Programación Orientada a Objetos – Módulo 7

NOTA: Este apunte está todavía en revisión. Vamos a publicar una versión definitiva más adelante.

Casos Excepcionales

Hasta el momento hemos visto muchos casos en los que asumimos que "todo iba a resultar como esperábamos"... pero, ¿qué pasa si eso no es taaaaaan tan así? ¿Cómo nos enfrentamos a situaciones poco comunes sin llenar de "if" el programa?

Existen mensajes para los cuales uno establece qué es lo que debe ocurrir si se da una situación anormal. No digamos "inesperada", porque se supone que estamos planteando que esto puede suceder, pero sí podemos plantear que la situación no es habitual.

Como ejemplo, vamos a tomar el caso del mensaje "`detect:`", que entienden las colecciones.

A ver... la idea de este mensaje es "encontrar un objeto incluido en la colección que cumple una determinada condición".
Pero, ¿qué pasa si no existe ninguno que la cumpla?... nuestro programa "explota".

Si sabemos que para un mensaje la condición puede no cumplirse por ninguno de los objetos de la colección, entonces deberíamos usar la versión alternativa "`detect: ifNone:`". Este otro mensaje (¡no olvidemos que es un mensaje distinto!) nos permite enviar un segundo parámetro, además del primero que era un bloque que representa la condición buscada. En este segundo parámetro indicamos lo que debe hacerse en esta "situación excepcional".

Otro ejemplo, muy similar, ocurre con las que entienden el mensaje "`at:`", como por ejemplo el diccionario. Si buscamos por una clave que puede no existir, entonces tenemos que tener en cuenta la posibilidad de que efectivamente no exista. Para esto, tenemos de nuevo la posibilidad de enviar un bloque como parámetro mediante el mensaje alternativo "`at: ifAbsent:`".

Algunos ejemplos de mensajes "explosivos" (potencialmente, claro) son:

- Dictionary >> at:
  Si no tiene la clave indicada.
- Collection >> detect:
  Si no existe un elemento que cumpla la condición del bloque.
- Collection >> remove:
  Si el elemento a eliminar no existe.
- Collection >> anyOne (o any)
  Si la colección está vacía.
- Object >> tomateUnMate
  Si el objeto receptor del mensaje no lo entiende (MessageNotUnderstood).
- La división, si es por cero (ZeroDivide)

Excepciones

Pero... los que explotan, ¿por qué explotan?

Imaginen cualquiera de los casos de colecciones... ¿Tiene sentido que para el `#at:` , `#detect:` , `#anyOne` … nos devuelve `nil` en lugar de explotar? Acaso `nil` no puede ser un elemento de la colección? ¡Claro que `nil` puede ser parte de nuestra colección!

Entonces, ¿podemos pasar todas estas acciones por alto? ¿Qué pasa con este código?

```
elementoCualquiera := unaColeccion anyOne.
elementoCualquiera class.
```

Si la colección está vacía, ¿cual es ese elemento? ¿Qué clase tiene? Acá se puede observar por qué estas situaciones son excepciones: no podemos poner un “valor de retorno inválido”, porque simplemente cualquier objeto podría llegar a ser la respuesta al mensaje y, por lo tanto, algo válido.

¿Qué es lo que pasa con estos errores?
Como vimos, hay varios conocidos ya. Lo que hay que tener en mente es que todos los errores son **atrapados** por alguien.

¿Qué significa que sean atrapados? Que **siempre hay alguien que detecta la existencia de ese error y lo gestiona, hace algo al respecto**.

El último en la cadena de responsabilidad para gestionar el error es el mismo Smalltalk, que es quien al detectar el error pone todo en una ventanita y nos muestra el stack de todo lo que se ejecutó hasta llegar al error detectado.
Pero casi nunca queremos que llegue hasta ese nivel... de hecho, a eso es a lo que llamamos que algo “explote”... y nadie quiere que su programa explote.

Yyyyy... ¿Entonces?

Veamos esto con un ejemplo tomado del ejercicio de la clase pasada de trenes y depósitos. En ese ejercicio, el último punto decía:

9 - Agregar, dentro de un depósito, una locomotora a una formación determinada, de forma tal que la formación pueda moverse.

- *Si la formación ya puede moverse, entonces no se hace nada.*
- *Si no, se le agrega una locomotora suelta del depósito cuyo arrastre útil sea mayor o igual a los kilos de empuje que le faltan a la formación. Si no hay ninguna locomotora suelta que cumpla esta condición, no se hace nada.*

```
Deposito >> agregarLocomotoraA: unaFormacion

      unaFormacion puedeMoverse ifFalse:
            [self agregarLocomotoraUtilSiHayA: unaFormacion]

Deposito >> agregarLocomotoraUtilSiHayA: unaFormacion
      |locomotora|
      locomotora := self locomotoraUtilPara: unaFormacion.
      locomotora ifNotNil: [unaFormacion agregarLocomotora: locomotora]

Deposito >> locomotoraUtilPara: unaFormacion
      ^self locomotoras detect: [:unaLocomotora |
                  unaLocomotora arrastreUtil >=
                  unaFormacion cuantoLeFaltaParaMoverse]
            ifNone: [nil].
```

La realidad es que “No se hace nada” es algo que no suele pasar, es muy raro que se de una situación así. Pensemos lo siguiente: la necesidad es que todas las formaciones puedan moverse.

