

# Capítulo 1

## Colecciones

Las colecciones son un concepto importante y poderoso al diseñar con objetos. En este capítulo veremos cómo se modela el conocimiento de un conjunto de referencias de un objeto y su utilización para resolver un problema concreto.

Agradezco a Victoria Pocládova, Carlos Lombardi, Leonardo Volinier y Jorge Silva por el artículo “Colecciones en Smalltalk” del sitio web

<http://pdep.com.ar/material/apuntes>

que en conjunto con el material que he preparado convergió en el presente apunte.

Fernando Dodino



# Índice general

<b>1. Colecciones</b>	<b>1</b>
1.1. Introducción . . . . .	3
1.1.1. ¿Qué es una colección? . . . . .	3
1.1.2. Representación de colecciones . . . . .	4
1.2. Interfaz de una colección . . . . .	6
1.3. Un ejemplo concreto . . . . .	6
1.3.1. Clases a crear - versión 0 . . . . .	7
1.3.2. Iniciamos un Playground . . . . .	7
1.3.3. Conocer el tamaño . . . . .	10
1.3.4. Saber si tiene elementos . . . . .	11
1.3.5. Clases a crear - versión 1 . . . . .	11
1.3.6. Agregando nuevas misiones . . . . .	14
1.3.7. Misiones abiertas y cumplidas . . . . .	14
1.3.8. Objetos bloque y declaratividad . . . . .	16
1.3.9. Transformar los elementos . . . . .	18
1.3.10. e . . . . .	18

## 1.1. Introducción

### 1.1.1. ¿Qué es una colección?

La colección nos permite representar un conjunto de objetos relacionados: los jugadores de un equipo de fútbol, un cardumen de peces, las cosas que un héroe guarda en su mochila, un ejército, son ejemplos de este tipo de abstracciones.

Otra definición posible es que una colección nos sirve para modelar una relación 1 a N:

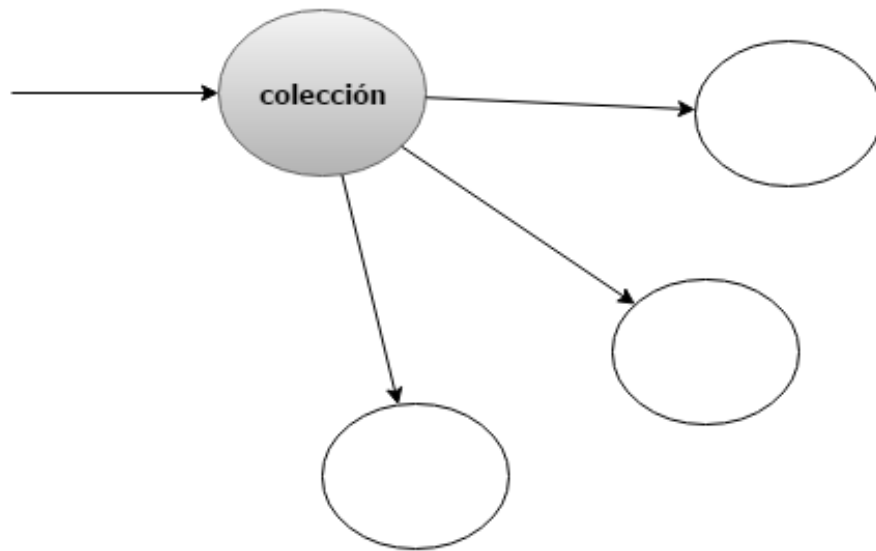


Figura 1.1: La colección es un conjunto de referencias a otros objetos

- Una factura tiene muchas líneas con productos
- Un escritor publicó varios libros
- Una fiesta tiene muchos invitados
- Un héroe tiene que cumplir varias misiones

A primera vista una colección es un conjunto de objetos. Si la vemos con más precisión nos damos cuenta que es más preciso pensarla como un conjunto de referencias: los elementos no están adentro de la colección, sino que la colección los conoce.

### 1.1.2. Representación de colecciones

Podemos graficar la relación dinámica entre un equipo de fútbol y los jugadores que lo integran mediante un diagrama de objetos. Este es un diagrama con características *dinámicas*, porque muestra el estado de los objetos en un momento determinado.

