EXAMEN PRIMER PARCIAL — **Concurrencia y Sistemas Distribuidos**

Unidades Didácticas 1 a 6 - Prácticas 1 y 2 — **20 de Abril de 2015**

Este examen tiene una duración total de 2 horas y 30 minutos. Consta de dos partes: TEORIA y PRÁCTICAS.

PARTE TEORIA

Esta parte tiene una puntuación máxima de **10 puntos**, que equivalen a **3** puntos de la nota final de la asignatura. Indique, para cada una de las siguientes **50 afirmaciones**, si éstas son verdaderas (**V**) o falsas (**F**).

Cada respuesta vale: correcta= 0.2, errónea= -0.2, vacía=0.

Importante: Los **primeros 3 errores no penalizarán**, de modo que tendrán una valoración equivalente a la de una respuesta vacía. A partir del 4º error (inclusive), sí se aplicará el decremento por respuesta errónea.

Sobre la programación concurrente ... (respecto a la programación secuencial)

1. La programación concurrente aprovecha mejor los recursos máquina. *JUSTIFICACIÓN: Efectivamente, es más eficiente, pues explota mejor los recursos máquina (Ver tema 1)*	V
2. La programación concurrente modela mejor aquellos tipos de problemas que se descomponen en actividades separadas que coordinan su ejecución. *JUSTIFICACIÓN: Resulta más adecuada para aquellos problemas que se definen de forma natural como una colección de actividades. (Ver tema 1)*	V
3. La programación secuencial ofrece mejores tiempos de respuesta a las acciones del usuario. *JUSTIFICACIÓN: La programación concurrente mejora la interactividad y flexibilidad (Ver tema 1)*	F
4. La programación concurrente lanza varios hilos que no necesitan coordinarse entre sí. *JUSTIFICACIÓN: Los hilos de una aplicación concurrente cooperan para llevar a cabo una tarea común, y por lo tanto necesitan comunicarse y coordinar su ejecución*	F
5. Un objeto constante (cuyo estado no cambia) puede compartirse de forma segura entre varios hilos sin necesidad de tomar precauciones especiales. *JUSTIFICACIÓN: Sólo pueden surgir problemas en objetos compartidos que cambian su estado (y por tanto un hilo puede acceder a un estado intermedio o inconsistente)*	V

Sobre la programación concurrente en Java...

6. Un hilo empieza a ejecutar el código indicado en su método *run*() en el momento en que creamos dicho hilo (ej. *new Thread(..)*) *JUSTIFICACIÓN: Hasta que no se ejecuta el método start no entra en preparados, y por tanto no es elegible para ejecución.*	F
7. Una aplicación concurrente termina cuando finaliza su hilo principal. *JUSTIFICACIÓN: En Java la aplicación concurrente solo termina cuando finalizan todos los hilos*	F
8. El método *sleep* permite pasar directamente de ejecución a preparado. *JUSTIFICACIÓN: Sleep supone pasar de ejecución a timed-waiting*	F
9. Cuando creamos en Java un objeto que debe compartirse entre distintos hilos, utilizamos la etiqueta 'synchronized' (ej. *synchronized Buffer b*). *JUSTIFICACIÓN: La etiqueta synchronized se aplica a cada método público de la clase a partir de la cual se declaran objetos a compartir, pero no al objeto propiamente dicho.*	F
10. El método *join* permite resolver la sincronización condicional. *JUSTIFICACIÓN: Join solo espera a que finalice el hilo indicado como argumento. No sirve para esperar hasta que se cumpla determinada condición.*	F
11. Utilizamos los métodos *wait*, *notify*, *notifyAll* para resolver la exclusión mutua. *JUSTIFICACIÓN: Para la exclusion mutua se utiliza la etiqueta synchronized sobre los métodos públicos.*	F
12. Todo objeto Java posee un lock y una variable condición implícitas. *JUSTIFICACIÓN: El lock implícito se usa para cerrar/abrir al principio/final de cada método sincronizado, y La variable condición implícita es el destino de las operaciones wait, notify, notifyAll*	V

Sobre el concepto de monitor y sus variantes...

