Escuela Técnica Superior de Ingeniería Informática

Código Plan: G59 (Grado en Ingeniería de Computadores)
Código Asignatura: 590008

Prácticas de Sistemas en Tiempo Real

Curso 2015/16

Ignacio Parra Alonso
✉ ignacio.parra@uah.es
☎ 918856624
🖷 918856641
DE-338

Universidad de Alcalá

Capítulo 8

Monitorización de un sistema de control

8.1 Objetivos

El objetivo de esta práctica es realizar un programa concurrente de complejidad media donde se ilustren las distintas etapas de la construcción de una aplicación de tiempo real, aunque no se exigirá el análisis de los tiempos de respuesta para no complicar la práctica en exceso.

En relación con el lenguaje Ada, en esta práctica se utilizarán dos elementos sintácticos nuevos: los objetos protegidos y el manejo de excepciones.

A la hora de abordar la construcción de una aplicación compleja se seguirá una serie de pasos, que agruparemos en fases, y que nos permiten ir ganando profundidad en el conocimiento de la naturaleza del problema que se tiene que resolver, así como en el conocimiento de las ventajas e inconvenientes de las distintas soluciones. Las fases que se proponen desde la ingeniería del software son:

1. Análisis y especificación de requisitos.
2. Diseño.
3. Implementación.
4. Integración y pruebas.
5. Mantenimiento.

Empecemos pues por la captura y especificación de los requisitos del programa que se implementará en esta práctica.

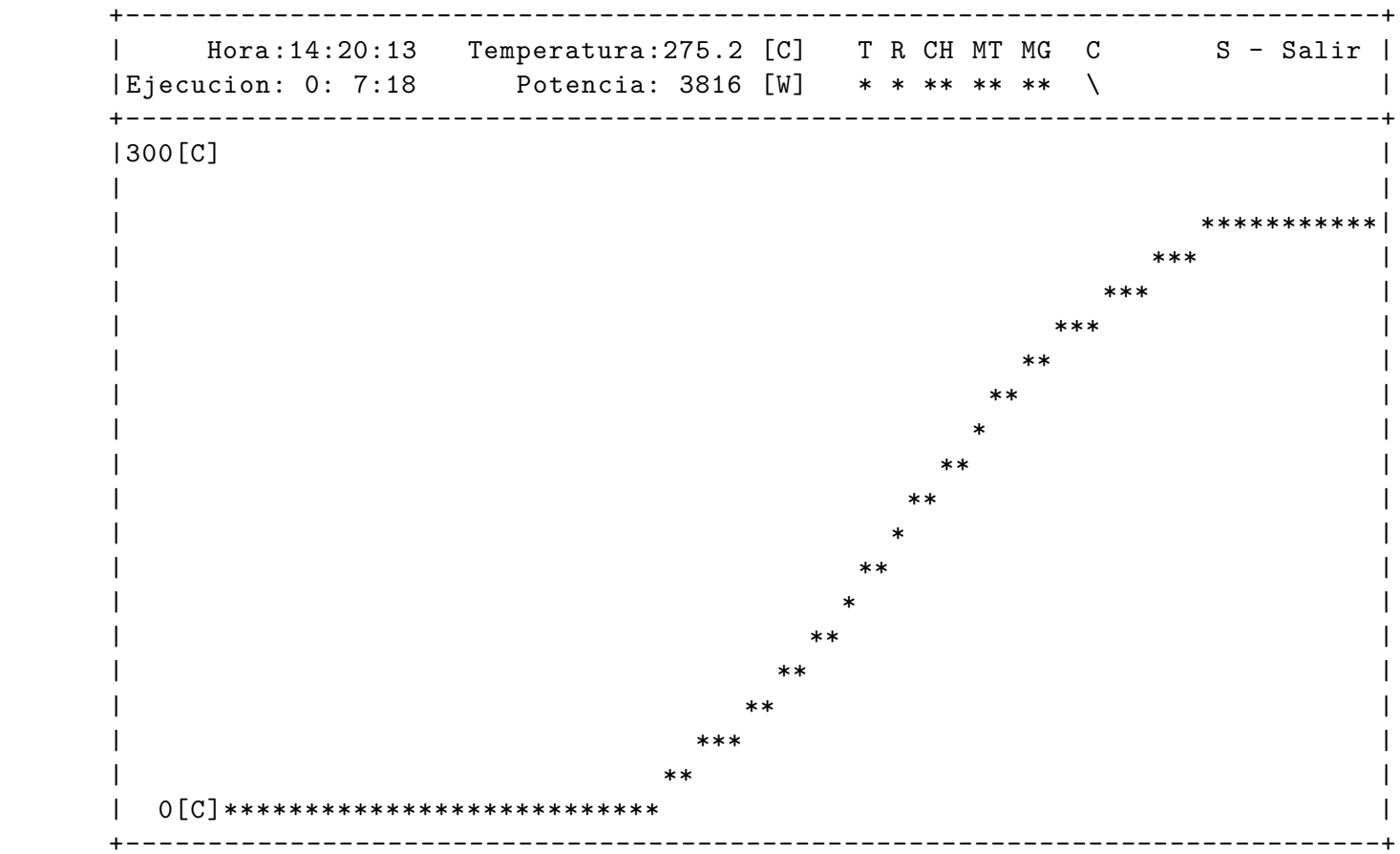
8.2 Análisis y especificación de requisitos

En el ejercicio profesional de la ingeniería del software un proyecto puede iniciarse a petición de un cliente que tiene unas necesidades que resolver y así lo manifiesta a nuestra organización. En nuestro caso el cliente es una empresa que fabrica hornos industriales y quiere que la regulación de la temperatura del horno se realice por programa.

