Patrones de comunicación entre componentes

por Fernando Dodino,
Nicolás Passerini,
Franco Bulgarelli

Versión 1.1-borrador

Mayo 2014

Índice

1 Introducción
2 Patrones para comunicar dos componentes: una clasificación
3 Call & return
    3.1 Solución en Haskell
        3.1.1 Consecuencias
    3.2 Solución en Groovy
    3.3 Efecto e inmutabilidad
4 Memoria compartida / Call-By-Reference
    4.1 Pila en C con memoria compartida
    4.2 Ejemplo en Groovy
    4.3 Pila en C con call by reference
    4.4 Otros ejemplos de memoria compartida
        4.4.1 Colas de mensajes
5 Excepciones
6 Continuaciones
    6.1 Introducción
    6.2 Implementando un try/catch en javascript
    6.3 Ejemplo en Seaside: una calculadora web
7 Eventos
    7.1 Interrupciones/Signals
8 Streams/Pipes
9 Conclusiones

1 Introducción

Cuando vimos el diseño de interfaces entre componentes[1], discutimos cómo diseñar la forma de comunicación entre dos componentes A y B (que pueden o no pertenecer al mismo módulo/sistema).

Algunas cosas que dijimos que podíamos tener en cuenta eran la siguientes:

- qué mensajes le mando
- quién le envía el pedido a quién
- qué información necesita el componente
- hacia qué lado va el conocimiento (qué dirección tiene)
- la tecnología concreta de comunicación: si el mensaje se procesa utilizando *sockets*, archivos, una base de datos, una cola de mensajes, *web services*, etc.
- si el mensaje es sincrónico o asincrónico
- qué interfaz voy a publicar si soy el que publica algo hacia afuera para que me usen
- qué interfaz voy a definir para mis objetos de negocio para comunicarme con el otro componente

En este texto, no hablaremos tanto de cuestiones de diseño, sino más bien haremos un tratamiento teórico de distintas formas de comunicar componentes. Y recuerden que cuando acá hablemos de componentes en general pondremos ejemplos en objetos, pero que realmente estas ideas aplican a muchas otras tecnologías[2].

2 Patrones para comunicar dos componentes: una clasificación

La forma en que se comunican A y B puede seguir alguno de los siguientes patrones de comunicación:

- Call & Return
- Memoria compartida
- Excepciones
- Continuaciones
- Eventos
- Paso de mensajes asincrónicos
- Streams/Pipes

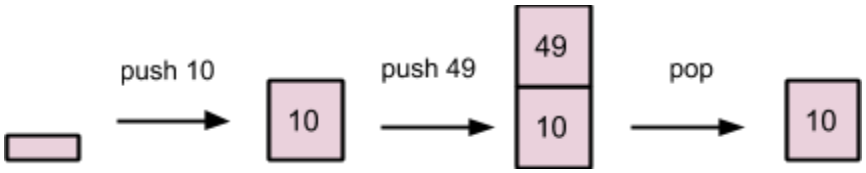
[1] Leer Diseño de interfaces entre componentes

[2] Recordar acá qué es un componente, también explicado en el apunte de Diseño de Interfaces

Comenzaremos mostrando varias formas de modelar una pila y la forma en que se utiliza. El objetivo es

1. crear una pila vacía
2. apilar (push) el número 10
3. luego el 49 (otro push)
4. y hacer un pop

La secuencia que muestra el estado de la pila sería:



3 Call & return

En el mecanismo de call & return la comunicación se da entre un componente invocante o llamador (*caller*) que invoca o referencia a otro componente invocado o llamado (*callee*). Si bien el flujo de información es bidireccional, es asimétrico:

- El componente invocado puede recibir *parámetros*, que le permiten al invocante transferirle información mediante *argumentos*.
- En el caso general el componente invocado puede producir un *valor de retorno*, mediante este mecanismo se logra la bidireccionalidad de la comunicación. En algunas tecnologías pueden existir limitaciones que restringen el uso de valores de retorno, en ese caso la comunicación será unidireccional.

