Desarrollo de Software para Sistemas Empotrados

1. Introducción
2. Plataformas para sistemas empotrados
3. Especificación y análisis de requisitos software en sistemas empotrados
4. *Diseño arquitectónico en sistemas empotrados*
5. Implementación software de sistemas empotrados

© Michael González Harbour y J.M. Drake, 8/nov/23

Desarrollo de Software para Sistemas Empotrados

4. Diseño arquitectónico en sistemas empotrados

- 4.1 Introducción
- 4.2 Diseño basado en modelos de control
- 4.3 Lenguajes de descripción de arquitecturas
- 4.4 AADL
- 4.5 Análisis y generación de código
- 4.6 Patrones arquitecturales
- Apéndice A: Patrones para tiempo real

4.1 Introducción

El paso central en el diseño de una arquitectura es el despliegue de las funciones a realizar en grupos de componentes

- con el objetivo de hacer una optimización a nivel de sistema

Los componentes deben tener:

- una alta cohesión interna
- interfaces externas sencillas

El despliegue da lugar a una estructura en la que se intentan compensar

- las prestaciones (las temporales entre ellas)
- los recursos consumidos (memoria, energía, ...)
- la componibilidad, sencillez, mantenibilidad
- la tolerancia a fallos

Diseño arquitectónico

Al final del diseño arquitectónico los requisitos han sido asignados a componentes y las interfaces entre ellos están especificadas de forma precisa tanto en el dominio funcional como en el temporal

- a partir de este momento el esfuerzo de diseño se puede partir en un conjunto de actividades concurrentes, cada una centrada en un componente individual

Algunas de las interfaces de los componentes con el entorno externo se pueden dejar para la fase de diseño detallado

- en particular elementos como la interfaz hombre/máquina, que puede ser compleja

© Michael González Harbour y J.M. Drake, 8/nov/23

Sistemas de seguridad crítica

El diseño de sistemas de seguridad crítica comienza con un análisis de seguridad

- análisis del árbol de fallos o
- análisis de los modos de fallo y sus efectos (FMEA)

Hay diversos estándares para el diseño de sistemas de seguridad crítica

- IEC 61508 para el equipo eléctrico y electrónico
- ARINC DO 178B para el software de equipos de aviónica

Otros aspectos del diseño

Un aspecto importante es el diseño para mantenibilidad

- contribuye a un menor coste total del ciclo de vida

4.2 Diseño basado en modelos de control

Para sistemas de control existen métodos basados en el modelado y simulación de la planta (entorno) y su controlador

Pasos:

1. Identificación y modelado matemático de la dinámica de la planta
    - se comparan los resultados del modelo y de la planta real para validar el modelo
2. Mediante un análisis matemático del modelo de la planta se sintetizan los algoritmos de control
3. Se simulan el modelo de la planta y el controlador con sus algoritmos de control para validar el diseño
    - *SiL* (Software in the Loop): el computador del controlador se simula y el tiempo también es simulado
    - *HiL* (Hardware in the Loop): se utiliza el computador real ejecutando los algoritmos de control y la ejecución es en tiempo real

© Michael González Harbour y J.M. Drake, 8/nov/23

Diseño basado en modelos de control (cont.)

Es posible simular no solo las condiciones operativas normales sino también las situaciones raras, por ejemplo fallos críticos

Una herramienta ampliamente usada en este entorno es MATLAB/ SIMULINK

- capaz de generar el código de los algoritmos de control de forma automática

© Michael González Harbour y J.M. Drake, 8/nov/23

4.3 Lenguajes de descripción de arquitecturas

UML/MARTE: permiten modelar la estructura y el comportamiento, incluyendo aspectos temporales

AADL (Architecture Analysis and Design Language)

- desarrollado por el Software Engineering Institute, Carnegie Mellon University
- estandarizado en 2004 por SAE (Society of Automotive Engineers)

GIOTTO: Lenguaje para diseñar arquitecturas basadas en el paradigma disparado por el tiempo

- se pueden anotar módulos funcionales con atributos temporales
- estos atributos se consideran al mapear los módulos sobre los elementos de la plataforma

© Michael González Harbour y J.M. Drake, 8/nov/23

Lenguajes de descripción de arquitecturas (cont.)

SystemC

- extensión de C++ que permite la simulación de diseños hardware/ software a nivel de arquitectura
- proporciona también una metodología para refinar un diseño hasta llegar al nivel RTL (Register Transfer Level) para el hardware o a un programa C para el software

© Michael González Harbour y J.M. Drake, 8/nov/23

4.4 AADL

Es un lenguaje textual y gráfico para desarrollo dirigido por modelos de sistemas empotrados y de tiempo real

- soportado por herramientas como OSATE, ALISA, IMV, MASIW, ...

Estándar SAE AS5506D

- AADL Versión 2.3, revisada en 2022

Se usa para diseñar y analizar arquitecturas HW y SW

El concepto básico es la noción de componente o bloque, que interactúa con otros mediante interfaces

- Un componente es una unidad software enriquecida con atributos como:
  - requisitos temporales, tiempos de ejecución de peor caso (WCET), huella de memoria, ...
- se pueden agrupar en paquetes

© Michael González Harbour y J.M. Drake, 8/nov/23

Herramienta OSATE

Editor gráfico y textual sincronizados

Página web: http://osate.org/

Descarga:

- https://osate-build.sei.cmu.edu/download/osate/stable/latest/products/

Instalación:

- Basta descomprimir el fichero en una carpeta y ejecutar `osate`

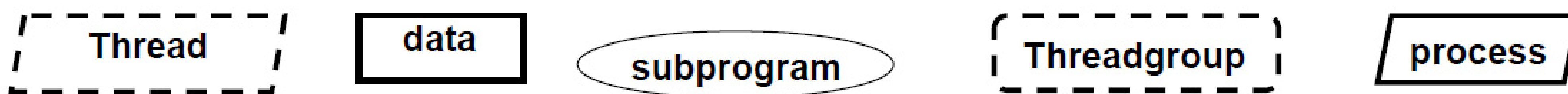
Mediante el plugin *ocarina*, que veremos más adelante

- Capacidad de generar código C o Ada de la arquitectura (esqueletos)
- Capacidad de crear ficheros de configuración de ARINC-653
- Capacidad de generar modelos de análisis Mast y Cheddar

Principales componentes AADL

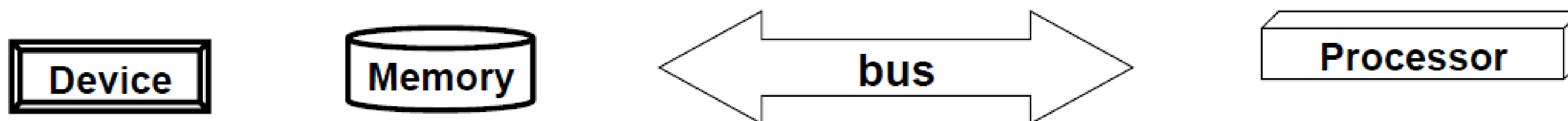
Componentes software

- thread, datos, subprograma, grupo de threads, proceso



Componentes de la plataforma

- memoria, bus, procesador, dispositivo, procesador virtual, bus virtual



Componentes híbridos

- sistema

© Michael González Harbour y J.M. Drake, 8/nov/23

Componentes e implementaciones

Muchos componentes (sistema, proceso, thread,...) tienen dos vistas:

- tipo
  - define las propiedades principales y las interfaces externas
- implementación
  - define los detalles internos

Un tipo de componente puede tener 0 o más implementaciones

- que detallan sus *subcomponentes* así como las *conexiones* entre ellos y con las interfaces externas
- la selección de una implementación permite la configuración del sistema

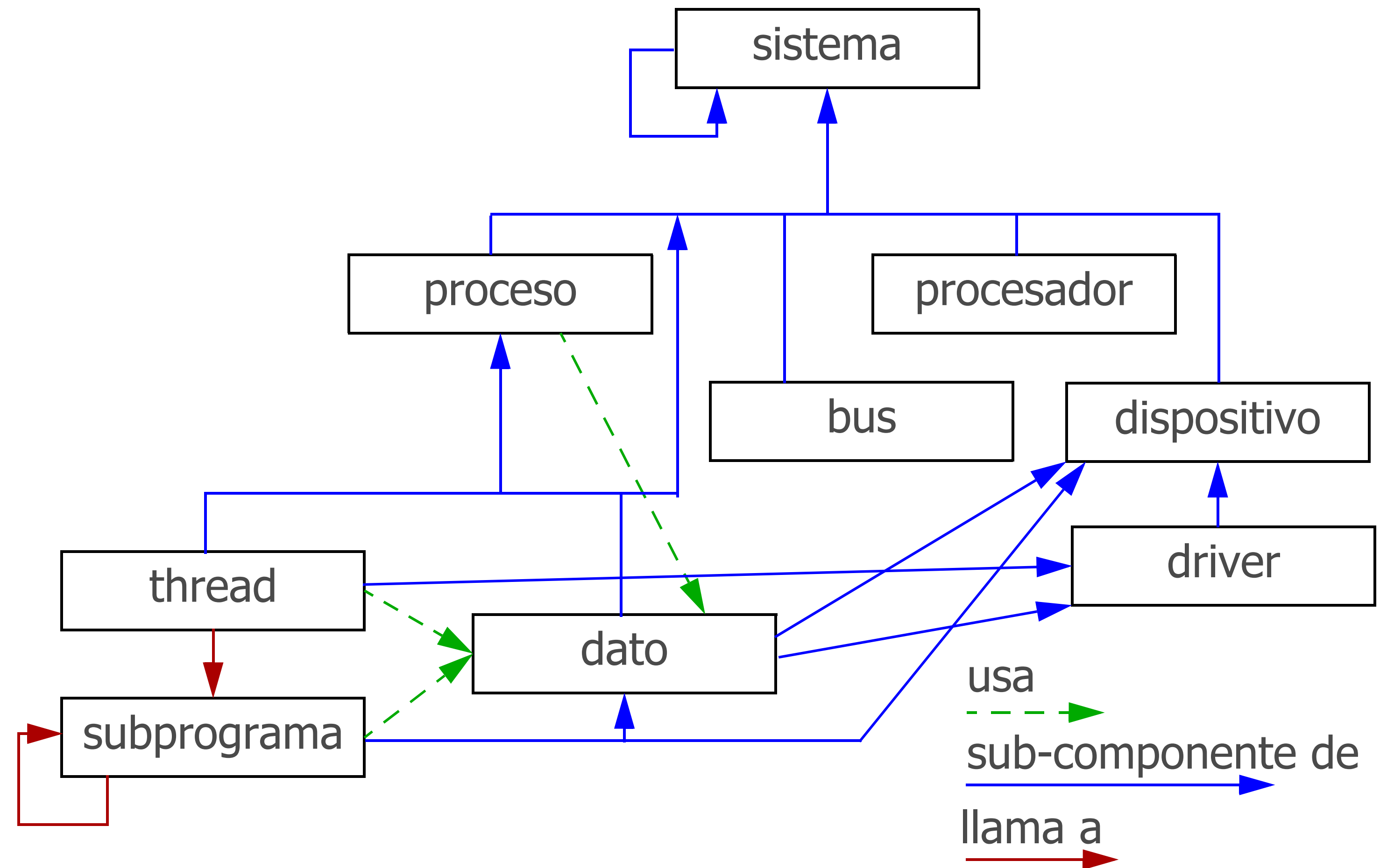
© Michael González Harbour y J.M. Drake, 8/nov/23

Componentes AADL (cont.)

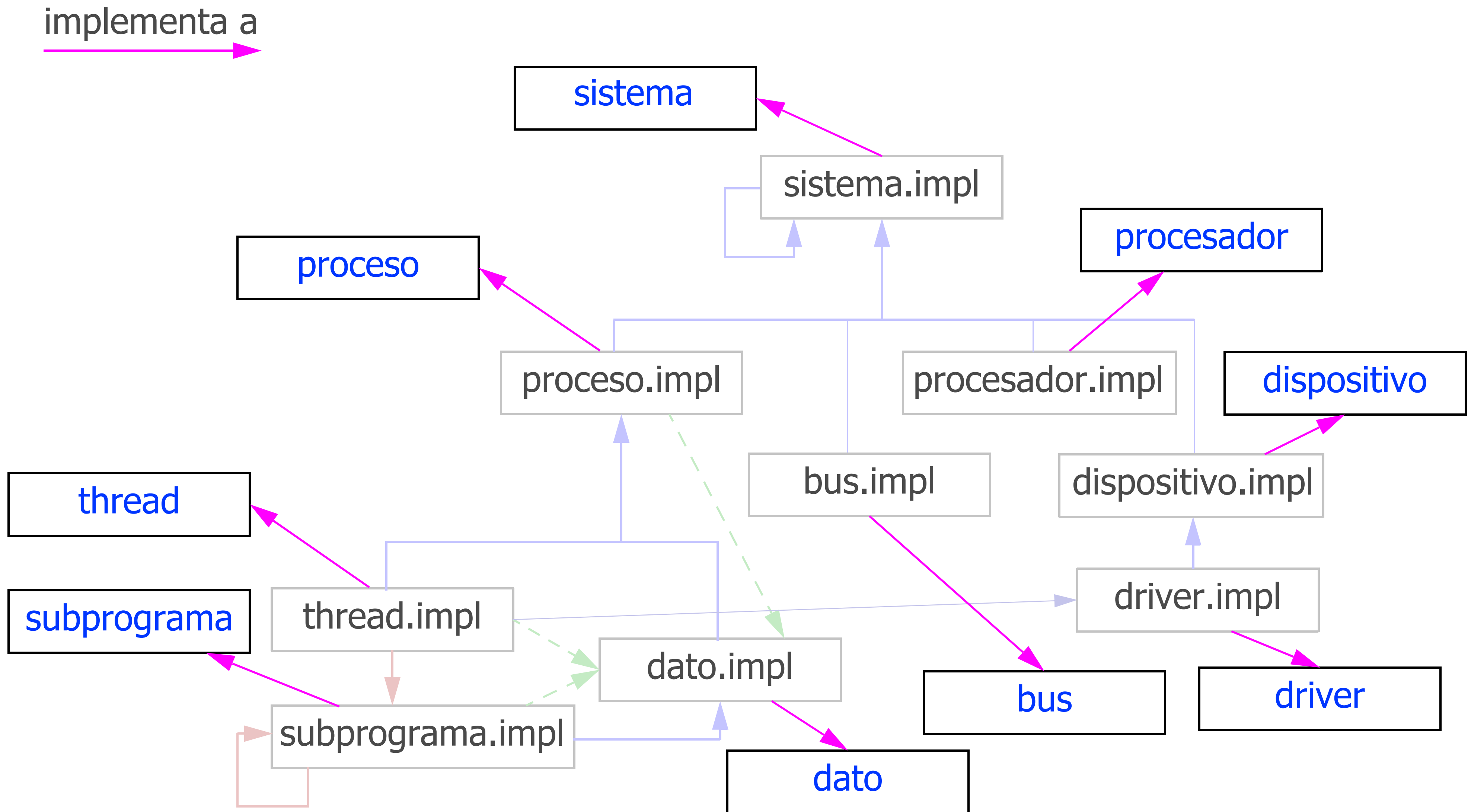
La descripción del sistema es jerárquica

Solo las *implementaciones* de componentes se pueden dividir en sub-componentes

El sistema se instancia creando una jerarquía de componentes desde la raíz (el sistema) a las hojas (subprogramas, ...)



Tipos e implementaciones



 © Michael González Harbour y J.M. Drake, 8/nov/23

Disposición de los componentes

Los componentes y sus implementaciones se suelen definir en el nivel más externo, conocido como *paquete*

- pueden organizarse en varios paquetes

Luego:

- cada tipo define sus interfaces (características o *features*)
- cada implementación define sus subcomponentes y conexiones

© Michael González Harbour y J.M. Drake, 8/nov/23

Interfaces entre componentes

Se llaman características (*features*). Ejemplos:

- *puertos* de comunicaciones para *eventos* y *datos*
- fuentes de *eventos* con o sin datos
- punteros (*access*) a datos, buses o subprogramas
- *parámetros*

Los componentes se comunican conectando sus *features* mediante *conexiones*

- conexiones genéricas entre features
- conexiones entre *puertos*
- entradas/salidas de *eventos* con o sin datos
- conexiones entre punteros (*access*)
- conexiones entre *parámetros*

Una secuencia de conexiones se puede definir como un flujo (*flow*)

Propiedades

Se pueden asociar propiedades a elementos tales como componentes, conexiones o *features*

- son atributos con tipo que representan valores o restricciones
- por ejemplo tiempo de ejecución, frecuencia de un reloj, prioridad, ...

Algunas propiedades están *predefinidas*, pero se pueden crear otras

- las propiedades predefinidas se pueden encontrar en "`Plugin Resources`", en la herramienta *OSATE*
  - Predeclared_Property_Sets

Resumen de propiedades predefinidas de interés en sistemas de tiempo real

Propiedad	Se aplica a
Communication_Properties	
Transmission_Time	bus, ...
Deployment_Properties	
Actual_Processor_Binding	(processor, device, ...) => thread, process, ...
Actual_Connection_Binding	(bus, ...) => feature, connection, ...
Prremptive_Scheduler	processor
Priority_Range	processor
Programming_Properties	
Source_Language	subprogram, ...
Source_Name	subprogram, ...