La captura de requisitos empieza por una reunión con el cliente para conocer mejor la naturaleza de su problema y el tipo de solución que quiere y a la vez asesorarle sobre la viabilidad de esa solución y las mejores alternativas, en caso de que no sea realizable la solución pedida por él.

El objetivo de estas reuniones es poder responder a la pregunta «¿qué debe hacer el sistema?». Al finalizar las mismas se estará en condiciones de redactar un documento, formalizado a ser posible, denominado especificación de requisitos. Algunos organismos internacionales, como por ejemplo la Agencia Europea del Espacio [6], dictan normas sobre la forma y el contenido que deben tener los documentos que se generan en el proceso de creación de una aplicación. Aquí no seguiremos ninguna de esas normas ya que el objetivo de la práctica es dar una idea general sobre el proceso de desarrollo de una aplicación y no la exposición de un modelo de proceso detallado; además, las limitaciones de tiempo así lo aconsejan. Por lo tanto, las especificaciones del programa se expondrán informalmente y son las siguientes:

«*El programa de control del horno se encargará de controlar y monitorizar las dos variables más importantes que modelan su comportamiento: la potencia aplicada y la temperatura alcanzada. El requisito funcional más importante consiste en conseguir que el horno alcance una temperatura de funcionamiento permanente de 250 C y la mantenga hasta que se dé la orden de salir. Esta es la única orden que el programa debe reconocer durante su ejecución y se introducirá pulsando la tecla* `s`*. Además de controlar el horno, el programa debe monitorizar el estado de sus variables, así como su propio estado interno. El resultado de esta monitorización se presentará en una pantalla textual con el aspecto siguiente:*



En esta pantalla hay dos partes diferenciadas. Las dos líneas superiores muestran, de forma textual, la información siguiente:

`Hora:14:20:13`	*Hora actual del sistema.*
`Ejecucion: 0: 7:18`	*Tiempo de ejecución del programa, medido desde el instante de arranque.*
`Temperatura:275.2 [C]`	*Temperatura actual del horno.*
`Potencia: 3816 [W]`	*Potencia que se le está suministrando al horno.*
`T R CH MT MG C`	
`* * ** ** ** \`	*Información referente al estado interno del programa. Se explicará en al apartado de diseño.*

En la parte inferior de la pantalla podemos ver una representación gráfica de la evolución de la temperatura durante los últimos 12 minutos.

El periodo de refresco de cada información es el siguiente: hora y tiempo de ejecución (1 s), temperatura y potencia (1 s), estado del sistema (0,2 s) y evolución de la temperatura (10 s).»

Hasta aquí hemos descrito lo que debe hacer el programa, veamos ahora cómo puede hacerse.

8.3 Diseño

Nuestra experiencia nos indica que la funcionalidad descrita en los requisitos se ajusta a un sistema de tiempo real. Se han propuesto diferentes métodos para diseñar este tipo de sistemas, uno de los más destacados es *HRT-HOOD* (*Hard Real-Time Hieralchical Object Oriented Design*) [3]. En esta práctica no se realizará un descripción de esta metodología porque excedería los límites de tiempo y espacio de la misma, sin embargo, utilizando una simbología simplificada combinación de *HRT-HOOD* y de la metodología *Booch* [2], describiremos la estructura de la aplicación en dos niveles: el diagrama de contexto y el diagrama de componentes.

En el *diagrama de contexto* mostramos la interacción de la aplicación con su entorno. Aquí se deben considerar los dispositivos físicos y programas externos con los que interactúa la aplicación. Estos elementos externos nos vienen impuestos por la naturaleza física del proceso que se quiere controlar y por las interfaces con el usuario que se deban implementar.

En nuestro problema, el sistema de control debe interactuar con el horno a través de dos registros, uno para leer la temperatura y otro para escribir la potencia. Por otro lado, la interfaz con el usuario se implementa a través de un teclado para leer órdenes y la pantalla para escribir los resultados de la monitorización. En la figura 8.1 se muestran estas interacciones.

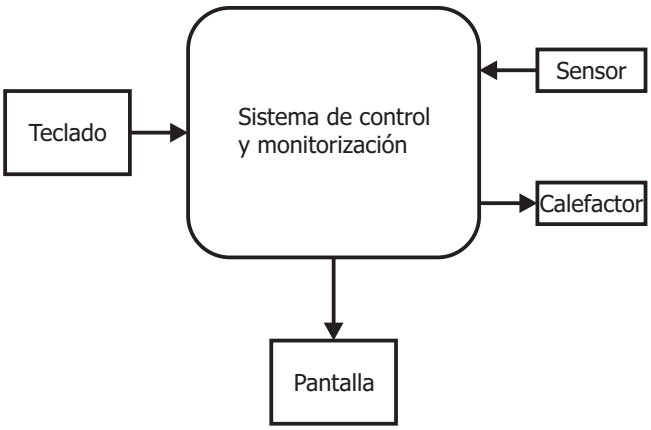


Fig. 8.1: Diagrama de contexto de la aplicación.

El diagrama de contexto muestra los dispositivos con los que el sistema intercambia información pero no aporta información que ayude en la implementación. Necesitamos un diagrama de más detalle que refleje la estructura interna de la aplicación y para ello consideraremos tres clases de elementos constructivos: actividades cíclicas, objetos protegidos y elementos de biblioteca.

Las *actividades cíclicas* representan tareas periódicas que se ejecutan concurrentemente en la aplicación y sobre ellas se pueden realizar dos operaciones: comprobar si sigue viva y ordenarle que termine. La actividad cíclica se define mediante un periodo con el que tendrá que ejecutar una secuencia de instrucciones a las que llamamos *código cíclico*. En la figura 8.2 podemos ver el símbolo que se utiliza para representar una actividad cíclica.

