión mutua, pone en la mesa el útil demandado y finalmente
* libera la exclusión mutua e indica que hay un objeto en la mesa.
* A continuación entra en el estado 'dormido' un rato, simulando que
* todavía no ha consultado el tablón de pedidos y a continuación vuelve
* a iniciar el bucle.
*/
public void run()
{
    String utensilio = new String();
    int espera;
    while(true)
    {
        vg.peticionLleno.WAIT();
        vg.mutex.WAIT();
        utensilio = taller.atenderPedido();
        if(utensilio.equals("bote"))
            espera = 300; //Tarda más en volver con el bote, pesa más.
        else
            espera = 200; //Tarda menos en volver con el pincel, pesa menos.
        vg.mutex.SIGNAL();
        vg.peticionVacio.SIGNAL();

        hacerTiempo(espera); //Mientras va y vuelve al almacén.

        vg.mesaVacia.WAIT();
        vg.mutex.WAIT();
        taller.ponerMesa(utensilio);
        vg.mutex.SIGNAL();
        vg.mesaLlena.SIGNAL();

        hacerTiempo(100); //Tarda en darse la vuelta para consultar
    } //el tablón de pedidos.
}

/**
* Método privado para introducir intervalos de espera entre las acciones
* del bedel; sirve para enmascarar el bloque try-catch y mejorar la
* legibilidad.
```

```
* Hace una llamada a sleep() (dentro de try-catch) para que el hilo pase
* al estado 'dormido' y el intervalo de tiempo no sea mediante espera
* activa.
*
* @param espera Tiempo en ms que debe dormir el hilo.
*/
private void hacerTiempo(int espera)
{
    try {Thread.sleep(espera);}
    catch (InterruptedException e) {}
}
}
```

Clase Taller

```
/**
 * Clase que sirve para enmascarar las llamadas a las funciones de las
 * distintas colas, a fin de mejorar la legibilidad del código de la
 * clase Alumno.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class Taller
{
    //Almacenes de datos
    private Pizarra pizarra;
    private Caballete caballetes;
    private Peticion pedido;
    private Mesa mesa;

    /**
     * Constructor de la clase Taller.
     * Obtiene constantes de la clase VariablesGlobales e inicia las
     * colas pizarra, caballetes, pedido y mesa (usando las constantes
     * obtenidas antes).
     *
     * @param vg objeto que contiene las constantes globales y los semáforos
     * comunes a profesor, bedel y alumno.
     */
    public Taller(VariablesGlobales vg)
    {
        pizarra = new Pizarra(vg.nCuadros);
        caballetes = new Caballete(vg.nCaballetes);
        pedido = new Peticion(vg.nPeticiones);
        mesa = new Mesa(vg.nPeticiones);
    }

    /**
     * Método que escribe 'nEscribir' títulos en la cola FIFO pizarra.
     */
}
```

```
*
* Inicialmente será llamado con el número máximo de títulos que admite
* la pizarra (para inicializarla), en las siguientes llamadas se
* introducirán títulos de uno en uno a medida que vaya habiendo sitio.
* Escribe en la pizarra e indica por pantalla usando un bucle for de
* 'nEscribir' iteraciones.
*
* @param titulo[] Array de enteros que representa los títulos que se
* quiere escribir en la pizarra.
* @param nEscribir Número de elementos del array que deben ser escritos
* en la pizarra (empezando por el que tienen índice 0).
*/
public void escribirPizarra(int[] titulo, int nEscribir)
{
    for(int i=0; i<nEscribir; i++)
    {
        pizarra.ponerTitulo(titulo[i]);
        escribeCuadro(true, -1, titulo[i]);
    }
}

/**
* Método que lee y consume un título de la pizarra.
*
* Se reserva un cuadro (se actualiza el puntero frente pero no el tamaño de la
* cola pizarra) mediante una llamada al método reservarTitulo de la cola
* pizarra, se informa del título escogido y de que se empieza a pintar.
*
* @param alumno Identificador del alumno en la clase, necesario para la
* salida de mensajes por pantalla.
*/
public void leerPizarra(int alumno)
{
    int l = pizarra.reservarTitulo();
    escribeCuadro(false, alumno, l);
    System.out.println("Alumno " + alumno + ": Tengo todo, empiezo a pintar el
cuadro");
}