Propiedad	Se aplica a
Source_Text	subprogram, ...
Device_Driver	device
Thread_Properties	
Dispatch_Protocol	thread, device
POSIX_Scheduling_Policy	thread, ...
Priority	thread, process, device, data, data_access, ...
Concurrency_Control_Protocol	data, data_access
Timing_Properties	
Deadline	thread, process, ...
Compute_Execution_Time	thread, subprogram, ...
Dispatch_Jitter	thread, ...
Dispatch_Offset	thread
Period	thread, process, device, ...
Clock_Jitter	processor, device, system
Thread_Swap_Execution_Time	processor, system

Resumen de otras propiedades

- Requieren un "`with`" del conjunto de propiedades

Propiedad	Se aplica a
Data_Model	
`Data_Representation`	`data, feature`
ARINC653	
`Module_Major_Frame`	`processor, virtual processor`
`Partition_Identifier`	`processor, virtual processor`
`Partition_Name`	`processor, virtual processor`
`Schedule_Window`	`processor, virtual processor`
`Module_Schedule`	`processor, virtual processor`
Deployment (presente al instalar ocarina, en share/ocarina/AADLv2)	
`Execution_Platform`	`processor, virtual processor`

Vistas

AADL tiene una representación textual que contiene todos los detalles del sistema

También tiene una representación gráfica para ofrecer vistas concretas de algunos aspectos

- ocultando detalles internos

AADL también se puede expresar mediante UML/MARTE

Ejemplos

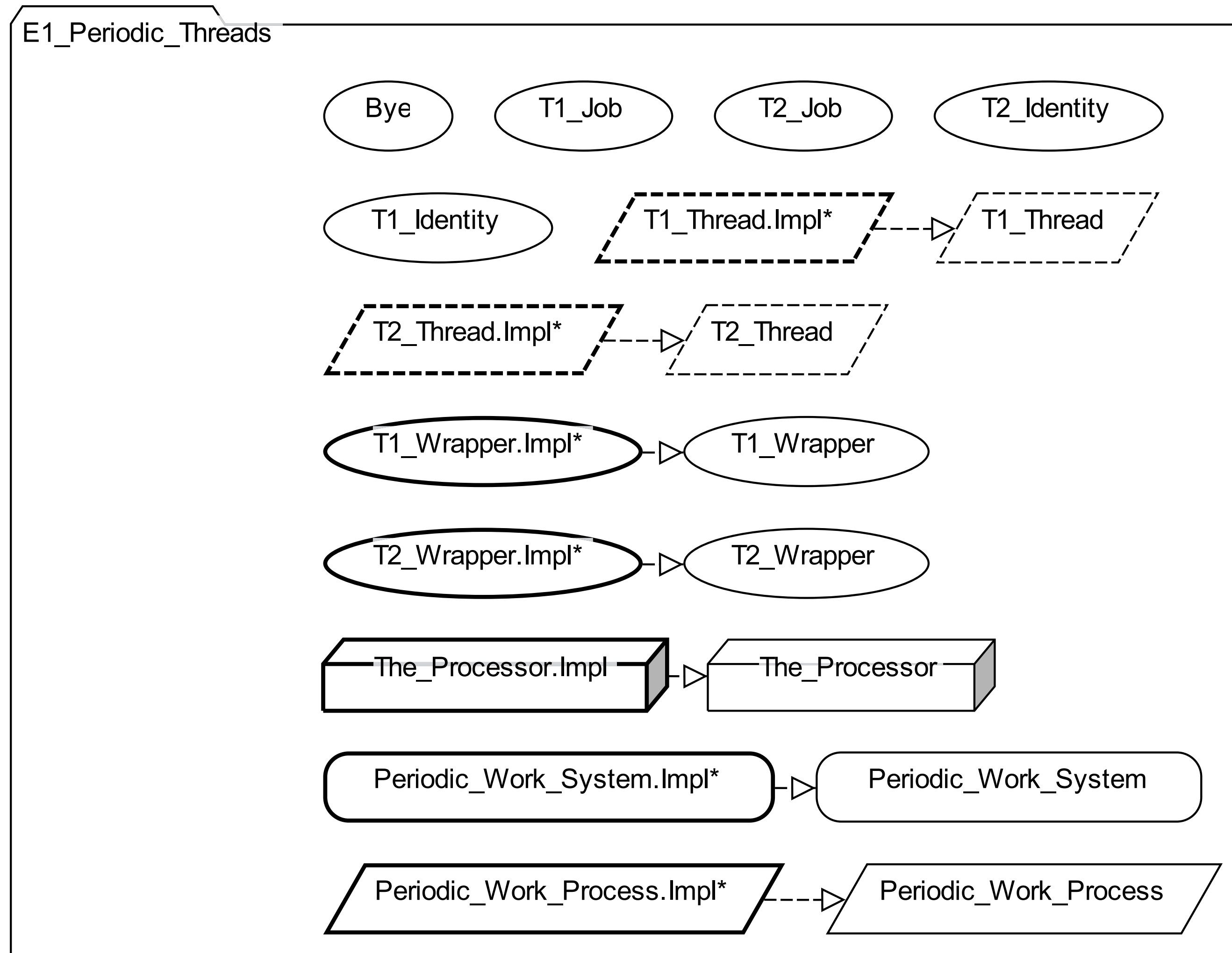
Modelar con OSATE los siguientes ejemplos:

- *Ejemplo 1*: Sistema con dos threads periódicos
- *Ejemplo 2*: Sistema con dos threads que comparten un recurso de datos
- *Ejemplo 3*: Sistema con una rutina de servicio de interrupción
- *Ejemplo 4*: Sistema distribuido con un thread productor y otro consumidor

Ver la descripción textual en los ejemplos del capítulo 4

© Michael González Harbour y J.M. Drake, 8/nov/23

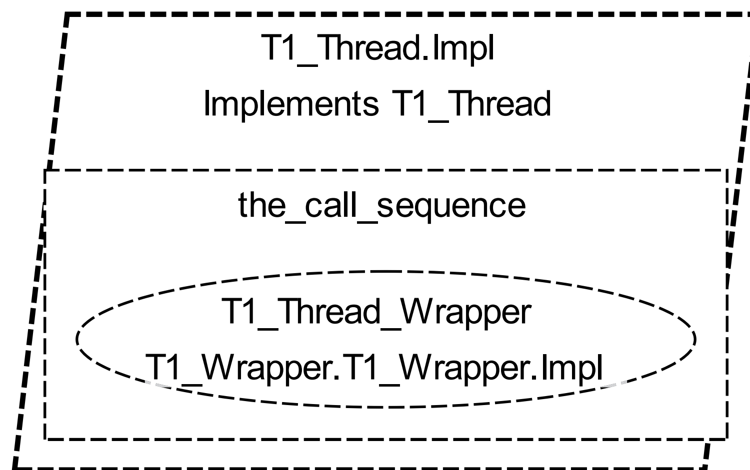
Ejemplo 1: sistema con dos threads periódicos

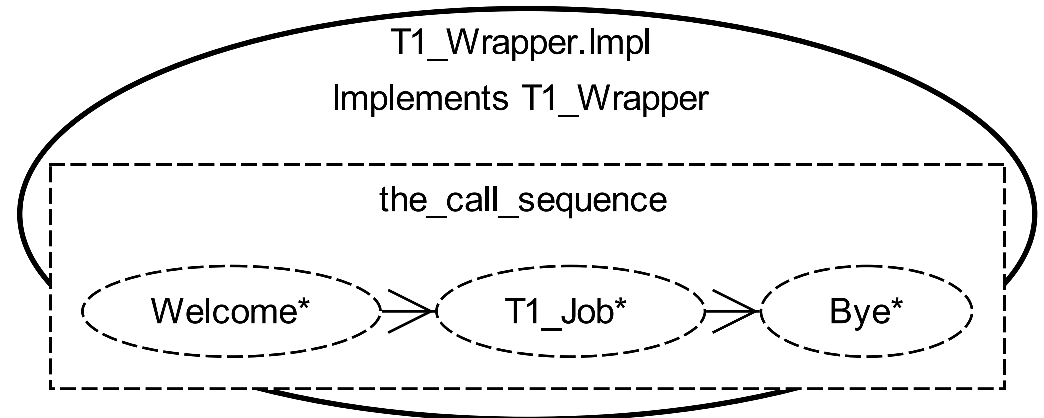


Paquete

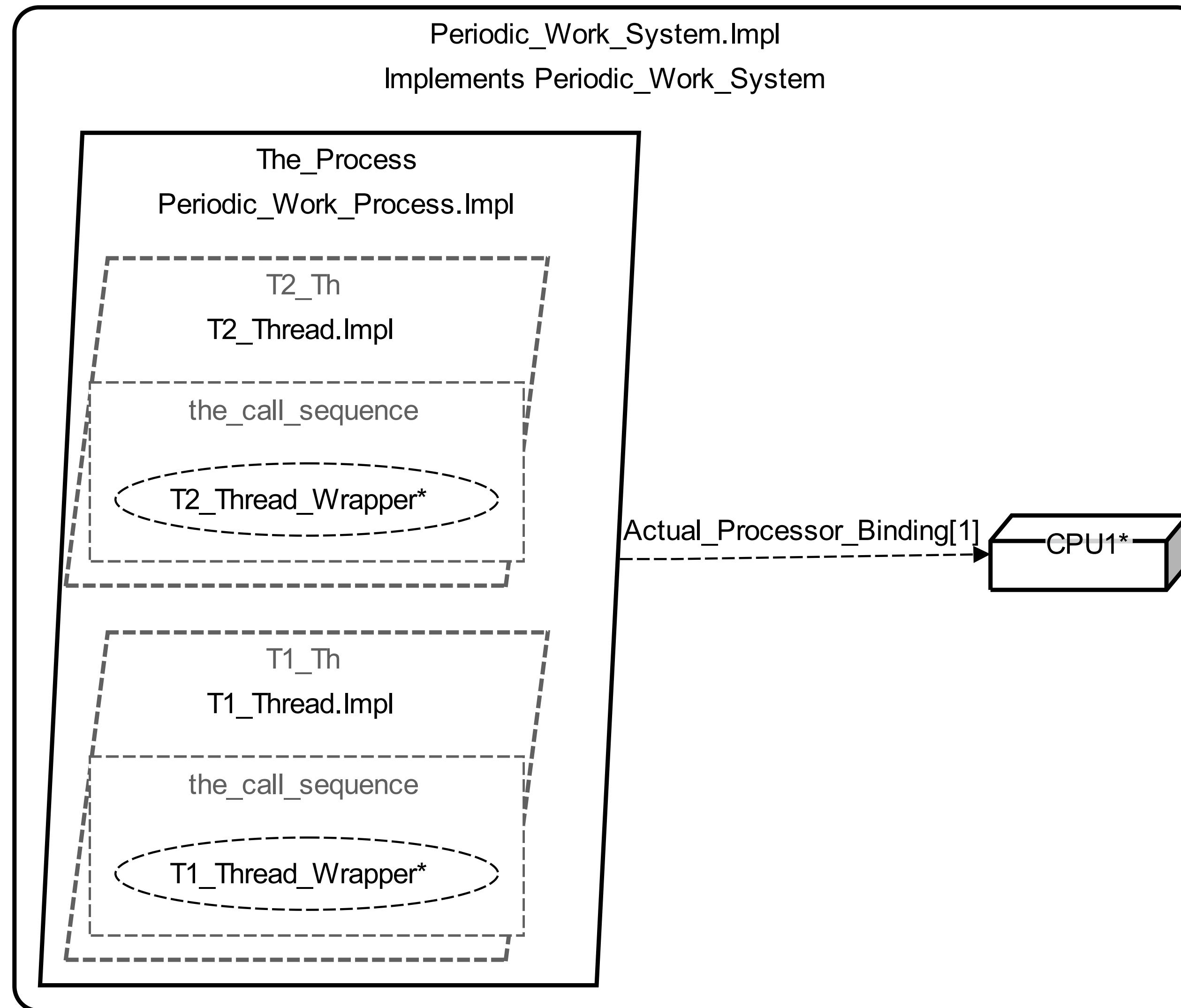
Ejemplo 1: subprogramas

Trabajo del thread 1



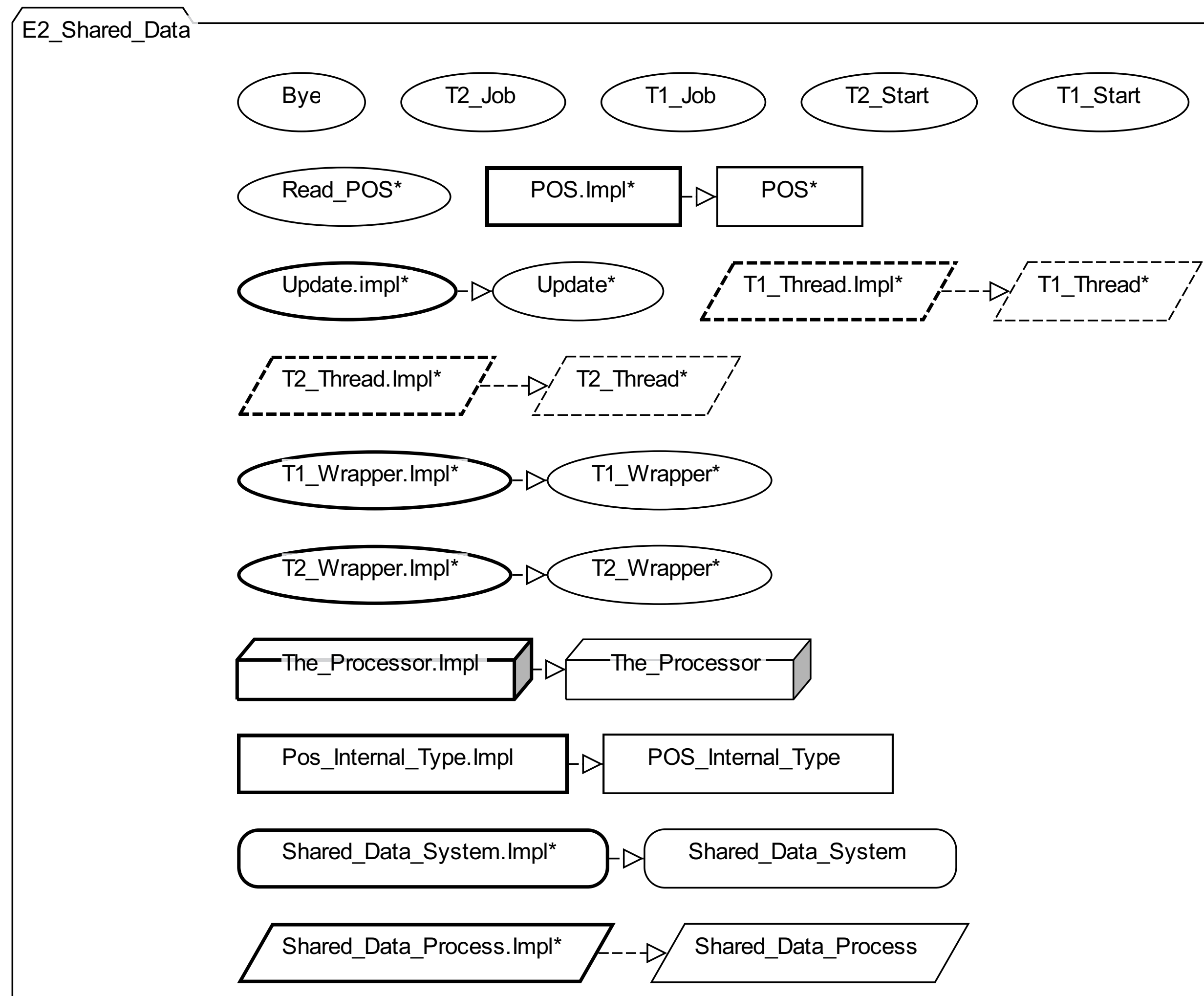


Ejemplo 1: sistema, proceso y threads



© Michael González Harbour y J.M. Drake, 8/nov/23

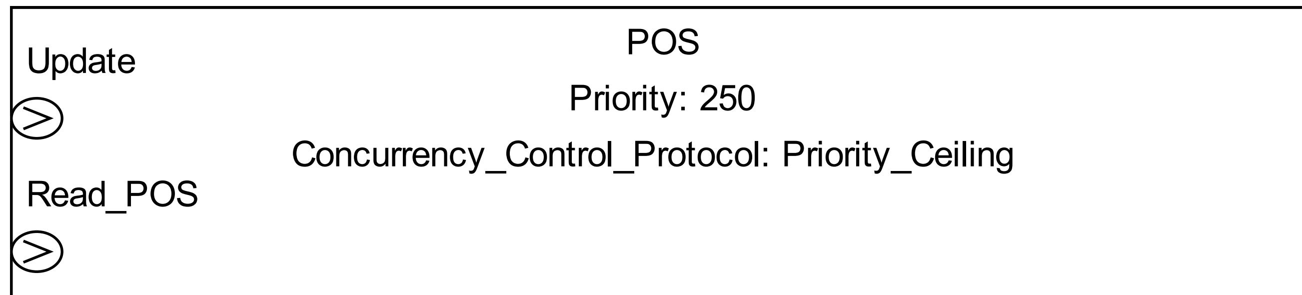
Ejemplo 2: Sistema con dos threads que comparten un recurso de datos



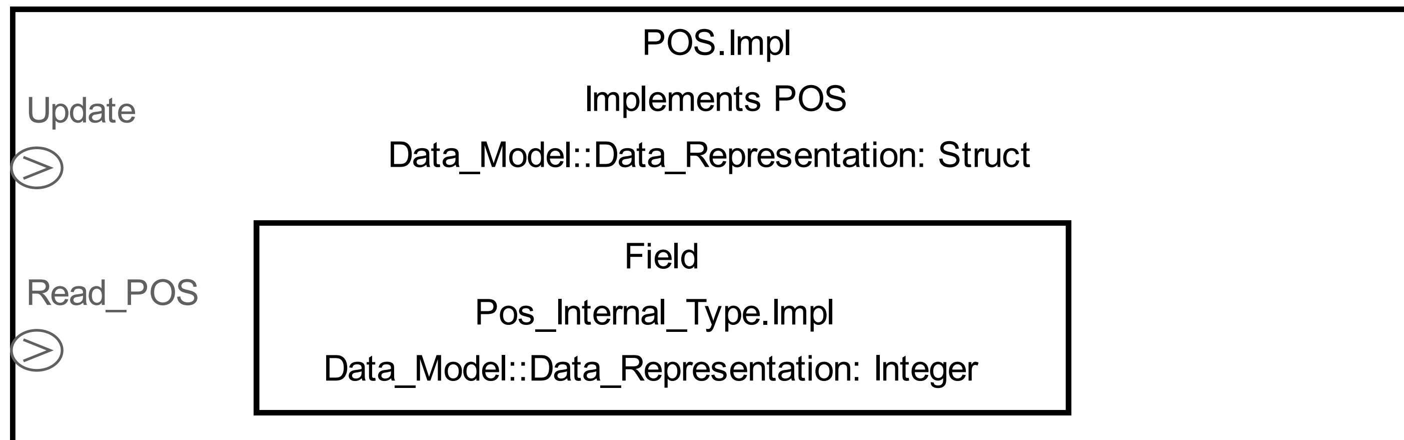
Paquete

© Michael González Harbour y J.M. Drake, 8/nov/23

Ejemplo 2: dato compartido



POS

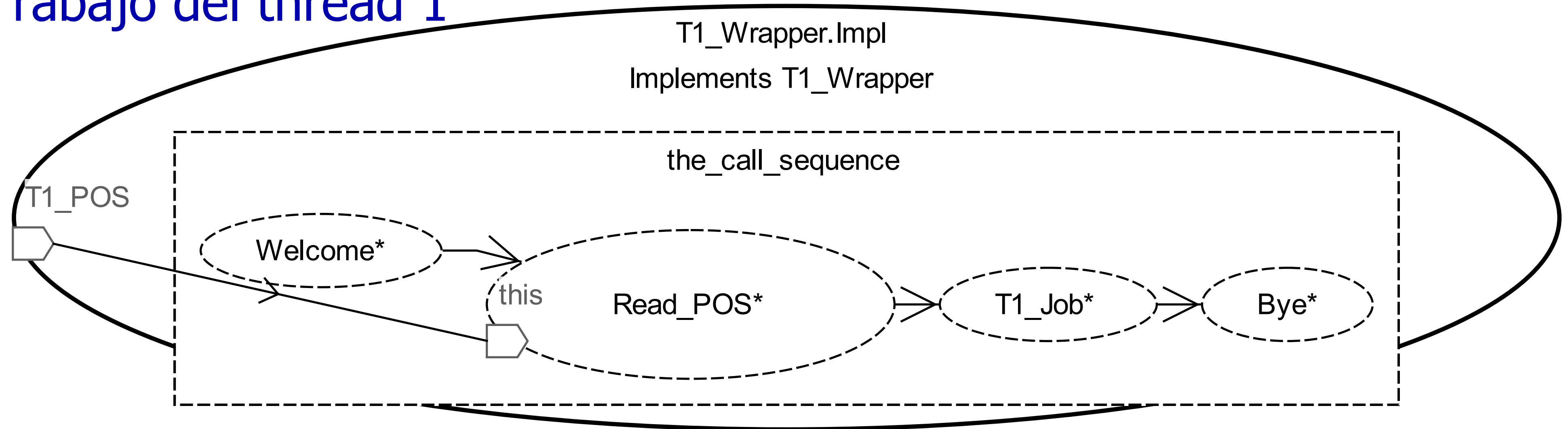


POS.Impl

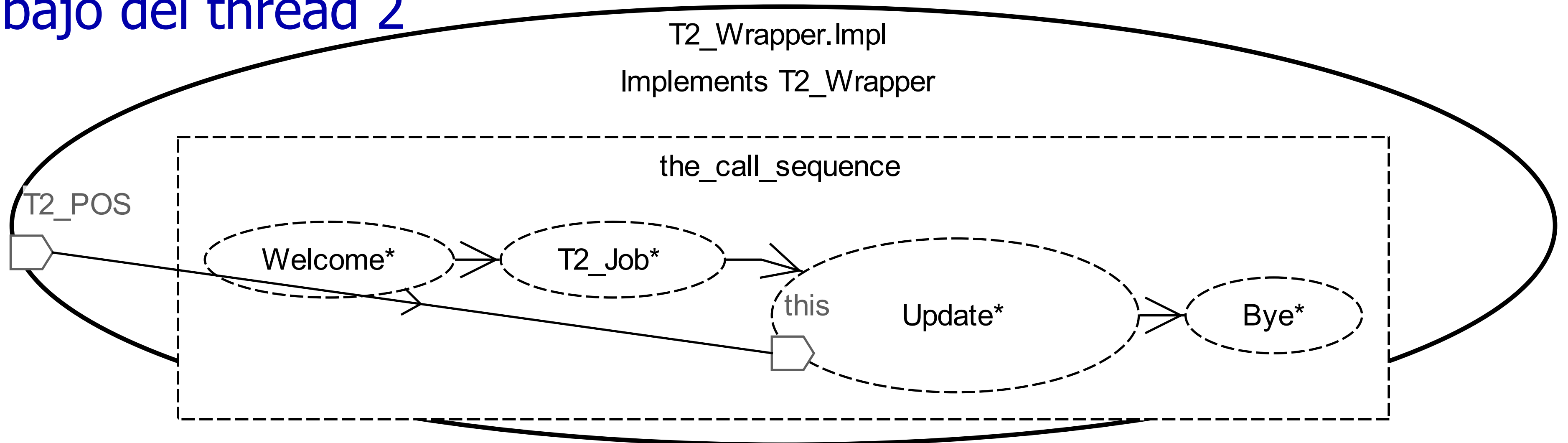
© Michael González Harbour y J.M. Drake, 8/nov/23

