Watchdog y Timers

Programación concurrente y Distribuída

Curso 2011-12

Miguel Telleria, Laura Barros, J.M. Drake

`telleriam AT unican.es`

Computadores y Tiempo Real

http://www.ctr.unican.es

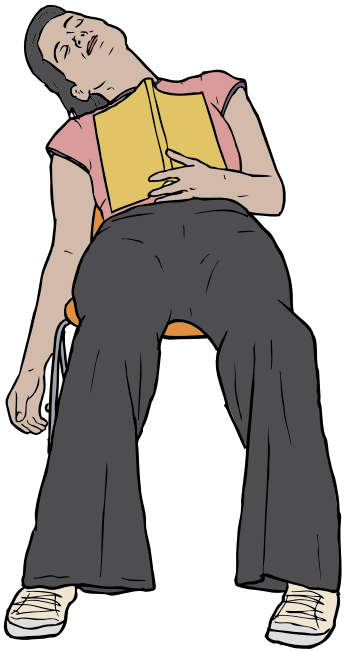
Contenido

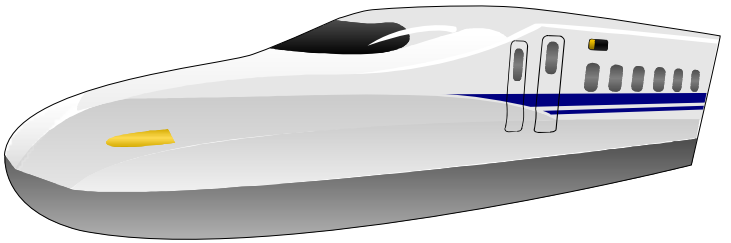
- Patrón de watchdog e implementaciones
- Uso del watchdog en la cena de los filósofos
- Timer casero

Patrón de Watchdog

Ejemplo: Conductor de tren dormido

- Elemento que se activa por omisión de acción.
- El conductor ha de pulsar un botón cada cierto tiempo indicando que está activo.
- En cuanto transcurra un intervalo de un periodo sin que el conductor haya mandado la señal, el watchdog asume que el conductor está muerto y detiene el tren.