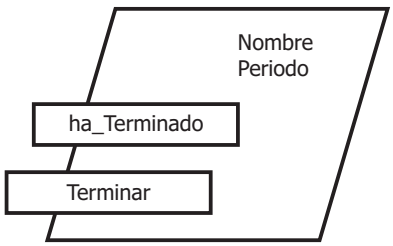


Fig. 8.2: Actividad cíclica.

Los *objetos protegidos* se utilizan para almacenar el estado del sistema, exportan operaciones de consulta del estado y modificación del mismo. Estas operaciones son invocadas por las actividades cíclicas para actuar según el estado en el que se encuentre el sistema, por ello las consultas y modificaciones se deben realizar en exclusión mutua, garantizando así que el sistema no queda en un estado inconsistente. Los mecanismos de acceso en exclusión mutua están encapsulados en el propio objeto protegido. En la figura 8.3 podemos ver el símbolo que se utiliza para representar un objeto protegido que exporta dos operaciones.

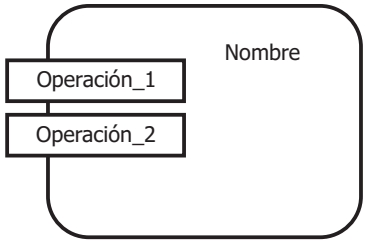


Fig. 8.3: Objeto protegido.

Los *elementos de biblioteca* son entidades pasivas que almacenan colecciones de código reentrante que es utilizado por el resto del sistema como apoyo para la implementación de algoritmos. Son las típicas bibliotecas de funciones o procedimientos. Un elemento de biblioteca exporta tipos y operaciones. En la figura 8.4 podemos ver el símbolo que se utiliza para representar un elemento de biblioteca que exporta un tipo y dos operaciones.

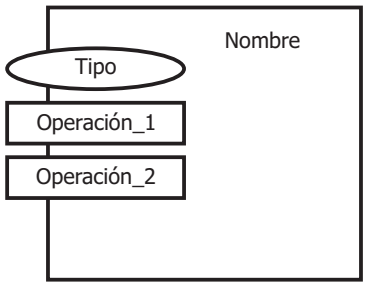


Fig. 8.4: Elemento de biblioteca.

Los elementos de biblioteca se pueden utilizar también para representar *código reutilizable*. En este caso sirven de plantillas que utilizaremos para construir elementos de biblioteca concretos. En la figura 8.5 podemos ver el símbolo que se utiliza para representar un elemento de biblioteca genérico que exporta un tipo, dos operaciones y está parametrizado con un tipo y una constante.

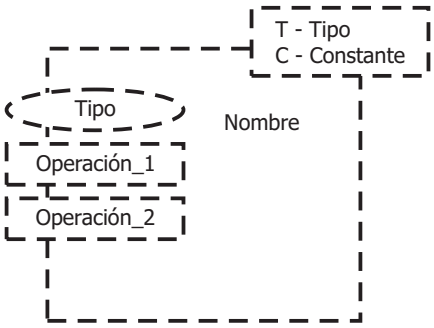


Fig. 8.5: Elemento de biblioteca genérico.

Con estos elementos estamos en condiciones de realizar el *diagrama de componentes* que describe la estructura de nuestra aplicación.

En la figura 8.6 podemos ver los componentes de la aplicación y las relaciones que hay entre ellos. Hay que explicar que las flechas de trazo continuo que unen los distintos componentes indican relaciones de visibilidad y uso pero no están relacionadas con las direcciones del flujo de información. Así por ejemplo, la flecha dirigida desde el objeto protegido `Estado` hacia el elemento de biblioteca `Calefactor` indica que el primer componente utiliza alguno de los elementos definidos en el segundo y por lo tanto `Calefactor` debe ser visible para `Estado`. Por otro lado, las líneas de trazo discontinuo indican relaciones con los elementos externos a la aplicación y que ya aparecen en el diagrama de contexto.

Para completar el diseño de la aplicación hay que especificar la semántica de cada componente indicando qué hace cada uno de ellos. Esta especificación, junto con los diagramas anteriores constituyen los documentos básicos que genera el diseñador de la aplicación y los debe generar teniendo en cuenta la especificación de requisitos suministrada por el analista. Se suele recurrir a los lenguajes de descripción de programas (*PDL–program description language*) para especificar el comportamiento de cada uno de los componentes. El Instituto de Ingeniería Eléctrica y Electrónica de Estados Unidos (*IEEE*) ha definido un *PDL* basado en el lenguaje Ada [8] al que llama *Ada-PDL* y que usaremos en esta práctica en una versión adaptada al español con el nombre *LDP-español*. Nuestro LDP es un seudocódigo con las siguientes estructuras de control [5]:

- Selección

```
1 SI <seudocódigo de la condición> ENTONCES
2     <seudocódigo>
3 SI−NO
4     <seudocódigo>
5 FIN−SI
```

- Selección por casos

```
1 CASO <especificación del elemento selector>
2     SI−ES <caso 1> HACER <seudocódigo>
3     SI−ES <caso 2> HACER <seudocódigo>
4     ...
5     OTROS <seudocódigo>
6 FIN−CASO
```

- Iteración con pre-condición

```
1 MIENTRAS <seudocódigo de la condición> HACER
2     <seudocódigo>
3 FIN−MIENTRAS
```

- Iteración con pos-condición

```
1 REPETIR
2     <seudocódigo>
3 HASTA <seudocódigo de la condición>
```

- Iteración con número de iteraciones conocido

```
1 PARA−CADA <especificación del elemento índice> HACER
2     <seudocódigo>
3 FIN−PARA−CADA
```

