2.4. Programación orientada a procesos

Una de las principales limitaciones de un simulador orientado a sucesos es que los sucesos se ejecutan de forma secuencial, aunque ocurran en el mismo tiempo de simulación. Esto implica que el tiempo de ejecución sea directamente proporcional al número de sucesos ejecutados por el simulador, que suele ser bastante elevado. Además, el planificador de sucesos puede llegar a desbordar la memoria cuando los sucesos ocurren en tiempos muy próximos.

Como alternativa, se propone el concepto de *proceso*, que trata de paliar estas limitaciones. Con los procesos vamos a tratar de identificar las secuencias de sucesos *recurrentes* que pueden ejecutarse en paralelo. Para ello vamos a utilizar el diagrama de sucesos. Por ejemplo, en el diagrama de la figura 2.1 podemos identificar dos secuencias recurrentes: el suceso periódico de llegada de clientes (suceso ①), y la secuencia de sucesos ③ y ④, que representa el procesamiento de los clientes en la estación de servicio. Fíjate que se trata de dos secuencias repetitivas que pueden ejecutarse en paralelo.

De forma esquemática, podríamos expresar estos procesos como sigue:

Proceso 1	**Proceso 2**
Repite	Mientras c2
Ejecuta ①	Ejecuta ③
Espera Ta	Espera Ts
	Ejecuta ④

Según el diagrama de sucesos, Ta es el tiempo entre llegadas, Ts es el tiempo de servicio, y c2 es la condición de que la cola no esté vacía. El código de los sucesos de este ejemplo se describieron al principio del tema.

En los procesos hemos introducido la instrucción Espera para indicar que el proceso se detiene y planifica su reanudación según el tiempo especificado. En realidad, este será tiempo de simulación (según el reloj de la simulación), y no tiene relación directa con el de ejecución (tiempo de CPU). Más tarde veremos como se gestiona la ejecución de estos procesos.

De lo anterior, podemos deducir que un proceso puede estar en uno de los siguientes estados:

- *Activo*, cuando el proceso está ejecutándose.
- *Planificado*, cuando el proceso está detenido esperando su reanudación para un determinado tiempo.
- *Pasivo*, cuando no está ni planificado ni activo, aunque puede ser activado o planificado por otro proceso.
- *Terminado*, cuando el proceso ha finalizado totalmente su ejecución. Un proceso terminado ya no podrá ser nunca más activado ni planificado.

Volviendo al ejemplo anterior, todavía tenemos que ubicar el suceso ② en los procesos anteriores. Dado que la ejecución de este suceso planifica la ejecución del suceso ③, ubicado en el Proceso 2, éste sólo puede colocarse en el Proceso 1. La conexión entre ambos procesos se realiza con la instrucción Activa, la cual permite activar un proceso determinado desde otro proceso distinto. Con esta instrucción estamos implementando la planificación del Proceso 2 desde el Proceso 1. La implementación definitiva de estos dos procesos sería la siguiente:

Proceso 1	**Proceso 2**
1 Repite	1 Repite
2 Ejecuta ①	2 Mientras c2
3 Ejecuta ②	3 Ejecuta ③
4 Si c1	4 Espera Ts
5 Activa Proceso 2	5 Ejecuta ④
6 Espera Ta	6 Cancela

```
1   Inicializa tiempo de simulación
2   Inicializa variables de estado
3   Inicializa planificador de procesos
4   Mientras hayan procesos activos y/o planificados hacer
5       Si no hay proceso activo
6           Selecciona el siguiente proceso planificado con tiempo mínimo T
7           Avanza el tiempo de simulación a T
8           Ejecuta el proceso seleccionado (pasa a activo)
9   Muestra los resultados de la simulación
```

Figura 2.5: Algoritmo para la ejecución de procesos.

Normalmente, los procesos recurrentes se ejecutan de forma indefinida utilizando un bucle infinito. Fíjate que en el Proceso 2 hemos introducido además la instrucción Cancela, para indicar que este proceso pasa a un estado *pasivo* cuando no tenga clientes que procesar (la cola se ha quedado vacía). En este caso, cuando llegue un nuevo cliente en el Proceso 1 se activará de nuevo el Proceso 2, y pasará a procesar los nuevos clientes de la cola.

Es importante destacar que si el Proceso 2 no incluyese el bucle infinito, al volverse a despertar su hilo terminaría (pasaría al estado *terminado*), y no podría volverse a activar nunca más.

