Metodologías, procesos y entornos para sistemas de tiempo real

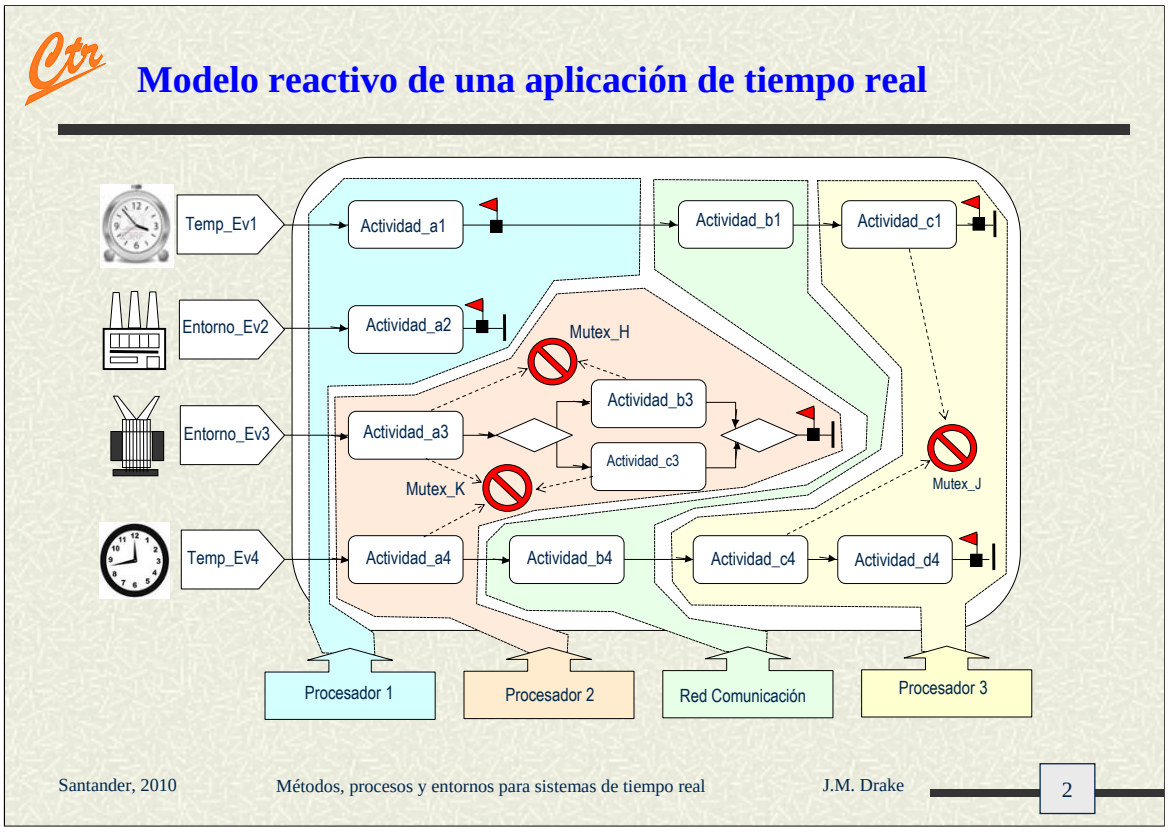
Master de Computación

Diseño de una aplicación basada en objetos de tiempo real

José M. Drake
Computadores y Tiempo Real

Santander, 2010

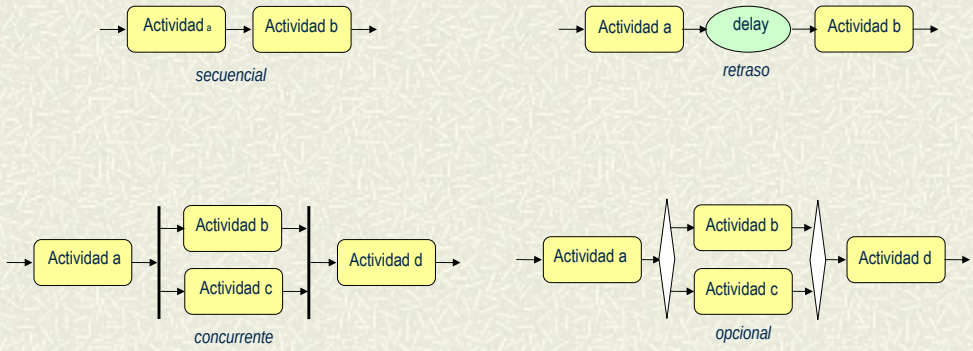
1



Una aplicación de tiempo real se concibe como un conjunto de transacciones (*end_to_end_flow*) que se ejecutan concurrentemente en la plataforma en que se encuentra instalada:

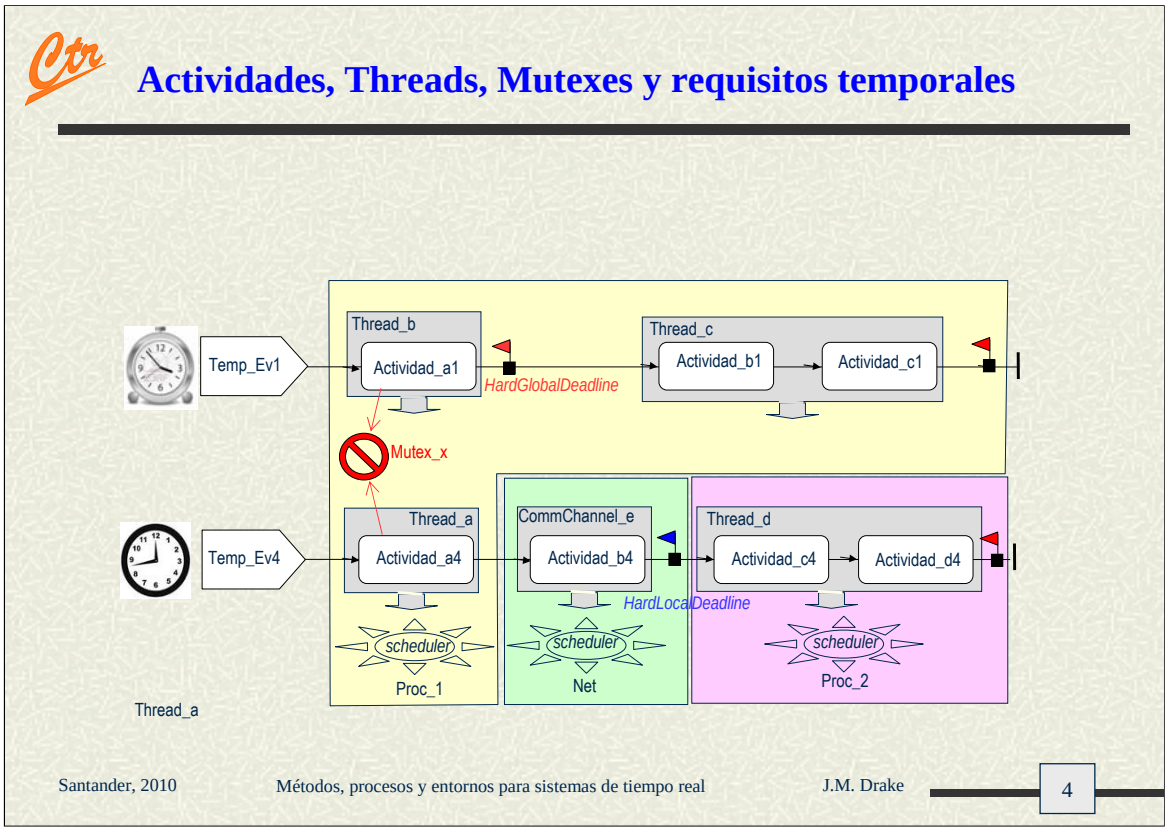
•Cada transacción se inicia en respuesta a un determinado flujo de eventos al que atiende la aplicación. Por cada ocurrencia de un evento se activa la transacción y ejecuta el conjunto de actividades que constituye la respuesta al evento que especifica su funcionalidad. Los eventos pueden proceder de los dispositivos periféricos hardware (evento de entorno), o de un reloj de la plataforma (evento temporizado) que ha sido programado por la propia aplicación para que los genere con una determinada cadencia.