Ejemplo 2: subprogramas

Trabajo del thread 1

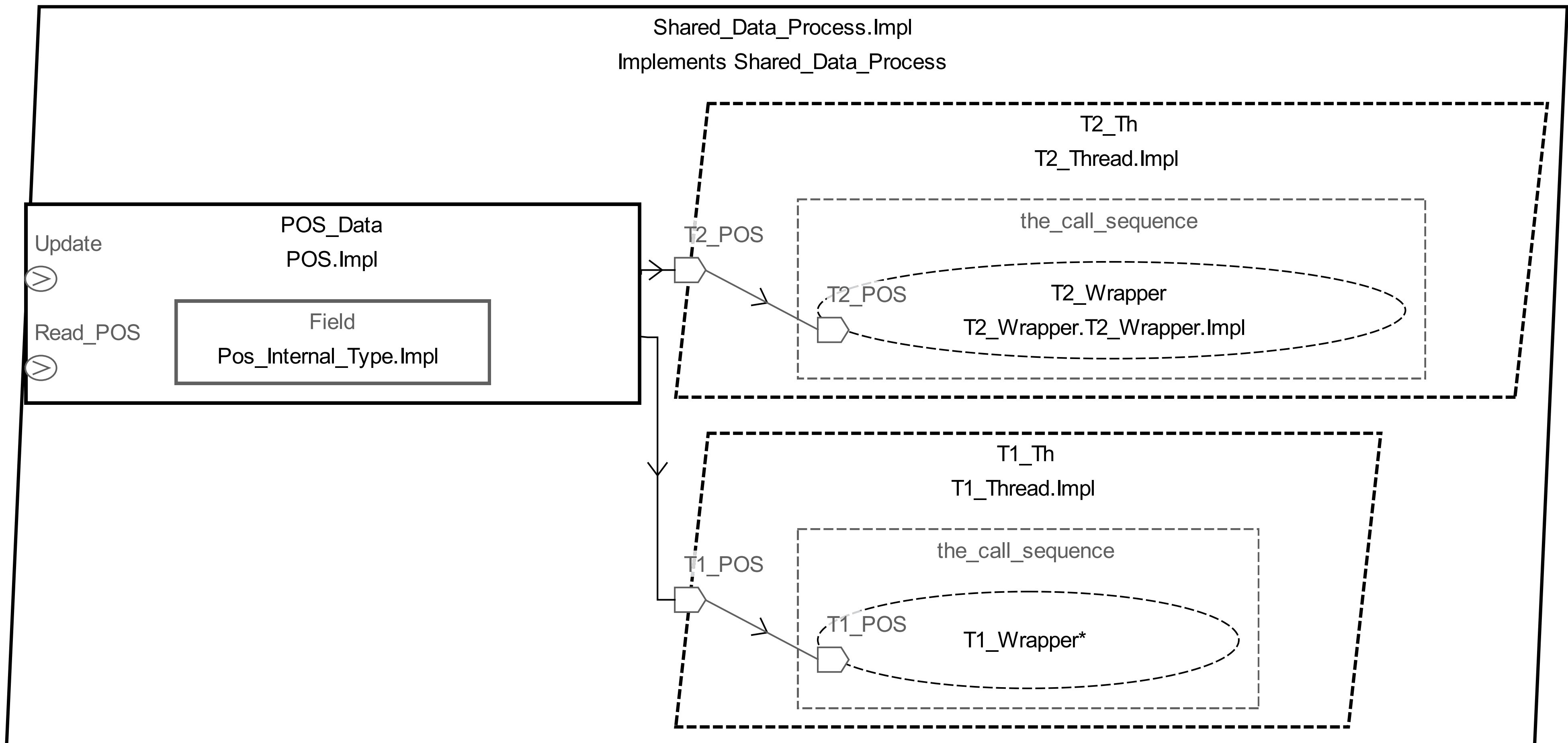


Trabajo del thread 2

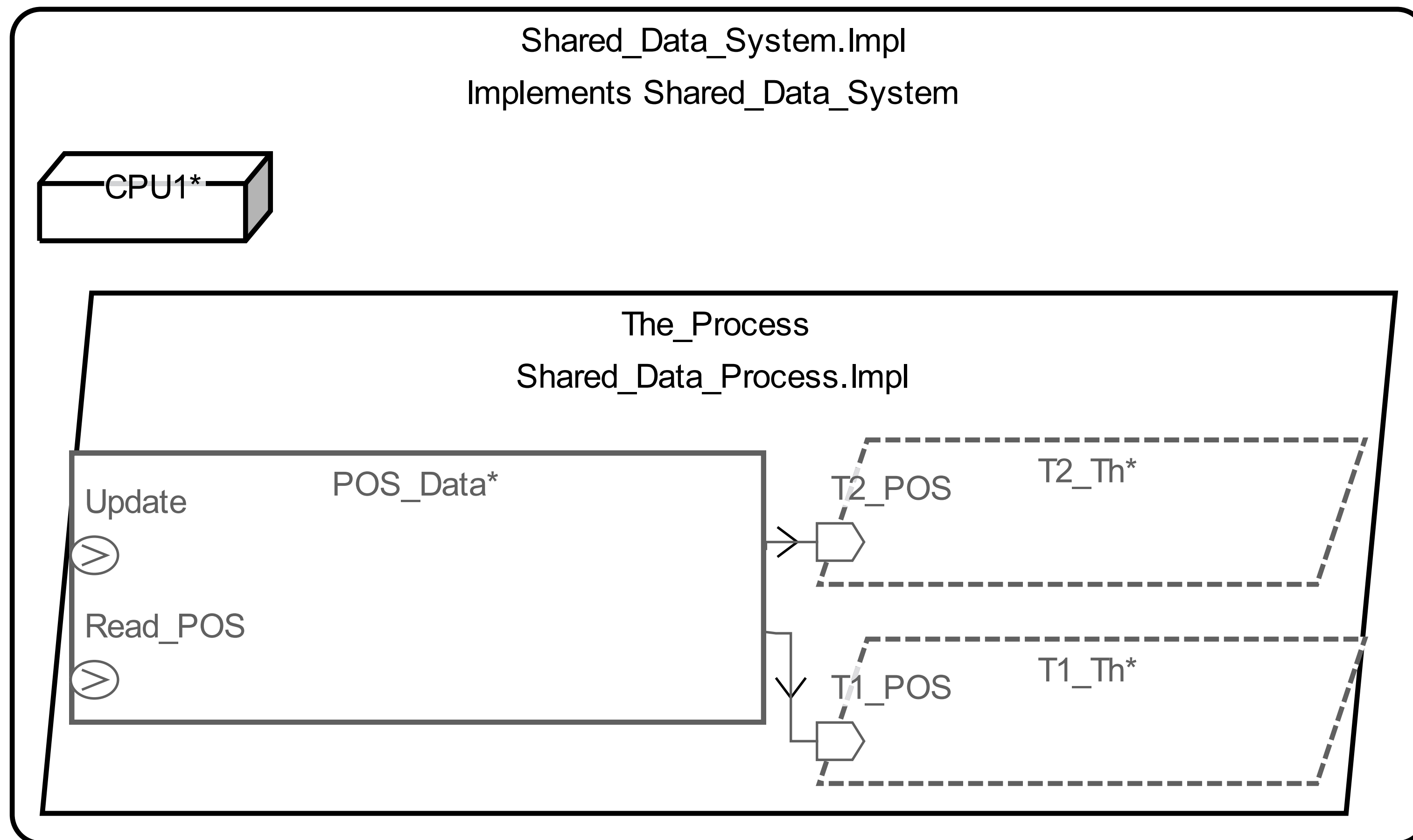


© Michael González Harbour y J.M. Drake, 8/nov/23

Ejemplo 2: proceso con dato y threads

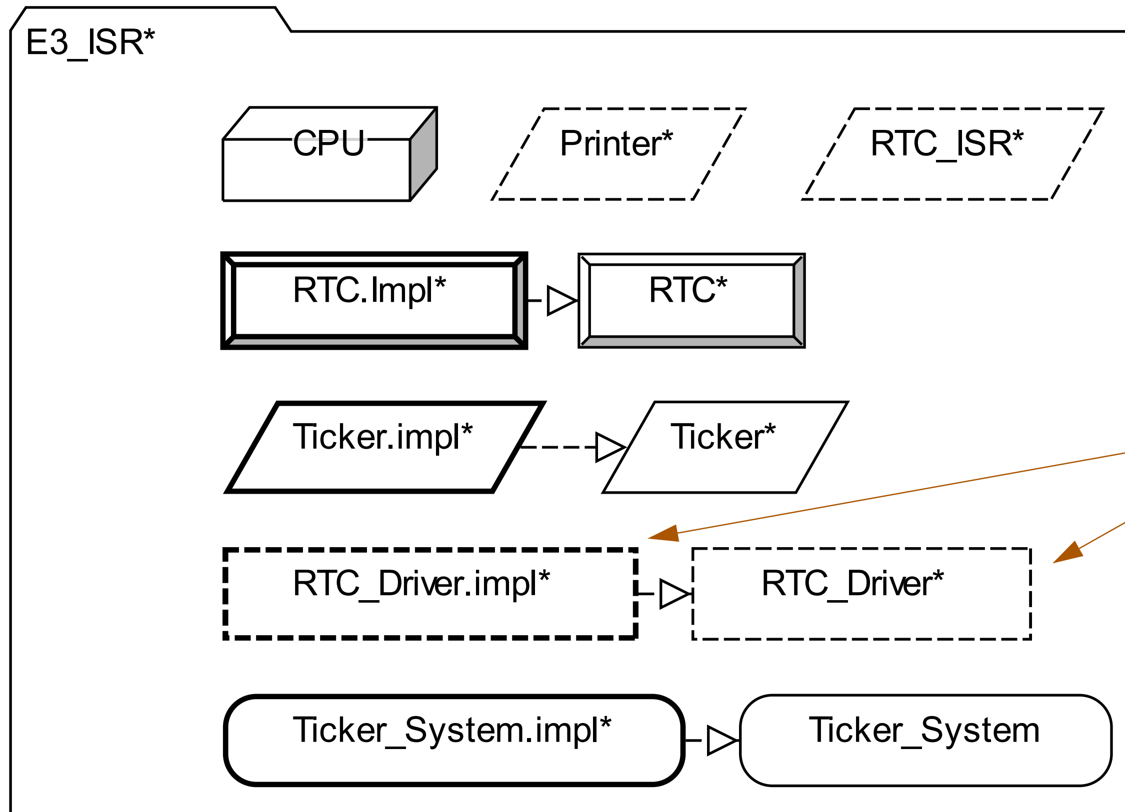


Ejemplo 2: sistema



Ejemplo 3: Sistema con una rutina de servicio de interrupción

Paquete

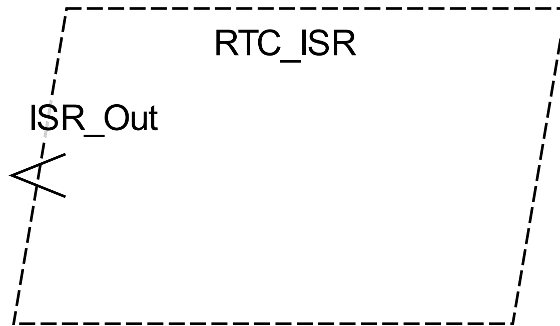


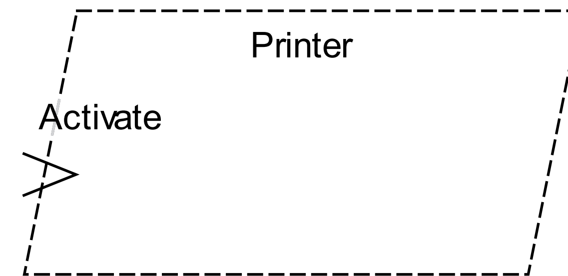
Driver: clasificador abstracto

Ejemplo 3: threads de interrupción y de usuario

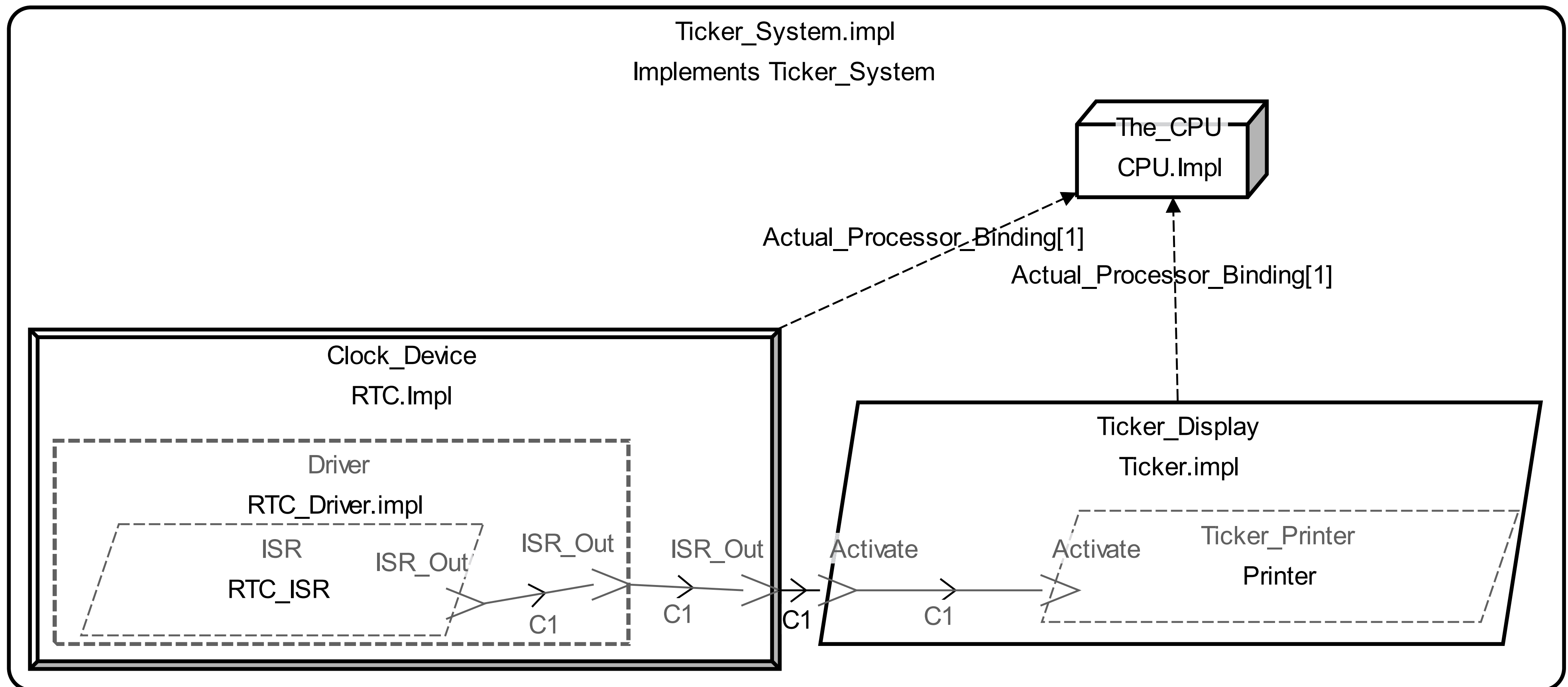
RTC_ISR

Printer

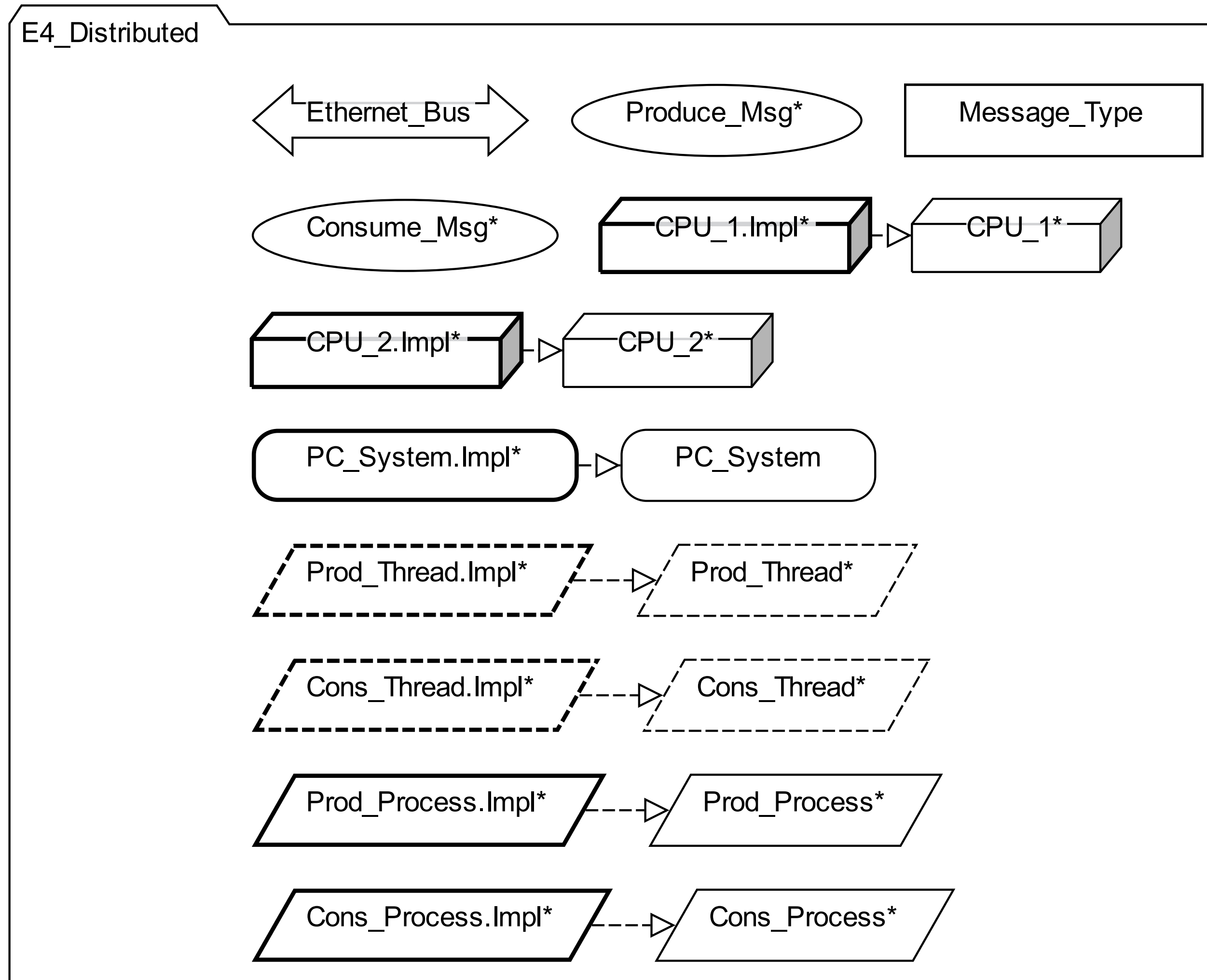




Ejemplo 3: sistema

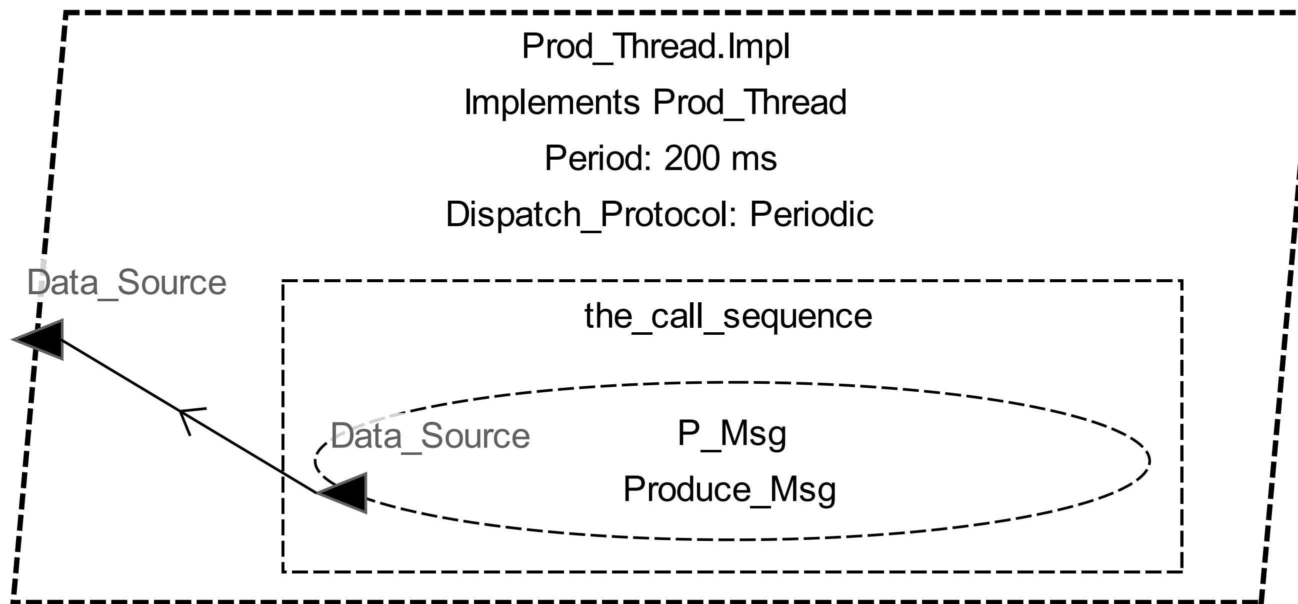


Ejemplo 4: Sistema distribuido con un thread productor y otro consumidor



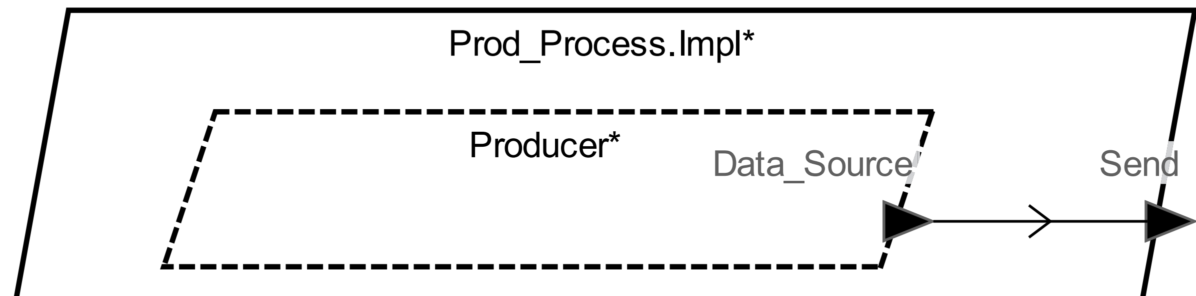
Package

E4: thread y proceso productor



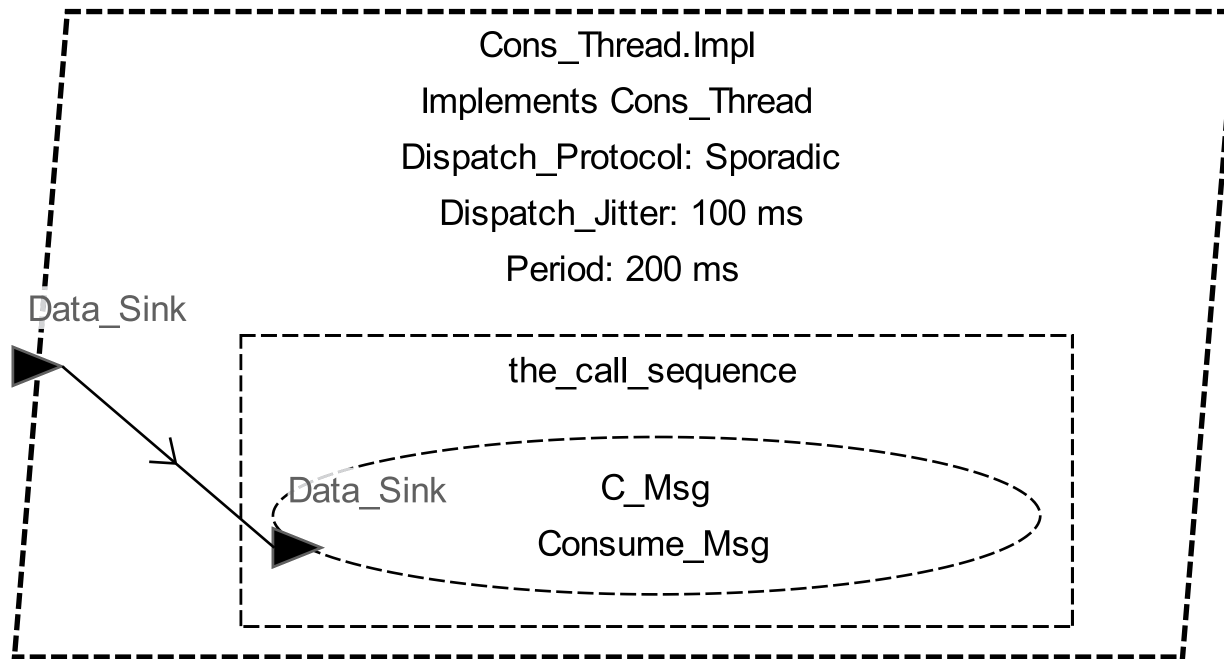
Prod_Thread.Impl

Prod_Process.Impl



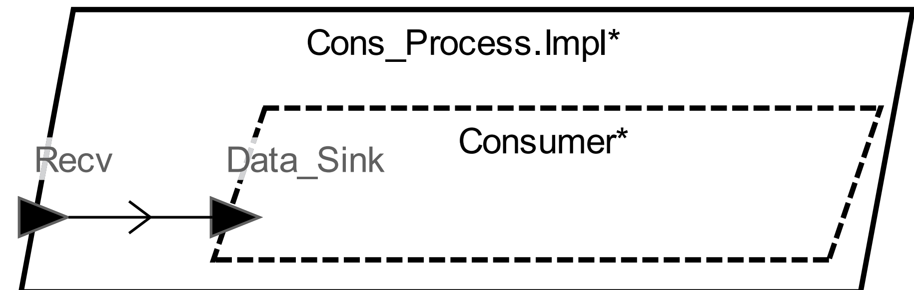
© Michael González Harbour y J.M. Drake, 8/nov/23

E4: thread y proceso consumidor

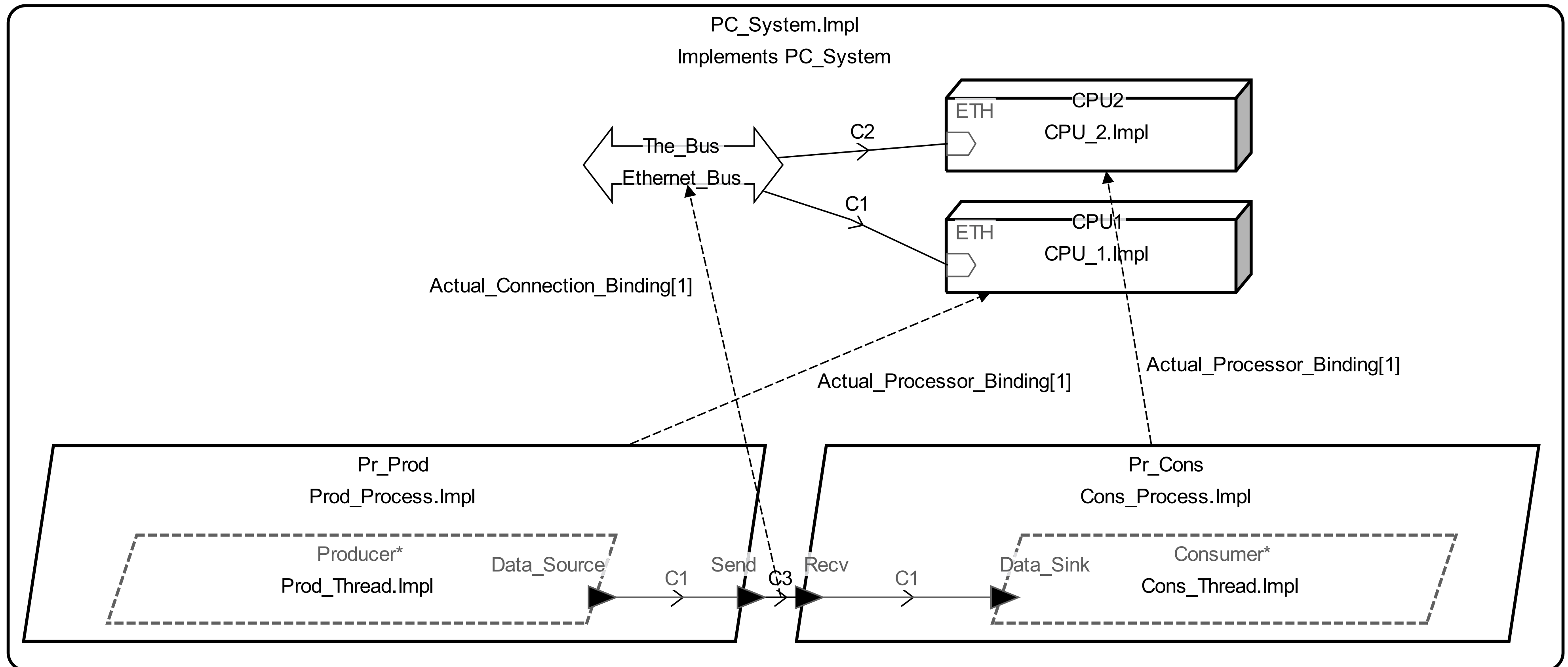


Cons_Thread.Impl

Cons_Process.Impl



E4: Sistema



4.5 Análisis y generación de código

Existen diversas herramientas de generación de modelos de análisis y de código a partir de un diseño AADL

- Generan modelos de análisis
- Generan el código ligado a la arquitectura
- Faltaría generar el código de negocio de cada componente

Una de estas herramientas es OCARINA, que puede generar C y Ada:

http://www.openaadl.org/ocarina.html

Aunque es multiplataforma, está enfocada principalmente hacia Linux

Plugin de ocarina para OSATE

Instalar plugin *ocarina*: Desde OSATE,
Help-> Install Additional OSATE Components

- y elegir Ocarina Connector

Tras instalar Ocarina (ver página siguiente), configurar OSATE para encontrar esta instalación:

Window->Preferences->OSATE->Ocarina

© Michael González Harbour y J.M. Drake, 8/nov/23

Instalación de Ocarina

Después de instalar el *plugin*, es preciso instalar la propia herramienta

Hay una versión precompilada antigua (2021), solo para Linux 64 bits, en:

- https://github.com/OpenAADL/ocarina/releases

Sin embargo, se recomienda una versión moderna compilando el código fuente en Linux 64 bits:

- Primero asegurarse de tener instalados los siguientes paquetes Linux:
  - sudo apt-get update
  - sudo apt-get install git
  - sudo apt-get install gnat
  - sudo apt-get install gprbuild
  - sudo apt-get install autoconf
  - sudo apt-get install gcc-multilib

Instalación de Ocarina (cont.)

- Instalar el script `build_ocarina.sh` con:

```
git clone https://github.com/OpenAADL/ocarina-build.git
```

- Desde la carpeta `ocarina-build`, ejecutar el script con:

```
./build_ocarina.sh --scenario=fresh-install --prefix=dir
```

  - donde `dir` es el directorio donde se quiere instalar

© Michael González Harbour y J.M. Drake, 8/nov/23

Uso de Ocarina desde Osate

Para generar código C

- Colocar en la carpeta del proyecto el archivo con el código C que implementa las operaciones del sistema
  - únicamente los archivos que corresponden al sistema que se va a generar
- Seleccionar con el ratón la implementación del sistema en el fichero AADL
- `OSATE->Ocarina->Generate PolyORB-HI- C Code`
- Se crea una carpeta de nombre similar a la implementación del sistema

© Michael González Harbour y J.M. Drake, 8/nov/23

Uso de Ocarina desde línea de comando

Alternativamente, se puede ejecutar Ocarina desde un terminal

Para generar el modelo MAST (suponiendo que ocarina está en `/usr/bin`):

```
/usr/bin/ocarina -aadlv2 -y -g mast nombre.aadl
```

Para generar código C para la plataforma PolyORB-HI:

```