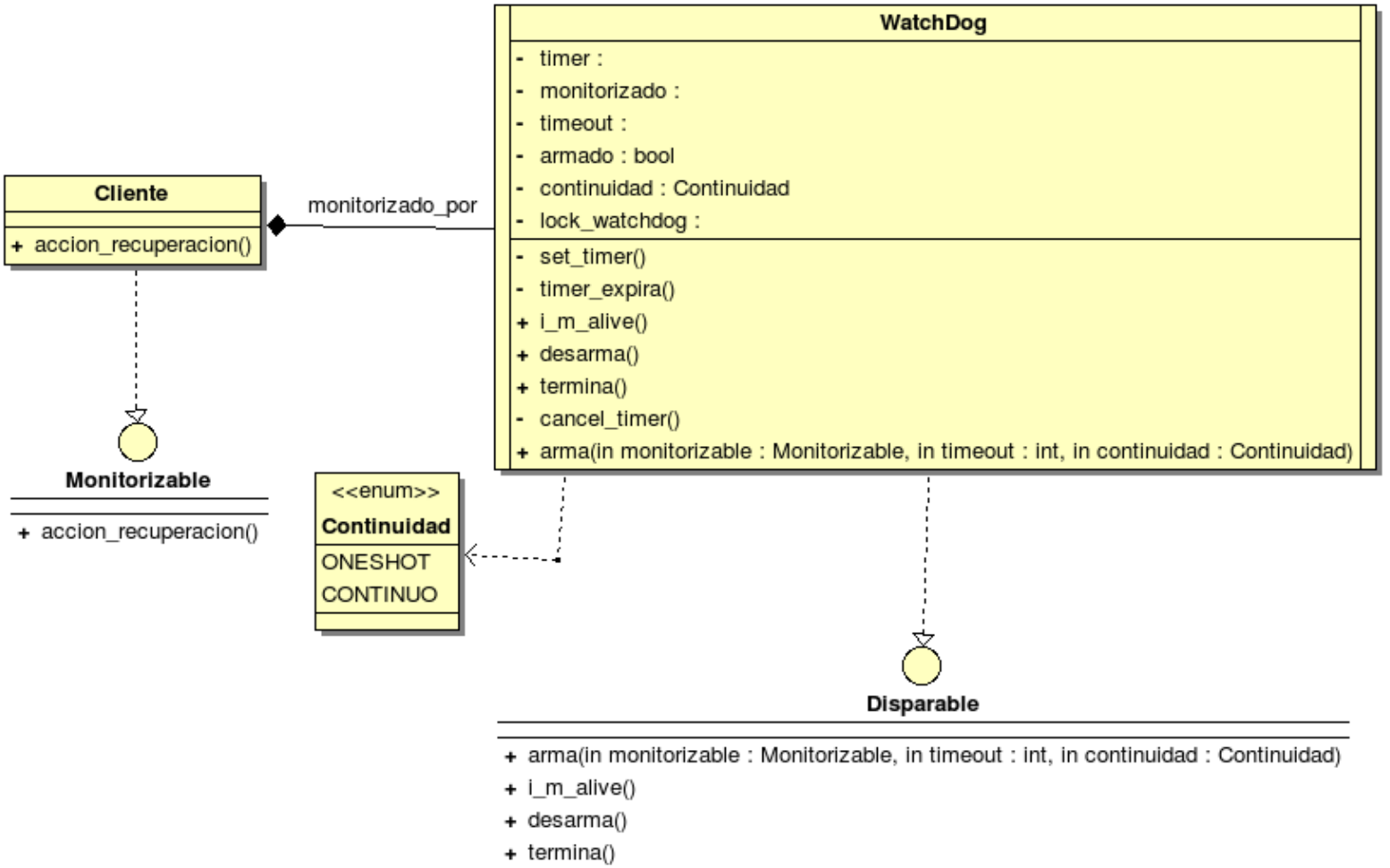


Modos de funcionamiento del Watchdog

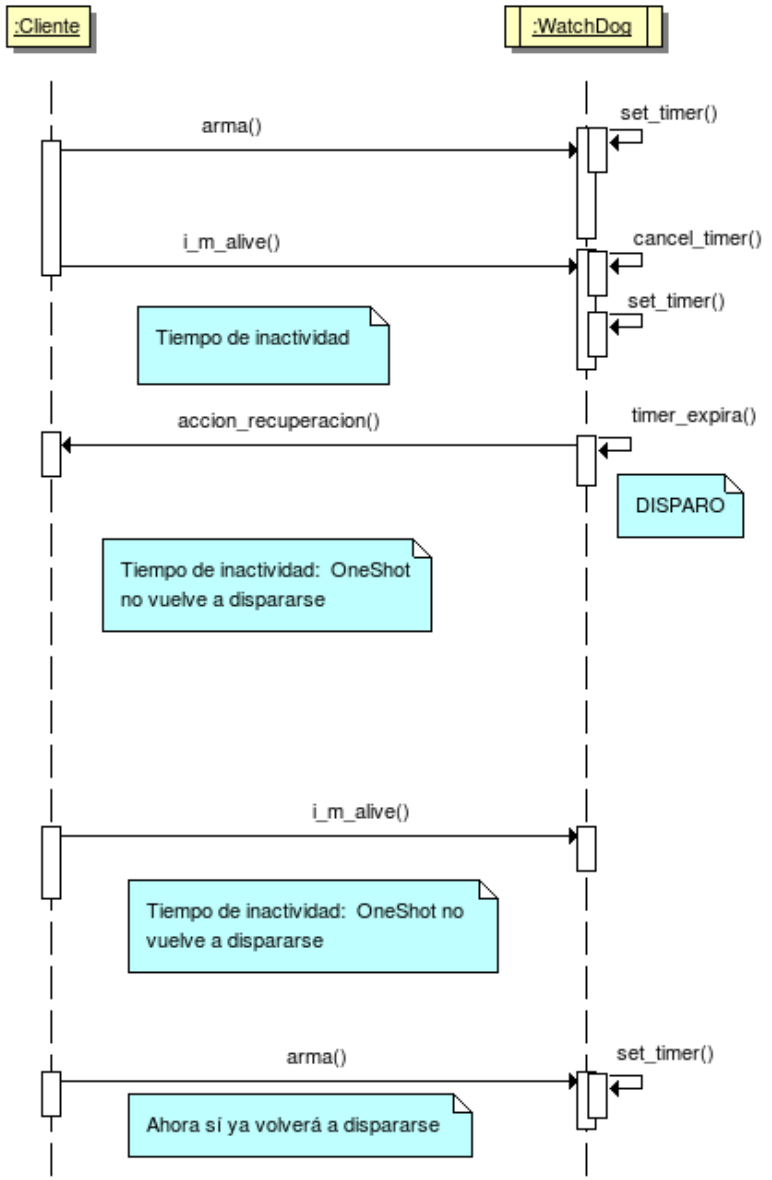
- OneShot:
  - Cuando se pasa el periodo sin actividad se dispara una vez y automáticamente se desarma.
  - Si el periodo de actividad se duplica o triplica no se vuelve a disparar.
  - Es necesario volver a armarlo para que vuelva a dispararse.

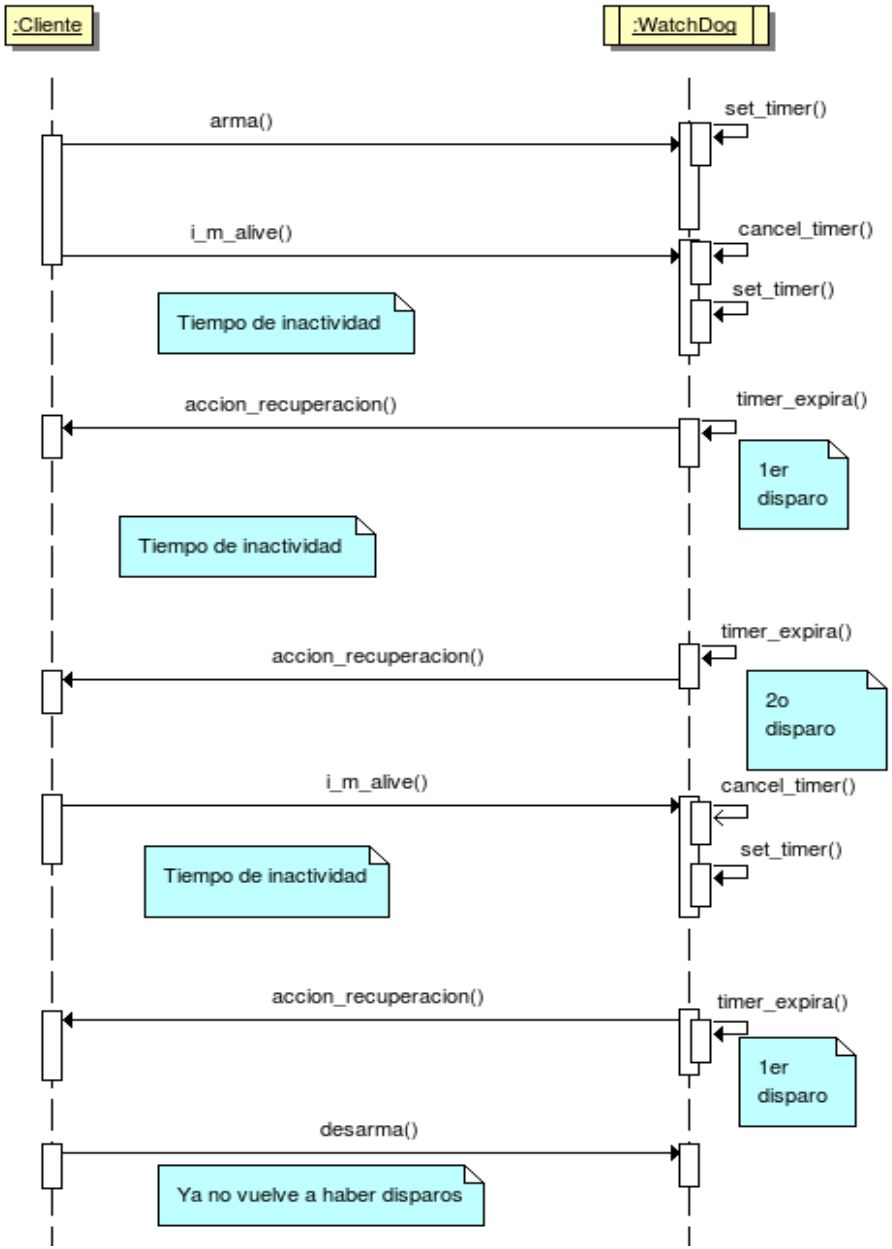
- Continuo
  - Una vez armado siempre que haya un periodo de inactividad se dispara.
  - Cada vez que el periodo de inactividad se duplica se vuelve a disparar
  - Para que cese de dispararse es necesario desarmarlo.