Como puede verse, este seudocódigo se corresponde claramente con las estructuras de control típicas que tienen todos los lenguajes de programación estructurados. Cabe preguntarse entonces por qué usar este lenguaje intermedio y no un lenguaje de programación para especificar cada componente. El motivo principal es que el diseño no debe entrar en detalles de implementación, fase que abordaremos más tarde,

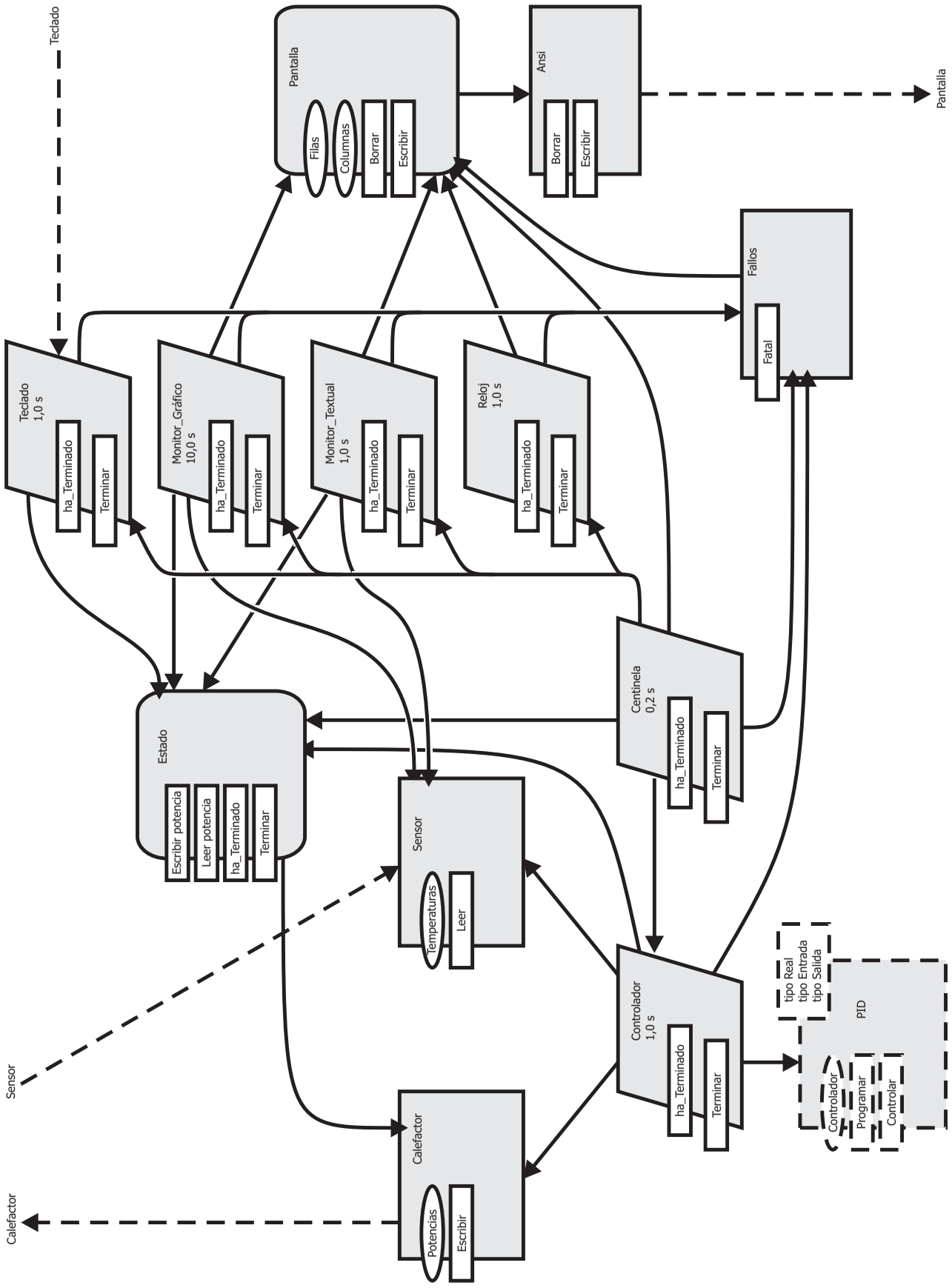


Fig. 8.6: Diagrama de componentes de la aplicación.

