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 ser cumplida 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`
  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 devuelva `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 situaciones por alto? ¿Qué pasa con este código?

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

Si la colección está vacía, ¿cuál es ese elemento? ¿Qué clase tiene? Acá se puede observar un posible ejemplo de 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.

Otro ejemplo es la mencionada división por cero... ¿qué podría devolver? ¿Un número muy grande? ¿Un número muy chico? ¿Cero? Esos son claramente valores válidos en una división. `nil` tampoco es una opción, pero por un motivo distinto... el emisor del mensaje espera un número como resultado, y si le doy el objeto indefinido `nil` seguramente no va a saber qué hacer.

Los anteriores ejemplos son de “consultas” en las que se espera una respuesta, pero este tipo de situaciones no tiene porqué estar acotada a eso. El caso del mensaje que no se entiende bien puede

ser una orden o una consulta, y el del `remove:` es una orden directamente. En estos casos, corresponde “avisarle a alguien” que lo que se pidió no se pudo hacer.

¿Qué es lo que pasa con estos casos?
Para lidiar con estas situaciones, muchos lenguajes tienen mecanismos especiales que permiten “alterar el flujo de ejecución normal del programa”. Un poquito más adelante vamos a explicar qué queremos decir con eso. En el caso de Smalltalk, el mencionado mecanismo se encuentra implementado mediante **excepciones**. En particular, en este curso nos vamos a concentrar en un subconjunto particular de las excepciones de Smalltalk: los **errores**.

Como vimos, entre los casos raros hay varios conocidos. Lo que hay que tener en mente es que todos los errores son **atrapados** por alguien. ¿Y qué significa que sean atrapados? Que hay alguien que detecta la existencia de ese error y lo gestiona, hace algo al respecto. Si nosotros no asignamos un responsable explícitamente, 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, normalmente, nadie quiere que su programa explote.

Yyyyy... ¿Entonces?

Veamos esto con un ejemplo tomado del ejercicio de la clase pasada de trenes y depósitos. 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.*
- *Sino, 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.*

Y nos quedó un código de este estilo[1]:

```
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 dé una situación así. Ajustemos un poquito el enunciado para ver algo más habitual:

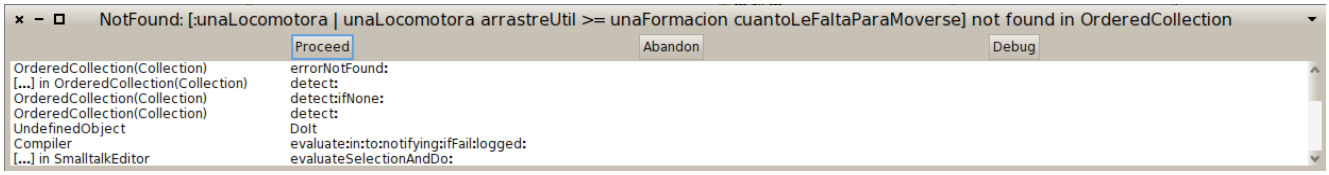
Se cambia el requerimiento... Ahora:

- *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.*

```
Deposito >> agregarLocomotoraA: unaFormacion

      | locomotora |
      unaFormacion puedeMoverse ifFalse: [
            locomotora := self locomotoraUtilPara: unaFormacion.
            unaFormacion agregarLocomotora: locomotora.
            self locomotoras remove: locomotora ]

Deposito >> locomotoraUtilPara: unaFormacion

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

¿Qué pasa cuando `detect:` no encuentra un objeto que cumpla la condición? Se obtiene un error, que es exactamente lo que nos piden. El problema es que obtenemos algo como esto:

[1] Puede variar ligeramente año a año, no siempre queda igualito :)



Claramente, no es el mensaje más feliz del mundo. Mejor va a ser encargarnos nosotros mismos de elegir el mensaje. Ajustamos nuevamente el requerimiento:

- *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'.*

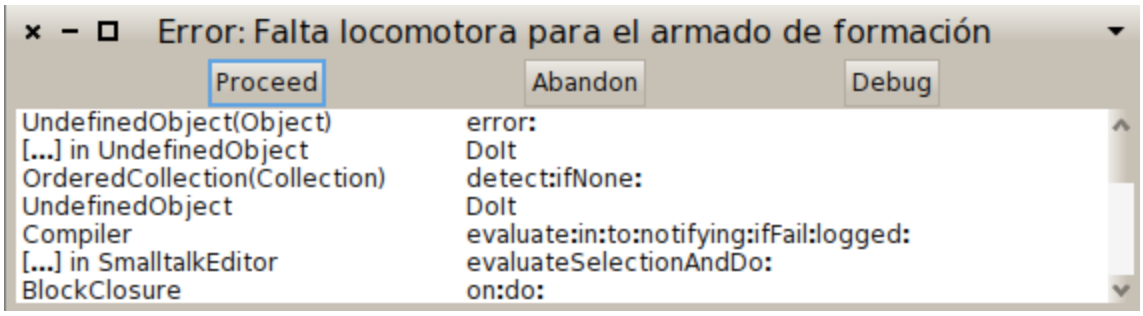
Para lograr eso, vamos a dejar de usar el `detect:` y vamos a volver a usar `detect:ifNone:`.

```
Deposito >> locomotoraUtilPara: unaFormacion

      ^self locomotoras
            detect: [:unaLocomotora |
                  unaLocomotora arrastreUtil >=
                  unaFormacion cuantoLeFaltaParaMoverse ]
            ifNone: [Error signal:
                  'Falta locomotora para el armado de formación']
```