Ajustemos un poquito el enunciado para ver un caso más habitual:

Se cambia el requerimiento... Ahora:

- *La locomotora encontrada debe pasar del depósito a la formación.*
- *Si no hay ninguna locomotora que cumpla esa condición entonces se deberá lanzar un error con la descripción: 'Falta locomotora para el armado de la formación'.*

```
Deposito >> agregarLocomotoraA: unaFormacion

      | locomotora |
      unaFormacion puedeMoverse ifFalse: [
            (locomotora := self locomotoraUtilPara: unaFormacion)
                  ifNil: [ self error: 'Falta locomotora para el armado de
            formación' ].
            self locomotoras remove: locomotora.
            unaFormacion agregarLocomotora: locomotora.
      ]

Deposito >> locomotoraUtilPara: unaFormacion

            ^self locomotoras detect: [:unaLocomotora |
                              unaLocomotora arrastreUtil >=
                              unaFormacion cuantoLeFaltaParaMoverse ]
                        ifNone: [nil].
```

¡¡Un momento cerebrito!!

¿Qué pasó ahí? ¿Qué diferencia encuentran entre la primera solución y la nueva?

El mensaje "error:" podríamos decir que es una de las explosiones que mencionábamos antes... una explosión que puede ser controlada. Cuando se envía ese mensaje, se corta el flujo de ejecución normal del programa. Es decir, a continuación no se ejecuta la siguiente línea de código, sino que pega un salto.

¿A dónde? Básicamente "a donde lo estén esperando"... o lo que es lo mismo, al encargado de atrapar a ese error. Si no definimos nosotros a un responsable, entonces "por omisión" el responsable va a ser el mismo Smalltalk que, como dijimos antes, va a frenar todo y nos va a mostrar la pantallita con el stack.

Mientras lo vamos digiriendo un poco, agregamos algo más:

10 - Agregamos al modelo a los supervisores, que son los encargados de controlar cada depósito; su trabajo es el de mantener organizadas las formaciones y locomotoras. También son responsables de emitir órdenes de compra de locomotoras según corresponda.

- *Hacer que el supervisor complete las formaciones que no pueden moverse con una locomotora cualquiera del depósito con el arrastre útil necesario para lograr que se mueva. En caso de no haber alguna, debe realizar una orden de compra registrando para qué formación es.*

Esto podemos hacerlo aproximadamente así:

```
Supervisor >> completarFormaciones

	deposito formacionesQueNoPuedenMoverse do:
		[ :unaFormacion | deposito agregarLocomotoraA: unaFormacion ]
```

Ahora, esto hace *una parte* de lo que dice arriba, ¿qué pasa si en el medio no se puede completar una formación? Sabemos que la ejecución se corta y además va a tirar un error por pantalla. No solo no estamos creando la orden de compra, sino que además las siguientes formaciones, las que venían después de la que lanzó el error, tampoco fueron completadas.

¿Cómo hacemos para que el error no nos explote en la cara?

Hasta ahora, lo único que logramos es hacerlo explotar con un mensaje más lindo y nada más... Para lograr atraparlo, disponemos de un mensaje que entienden los bloques (que como ya sabemos son objetos, y por lo tanto les podemos mandar mensajes, como por ejemplo value) y se llama `on:do:`.

```
[ "Código con un error potencial" ]
on: ClaseDeError
do: [ "Algo que se evalúa únicamente si ocurre el error" ]
```

¿Qué va a hacer?
Se va a ejecutar el código dentro del primer bloque, y cuando se produzca un error que coincida con la clase del error, va a ejecutar el código del bloque del `do:` .
Por ejemplo...

```
[ 5 / 0  ]
on: Error
do: [ "Código para informar de la división por cero" ]
```

<Esto se puede pasar por alto porque el bloque podría no recibir nada, pero no me parece correcto>
Este bloque recibe un parámetro, ¿se dan una idea de que puede ser ese parámetro?

```
[ 5 / 0.
  5 mensajeLoco.
]     on: Error
      do: [ "Algo se rompió, pero... ¿qué?" ]
```

¿Qué es lo que pasa acá? ¿Cuántos errores tengo? ¿Cómo sé qué error ocurrió?
Si, acá lo vemos seguro, rompe en la primera línea porque hay una división por cero... pero tranquilamente podría recibir ese bloque por parámetro. ¿Cómo hago para distinguir qué corresponde en ese caso?

```
[ 5 / 0.
  5 mensajeLoco.
]     on: Error
      do: [ :error | "Ahora tengo el objeto que representa al error en mi
                        poder, tengo que hacer algo con ese objeto" ]
```

Bien... ¿y qué puedo hacer con ese error?
Puedo, por ejemplo, pedirle su mensaje de descripción. Eso se puede hacer mediante el mensaje `messageText`.
Otra cosa que puedo hacer es no manejarlo, y "tirarlo" o "re-lanzarlo" para que lo agarre otro de más arriba, enviándole el mensaje `pass`. Claro que esto último, por sí solo, no tendría sentido. Debería hacer algo más en el medio: para volver a tirar el error sin más, ni siquiera lo manejo.