/usr/bin/ocarina -aadlv2 -y -g polyorb_hi_c
nombre.aadl
```

En sucesivas ejecuciones usar la opción -z para limpiar los ficheros generados anteriormente:

```
/usr/bin/ocarina -aadlv2 -y -g polyorb_hi_c -z
nombre.aadl
```


© Michael González Harbour y J.M. Drake, 8/nov/23

Compilación y ejecución del código generado con ocarina

Poner en la carpeta del proyecto el código de usuario (poner código de un solo sistema)

- los prototipos se encuentran en el código generado en `subprograms.c`

Ejecutar `make` en la carpeta del código generado

- cuyo nombre es el del sistema aadl: `nombre-sistema`

Para ejecutar el programa, encontrarlo en la carpeta cuyo nombre es `nombre-sistema/nombre-proceso`

Ejercicio 4: modelar con OSATE

Ejercicio 4.1. Sistema distribuido basado en bus CAN, con un flujo de principio a fin que:

- se inicia en una tarea periódica en un procesador
- sigue por otra tarea en un segundo procesador
- finaliza con una tercera tarea en el primer procesador
- estudiar la forma de modelar los mensajes

© Michael González Harbour y J.M. Drake, 8/nov/23

Apéndice A. Patrones arquitecturales

Un patrón de diseño es una solución general a un problema que es frecuente

- la generalización de la solución solo merece la pena si se puede aplicar varias veces: reutilización
- la generalización permitirá que sea aplicada en diferentes dominios de aplicación

El patrón tiene como objetivo optimizar uno o varios aspectos del sistema a un coste aceptable

- y como consecuencia empeorar otros

La optimización requiere

- establecer criterios de optimización
- elegir varios patrones para ver cuál cumple mejor los criterios

Elementos de un patrón

Nombre

- breve descripción

El problema

- descripción del contexto en el que aplicar el patrón

La solución

- elementos del diseño, sus roles y sus colaboraciones
- y la implementación

Las consecuencias

- ventajas e inconvenientes

© Michael González Harbour y J.M. Drake, 8/nov/23

Instanciación de patrones

Consiste en *aplicar* un patrón para resolver un determinado problema

Se aplica a un conjunto de elementos software relacionados

- pueden ser elementos simples como clases, tipos, métodos o variables
- o elementos más complejos como subsistemas y componentes

Para realizar la instanciación será necesario *identificar* los elementos que cumplen los roles necesarios en el patrón

También será necesario *evaluar* los criterios de optimización

- asignándoles pesos a cada uno de ellos
- y evaluando el grado de cumplimiento en cada patrón
- la evaluación global será el total = peso*grado de cumplimiento

Patrones para sistemas empotrados

El libro de Bruce Powel Douglass presenta numerosos patrones:

- Patrones para el acceso al hardware
  - intermediario hardware
  - adaptador hardware
  - Mediador
  - Observador
  - Eliminación de rebotes
  - Interrupción
  - Muestreo periódico

© Michael González Harbour y J.M. Drake, 8/nov/23

Patrones para sistemas empotrados

Patrones de concurrencia y gestión de recursos

- de planificación
  - ejecutivo cíclico
  - prioridades fijas
- de coordinación entre tareas
  - región crítica
  - llamada con guarda
  - Encolado
  - Punto de encuentro (rendezvous)
- de evitación de deadlocks
  - Bloqueo simultáneo
  - Bloqueo ordenado

© Michael González Harbour y J.M. Drake, 8/nov/23

Patrones para sistemas empotrados

Patrones para máquinas de estados

- receptor de evento único
- receptor de múltiples eventos
- tabla de estados
- estado
- estados AND
- estados AND descompuestos

Patrones para seguridad y fiabilidad

- Complemento a uno
- CRC
- Datos inteligentes
- Canal
- Canal protegido simple
- Canal dual

Apéndice B: Patrones para tiempo real [8]

Patrones arquitecturales:

- Arquitectura reactiva
- Máquina de estados gobernada por eventos

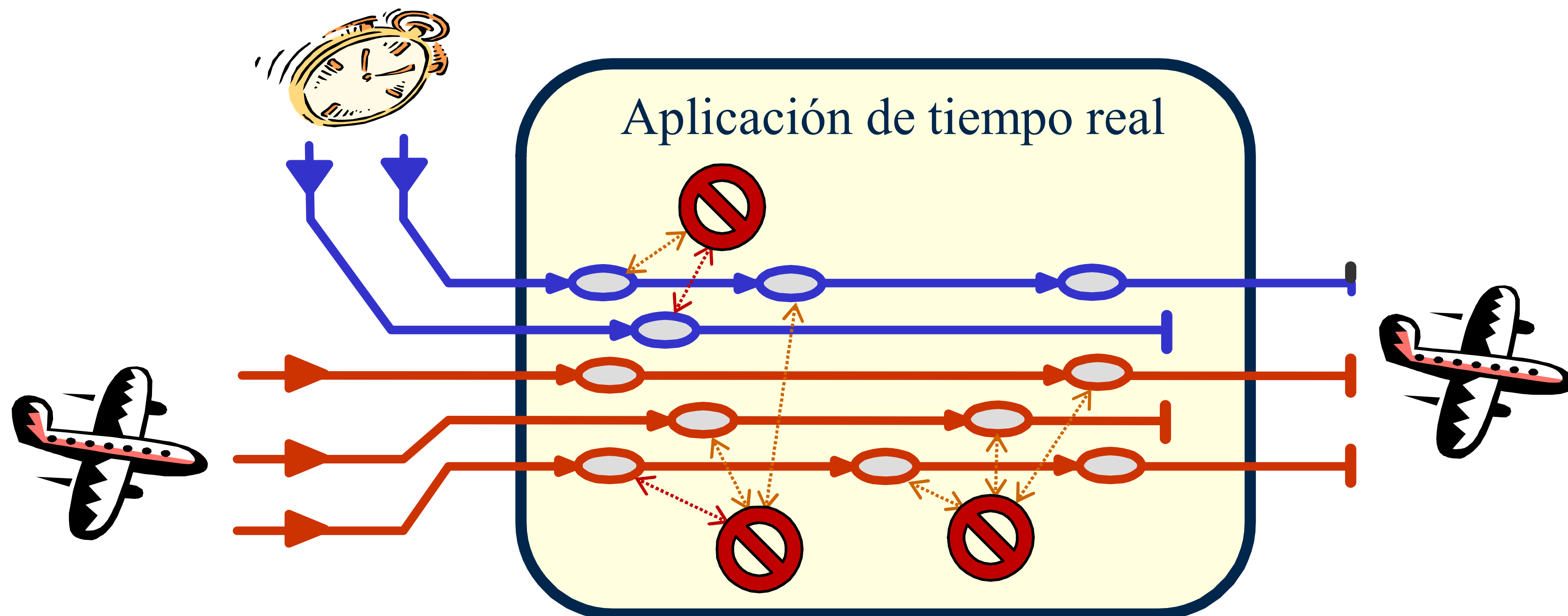
Patrones de concurrencia:

- Ejecutor
- Ejecución periódica
- Fuente de eventos síncrona bloqueante
- Fuente de eventos síncrona no bloqueante
- Fuente de eventos asíncrona callBack
- Publicador/subscriptor
- Recurso

a) Arquitectura reactiva

La aplicación se concibe (especifica, analiza y diseña) como el conjunto de respuestas a los eventos que atiende

Cada respuesta tiene definido el evento que la activa, el conjunto de actividades de que se compone, y posiblemente unos plazos temporales que debe satisfacer



Arquitectura reactiva (cont.)

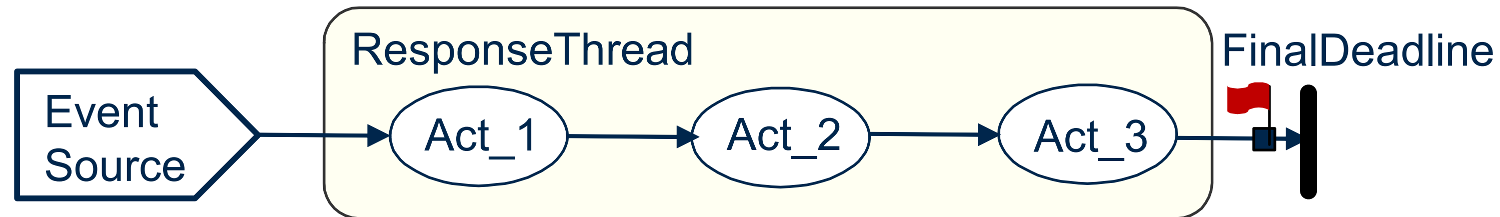
Cada respuesta es planificada por uno o varios threads dedicados exclusivamente a ella. El thread mantiene el flujo de control entre las actividades de la respuesta

Entre las diferentes respuestas no existen relaciones de flujo de control, pero sí bloqueos en base que acceden a recursos comunes que han de ser ejecutados con exclusión mutua

© Michael González Harbour y J.M. Drake, 8/nov/23

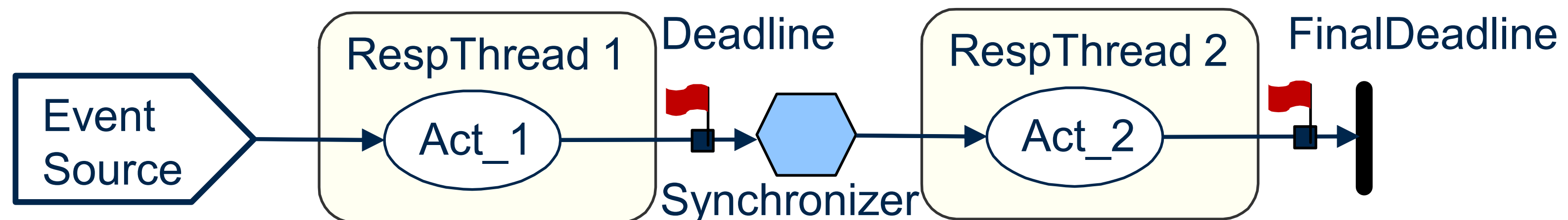
Asignación a threads de las respuestas

Normalmente el requisito temporal se establece sobre la finalización de la respuesta. En este caso un único thread planifica toda la respuesta



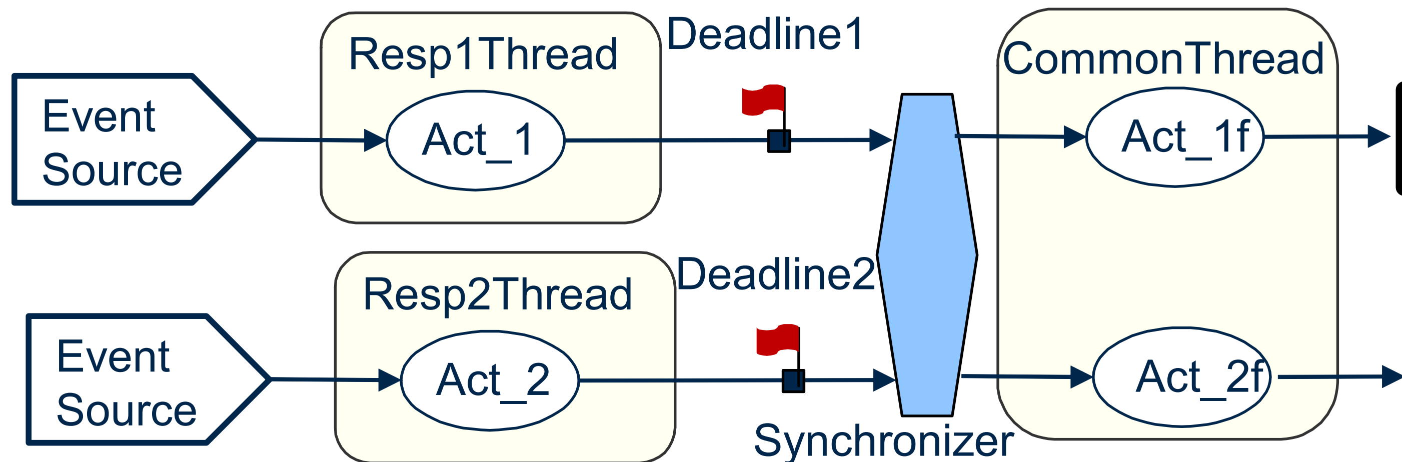
Si la respuesta tiene múltiples requisitos temporales, se introduce un thread por cada segmento previo al requisito temporal

- Entre los threads de la misma respuesta se requiere un elemento de sincronización que transfiera el flujo de control



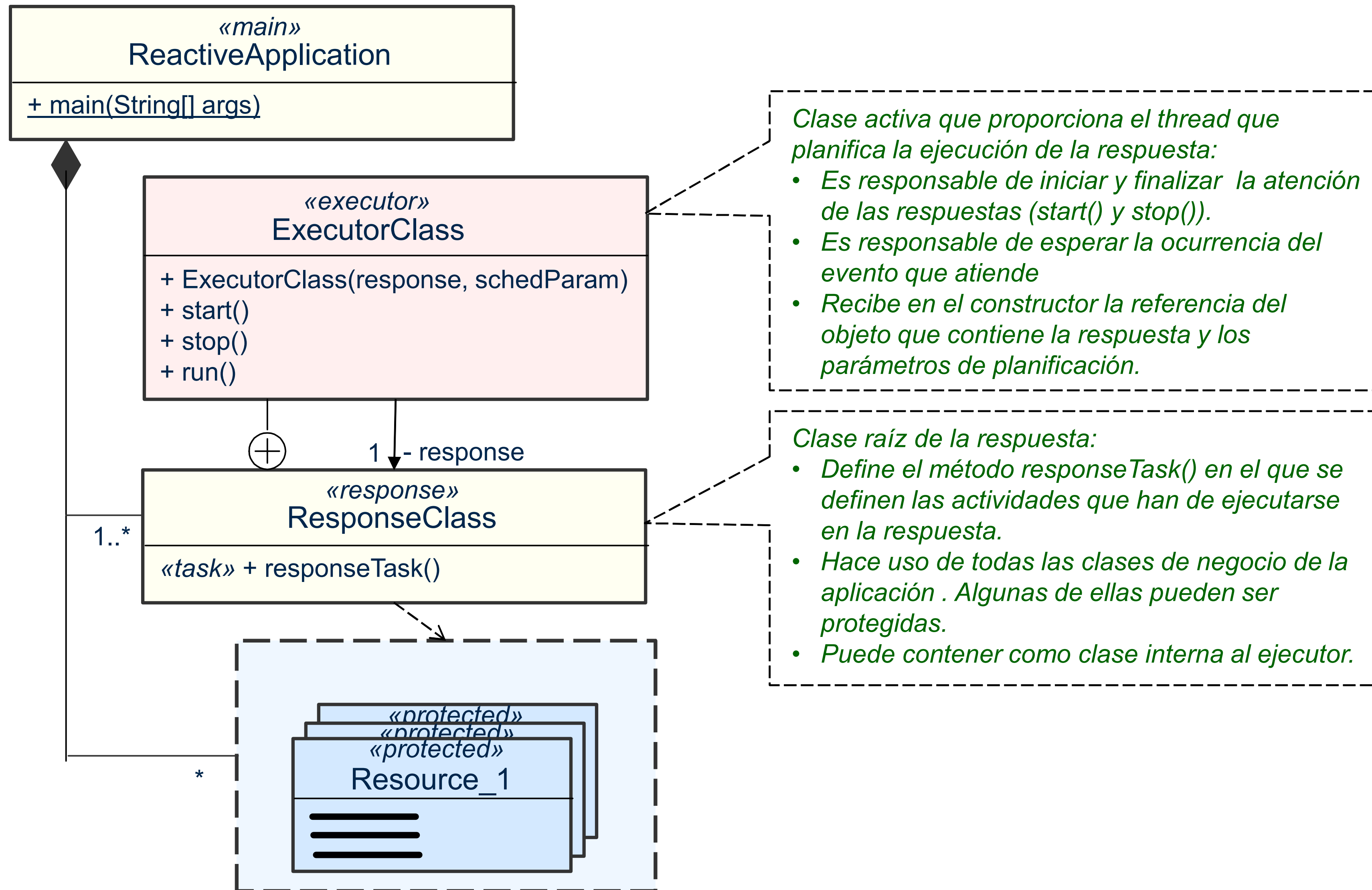
Asignación a threads (cont.)

Si el segmento final de la respuesta no tiene requisito temporal, se puede utilizar para su planificación un thread común a diferentes respuestas