Flujo de control



Santander, 2010 Métodos, procesos y entornos para sistemas de tiempo real J.M. Drake 3

•Cada transacción está constituida por un conjunto de actividades que están relacionadas entre sí por una determinada relación de precedencia o flujo de control. Esto es, cada actividad se inicia o bien en respuesta al evento externo que activa la transacción, o bien como consecuencia de que otra u otras actividades de la misma transacción hayan finalizado. Como se muestra en la figura, las relaciones de flujo de control entre las diferentes actividades de una transacción pueden ser muy variadas, e incluyen entre otras, secuencialidad, concurrencia, alternativas opcionales, sincronización, retrasos, etc.



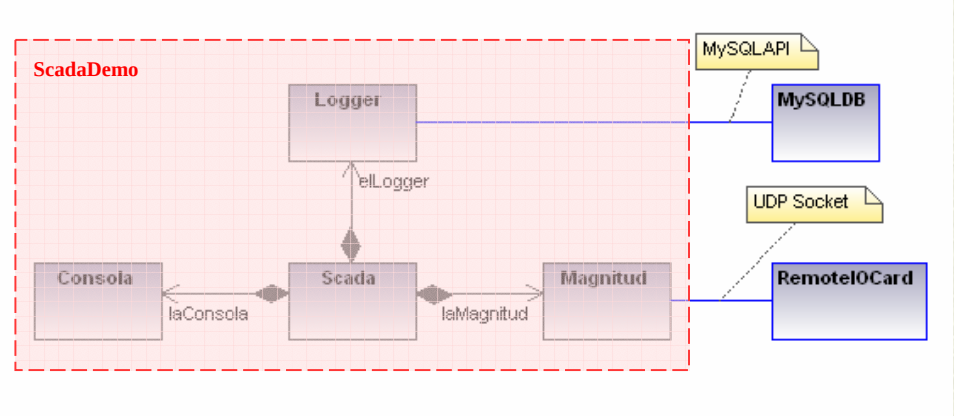
•Una actividad de una transacción consiste en la ejecución de un código en un procesador o la transmisión de un mensaje por una red. Cada actividad requiere para su ejecución un cierto tiempo de procesamiento o transmisión. Este tiempo puede depender del estado de la aplicación cuando se ejecuta y no tiene que ser el mismo en cada activación.
•Una actividad puede requerir, de acuerdo con su naturaleza, ser ejecutada en régimen de exclusión mutua con la ejecución de otras actividades de la aplicación. Esto se implementa en el código de las actividades requiriendo que ambas accedan a un mismo mutex.
•Al tiempo en el que termina una actividad de una transacción, se le pueden asignar requisitos de restricción temporal. Esta restricción temporal puede ser relativa al instante en que se produjo el evento que activó la transacción (requisito temporal global) o al instante en que se ha activado la propia actividad (requisito temporal local). El requisto puede consistir en un plazo, en una limitación de la variabilidad (jitter) o en una tasa de cumplimiento.

Conclusión

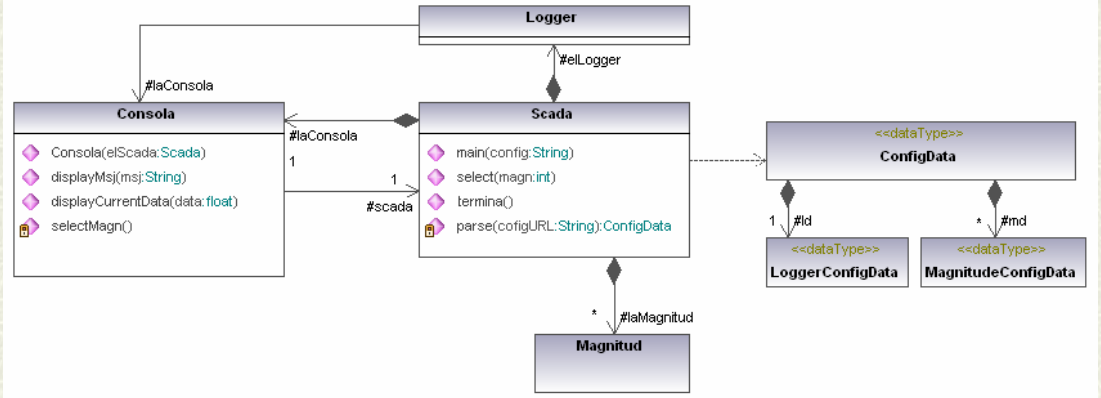
- Una aplicación de tiempo real está constituida por un conjunto de actividades organizadas por relaciones de flujo de control en respuesta a eventos temporizados o procedentes del entorno.
  - La ejecución de las actividades que corresponden a la respuesta a diferentes eventos se realiza concurrentemente y con flujos de control independientes. Entre la ejecución de dos actividades que pertenecen a respuestas de eventos distintos no existen dependencia de flujo de control.
  - Pero si ambas requieren para su ejecución un mismo recurso, existirá entre ellas una relación de exclusión mutua en su ejecución.
  - Los recursos son de dos tipos:
    - Los procesadores o redes de comunicación si las actividades los utilizan para ser ejecutadas,
    - Los mutexes que establecen exclusión mutua por sincronización.
  - La aplicación puede requerir que ciertas actividades de las respuestas finalicen en determinados plazos temporales.
- El diseño de planificabilidad de una aplicación de tiempo real consiste en organizar la ejecución de las actividades de las diferentes transacciones con el suficiente nivel de concurrencia, y dotar a los *threads* en los que se planifican con la adecuada prioridad para que en todos los casos la ejecución de las actividades se realice en el orden adecuado, de forma que todas ellas finalicen antes de los plazos temporales que tienen asignados.

Diseño de una aplicación orientada a objetos

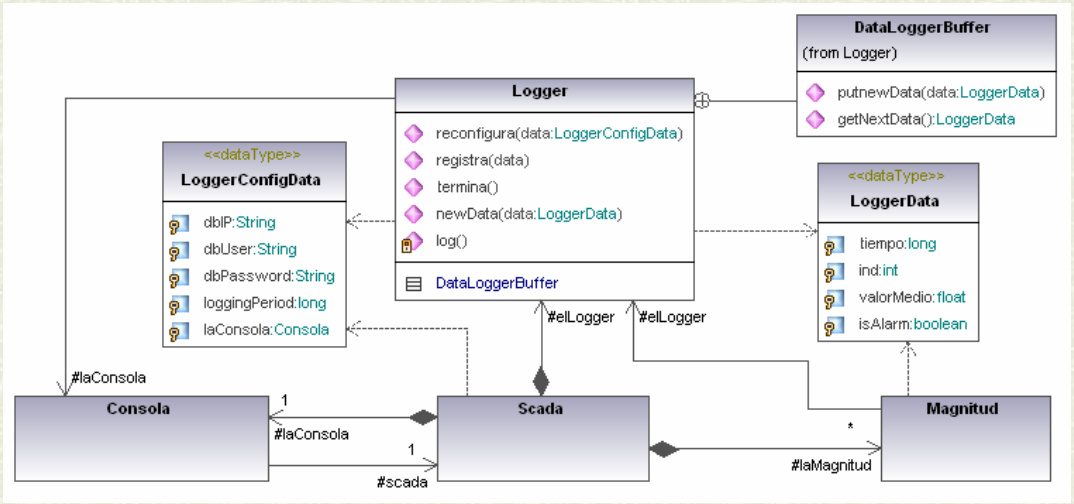
- Una aplicación diseñada utilizando el paradigma de orientación a objetos se diseña en base a clases que se identifican con tipos de objetos que existen en su dominio de aplicación.



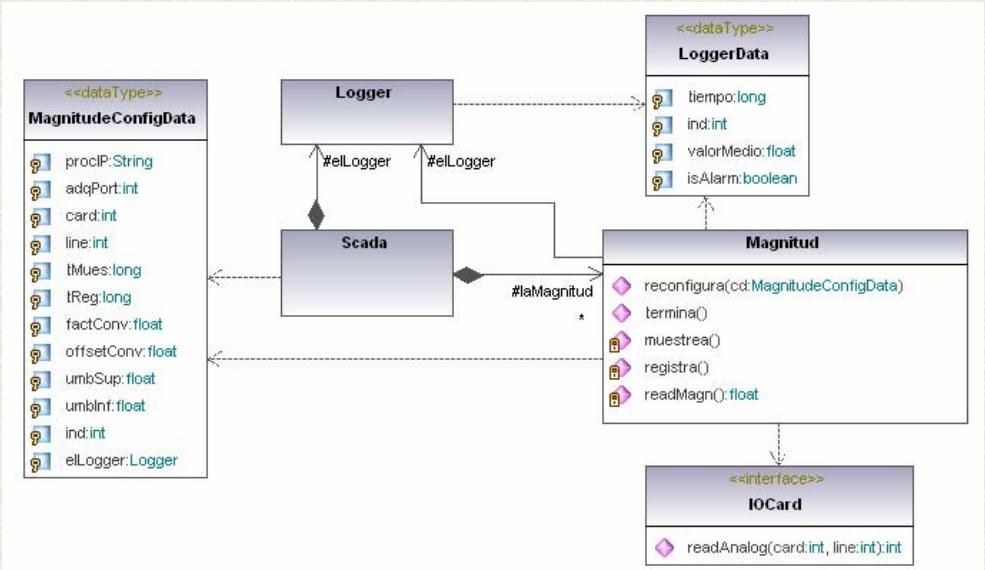
Cada clase se diseña desde el punto de vista funcional (1).