Diagrama de clases



OneShot (izda) Continuo (dcha)





API del Watchdog

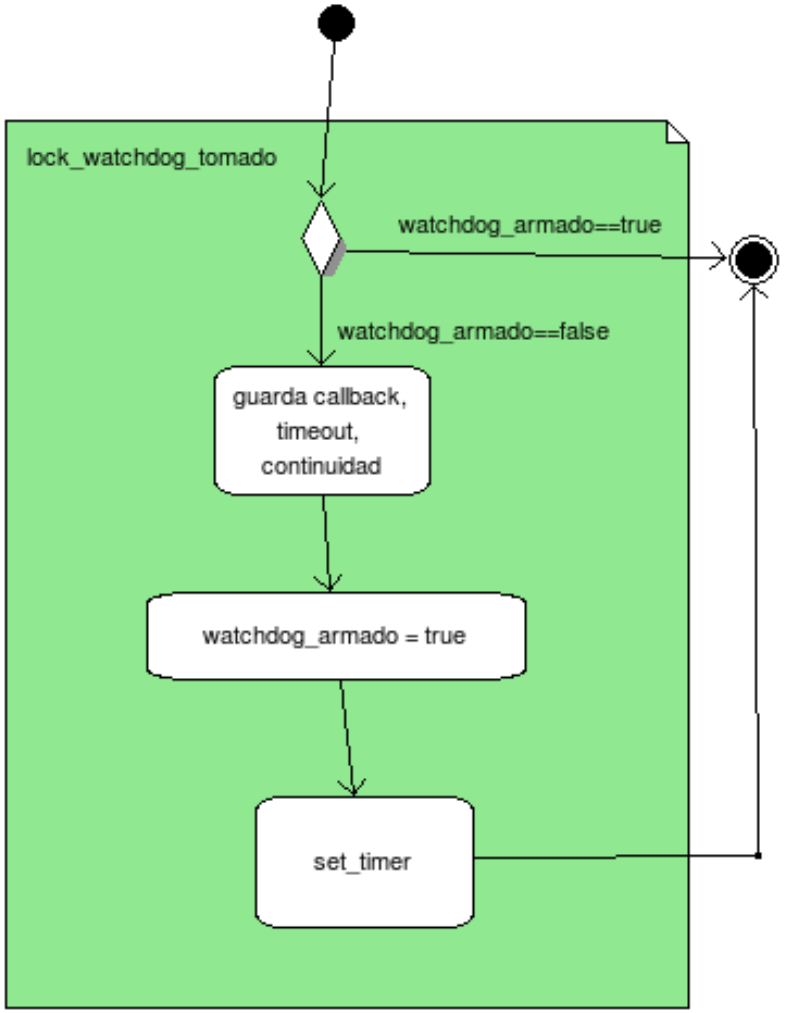
- `boolean arma(callback, timeout, continuidad)`
  - Arma el watchdog en el modo dado por continuidad (CONTINUO o ONESHOT) para que se dispare si hay un intervalo de intactividad de timeout.
  - Para cambiar los parámetros del timer hay que desarmarlo antes.
- `void i_m_alive()`
  - Señala al watchdog que el cliente está vivo. Si el watchdog está armado resetea el timer de inactividad.
- `void desarma()`
  - Desarma el timer, por lo que no se disparará hasta que vuelva a ser armado.
- `void termina()`
  - Termina el timer liberando los recursos. No se puede volver a armar.

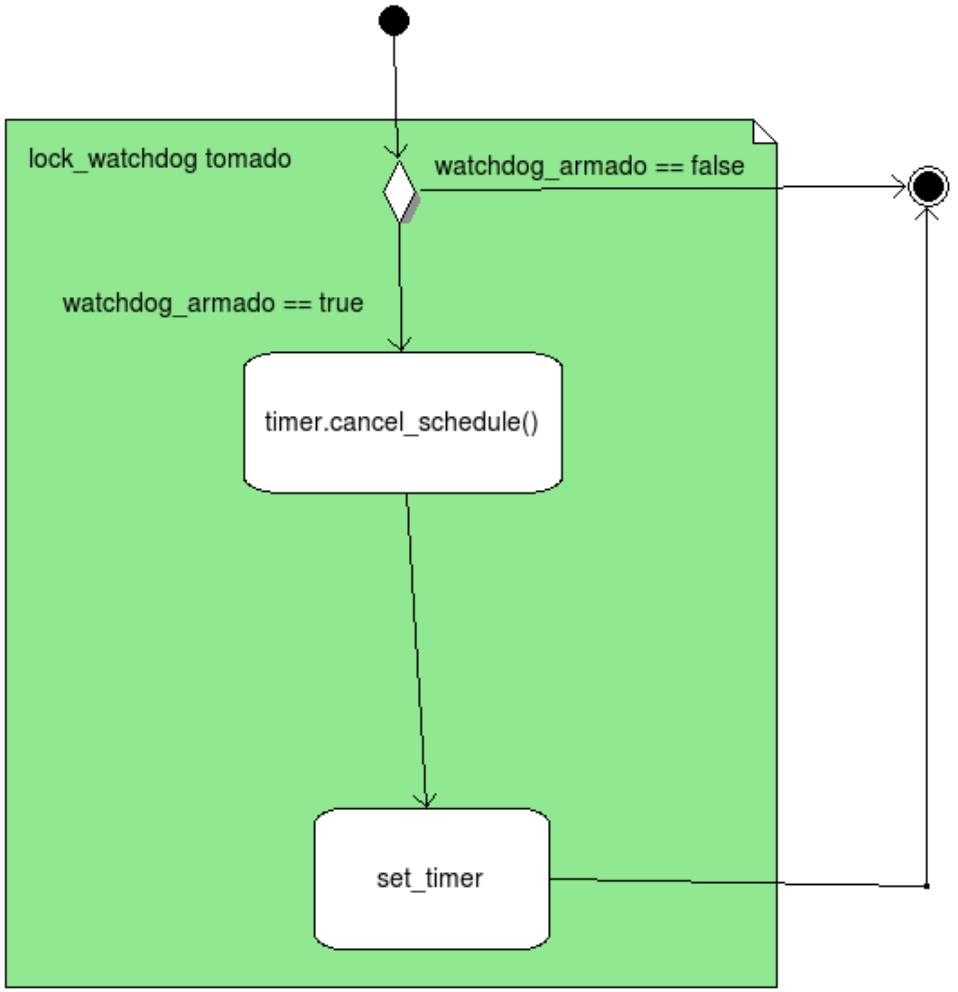
Condiciones

- Mientras el watchdog esté armado no se puede volver a armar.
  - Es necesario que se desarme (en el caso de oneShot también vale una expiración) para poder cambiar los parámetros.
- El callback es llamado desde el thread interno del watchdog.
- Cuando el timer está desarmado el i_m_alive() no produce ningún efecto.
- Las acciones
  - arma(), desarma(), i_m_alive(), expiración, termina()

