UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Comunicación entre aplicaciones desarrolladas con MetaCard: paso de mensajes

Apellidos, nombre	Agustí i Melchor, Manuel (magusti@disca.upv.es)
Departamento	Departamento de Informática de Sistemas y Computadores
Centro	Escola Tècnica Superior d'Enginyeria Informàtica Universitat Politècnica de València

1 Resumen de las ideas clave

Este artículo es caso de estudio enfocado a la experimentación sobre vías de comunicación, en tiempo de ejecución, entre aplicaciones desarrolladas con *MetaCard* [1]. Para ello utilizaremos la gestión de eventos y el paso de mensajes entre objetos del lenguaje propio de *MetaCard*: *MetaTalk.*

1.1 Objetivos

El objetivo global es exponer formas de comunicación entre aplicaciones desarrolladas con *MetaCard.* Para ello nos centraremos en:

- El envío de mensajes entre pilas.
- El uso de la orden **call** como una alternativa a la orden **send**.

Ambas son órdenes para generar eventos. El interés de la orden **call** es que permite hacerlo sin cambiar el contexto. Veremos la importancia de esto más adelante en cuanto presentemos las limitaciones de **send** y el concepto de contexto de un objeto.

En este documento se exponen pequeños listados de código relacionados con las diferentes órdenes que se exponen para dar pie al usuario a que investigue su uso, ejecutando la aplicación, probando el código y consultando la ayuda incluída dentro de la propia herramienta. No espere largas y detalladas explicaciones del código: sea activo, pruébelo.

2 Introducción

Veamos que es posible comunicar dos pilas en tiempo de ejecución mediante el envío de mensajes con el fin de realizar una acción en paralelo, repartiéndose el tiempo de procesador de la máquina y de ofrecer un mecanismo de enviar información sin interrumpir la ejecución de ninguna de las dos pilas, ni que esta hayan de estar continuamente monitorizando lo que sucede en la otra pila.

En *MetaCard*, las cosas no suceden hasta que llega un evento que las desencadene. El objeto al que le llega un evento, reaccionará ante él si dispone de un manejador para ese evento.

Los eventos se pueden originar de dos formas: sintéticamente o por el sistema. En el primer caso, los eventos sintéticos, son los que que generemos con la orden **send** de *MetaTalk*. El segundo caso en que el sistema genera los eventos es el más frecuente. De este tipo seguro que ya tiene experiencia, por ejemplo, el lector que ya haya desarrollado algún ejemplo seguro que ha utilizado el evento **mouseUp** para realizar una acción cuando se “pincha” con el ratón sobre un objeto. Esto es, habrá declarado un manejador con *MetaTalk* para ese evento, en el código de un determinado objeto,

Esto se circunscribe a los objetos en una pila, pero ¿qué pasa si los objetos están en pilas diferentes? Veamos cómo se implementa el envío de mensajes entre pilas diferentes y qué es el “contexto de un objeto”.

La fig. 1 muestra dos pilas (una subpila y una pila principal para ser más exactos) que realizan bucles y muestran el contenido de variables. Es el entorno de experimentación para ver la diferencia y la utilidad de las instrucciones de envío y cancelación de mensajes. La subpila que llamaremos "aux" (la de la derecha en la fig. 1) aparece en cuanto abrimos la principal, por que hemos dispuesto la instrucción correspondiente en el código de la pila principal, véase listado 1