3.1 Solución en Haskell

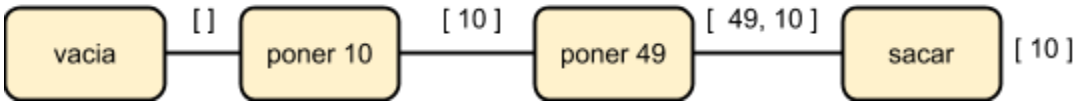
El ejemplo más preciso de esta idea se encuentra en los lenguajes funcionales puros, es decir, sin la posibilidad de efecto. La ausencia de efecto obliga a que toda la comunicación sea por medio de los parámetros.

```
type Pila a = [a]

vacia :: Pila a
vacia = []

poner :: a -> Pila a -> Pila a
poner elem pila = elem:pila

sacar :: Pila a -> Pila a
sacar (elem:resto) = resto
```

Claro, “sacar” o “poner” en realidad implica generar una nueva pila, no modificar la existente. Partimos de una pila generada con la función `vacia`. Entonces:

1. `poner 10 vacia` apila el elemento 10 en una lista vacía
2. a partir de aquí, la composición nos permite encadenar las sucesivas llamadas (call & return) para obtener los distintos estados en los que queda la pila:

```
> (sacar . poner 49 . poner 10) vacia
[10]
```



3.1.1 Consecuencias

Si bien la comunicación puede ser bidireccional, el conocimiento (y por lo tanto el acoplamiento) es *a priori* unidireccional, es decir, el componente llamado no tiene ningún conocimiento del origen del mensaje y aún puede devolver información sin tener conocimiento del destino de la misma.

3.2 Solución en Groovy

Mientras que en Haskell definimos un tipo de dato, en Groovy generamos un objeto Pila como abstracción. Pero nosotros podemos mantener el esquema call & return, haciendo que tanto poner como sacar devuelvan una nueva pila:

```
def poner(valor) {
	new Pila(valores + valor)
}

def sacar() {
	new Pila(valores - tope())
}
```

donde tope() se define como

```
def tope() {
	if (valores) {
		valores.last()
	} else {
		null
	}
}
```

Las operaciones + y - sobre el conjunto de valores construyen **un nuevo valor** como vemos en el contrato de groovy.List

minus (-)

public List **minus**(Object removeMe)

Create a ***new List*** composed of the elements of the first list minus every occurrence of the given element to remove.
assert ["a", 5, 5, true] - 5 == ["a", true]

Parameters:

removeMe - an element to remove from the list.

Returns:

the resulting List with the given element removed

Since:

1.0

Lo mismo para plus (+).

Ese nuevo valor (la lista con los elementos que conforman la pila) se pasa al constructor privado de Pila:

```
/** Constante que permite definir una pila vacía */
public static empty = new Pila()

/** Constructores privados */
private Pila() { }

private Pila(valores) {
	this.valores = valores
}
```

Entonces si bien tiene estado, la Pila es un objeto **inmutable**, la única forma de modificarla es crear un nuevo objeto Pila. La solución es similar a la que planteamos anteriormente en Haskell:

```
Pila.empty.poner(10).poner(49).sacar()
```

Más allá de la notación objeto-mensaje esta solución no difiere mucho de la anterior, se siguen encadenando los mensajes de manera que el output resultante del último mensaje se utiliza como input del llamado al siguiente.

Aun cuando el lenguaje soporta trabajar con efecto *somos nosotros los que evitamos que la Pila tenga efecto*: es una decisión **de diseño** que esto ocurra.

3.3 Efecto e inmutabilidad

La idea de call and return está asociada a la idea de inmutabilidad. Por ejemplo, al sumar o restar BigDecimals de Java, se devuelve un nuevo objeto. Entonces si queremos que haya efecto, tenemos que trabajar la asignación manualmente:

```
>>Cliente
public void pagar(BigDecimal monto) {
      saldo = saldo.subtract(monto);
}
```

De la misma manera, el List de Scala es inmutable, entonces no podemos “agregar” ni “eliminar” elementos, sin guardar la referencia a la nueva lista (y desechando la anterior):

```
>>Cliente
def facturar(factura: Factura) {
    facturas = facturas :+ factura
}
```

4 Memoria compartida / Call-By-Reference

4.1 Pila en C con memoria compartida

Volvemos a nuestro ejemplo de la pila, ahora en lenguaje C:

```
#include <stdio.h>
#define SIZE 50

/** Estructura de datos global **/
int current, stack[SIZE];

void init() {
      current = 0;
}

int top(void) {
      return stack[current];
}
```

Tenemos dos variables globales: la pila y la posición actual del último elemento. Ahora definimos operaciones push() y pop() -equivalentes a poner y sacar-:

```
void push(int value) {
```

```
	if (current == MAX_SIZE) {
		printf("Stack Overflow.\n");
		exit(1);
	}
	current++;
	stack[current] = value;
}

void pop(void) {
	if (current == 0) {
		printf("Stack Underflow.\n");
		exit(2);
	}
	current--;
}
```