Como puedes apreciar, el diagrama de sucesos se ha simplificado a dos procesos: el proceso de llegadas y el proceso de servicios. Con esta nueva visión, conseguimos reducir el número de planificaciones, y además permitimos su ejecución en paralelo.

Para realizar la ejecución de un conjunto de procesos, tendremos que mantener al menos dos estructuras de datos:

- *Procesos Planificados*, que es una lista con los procesos que están esperando su reanudación en un tiempo determinado. Esta estructura es muy parecida a la de los sucesos planificados que vimos en la sección anterior.
- *Procesos Activos*, que contiene los procesos que están siendo ejecutados actualmente.

El algoritmo de ejecución de un simulador con procesos es muy parecido al que hemos visto para los simuladores orientados a sucesos. En la figura 2.5 se resume este algoritmo. El simulador tiene que gestionar ahora los procesos planificados, de tal forma que si en un momento dado no hay ningún proceso activo, se saca del planificador el proceso "dormido" que tenga menor tiempo de simulación, se avanza el reloj de simulación y se activa dicho proceso.

Veamos ahora parte de la ejecución de los procesos del ejemplo mediante la traza de la figura 2.6. Junto a cada proceso hemos indicado la línea de código que se está ejecutando (o planificando) en cada momento. Para esta traza se ha tomado un tiempo entre llegadas de $Ta = 0{,}5$ y un tiempo de servicio de $Ts = 0{,}6$. Se deja como ejercicio completar la traza hasta procesar cuatro clientes. En la traza hemos sombreado los procesos planificados por el proceso activo en esos momentos.

2.5. La biblioteca de JavaSim

La mayor parte de los simuladores y paquetes de software comerciales para la simulación de sucesos discretos se basa en procesos, por ejemplo GPSS, SIMSCRIPT II.5 y SLAM II. Puedes consultar la bibliografía si quieres conocer más detalles de estos lenguajes.

Planificador			Estación		
T_{sim}	*Activo*	*Planificados*	$\sharp Cliente$	*Servidor*	*Cola*
0	P1(1)	-	-	libre	[ ]
0	P1(2)	-	1	”	”
0	P1(3)	-	”		[1]
0	P1(4,5)	P2(1),0.0	”	”	”
0	P1(6)	P2(1),0.0	”	”	”
0		P1(1),0.5			
0	-	P2(1),0.0 √	”	”	”
0		P1(1),0.5			
0	P2(1,2)	P1(1),0.5	”	”	”
0	P2(3)	P1(1),0.5	”	ocupado	[ ]
0	P2(4)	P1(1),0.5	”	”	”
0		P2(5),0.6			
0	-	P1(1),0.5 √	”	”	”
0		P2(5),0.6			
0.5	P1(1)	P2(5),0.6	-	”	”
0.5	P1(2)	P2(5),0.6	2	ocupado	”
0.5	P1(3)	P2(5),0.6	”	”	[2]
0.5	P1(4)	P2(5),0.6	”	”	”
0.5	P1(6)	P2(5),0.6	”	”	”
		P1(1),1.0			
0.5	-	P2(5),0.6 √	-	”	”
		P1(1),1.0			
0.6	P2(5)	P1(1),1.0	1	”	”
0.6	P2(2,3)	P1(1),1.0	2	”	[ ]
0.6	P2(4)	P1(1),1.0	”	”	”
		P2(5),1.2			
...	...	...	...	...	...

Figura 2.6: Traza de ejecución basada en procesos.

Posiblemente, uno de los percusores del paradigma orientado a procesos ha sido SIMULA-67. Este ha sido uno de los primeros lenguajes orientados a objeto con nociones de concurrencia (o pseudo-paralelismo). Además fue percusor de otros lenguajes bien conocidos como Algol-68 y Pascal. El lenguaje que vamos a ver en este tema, JavaSim, está basado en las primitivas propuestas en el lenguaje SIMULA-67.

JavaSim tiene sus raíces en un proyecto de investigación de aplicaciones distribuidas en C++ (C++Sim) [1], y su principal característica es la definición de los procesos de simulación como *hilos* independientes de ejecución (o *threads*).

De forma resumida, un *hilo* consiste en un fragmento de código que se ejecuta de forma *simultánea* al programa (o hilo) que lanzó a ejecutar dicho fragmento. La palabra *simultánea* no debe tomarse al pie de la letra, ya que si el ordenador no tiene varios procesadores no podemos hablar de paralelismo, sino de pseudo-paralelismo. La programación por hilos ha tenido un fuerte auge a partir del desarrollo de entornos multimedia, ya que la detección de sucesos en entornos de usuario (pulsar ratón, animaciones, etc.) se realiza mediante la ejecución simultánea de varios hilos. También en sistemas distribuidos, especialmente los sistemas cliente/servidor, utilizan la programación multi-hilo en los servidores. Así, cada petición de un cliente se procesa en un hilo diferente.