13. El concepto de monitor evita la necesidad de que distintos hilos compartan memoria. *JUSTIFICACIÓN: Precisamente el objetivo del monitor es regular el acceso a variables compartidas para garantizar una compartición segura.*	F
14. Un monitor es una clase que además resuelve exclusión mutua y sincronización condicional. *JUSTIFICACIÓN: Efectivamente, el monitor garantiza que sus métodos se ejecutan en exclusión mutua y proporciona las variables condición para realizar la sincronización condicional (Ver tema 3)*	V
15. Los monitores tipo Brinch-Hansen y Hoare garantizan que tras una operación *Notify* el hilo reactivado encuentra el estado del monitor exactamente igual que estaba cuando se ejecutó dicho *Notify.* *JUSTIFICACIÓN: Es cierto, porque el hilo reactivado pasa a controlar inmediatamente el monitor, mientras que el hilo que ha ejecutado Notify deja de ejecutar código del monitor (en el modelo de Hansen porque finaliza la operación, en Hoare porque espera en una cola especial a que el monitor quede libre)*	V
16. En un monitor tipo Brinch-Hansen no puede implementarse la operación *NotifyAll*, porque no podría garantizar la exclusión mutua en caso de reactivar a más de un hilo. *JUSTIFICACIÓN: Por definición de este modelo de monitor el hilo reactivado toma el control del monitor: si hay más de uno, no se cumple la exclusión mutua.*	V
17. En Java, la sentencia *notify*() o *notifyAll*() debe ser obligatoriamente la última del método correspondiente. *JUSTIFICACIÓN: Eso solo ocurre en el modelo de Brinch Hanse, y Java sigue el modelo Lampson-Redell*	F
18. El monitor tipo Lampson-Redell utiliza una cola especial prioritaria sobre la entrada, donde esperan aquellos que han ejecutado *Notify.* *JUSTIFICACIÓN: Esa definición corresponde al modelo de Hoare*	F

En un bosque conviven 10 lobos y 10 corderos que comparten un río al que acceden para beber. El acceso al río se gestiona por el siguiente código:

```
1   monitor Rio {
2     condition entrar,salir;
3     int dentro=0;
4
5     entry void lobo_entrar()
6     {
7        // lobo tiene hambre y sed
8        if ( dentro == 1) comer_cordero();
9     }
10
11     public void lobo_salir()
12    {   // lobo deja de beber  }
13      entry void cordero_entrar()
14      { // cordero tiene sed
15         entrar.notify();
16         if ( dentro == 0) entrar.wait();
17         dentro++;
18         salir.notify();
19       }
20
21     entry void cordero_salir()
22     {   salir.notify();
23         if ( dentro == 2) salir.wait();
24         dentro--;
25         // cordero deja de beber
26       }
27   } // fin monitor
```

Asumiendo la variante de monitores de Hoare, y que tanto los lobos como los corderos respetan siempre el protocolo de invocar el método *lobo/cordero_entrar*(), antes de acceder al río para beber y *lobo/cordero_salir*() cuando abandonan el río...