Ambas funciones tienen **efecto colateral**, porque modifican el estado de las variables cuyo alcance supera al contexto de las funciones push y pop. Estas variables globales forman nuestro espacio de memoria compartido. Vemos cómo queda nuestro main:

```
int main(void) {
	init();
	push(10);
	push(49);
	pop();
	printf("el tope de la pila es %d\n", top());
	return 0;
}
```

Algunas consecuencias de esta forma de compartir información son:

- No es posible tener dos pilas simultáneamente, en caso de intentarlo se mezclará la información de ambas. Eso complica enormemente la posibilidad de tener testeo unitario automatizado: al tener una única estructura global no puedo generar un fixture o juego de datos con diferentes pilas y tampoco es fácil que cada prueba sea independiente (porque hay que deshacer manualmente los efectos de cada test)
- La modificación de cualquiera de las operaciones que acceden a la estructura de datos implica revisar su buen comportamiento en relación con todas las demás (que, como se dijo antes, puede no ser posible saber exactamente cuáles son). Lo mismo ocurre si se desea modificar la estructura de los datos compartidos. Por este motivo trabajar con memorias compartidas globales producen componentes con un alto grado de acoplamiento.

Por otro lado, existen tecnologías que permiten definir variables con alcance o "scope" más

limitado que *global*, reduciendo de esta manera el acoplamiento únicamente a los componentes que tienen acceso al *scope* específico. Un caso particular de *scope* son las variables de instancia de un objeto, como se muestra en el ejemplo siguiente.

4.2 Ejemplo en Groovy

Volvemos sobre el ejemplo de la pila utilizando el concepto de memoria compartida en objetos. Entonces la pila tendrá un estado **mutable**, vamos a utilizarla como un lugar donde cada envío de mensaje tendrá efecto colateral y ese efecto determinará el estado final que quiero conseguir.

Vemos la implementación, donde aprovecharemos los métodos push y pop propios del List de groovy:

Clase Pila que define un atributo

```
valores = []

...
```

y los métodos poner y sacar delegan a los correspondientes push() y pop() de las colecciones de Groovy:

```
def poner(valor) {
      valores.push(valor)
}

def sacar() {
      valores.pop()
}
```

Vemos cómo se trabaja la Pila:

```
pila = new Pila()
pila.poner(10)
pila.poner(49)
pila.sacar()
```

Podríamos encadenar los mensajes con unos pequeños cambios:

```
def poner(valor) {
      valores.push(valor)
      this
}
```

```
def sacar() {
	valores.pop()
	this
}
```

Eso nos permite encadenar los mensajes al trabajar con la pila:
Vemos cómo se trabaja la Pila:

```
pila = new Pila().poner(10).poner(49).sacar()
```

Y si bien la forma de usar la pila se parece al patrón call & return, aquí hay una importante diferencia: la forma de comunicarse entre el poner y sacar es a través de un espacio de memoria compartido. No es una variable global, tiene un scope menor cuyo alcance lo da el objeto a través de un **efecto colateral buscado**. Otra ventaja es que puedo referenciar a dos pilas con estados diferentes que no se mezclan entre sí.

4.3 Pila en C con call by reference

Un ejemplo equivalente al que vimos en Groovy con memoria compartida se da utilizando a la pila en C como una estructura o **tipo abstracto de dato** (TAD) a la que pasamos por referencia, para poder darle el efecto colateral buscado en las funciones pop() y push(int)[3]:

```
int main(void) {
	stack* pila = empty();
	push(pila, 10);
	push(pila, 49);
	pop(pila);
	printf("El tope es %d\n", top(pila));
	return 0;
}
```

Vemos la definición de la pila:

```
typedef struct stack {
	int current, stack[SIZE];
} stack;
```

y cómo implementar pop() y push():

```
void push(stack* pila, int value) {
	if (pila->current == MAX_SIZE) {
		printf("Stack Overflow.\n");
		exit(1);
	}
```

[3] Recomendamos al lector ver el ejemplo de la cátedra que está implementado con testeo unitario automatizado