En JavaSim un proceso de simulación es una clase especial de hilo, donde además debe registrarse un tiempo de simulación. A continuación vamos a describir los aspectos más relevantes de JavaSim, dejando que los detalles sean consultados en el manual de referencia.

2.5.1. La clase Thread

Un *hilo* es un objeto que tiene la propiedad de poder ejecutar código de forma concurrente al programa o hilo que lo invoca. El código a ejecutar se especifica dentro de un método especial del objeto; en Java es el método `run()` del interfaz Runnable, el cual es implementado por la clase Thread. Este método sólo se ejecuta cuando se activa el hilo, y se puede *congelar* o *cancelar* a sí mismo o desde otros hilos.

Bajo este enfoque, un programa es en sí mismo un hilo, denominado main, que puede congelarse y terminar como el resto de hilos. Sin embargo, la finalización del hilo main supone la finalización de todos los hilos derivados de su ejecución.

Originalmente, la clase Thread tenía en las versiones iniciales de Java los siguientes métodos para gestionar y sincronizar la ejecución de los hilos de un programa:

- `start()`: el cual inicia la ejecución de un hilo.
- `stop()`: el cual finaliza la ejecución de un hilo, generando una excepción por la interrupción.
- `suspend()`: que congela la ejecución de un hilo.
- `resume()`: que reanuda la ejecución de un hilo congelado.

Todos estos métodos han quedado obsoletos a partir de la versión 1.4 de Java debido a problemas de interbloqueos que se producían de forma aleatoria en la ejecución de programas multi-hilo [2].

El mecanismo alternativo propuesto para la sincronización de hilos consiste en la utilización de los métodos `wait()` y `notify()` de la clase Object, que es la clase de la cual heredan todas las clases de Java, incluida la clase Thread. El método `wait()` permite detener el hilo que se está ejecutando actualmente, mientras que `notify()` permite poner en marcha desde el hilo activo un hilo previamente detenido.

Veamos un ejemplo del uso de estos métodos. En primer lugar definiremos una clase para un hilo que simplemente espera un determinado tiempo y devuelve el control al hilo principal (main).

[1] Proyecto Arjuna:http://www.distribution.cs.ncl.ac.uk/history/

[2] http://java.sun.com/j2se/1.4.1/docs/guide/misc/threadPrimitiveDeprecation.html


```
import java.lang.Thread.*

public class Ciclo extends Thread
{
   public Object cerrojo= new Object();

   public void run() {
      try {
         sleep(2000); //Duerme 2000 ms
         synchronized(cerrojo)
         {
             cerrojo.notify(); //despierta el hilo main
         }
   }
}
```

La forma de activar este hilo desde el hilo principal se realiza como sigue:

```
public class Main
{
   public static void main() {
      Ciclo p= new Ciclo(); //Creamos el hilo
      p.start(); //Arrancamos el hilo
      synchronized(p.cerrojo)
      {
         try
         {
             cerrojo.wait(); //duerme el hilo main
         }
         catch (InterruptedException e)
         {
            System.exit(1); //ejecución interrumpida
         }
      }
      System.exit(0); //finaliza el programa
   }
}
```

2.5.2. La clase SimulationProcess

Los procesos de simulación que hemos definido en las secciones anteriores se implementan con la clase SimulationProcess, que es una subclase de Thread. Su interfaz se resume a continuación:

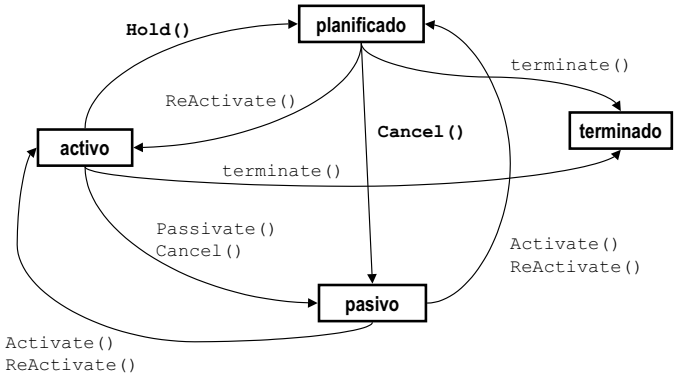

Figura 2.7: Transiciones entre estados de un proceso.