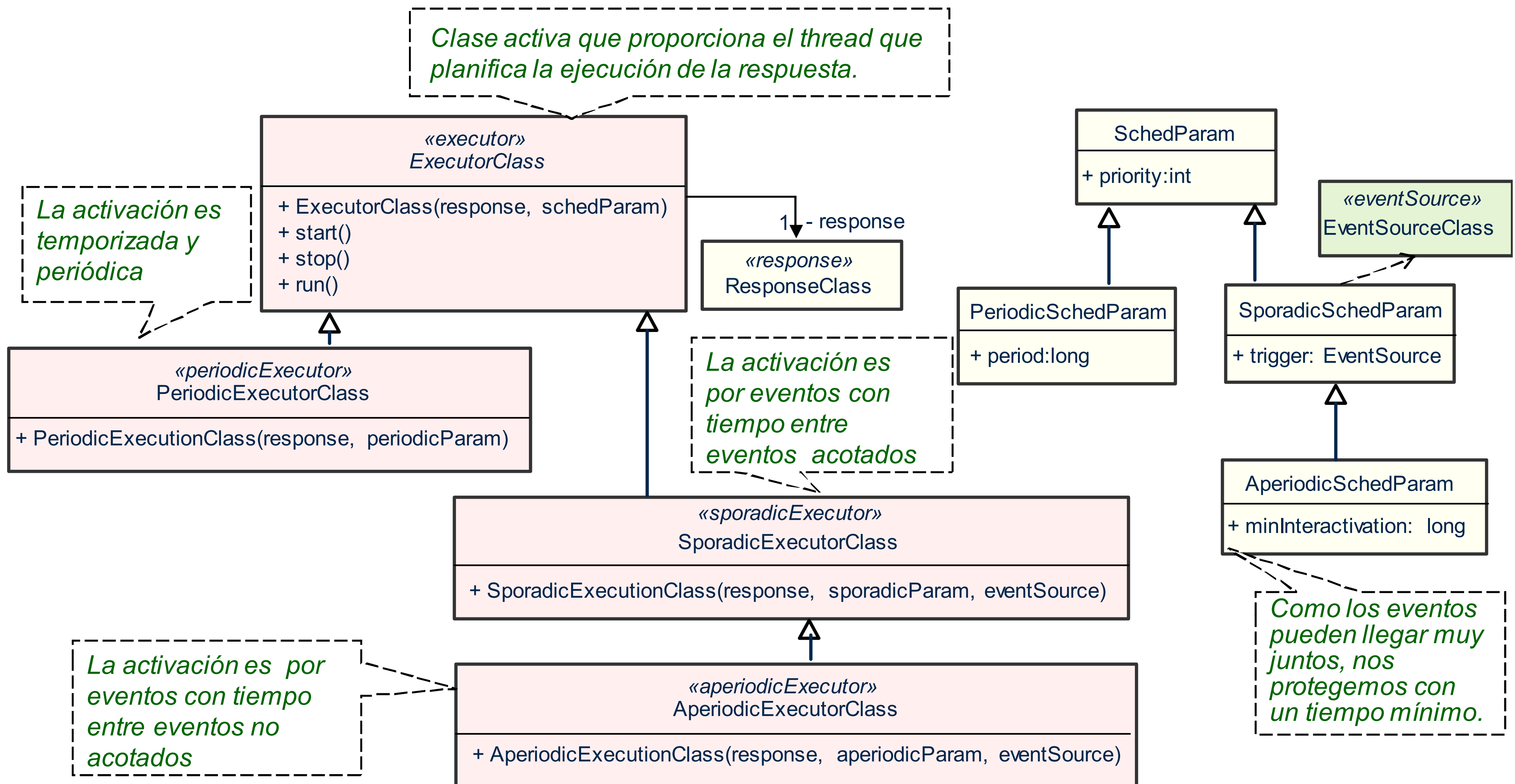


© Michael González Harbour y J.M. Drake, 8/nov/23

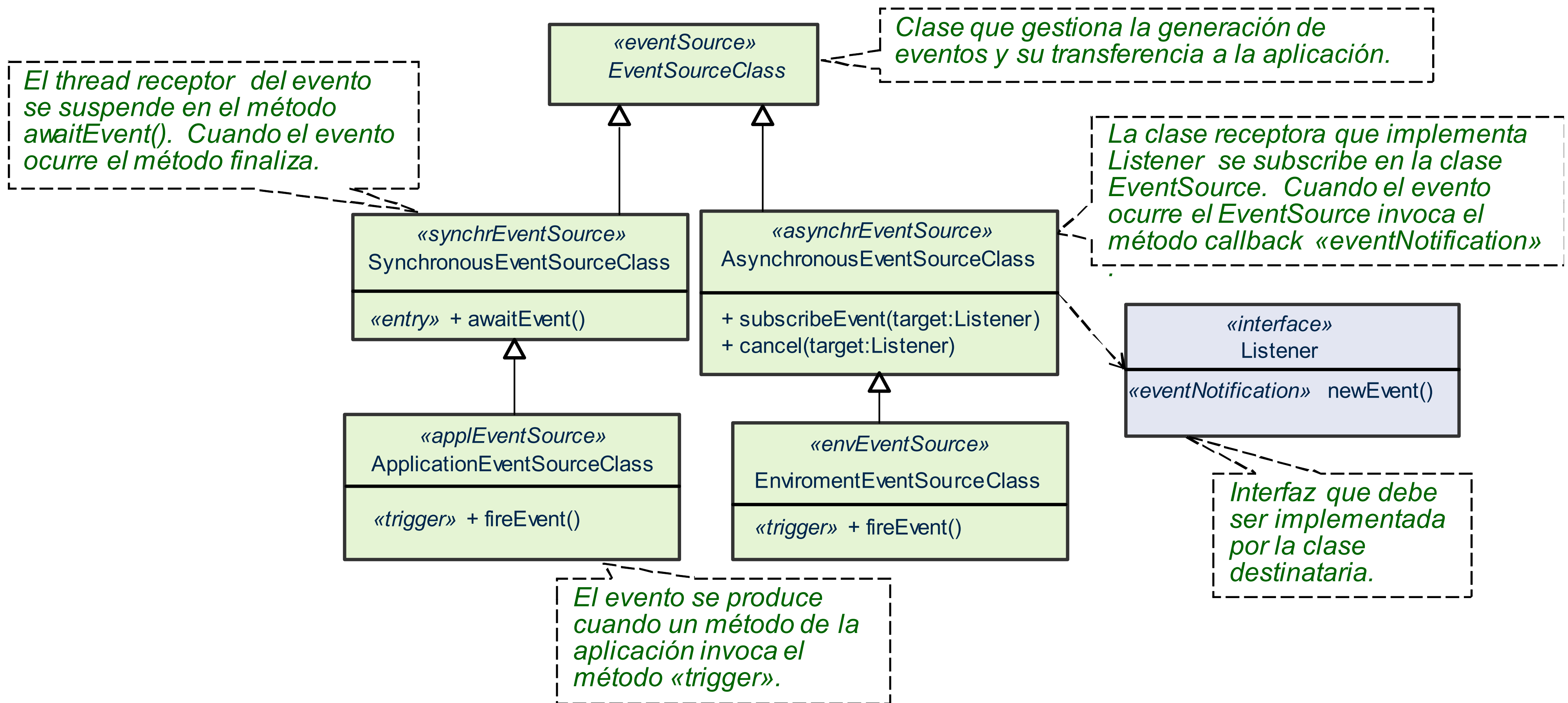
Elementos del patrón reactivo



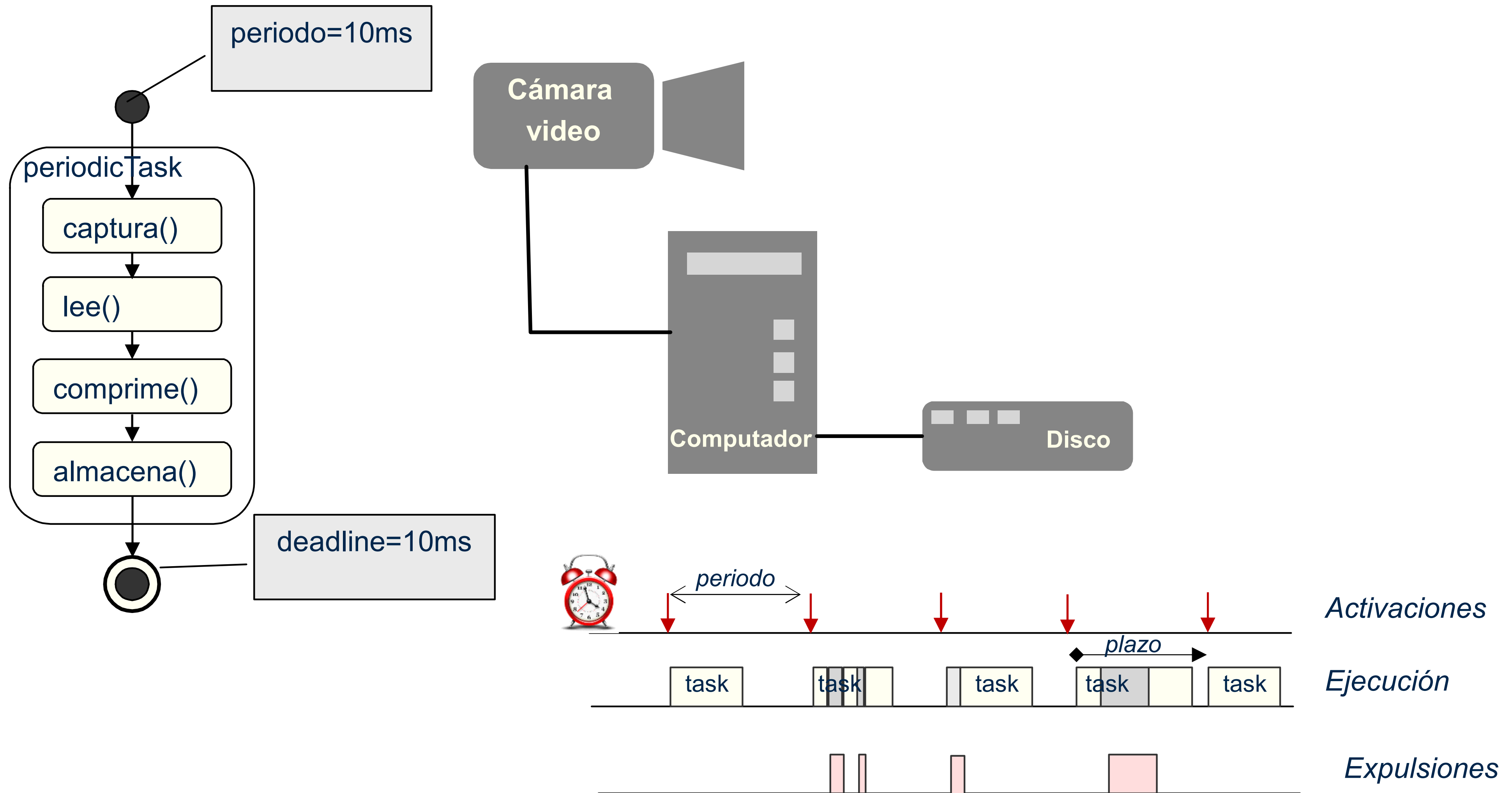
Tipos de ejecutores



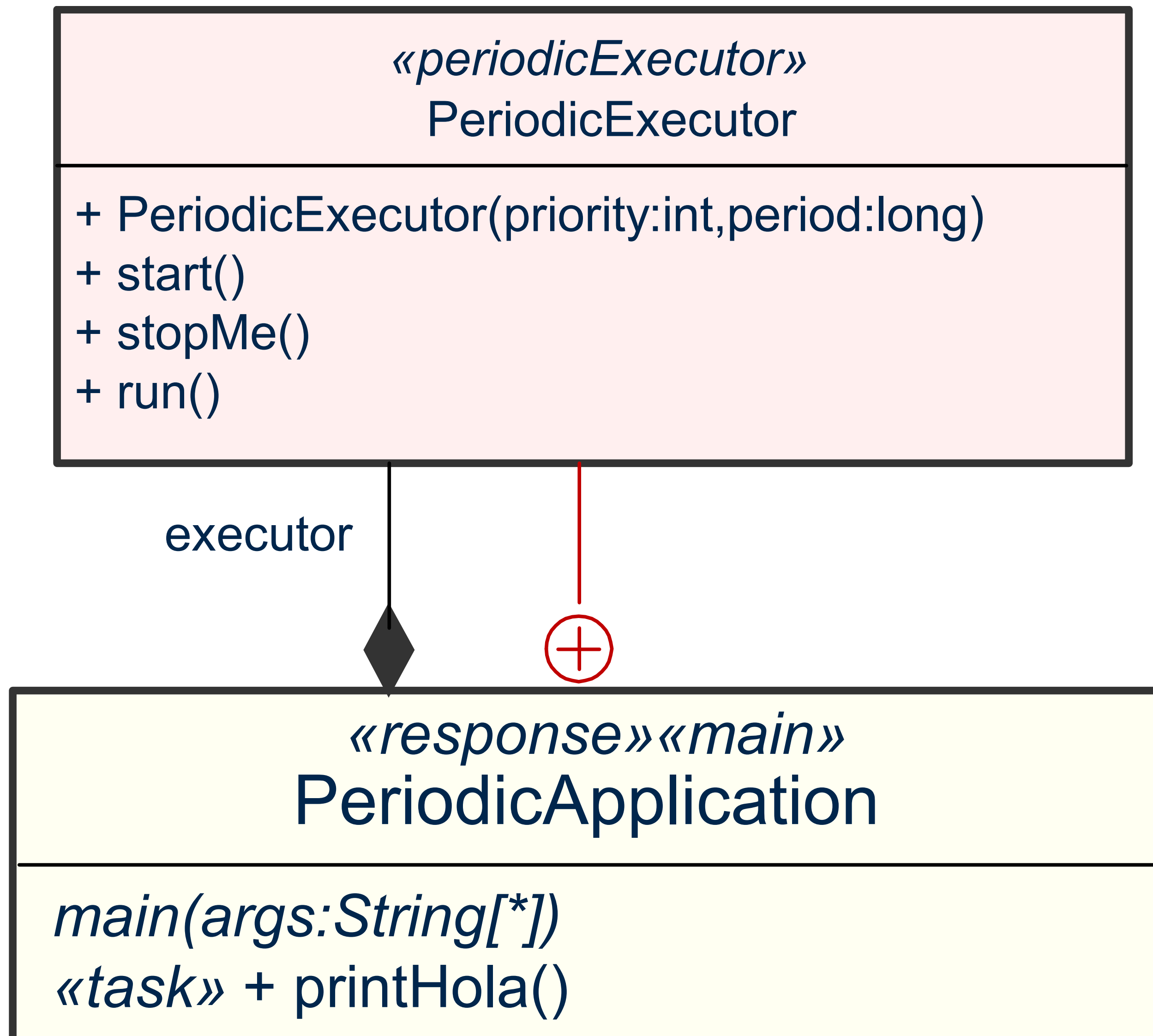
Fuentes de eventos



Ejemplo de una tarea periódica: Captura de vídeo



Ejemplo: PeriodicExecutor Class



Ejemplo: PeriodicExecutor Class (código)

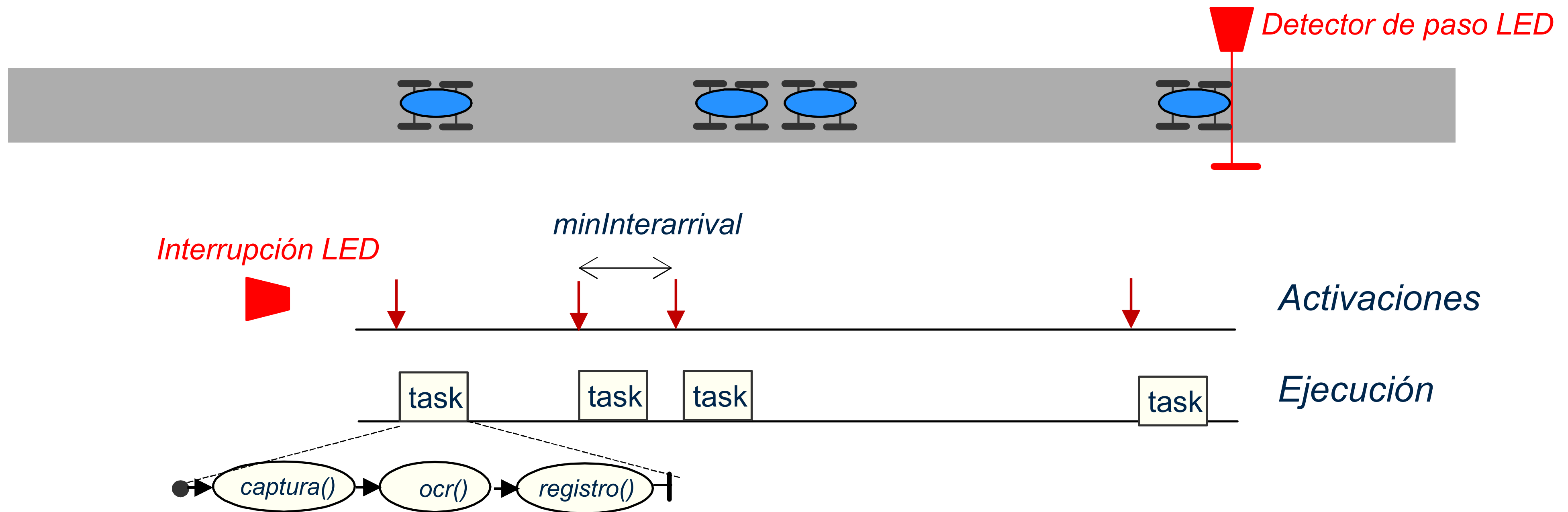
```
public class PeriodicApplication {
   PeriodicExecutor executor=new PeriodicExecutor(5,1000);

   public static void main(String[] args) {
       PeriodicApplication pa=new PeriodicApplication();
       pa.executor.start();
       try {
          Thread.sleep(10000);
       }catch (InterruptedException e) {}
       pa.executor.stopMe();
   }

   // simula el trabajo útil del ejecutor periódico
   private void printHola(){
      System.out.println("Hola");
   }
```

Ejemplo: PeriodicExecutor Class (código)

```
private class PeriodicExecutor extends Thread{
   long period;
   volatile boolean stop=false;
   long nextActivation=System.currentTimeMillis();
   public PeriodicExecutor(int priority,long period){
     this.period=period;
     setPriority(priority);
   }
   public void run(){
      while(!stop){
         nextActivation+=period;
         printHola();
         try {sleep(nextActivation-
                 System.currentTimeMillis());
         } catch (InterruptedException e) {}
      }
   }
   public void stopMe(){
       stop=true;
   }
}
```

Ejemplo de tarea esporádica: Captura de matrículas

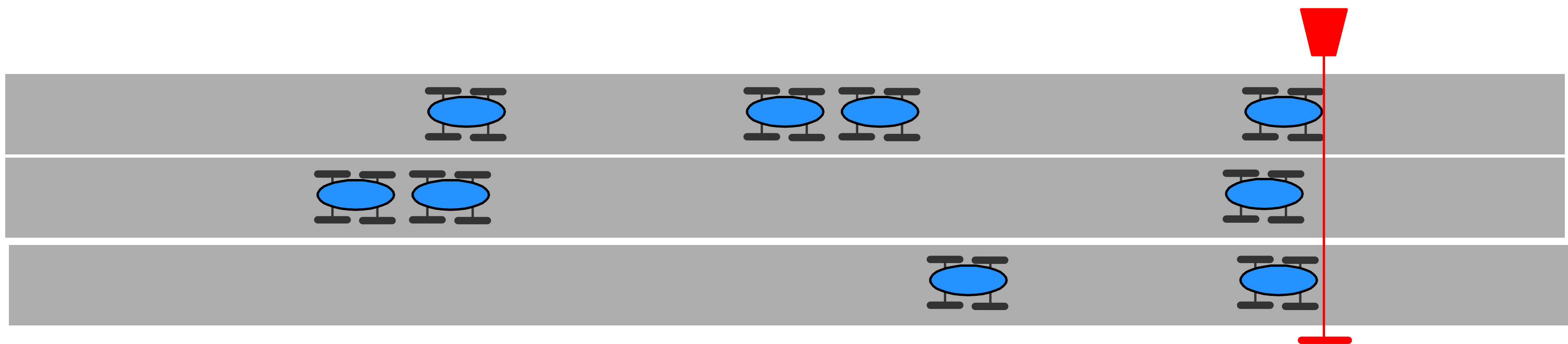
Con tiempo mínimo entre llegadas

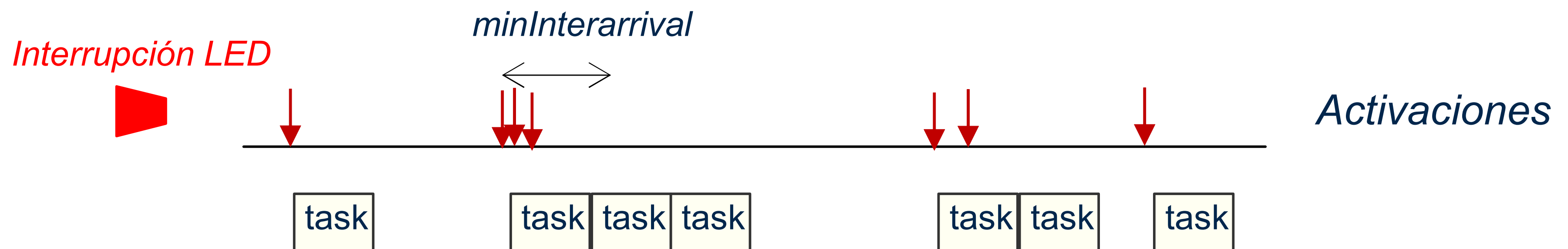


© Michael González Harbour y J.M. Drake, 8/nov/23

Ejemplo de tarea esporádica: Captura de matrículas

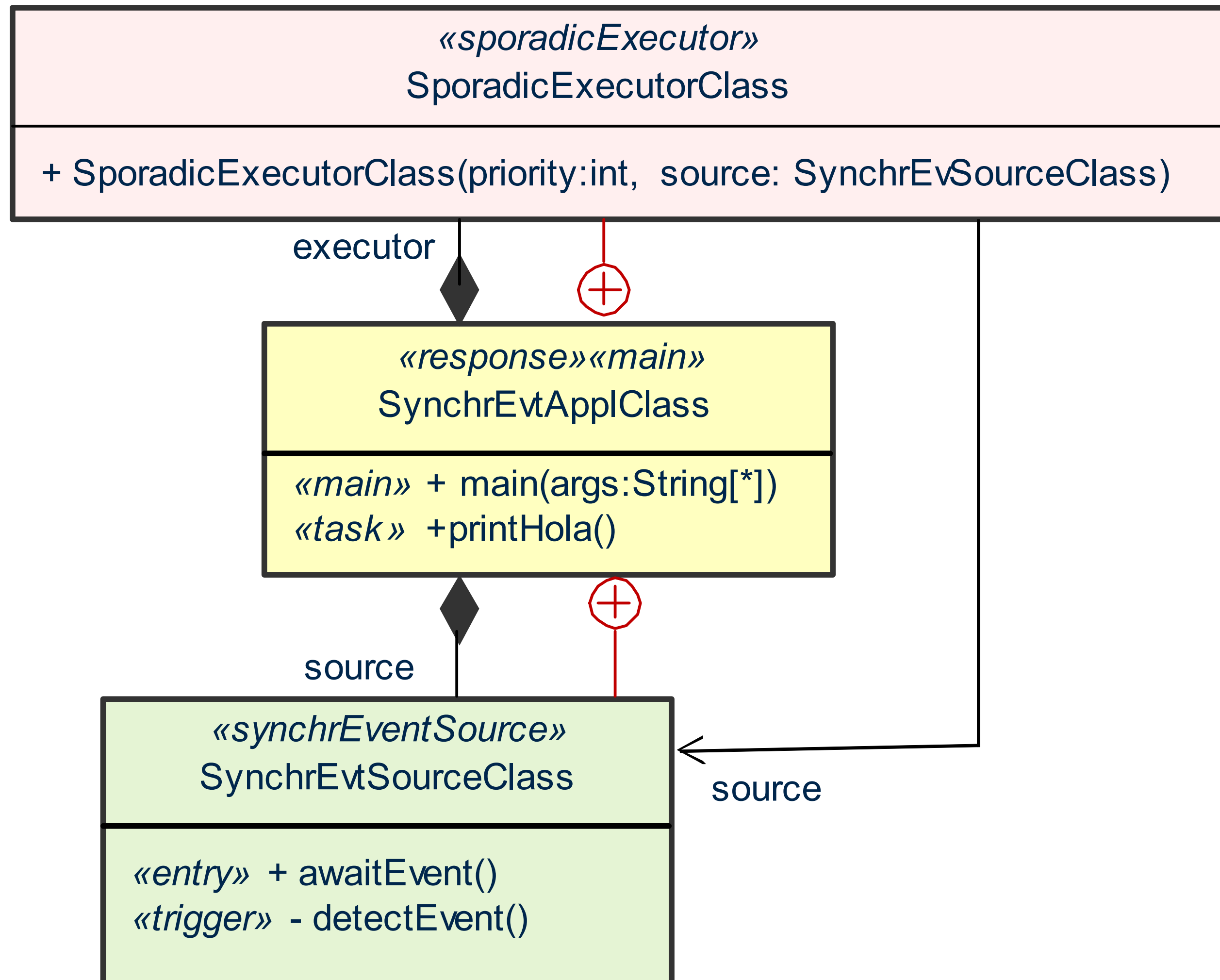
Con activación a ráfagas (bursty)





© Michael González Harbour y J.M. Drake, 8/nov/23

Ejemplo de ejecutor esporádico con evento síncrono



© Michael González Harbour y J.M. Drake, 8/nov/23

Ejecutor esporádico: código

```
public class SynchrEvAppClass {

   SynchrEvSourceClass source=
       new SynchrEvSourceClass();
   SporadicExecutor executor=
       new SporadicExecutorClass(5,source);

   public static void main(String[] args) {
      SynchrEvApp app=new SynchrEvApp();
      app.executor.start();
      try{
         Thread.sleep(15000);
      }catch(InterruptedException e){}
      app.executor.stopMe();
    }

   // simula el trabajo útil del ejecutor esporádico
    void printHola(){
       System.out.println("Hola");
    }
```


© Michael González Harbour y J.M. Drake, 8/nov/23

Ejecutor esporádico: código

```
private class SporadicExecutorClass extends Thread{
  SynchrEvSourceClass source;
  volatile boolean stop=false;
  public SporadicExecutorClass(
     int priority,SynchrEvSourceClass source)
  {
     this.source=source;
     setPriority(priority);
  }
  public void run(){
     while(!stop){
        source.awaitEvent();
        printHola();
     }
  }

  public void stopMe(){
     stop=true;
  }
}
```