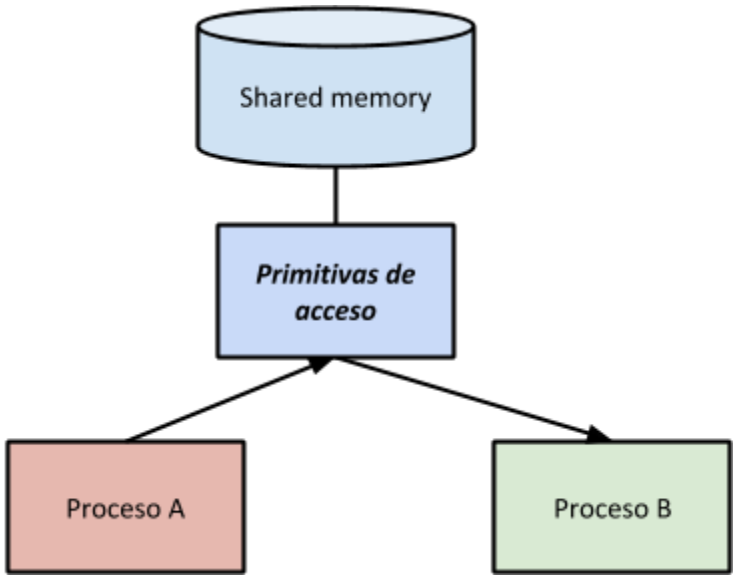
```
	pila->current++;
	pila->stack[pila->current] = value;
}

void pop(stack* pila) {
	if (pila->current == EMPTY_STACK) {
		printf("Stack Underflow.\n");
		exit(2);
	}
	pila->current--;
}
```

Las líneas marcadas en negrita y cursiva marcan dónde se produce el efecto en la pila.

4.4 Otros ejemplos de memoria compartida

La memoria compartida es un espacio común de datos en el que múltiples componentes (por ejemplo: procedimientos) pueden leer y escribir información. Implementaciones típicas de memoria compartida son las variables globales o las bases de datos como herramienta de integración entre componentes.

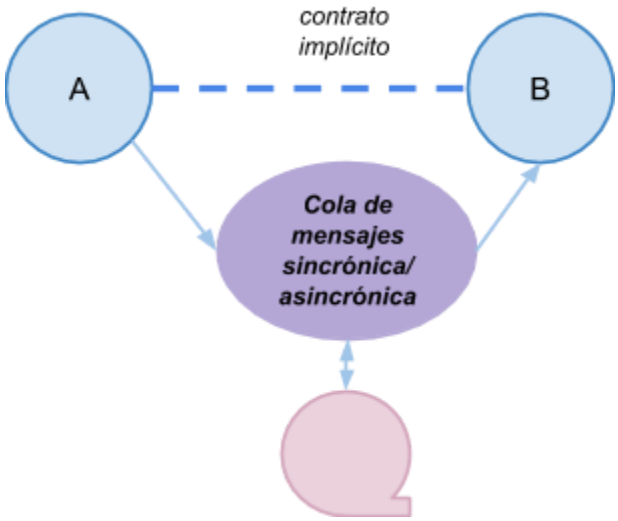
Al diseñar está bueno modelar un componente que defina primitivas (para acceder a las variables globales, o bien a las tablas) en lugar de acceder directamente a esa memoria compartida:



4.4.1 Colas de mensajes

La idea de memoria compartida muchas veces se utiliza entre bastidores para implementar otros patrones de comunicación como, por ejemplo, el de paso de mensajes.

En muchos casos la comunicación entre A y B no es trivial. Por ejemplo, en algunas tecnologías, los mensajes puede perderse, o el receptor estar saturado[4]. Entonces necesitamos un componente C o *middleware* que maneje la comunicación[5].



El tratamiento de colas de mensajes merece un apunte aparte. Para el curioso, dejamos en link a RabbitMQ, que es no sólo una implementación igualmente completa y simple de distintos tipos de configuraciones de colas de mensajes, sino que también cuenta con excelentes ejemplos introductorios.

5 Excepciones

Las excepciones son otra forma de establecer una comunicación entre dos componentes, donde

- se comparte estado o información: un mensaje de error, el estado de un objeto o el error original
- se modifica el flujo de envío de mensajes.

Recomendamos leer el apunte específico del tema.