¡¡Un momento cerebrito!!

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

El mensaje `signal:` enviado a la clase `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, como se mencionaba anteriormente. 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 código encargado de atrapar a ese error. Si no definimos una porción de código específicamente responsable de esto, entonces "por omisión" el responsable va a ser el mismo ambiente Smalltalk que, como dijimos antes, va a frenar todo y nos va a mostrar la pantallita con el stack, pero con un mensaje personalizado:



Mientras lo vamos digiriendo un poco, agregamos otro requerimiento 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 de un depósito (cada supervisor tiene un único depósito asignado). También son responsables de emitir órdenes de compra de locomotoras cuando haga falta.

- *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.*

Podemos plantear una primera aproximación de la siguiente forma:

```
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 pedazos de código a los cuales les podemos mandar mensajes como `value`) y se llama `on:do:`.

```
[ "Código con un error potencial" ]
      on: ClaseDeError
      do: [ "Algo que sucede únicamente cuando 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 o cualquier subclase de la misma, va a ejecutar el código del bloque del `do:` .
Por ejemplo...

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

Acá podemos usar 2 tipos de bloques: Sin parámetros, o con un parámetro. Para el caso del bloque que 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`. Algo un poco más avanzado sería enviárselo a otro objeto que lleve un registro de los errores, por ejemplo.

Otra cosa que puedo hacer es propagarlo, "lanzarlo" nuevamente para que lo agarre otro método de más arriba, enviándole `pass`. Claro que eso, 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, lo dejo pasar solito. ¿Qué ganamos con atrapar el error si sólo lo vuelvo a lanzar? El resultado obtenido es **exactamente** el

mismo que al no atraparlo, por lo tanto estoy agregando código sin ningún beneficio. Es igual a lo que sucede cuando, en una redefinición de un método, únicamente hago algo como esto:

```
selector
      ^ super selector.
```

En el código anterior, le estamos diciendo explícitamente que se use la definición de la superclase... ¿Qué pasa si eso no está? Nada... el method lookup hace su trabajo y se usa la definición de la superclase, es decir, hacemos **lo mismo** que con ese código, pero sin necesidad de codificar para obtenerlo. Bueno, con el uso de `pass` sin ningún otro agregado ocurre lo mismo:

```
[ ... ] on: Error do: [ :error | error pass ]
```

Estamos indicando que queremos propagar ese `error` que acabamos de atrapar... lo cual es lo mismo que ocurre si no atrapamos el error en absoluto.

Ahora volvamos a los ejemplos correctos en lugar de divagar sobre lo que no hay que hacer, ¿cómo queda nuestro código con la gestión del error incluida?

Quizás, sólo quizás, lo primero que pensemos sea esto:

```
Supervisor >> completarFormaciones

      [ deposito formacionesSinLocomotoras do:
            [ :unaFormacion | deposito agregarLocomotoraA: unaFormacion ]
      ]     on: Error
            do: [ ". . . . . ¿y ahora?" ]
```

Pero esto no nos sirve, por varios motivos. Por una parte, porque necesitamos conocer qué formación es la que tuvo problemas. Acá tenemos una única gestión de errores para todas las formaciones en conjunto. Por otra parte, si acá obtenemos un error se corta la iteración del `do:` y las demás formaciones que se iban a tratar después no llegan a considerarse. 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.