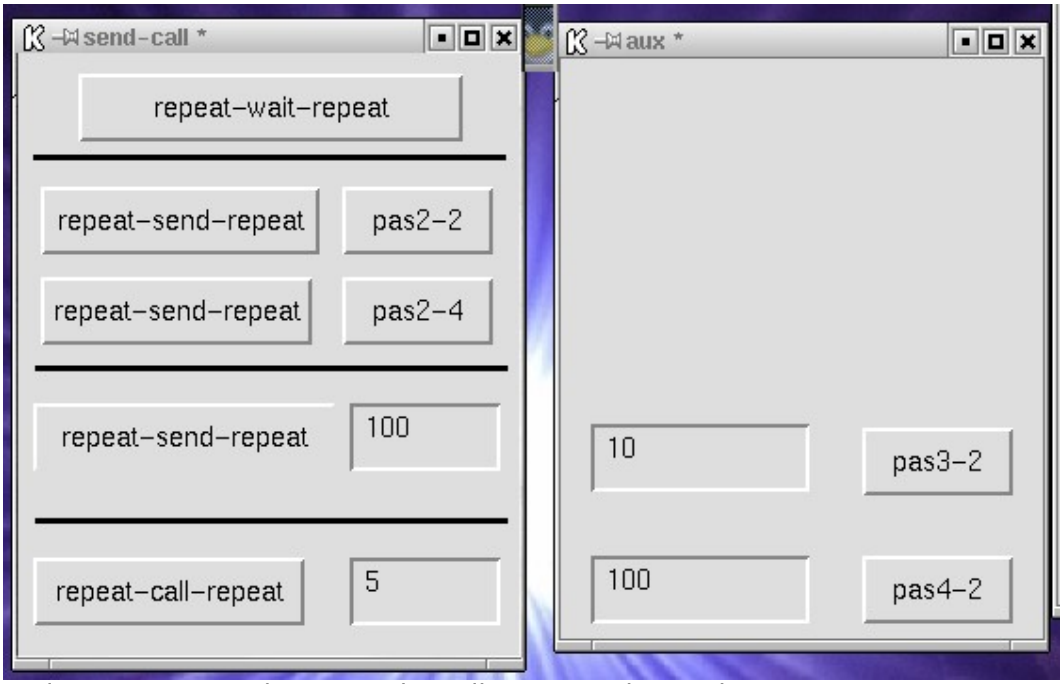

Figura 1: Instrucciones send y call: entorno de pruebas compuesto por una pila principal (izquierda) y una subpila (llamada "aux" a la derecha).

```
on openCard
  go stack "aux"
end openCard
```

Listado 1. Abriendo la subpila "aux" desde la pila inicial.

2.1 Implementación básica

La primera, posiblemente, sería el típico bucle **repeat** en el que se ha introducido la orden **wait** para que tenga la máquina ocupada y también al usuario esperando, puesto que no puede ser interrumpida. Se muestra en el listado 2, indicando a qué objeto pertenece esta porción de código.

Lo peor de esta versión es que durante 102 segundos no se puede hacer otra cosa que esperar: no es posible romper esta secuencia de código aunque se de

alguna razón para hacerlo, como por ejemplo ... que el usuario quiera hacerlo. ¡Bien pensado!

```
# Botón repeat-wait-repeat pas1
on mouseUp
  repeat with i = 1 to 10
    put i
    wait 100 milliseconds
  end repeat
  wait 102 seconds
  repeat with i = 11 to 20
    put i
    wait 100 milliseconds
  end repeat
end mouseUp
```

Listado 2. Implementación basada en un bucle sin posibilidad de ser cancelada.

3 Desarrollo de soluciones alternativas

Vamos a ver tres situaciones y su solución en base al código que las implementa. Todas ellas sobre el mismo ejemplo que se acaba de esbozar y que se resume en la en la fig. 1 .

Los trazos gruesos de la pila principal, la que se muestra a la izquierda en la fig. 1, se observará que existen cuatro agrupaciones de controles separadas por aquellos trazos. Separan tres variantes de implementaciones de cómo realizar el mismo bucle.

Ahora que hemos planteado el problema inicial, desarrollaremos las alternativas de solución en los siguientes subapartados.

3.1 Reescribir los bucles como mensajes

La segunda versión demuestra que es totalmente funcional repartir el código entre dos objetos y utilizar la orden **send** para generar un evento que haga que se ejecute el código situado en otro objeto.

Terminada la ejecución de aquel, el control retorna a la instrucción siguiente a la que envió el mensaje. Con lo que el resultado final es el mismo que en la versión anterior. Lo vemos en el código de los listados 3 y 4, donde se indica a qué objeto pertenece cada trozo de código.

```
# Botón pas2-1
on mouseUp
  repeat with i = 1 to 10
    put i
    wait 100 milliseconds
  end repeat
  send "mouseUp" to button "pas2-2"
  repeat with i = 11 to 20
    put i
    wait 100 milliseconds
  end repeat
end mouseUp

# Botón pas2-2
on mouseUp
  wait 2 seconds
end mouseUp
```

Listado 3. Implementación de un bucle con instrucciones de mensajes: pila principal.

Y así podemos seguir anidando llamadas. Incluso desde el segundo botón de vuelta hacia el primero como demuestra la segunda fila de este segundo bloque en la que se ha añadido una instrucción al segundo botón, en este caso llamado "pas2-4".

```
# Botón pas2-3
on mouseUp
  repeat with i = 1 to 10
    put i
    wait 100 milliseconds
  end repeat
  send "mouseUp" to button "pas2-4"
  repeat with i = 11 to 20
    put i
    wait 100 milliseconds
  end repeat
```

```
end mouseUp


on acabarConter
  answer warning "I si no l'acabe , simpatic?!"
end acabarConter


# Botón pas2-4
on mouseUp
  wait 2 seconds
  send "acabarConter" to button "pas2-3"
end mouseUp
```

Listado 4. Implementación de un bucle con instrucciones de mensajes: “aux”, pila secundaria, segunda parte.