[4] Esto no ocurre en el envío de mensajes entre objetos, dado que en su forma más básica siempre el receptor está disponible (está en memoria), los mensajes que se le envían siempre llegan, independientemente de su tamaño y condiciones externas, y el receptor responde de forma sincrónica. Es decir, un sistema de objetos es un caso ideal de comunicación entre dos componentes: se tiene ancho de banda ilimitado, el receptor siempre está disponible, y la comunicación libre está errores.
[5] Recomendamos la lectura de los artículos http://en.wikipedia.org/wiki/Message_passing | http://docs.oracle.com/javaee/1.3/jms/tutorial/ | http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html

6 Continuaciones

6.1 Introducción

Supongamos que tenemos la función even, en javascript

```
function even(n) {
      return n % 2 == 0;
}
```

Vamos a modificar la interfaz de la función para que acepte un parámetro más: una función que nos diga qué hacer cuando determinamos si n es o no par, cómo **continuar**.

`function evenCPS(n, f) {`
`  f(n % 2 == 0);`
`}`
	Equivalente en Haskell:
	`evenCPS value funcion`
	`  = (funcion . even) value`

La función se puede evaluar de muchas formas posibles:

```
evenCPS(2, alert);

evenCPS(2, function(valor) {
        document.write(valor + "<br>")
});
```

En el primero de los casos pasamos una función estándar alert de javascript que muestra mensaje en una ventana pop up. En el segundo caso creamos una lambda, closure, o función anónima que imprime en un documento HTML el valor que recibe como parámetro.

```
function(valor) {
        document.write(valor + "<br>")
}
```

es equivalente a

```
\valor -> imprimir valor          de Haskell o
[ valor | println(valor) ]        de Xtend.
```

¿Por qué el sufijo CPS? Porque el formato de estas funciones recibe el nombre **continuation passing style**.

La **continuación** es un concepto que representa

- el estado actual del programa (el juego de valores, el contexto)

- el conjunto de instrucciones a ejecutar a partir de aquí.

Ahora bien, ¿qué ventajas trae pensar de esta manera?

- puedo modificar fácilmente el "flujo de ejecución de un programa", o yendo a algo más propio de diseño: puedo diferir o adelantar el momento en que cada componente resuelve su responsabilidad
- relacionado con esto es que puedo pensar en componentes que trabajen *asincrónicamente*, como sucedía en el patrón Command
    - genero o reifico una responsabilidad (implementado a través de una abstracción que representa código: se puede llamar command, función, expresión lambda, método, closure, etc.)
    - y se la paso a otro componente para que se evalúe/ejecute después
    - incluso puedo pensar en que sólo se evalúe/ejecute en determinadas circunstancias (cuando no hubo error en el otro componente, por ejemplo)

6.2 Implementando un try/catch en javascript

Queremos dividir dos números que ingresamos en una página web, donde tendremos dos inputs (el dividendo y el divisor) y un link para evaluar el cálculo[6]:

```
<html>
<head>
      ...
</head>
<body>
      <div>
            Dividendo:
            <input type="text" id="dividendo">
      </div>
      <div>
            Divisor:
            <input type="text" id="divisor" title="distinto de cero">
      </div>
      <div>
            <a href="javascript:dividir();">Dividir</a>
      </div>
      <div id="mensaje" class="mensaje"></div>
</body>
</html>
```

Está claro que no podemos dividir por cero, entonces la división propiamente dicha tendrá cuatro parámetros:

[6] Para profundizar en el diseño de aplicaciones web véanse los apuntes de la cátedra

- el valor del dividendo
- el valor del divisor
- una función que nos diga qué hacer si la división se puede resolver
- una función que nos diga qué hacer si hay un error al querer hacer la división.

```
function dividir() {
	divisionTryCatch(
		$("dividendo").value, // parámetro 1
		$("divisor").value, // parámetro 2
		function(res) {
			$("mensaje").innerHTML = "El resultado es " + res;
		}, // parámetro 3: construyo una lambda anónima
		muestraError); // parámetro 4: paso una función existente
}

function divisionTryCatch(dividendo, divisor, anteExito, anteFalla) {
	if (divisor === 0) {
		anteFalla("el divisor debe ser distinto de cero");
	} else {
		anteExito(dividendo / divisor);
	}
}

function muestraError(ex) {
	$("mensaje").innerHTML = "";
	alert("Error al dividir: " + ex);
}

function $(id) {
	return document.getElementById(id);
}
```

Para simplificar este ejemplo didáctico, asumimos que el usuario ingresa valores numéricos correctos.

Si bien en javascript contamos con una estructura de control try/catch, este esquema que trabaja con funciones es bastante popular porque nos permite trabajar en forma asincrónica, como ocurre cuando disparamos pedidos AJAX[7].