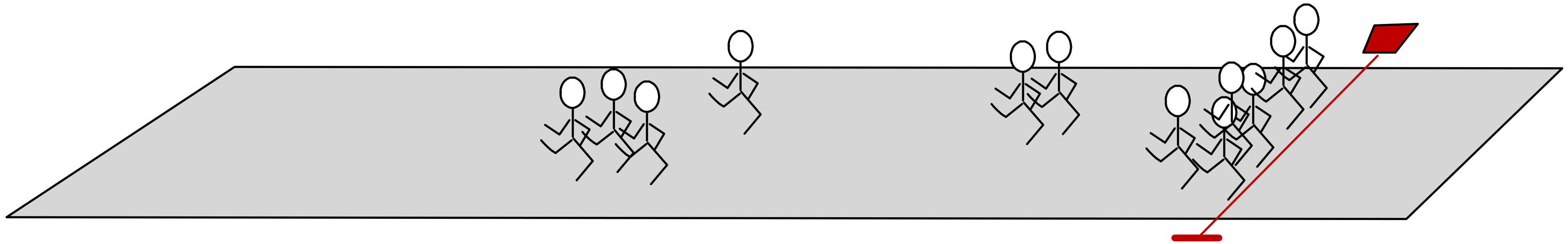

© Michael González Harbour y J.M. Drake, 8/nov/23

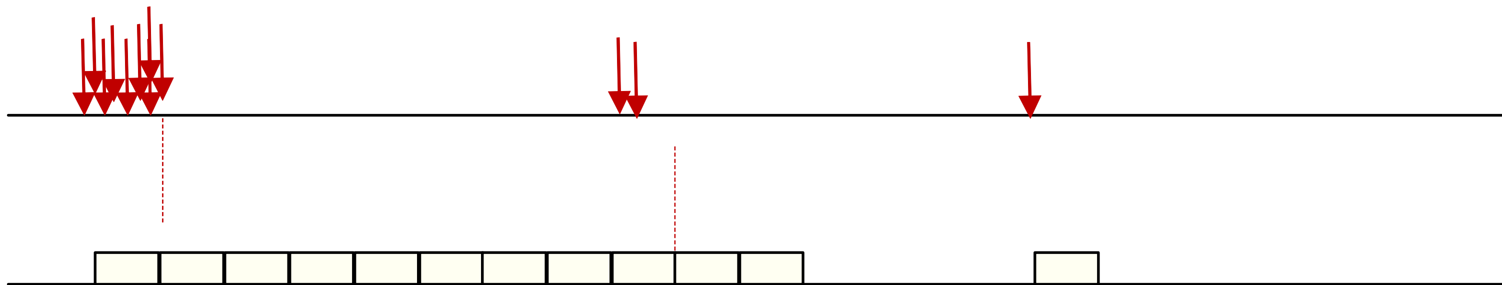
Ejecutor esporádico: código

```
    private class SynchrEvSourceClass{
       public boolean hasEvent=false;
       public synchronized void awaitEvent(){
          while(!hasEvent){
            try {
               wait();
            }catch (InterruptedException e){}
          }
          hasEvent=false;
        }
        public void detectEvent(){
           ….
          hasEvent=true;
          notifyAll();
        }
    }
}
```


© Michael González Harbour y J.M. Drake, 8/nov/23

Activación aperiódica: Control de un maratón





© Michael González Harbour y J.M. Drake, 8/nov/23

Ejecutor aperiódico con evento síncrono

«aperiodicExecutor»
AperiodicExecutorClass

+AperiodicExecutionClass(priority:int, minInterarival:long, source: SynchrEvSourceClass)

executor

«response»«main»
AperiodicAppClass

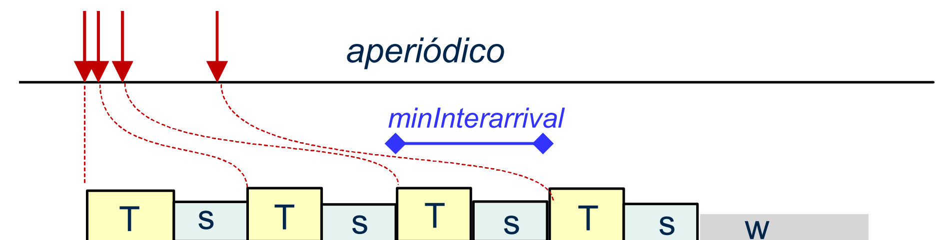
«main» + main(args:String[*])
«task» +printHola()

source

«synchrEventSource»
SynchrEvtSourceClass

«entry» + awaitEvent()
«trigger» - detectEvent()

source



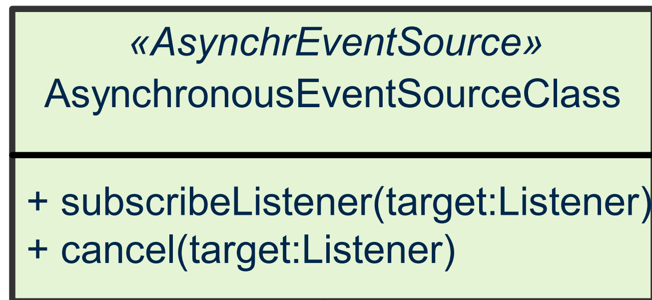
Código del ejecutor aperiódico

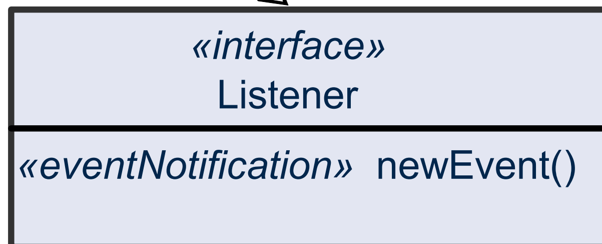
```
private class AperiodicExecutorClass extends Thread{

   SynchrEvSource source;
   long minInterarrival;

   public AperiodicExecutor(int priority,
                            long minInterarrival,
                            SynchrEvSourceClass source)
   {
      this.source=source;
      this.minInterarrival=minInterarrival;
      setPriority(priority);
   }
```

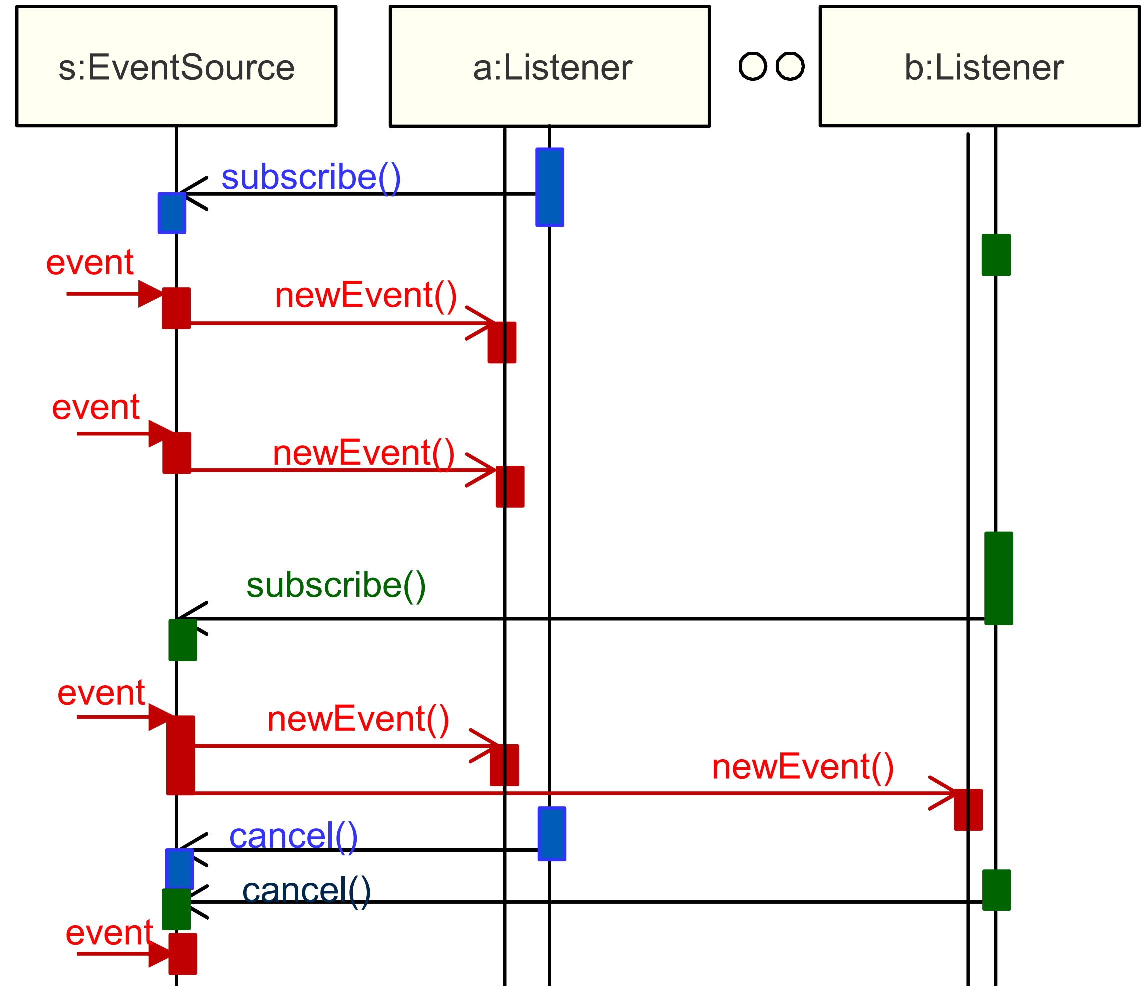
Código del ejecutor aperiódico

```
    public void run(){
        long pExec,cExec;
        while(true){
            pExec=System.currentTimeMillis();
            source.awaitEvent();
            printHola();
            cExec=System.currentTimeMillis();
            if(minInterarrival-cExec+pExec>0)
                try {sleep (minInterarrival-cExec+pExec);
                }catch (InterruptedException e) {}
            }
        }
    }
}
```

Publicador-suscriptor



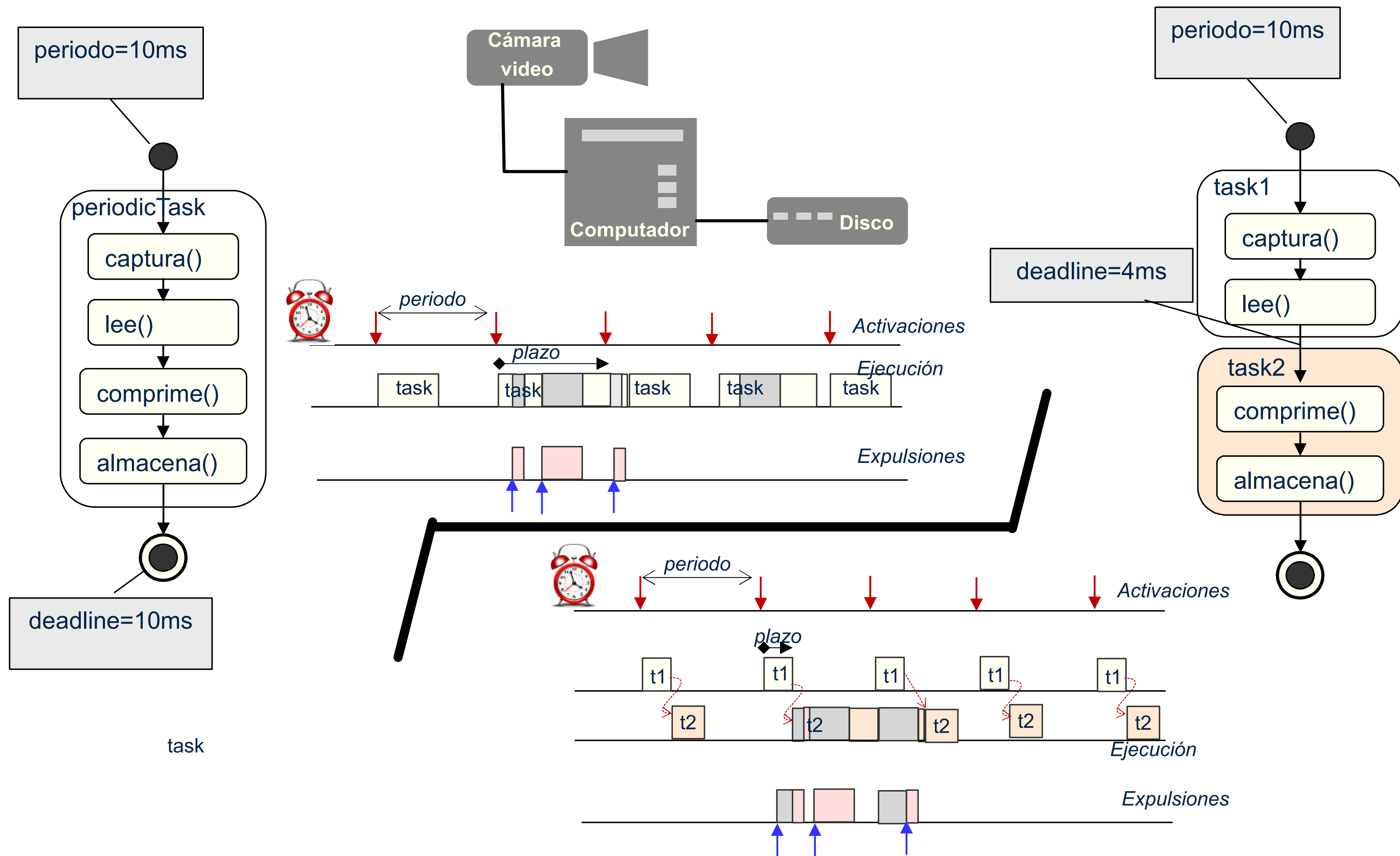




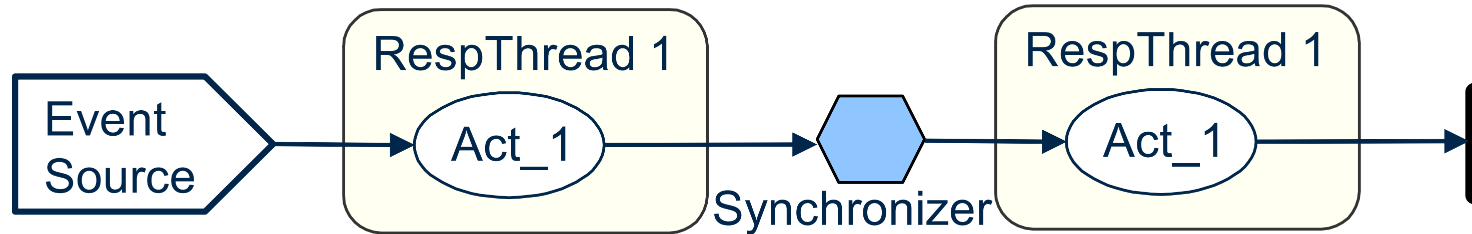
Publicador-suscriptor: código

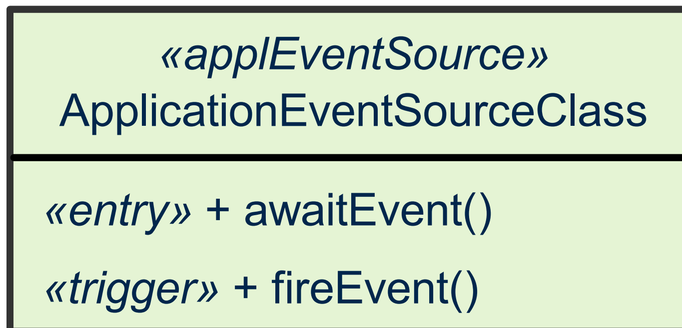
```
import java.util.LinkedList;

public class AsynchronousEventSourceClass {
   LinkedList<Listener> subscribers=new LinkedList<Listener>();

    public void subscribeEvent(Listener target){
       subscribers.add(target);
    }
    public void cancel(Listener target){
       subscribers.remove(target);
    }

    public void detectEvent(){
        // ...
        // Cuando se detecta un evento
        for(Listener l:subscribers){
           l.newEvent();
        }
       // ...
     }
}
```

Transferencia de flujo entre threads



Application event source class

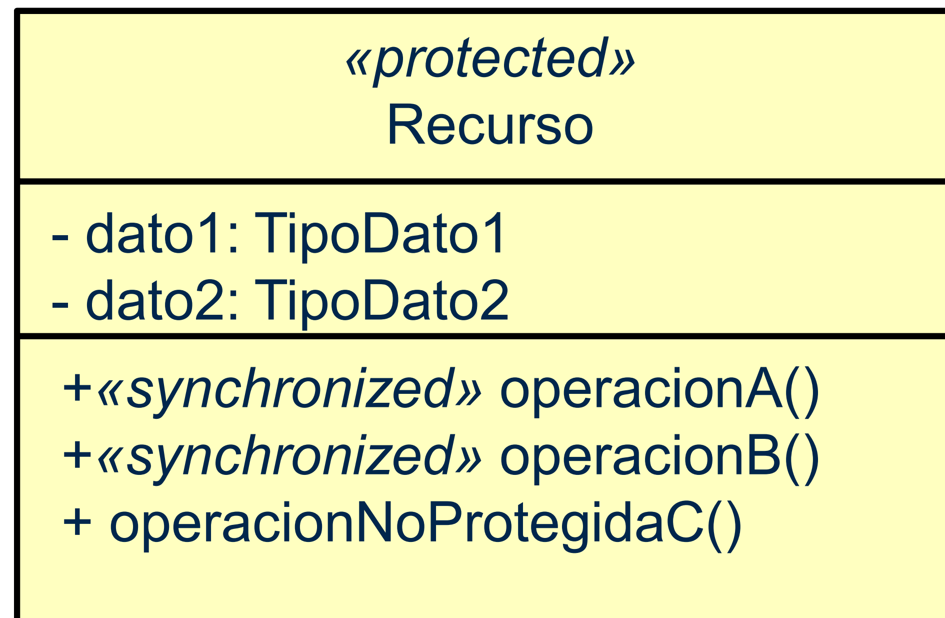





```
public class ApplicationEventSource {

  public synchronized void awaitEvent(){
      try {
          wait();
      } catch (InterruptedException e) {}
   }

   public synchronized void fireEvent(){
      notify();
   }
}
```

Recursos protegidos

Cuando varios threads actualizan concurrentemente el estado de un objeto, los métodos de acceso deben estar cualificados como `synchronized`



© Michael González Harbour y J.M. Drake, 8/nov/23

b) Máquina de estados gobernada por eventos

El elemento central de la aplicación es una clase que implementa una maquina de estados:

- Un estado es una situación del sistema con un comportamiento cualitativamente diferenciado

La transición entre estados se produce por:

- La ocurrencia de un evento
- Por la finalización de un plazo temporal
- Por la finalización de la actividad asignada al estado

Las transiciones entre estados puede ser inhibidas por una condición de guarda booleana

© Michael González Harbour y J.M. Drake, 8/nov/23

Máquinas de estados (cont.)