/**
* Método que hace efectivo el consumo de un título.
*
* Se informa de la finalización del cuadro y se actualiza la cantidad de
* títulos haciendo una llamada al método borrarTitulo() de la cola pizarra
* (que únicamente llama al método disminuirTamanho() heredado de ColaFIFO).
*
* @param alumno Identificador del alumno en la clase, necesario para la
* salida de mensajes por pantalla.
*/
public void borrarPizarra(int alumno)
```

```
{
    System.out.println("Alumno " + alumno + ": Aviso al profesor y me voy al pasillo a
descansar");
    pizarra.borrarTitulo();    //Hace efectivo el borrado
}

/**
 * Método que produce un caballete libre.
 *
 * Actualiza punteros mediante la llamada a ponerCaballete() de la
 * cola caballete e informa de que ha acabado
 *
 * @param alumno Identificador numérico del alumno en la clase.
 */
public void devolverCaballete(int alumno)
{
    caballetes.ponerCaballete();
    System.out.println("Alumno " + alumno + ": He acabado de pintar el cuadro,
devuelvo el caballete");
}

/**
 * Método que consume un caballete libre.
 *
 * Obtiene un caballete y lo indica por pantalla.
 *
 * @param alumno Identificador numérico del alumno en la clase.
 */
public void pedirCaballete(int alumno)
{
    caballetes.quitarCaballete();
    System.out.println("Alumno " + alumno + ": He conseguido el caballete");
}

/**
 * Método que produce pedidos para ser atendidos por el bedel.
 *
 * Se comprueba mediante el parámetro 'primeraVez' si no es o si es la
 * primera vez que se hace uso del método (en cuyo caso se indica o no por
 * pantalla que se vuelve al trabajo), a continuación se pone el pedido en
 * la cola 'pedido' mediante la llamada al método ponerPedido().
 *
 * @param util Cadena de caracteres que indica el tipo de utensilio que se va
 * a poner en la cola de pedidos. Puede ser 'bote' o 'pincel', pero
 * no se hace comprobación de la corrección de la cadena.
 * @param alumno Identificador numérico del alumno en la clase.
 * @param primeraVez Indicador de si es la primera vez (true) o no (false) que
 * se hace un pedido al bedel.
 */
public void hacerPedido(String util, int alumno, boolean primeraVez)
```

```
{
    if(!primeraVez)
        System.out.println("Alumno " + alumno + ": Vuelvo al trabajo");
    pedido.ponerPedido(util);
}

/**
 * Método que consume pedidos, usado por el bedel cuando presta atención
 * a uno nuevo.
 *
 * Obtiene un pedido de la cola e informa por pantalla de qué utensilio ha
 * sacado, a continuación devuelve la cadena correspondiente al utensilio.
 *
 * @return Cadena que indica el tipo de pedido que ha atendido. Puede ser un
 *         'bote' o un 'pincel', aunque no se hace comprobación de lo obtenido
 *         más que para así indicarlo por pantalla.
 */
public String atenderPedido()
{
    String p;

    p = pedido.quitarPedido();
    if(p.equals("bote"))
        System.out.println("Bedel: Me han pedido un bote, voy al almacén a buscarlo");
    else
        System.out.println("Bedel: Me han pedido un pincel, voy al almacén a buscarlo");

    return p;
}

/**
 * Método que produce utensilios en respuesta a los pedidos, depositándolos en
 * una mesa gestionada como cola FIFO.
 *
 * Similar a 'devolverCaballero()', pone un utensilio en la mesa llamando
 * al método ponerMesa y selecciona el tipo de mensaje en función del tipo de
 * utensilio depositado en la mesa.
 *
 * @param util Cadena que indica el tipo de utensilio que se va a poner en la
 *             mesa. El utensilio puede ser 'bote' o 'pintura', pero no se
 *             hace comprobación del mismo más que para distinguir entre los
 *             mensajes que se sacan por pantalla.
 */
public void ponerMesa(String util)
{
    mesa.ponerMesa(util);
    if(mesa.utensilioEs("bote"))
        System.out.println("Bedel: He puesto en la mesa un bote");
    else
        System.out.println("Bedel: He puesto en la mesa un pincel");
}
```