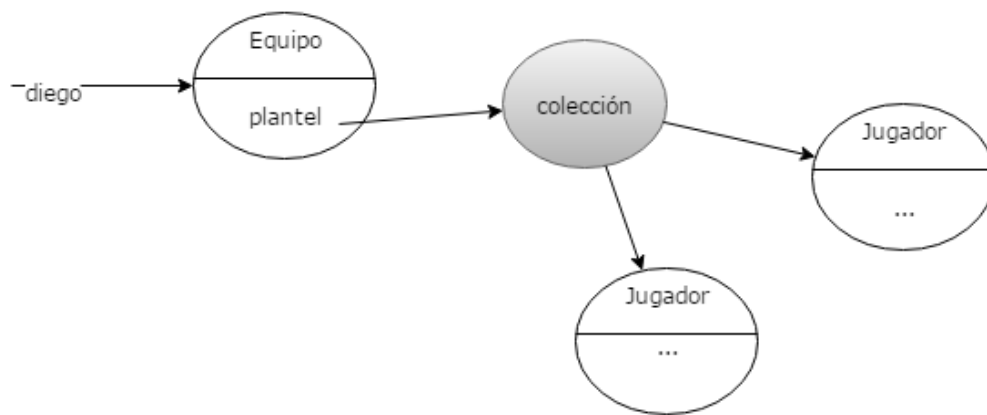


Figura 1.2: El plantel de jugadores de un equipo como una colección de objetos

También podemos generar un diagrama de clases en UML de la misma relación:

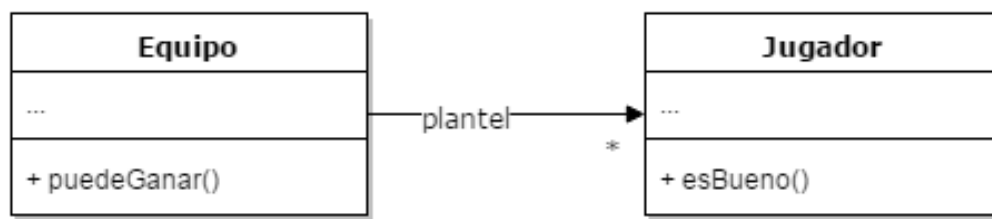


Figura 1.3: El plantel de jugadores, visto al nivel de clases

Este es un diagrama con características *estáticas*, porque no depende de un caso particular sino que muestra las relaciones entre las clases.

¿Qué es lo que cuenta el diagrama? Que un equipo **tiene** jugadores, el conector marca una relación de **asociación**: hay un atributo plantel en la clase Equipo (el nombre del atributo se marca en uno de los extremos de la asociación). El asterisco (\*) muestra la multiplicidad: un equipo tiene muchos jugadores. La dirección marca qué objeto conoce a los otros: como la flecha va de Equipo a Jugador sabemos que cada equipo tiene n jugadores, no conocemos qué características tiene la relación de Jugador a Equipo, pero se pueden dar dos opciones:

- un jugador pertenece a un solo equipo, en ese caso la relación Equipo-Jugador es de **uno a muchos**
- un jugador participa en una relación con varios equipos (por ejemplo,

porque nos interesa saber en qué equipos jugó). En ese caso la relación es de **muchos a muchos**

## 1.2. Interfaz de una colección

Supongamos que tenemos un álbum de fotos, otra representación posible de una colección de objetos. ¿Qué podemos hacer con esas fotografías?

- Mirarlas, “recorrerlas”: iterar una colección
- Averiguar cuántas fotos hay: saber su longitud
- Saber si está una determinada foto en el álbum: saber si un elemento pertenece a la colección
- Pegar una foto nueva: agregar un elemento a la colección
- Regalar una foto a alguien: eliminar un elemento de la colección
- Buscar qué fotos son de Ushuaia: filtrar/seleccionar elementos de una colección
- Anotar las personas que salieron en mis fotos: transformar los elementos de una colección
- Saber si hay alguna foto de Navidad: determinar si alguno/todos los elementos satisfacen un criterio

En el último requerimiento aparece también la idea de conjunto vacío. En general podemos asociar la noción matemática de conjunto a la colección, aunque sabemos que el conjunto matemático no tiene orden, ni se “recorre”, mientras que en la colección eso depende de la intención que nosotros tengamos, como veremos más adelante.