y debe ser independiente del lenguaje. Por otro lado en el diseño se describen las unidades de una forma lo suficientemente precisa como para que el programador pueda implementar el componente, pero sin descender al detalle que requieren los compiladores para poder generar la aplicación. Teniendo presentes estas consideraciones pasaremos a describir el comportamiento de los componentes del sistema, empezando por las actividades cíclicas:

```
ACTIVIDAD Reloj
   REPETIR con periodo 1,0 s
      leer la hora actual
      calcular el tiempo de ejecución
      formatear los datos para presentarlos
      presentar en pantalla los datos formateados
   HASTA siempre
FIN Reloj
```

```
ACTIVIDAD Controlador
   calcular los parámetros ki, kp y kd del controlador PID
   programar el controlador PID
   REPETIR con periodo 1,0 s
      leer la temperatura del horno
      calcular la potencia aplicable
      escribir la potencia en el controlador
      escribir la potencia en el estado
   HASTA siempre
FIN Controlador
```

```
ACTIVIDAD Teclado
   REPETIR con periodo 1,0 s
      leer una tecla de forma inmediata
      SI hay una tecla diponible ENTONCES
         CASO tecla
            SI-ES 's' o 'S' HACER
               cambiar el estado del sistema para terminar su ejecución
            OTROS no hacer nada
         FIN-CASO
      FIN-SI
   HASTA siempre
FIN Teclado
```

```
ACTIVIDAD Monitor_Textual
   REPETIR con periodo 1,0 s
      leer la temperatura del sensor
      leer la potencia del estado
      formatear los datos para presentarlos
      presentar en pantalla los datos formateados
   HASTA siempre
FIN Monitor_Textual
```

```
ACTIVIDAD Monitor_Gráfico
   REPETIR con periodo 10,0 s
      leer la temperatura del sensor
      escribir la temperatura en la historia de temperaturas de los últimos 12 minutos
      presentar en pantalla la historia de temperaturas de los últimos 12 minutos
   HASTA siempre
```

```
7  FIN Monitor_Gráfico
```

```
1  ACTIVIDAD Centinela
2     REPETIR con periodo 0,2 s
3        SI el estado ha terminado ENTONCES
4           terminar las actividades cíclicas: Teclado, Reloj, Controlador, Monitor_Textual y »
                  » Monitor_Gráfico
5           borrar la pantalla
6           terminar la actividad cíclica Centinela
7        SI–NO
8           SI ha terminado alguna de las actividades cíclicas: Teclado, Reloj, Controlador, »
                  » Monitor_Textual o Monitor_Gráfico
9           ENTONCES
10             poner parpadeante su indicador correspondiente
11          FIN–SI
12          cambiar el indicador de la actividad cíclica Centinela para mostrar que sigue viva
13       FIN–SI
14    HASTA siempre
15 FIN Centinela
```

Como se ha indicado en el apartado de análisis, el estado de las tareas se indica con las siguientes dos líneas:

```
T R CH MT MG  C
* * ** ** **  \
```

La primera línea contiene el nombre de cada tarea: `T`–`Teclado`, `R`–`Reloj`, `CH`–`Controlador`, `MT`–`Monitor_Textual`, `MG`–`Monitor_Gráfico` y `C`–`Centinela`. El asterisco que aparece debajo de cada tarea indica que la tarea sigue viva; en caso de que muera el asterisco debe ponerse parpadeante. La barra que aparece debajo de la tarea centinela es en realidad una barra que gira siguiendo la secuencia `|/-\|/-\` (si presentamos esta secuencia de caracteres en la misma posición de pantalla y con una cadencia de 0,2 s, se producirá la sensación de que la barra gira).

Como puede comprobarse, la especificación del comportamiento de cada componente se hace a muy alto nivel y la información que contiene debe ser considerada como indicación para el programador de cada componente. Lo importante es que en la implementación se respete la interfaz y la funcionalidad de cada componente, así se garantiza que la integración de todos ellos se realizará con un mínimo de errores.

Las operaciones que exporta cada actividad cíclica se deben traducir en órdenes para esa actividad o consultas del estado de la misma. En nuestro ejemplo todas las actividades cíclicas exportan las mismas operaciones: `Terminar`, para terminar la ejecución de esa actividad y `ha_Terminado` para consultar si la actividad sigue viva o no. Si utilizamos el LDP-español podemos describir estas operaciones de la forma siguiente:

```
1  ACTIVIDAD Reloj
2     OPERACIÓN Terminar
3        Terminar la ejecución de la actividad cíclica
4     FIN Terminar
5     OPERACIÓN ha_Terminado
6        SI la actividad cíclica ha terminado ENTONCES
7           devolver verdadero
8        SI–NO
9           devolver falso
10       FIN–SI
11    FIN ha_Terminado
12 FIN Reloj
```

El resto de actividades exportan las mismas operaciones y no las repetiremos aquí.

Los objetos protegidos se utilizan para guardar el estado del sistema y controlar el acceso a recursos compartidos, en la especificación de su comportamiento se deben definir los datos que protegen y las operaciones permitidas para manipular esos datos. En nuestra aplicación tenemos los objetos `Estado` y `Pantalla`; el primero almacena el estado de la aplicación mientras que el segundo controla el acceso al dispositivo físico de presentación. Ambos componentes se deben definir como objetos protegidos ya que son utilizados por varias actividades cíclicas. La definición de cada objeto puede ser la siguiente:

```
1  OBJETO−PROTEGIDO Estado
2     DATO−PROTEGIDO Potencia de tipo Potencias
3        −− Es la potencia suministrada al Calefactor
4     DATO−PROTEGIDO Terminado de tipo Boolean
5        −− Empieza valiendo Falso y cambia a Verdad cuando se pulsa la tecla de salir
6
7     OPERACIÓN Escribir (Potencia: Potencias)
8        la_Potencia := Potencias
9     FIN Escribir
10    OPERACIÓN Leer
11       devuelve la_Potencia
12    FIN Leer
13    OPERACIÓN ha_Terminado
14       devuelve Terminado
15    FIN ha_terminado
16    OPERACIÓN Terminar
17       Terminado := Verdad
18    FIN Terminar
19 FIN Estado
```

```
1  OBJETO−PROTEGIDO Pantalla
2     TIPO Filas −− número natural en el rango 0..24
3     TIPO Columnas −− número natural en el rango 0..78
4
5     −− En la implementación de estas operaciones se usarán las operaciones
6     −− exportadas por el componente "Ansi"
7     OPERACIÓN Borrar
8        Borrar la pantalla
9     FIN Borrar
10    OPERACIÓN Escribir (Cadena: Cadena de caracteres; Fila: Filas; Columna: Columnas)
11       recortar la Cadena para que quepa en una línea
12       poner el cursor de la pantalla en la Fila y Columna indicadas
13       escribir la Cadena en la pantalla
14    FIN Escribir
15 FIN Estado
```