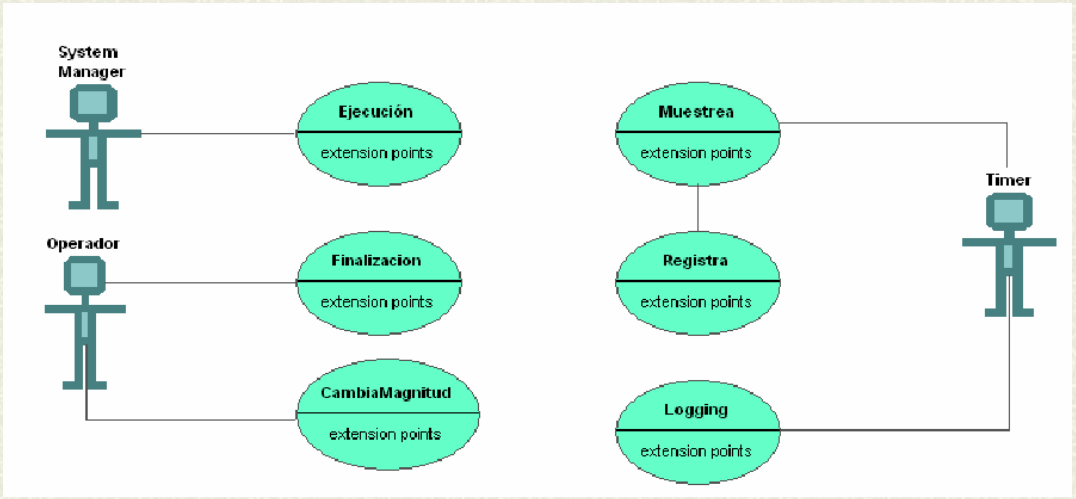
Cada clase se diseña desde el punto de vista funcional (2).



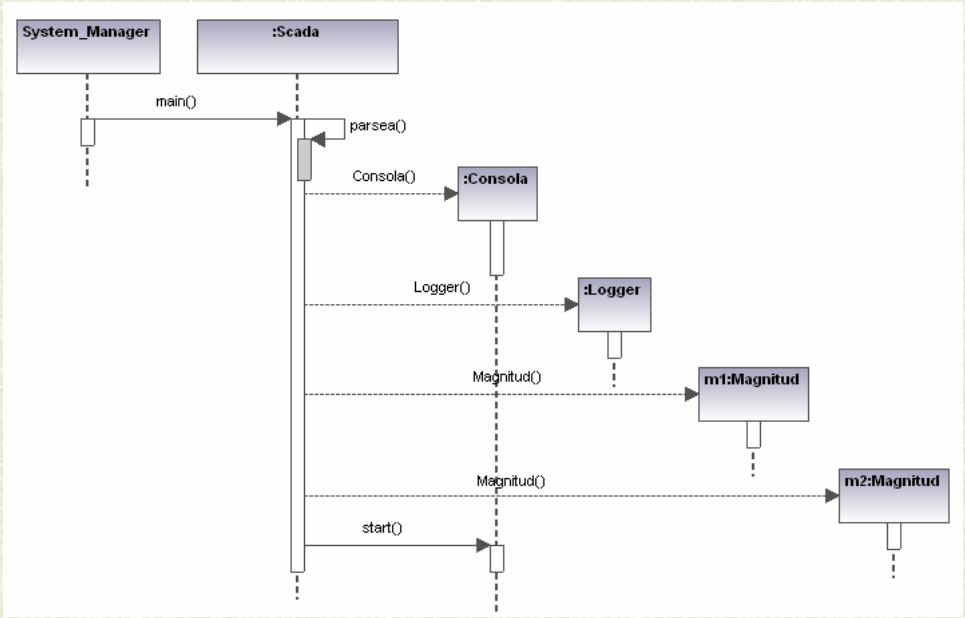
Cada clase se diseña desde el punto de vista funcional (3).

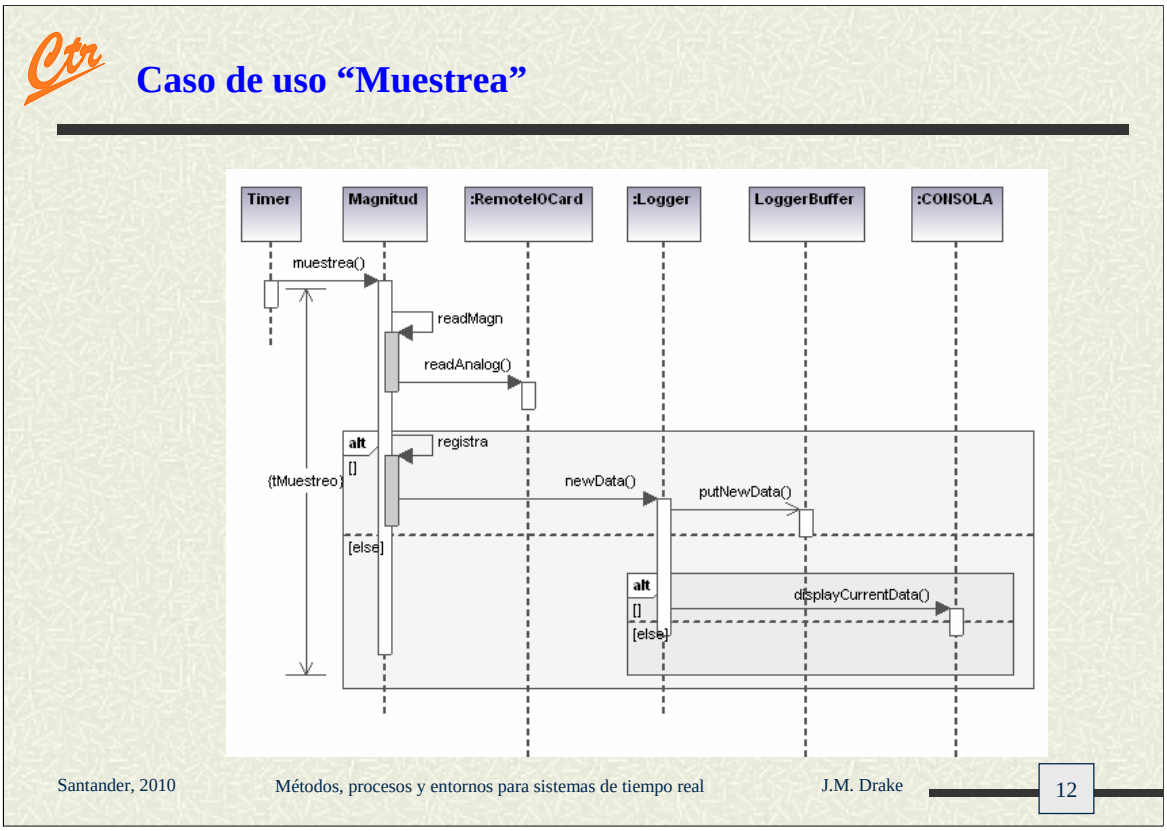


Verificación de la completitud del diseño funcional



Caso de uso "Ejecución"

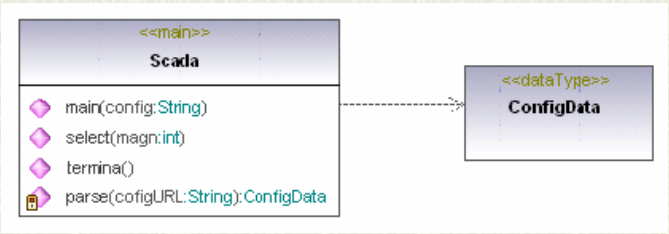




Caso de uso “Muestrea”

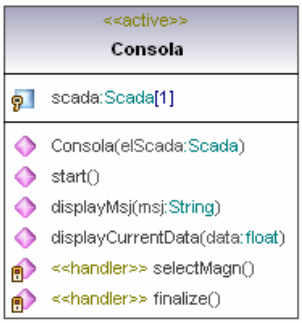
Identificación de la clase que lanza una partición.