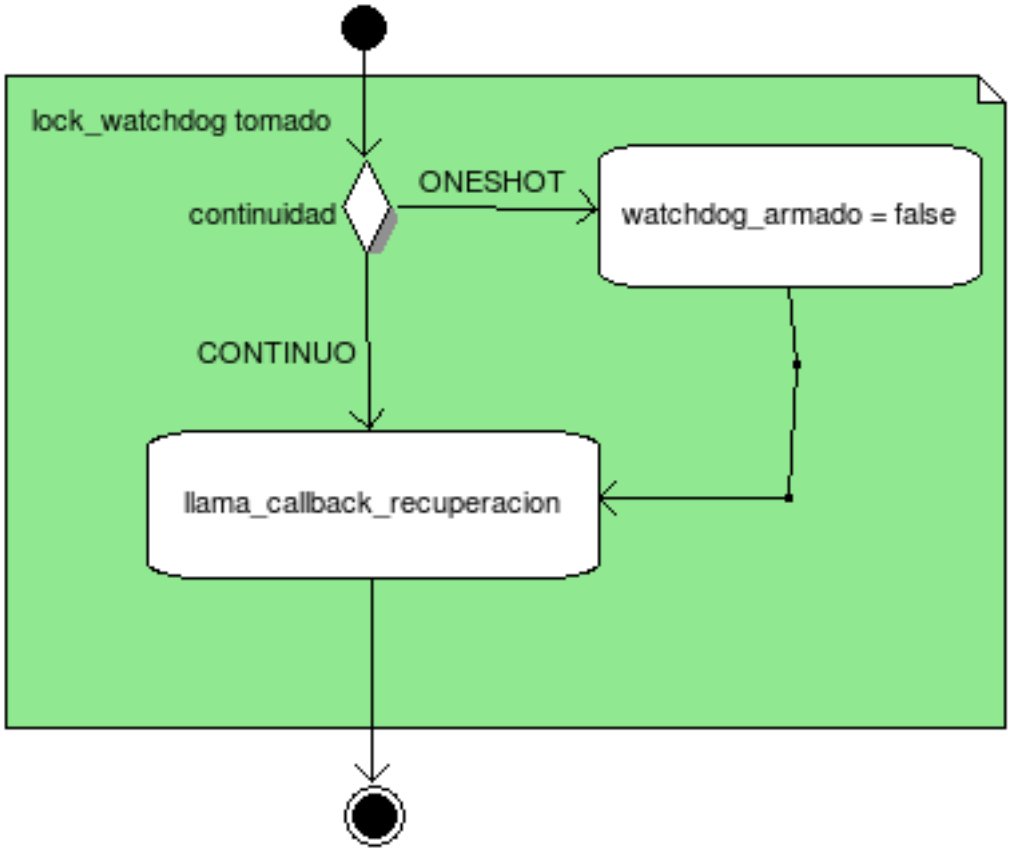
  son susceptibles de venir de threads distintos y por lo tanto deben estar protegidas para que se ejecuten atómicamente.

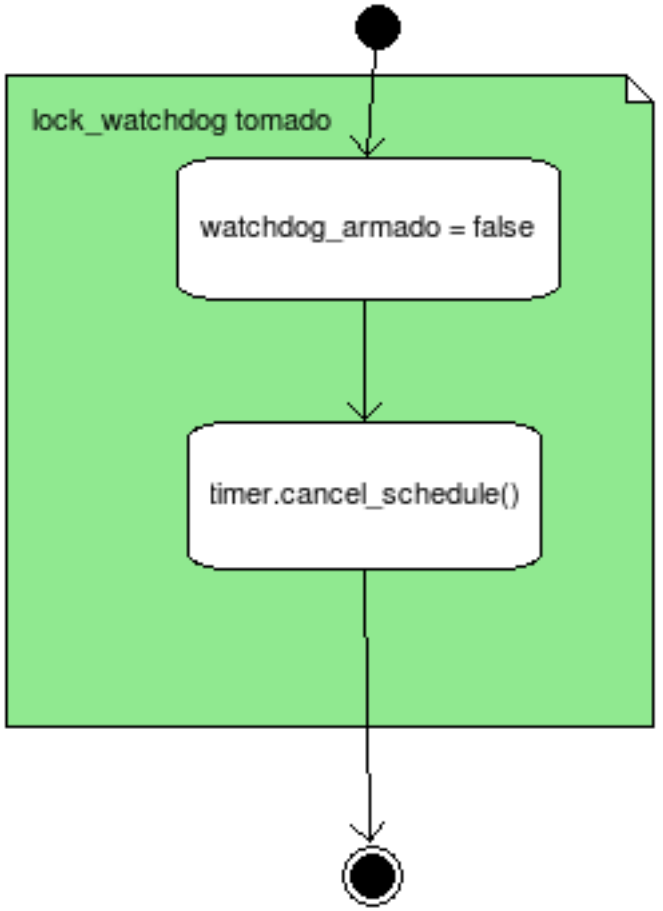
arma (izda) y i_m_alive (dcha)



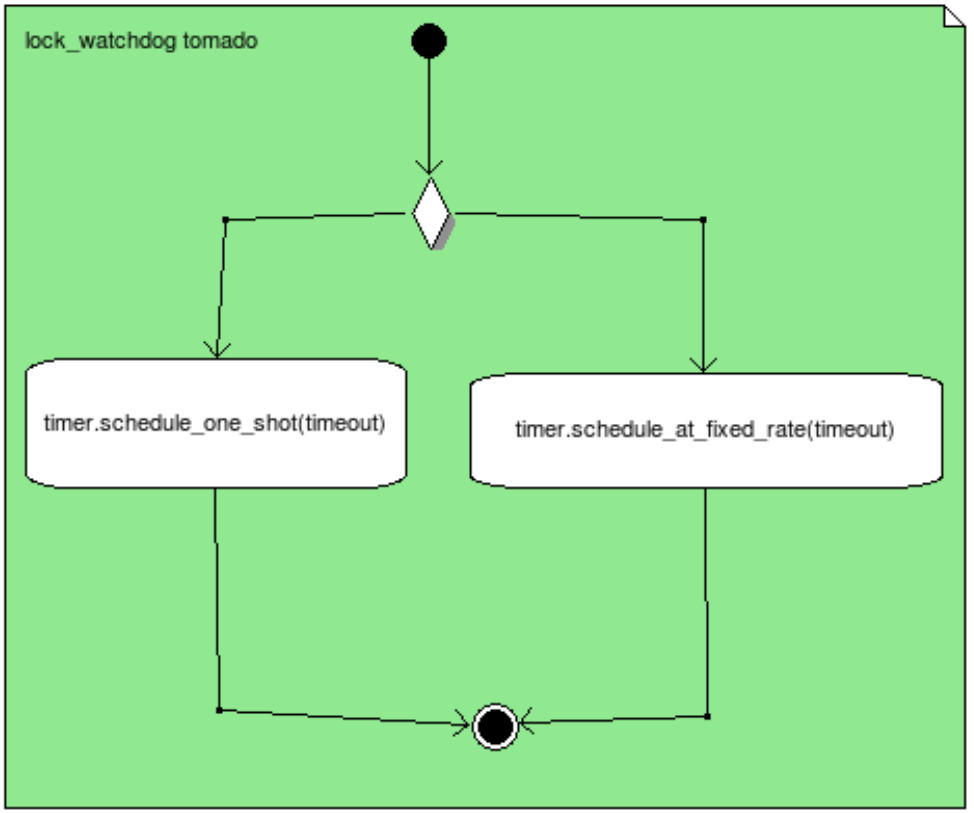


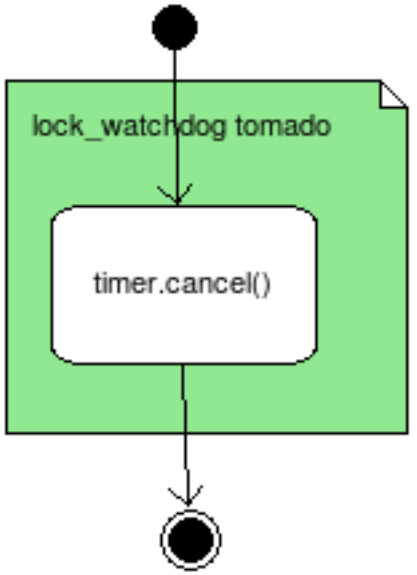
expira_timer (izda) y desarma (dcha)





set_timer y termina





Implementaciones

- Proponemos 3 implementaciones en función del timer interno
  - Con la clase Timer de java
  - Con un ThreadPool de un único thread
  - Con un “timer casero” construido con un thread y locks de wait() y notify()
- Todas las implementaciones son similares

