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 Pocladova, 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

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:

- Una factura tiene muchas líneas con productos
- Un escritor publicó varios libros
- Una fiesta tiene muchos invitados

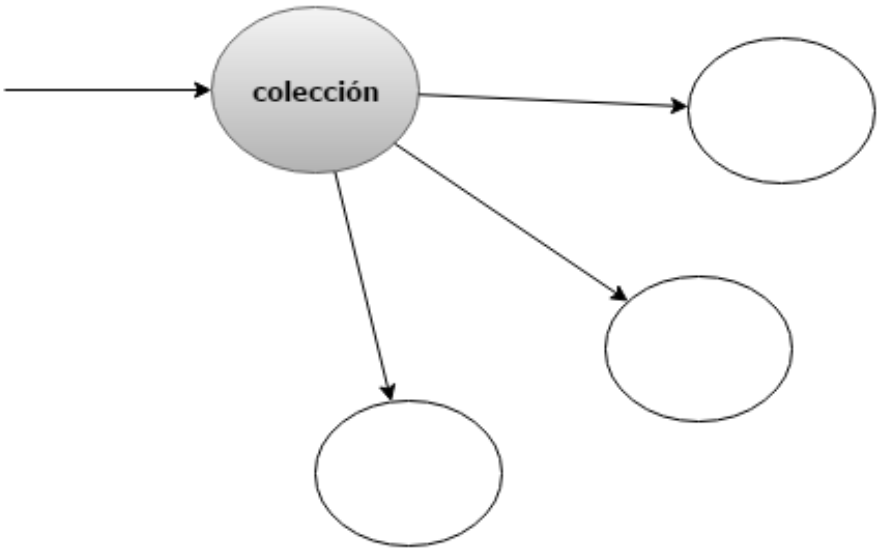

Figura 1.1: La colección es un conjunto de referencias a otros objetos

- 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.

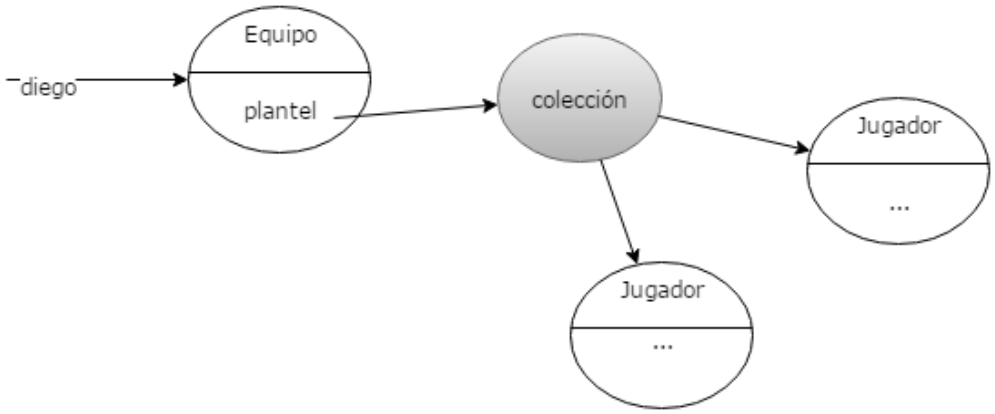

Figura 1.2: El plantel de jugadores de un equipo como una colección de objetos

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

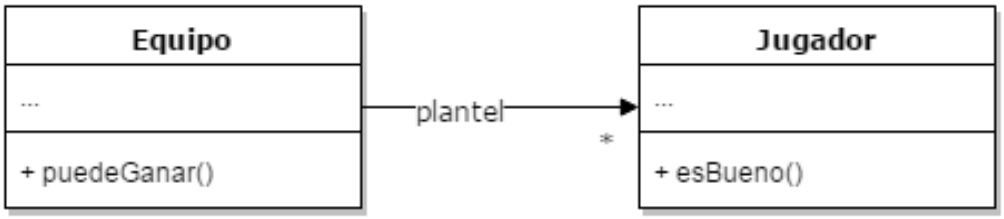

Figura 1.3: El plantel de jugadores, visto al nivel de clases

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 album 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 album: 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 en las que el encargado es un ser justo, 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: #Heroe
   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:

```
#Heroe
agregarMision: unaMision
  misiones add: unaMision
```

1.3.2. Iniciamos un Playground

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

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

Al intentar enviar el mensaje agregarMision: a diego se produce este error

Figura 1.4: 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.