## 1.3. Un ejemplo concreto

En ciertos casilleros el héroe puede encontrar misiones y nuevos objetivos. Por ejemplo, un mago puede encargarle buscar un ítem mágico en una montaña lejana. Un anciano puede encargarle liberar a su hija de los terribles trolls que habitan en la gruta de los sin nariz. Cada vez que un héroe tiene uno de

estos encuentros, él anota los datos de la misión en su diario personal. Cada vez que una misión es superada, el héroe la marca como “cumplida”. Toda misión suma en el camino del héroe: las misiones tienen una recompensa de oro, y de respeto.

¿Qué abstracciones surgen? El héroe ahora tiene una colección de misiones. En principio vamos a pensar en dos tipos de misiones: 1) buscar un ítem mágico, 2) liberar a una doncella. Las misiones deben tener una recompensa (más adelante podemos modelar unidades de oro o de respeto para ello), un solicitante y el estado, que puede ser pendiente o cumplida.

Además nos avisan que existen misiones difíciles, que son aquellas que tienen más de 2 meses de iniciada, y además

- si la montaña donde está el ítem a buscar queda a más de 100 kms. o bien
- si la doncella a liberar está custodiada por más de 4 trolls

### 1.3.1. Clases a crear - versión 0

Vamos a crear lo mínimo necesario para poder meternos de lleno en el ejemplo de las colecciones. Ahora el héroe tendrá una colección de misiones:

```
Object subclass: #Héroe
  instanceVariableNames: 'misiones ...'
```

También vamos a crear una misión posible: `BuscarItemMagico`, por el momento sin atributos, porque nos queremos seguir concentrando en la interfaz y no en la implementación, es decir, qué me ofrece el objeto y no cómo lo resuelve.

Para poder agregar una misión, vamos a definir un método explícito:

```
#Héroe
agregarMision: unaMision
  misiones add: unaMision
```

### 1.3.2. Iniciamos un Playground

Abrimos un Workspace de trabajo o Playground y vamos a inicializar un juego de variables nuevo:

```
diego := Héroe new
  agregarMision: (BuscarItemMagico new)
```

Al intentar enviar el mensaje agregarMision: recibimos el primer error, no existe la clase BuscarItemMagico, entonces lo creamos:

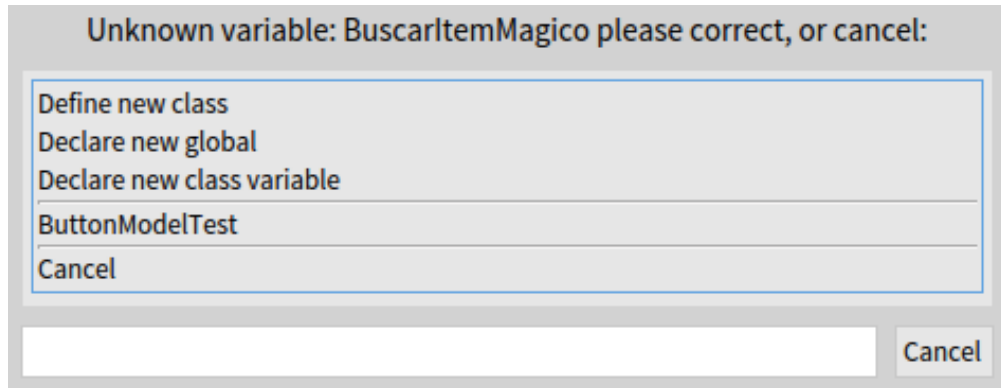


Figura 1.4: Creando una clase a demanda

Elegimos la opción “Define new class” sin preocuparnos todavía por los atributos

```
Object subclass: #BuscarItemMagico
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Kernel-Objects'
```

Una vez resuelta la creación de la clase, aparece un nuevo error, seleccionamos la opción Debug...

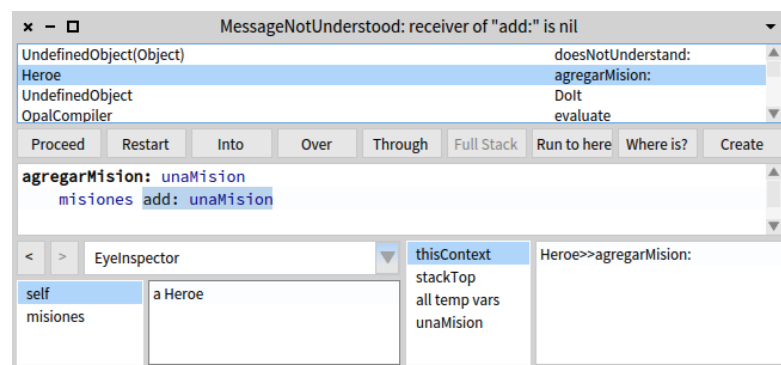


Figura 1.5: La pantalla de *debugging* muestra que el error ocurre al enviar el mensaje add: a misiones

Esto se da porque la referencia a misiones quedó en nil.