Watchdog Timer: Objetos internos

```
public class Watchdog implements Disparable
{
    private long timeout_ms;
    private Continuidad continuidad;
    private Monitorizable observado;
    // Lock de operaciones del watchdog
    private Object lock_watchdog;
    private boolean watchdog_armado;
    // Temporizador interno (en este caso un timer)
    private Timer timer;
    private ExpiraTimerTask expira_timertask;

    public Watchdog()
    {
        this.timer = new Timer();
        lock_watchdog = new Object();
        watchdog_armado = false;
    }
    ...
    private class ExpiraTimerTask extends TimerTask
    {
        public void run()
        {
            ...
        }
    }
}
```

Watchdog Timer: Lock watchdog

```
public boolean arma(controlado,
                    timeout_ms,
                    continuidad)
{
     synchronized(lock_watchdog)
     {
          ...
     }
}

public void i_m_alive()
{
     synchronized(lock_watchdog)
     {
          ...
     }
}
```

```
public void desarma()
{
     synchronized(lock_watchdog)
     {
          ...
     }
}

public void termina()
{
     synchronized(lock_watchdog)
     {
          ...
     }
}

private class ExpiraTimerTask extends TimerTask
{
     public void run()
     {
          synchronized(lock_watchdog)
          {
               ...
          }

     }
}
```

Watchdog timer: arma

```
public boolean arma(Monitorizable controlado,
                    long timeout_ms,
                    Continuidad continuidad)
{
    synchronized(lock_watchdog)
    {
        if (watchdog_armado)
        {
            // No armamos el watchdog sin cancelarlo antes
            return false;
        }

        watchdog_armado = true;
        this.observado = controlado;
        this.timeout_ms = timeout_ms;
        this.continuidad = continuidad;

        set_timer();
        return true;
    }
}
```

watchdog Timer: set_timer

```
private void set_timer()
{
    // Necesario porque en los timers una vez el timertask esta
    // cancelado no se puede volver a planificar
    expira_timertask = new ExpiraTimerTask();

    if (continuidad == Continuidad.CONTINUO)
    {
        timer.scheduleAtFixedRate(expira_timertask, timeout_ms,
                                  timeout_ms);
    }
    else if (continuidad == Continuidad.ONESHOT)
    {
        timer.schedule(expira_timertask, timeout_ms);
    }
}
```

Expiracion del timer

```
private class ExpiraTimerTask extends TimerTask
{
    public void run()
    {
        synchronized(lock_watchdog)
        {
            if (continuidad == Continuidad.ONESHOT)
            {
                watchdog_armado = false;
            }
            observado.accion_recuperacion();
        }
    }
}
```

i_m_alive, desarma, termina

```
public void i_m_alive()
{
     synchronized(lock_watchdog)
     {
          if (watchdog_armado)
          {
               expira_timertask.cancel();
               set_timer();
          }
     }
}

public void desarma()
{
     synchronized(lock_watchdog)
     {
          watchdog_armado = false;
          expira_timertask.cancel();
     }
}
```

```
public void termina()
{
     synchronized(lock_watchdog)
     {
          timer.cancel();
     }
}
```

Diferentes implementaciones

	Timer	ThreadPool	Timer casero
timer	Timer (java.lang.Timer)	ScheduledThreadPoolExecutor(1)	TimerCasero
expiraTask	TimerTask	Runnable (con futurible)	Expirable
timer.set_one_shot	new TimerTask() timer.schedule()	executor.schedule()	timer.set_timer_one_shot()
timer.set_fixed_rate()	new TimerTask() timer.scheduleAtFixedRate()	executor.scheduleAtFixedRate()	timer.set_timer_continuo()
timer.schedule_cancel()	timertask.cancel()	futurible.cancel()	timer.cancela_timer()
timer.terminate()	timer.cancel()	executor.shutdown()	timer.termina()

Cena de filósofos con watchdog

- Objetivo:
  - Una vez el **bloqueo producido**, resolverlo obligando a uno de los filósofos a liberar el recurso.
- Idea
  - Un watchdog vigila que haya actividad (por ejemplo cogiendo los palillos).
  - Si el watchdog expira, el comedot toma las medidas necesarias.

Estrategia

- El watchdog se arma...
  - En el comedor al inicializarse con un timeout suficientemente grande.
  - Elegimos un watchdog periódico (para que no haya que rearmarle cuando expira)
  - El propio comedor es notificado con la accion_recuperacion.
- El watchdog se alimenta (i_m_alive)
  - Los filósofos al coger los palillos
  - Salta cuando ningún filósofo coge un palillo durante un intervalo largo
- Procedimiento de recuperación (accion_recuperacion)
  - El comedor elige un filósofo y deduce el palillo que está tomando.
  - Se envia un **interrupt() al thread** (filósofo) y **notify() al lock** (palillo).
  - El filósofo se despierta interrumpido (**InterruptedException**)
    - Libera los recursos y se pone a pensar
- El watchdog se termina
  - El programa principal llama al comedor cuando se han terminado los threads

Cambios en Comedor (I)

- El comedor ahora tiene watchdog y lista_filósofos

```
public class Comedor extends Thread implements Monitorizable
{
    ...
    private Watchdog watchdog;
    private Filosofo lista_filosofos[];

    // Lo llama el programa principal una vez esten instanciados los
    // filosofos
    public void dame_la_lista_de_filosofos(Filosofo[] lista)
    {
        this.lista_filosofos = lista;
    }

    public Comedor(int numeroSillas) {
        super();
        elNumeroSillas=numeroSillas;
        // La construcci?n del GestorSillas y los Palillo se
        // retrasa hasta el procedimiento run() ya que necesita que
        // el Comedor est? construido.
        watchdog = new Watchdog();
        watchdog.arma(this, 5000, Continuidad.CONTINUO);
    }
    ...
```

Cambios en el comedor (II): accion_recuperacion

```
public void accion_recuperacion()
{
    // Todos los filosofos estan bloqueados.
    // Asi que elegimos siempre el mismo
    // id_palillo 0
    // filosofo 3
    int id_palillo = 0;
    int num_filosofo = 3;

    System.out.println("WATCHDOG:  Interrumpimos al filosofo " +
                       num_filosofo + " notificamos palillo " +
                       id_palillo);
    lista_filosofos[num_filosofo].interrupt();

    synchronized(listaPalillos[id_palillo])
    {
        listaPalillos[id_palillo].notify();
    }
}
```