[7] El lector curioso puede ver los siguientes artículos: http://matt.might.net/articles/by-example-continuation-passing-style/ | http://marijnhaverbeke.nl/cps/ | http://www.slideshare.net/domenicdenicola/callbacks-promises-and-coroutines-oh-my-the-evolution-of-asynchronicity-in-javascript

6.3 Ejemplo en Seaside: una calculadora web

El framework web Seaside[8] (disponible para varias versiones de Smalltalk) permite trabajar el concepto de continuation de una forma más que interesante. Veamos el siguiente ejemplo: creamos una clase CIUCalculadora, que hereda de WATask, una clase que viene con el framework Seaside y que define un método go con una serie de pasos a cumplir:

```
go
    |sumando1 sumando2|
    sumando1 := (self request: 'Ingrese el primer sumando') asNumber.
    sumando2 := (self request: 'Ingrese el segundo sumando') asNumber.
    self inform: (sumando1 + sumando2) asString.
```

Al ingresar por el browser a la aplicación web, vemos algo no tan trivial de conseguir en otras tecnologías:

- Al usuario se le presenta una pantalla con un input y el mensaje "**Ingrese el primer sumando**". Al hacer submit ...
- aparece una nueva segunda pantalla con otro input y el mensaje "**Ingrese el segundo sumando**". Y al hacer submit...
- aparece una tercer pantalla con el resultado de la suma.

Vemos entonces que cada pantalla es exactamente cada una de las líneas del método "go". Esas líneas de código mágicas, utilizan mensajes especiales como "**request**" e "**inform",** que se podrían pensar como pasos de una secuencia (o wizard). Y esos pasos serían de hecho pedidos al usuario, que requieren su interacción.

Si bien a la vista de quien programa parece estar todo el comportamiento en un único método, Seaside se encarga de generar **continuations** de manera de separar cada paso en una página diferente. Además sabe mantener el estado (value1, value2, etc.) y la navegación.

7 Eventos

El patrón Observer muestra otra forma común de establecer la comunicación entre dos componentes:

[8] www.seaside.st

- en un momento t_1 B se registra como interesado en determinados eventos de A
- en diferentes momentos t_2, t_3 el componente A indica a B que se produjeron determinados eventos, ya sea porque
    - el estado de A cambió o está a punto de cambiar
    - porque A recibió un mensaje esperado
    - etc.

Hemos visto que esta técnica permite bajar el acoplamiento entre ambos componentes, en general A no conoce todos los interesados en cada evento, pero podría pasar que B también tuviera un observado que fuera un objeto polimórfico.

Este mecanismo de comunicación sugiere un esquema **publisher-subscribe**, donde el suscriptor/observer sólo recibe notificaciones cuando determinados eventos se producen, en lugar de tener que estar chequeando continuamente si hay novedades, lo que se conoce como polling.

En el caso de los observers, la reificación del evento es manual: debemos diseñarlo nosotros, en otros lenguajes existe la posibilidad de tener *eventos nativos*, como veremos a continuación.

7.1 Interrupciones/Signals

Las signals son mecanismos de bajo nivel que permiten establecer notificaciones ante interrupciones.