```
}

/**
 * Método que consume utensilios de tipo 'bote'.
 *
 * Si el primer objeto que hay en la mesa es un bote, lo quita e informa
 * de lo mismo por pantalla, devolviendo true para informar de que ha tendio
 * éxito; si no es un bote el primero de la cola, devuelve false indicando
 * que no ha extraído nada.
 * El método utensilioEs() de la cola mesa consulta el frente de la misma
 * sin retirar nada.
 *
 * @param alumno Identificador numérico del alumno en la clase.
 */
public boolean quitarBoteMesa(int alumno)
{
    if(mesa.utensilioEs("bote"))
    {
        mesa.quitarMesa();
        System.out.println("Alumno " + alumno + ": He conseguido el bote de pintura");
        return true;
    }
    return false;
}

/**
 * Método que consume utensilios de tipo 'pincel'.
 *
 * Si el primer objeto que hay en la mesa es un pincel, lo quita e informa
 * de lo mismo por pantalla, devolviendo true para informar de que ha tendio
 * éxito; si no es un pincel el primero de la cola, devuelve false indicando
 * que no ha extraído nada.
 * El método utensilioEs() de la cola mesa consulta el frente de la misma
 * sin retirar nada.
 *
 * @param alumno Identificador numérico del alumno en la clase.
 */
public boolean quitarPincelMesa(int alumno)
{
    if(mesa.utensilioEs("pincel"))
    {
        mesa.quitarMesa();
        System.out.println("Alumno " + alumno + ": He conseguido el pincel");
        return true;
    }
    return false;
}

/**
 * Método para escribir por pantalla el título del cuadro que se propone
```

```
* (profesor) o que se va a pintar (alumno).
*
* Necesita saber si es el profesor (true/false) o el alumno; en el primer
* caso prescinde del parámetro alumno y en el segundo necesita el identificador
* (entero) del mismo. Necesita también el identificador (entero) del cuadro.
*
* Empieza obteniendo la cadena del título mediante una llamada al método
* estático dameCuadro() de la clase RepositorioCuadros; para ello el método
* hace una conversión interna del identificador a fin de que no se pida una
* cadena que no figura en el repositorio. A continuación se hace una llamada
* a println con los parámetros 'alumno' y 'tempTitulo' (si es el alumno) o
* con el parámetro 'tempTitulo' si es el profesor.
*
* @param profesor Indica si es el profesor (true) el hace la llamada o bien un
* alumno (false).
* @param alumno Identificador numérico del alumno en la clase.
* @param numCuadro Identificador numérico del título del cuadro que se propone
* (en caso de que sea el profesor) o que se va a pintar (en
* caso de que sea el alumno).
*/
public void escribeCuadro(boolean profesor, int alumno, int numCuadro)
{
    String tempTitulo = new RepositorioCuadros().dameCuadro(numCuadro);
    if(profesor)
        System.out.println("Profesor: He anotado en la pizarra " + tempTitulo);
    else
        System.out.println("Alumno " + alumno + ": Escojo para pintar " + tempTitulo);
}
}
```

Clase SemaforoConCola

```
/**
 * Implementa un semáforo general con una cola FIFO asociada, para
 * permitir que se despierten los hilos en el mismo orden en que
 * se fueron bloqueando.
 *
 * @author Víctor M. Álvarez Pérez (A partir del libro de Palma et al.)
 * @version Junio de 2007
 */
public class SemaforoConCola extends SemaforoGeneral
{
    Hilos bloqueados; //Cola FIFO para bloquear los hilos

    /**
     * Constructor de la clase SemaforoConCola.
     * Pone el valor inicial del semáforo e inicia la cola.
     *
     * @param valorInicial Establece el valor inicial del semáforo.
     */
}
```

```
* @param nMaxHilos Establece el número máximo de hilos que
* se pueden almacenar simultáneamente en la cola.
*/
public SemaforoConCola(int valorInicial, int nMaxHilos)
{
    super(valorInicial);
    bloqueados = new Hilos(nMaxHilos);
}