Propagación de la interrupción de Palillo a Filosofo

```
class Palillo
{
    public class SueltaPalilloException extends Exception {};

    public synchronized boolean coge() throws SueltaPalilloException
    {
        while (palilloTomado)
        {
            try {
                wait();
            } catch (InterruptedException e) {
                throw new SueltaPalilloException();
            }
        }
        palilloTomado= true;
        return true;
    }
}

public class Comedor extends Thread implements Monitorizable{

    public Palillo cojoPalilloIzquierdo(Filosofo filosofo)
                                throws Palillo.SueltaPalilloException
    {
        ...
    }
```

Cambios en filosofo

- La interrupción llega esperando al palillo izquierdo

```
case ESPERANDO_IZQUIERDO:
    try
    {
        palilloIzquierdo=elComedor.cojoPalilloIzquierdo(this);
    }
    catch (Palillo.SueltaPalilloException e)
    {
        me_han_interrumpido = true;
    }

    if (me_han_interrumpido)
    {
        System.out.println("Filosofo " + id + " thread " +
                    Thread.currentThread() + " interrumpido");
        elComedor.dejoPalillo(palilloDerecho);
        elComedor.dejoSilla();
        cambiarEstado(EstadoFilosofo.PENSANDO);
    }
    else
    {
        ... // Implementacion del caso normal
    }
```

Arreglos en el programa principal

- Se pasa la lista de filósofos al comedor.
- Se llama al comedor para que termine el watchdog.

```
public static void main(String[] args)
{
...
    Filosofo listaFilosofos[];
    Comedor elComedor=new Comedor(NUMERO_SILLAS);

    listaFilosofos=new Filosofo[Comedor.numeroPlatos()];
    ... // instantacion de los filosofos

    elComedor.dame_la_lista_de_filosofos(listaFilosofos);

    ... // Se espera y se les manda el termina y el join

    System.out.println("MAIN: Pido que se pare el watchdog");
    elComedor.para_el_watchdog(); // Aqui le manda un termina()
}
```

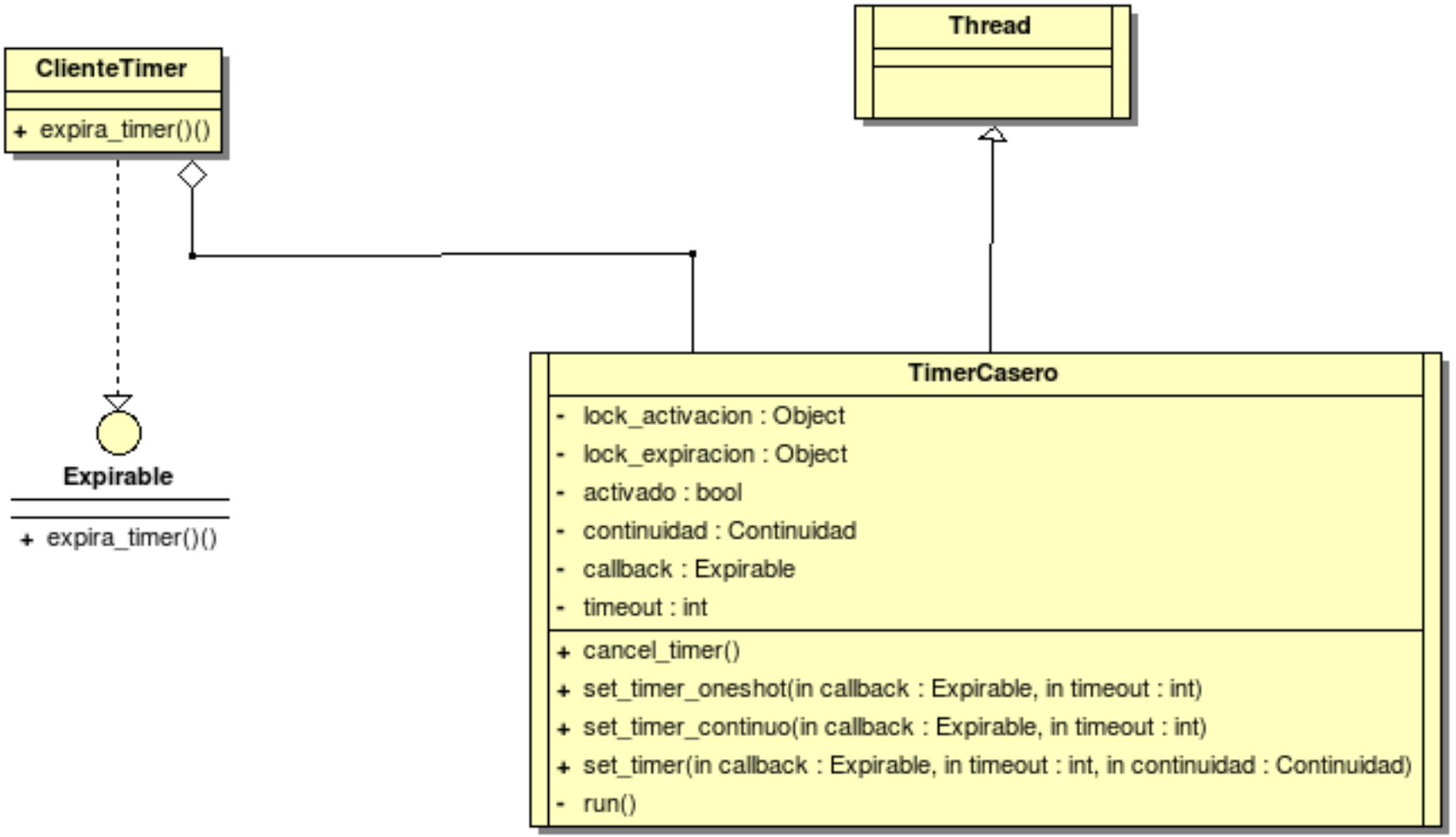
Timer casero
(avanzado)