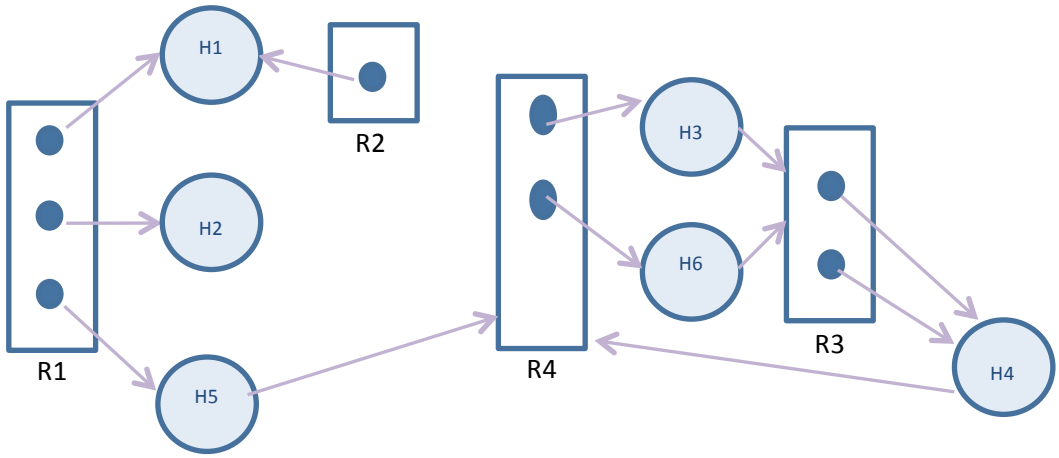
19. Los lobos y los corderos acceden al río en exclusión mutua. Es decir, si hay lobos bebiendo no puede haber corderos bebiendo y viceversa. *JUSTIFICACIÓN: Los lobos pueden entrar libremente (sin ninguna condición de espera), y los corderos no comprueban si hay lobos antes de entrar. Supongamos que se invocan sucesivamente lobo_entrar (el lobo entra), cordero_entrar (el cordero espera), cordero_entrar (reactiva al que esperaba).*	F

20. Si no paran de llegar corderos al río, los lobos nunca pueden entrar a beber por lo que hay inanición para los lobos. *JUSTIFICACIÓN: Los lobos pueden entrar libremente (nunca esperan, aunque haya corderos bebiendo)*	F
21. Si la variante de monitor utilizada es la de Brinch Hansen, se extinguirán los corderos del bosque, pues los lobos se los irán comiendo uno a uno cuando accedan al río a beber. *JUSTIFICACIÓN: En Brinch-Hansen, cuando un hilo ejecuta notify() es expulsado del monitor (por lo que si hay más sentencias en el método que está ejecutando, esas sentencias no se ejecutan). Por tanto, en el método "cordero_entrar" los corderos solamente realizarán el método "entrar.notify()", y lo mismo ocurre con "cordero_salir", que solamente realizan "salir.notify()". Como no se actualiza "dentro" (que tendrá siempre el valor 0), los lobos no se comerán a los corderos.*	F
22. Con este código, los lobos no se podrán comer nunca a los corderos. *JUSTIFICACIÓN: Los corderos siempre entran en grupo y salen en grupo (el primero en llegar espera a otro, y cuando alguien quiere salir y quedan 2 bebiendo, espera a que lleguen otros). En consecuencia, cuando un lobo controla el monitor, "dentro" no vale 1.*	V
23. La variable *dentro* indica correctamente el número de corderos bebiendo simultáneamente. *JUSTIFICACIÓN: Como un cordero está bebiendo a partir del momento en que finaliza cordero_entrar, y sigue bebiendo hasta que finaliza cordero_salir, es cierta.*	V
24. Si la variante de monitor utilizada es la de Lampson y Redell, corderos y lobos pueden beber simultáneamente en el río, y los lobos no se podrán comer nunca a los corderos. *JUSTIFICACIÓN: Con Lampson-Redell el primer cordero espera, y cuando el segundo en llegar lanza el aviso es el segundo el que continua, y entra él en el río. Si ya había un lobo esperando en la entrada antes del cordero reactivado, el lobo podrá comerse al cordero que ya está bebiendo)*	F

Sobre las herramientas de la biblioteca *java.util.concurrent*...