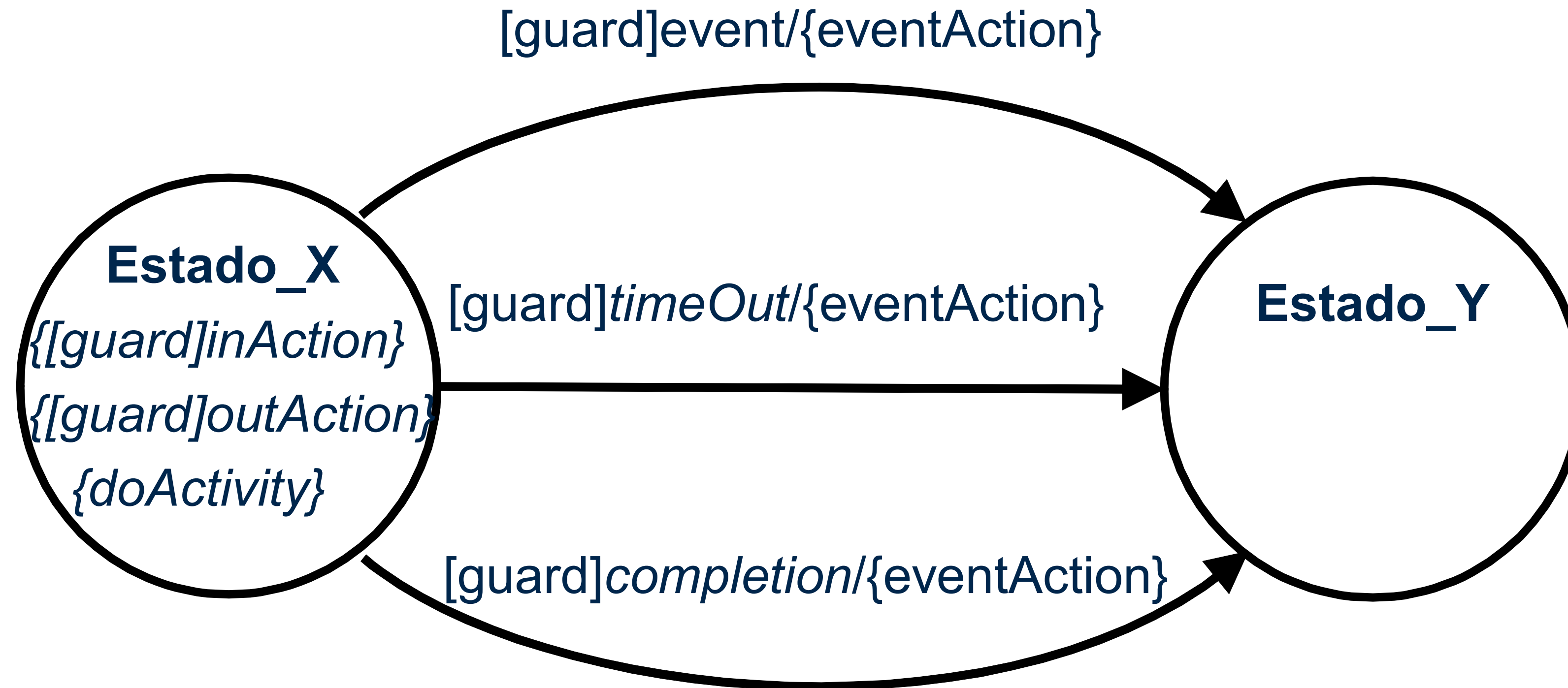
Las operaciones de la aplicación son acciones o actividades que se asocian a:

- La ocurrencia de un evento «`eventAction`»
- La entrada en un estado «`inAction`»
- La salida de un estado «`outAction`»
- Una actividad asociada a la permanencia en el estado «`doActivity`»

Action: tarea que se inicia y se finaliza, no interrumpible por la ocurrencia de un evento

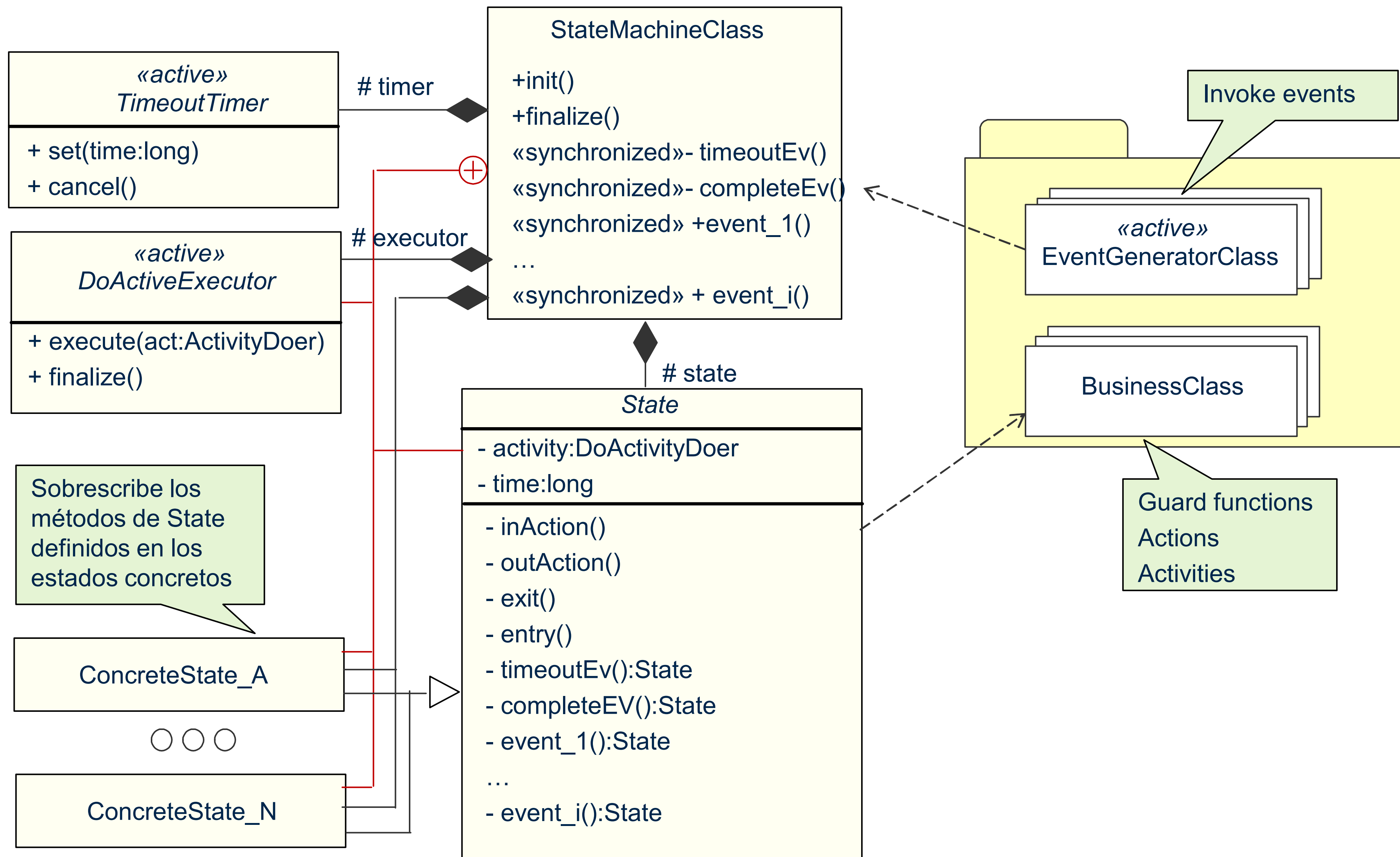
Activity: tarea que puede ser interrumpida por la ocurrencia de un evento

Máquinas de estados (cont.)

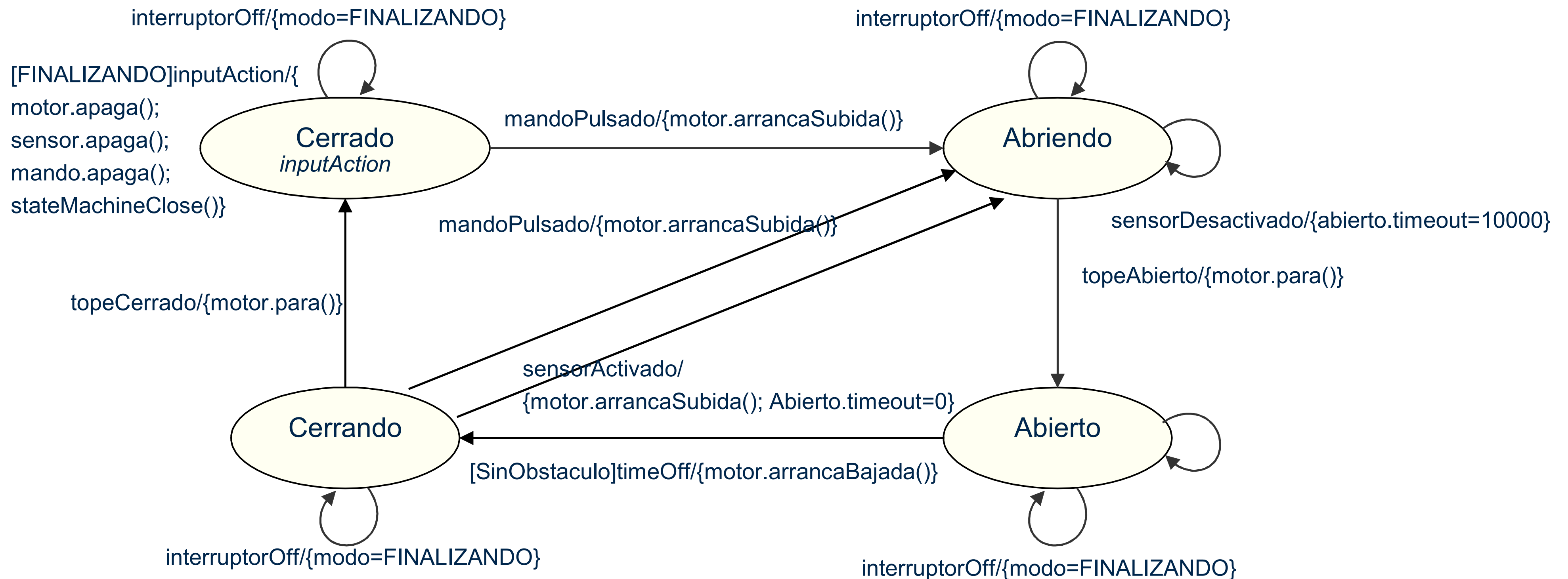


La actividad `doActivity` debe ser interrumpible cuando ocurre un evento que suponga una transición de estado

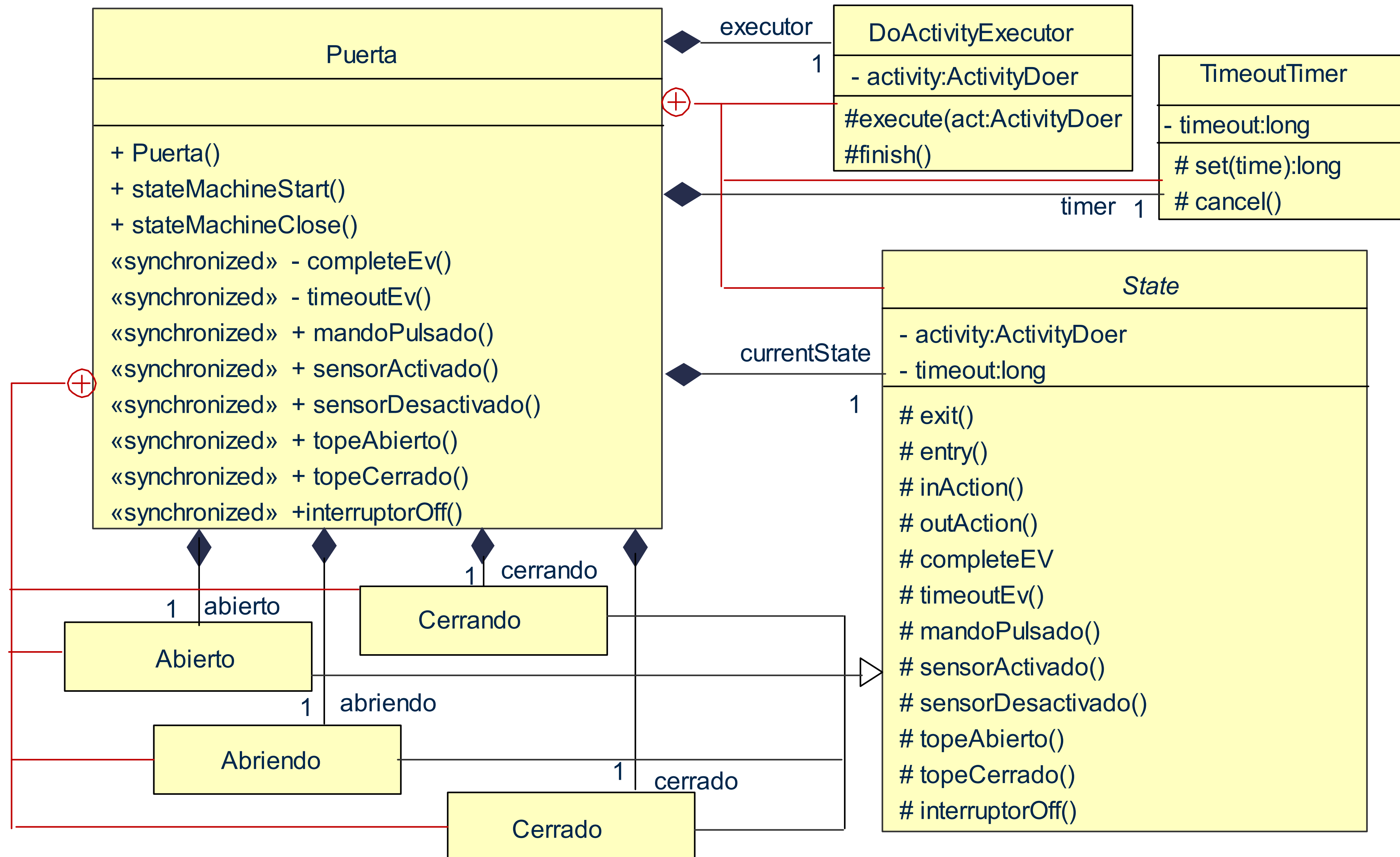
Elementos básicos del patrón



Ejemplo: control de una puerta

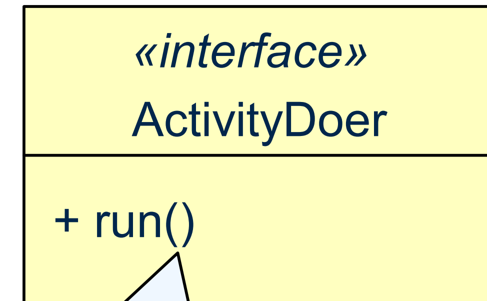


Puerta: Clases de la máquina estados



Estado abstracto

```
private abstract class State{

   protected ActivityDoer activity=null;
   protected long timeout=0;

   protected void exit(){
      executor.finish();
      currentState.outAction();
      timer.cancel();
   }

   protected void entry(){
      currentState.inAction();
      executor.execute(currentState.activity);
      timer.set(currentState.timeout);
   }
```



throws InterruptedException

© Michael González Harbour y J.M. Drake, 8/nov/23

Estado abstracto (cont.)

```
   protected void inAction(){}
   protected void outAction(){}
   protected void timeoutEv(){}
   protected void completeEv(){}

   protected void mandoPulsado(){}
   protected void sensorActivado(){}
   protected void sensorDesactivado(){}
   protected void topeAbierto(){}
   protected void topeCerrado(){}
   protected void interruptorOff(){}
}
```


© Michael González Harbour y J.M. Drake, 8/nov/23

Ejemplo del estado concreto: Cerrado

```
private class Cerrado extends State{

  private Cerrado(){}

  protected void mandoPulsado(){
    exit();
    motor.arrancaSubida();
    currentState=abriendo;
    entry();
   }

  protected void interruptorOff(){
    exit();
    currentState=cerrado;
    entry();
  }
```


© Michael González Harbour y J.M. Drake, 8/nov/23

Ejemplo del estado concreto: Cerrado

```
  protected void inAction(){
   if(modo==ModoOperacion.FINALIZANDO){
     motor.apaga(); // Cierra motor
     // Cierra sensor
    Garaje.driver.writeOutRegData(bitInterruptorSensor,false);
     // Cierra mando
     Garaje.driver.writeOutRegData(bitInterruptorMando,false);
     stateMachineClose(); // Cierra la máquina de estados
  }
 }
}
```

Código de DoActivityExecutor

```
private class DoActivityExecutor extends Thread{

   ActivityDoer activity=null;

   protected synchronized void execute(ActivityDoer activity){
      this.activity=activity;
      notify();
   }

   protected synchronized void finish(){
      if (activity!=null){
         activity=null;
         interrupt();
         try{
            wait();
         }catch(InterruptedException e){}
      }
   }
```


© Michael González Harbour y J.M. Drake, 8/nov/23

Código de DoActivityExecutor

```
    public void run(){
        while(true){
            synchronized(this){
                notify();
                while(activity==null){
                    try{wait();
                    }catch(InterruptedException e){}
                }
            }
            try{
                activity.run();
            }catch (InterruptedException e){
                continue;
            }
            completeEv();
        }
    }
}
```


© Michael González Harbour y J.M. Drake, 8/nov/23

Código de TimeoutTimer

```
private class TimeoutTimer extends Thread{

   long time=0;

   protected synchronized void set(long time){
      this.time=time;
      notify();
   }

   protected synchronized void cancel(){
      if(time!=0){
         time=0;
         interrupt();
      }
    }
```


© Michael González Harbour y J.M. Drake, 8/nov/23

Código de TimeoutTimer

```
   public void run(){
      while (true){
         synchronized(this){
            while(time==0){
               try{wait();
               }catch(InterruptedException e){}
            }
         }
         try{sleep(time);
              timeoutEv();
         }catch(InterruptedException e){}
      }
   }
}
```


© Michael González Harbour y J.M. Drake, 8/nov/23

Bibliografía

[1] “Real-Time Systems: Design Principles for Distributed Embedded Applications”. Hermann Kopetz. Springer, 2011

[2] “AADL Resource pages”: http://www.openaadl.org/index.html

[3] “AADL tutorial”: http://www.openaadl.org/post/2013/04/15/aadl-tutorial/

[4] “AADL tutorial at MODELS'15, in Ottawa, Canada”: http://www.openaadl.org/post/2015/09/28/models/

[5] “SAE AADL V2: An Overview”: https://wiki.sei.cmu.edu/aadl/images/7/73/AADLV2Overview-AADLUserDay-Feb_2010.pdf

[6] OSATE Graphical Editor Video Presentation: http://www.aadl.info/aadl/currentsite//documents/20140327-osate-ge-presentation.wmv

[7] “Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit”. Bruce Powel Douglass. Elsevier, 2010

[8] “Patrones de diseño de tiempo real”. José M. Drake, Patricia López Martínez, Michael González Harbour, Laura Barros, Cesar Cuevas y José María Martínez Lanza.

Grupo de Ingeniería Software y Tiempo Real (ISTR), Universidad de Cantabria. http://hdl.handle.net/10902/8420

[9] “Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language”. Peter H. Feiler, David P. Gluch. Addison-Wesley, 2012. ISBN: 0-321-88894-4

[10] AADL v2.1 Syntax Card: https://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwi9gOzo_dzQAhWB2xoKHS6FDoQQFggdMAA&url=https%3A%2F%2Fwiki.sei.cmu.edu%2Faadl%2Fimages%2Fd%2Fd2%2FAADL_V2.1_Syntax_Card.pdf&usg=AFQjCNF-w2KK356QhFqie5_nshUUzmSzLg&sig2=pNxBvsKQyQyHhf_RRfWJBA&cad=rja