Timer casero

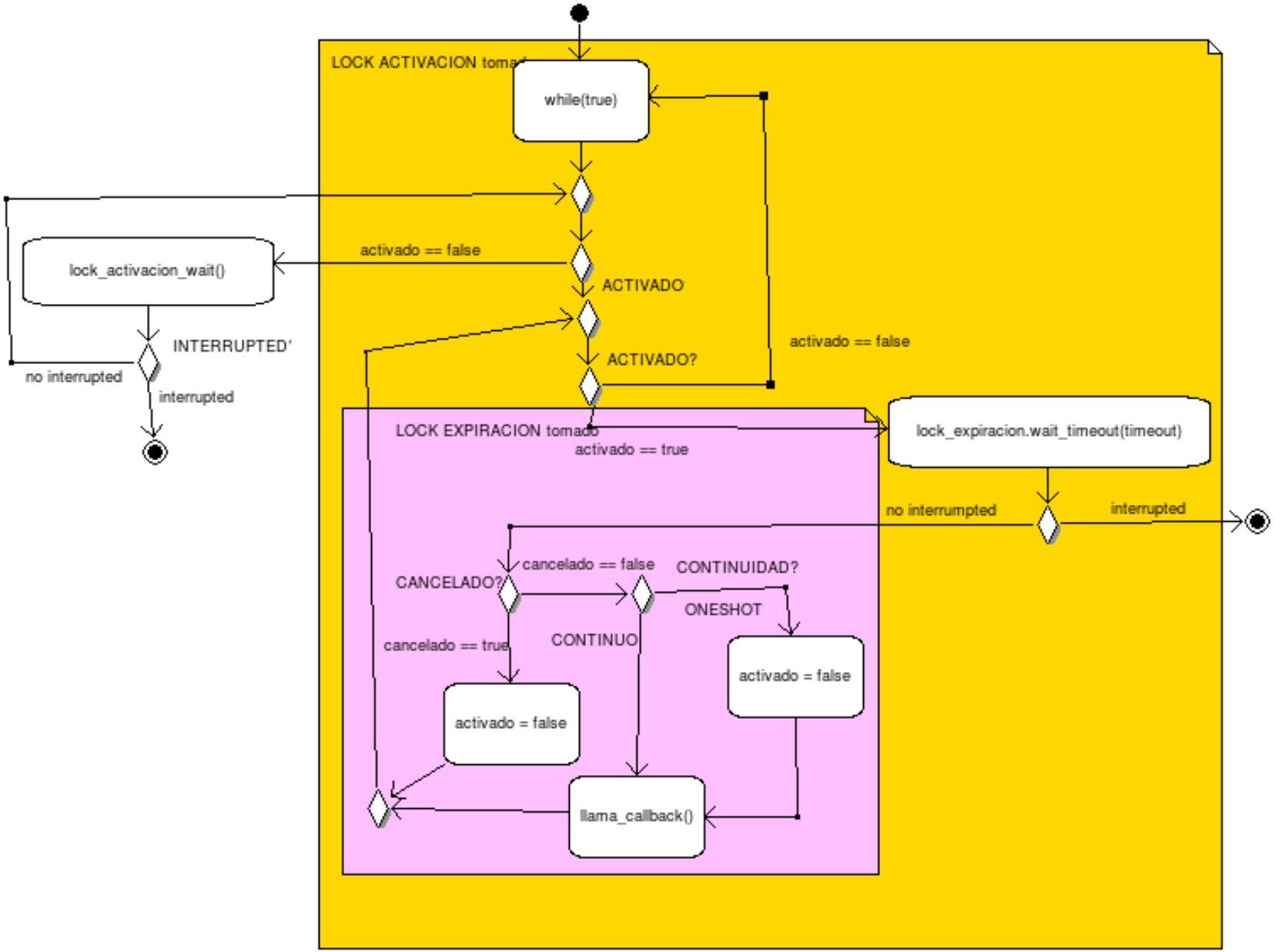
- Implementación mínima de la clase Timer para ser usada desde el watchdog.
  - Sólo guarda un único posible evento (periódico o one-shot)
  - Las llamadas a set_timer(), cancel_timer y termina() no se mezclan