SimulationProcess
+ Object cerrojo
+ void run() + void Activate() + void ActivateBefore(SimulationProcess p) + void ActivateAfter(SimulationProcess p) + void ActivateAt(double AtTime,boolean prior) + void ReActivate() + void Cancel() + void terminate() + void finalize() + final double Time() + final double evtime() + static SimulationProcess current() + static double CurrentTime() + boolean idle() + boolean passivated() + boolean terminated() # void Hold(double t) # void SimulationProcess()

Debido a que el contructor de esta clase está protegido, siempre que queramos implementar un proceso, debermos crear para él una subclase de SimulationProcess. Una vez definida esta clase, el código del proceso se incluirá en su método `run()`.

Buena parte de los métodos anteriores sirven para planificar y coordinar distintos procesos. En la figura 2.7 relacionamos estos métodos con los estados de un proceso.

Otros métodos de interés de esta clase son:

- `CurrentTime()` y `Time()` que nos proporciona el tiempo actual de simulación. La diferencia entre ambos se debe a que el primero puede utilizarse sin instanciar la clase (static).
- `current()` y `next_ev()` devuelven respectivamente el proceso que se está ejecutando actualmente y el siguiente proceso planificado a ser ejecutado (será NULL si no hay planificados).
- `evtime()` devuelve el tiempo en el que el proceso ha planificado su reanudación. Si no está planificado devuelve la constante `Never`.

Ejemplos. A modo de ejemplo vamos a mostrar como se implementarían los dos procesos del ejemplo anterior. Al Proceso 1 que representa el proceso de llegadas le denominaremos Llegadas, y al Proceso 2 que representa el proceso de servicios, lo denominaremos Estacion. En la tabla siguiente se muestra por un lado los atributos de las clases (primera fila), y el método `run()` que implementa su funcionamiento. Recuerda que ambas clases deben ser subclases de SimulationProcess.

Llegadas.java	**Estacion.java**

`public Estacion M;`
`private double Ta=0.5;`
	`public Cola Q = new Cola();`
	`public boolean funciona=false;`
	`public int procesados;`
	`private double Ts=0.6;`

`public void run() {`
`  for(;;){`
`  //Suceso ①`
`  Cliente J = new Cliente();`
 	
`  //Suceso ②`
`  M.Q.Encola(J);`
`  vacia=M.Q.Vacia()`
`  if (Vacia && !M.funciona)`
`  M.Activate();`
`  Hold(Ta);`
`  }`
`}`
	`public void run() {`
	`  for(;;){`
	`  funciona=true;`
	`  while(!Q.Vacia()){`
	`  //Suceso ③`
	`  J=Q.Desencola();`
	`  Hold(Tserv);`
	`  //Suceso ④`
	`  procesados++;`
	`  }`
	`  funciona=false;`
	`  Cancel();`
	`  }`
	`}`

En el código anterior hemos omitido la captura de excepciones (sentencias `try`) para hacerlo más legible. Concretamente, los métodos `Hold` y `Activate` exigen la captura de la excepción `SimulationException`. Esta excepción se lanzará por ejemplo cuando se intente planificar un proceso con tiempos negativos o se intente activar un proceso terminado.

La variable `Estacion M` contiene la referencia al objeto de la clase Estacion que recibe los clientes del proceso de llegadas. Esta referencia puede pasarse en el constructor de Llegadas o bien utilizar una variable estática externa. Fíjate también que la variable `Q` es la variable que representa la cola de la estación.

2.5.3. La clase Scheduler

La clase Scheduler es la encargada de gestionar la ejecución de los procesos. En otras palabras, constituye la estructura de datos que maneja los procesos planificados y activos. Al igual que SimulationProcess, esta clase también constituye un hilo, es decir es una subclase de Thread.

El interfaz de esta clase es como sigue:

Scheduler
+ Scheduler()
+ static void stopSimulation ()
+ static void startSimulation()
+ static boolean simulationReset()
+ static void reset()
+ static double CurrentTime()

Esta clase es abstracta, con lo cual no debe crearse ningún objeto para ella. Para utilizarla procederemos como sigue. Antes de iniciar el ciclo de simulación activaremos el planificador

con `startSimulation()`, y al finalizar la simulación lo detendremos con `stopSimulation()`. En algunos casos, sobre todo para el tratamiento de los resultados, necesitaremos reiniciar varias veces la simulación. Para ello, se utiliza el método `reset()`, y para saber si el simulador se está reiniciando, se usa el método `simulationReset()`.