25. El objeto *CountDownLatch* es una barrera que una vez abierta ya no puede ser utilizada de nuevo. *JUSTIFICACIÓN: No es reutilizable, porque una vez abierta no vuelve al estado inicial, sino que permanece abierta para siempre (ni la operación await ni countDown tendrían ningún efecto)*	V
26. Dados 5 hilos de la clase A y 3 hilos de la clase B que comparten el mismo objeto "c" de tipo *CountDownLatch* inicializado a 5, sabiendo que los hilos A ejecutan *c.await*() y los hilos B ejecutan *c.countDown*(), todos los hilos A quedarán suspendidos. *JUSTIFICACIÓN: Como el valor inicial es 5, y sólo se ejecutarían 3 operaciones countDown (por parte de cada uno de los 3 hilos de tipo B), el contador no llega a 0 y la barrera no se abre: los 5 hilos tipo A que habían ejecutado await quedan suspendidos.*	V
27. La clase *Executor* ofrece principalmente métodos que permiten medir intervalos de tiempo de forma exacta, haciendo uso del tipo enumerado *TimeUnit*. *JUSTIFICACIÓN: La clase Executor sirve para facilitar el desarrollo de patrones de ejecución adecuados para servidores (ej pool de hilos, etc.)*	F
28. Dados 5 hilos de la clase A y 3 hilos de la clase B que comparten el mismo objeto "c" de tipo *CyclicBarrier* inicializado a 4, sabiendo que los hilos A ejecutan *c.await*() y los hilos B también ejecutan *c.await*(), algún hilo de la clase A podrá quedarse suspendido indefinidamente. *JUSTIFICACIÓN: Dado que todos los hilos ejecutan await, no distinguimos entre hilos A y B. Los 3 primeros esperan, y en cuanto llega el cuarto la barrera se abre y pasan los 4: en ese momento se restauran las condiciones iniciales, y con los 4 restantes pasa exactamente lo mismo*	F
29. Dados 5 hilos de la clase A y 3 hilos de la clase B que comparten el mismo objeto "c" de tipo *Semaphore* inicializado a 3, sabiendo que los hilos A ejecutan *c.acquire()* y los hilos B ejecutan *c.release*(), todos los hilos B quedarán suspendidos, pues un hilo no puede realizar el método "*release*()" sobre un semáforo si previamente no ha adquirido un permiso de dicho semáforo. *JUSTIFICACIÓN: La operación release no provoca la suspensión del hilo, y tampoco existe ninguna limitación en cuanto a la posibilidad de invocar release*	F

Dado el siguiente grafo de asignación de recursos:



30. Si los procesos H1 y H2 solicitan cada uno una instancia del recurso R4, se producirá un interbloqueo. *JUSTIFICACIÓN: Ninguno de los procesos podría proseguir, y todos estarían esperándose mutuamente (def. de interbloqueo)*	V
31. Existe una secuencia segura dada por: H1, H2, H5, H4, H3 y H6. *JUSTIFICACIÓN: H1 y H2 se podrían ejecutar, pero H5 no podría continuar.*	F
32. Existe un ciclo dirigido y como el recurso R2 tiene una única instancia, podemos afirmar que hay un interbloqueo. *JUSTIFICACIÓN: Cuando existe un ciclo, solo podemos afirmar que hay interbloqueo cuando TODOS los recursos involucrados poseen una única instancia*	F
33. Se encuentran en interbloqueo los procesos H3 y H6. *JUSTIFICACIÓN: Forman parte de un grupo mayor de procesos interbloqueados.*	V
34. El grafo de asignación de recursos puede utilizarse como mecanismo para romper las condiciones de Coffman. *JUSTIFICACIÓN: Si se rompen condiciones de Coffman no hay posibilidad de interbloqueo, y no necesitamos el grafo. El grafo hace falta cuando no se pueden prevenir los interbloqueos y queremos aplicar evitación (algoritmo del banquero)*	F
35. En muchos sistemas operativos, como Unix o Windows, se utilizan grafos de asignación de recursos para evitar los interbloqueos. *JUSTIFICACIÓN: Lo habitual en los Sistemas Operativos (ej Unix y Windows) es no hacer nada respecto a los interbloqueos*	F