3.2 Repartir el código entre las dos aplicaciones

En la tercera implementación, vamos a dar un salto repartiendo los controles, de manera que simule una situación en la que tuviesen que trabajar en conjunto y que estén físicamente situados en dos pilas diferentes. Llegados a este punto, nuestro sencillo ejercicio se convierte en una sencilla situación de comunicación entre pilas.

```
# Botón pas3-1
local delay


on mouseUp
  hide me
  put field "valor" into delay
  repeat with i = 1 to 10
    wait delay milliseconds
  end repeat
  put "(3-1) delay" && delay
  send "mouseUp" to button "pas3-2" of stack "aux"
  repeat with i = 11 to 20
    wait delay milliseconds
```

```
  end repeat
  show me
end mouseUp
```

Listado 5. Implementación de un bucle con instrucciones de mensajes: “aux”, pila secundaria.

Ahora los controles ejecutan el mismo código pero los valores de espera los obtienen de los campos de texto; con lo que el usuario podría, en tiempo de ejecución, modificarlos y con ellos el comportamiento de la aplicación. El listado 5 muestra esta aproximación.

Para ello se ha creado un campo de texto “valor” que contiene el número de milisegundos con que se simula la actividad de cada pasada del bucle y lo hemos replicado en ambas pilas. Para hacer más divertido el código, hemos hecho que los objetos se oculten cuando reciban el evento y reaparezcan al final. Así se pueden tomar referencias para contar tiempos y el código ha variado mínimamente. Y, para acabar de complicarlo, habrá observado que el botón *pas3-2* a quien llama ahora está en la subpila *aux*.

¿Ve la diferencia de la ejecución aunque el código de este botón sea el mismo de los anteriores *pas2-2* y *pas2-4*? Efectivamente, utiliza el valor de un campo de texto que se llama "valor", pero como sólo se utiliza el nombre corto del mismo, se asume que es el que está en la tarjeta activa de la pila activa. Y, ejecutando el código de *pas3-2, véase listado 6,* es la única tarjeta de la subpila *aux*.

```
# Botón pas3-2
local delay


on mouseUp
  hide me
  put field "valor" into delay
  wait delay seconds
  put "(3-2) delay" && delay
  show me
end mouseUp
```

Listado 6. Implementación de un bucle con instrucciones de mensajes: “aux”, pila secundaria.

Con lo que se tienen valores diferentes en ambos campos de texto "valor" ... Lo cual no es tan difícil que suceda: a veces utilizamos el mismo nombre para dos controles diferentes y no solemos especificar el nombre completo de un control,

en este caso tenemos un *button* "*pas3-2*" *of card 1 of stack* "*send-call*" y un *button* "*pas3-2*" *of card 1 of stack* "*aux*". Por supuesto la referencia a la tarjeta se podría haber omitido, puesto que en nuestro caso sólo hay una en cada pila.

3.3 Generalización para un número de controles alto

Esa permisividad a la hora de especificar completamente el nombre de un control que, en principio, es una ventaja; lo deja de ser si tenemos muchos controles en esa situación y hemos de ir continuamente referenciándolos por su nombre completo.

No sólo por el trabajo de escribirlos, sino por que de no hacerlo podríamos estar trabajando con los datos equivocados y el código sería correcto.

Para solucionarlo, podemos hacer uso de la orden **call** que generará el evento sobre el objeto; pero sin cambiar el contexto, es decir, la referencia a donde localizar los objetos (de hecho puede ver como la ejecución de la orden

```
answer the defaultStack
```

que está en el botón "pas4-2" y que devuelve el nombre de la pila principal en lugar de la pila "aux". El listado 7 muestra el código correspondiente a los nuevos controles que implementan esta aproximación.

```
# Botón pas4-1
local delay
on mouseUp
  hide me
  put field "valor2" into delay
  repeat with i = 1 to 10
    wait delay milliseconds
  end repeat
  put "(" & the short name of me & ") delay" && delay
  call "mouseUp" to button "pas4-2" of stack "aux"
  repeat with i = 11 to 20
    wait delay milliseconds
  end repeat
  show me
end mouseUp
```

```
# Botón pas4-2
local delay

on mouseUp
  hide me
 answer the defaultStack
  --  answer information delay
  put field "valor2" into delay
  put "(" & the short name of me & ") delay" && delay
  wait delay seconds

  show me
end mouseUp
```

Listado 7. Implementación de un bucle con instrucciones **call,** en lugar de **send**.

4 Conclusiones

En este trabajo se han mostrado los elementos básicos para la comunicación entre aplicaciones desarrolladas con *MetaCard.* Hemos experimentado sobre vías de comunicación, en tiempo de ejecución. Para ello hemos echo uso de los eventos y el paso de mensajes entre objetos.

Hemos avanzado examinando el código alternativo entregado, hasta llegar al que los mensajes se envíen entre pilas y hasta el uso de la orden **call** como una alternativa a la orden con que se generan habitualmente lo eventos, la orden **send**.

5 Bibliografía

[1] *MetaCard*. Disponible en <http://www.metacard.com/>