Por ejemplo, vamos a ver como se implementa la clase que representa el ciclo de simulación para una estación M/M/1.

```
public class Simulador extends SimulationProcess
{
     public void run()
     {
        try {
           //Creamos los procesos
           A = new Llegadas();
           M = new Estacion();
           //Activamos de proceso inicial
           A.Activate();
           //Comenzamos la simulación
            Scheduler.startSimulation();
           while (M.procesados<1000)
              Hold(1000);
           //Finalizamos la simulación
           Scheduler.stopSimulation();
           //Finalizamos los procesos
           A.terminate();
           B.terminate();

           //Despertamos al hilo principal
           synchronized(cerrojo)
              { this.cerrojo.notify(); }
        }
        catch(SimulationException e) {}
        catch(RestartException e) {}
     }
}
```

Como puedes comprobar, el propio simulador es también un proceso, el cual se encarga de arrancar todos los procesos del modelo y de controlar el final de la simulación. Al principio sólo es necesario arrancar los procesos iniciales (que son los que contienen los sucesos iniciales del diagrama de sucesos); en el ejemplo éste es el proceso de llegadas `A`. Una vez iniciada la simulación, el proceso del simulador se dedica únicamente a vigilar periódicamente que la condición de finalización de la simulación se cumple (`M.procesados>=1000`).

Una vez terminada la simulación, el simulador debe notificarlo al planificador (`Scheduler.stopSimulation(`
y terminar todos los procesos que se pusieron en marcha. Finalmente, se retorna el control al hilo principal tal y como indicamos en la sección anterior.

2.5.4. Generación de números aleatorios

Todos los generadores de números aleatorios en JavaSim son subclases de la clase RandomStream, la cual representa la distribución uniforme $U(0,1)$. Esta clase implementa un generador congruencial multiplicativo con $a = 5^5$ y $p = 2^{26}$. En el tema siguiente veremos con detalle cómo se generan números aleatorios con estos parámetros.

El interfaz de la clase es el siguiente:

RandomStream
+ RandomStream()
+ RandomStream(long MGSeed, long LCGSeed)
+ double getNumber()
+ double Error()
# double Uniform();

Lo más destacable de esta clase es el método `getNumber()` que se utiliza para obtener uno a uno los valores de la secuencia de números aleatorios.

Para nuestros simuladores utilizaremos las siguientes clases (constructures) derivadas de RandomStream:

- UniformStream(double a, double b, int StreamSelect), que representa una variable con distribución $U(a, b)$
- ExponentialStream(double Media, int StreamSelect), que representa una distribución exponencial con media *Media*.
- ErlangStream(double Media, double SD, int StreamSelect), que representa una distribución *Erlang* con media *Media* y desviación estándar SD.
- HyperExponentialStream(double Media, int StreamSelect), ídem para una distribución exponencial.
- NormalStream(double Media, int StreamSelect), ídem para una distribución normal.

El último argumento de todos estos constructores representa el número de la secuencia aleatoria seleccionada, es decir, la semilla (o primer número de la serie) que utiliza en el generador para crear toda la serie. De este modo, con distintos valores del selector obtenemos secuencias diferentes de números aleatorios que siguen la distribución correspondiente. Este argumento será de especial utilidad en la fase de experimentación, donde necesitaremos realizar varias simulaciones con diferentes secuencias.

Cada vez que necesitemos una variable aleatoria en nuestro programa, crearemos un objeto con la clase que represente su distribución. Después generaremos valores utilizando el método `getNumber()` sobre el objeto creado.

Ejemplo. En el proceso Llegadas del ejemplo anterior podemos incluir un generador de números aleatorios que sigan una distribución exponencial para simular el tiempo entre llegadas de los clientes. Para ello, creamos un objeto de la clase ExponentialStream en el constructor de la clase Llegadas y la usaremos en el método `run()` para la obtención de valores para los tiempos entre llegadas.

```
public class Llegadas extends SimulationProcess
{
     public ExponentialStream Ta = null;

     public Llegadas(double T)
     {
        Ta = new ExponentialStream(T,1));
     }

     public void run()
     {
```

```
        for(;;){
           Cliente J = new Cliente();
           //Suceso ②
           M.Q.Encola(J);
           vacia=M.Q.Vacia()
           if (Vacia && !M.funciona)
              M.Activate();
           Hold( Ta.getNumber() );
     }
}
```

De nuevo hemos omitido por claridad las sentencias `try-catch` para capturar las excepciones producidas por `Hold`, `Activate` y `getNumber`.