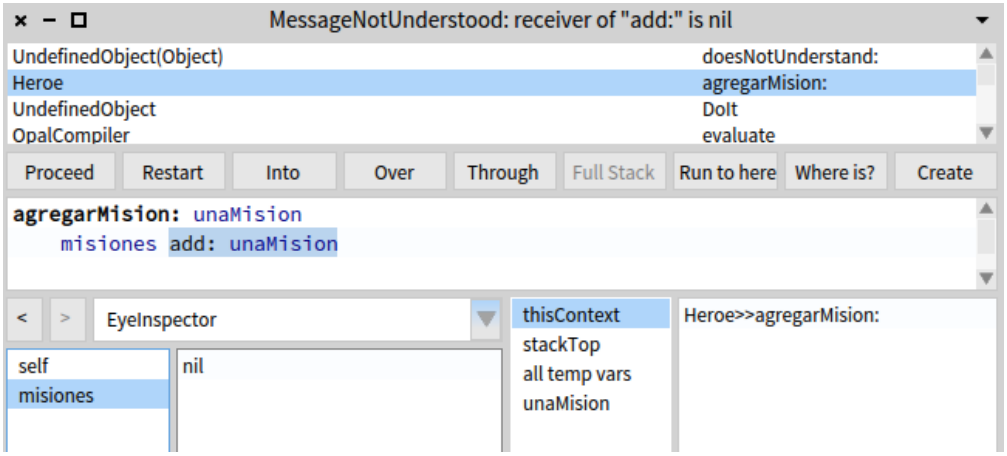

Figura 1.5: misiones es una referencia a nil, ¡falta inicializarla!