Vemos un ejemplo[9] [10]:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void sighup(); /* routines child will call upon sigtrap */
void sigint();
void sigquit();
```

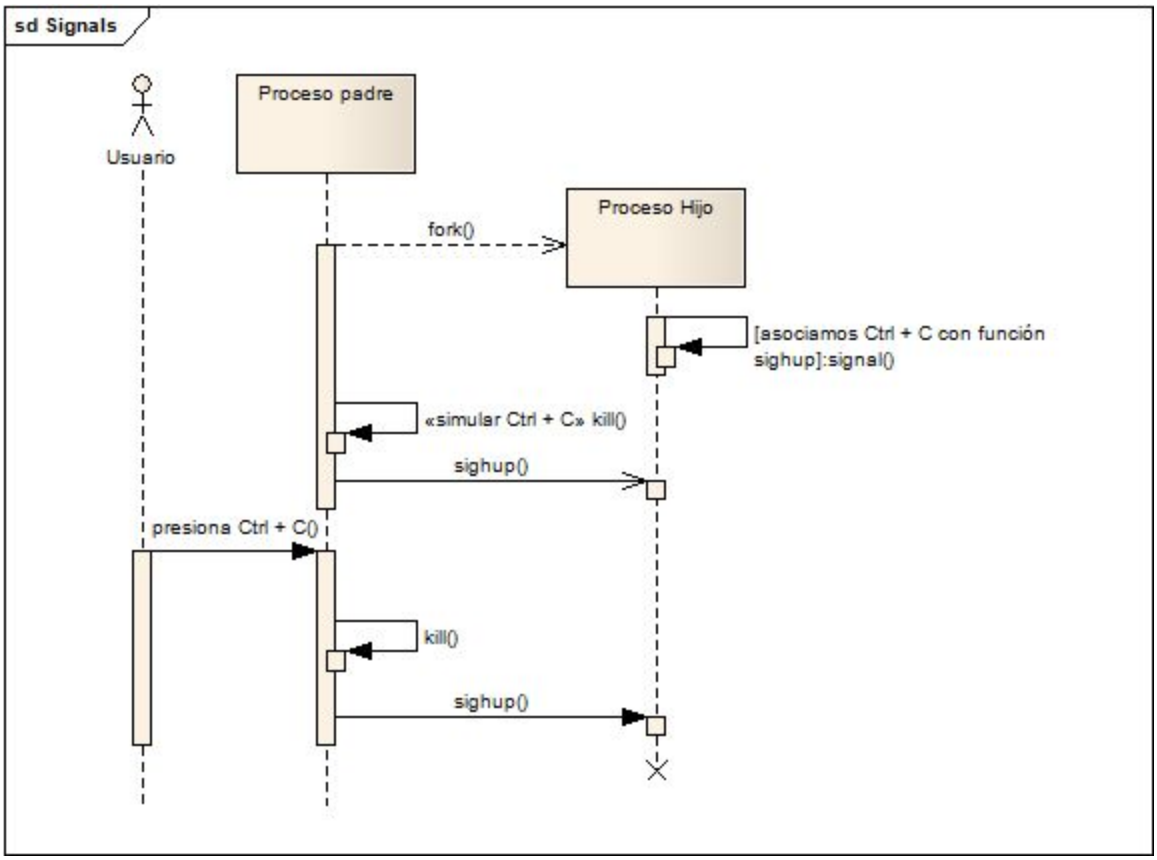
[9] Este ejemplo extraído de http://www.cs.cf.ac.uk/Dave/C/node24.html tiene como objetivo presentar las signals como un mecanismo de comunicación entre procesos (IPC), dejamos para más adelante muchas refactorizaciones posibles para una clase de Diseño Estructurado: reificar el proceso como una abstracción, reemplazar así el entero pid por funciones de un TAD proceso, discriminar el comportamiento del proceso padre e hijo, encontrar funciones que tengan nombres más representativos y eliminen la necesidad de comentar tanto código, etc.

[10] Recomendamos los siguientes links: http://pymotw.com/2/signal/ (muestra implementaciones en Python) y estas diapositivas de la Universidad de Princeton.

```c
int main(void) {
    /* Creamos un proceso hijo - TODO: Manejar el error */
    int pid = fork();
    int procesoHijo = pid == 0;

    if (procesoHijo) { /* Proceso hijo */
          /* Asociamos funciones para responder a cada signal */
          signal(SIGHUP, sighup);
          signal(SIGINT, sigint);
          signal(SIGBUS, sigquit);
          for (;;)
                ; /* loop infinito */
    } else /* Proceso padre */{
          /* pid mantiene el identificador del proceso hijo */
          sleep(20);
          printf("\nPARENT: sending SIGHUP\n\n");
          kill(pid, SIGHUP);
          sleep(10);
          printf("\nPARENT: sending SIGINT\n\n");
          kill(pid, SIGINT);
          sleep(10);
          printf("\nPARENT: sending SIGBUS\n\n");
          sleep(2);
          kill(pid, SIGBUS);
          sleep(10);
    }
    return 0;

}

void sighup() {
    printf("CHILD: I have received a SIGHUP\n");
    signal(SIGHUP, sighup); /* reset signal */
}

void sigint() {
    printf("CHILD: I have received a SIGINT\n");
    signal(SIGINT, sigint); /* reset signal */
}

void sigquit() {
    printf("My DADDY has Killed me!!!\n");
```

```
    exit(0);
}
```

Acompañamos un diagrama de secuencia de la solución:



Podemos ver cómo corre cada proceso en el sistema operativo utilizando la técnica de pipes, que se explica en la siguiente sección:

```
~$ ps -fe | grep ipc-signal | grep -v grep
fer  2349  1978  0 12:01 ?  00:00:00 /home/fer/ipc-signal-C/Debug/ipc-signal-C
fer  2353  2349 96 12:01 ?  00:00:14 /home/fer/ipc-signal-C/Debug/ipc-signal-C
```

(2349 es el PID del padre y 2352 el PID del hijo)

Si bien las signals del sistema operativo están modelando una excepción, y la forma de construir una signal se asemeja a la construcción de una continuation, las ubicamos en la categoría eventos porque:

- declaro qué se va a hacer cuando pase 'x' cosa (ese qué se va a hacer, una vez más se representa por un puntero a función, o sea una *lambda*),
- una vez manejada la señal vuelve al flujo de ejecución desde donde se recibió la señal, cosa que no pasa con las excepciones que una vez que son lanzadas, donde se corta el flujo de envío de mensajes
- entonces debemos controlar manualmente el estado en el que nos encontramos antes

de continuar la ejecución de nuestra aplicación.
- las signals funcionan con interrupciones del sistema operativo (como cuando el usuario presiona un Ctrl + C), esto entonces excluye al proceso padre disparar el kill/raise de la signal.

8 Streams/Pipes

El trabajo con pipes muestra una implementación similar del patrón call & return para comandos del sistema operativo (Linux/Unix):

```
$ ps aux | grep conky | grep -v grep | awk '{print $2}' | xargs kill[11]
```

Cada pipe separa el output del proceso anterior que es el input del pedido siguiente. En el ejemplo de arriba:

- vemos la lista de procesos activos en el sistema
- de esa lista de procesos filtramos los nombres de proceso 'conky' (esto incluye nuestro grep)
- de ese filtro eliminamos nuestro grep
- tomamos la segunda columna que tiene el identificador del proceso (pid = process id)
- y finalmente pasamos ese número de proceso a la instrucción kill para cortar la ejecución.

A continuación vemos otro ejemplo:

```
youtube-dl $1 -q -o - | ffmpeg -i - $2
```

Este script permite bajar un video de youtube (primer parámetro) y grabarlo en un archivo .mpeg (segundo parámetro). El pipe genera un buffer intermedio antes de bajarlo físicamente a un archivo del filesystem.

Para más información recomendamos http://www.linfo.org/pipes.html

9 Conclusiones

Los patrones son eso, patrones. Bien puede pasar que tengamos esquemas mixtos o que se puedan interpretar de una u otra forma... por ejemplo las signals del sistema operativo podrían pensarse como un mecanismo de excepción o como un evento en distintos contextos... no es tan importante encajar nuestra solución en una categorización exacta sino que los patrones sirven en la medida en que nos ayudan a pensar.

[11] Ejemplo extraído de http://unix.stackexchange.com/questions/30759/whats-a-good-example-of-piping-commands-together

Si bien podríamos haber analizado las excepciones, los signals (y el call & return, y las continuations) en términos del control de flujo, nos interesa pensar cómo diseñamos la comunicación entre dos componentes para poder medir qué gano y qué pierdo en cada solución. Y uno de los parámetros que más nos importa es el grado de acoplamiento que A y B tienen.

Es decir, si mi forma de comunicación es la invocación / mensaje, para A comunicarle algo a B necesita conocerlo, necesita tener una referencia explícita. Eso es lo natural, lo más sencillo. Si nos interesa buscar formas de comunicación en las que A envía información a B sin saber nada de B, entonces pensamos ejemplos alternativos:

1. B le manda un mensaje a A, claro está.
    a. A también puede comunicarse con B, a través del valor de retorno del mensaje. La comunicación que yo hago a través del valor de retorno es mucho más desacoplada de la que hago cuando mando un mensaje, no tengo idea de a quién le estoy contestando, el llamado no conoce al llamador.
    b. En un esquema con call-by-reference además puede mandarle mensajes a los parámetros. Acá el desacoplamiento se da porque yo no sé quiénes son esos parámetros, sólo pido que entiendan algún mensaje (en Java podría ser con interfaces por ejemplo). Podría asociarse con las ideas de inyección de dependencias o inversión de control.
    c. Si A tira una excepción también le está mandando info a B: no es sólo los datos en la excepción, sino el mensaje de "algo salió mal", codificado en el tipo de la excepción. Esto tiene un desacoplamiento aún más profundo que el valor de retorno, porque no tengo idea de quién va a atrapar la excepción, podría incluso no ser el que me llamó... en definitiva no me estoy comunicando con B sino con el primer componente que quiera catchearla.
    d. Una continuation es también un caso de inversión de control o de delegación extrema.
2. Eventos / Observer: Otro ejemplo donde el objeto / componente que produce el evento le comunica algo a otros sin saber quiénes son.