2.5.5. Recogiendo resultados

JavaSim también proporciona un juego de clases para recoger resultados de la simulación y realizar estadísticas con ellos. La clase básica es Mean, cuyo interfaz es el siguiente:

Mean
+ Mean()
+ void setValue(double)
+ int numberOfSamples () const
+ double min()
+ double max()
+ double sum()
+ double mean()
+ void print()
+ void *reset()*

Para recoger las distintas observaciones del experimento utilizaremos el método `setValue()`. Una vez recogidos todos los datos, se pueden consultar algunos de sus estimadores estadísticos: número de muestras, mínimo, máximo, suma y media (`mean()`).

Existen otras clases derivadas de Mean que nos permiten extraer otros resultados de caracter estadístico, destacaremos las siguientes:

- Variance, que proporciona los métodos `variance()`, `stdDev()` y `confidence(double)` para determinar respectivamente la varianza, desviación típica e intervalo de confianza de los datos recogidos.
- TimeVariance es subclase de `Variance` y proporciona el método `timeAverage()` para calcular el promedio temporal de los resultados. Es decir, tiene en cuenta el tiempo en el que se ha recogido cada dato para promediar con el tiempo de simulación.

Para más información sobre estas clases y otras que no se han visto en estos apuentes, se recomienda consultar el manual de referencia de JavaSim.

Ejemplo. A modo de ejemplo vamos a modificar el proceso Estacion para tomar datos estadísticos de la cola de espera. Para ello creamos la variable `enCola` que representará la media temporal de los clientes encolados. Esta variable se actualiza en el método `run()`:

```
public class Estacion extends SimulationProcess
{
```

```
    public TimeVariance enCola=new TimeVariance();

    public void run()
    {
       for(;;)
       {
          funciona=true;
          while(!Q.Vacia())
          {
             //Suceso ③
             J=Q.Desencola();
             enCola.setValue(Q.Tamanyo());
             Hold(Ts);
             //Suceso ④
             procesados++;
          }
          funciona=false;
          Cancel();
       }
    }
}
```

Además, habría que realizar también las observaciones cuando se encola cada cliente en el proceso de llegadas:

```
public class Llegadas extends SimulationProcess
{
    public ExponentialStream Ta = null;

    public Llegadas(double T)
    {
       Ta = new ExponentialStream(T,1));
    }

    public void run()
    {
       for(;;){
          Cliente J = new Cliente();
          //Suceso ②
          M.Q.Encola(J);
          enCola.setValue(M.Q.Tamanyo());
          vacia=M.Q.Vacia()
          if (Vacia && !M.funciona)
             M.Activate();
          Hold(Ta.getNumber());
    }
}
```

También podemos estimar el tiempo medio de respuesta del sistema recogiendo los tiempos de entrada y salida de cada cliente. Para ello definimos una variable estática dentro de la clase que controla la simulación (Simulador), a la cual denominamos `TRespuesta`. Los

tiempos de entrada y de salida se toman en el contructor de la clase Cliente y en el método `finished()`, tal y como se muestra a continuación:

```
public class Cliente
{
     public Cliente()
     {
        Tllegada = Scheduler.CurrentTime();
     }

     public void finished()
     {
        Tsalida = Scheduler.CurrentTime();
        TRespuesta.setValue(Tsalida - Tllegada);
     }
     private double Tllegada;
     private double Tsalida;
}
```

Todos estos datos estadísticos se mostrarían en al finalizar el ciclo de simulación, en el cuerpo de la clase Simulador, tal y como se muestra a continuación:

```
public class Simulador extends SimulationProcess
{
     public void run()
     {
        try {
           //Creamos los procesos
           A = new Llegadas();
           M = new Estacion();
           //Activamos de proceso inicial
           A.Activate();
           //Comenzamos la simulación
           Scheduler.startSimulation();
           while (M.procesados<1000)
              Hold(1000);

           System.out.println(``Estadísticas:'');
           System.out.println(``Tiempo medio de respuesta:''+TRespuesta.mean());
           System.out.println(``Tamaño medio de la cola:''+M.EnCola.mean());

           //Finalizamos la simulación
           Scheduler.stopSimulation();
           //Finalizamos los procesos
           A.terminate();
           B.terminate();

           //Despertamos al hilo principal
           synchronized(cerrojo)
              { this.cerrojo.notify(); }
        }
        catch(SimulationException e) {}
```



```
        catch(RestartException e) {}
    }
    public static Mean TRespuesta= new Mean() ;
}
```

Es importante destacar que los resultados deben mostrarse antes de terminar la simulación con `stopSimulation()`, ya que de lo contrario los datos mostrados podrían no ser correctos.