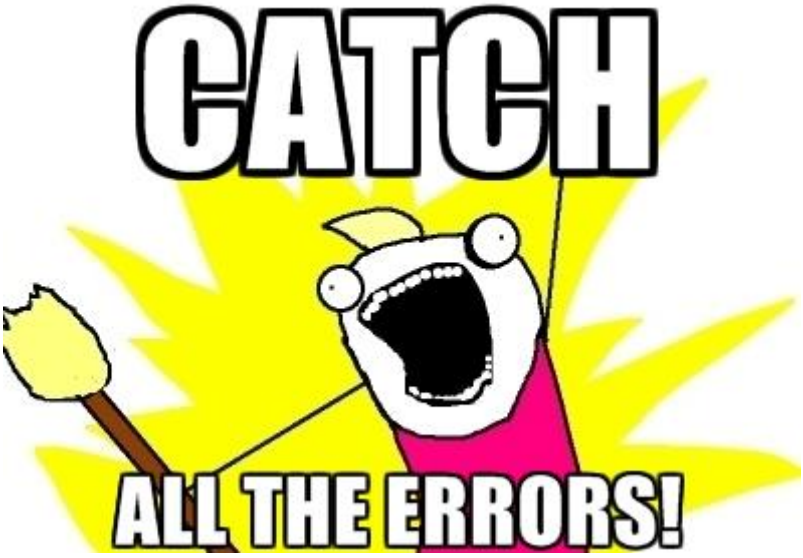
```
Supervisor >> completarFormaciones

      deposito formacionesSinLocomotoras
            do: [ :unaFormacion | self completarFormacion: unaFormacion ]

Supervisor >> completarFormacion: unaFormacion

      [deposito agregarLocomotoraA: unaFormacion]
```

```
            on: Error
            do: [ self crearOrdenDeCompraLocomotoraUtilPara: unaFormacion]
```

Ahora cambiemos el código original del depósito por este otro:

```
Deposito >> agregarLocomotoraA: unaFormacion
      | locomotora |
      "Nótese que le falta una letra, porque quiero que se rompa"
      unaFormacion puedeMovers ifFalse: [
            locomotora := self locomotoraUtilPara: unaFormacion.
            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 mensaje `Error signal:` es muy lindo, pero no nos sirve, porque nos genera un `Error` que es algo muy genérico y no lo podemos diferenciar de otros errores... ¿qué 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. Un error, sí, pero no cualquier clase de error. Un error más específico, es

decir, una subclase de `Error`. En particular, una clase propia[2] de nuestro modelo sería una buena idea, así sabemos que va a ser tan específica o genérica como la necesitemos y no se va a confundir con excepciones de "clases ajenas". Entonces, volvemos a la versión correcta de `agregarLocomotaA:`.

```
Deposito >> agregarLocomotoraA: unaFormacion

      | locomotora |
      unaFormacion puedeMoverse ifFalse: [
            locomotora := self locomotoraUtilPara: unaFormacion.
            unaFormacion agregarLocomotora: locomotora.
            self locomotoras remove: locomotora.
      ]
```

Y en lugar de lanzar un error genérico, cambiamos por un error de una clase propia:

```
Deposito >> locomotoraUtilPara: unaFormacion

^ self locomotoras
      detect: [ :unaLocomotora |
            unaLocomotora arrastreUtil >=
            unaFormacion cuantoLeFaltaParaMoverse ]
      ifNone: [ FaltaLocomotoraError signal:
            'Falta locomotora para el armado de formación'].
```

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 parámetro. Para cambiar la descripción, sólo necesitamos redefinir `messageText` y listo.
2. Podríamos mandarle otra información en el signal, como ser para qué formación ocurrió, o la cantidad de vagones, o lo que sea: es NUESTRO error y le ponemos la información que nos parezca necesaria.

Ahora volvemos a nuestro código en la clase `Supervisor` y lo modificamos para gestionar específicamente el error de la falta de locomotora. De esta forma podemos evitar confundir cualquier otro error que no sea de nuestro dominio. Si alguien escribe mal el nombre de un método, no está tan mal que su programa explote porque no se atrapó el error. Pero si falta una locomotora que pueda hacer que una formación se mueva, entonces ahí sí: "el supervisor sabe qué debe hacer".

```
Supervisor >> completarFormaciones

      deposito formacionesSinLocomodoras do: [ :unaFormacion |
```

[2] Nosotros vamos a generar una subclase de `Error`. Existen otras formas, pero se van del alcance del curso.

```
            self completarFormacion: unaFormacion
      ]

Supervisor >> completarFormacion: unaFormacion

      [ deposito agregarLocomotoraA: unaFormacion ]
            on: FaltaLocomotoraError
            do: [ :error |
                  self crearOrdenDeCompraLocomotoraUtilPara: unaFormacion ]
```

De esta manera, cualquier otro error que ocurra va a ser propagado, es decir que no lo vamos a estar gestionando y lo atrapará quien deba hacerlo. Nosotros nos vamos a ocupar únicamente de atrapar lo que queremos.

Notas finales

- Recordar que las *excepciones* deben usarse para casos *excepcionales*. No debemos tratar de manejar todo el flujo de ejecución del programa mediante excepciones.

- Si vamos a atrapar una excepción, es porque tenemos que hacer algo en esa situación. No podemos “absorberla” y no hacer nada. Si la situación no necesita un trato especial, entonces probablemente no debería manejarse por medio de excepciones. Atrapar una excepción y no hacer nada no tiene sentido.