- La instanciación de una aplicación resulta de invocar el método estático main() de la clase principal estereotipada como <<*main*>>.
- Hay una clase de este tipo por cada partición.



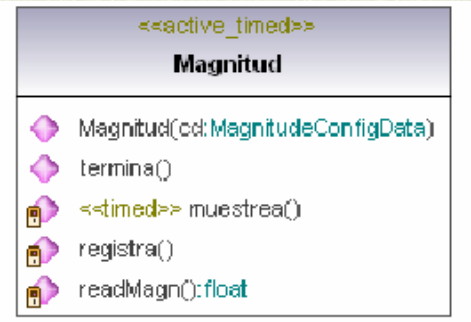
Clase activa que atiende eventos

- Las clases que tienen capacidad de atender a eventos del entorno se estereotipan como *<<active>>*, y deben incluir entre sus métodos privados el método que se invoca cuando el evento ocurre. Estos métodos se estereotipan como *<<handler>> y* se ejecutan en *threads* internos propios de la clase y específicos para cada tipo de evento.
- Cuando la respuesta a un evento dura más que el tiempo entre eventos, o bien se encolan los eventos para su atención secuencial, o se dispone de un grupo de *threads* (*thread pool*) para su atención concurrente. En este último caso los etiquetamos como *<<concurrent_handler>>*, para indicar que cada ocurrencia del evento se planifica en un *thread* independiente.



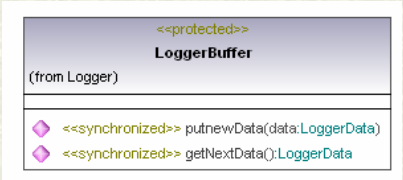
Clases <<active_timed>>

- Las clases que tienen capacidad de programar el reloj, y atender los eventos temporizados que proceden de él, se estereotipan como <<*active_timed*>>, y deben incluir entre sus métodos privados el método que se invoca cuando ocurre el evento temporizado. Estos métodos se ejecutan en *threads* internos propios de la clase, y se estereotipan como <<*timed*>>.



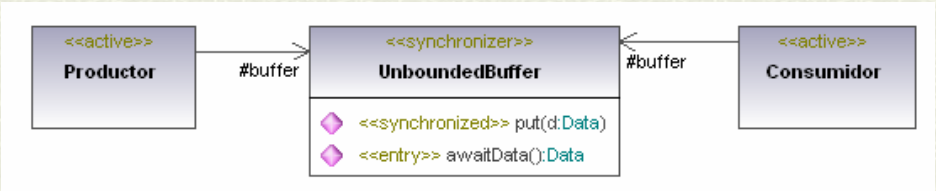
Clases <<protected>>

- Las clases que incluyen un mutex para garantizar que sea seguro el acceso a su estado interno cuando *threads* externos invocan concurrentemente sus métodos, se estereotipan como <<*protected*>>, y los métodos de su interfaz pública que requieren acceder al mutex para ser ejecutados se estereotipan como <<*synchronized*>>.



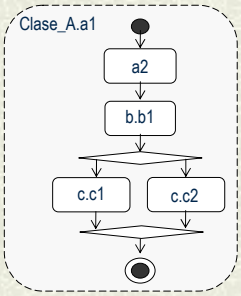
Clases <<synchronizer>>

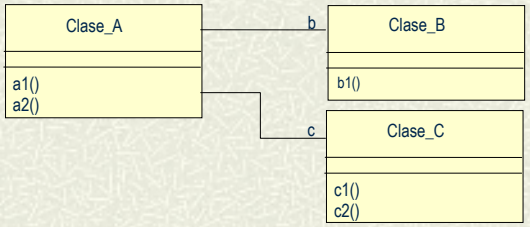
- Las clases protegidas que permiten que un *thread* externo que invoque a ciertos métodos se suspendan en espera a que el objeto alcance un determinado estado se estereotipan como <<*synchronizer*>>. El método público en el que el *thread* puede quedar suspendido los estereotipamos como <<*entry*>>.



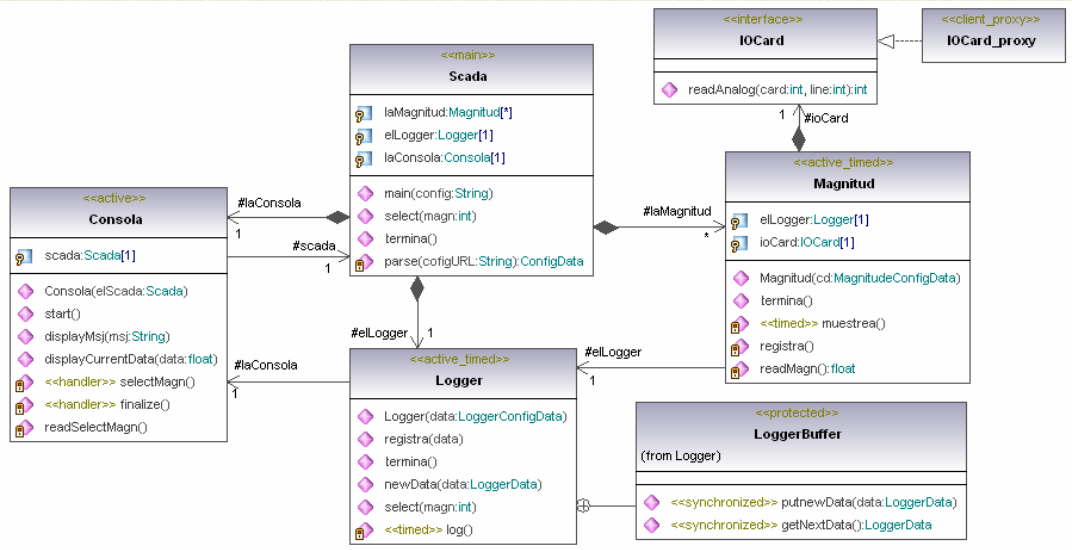
Invocaciones anidadas

- La construcción del modelo reactivo de una aplicación de tiempo real a partir del modelo de clases requiere una información interna más detallada sobre la naturaleza de los métodos privados y públicos que se declaran en las clases:.
  - Mediante un diagrama de actividad, se debe describir la estructura interna de cada operación compleja, esto es que otros métodos de la propia clase, y qué métodos de objetos de otras clases asociadas a ellas son invocados internamente por ella.
  - Se deben explicitar en el diagrama de clases todos los métodos que intervienen en las transacciones de tiempo real, o los usados por ellos.



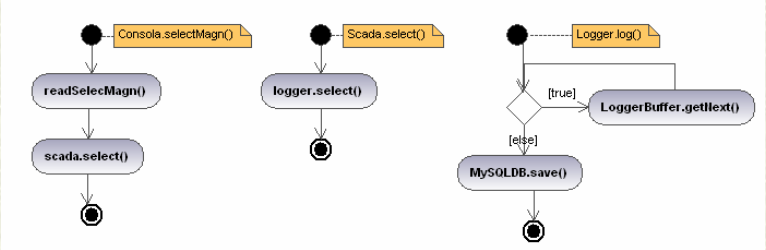


Diseño de la aplicación Scada

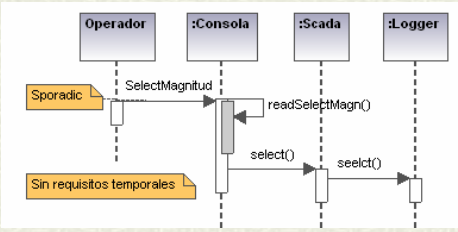


Transacciones sin requisitos de tiempo real

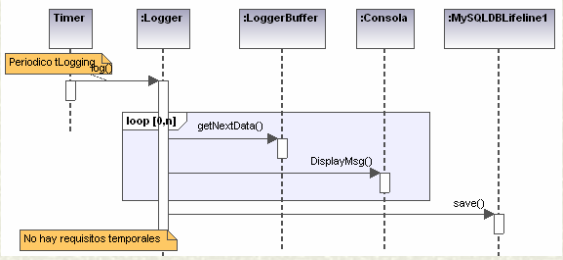
Declaración de estructura interna de métodos