Entonces, ¿cómo queda nuestro código con la gestión del error incluida?

Quizá lo primero que pensemos sea algo como esto:

```
Supervisor >> completarFormaciones

      [ deposito formacionesSinLocomodoras do: [ :unaFormacion |
```

```
            deposito agregarLocomotoraA: unaFormacion ]
      ] on: Error do: [ :error | "No se rompe. . . . . ¿y ahora?" ]
```

Si pensaron en otra cosa, bien por ustedes. Esto no sirve, porque necesitamos la formación. Acá tenemos una única gestión de errores para todas las formaciones en conjunto.

Lo que en realidad nosotros necesitamos es gestionar individualmente la posibilidad de error en cada formación. Mejor cambiamos un poquito y ponemos la gestión del error dentro del bloque del do:, ya que ahí (dentro de ese bloque) estamos en un contexto donde sólo nos importa una única formación a la vez.

```
      Supervisor >> completarFormaciones

      deposito formacionesSinLocomodoras do: [ :unaFormacion |
            [ deposito agregarLocomotoraA: unaFormacion ]
                  on: Error do: [ :error |
                        self crearOrdenDeCompralocomotoraUtilPara: unaFormacion.]
      ]
```

Ahora cambiemos el código original de formacion a este:

```
Deposito >> agregarLocomotoraA: unaFormacion
      | locomotora |
      "Nótese que le falta una letra, porque quiero que se rompa"
      unaFormacion puedeMovers ifFalse: [
        (locomotora := self locomotoraUtilPara: unaFormacion)
        ifNil: [ self error: 'Falta locomotora para el armado de la formación' ].
      ]
      unaFormacion agregarLocomotora: locomotora.
      self locomotoras remove: locomotora.
```

¿Qué va a pasar ahora si ejecutamos el código del Supervisor? Vamos a mandar a comprar un montón de locomotoras :-D … wiiiiiiiiiiiiiiii ¡Locomotoras para todos y todas!

Pero eso no es lo que queremos que pase, ¿verdad?

Entonces, este error es muy lindo, pero no nos sirve, porque nos genera un Error que es algo muy genérico, ¿que se les ocurre?

Miren fijo un ratito lo que escribimos al presentar el mensaje on: do:...

```
["código con error potencial"] on: ClaseDeError do: ["Gestión del error"]
```

Claaaro, usar una clase distinta para poder gestionarla en forma diferenciada según el primer parámetro del mensaje. En particular, una clase propia de nuestro modelo sería una buena idea, así sabemos que va a ser tan específica o genérica como la necesitemos.

```
Deposito >> agregarLocomotoraA: unaFormacion

      | locomotora |
      unaFormacion puedeMoverse ifFalse: [
            (locomotora := self locomotoraUtilPara: unaFormacion)
                  ifNil: [ FaltaLocomotoraError signal: 'Falta locomotora
                  para el armado de formación' ].
            unaFormacion agregarLocomotora: locomotora.
            self locomotoras remove: locomotora.
      ]

Deposito >> locomotoraUtilPara: unaFormacion

      ^self locomotoras detect: [:unaLocomotora |
            unaLocomotora arrastreUtil >=
            unaFormacion cuantoLeFaltaParaMoverse ]
      ifNone: [ nil ].
```

Un par de cosas:

1. Ya tenemos una clase para nuestro error. La descripción bien podría ser parte de la clase, con lo cual solo le mandaríamos el mensaje `signal`, sin parámetros. Para cambiar la descripcion, redefinimos `messageText`.

2. Podríamos mandarle otra información en el signal, como podría ser para qué formación ocurrió; o la cantidad de vagones, lo que sea, es NUESTRO error y le ponemos la información que nos parezca necesaria.

Ahora cambiamos nuestro código en la clase Supervisor, para gestionar específicamente el error de la falta de locomotora, no cualquier error. Si alguien escribe mal el nombre de un método, no está tan mal que su programa explote. Pero si falta una locomotora que pueda hacer que una formación se mueva, entonces ahí si: "el supervisor sabe qué debe hacer".

```
Supervisor >> completarFormaciones

deposito formacionesSinLocomodoras do: [ :unaFormacion |
      [ deposito agregarLocomotoraA: unaFormacion ]
            on: FaltaLocomotoraError
            do: [ :error |
                  "En este ejemplo no usamos el error, pero bien podríamos"
                  self crearOrdenDeCompraLocomotoraUtilPara: unaFormacion.
            ]
]
```

De esta manera, ante cualquier **otro** error que ocurra, no lo vamos a manejar y lo atrapará quien deba.