Sea el conjunto de tareas en un sistema de tiempo real descrito por la siguiente tabla:

Tarea	Periodo (T)	Cómputo (C)	Plazo (D)	Prioridad	Usa Si (t)
A	10	2	6	1	S1 (3)
B	15	3	10	2	S2 (5)
C	20	4	15	3	
D	25	2	20	4	S2 (3)
E	30	5	30	5	S1 (2)

Asumiendo una asignación de prioridades en la que la tarea con menor valor numérico será la más prioritaria, un algoritmo de planificación por prioridades fijas expulsivas, los semáforos utilizan el protocolo del techo de prioridad inmediato y que la notación Si (t) significa que la tarea usa el semáforo Si durante t unidades de tiempo,

36. El techo de S1 es 3, que corresponde a la mayor duración de las secciones críticas guardadas por S1. *JUSTIFICACIÓN: El techo de prioridad del semáforo corresponde a la prioridad máxima entre los procesos que utilizan dicho semáforo: no tiene ninguna relación con la duración de las secciones críticas protegidas por ese semáforo.*	F

37. El factor de bloqueo de la tarea B es 3. *JUSTIFICACIÓN: Las menos prioritarias que B son C,D,E. Entre ellas usan los semáforos S1 y S2, ambos con un techo de prioridad >= que el de B. La sección crítica más larga protegida por ellos en los hilos C,D,E es 3.*	V
38. El factor de bloqueo de la tarea C es cero, pues no utiliza ningún semáforo. *JUSTIFICACIÓN: Aunque C no espera nunca en una operación P, puede que tenga que esperar porque otra tarea menos prioritaria (D o E) ha modificado temporalmente su prioridad para asumir la del techo de prioridad del semáforo que protege la sección crítica que está ejecutando. Por lo tanto, puede tener que esperar por culpa de otras tareas menos prioritarias (D o E).*	F
39. El tiempo de respuesta de peor caso de la tarea E es 29, por lo que queda garantizado el cumplimiento del plazo en todas sus activaciones. *JUSTIFICACIÓN: El factor de bloqueo para E es 0 (no hay tareas menos prioritarias), y el tiempo de respuesta queda como sigue:* *Primera estimación: Re = Ce + Ca + Cb + Cc + Cd = 16* *Iteración: Re = Ce + ceil(16/Ta)*Ca + ceil(16/Tb)*Cb + ceil(16/Tc)*Cc + ceil(16/Td)*Cd = Ce + 2Ca + 2Cb + Cc + Cd = 5+ 2*2 + 2*3 + 4 + 2 = 21* *Iteración: Ce + ceil(21/Ta)*Ca + ceil(21/Tb)*Cb + ceil(21/Tc)*Cc + ceil(21/Td)*Cd = Ce + 3Ca + 2Cb + 2Cc + Cd = 27* *Iteración: Ce + ceil(27/Ta)*Ca + ceil(27/Tb)*Cb + ceil(27/Tc)*Cc + ceil(27/Td)*Cd = Ce + 3Ca + 2Cb + 2Cc + 2Cd = 29* *Iteración: Ce + ceil(29/Ta)*Ca + ceil(29/Tb)*Cb + ceil(29/Tc)*Cc + ceil(29/Td)*Cd = Ce + 3Ca + 2Cb + 2Cc + 2Cd = 29* *Luego el valor final es 29*	V