Transacción SelectMagnitud

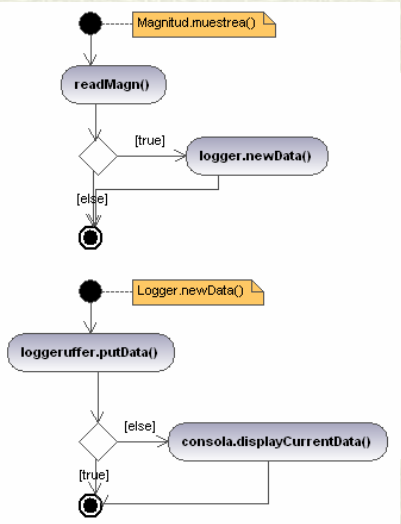


Transacción Logging

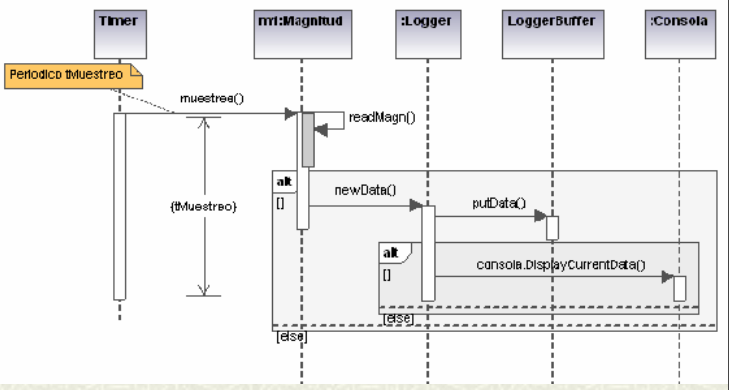


Transacción Muestreo

Declaración de operaciones

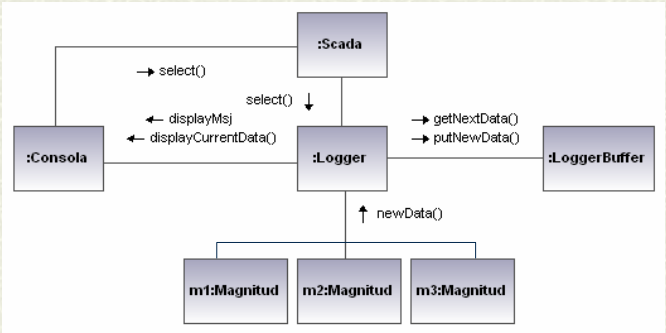


Transaccion m1:Muestreo



Situación de tiempo real y transacciones

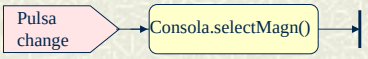
Implementación



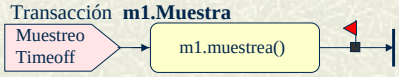
Transacciones

Transacción **LoggerLogging**

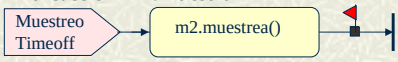
Transacción **Consola.SelectMagnitud**



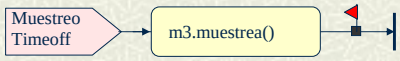
Transacción **m1.Muestra**



Transacción **m2.Muestra**



Transacción **m3.Muestra**



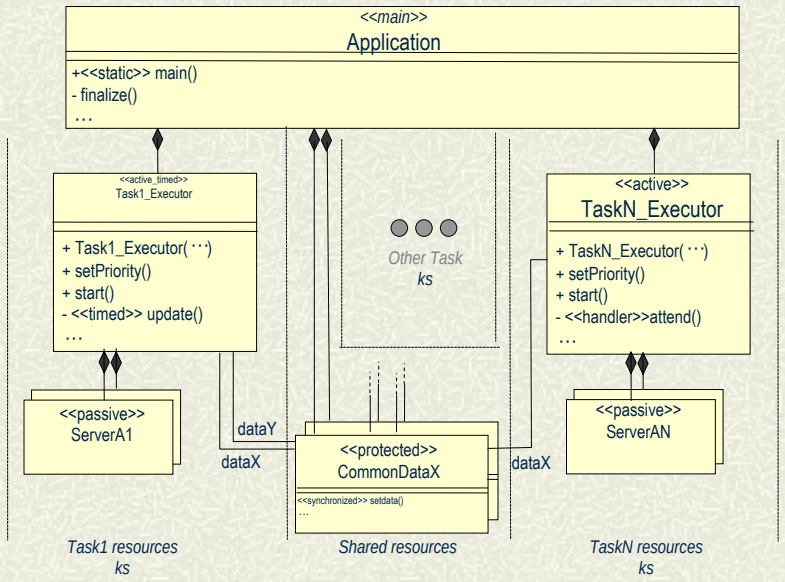
Casos de referencia de aplicaciones de tiempo real

- Aplicaciones con requisitos simples de diseño que forman parte de muchas aplicaciones de tiempo real complejas.
- Su objetivo es tener una guía para identificar y diseñar aplicaciones complejas en los que se presenten los casos contemplados:
  - Tareas concurrentes con información compartida.
  - Tarea con múltiples requisitos temporales.
  - Tarea con eventos de activación aperiódicos.
  - Tarea de tiempo real que comunica con partición de no tiempo real.
  - Tarea asíncrona activada por evento hardware.

Tareas concurrentes con información compartida

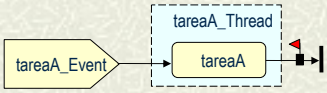
- Su función es implementar un conjunto de flujos independientes iniciados por eventos con un patrón propio.
- La aplicación se ejecuta en un único procesador.
- Cada tareas es la respuesta del sistema a los eventos de entrada, que debe ser ejecutadas dentro de un plazo determinado. Estos plazos son siempre inferiores al intervalo mínimo entre eventos del evento que las activa.
- Las tareas intercambian información a través de estructuras de datos compartidas, a las cuales acceden asíncronamente para leer o escribir, pero utilizando mutexes que garantizan que sea segura la actualización concurrente de la información.

Tareas concurrentes con información compartida(2)

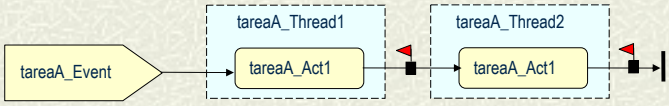


Tareas con múltples requisitos temporales

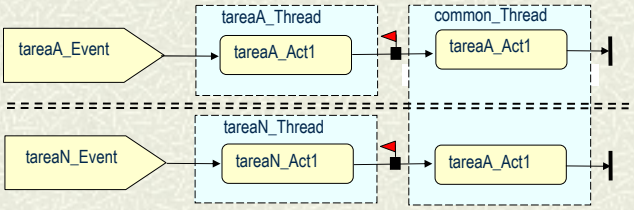
- Diseño de una tarea que tienen restricciones temporales diferentes en algunas de sus fases de ejecución.
- Ejecutar todas las fases de la rspuesta con un único *thread* y con un único parámetro de planificación no es adecuado.



(a) Tarea con restricción temporal final planificada en un único thread.

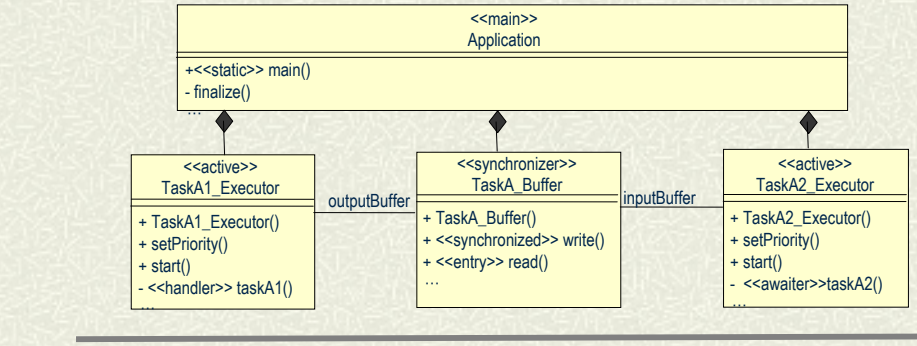


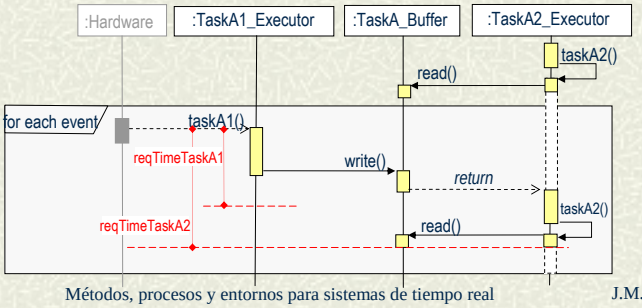
(b) Tarea con restricciones temporales intermedias planificada en múltiples *threads*.