Por último abordaremos la descripción de los elementos de biblioteca. Estos componentes se comportan como colecciones de operaciones y exportan tipos, constantes y operaciones. Puesto que estos componentes son reentrantes, no tienen ningún estado interno asociado y pueden ser usuados simultáneamente por varias actividades concurrentes sin que sea necesario habilitar mecanismos de sincronización, por ello los elementos de biblioteca no pueden exportar variables. Veamos ahora los elementos de biblioteca que hay en nuestra aplicación:

```
1  BIBLIOTECA Sensor
2     TIPO Temperaturas −− número real en el intervalo [−25,0; 500,0]
3
```

```
    OPERACIÓN Leer (la_Temperatura: salida Temperaruras)
        la_Temperatura := leer la temperatura del sensor del horno
    FIN Leer
FIN Sensor
```

```
BIBLIOTECA Calefactor
    TIPO Potencias -- número real en el intervalo [0,0; 10_000,0]

    OPERACIÓN Escribir (la_Potencia: Potencias)
        escribr la_Potencia en el calefactor del horno
    FIN Escribir
FIN Calefactor
```

```
BIBLIOTECA Ansi
    OPERACIÓN Borrar_Pantalla
        escribir las secuencias de escape del controlador ANSI para borrar la pantalla
    FIN Borrar_Pantalla
    OPERACIÓN Poner_Cursor (Fila, Columna: Natural)
        escribir las secuencias de escape del controlador ANSI para poner el cursor en la Fila y »
            » Columna indicadas
    FIN Poner_Cursor
FIN Ansi
```

Para ver la forma de utilizar las secuencias de escape *ANSI* se puede consultar [4].

```
BIBLIOTECA Fallos
    OPERACIÓN Fatal (F: Fallo; Mensaje: Cadena de caracteres)
        escribir un texto parpadeante en el centro de la pantalla con la información referente al fallo »
            » producido y el Mensaje indicado
    FIN Fatal
FIN Fallos
```

```
BIBLIOTECA PID
Parámetros genéricos
    TIPO Real -- número real con cualquier precisión y cualquier rango
    TIPO Entrada -- número real con cualquier precisión y cualquier rango
    TIPO Salida -- número real con cualquier precisión y cualquier rango

Tipos concretos
    TIPO Controlador -- Encapsula los parámetros que definen un controlador PID y su estado

    OPERACIÓN Programar (el_Controlador: entrada/salida Controlador; Kp, Ki, Kd: Real)
        cambiar los valores de el_Controlador de acuerdo con los parámetros Kp, Ki y Kd
        poner el estado de el_Controlador en condiciones de reposo inicial
    FIN Programar
    OPERACIÓN Controlar (con_el_Controlador: entrada/salida Controlador; R: Entrada; C: »
            » Entrada; U: salida Salida)
        U := resultado de aplicar un algoritmo de control proporcional-integral-derivativo teniendo en »
            » cuenta los parámetros de "con_el_Controlador", su estado y los parámetros de »
            » entrada R (referencia) y C (valor de la salida)
        actualizar el estado de "con_el_Controlador"
    FIN Controlar
FIN PID
```

Ya hemos visto que en la especificación de los componentes se describen la funcionalidad y los datos que se intercambian (datos externos). En el *diccionario de datos* se recogen los nombres de todos los tipos de datos externos y sus definiciones. Esto le facilitará al programador la tarea de implementación al acortar los tiempos de búsqueda de estas definiciones.

```
1 Diccionario
2     Filas −− número natural en el rango 0..24
3     Columnas −− número natural en el rango 0..78
4     Temperaturas −− número real en el intervalo [−25,0; 500,0]
5     Potencias −− número real en el intervalo [0,0; 10_000,0]
6     Controlador −− Tipo privado. Su definición depende del algoritmo empleado para implementar el »
          » control PID
7 FIN Diccionario
```

En este punto finalizamos el diseño de la aplicación y como podemos ver los objetivos conseguidos han sido describir la arquitectura y comportamiento del sistema a través del concepto de componente y las relaciones entre componentes. Como podemos ver la descripción no entra en detalles de implementación, se dice que es de alto nivel e independiente del lenguaje de programación. El diseño tiene que despejarle al programador todas las dudas sobre lo que debe hacer; la respuesta a ¿cómo hacerlo? la debe suministrar el propio programador, él conocerá cómo utilizar el lenguaje de programación elegido para expresar el diseño realizado en esta fase. Es importante que el diseño no contenga ambigüedades y pueda traducirse siguiendo reglas precisas, lo ideal sería que el diseño pudiese traducirse mecánicamente y que hubiese herramientas automáticas de generación de código. Este objetivo está siendo conseguido en gran medida por las metodologías de desarrollo modernas como pueden ser *HRT-HOOD* o *UML* (*unified modeling language*) [7] y herramientas como Rational-Rose que pueden generar código Ada, C++, Java, etc. a partir de esquemas de diseño. En el apartado siguiente vamos a exponer algunas reglas para traducir sistemáticamente el diseño anterior a lenguaje Ada.