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

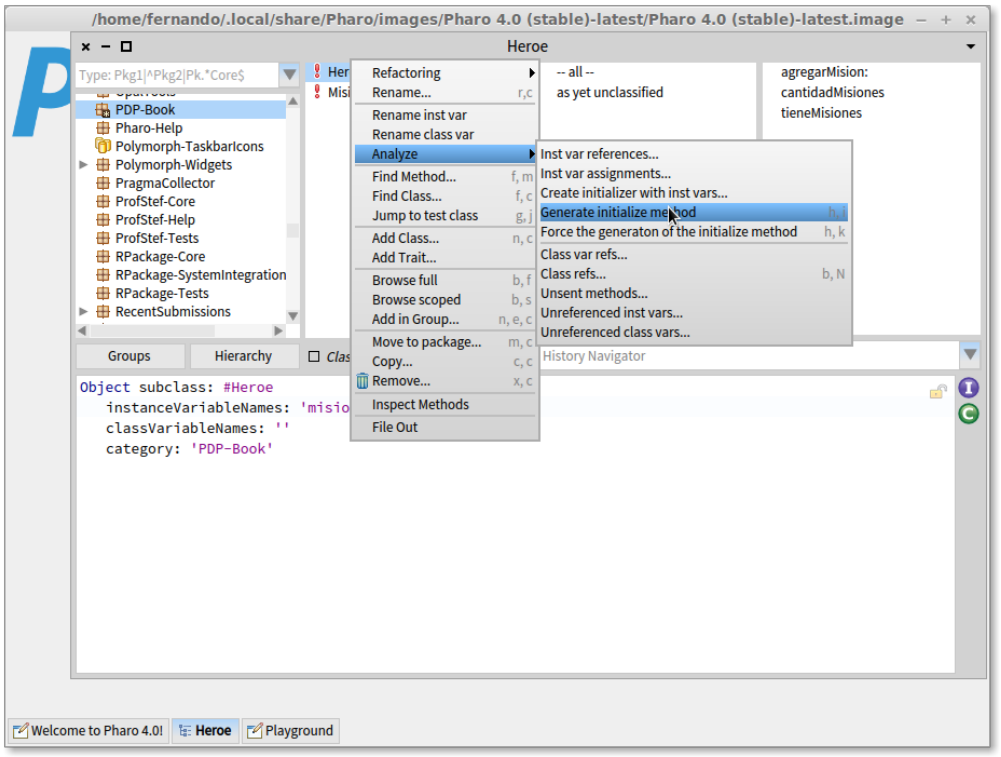

Figura 1.6: Generando un método initialize a través del IDE

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.
```

y 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:

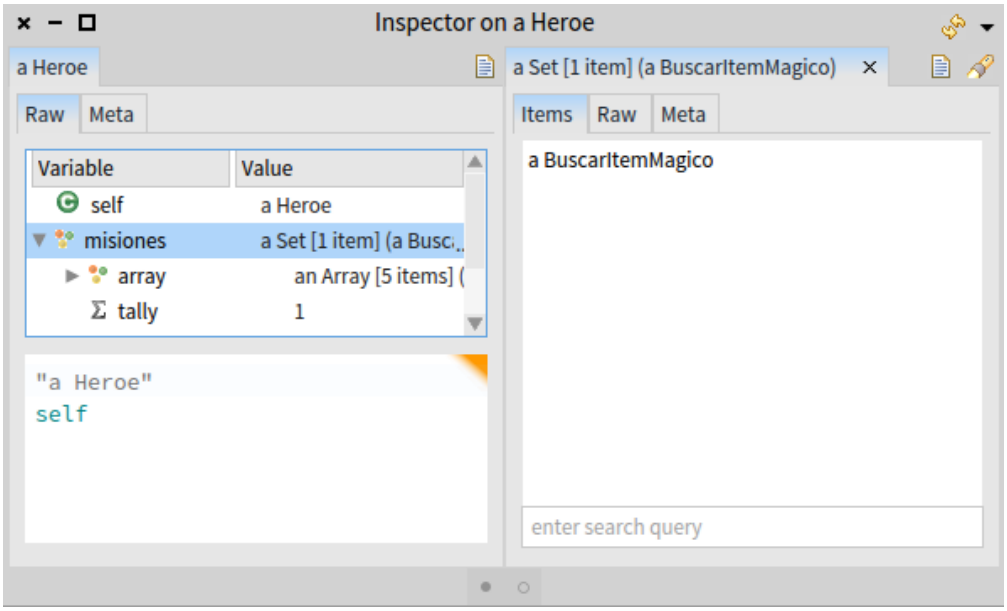

Figura 1.7: diego es un héroe y tiene una referencia en la colección misiones

1.3.3. Conocer el tamaño

¿Cómo sabemos cuántas misiones tiene un héroe?

```
#Heroe
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*:

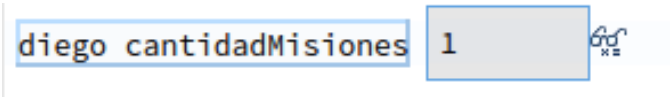

Figura 1.8: diego tiene por el momento una sola misión

1.3.4. Saber si tiene elementos

Queremos saber si un héroe tiene misiones...

```
// Heroe
```

```
// 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 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 Dos subclases heredarán de Mision:

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

* Interfaz de las colecciones siguiendo el ejemplo * Conocer el tamaño * Agregar un elemento * Sacar un elemento * Saber si tiene elementos * Buscar un elemento * Filtrar elementos que cumplan un criterio * 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

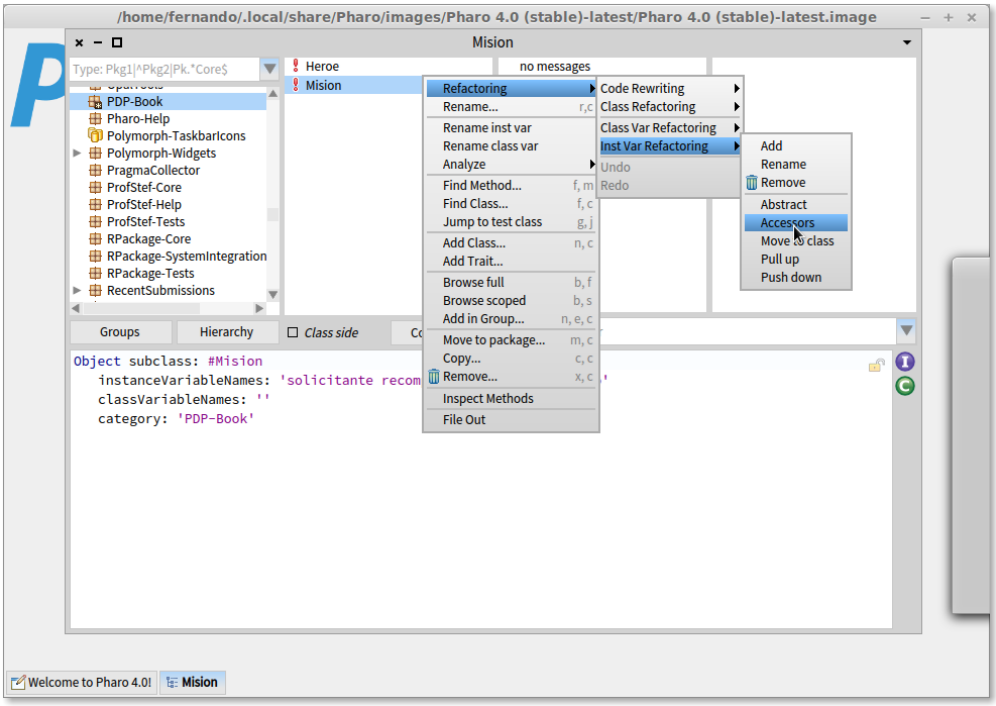

Figura 1.9: El plantel de jugadores de un equipo como una colección de objetos