La implementación completa de este ejemplo en JavaSim la puedes encontrar en la página Web de la asignatura.

2.6. SimPy

En las secciones anteriores nos hemos centrado en dos paquetes para crear simuladores basados en sucesos (SMPL) y procesos (JavaSim). Como se mencionó anteriormente, existen otros paquetes basados en otros lenguajes, como C++SIM en C++, Simula en Pascal, etc.

En esta sección vamos a describir las pricipales características del paquete de simulación SimPy para Python[3]. Como veremos, este paquete combina elementos de simulación de los lenguajes SMPL y JavaScript. Concretamente, de SMPL toma la gestión de recursos mediante las primitivas request y release, y de JavaSim las primitivas de gestión de procesos.

El bucle de simulación. Un simulador en SimPy debe incluir simpre las siguientes sentencias:

```
from SimPy.Simulation import *

initialize()
simulate(until=10000.0)
```

La primera línea importa la biblioteca de SimPy. La segunda línea inicializa todo el entorno de simulación (reloj, planificador de procesos, etc.) Finalmente, la sentencia `simulate` simula 10000,0 unidades de tiempo. En este caso, la simulación acaba inmediatamente ya que no hay sucesos que ejecutar.

Observa que la simulación siempre se controla con el tiempo de simulación máximo. Si se quiere controlar la simulación con cualquier otra condición, por ejemplo el número de clientes procesados, será necesario consultar periódicamente dicha condición y terminar la simulación cuando ésta se cumpla. Para ello, SimPy proporciona la sentencia stopSimulation() que detiene la simulación desde cualquier parte del programa. También existen otros mecanismos más sofisticados para controlar la simulación, que pueden consultarse en el manual de SimPy.

Procesos. Si queremos incluir sucesos, debemos crear los procesos correspondientes, al igual que en JavaSim. Para ello, crearemos subclases de la clase Process, En esta clase definiremos un método de ejecución que, al contrario que en JavaSim, puede tomar cualquier nombre y puede tener argumentos de entrada.

Por ejemplo, el siguiente proceso simplemente espera 100,0 unidades de tiempo, y cambia la variable de estado fin a True.

[3] http://simpy.sourceforge.net

```
from SimPy.Simulation import *

class Espera(Process):
        def __init__(self):
           Process.__init__(self)
        def run(self):
           yield hold, self, 100.0
           fin=True

initialize()
fin=False
p=Espera()
activate(p,p.run())
simulate(until=10000.0)
```

Para que un proceso funcione como tal, es necesario incluir siempre su constructor (`__init__`), y dentro de él, invocar al contructor de la superclase Process. En el constructor también podrían incluirse las variables de estado locales al proceso. Para activar por primera vez un proceso utilizaremos la sentencia activate, donde indicaremos cuál es el proceso concreto a activar, y el método que debe ejecutarse. Opcionalmente podría indicarse también el tiempo de simulación en el que debe activarse (parámetro at), el cual por defecto es cero.

Si por algún motivo un proceso p se duerme a sí mismo, utilizando yield passivate,self o es cancelado desde otro proceso, utilizando self.cancel(p), debe ser activado de nuevo con la sentencia reactivate(p). De este modo, activate solo debe ser utilizado la primera vez que activamos el proceso.

El siguiente programa es una traducción a SimPy de la implementación en JavaSim de una estación M/M/1. En él se puede ver el funcionamiento de los procesos en SimPy.

```
from SimPy.Simulation import *
from random import Random
class Cliente:
        def __init__(self,id):
           self.id=id
           self.Tinicio=0
        def finaliza(self):
           global TR
           TR+=(now()-self.Tinicio) ###acumula tiempo respuesta

class Estacion(Process):
        def __init__(self,Tservicio):
           Process.__init__(self)
           self.Q=[]
           self.funciona=False
           self.Tservicio=Tservicio
           self.Procesados=0

        def run(self):
           global TR
           while True:
              self.funciona=True
              while self.Q!=[]:
                 J = self.Q[0]
```



```
                del self.Q[0]
                yield hold,self,self.Tservicio
                self.Procesados+=1
                J.finaliza()
                del J
            self.funciona=False
            yield passivate,self

class Llegadas(Process):
        def __init__(self,id,Tmedio):
            Process.__init__(self)
            self.id=id
            self.cuenta=0
            self.Tmedio=Tmedio
            self.g=Random(333555)
        def run(self):
            global MAQ
            while True:
                self.cuenta+=1
                J = Cliente(self.cuenta)
                J.Tinicio=now()
                MAQ.Q.append(J)
                if not MAQ.funciona:
                    reactivate(MAQ)
                yield hold,self,self.g.expovariate(1.0/self.Tmedio)

initialize()
TR=0.0
LL=Llegadas("llegadas", 1.0)
MAQ=Estacion(1,1.0)
activate(LL,LL.run())
activate(MAQ,MAQ.run())
simulate(until=2000.0)

print "Procesados:", MAQ.Procesados
print "Tiempo medio de respuesta:", TR/MAQ.Procesados
```