Dado el siguiente código:

```
public class BufferMin{
     private int[] data;
     private int elems, head, tail, N;
     Condition notFull, notMinimum;
     ReentrantLock rl;

 public BufferMin(int N){
    data= new int[N];
    this.N=N;
    head=tail=elems=0;
    rl=new ReentrantLock();
    notFull=rl.newCondition();
    notMinimum=rl.newCondition();
}
```

```
public int getPairs()
{  int x, y, sum;
    try{
      rl.lock();
      while(elems<2){
                try { notMinimum.await();
                } catch(InterruptedException e) {} }
     x=data[head];  head=(head+1)%N;
     y=data[head];  head=(head+1)%N;
     sum=x+y;
     elems=elems-2;
     notFull.signal();
 } finally { rl.unlock();}
  return (sum);
}

public void insert(int x) {
 try{
     rl.lock();
     while (elems==N) {
       try {notFull.await();
       }  catch(InterruptedException e) {} }
    data[tail]=x; tail= (tail+1)%N; elems++;
    if (elems>=2)
      notMinimum.signal();
 } finally {rl.unlock();}
}
```

40. Para que la clase *BufferMin* pueda ser utilizada por varios hilos de forma segura (*thread-safe*) debemos etiquetar sus métodos con "*synchronized*". *JUSTIFICACIÓN: Estamos utilizando de forma explícita una variable lock y operaciones lock/unlock sobre la misma (tal y como se ha estudiado en el tema 4)*	F
41. Si varios hilos utilizan los métodos *getPairs*() e *insert*() de la clase *BufferMin*, se pueden producir condiciones de carrera al acceder de forma concurrente al parámetro *data*. *JUSTIFICACIÓN: El código anterior convierte dichas operaciones en atómicas: no hay condiciones de carrera*	F

42. Para que la clase *BufferMin* pueda ser considerada como un monitor en Java, solamente se podría haber declarado una única variable condición. *JUSTIFICACIÓN: Al declarar explícitamente una variable lock (ReentrantLock rl), podemos crear a partir de ella las variables condición que consideremos oportunas.*	F
43. La exclusión mutua de la clase *BufferMin* se podría haber implementado con un objeto *Semaphore mutex* inicializado a 1, y la sincronización condicional con las variables condición *notFull* y *notMinimum*. *JUSTIFICACIÓN: Sólo podemos crear las variables Condition a partir de una variable Lock (mediante el método newCondition de la misma)*	F
44. La clase *BufferMin* se podría haber implementado directamente con un objeto de tipo *BlockingQueue*, utilizando sus métodos *take*() y *put*() de forma apropiada, al tratarse de un problema de tipo productor-consumidor. *JUSTIFICACIÓN: BlockingQueue ofrece métodos para insertar y extraer de forma bloqueante y no bloqueante. Aunque no hay un método para extraer dos ítems como acción atómica, podría simularse ese comportamiento.*	V

Sobre los sistemas de tiempo real:

45. La inversión de prioridades ocurre cuando una tarea menos prioritaria se ejecuta por delante de una más prioritaria con la que comparte un recurso que mantiene bloqueado para regular su acceso en exclusión mutua. *JUSTIFICACIÓN: Esa es la definición de inversión de prioridades*	V
46. El protocolo del techo de prioridad inmediato impide que una tarea menos prioritaria se ejecute por delante de otra más prioritaria con la que comparte un recurso que mantiene bloqueado para regular su acceso en exclusión mutua. *JUSTIFICACIÓN: Ocurre lo contrario, es decir, una tarea a priori menos prioritaria puede convertirse temporalmente en más prioritaria.*	F
47. El protocolo del techo de prioridad inmediato evita los interbloqueos. *JUSTIFICACIÓN: Es una de las propiedades del algoritmo.*	V
48. El test de planificabilidad basado en el cálculo de los tiempos de respuesta estudiado, es válido para cualquier algoritmo de planificación del procesador. *JUSTIFICACIÓN: El test asume un algoritmo de prioridades fijas expulsivas*	F
49. Si un conjunto de tareas es planificable bajo una cierta asignación de prioridades, lo seguirá siendo aunque cambiemos las prioridades asignadas a las tareas. *JUSTIFICACIÓN: Al cambiar las prioridades cambian los tiempos de respuesta*	F
50. El peor caso de planificación posible para un conjunto de tareas planificadas por prioridades fijas expulsivas, se da cuando todas ellas consumen su tiempo de cómputo de peor caso, independientemente de cuándo se activen. *JUSTIFICACIÓN: El momento de activación es importante: hay que considerar que cada tarea consume su tiempo de cómputo en el peor caso Y además se lanzan todas las tareas a la vez*	F