8.4 Implementación

Es labor del analista o del diseñador seleccionar el lenguaje o lenguajes que se emplearán en la implementación del sistema. Muchos organismos oficiales imponen el lenguaje de programación que debe emplearse en los proyectos que se desarrollen para ellos; esto se debe, fundamentalmente, a criterios de uniformidad. En el ejemplo que se está desarrollando se ha elegido Ada como lenguaje de implementación y no es necesario a estas alturas justificar esta elección. El diseñador debe decidir cuántos módulos tendrá la aplicación, cómo se llamarán y en qué lenguaje se debe escribir cada uno de ellos. Dado que el lenguaje elegido es Ada y este lenguaje tiene recursos sintácticos para expresar actividades cíclicas, objetos protegidos y elementos de biblioteca, todos los módulos se escribirán en este lenguaje. En cuanto al número de módulos la decisión es simple, tiene que haber dos módulos por componente y estos módulos tendrán la forma de un paquete, uno de ellos almacenará la especificación y el otro el cuerpo. Empezaremos viendo la forma que tienen los módulos donde se implementan las actividades cíclicas:

```
package Nombre_del_Componente is
   Ts: constant := ...; -- Período de la tarea
   function ha_Terminado return Boolean;
   procedure Terminar;
end Nombre_del_Componente;
```

Como vemos, en la especificación de cada actividad cíclica aparecen tres elementos: el periodo de la actividad, la operación para preguntar si la actividad sigue viva y la operación para ordenar que la actividad termine. Toda la información necesaria para escribir esta especificación la hemos extraído del esquema de diseño y del seudocódigo LDP-español.

El cuerpo de todas las actividades cíclicas se ajusta a la plantilla siguiente:

```
with Ada.Real_Time,
     Fallos,
```

```
     ... -- otros componentes, para ello consultar el diagrama
         -- de componentes
use Ada.Real_Time;

package body Nombre_del_Componente is
   task la_Tarea;
   procedure Código_Periódico;
   task body la_Tarea is
      Siguiente_Instante: Time := Clock;
      Periodo: constant Time_Span:=To_Time_Span(Ts);
   begin
      loop
         delay until Siguiente_Instante;
         Código_Periódico;
         Siguiente_Instante := Siguiente_Instante + Periodo;
      end loop;
   exception
      when Evento: others => Fallos.Fatal (Evento,
                                           "Nombre_del_Componente.adb");
   end la_Tarea;

   -- Variables y constantes que deben conservar sus valores entre las
   -- sucesivas llamadas al "Código_Periódico"
   ...
   procedure Código_Periódico is
   ...
   end Código_Periódico;

   function ha_Terminado return Boolean is
   begin
      return la_Tarea'Terminated;
   end ha_Terminado;
   procedure Terminar is
   begin
      if not la_Tarea'Terminated then
         abort la_Tarea;
      end if;
   end Terminar;
end Nombre_del_Componente;
```

Los objetos protegidos se deben implementar como paquetes que encapsulan un objeto protegido del lenguaje Ada. Así, por ejemplo, un objeto protegido que exporta las operaciones `Leer` y `Escribir` tiene la siguiente especificación:

```
package Nombre_del_Componente is
   procedure Escribir (el_Valor: Tipo_de_los_Datos);
   procedure Leer (el_Valor: out Tipo_de_los_Datos);
end Nombre_del_Componente;
```

El cuerpo siguiente se corresponde con la especificación anterior:

```
package body Nombre_del_Componente is
   protected el_Objeto_Protegido is
      procedure Escribir (el_Valor: Tipo_de_los_Datos);
      procedure Leer (el_Valor: out Tipo_de_los_Datos);
```

```
      private
         Datos: Tipo_de_los_Datos := Valor_Inicial;
      end el_Objeto_Protegido;

      protected body el_Objeto_Protegido is
         procedure Escribir (el_Valor: Tipo_de_los_Datos) is
         begin
            los_Datos := el_Valor;
         end Escribir
         procedure Leer (el_Valor: out Tipo_de_los_Datos) is
         begin
            el_Valor := los_Datos;
         end Leer;
      end el_Objeto_Protegido;

      procedure Escribir (el_Valor: Tipo_de_los_Datos) is
      begin
         el_Objeto_Protegido.Escribir (el_Valor);
      end Escribir;
      procedure Leer (el_Valor: out Tipo_de_los_Datos) is
      begin
         el_Objeto_Protegido.Leer (el_Valor);
      end Leer;
   end Nombre_del_Componente;
```

Los elementos de biblioteca también se implementarán como paquetes, poniendo en la especificación la definición de los tipos externos y la interfaz de las operaciones; y en el cuerpo el código de las operaciones.

Los módulos que restan por escribir en nuestra aplicación son:

`Ansi.ads`, `Ansi.adb`,
`Calefactor.ads`, `Calefactor.adb`,
`Centinela.ads`, `Centinela.adb`,
`Controlador.ads`, `Controlador.adb`,
`Estado.ads`, `Estado.adb`,
`Fallos.ads`, `Fallos.adb`,
`Monitor_Gráfico.ads`, `Monitor_Gráfico.adb`
`Monitor_Textual.ads`, `Monitor_Textual.adb`,
`Pantalla.ads`, `Pantalla.adb`,
`PID.ads`, `PID.adb`,
`Reloj.ads`, `Reloj.adb`,
`Sensor.ads`, `Sensor.adb`,
`Teclado.ads` y `Teclado.adb`,

Para facilitar la realización de la práctica, a continuación podemos ver la implementación de los paquetes `Ansi` y `Fallos`:

Fichero 8.1: Especificación del paquete `Ansi` (`Ansi.ads`)

```
1  -- Paquete para escribir en pantalla usando las secuencias de escape del
2  -- controlador ANSI.SYS.
3  -- Para poder usar este paquete con efectividad es necesario que en el
4  -- fichero CONFIG.SYS figure una línea como la siguiente:
5  -- DEVICE = C:\DOS\ANSI.SYS
6  --
7  package Ansi is
8     procedure Borrar_Pantalla;
```

```
9     procedure Poner_Cursor (Fila, Columna: Natural);
10 end Ansi;
```

Fichero 8.2: Cuerpo del paquete `Ansi` (`Ansi.adb`)