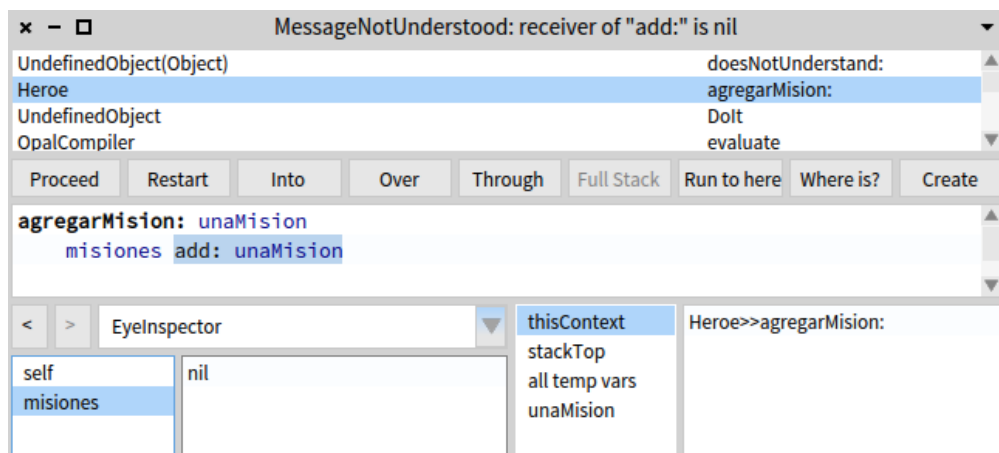


Figura 1.6: misiones es una referencia a *nil*, ¡falta inicializarla!

Entonces debemos inicializar a diego cuando creamos el guerrero, esto lo podemos hacer manualmente o con la opción Analyze >Generate initialize method

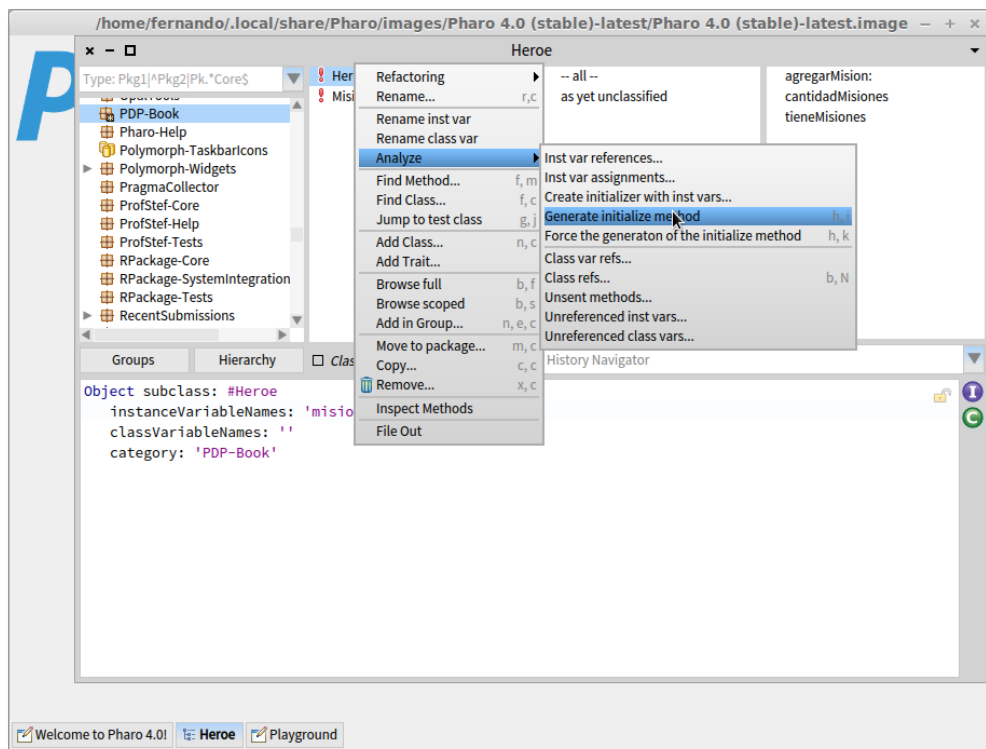


Figura 1.7: Generando un método initialize a través del IDE

Comenzaremos usando un Set como colección, esto presupone que no nos importa el orden en el que almacenamos las misiones y que no hay elementos duplicados: puede haber muchas liberaciones de doncellas, pero cada una representa una misión distinta. El Set es la implementación más equivalente al concepto matemático de conjunto que presentamos anteriormente.