/**
 * Operación wait sobre el semáforo.
 * Sobreescribe el mismo método de SemaforoBinario.
 * Decrementa el valor del contador interno si éste es mayor que 0,
 * si es igual a 0 el hilo que ha hecho la llamada pasará a formar
 * parte de una cola FIFO de hilos bloqueados (se lanza una excepción
 * en caso de superarse el tamaño de la cola) y se bloqueará en el
 * semáforo.
 * Cuando se trate de desbloquear un hilo después de una llamada a
 * signal, se comprobará si el hilo que se quiere desbloquear es el
 * primero de la cola. En caso afirmativo se extrae de la cola antes de
 * desbloquear (se lanza una excepción en caso de no haber hilos en la
 * cola); por el contrario, si no es el primero de la cola se notifica
 * (notify) que se quiere desbloquear otro y el actual pasa a bloquearse
 * nuevamente al no cumplirse la condición de salida del bucle while.
 */
synchronized public void WAIT()
{
    boolean condSalida = false;
    while(contador == 0)
    {
        try
        {
            bloquearHilo(Thread.currentThread());
        }
        do
        {
            wait();

            condSalida =
                bloqueados.primerosEsIgualA(Thread.currentThread());
            if(!condSalida) notify();
        } while(!condSalida);
        condSalida = false;
        desbloquearHilo();
    }
    catch(InterruptedException e) {}
    contador--;
}

/**
 * Método privado para bloquear un hilo.
 * Se incluye para enmascarar el bloque try-catch en el que es
```

```
* necesario incluir la llamada a ponerHilo y así facilitar la lectura
* del método WAIT.
*/
private void bloquearHilo(Thread proc)
{
    try
    {
        bloqueados.ponerHilo(proc);
    }
    catch(Overflow e)
    {
        System.err.println("Superada la capacidad de la cola de hilos bloqueados. Finaliza
la ejecucion.");
    }
}
/**
 * Método privado para desbloquear un hilo.
 * Se incluye para enmascarar el bloque try-catch en el que es
 * necesario incluir la llamada a quitarHilo y así facilitar la lectura
 * del método WAIT.
 */
private void desbloquearHilo()
{
    try
    {
        bloqueados.quitarHilo();
    }
    catch(Empty e)
    {
        System.err.println("No se puede desbloquear hilo por estar la cola vacía. Finaliza
la ejecucion.");
    }
}
}
```

Clase SemaforoGeneral

```
/**
 * Implementa un semáforo general.
 * Para ello sobrescribe el método SIGNAL de la clase
 * SemaforoBinario de la que hereda.
 *
 * @author Víctor M. Álvarez Pérez (Copiado del libro de Palma et al.)
 * @version Junio de 2007
 */
public class SemaforoGeneral extends SemaforoBinario
{
    /**
     * Constructor de la clase SemaforoGeneral.
     * Pone el valor inicial del semáforo.
     *
     * @param valorInicial Establece el valor inicial del semáforo.
     */
    public SemaforoGeneral(int valorInicial)
    {
        super(valorInicial);
    }

    /**
     * Operación signal sobre el semáforo.
     * Sobrescribe el mismo método de SemaforoBinario.
     * Incrementa el contador interno y a continuación trata de desbloquear
     * un proceso. En el caso de que no haya ningún proceso bloqueado,
     * el valor del semáforo aumenta en una unidad. Si se desbloquea un
     * proceso no se altera el valor del semáforo puesto que WAIT
     * decrementa el contador como última instrucción.
     */
    synchronized public void SIGNAL()
    {
        contador++;
        notify();
    }
}
```

Clase SemaforoBinario

```
/**
 * Implementa un semáforo binario.
 *
 * @author Víctor M. Álvarez Pérez (Copiado del libro de Palma et al.)
 * @version Junio de 2007
 */
public class SemaforoBinario
{
    protected int contador = 0;    //Valor del semáforo en cada momento.
                                   //Con el constructor por defecto se establece a 0.

    /**
     * Constructor de la clase SemaforoBinario.
     * No es necesario para la clase SemaforoBinario, se incluye
     * para facilitar la herencia.
     *
     * @param valorInicial Establece el valor inicial del semáforo.
     */
    public SemaforoBinario(int valorInicial)
    {
        contador = valorInicial;
    }

    /**
     * Operación wait sobre el semáforo.
     * Decrementa el valor del contador interno si éste es mayor que 0,
     * si es igual a 0 el proceso que ha hecho la llamada se bloqueará
     * en el semáforo.
     */
    synchronized public void WAIT()
    {
        while(contador == 0)
            try {wait();}
            catch(InterruptedException e) {}
        contador--;
    }