(c) Tareas sin restricción final con thread común de finalización.

Solución Tareas con múltiples requisitos temporales.



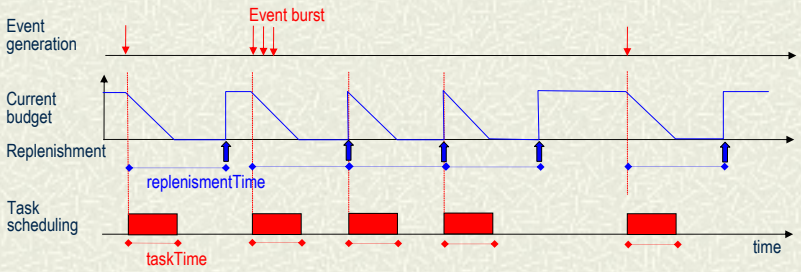


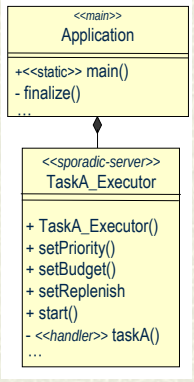
Tarea con eventos de activación aperiódicos

- Una aplicación de tiempo real que ha de responder con requisitos de tiempo real a eventos que no tienen una cota inferior al tiempo entre eventos (aperiódico).
- La aplicación no es planificable, ya que estrictamente puede haber un número ilimitado de eventos en cualquier instante, y por ello, ni la respuesta al evento niotras respuestas que se ejecutan con prioridad más baja que ella son planificables.
- La situación real no es un flujo de eventos con patrón estrictamente aperiódico, sino un patrón de tipo ráfaga, con una acumulación limitada de eventos en un instante (incluso con intervalo entre eventos muy bajo), pero con frecuencia de ocurrencia de eventos finita y limitada.

Servidor esporádico

- La solución en estos casos es la atención de estos eventos con un servidor esporádico, que es un tipo especializado de *thread*, el cual tiene asignado un crédito de tiempo de ejecución específco (*budget*). Cuando la carga de trabajo consume el crédito, se suspende (o baja su prioridad a un nivel tal que su ejecución no afecte a otras respuestas de tiempo real). Posteriormente, transcurrido un tiempo de reposición (*replenishment time*) especificado el presupuesto es repuesto, y la ejecución de la tarea se reanuda.

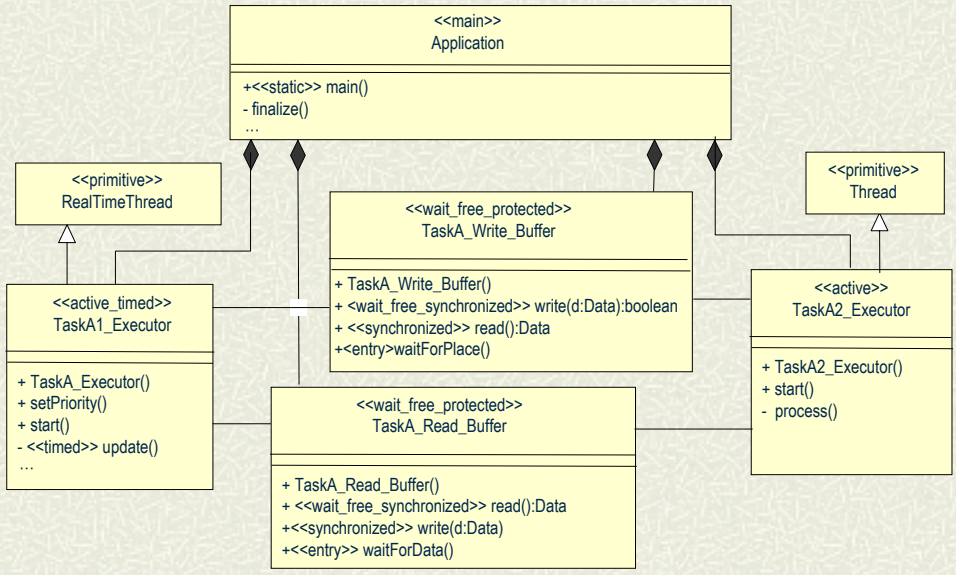




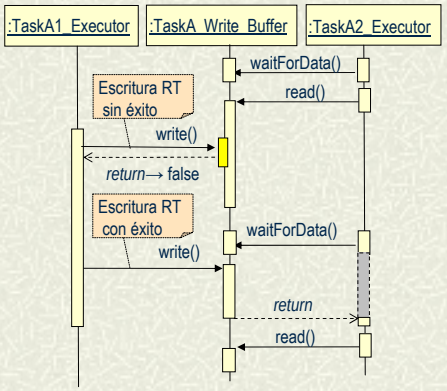
Tarea de tiempo real que comunica con partición de no tiempo real.

- un sistema de tiempo real en el que las transacciones de tiempo real intercambian información con subsistemas que se ejecutan en la misma plataforma y no son de tiempo real, sin que con ello comprometan el cumplimiento de los requisitos temporales que tienen establecidos.
- Una tarea que se planifica en un *thread* de tiempo real se comunica con un *thread* de la partición de no tiempo real. En estos casos no basta un objeto protegido (<<*protected*>>) convencional, ya que cuando el mutex está tomado por un *thread* de no tiempo real, el *thread* está sometido a los tiempos de retraso no limitados.
- Se resuelve con unos objetos protegidos que ofrecen al *thread* de tiempo real métodos sincronizados no bloqueantes. Cuando son invocados por el *thread* de tiempo real, si el mutex está tomado el método retorna señalizando el fallo en el acceso, mientras que si el mutex está libre, es tomado, y el método se ejecuta completamente, y señaliza su éxito.
- Las clases que contienen estos métodos los estereotipamos como <<*wait_free_ protected*>> y los métodos los estereotipamos como <<*wait_free_ synchronized*>>.

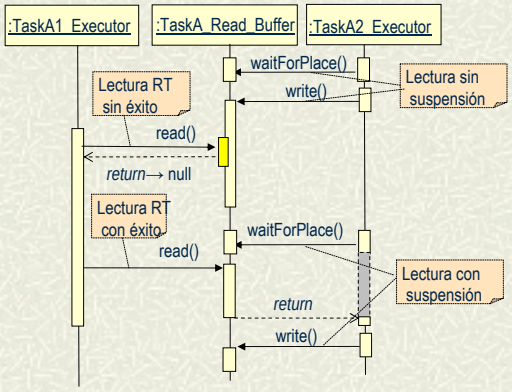
Objetos <<wait_free_protected>>



Comunicación de thread tiempo real con thread no tiempo real.



(a) Escritura de un dato por el RT-Thread

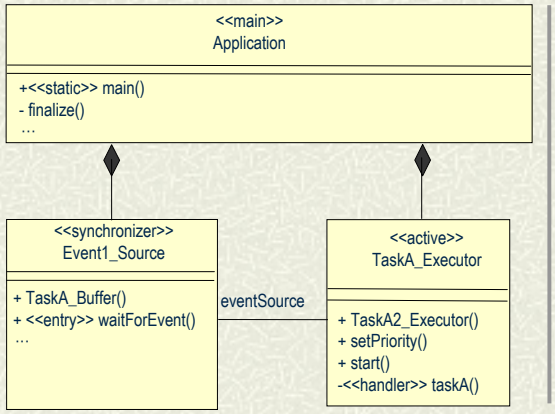


(a) Lectura de un dato por el RT-Thread

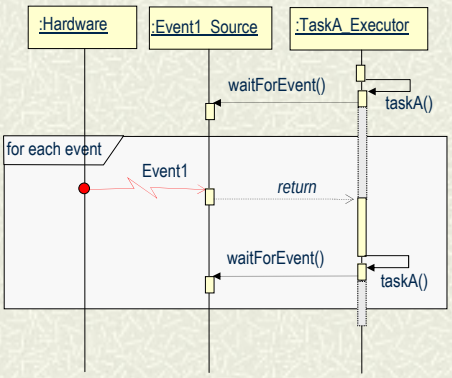
Tarea asíncrona activada por evento hardware