Ahora sí nuestro método nos queda

```
#Heroe
initialize
  super initialize.
  misiones := Set new.
```

Al grabarlo volvemos al Playground y ejecutamos nuevamente el código mediante Do It

```
diego := Heroe new
agregarMision: (BuscarItemMagico new)
```

Vemos que el mensaje tuvo efecto inspeccionando la referencia diego: escribimos diego y luego Inspect It:

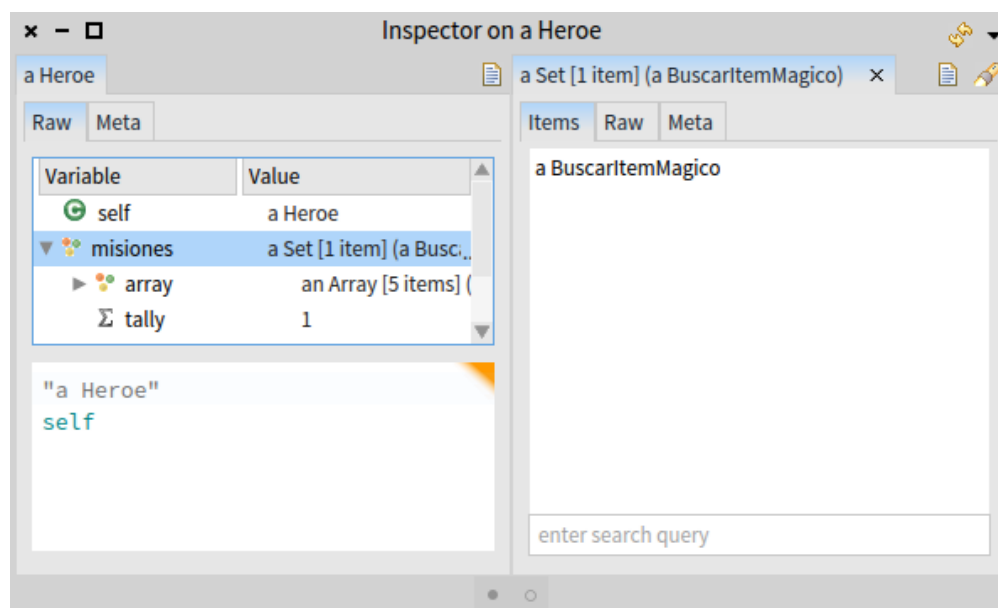


Figura 1.8: diego es un héroe y tiene una referencia en la colección misiones

### 1.3.3. Conocer el tamaño

¿Cómo sabemos cuántas misiones tiene un héroe?

```
#Héroe
cantidadMisiones
  ^misiones size
```

Y lo probamos, sabiendo que nos importa lo que va a devolver porque no es un método que tenga efecto, sino que devuelve información, entonces elegimos la opción *Print It*:

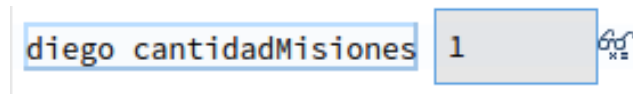


Figura 1.9: diego tiene por el momento una sola misión

### 1.3.4. Saber si tiene elementos

Queremos saber si un héroe tiene misiones...

```
#Héroe
// Opcion 1
tieneMisiones
  ^misiones notEmpty
```

```
// Opcion 2
tieneMisiones
  ^misiones size > 0
```

Ambas opciones parecen similares, de todas maneras la primera opción es más *expresiva*. En la segunda opción hay una traducción implícita: `size > 0`... ah, es si tiene elementos. Es un detalle, pero un detalle que implica tiempo que se pierde cada vez que vaya a leer la implementación de este método.