    /**
     * Operación signal sobre el semáforo.
     * Pone el contador interno a 1 y trata de desbloquear algún proceso
     * bloqueado en el semáforo.
     */
    synchronized public void SIGNAL()
    {
        contador = 1;
        notify();
    }
}
```

Clase Hilos

```
/**
 * Particulariza el uso del colaFIFO a la idiosincrasia de uso
 * de una cola de hilos.
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class Hilos extends ColaFIFO
{
    /**
     * Constructor de la clase Hilos
     *
     * @param maxHilos El máximo número de hilos que pueden
     *                estar simultáneamente en la cola.
     */
    public Hilos(int maxHilos)
    {
        // Usa el constructor de la clase padre.
        super(maxHilos);
    }

    /**
     * Pone un hilo en la cola
     *
     * @param hilo Identificador del hilo que se quiere
     *             guardar en la cola.
     */
    public void ponerHilo(Thread hilo) throws Overflow
    {
        if(!estaLleno())
            poner(hilo);
        else
            throw new Overflow();
    }

    /**
     * Obtiene el primer hilo de los que están depositados en la cola
     *
     * @return El identificador del hilo que se demanda.
     */
    public Thread quitarHilo() throws Empty
    {
        if(!estaVacio())
            return (Thread) quitar();
        else
            throw new Empty();
    }
}
```

```
/**
 * Permite comprobar si el hilo que está en primer lugar coincide
 * con el que se pasa como parámetro, sin necesidad de retirarlo.
 * La comprobación se hace en base al nombre de ambos hilos.
 *
 * @param hilo El identificador del hilo que se quiere comparar con
 *             el que está en primer lugar.
 * @return Verdadero (true) si el nombre del hilo que está en primer
 *         lugar coincide con el nombre del hilo 'hilo', falso (false)
 *         en caso contrario.
 */
public boolean primeroEsIgualA(Thread hilo)
{
    Thread temp = (Thread) consultarFrente();
    String tempNombre = temp.getName();
    return tempNombre.equals(hilo.getName());
}
```

Clase Overflow

```
/**
 * Excepción para cuando se ha superado el tamaño fijado para una
 * estructura de datos
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class Overflow extends Exception
{
    /**
     * Constructor por defecto, llama al constructor de la clase Exception
     */
    public Overflow()
    {
        super();
    }
}
```

Clase Empty

```
/**
 * Excepción para cuando no hay elementos en una estructura de datos
 *
 * @author Víctor M. Álvarez Pérez
 * @version Junio de 2007
 */
public class Empty extends Exception
{
    /**
     * Constructor por defecto, llama al constructor de la clase Exception
     */
    public Empty()
    {
        super();
    }
}
```

4.- Bibliografía.

- Barnes, D.; Kölling M. (2002) “Objects First with Java. A practical Introduction using BlueJ” Ed. Pearson Education-Prentice Hall.
- Davies, G.L. (1992) “Pascal-FC version 5. Language Reference Manual” Documento en pdf que acompaña al compilador.
- Davies, G.L. (1992) “Pascal-FC version 5. User Guide for PC Compatibles PC-PP” Documento en pdf que acompaña al compilador.
- Eckel, B. (2002) “Piensa en Java”. 2ª edición. Ed. Prentice Hall.
- Esteban, A. (2000) “Programación en Java”. Ed. Grupo Eidos.
- García de Jalón, J. et al (2000) “Aprenda Java como si estuviera en primero”. Editado en pdf por la Escuela Superior de Ingenieros Industriales de San Sebastián. Univ. de Navarra.
- Goetz, B. et al (2006) “Java Concurrency in Practice”. Addison Wesley Professional.
<http://java.sun.com/books/Series/Tutorial/index.html> (También traducción al castellano).
- Morero, F. (2000) “Introducción a la OOP”. Ed. Grupo Eidos.
- Palma, J.T. et al (2003) “Programación Concurrente”. Ed. Thomson.
- Pérez, J.E. (1990) “Programación Concurrente”. 2ª edición. Ed. Rueda.
- Schildt, H. (2007) “Fundamentos de Java”. 3ª edición. Ed. McGraw-Hill.
- Zakhour, S. et al (2006) “The Java Tutorial Fourth Edition: A Short Guide on the Basics”. AddisonWesley Professional.