- Un sistema de tiempo real en el que la ejecución de una tarea se activa a través de un evento hardware procedente del entorno.
- Se ha resuelto mediante una clase activa que espera al evento, y que cuando ocurre ejecuta el método privado estereotipado como <<*handler*>>. Aquí se detalla la implementación de la espera al evento con mayor detalle.
- Si el evento que se gestiona es una señal del sistema, no se requiere mayor detalle de la clase activa salvo identificar la señal que se atiende.
- En sistemas de tiempo real, no se utilizan señales del sistema, por:
  - Los eventos del entorno suelen ser gestionan por driver, y los no suelen utilizar señales del sistema para transferir los eventos hardware detectados.
  - No están bien definidos los parámetros de planificación del *thread* que gestionan las señales.
- Se utiliza una estrategia en la que el evento hardware se atiende por un *thread* propio introducido por la aplicación, con parámetros de planificación bien definidos. La interrupción se manifiesta como un cambio de estado de un objeto <<*synchronizer*>>, que permite finalizar un método de tipo <<*entry*>> que el *thread* que va a atender el evento ha invocado previamente y dentro de él ha quedado suspendido en espera al evento.

Atención de un evento del entorno



(a) Diseño.



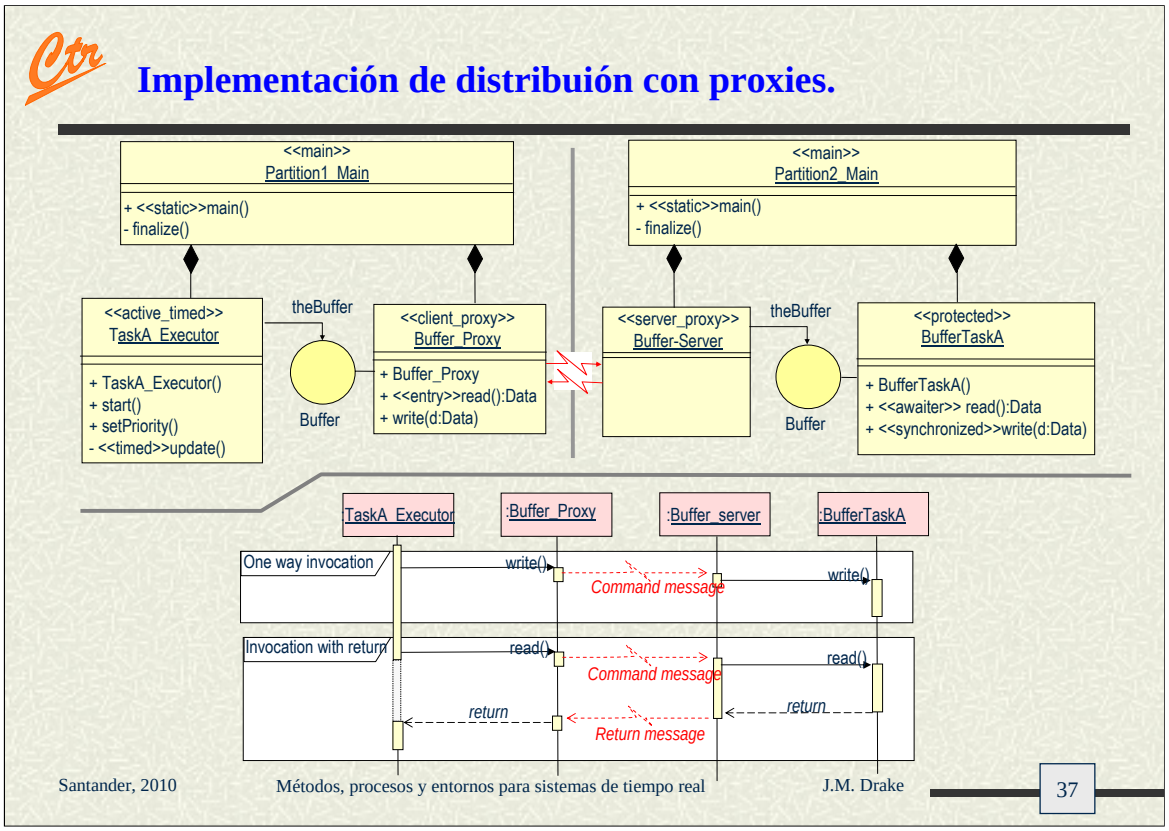
(b) Interacciones en la atención a un evento.

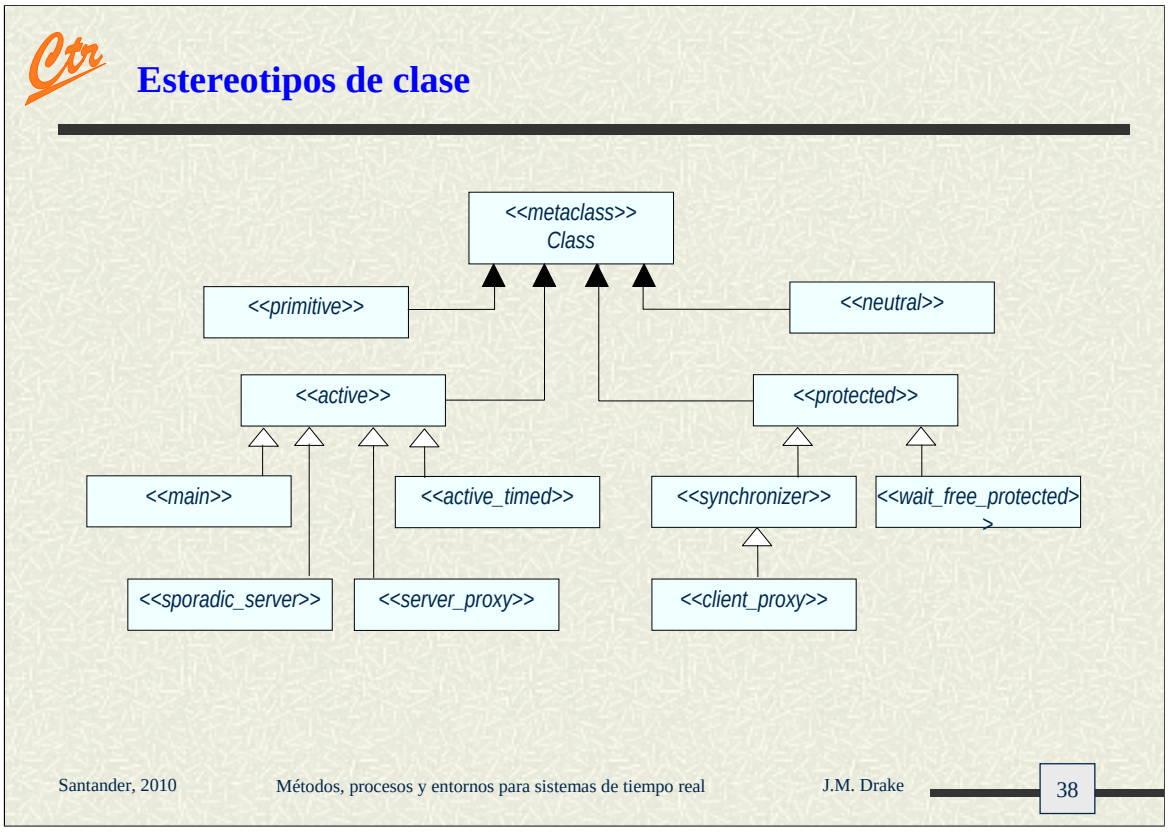
Respuesta distribuida en múltiples procesadores.

- Se consideran aplicaciones que ejecutan las tareas de una transacción en diferentes procesadores de la plataforma, y que transfieren entre ellas el flujo de control por medio de mensajes que intercambian a través de la red de comunicaciones que existe entre los procesadores.
- Razones para la distribución:
  - El entorno sobre el que opera la aplicación está espacial o geográficamente distribuido, y se minimiza el costo de la infraestructura hardware si la plataforma se construye con múltiples procesadores, cada uno de ellos ubicado en la proximidades del punto sobre el que actúa, y una red de comunicaciones entre ellos.
  - La aplicación hace uso de servidores ya desplegados como una infraestructura lógica que es compartida con otras aplicaciones.
  - Se necesita mayor capacidad computacional que la que proporciona un unico nudo, y la aplicación distribuye la ejecución de sus tareas en diferentes nudos a fin de acumular su capacidad y acortar los tiempos de respuesta en base a ejecutar sus tareas con concurrencia física.
  - Secciones de la aplicación están implementados para diferentes tipos de nudo de ejecución, y por tanto, cada sección se despliega en el nudo con el que es compatible.