Lo probamos...

```
diego tieneMisiones
```

### 1.3.5. Clases a crear - versión 1

Creamos la clase *Mision*, con atributos solicitante, recompensa, fecha de inicio y fecha de cumplimiento. Para cada uno de ellos definiremos los accessors correspondientes, haciendo botón derecho sobre la clase *Mision* ¿Refactoring ¿Inst Var Refactoring ¿Accessors

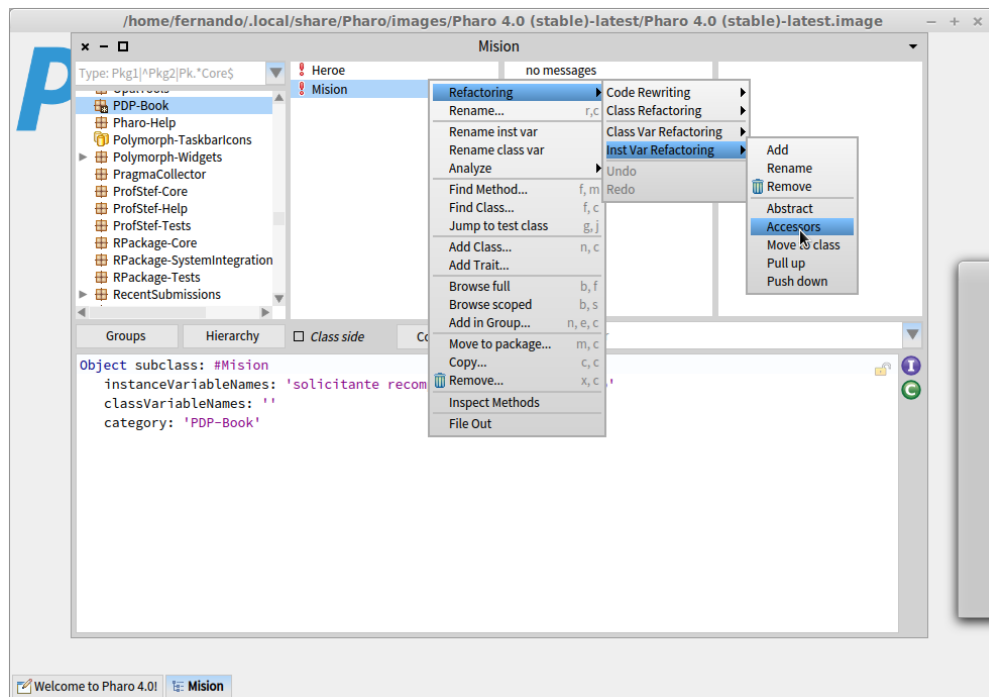


Figura 1.10: El plantel de jugadores de un equipo como una colección de objetos

Dos subclases heredarán de Mision:

- BuscarItemMagico, necesitamos el atributo distanciaMontania
- LiberarDoncella, del cual necesitamos el atributo trollsSeguridad

Cambiaremos dinámicamente la superclase de BuscarItemMagico...

```
Mision subclass: #BuscarItemMagico
  instanceVariableNames: 'distanciaMontania'
  classVariableNames: ''
  category: 'PDP-Book'
```

Damos ok al mensaje de advertencia y ahora tenemos a BuscarItemMagico como subclase de Mision.

Ahora definimos LiberarDoncella:

```
Mision subclass: #LiberarDoncella
  instanceVariableNames: 'trollsSeguridad'
  classVariableNames: ''
```

```
category: 'PDP-Book'
```

Una vez definidos los accessors, nos preguntamos: ¿qué comportamiento tiene una misión? Debe decirnos si es difícil. Sabemos que esto depende de algo general (más de 2 meses de iniciada una misión) y se especializa en cada subclase. Lo general... se codifica en la clase Misión:

```
#Mision
esDificil
  ^(self estaIniciadaHace: 60)
```

Claro, pero además debemos dejar que la misión delegue el comportamiento en cada implementación

```
#Mision
esDificil
  ^(self estaIniciadaHace: 60) && (self realmenteEsDificil)
```

Antes de resolver el método realmenteEsDificil, dejamos una posible implementación de estaIniciadaHace:

```
#Mision
estaIniciadaHace: xDias
  ^(self estaAbierta) && ((Date new subtractDate: fechaInicio) >
    xDias)

estaAbierta
  ^fechaCumplimiento isNil
```

Ahora sí el método realmenteEsDificil depende de este requerimiento:

- si la montaña donde está el ítem a buscar queda a más de 100 kms. o bien
- si la doncella a liberar está custodiada por más de 4 trolls

```
#BuscarItemMagico
realmenteEsDificil
  ^distanciaMontania > 100

#LiberarDoncella
realmenteEsDificil
  ^trollsSeguridad > 4
```

Por motivos didácticos no vamos a explicar cómo se prueba la funcionalidad recientemente incorporada, pero sabemos que este paso es fundamental para no tener inconvenientes con los pasos que vamos a hacer a continuación.