PARTE PRACTICAS

Esta parte tiene una puntuación máxima de **10 puntos**, que equivalen a **1** punto de la nota final de la asignatura. Indique, para cada una de las siguientes **8 afirmaciones**, si éstas son verdaderas (**V**) o falsas (**F**).

Cada respuesta vale: correcta= 1.25, errónea= -1.25, vacía=0.

Importante: El **primer error no penalizará**, de modo que tendrá una valoración equivalente a la de una respuesta vacía. A partir del 2º error (inclusive), sí se aplicará el decremento por respuesta errónea.

Sobre la práctica 1 "Uso compartido de una piscina":

1. En la piscina Pool2 (pueden nadar un máximo de K/I niños por instructor), se debe invocar *notifyAll* cuando un instructor entra en la piscina. *JUSTIFICACIÓN: Es posible que la entrada de ese instructor permita nadar a algunos niños que estaban esperando.*	V
2. Cuando la llamada al método *kidSwims* de la piscina finaliza, sabemos que el niño que ha hecho la llamada está nadando. *JUSTIFICACIÓN: La operación tardará más o menos en completarse (quizá tenga que esperar en la variable condición), pero una vez finalizada tenemos la seguridad de que está nadando*	V
3. En la piscina Pool1 (los niños no pueden nadar solos), para representar el estado del objeto compartido es suficiente con una variable de tipo entero (*nSwimKids*) y una variable de tipo booleano (*instrInPool*). *JUSTIFICACIÓN: Si utilizamos una variable lógica para los instructores, cuando sale alguno de ellos no podremos determinar si es el último o no.*	F
4. En la piscina Pool3 (máximo aforo permitido) no es necesario añadir otra variable de tipo entero al estado de la piscina, respecto de la implementación de Pool2. *JUSTIFICACIÓN: Para determinar la cantidad actual de nadadores podemos añadir una variable que se incrementa cuando entra un niño o instructor y se decrementa un niño o instructor, pero también podemos deducirla como la suma de niños + instructores que están nadando (dichas variables ya eran necesarias en las piscinas Pool1 y Pool2)*	V

Sobre la práctica 2 "Los cinco filósofos comensales":

5. La clase *Log* contiene métodos que deben ser invocados cuando se produce un cambio en el estado de un filósofo, para que la interfaz gráfica pueda mostrar el estado correcto. *JUSTIFICACIÓN: Efectivamente (Ver el boletín de la práctica)*	V
6. La clase *LefthandPhilo* que se pedía implementar corresponde a un filósofo que, dependiendo de su posición en la mesa, coge primero uno de los tenedores u otro. *JUSTIFICACIÓN: El código de LeftHandPhilo siempre es igual: coge primero el tenedor izquierdo y luego el derecho.*	F
7. En la solución basada en las clases *PhiloBothOrNone* y *BothNoneTable*, el filósofo suelta el tenedor que ha cogido si no puede coger el segundo. *JUSTIFICACIÓN: Lo que hace realmente es que hasta que no determina que ambos están libres no coge ninguno, y cuando están ambos libres coge ambos a la vez.*	F
8. En *LimitedPhilo* (resolución de interbloqueos limitando el número de filósofos sentados a la mesa) se debe añadir una nueva etapa en la gestión del recurso compartido que hacen los filósofos y consiste en pedir permiso antes de sentarse a la mesa (*enter*) y avisar de que se va a salir (*exit*). *JUSTIFICACIÓN: De hecho esa es la única diferencia entre un LimitedPhilo y un Philo*	V