Recursos. De forma similar a SMPL, SimPy también permite la gestión interna de los recursos. Para ello, proporciona la clase Resource. Las instancias de esta clase representan estaciones de servicio con uno o más servidores, donde la cola de espera puede gestionarse con una disciplina FIFO (por defecto), o mediante prioridades.

Como en SMPL, la gestión de los recursos se realiza mediante las setencias request y release, aunque ahora no será necesario indicar explicitamente el cliente que realiza esas operaciones.

Dado que entre ambas operaciones debe transcurrir el tiempo de servicio, será necesario definir un proceso para incluirlos. Normalmente, denominaremos a dicho proceso Cliente, ya que representará los sucesos que va realizando un cliente en el sistema simulado. A modo de ejemplo, implementaremos la estación M/M/1 utilizando los recursos de SimPy.

```
from SimPy.Simulation import *
from random import Random
```

```
class Cliente(Process):
        def __init__(self,id):
           Process.__init__()
           self.id=id
           self.Tinicio=0
        def run(self,MAQ,Tservicio):
           global TR,Procesados
           self.Tinicio=now()
           yield request,self,MAQ
           yield hold,self,Tservicio
           yield release,self,MAQ
           Procesados+=1
           TR+=(now()-self.Tinicio) ###acumula tiempo respuesta

class Llegadas(Process):
        def __init__(self,id,Tmedio):
           Process.__init__(self)
           self.id=id
           self.cuenta=0
           self.Tmedio=Tmedio
           self.g=Random(333555)
        def run(self):
           global MAQ
           while True:
              self.cuenta+=1
              J = Cliente(self.cuenta)
              activate(J,J.run())
              yield hold,self,self.g.expovariate(1.0/self.Tmedio)

initialize()
TR=0.0
Procesados=0
LL=Llegadas("llegadas", 1.0)
MAQ=Resource(capacity=1)
activate(LL,LL.run())
simulate(until=2000.0)

print "Procesados:", Procesados
print "Tiempo medio de respuesta:", TR/Procesados
```

Otras utilidades. SimPy utiliza la biblioteca random de Python para la generación de números y variables aleatorias. Explicaremos éstas con más detalle en el siguiente tema.

Por otro lado, al igual que JavaSim, SimPy proporciona herramientas para recoger resultados, denominados monitores. Éstos se crean instanciando la clase Monitor. Para registrar las observaciones se utiliza el método m.observe(dato), y para inicializar el monitor m.reset(). Durante o al final de la simulación, se pueden obtener estadísticas a partir de estos monitores, concretamente: la suma total de las observaciones (m.total()), el número de observaciones (m.count()), la media de las observaciones (m.mean()), la varianza (m.var()), la media temporal (m.timeAverage()) y un histograma de las observaciones (m.histogram(low,high,nbins)).

Para más información sobre éstas y otras funcionalidades de SimPy podéis consultar su manual en línea.

2.7. Bibliografía

A. M. Law, W. D. Kelton "Simulation Modeling & Analysis". Ed. McGraw-Hill (1984).

R. Jain "The Art of Computer Systems Performance Analysis". Ed. Wiley (1991).

MacDougall "Simulating Computer Systems: Techniques and Tools". MIT Press (1987).

G. Gordon "System Simulation". Editorial Prentice-Hall (1978).

M. C. Little y D. L. McCue "JavaSIM User's Guide", Libre Distribución.

G. L. Heileman "Estructuras de datos, algoritmos y Programación Orientada a Objetos". McGrawHill (1998).

J. García Jalón "Aprenda Java como si estuviera en primero". Escuela Superior de Ingenieros Industriales. Universidad de Navarra (2001).

T. Vignaux y K. Muller "SimPy Manual". http://simpy.sourceforge.net/SimPyDocs/Manual.html. Última visita: 21/10/2005.