Patrón de diseño Proxy

- La estrategia de diseño que se propone es el uso de *proxies*. Entre dos objetos que interaccionan haciendo uso del servicio de comunicaciones se incorporan dos objetos *proxys*, cada uno de los cuales representa en cada partición al otro objeto remoto con el que interactúa.
- La pareja de *proxys* resuelven internamente la interacción intercambiando mensajes a través de la red de comunicaciones:
  - El proxy cliente <<*client_proxy*>> es en general un objeto de sincronización con capacidad de suspender el *thread* del objeto cliente que invoca la tarea, y
  - El proxy servidor <<server_proxy>> es un objeto activo con un *thread* interno que realiza la invocación en la partición remota.


Implementación de distribuión con proxies.
<<main>>
Partition1_Main
+ <<static>>main()
- finalize()
<<main>>
Partition2_Main
+ <<static>>main()
- finalize()
<<active_timed>>
TaskA_Executor
+ TaskA_Executor()
+ start()
+ setPriority()
- <<timed>>update()
theBuffer
Buffer
<<client_proxy>>
Buffer_Proxy
+ Buffer_Proxy
+ <<entry>>read():Data
+ write(d:Data)
<<server_proxy>>
Buffer-Server
theBuffer
Buffer
<<protected>>
BufferTaskA
+ BufferTaskA()
+ <<awaiter>> read():Data
+ <<synchronized>>write(d:Data)
TaskA_Executor
:Buffer_Proxy
:Buffer_server
:BufferTaskA
One way invocation
write()
Command message
write()
Invocation with return
read()
Command message
read()
return
return
Return message
Santander, 2010
Métodos, procesos y entornos para sistemas de tiempo real
J.M. Drake
37



<<***active***>>: clase que especifica que cada objeto implementación de ella dispone de un *thread* propio para ejecutar concurrentemente una actividad interna. Es el estereotipo raíz de una familia de estereotipos mas especializados (*<<main>>*, *<<active_timed>>*,*<<sporadic_server>>* o *<<server_Proxy>>*) que proporcionan mas información sobre el origen del thread o de los eventos que atiende la clase.
<<***protected***>>: clase que por cada objeto instancia de ella tiene definido un mutex a fin de garantizar la exclusión mutua en la ejecución de los métodos con capacidad de actualizar concurrentemente su estado interno.
<<***neutral***>>: la ejecución de sus métodos por *threads* concurrentes, no conlleva ningún tipo de interacción entre ellos. Es el estereotipo por defecto, esto es una clase que no tiene estereotipo es de tipo <<neutral>>.
<<***primitive***>>: clase definida en otro contexto que se da por conocido y que por ello la clase no requiere estar especificada.
<<***main***>> (extends <<active>>): clase desde la que se lanza la ejecución de una aplicación. Es una clase activa con un *thread* que recibe del contexto desde la que se ordena la ejecución.
<<***active_timed***>> (*extends <<active>>*): clase activa en la que el *thread* interno atiende eventos temporizados procedentes del reloj del sistema.
<<***sporadic_sever***>> (*extends <<active>>*): clase activa en la que el *thread* interno solo está capacitado para ejecutar un tiempo igual al crédito especificado (*budget*) cada cierto periodo (*replenishment time*).
<***synchronizer***>> (*extends <<protected>>*): clase dotada con un mecanismo de sincronización del tipo variable de condición o semáforo, en el que se pueden suspender *thread* externos que invocan sus métodos en espera de que cambie su estado.
<<***client_proxy***>> (*extends <<sinchronizer>>*): clase protegida que dispone de algún mecanismo de sincronización interno con capacidad de suspender un *thread* y que se utiliza en la partición del cliente de una interacción distribuida entre objetos.
<<***server_proxy***>> (*extends <<active>>*): clase activa que atiende los mensajes del proxy cliente complementario, y que en base al mismo invoca la ejecución de métodos de un objeto de su propia partición.
<<***wait_free_protected***>> (*extends <<protected>>*): clase protegida que ofrece métodos protegidos no bloqueantes que en el caso de que el mutex esté tomado retornan sin suspensión notificando el no éxito de su ejecución, y que si por el contrario, encuentran el mutex libre, lo toman y ejecutan la operación con éxito.

Estereotipos de método

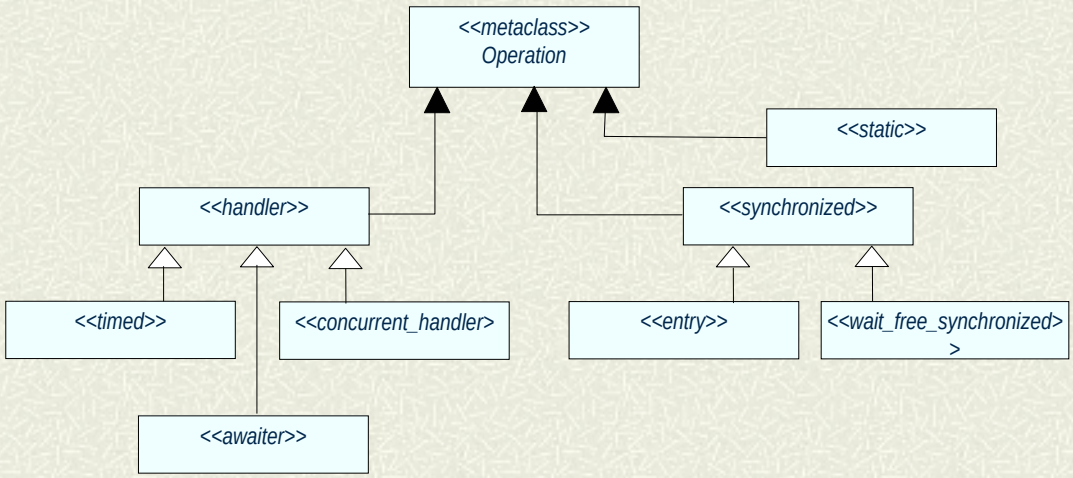


Figura 2.20: Estereotipos de diseño para métodos en aplicaciones de tiempo real.

<<handler>>: método habitualmente privado que es invocado internamente por un evento gestionado por los objetos de las clases *<<active>>*. Es el estereotipo raíz de una familia de estereotipos (*<<timed>>*, *<<awaiter>>*, *<<concurrent_handler>>*) que establecen mayor información sobre el tipo de evento al que responde, o detalles de los modos en que se ejecuta. El método es ejecutado en el *thread* interno del objeto.
<<timed>> (extends *<<handler>>*): método habitualmente privado que es invocado por un evento temporizado programado por el objeto instancia de la clase. El método es ejecutado en el *thread* interno del objeto.
<<concurrent_handler>> (extends *<<handler>>*): método habitualmente privado que es ejecutado concurrentemente en un *thread* interno diferente cada vez que se produce el evento al que responde. La clase activa que lo ofrece dispone de un grupo de *threads* (pool thread) instanciados para atender concurrentemente las sucesivas invocaciones.
<<awaiter>>(extends *<<handler>>*): método habitualmente privado que se suspende en un objeto *<<synchronizer>>* a la espera de que el objeto alcance el estado que habilita su ejecución.
<<synchronized>>: método público de una clase protegida *<<protected>>* que se ejecuta en régimen de exclusión mutua con la ejecución de otros métodos del mismo objeto instancia de la clase que también estén estereotipadas como *<<synchronized>>*. El método toma antes de su ejecución el mutex de la clase, y lo libera al concluir.
<<entry>> (extends *<<synchronized>>*): método público protegido con capacidad de suspender el *thread* externo que lo invoca en espera de que el estado interno del objeto sea el adecuado para ser ejecutado.
<<wait_free_Synchronized>> (extends *<<synchronized>>*): método público protegido no bloqueante. Cuando se invoca y el mutex del objeto está tomado, el método finaliza sin suspenderse indicando que no ha sido ejecutado. Por el contrario, si el mutex se encuentra libre, es tomado y su código es ejecutado en régimen de exclusión mutua.
<<static>>: método definido por clase y no por objeto.