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  package body Ansi is
3     procedure Borrar_Pantalla is
4     begin
5        Put (Ascii.Esc&"[2J");
6     end Borrar_Pantalla;
7
8     package Natural_Es is new Integer_Io(Natural);
9     use Natural_Es;
10    procedure Poner_Cursor(Fila, Columna: Natural) is
11    begin
12       Put (Ascii.Esc&"["); Put (Fila+1, Width => 0); Put (';');
13       Put (Columna+1, Width => 0); Put ('f');
14       -- Le sumamos 1 a la "Fila" y a la "Columna" porque el
15       -- origen de coordenadas es (1, 1) para el controlador "Ansi".
16       -- Los módulos que utilicen este paquete deben considerar el
17       -- origen en (0, 0).
18    end Poner_Cursor;
19 end Ansi;
```

Fichero 8.3: Especificación del paquete `Fallos` (`Fallos.ads`)

```
1  with Ada.Exceptions; use Ada.Exceptions;
2  package Fallos is
3     procedure Fatal (E: Exception_Occurrence; Mensaje : String);
4  end Fallos;
```

Fichero 8.4: Cuerpo del paquete `Fallos` (`Fallos.adb`)

```
with Pantalla;
package body Fallos is
   procedure Fatal (E: Exception_Occurrence; Mensaje: String) is
      task Aviso_Parpadeante;
      task body Aviso_Parpadeante is
      begin
         loop
            Pantalla.Escribir ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", 12, 8);
            Pantalla.Escribir ("␣+---------------------------------------------------------------+␣", 13, 8);
            Pantalla.Escribir ("␣|␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣", 14, 8);
            Pantalla.Escribir ("␣|␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣", 15, 8);
            Pantalla.Escribir ("␣|␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|␣", 16, 8);
            Pantalla.Escribir ("␣+---------------------------------------------------------------+␣", 17, 8);
            Pantalla.Escribir ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", 18, 8);
            Pantalla.Escribir ("Excepcion␣capturada␣por␣" & Mensaje, 14, 10);
            Pantalla.Escribir ("Se␣ha␣elevado␣" & Exception_Name (E), 15, 10);
            Pantalla.Escribir ("En␣el␣módulo␣" & Exception_Message (E), 16, 10);
            delay 1.0;
            Pantalla.Escribir ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", 12, 8);
            Pantalla.Escribir ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", 13, 8);
            Pantalla.Escribir ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", 14, 8);
            Pantalla.Escribir ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", 15, 8);
            Pantalla.Escribir ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", 16, 8);
            Pantalla.Escribir ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", 17, 8);
            Pantalla.Escribir ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣", 18, 8);
            delay 0.1;
         end loop;
      end Aviso_Parpadeante;
   begin
      null;
   end Fatal;
end Fallos;
```

8.5 Integración y pruebas

Una gran parte de los esfuerzos realizados en el desarrollo de la aplicación iban encaminados a dividirla en módulos sencillos de comprender y codificar, pero esos módulos por sí solos no sirven para nada, ahora hay que unirlos para formar un programa ejecutable que resuelva nuestro problema y al que hemos llamado aplicación o sistema. La fase de integración se ocupa de esta tarea y en sistemas grandes compuestos por cientos de módulos puede ser una tarea realmente compleja y problemática. Por lo tanto interesa también mecanizar al máximo esta parte del desarrollo. Una de las herramientas más utilizadas en el entorno Unix y que ha sido adaptada para otros sistemas operativos es el programa `make` [9, págs. 526–531], del cual existe una versión modificada para el compilador Gnat conocido como `gnatmake`.

Para integrar los módulos de nuestra aplicación hay que escribir un procedimiento que haga visibles las actividades cíclicas para ponerlas en marcha. El procedimiento `Principal` que se muestra más adelante puede ser un ejemplo. Como ya sabemos, basta con ejecutar la orden:

```
$ gnatmake principal
```

para generar al programa de prueba `Principal` a partir de los módulos que componenen la aplicación.

Programa 8.5: Procedimiento que pone en marcha la aplicación (`Principal.adb`)

```
1  with Pantalla,
2       Centinela,
3       Monitor_Textual;
4
5  procedure Principal is
6     Marco: array (Pantalla.Filas'Range) of String (1..79) := (
7        "+-----------------------------------------------------------------------------+",
8        "|      Hora:HH:MM:SS   Temperatura:eee.d [C]   T R CH MT MG  C       S - Salir |",
9        "|Ejecucion:HH:MM:SS       Potencia:eeeee [W]   - - --  --  -                   |",
10       "+-----------------------------------------------------------------------------+",
11       "|300[C]                                                                       |",
12       "|                                                                             |",
13       "|                                                                             |",
14       "|                                                                             |",
15       "|                                                                             |",
16       "|                                                                             |",
17       "|                                                                             |",
18       "|                                                                             |",
19       "|                                                                             |",
20       "|                                                                             |",
21       "|                                                                             |",
22       "|                                                                             |",
23       "|                                                                             |",
24       "|                                                                             |",
25       "|                                                                             |",
26       "|                                                                             |",
27       "|                                                                             |",
28       "|                                                                             |",
29       "|                                                                             |",
30       "|  0[C]                                                                       |",
31       "+-----------------------------------------------------------------------------+");
32 begin
33    -- Dibujar el "Marco"
34    Pantalla.Borrar;
35    for I in Marco'Range loop
36       Pantalla.Escribir (Marco(I), I, Pantalla.Columnas'First);
```

```
37    end loop;
38
39    -- En la versión definitiva se suprimirán las dos líneas de código siguientes
40    delay 60.0*2.0; -- Espera 2 minutos
41    Monitor_Textual.Terminar; -- Comprueba que el "Centinela" detecta la terminación de
42                              -- una tarea.
43 end Principal;
```