- Consideramos como threads
  - Thread de las llamadas
  - Thread interno

Diagrama de clases



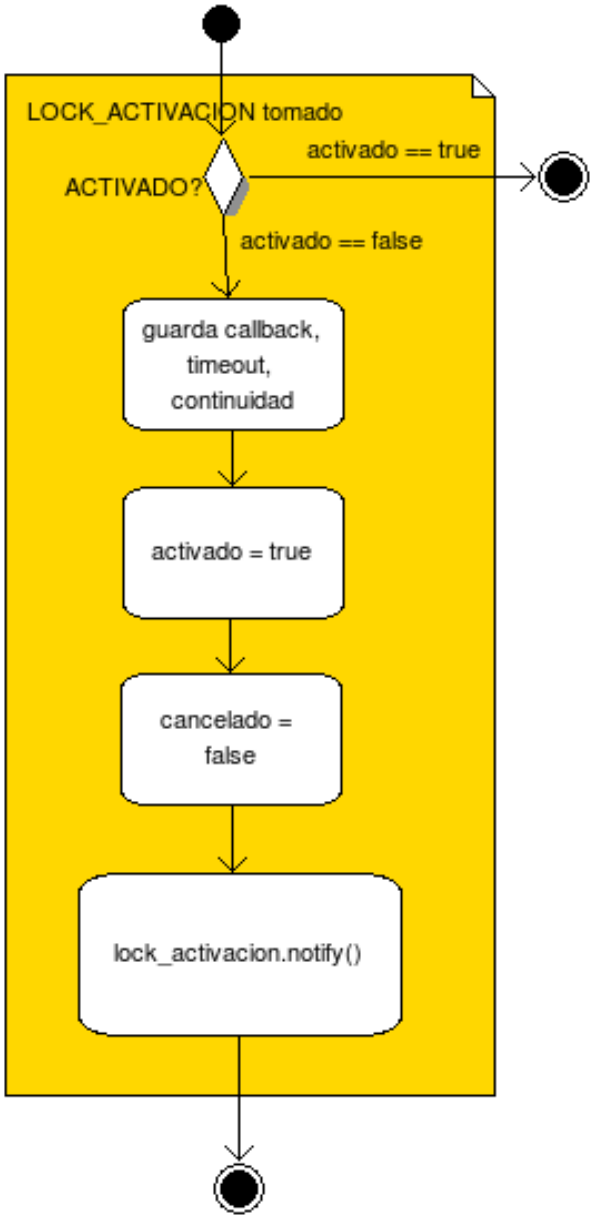
Thread del timer

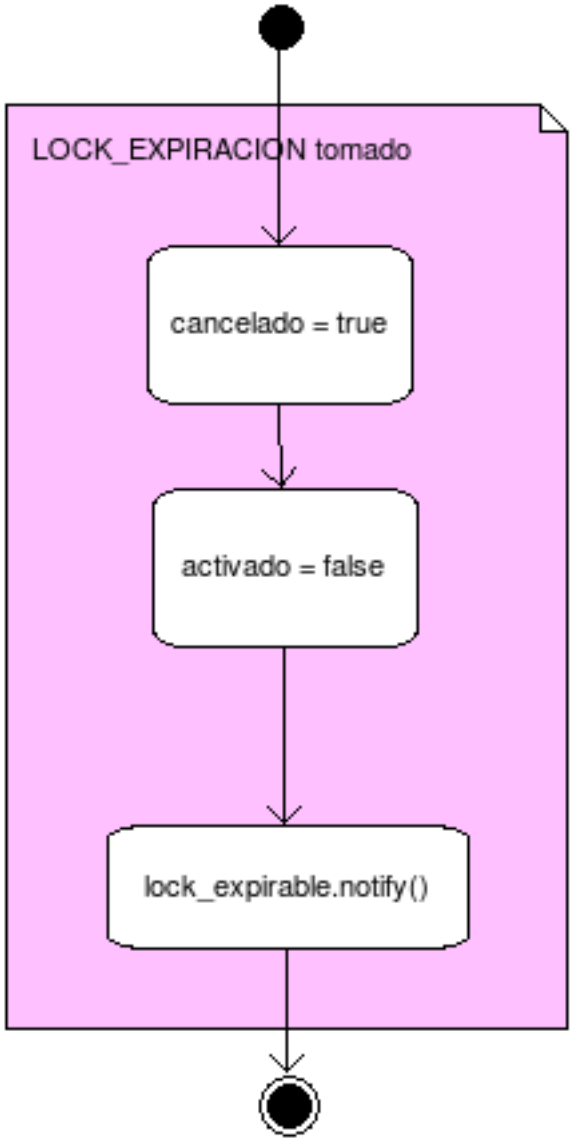


Claves de la implementación

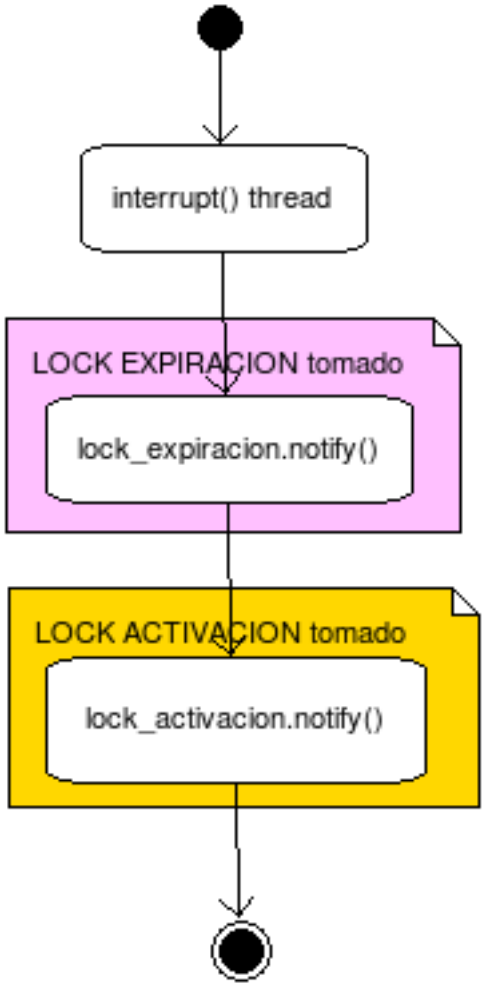
- Tenemos 2 locks
  - **lock_activacion**
    - Usado por **set_timer()** para notificar que el timer está activado.
    - El thread interno espera en él cuando no hay ningún schedule pendiente.
  - **lock_expiracion**
    - El thread interno hace espera en él con un timeout. Sale de él de 2 formas:
      - Porque ha sido cancelado (llega un notify()) de cancela_timer()
      - Porque se ha cumplido el timeout (entonces llama al callback)
    - El atributo **cancelado** sirve para distinguir ambos casos.
  - El wait con timeout de lock_activacion está incluído dentro de la sección critica del thread interno.
    - **set_timer()** sólo interviene con el thread interno en el **lock_activacion**.
- La terminación se hace interrumpiendo el thread
  - También se podría hacer con un flag **terminado** en el while infinito

set_timer() y cancel_timer()





termina_timer



Clase TimerCasero y atributos privados

```
public class TimerCasero extends Thread
{
    private enum EstadoTimer {DESACTIVADO, ACTIVADO};
    private enum Continuidad {ONESHOT, CONTINUO};

    private Expirable callback;
    private Continuidad continuidad;
    long timeout;

    Object lock_activacion;
    Object lock_expirable;
    boolean cancelado = false;
    private EstadoTimer estado_timer = EstadoTimer.DESACTIVADO;

    public TimerCasero()
    {
        lock_activacion = new Object();
        lock_expirable = new Object();
        start();
    }
    ...
}
```

Método run del thread interno (1)

```
public void run()
{
    synchronized(lock_activacion)
    {
        while(true)
        {
            // Esperamos la activacion
            while (estado_timer == EstadoTimer.DESACTIVADO)
            {
                try {
                    lock_activacion.wait();
                } catch (InterruptedException e) {
                    return;
                }
            }

            // Esperamos la desactivacion
            ...
```

Método run del thread interno (2)

```
                // Esperamos la desactivacion
                while (estado_timer == EstadoTimer.ACTIVADO )
                {
                    synchronized(lock_expirable)
                    {
                        try {
                            lock_expirable.wait(timeout + 20);
                        } catch (InterruptedException e) {
                            return;
                        }
                        if (!cancelado)
                        {
                            if (continuidad == Continuidad.ONESHOT)
                            {
                                estado_timer = stadoTimer.DESACTIVADO;
                            }
                            callback.expira_timer();
                        }
                        else
                        {
                            estado_timer = EstadoTimer.DESACTIVADO;
                        }
                    } // End synchronized(lock_expirable) desactivacion
                } // End while (estado_timer ACTIVADO)
            } // End while (true)
        } // End synchronized(lock_activacion)
    } // End de la funcion run()
```

set_timer() y cancel_timer()

```
private boolean set_timer(Expirable expirable,
                          long timeout, Continuidad continuo)
{
    synchronized(lock_activacion)
    {
        this.callback = expirable;
        this.timeout = timeout;
        this.continuidad = continuo;

        estado_timer = EstadoTimer.ACTIVADO;
        lock_activacion.notify();
        cancelado = false;

        return true;
    }
}
```

```
public void cancel_timer()
{
    synchronized(lock_expirable)
    {
        cancelado = true;
        estado_timer = EstadoTimer.DESACTIVADO;
        lock_expirable.notify();
    }
}
```

termina()

```
public void cancel_timer()
{
     synchronized(lock_expirable)
     {
          cancelado = true;
          estado_timer = EstadoTimer.DESACTIVADO;
          lock_expirable.notify();
     }
}
```

Limitaciones del timer casero

- Los métodos set_timer(), cancela_timer() y termina() no están sincronizados entre sí.
  - Si se llamasen en paralelo no se mantendría la consistencia.
  - Solución en el watchdog:
    - Se usa el lock_watchdog()
    - En el i_m_alive() cancel_timer() y set_timer() se llamanserializadamente.
  - Solución general:
    - Añadir un tercer lock (lock_operaciones) y usarlo en cada una de ellas.
- El método set_timer() es bloqueante si se llama cuando el timer está armado (puede tener que esperar todo el timeout)
  - Solución en el watchdog: Se usa la barrera watchdog_armado antes de llamar a set_timer().
  - Solución general. Añadir un nuevo booleano de barrera (distinto de estado y de cancelado) y protegerla con un 4º lock.