### 1.3.6. Agregando nuevas misiones

Vamos a asociar nuevas misiones a diego:

```
diego := Heroe new
agregarMision: (BuscarItemMagico new
  distanciaMontania: 1000;
  solicitante: 'Mago de Oz';
  recompensa: 2000;
  fechaInicio: (Date newDay: 2 month: 2 year: 2004);
  yourself);
agregarMision: (LiberarDoncella new
  trollsSeguridad: 3;
  solicitante: 'Old man';
  recompensa: 10000;
  fechaInicio: (Date yesterday);
  yourself);
agregarMision: (BuscarItemMagico new
  distanciaMontania: 30;
  solicitante: 'Mago Cacarulo';
  recompensa: 500;
  fechaInicio: (Date yesterday);
  yourself);
agregarMision: (LiberarDoncella new
  trollsSeguridad: 7;
  solicitante: 'Old man';
  recompensa: 12000;
  fechaInicio: (Date yesterday);
  yourself).
```

### 1.3.7. Misiones abiertas y cumplidas

Queremos saber ahora qué misiones están abiertas.

- El héroe conoce a sus misiones
- pero cada misión debe determinar si está abierta o no (es su **responsabilidad**).

```
#Heroe
misionesAbiertas
^misiones select: [ :mision | mision estaAbierta ]
```

Redefinamos el printOn: de Misión para reflejar un poco más de información sobre el objeto:

```
#Mision
printOn: aStream
  aStream nextPutAll: self descripcion;
  nextPutAll: ' pedido para ';
  nextPutAll: self solicitante.
```

Redefinimos el método descripción para cada subclase:

```
#BuscarItemMagico
descripcion
  ^'Buscar item magico'

#LiberarDoncella
descripcion
  ^'Liberar doncella'
```

Probamos entonces las misiones abiertas:

```
diego misionesAbiertas
  a Set(Liberar doncella pedido para Old man
        Liberar doncella pedido para Old man
        Buscar item magico pedido para Mago de Oz
        Buscar item magico pedido para Mago Cacarulo)
```

Claro, todas las misiones están abiertas. ¿Cómo cumplimos todas las misiones?

Tenemos el mensaje menos simpático para enseñar, el do:

```
#Heroe
cumplirMisiones
  self misionesAbiertas do: [ :mision | mision cumplir ]

#Mision
cumplir
  fechaCumplimiento := Date new
```

En el Playground le pedimos a diego que cumpla todas sus misiones pendientes:

```
diego cumplirMisiones
```

Y ahora veamos si efectivamente el método tuvo efecto:

```
diego misionesAbiertas
  a Set()
```

### 1.3.8. Objetos bloque y declaratividad

¿Por qué dijimos que el método `do:` es el menos simpático? Porque si bien ignoramos de qué manera se recorre la colección tenemos la posibilidad de hacer algo con cada elemento de la colección, por ejemplo

- ignorar determinados elementos
- o transformar los elementos de la colección
- o utilizar variables de diferente alcance: variables internas al bloque `do:`, variables locales en el mismo método, o variables globales, como las variables de instancia, o las de clase

A lo largo del libro intentaremos pensar nuestras soluciones utilizando bloques de más alto nivel, que aumenten la **declaratividad**, esto es no pensar tanto en el algoritmo, o cómo se resuelve, sino en lo que nosotros queremos lograr (el qué). Para eso utilizaremos objetos **closures** que modelan bloques de código (los que se encierran entre corchetes) y la rica interfaz de colecciones que provee Smalltalk.

¿Cómo se modela un bloque de código? Muy fácil:

```
factorialDe5 := [ 5 factorial ].
```

Esto no ejecuta el factorial de 5, sino que crea un objeto que sabe calcularlo. Para evaluarlo efectivamente, debemos pedirselo:

```
factorialDe5 value  
120
```

También podemos crear objetos bloque donde le pasemos parámetros:

```
factorial := [ :numero | numero factorial ].
```

Pero ahora ya no podemos evaluarlo con `value`:

```
factorial value
```

Esto nos muestra el mensaje de error:



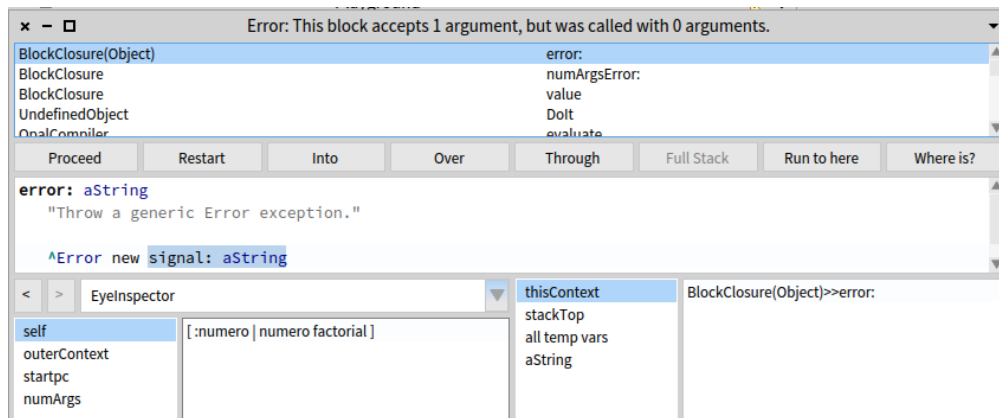


Figura 1.11: El bloque espera un parámetro y no se lo pasamos

Ahora si enviamos el valor, podemos calcular el factorial correspondiente:

```
factorial value: 5
```

Tener un objeto bloque nos permite modelar, por ejemplo, un criterio de búsqueda ad-hoc:

```
misionesConRecompensaMayorA: unValor
| criterio |
criterio := [ :mision | mision recompensa > unValor ].
^misiones select: criterio
```

Esto nos permite probar:

```
diego misionesConRecompensaMayorA: 2000
```

El criterio es un bloque que recibe un parámetro llamado mision (que es cualquier objeto que entienda el mensaje recompensa) y devuelve un valor booleano. Lo podemos probar en un workspace:

```
[ :mision | mision recompensa > 2000 ] value: (LiberarDoncella new
recompensa: 5000)
```

Cuántos objetos intervienen:

- un objeto bloque que espera un parámetro, y entiende el mensaje value:
- otro LiberarDoncella

Tener al bloque de código como abstracción me permite separar dos momentos: cuando creo al objeto y le digo lo que tiene que hacer, y cuando efecti-

vamente lo ejecuta (enviándole el mensaje `value`, `value:` o `value:value:` en base a la cantidad de parámetros que tenga).

### 1.3.9. Transformar los elementos

Nos piden determinar quiénes son los solicitantes de las misiones de un héroe. Lo que sabemos es que cada héroe tiene  $n$  misiones, y que cada misión tiene un solicitante, pero a su vez un solicitante puede pedir varias misiones (la relación Solicitante-Misión es  $n$  a 1, o *many-to-one*).

Entonces debemos

- transformar cada misión en un solicitante
- y luego la lista de solicitantes debemos pasarla a un Set, para eliminar los duplicados.

Vemos la resolución:

```
solicitantes
^(misiones collect: [ :mision | mision solicitante ]) asSet
```

Y lo probamos

```
diego solicitantes
a Set('Mago Cacarulo' 'Old man' 'Mago de Oz')
```

### 1.3.10. e

\* Interfaz de las colecciones siguiendo el ejemplo x\* Conocer el tamaño x\* Saber si tiene elementos x\* Agregar un elemento ?\* Sacar un elemento ?\* Buscar un elemento x\* Filtrar elementos que cumplan un criterio x\* Transformar los elementos de una colección generando otra colección \* Totalizar valores que almacenan objetos de una colección, reducir una colección a un valores \* Operatorias con conjuntos: includes/contains, union, intersection. \* Saber si todos/algún elemento cumple una condición \* Bloque de código y Declaratividad \* Iteradores externos e internos \* Tipos de colecciones. Estáticas vs. dinámicas. Ordenadas y sin ordenar. Con índices. Colecciones estáticas (Ejemplos de cada una de ellas. Formas de agregar elementos.) \* Array \* String \* Interval

Colecciones dinámicas \* Bag? \* Set \* List/Ordered Collection \* SortedCollection \* Map/Dictionary \* Comparativa general