Capítulo 1

Herencia

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.

Índice general

1.1. Introducción

En capítulos precedentes introdujimos el concepto de *clase*: un molde que define la estructura y comportamiento de los objetos creados a partir de ella. Las clases nos ofrecen tanto un marco conceptual como ventajas prácticas: por un lado nos permiten agrupar objetos similares bajo una misma abstracción, dándoles un nombre común (el nombre de la clase), y por otro lado nos permiten definir una única vez el comportamiento de un objeto y reutilizarlo en varios objetos. Por ejemplo, podemos podemos definir una clase `Heroe` que permita desplazar nuestros héroes entre distintos casilleros de la siguiente manera:

```
Object subclass: #Heroe
   instanceVariableNames: ''.

Heroe >> desplazarseA: unCasillero
    unCasillero recibirHeroe: self.
```

Consideremos ahora que queremos introducir distintas *especies* de héroes: arqueros y guerreros. Arqueros y guerreros son héroes que van a participar en distintas batallas épicas y por lo tanto deben poder hacer daño. Además, tanto arqueros como guerreros poseen, obviamente, distintas características de combate que podemos representar en código utilizando distintas clases `Arquero` y `Guerrero`:

```
Object subclass: #Arquero
   instanceVariableNames: 'flecha'.

Arquero >> danoCausado
    ^ flecha poder + self modificadorDestreza.

Object subclass: #Guerrero
   instanceVariableNames: 'arma'.

Guerrero >> danoCausado
    ^ arma poder + self modificadorFuerza.
```

Usar distintas clases para guerreros y arqueros nos permite definir distintos comportamientos para cada uno de ellos. Sin embargo, nuestros nuevos héroes ya no comparten el comportamiento que definimos anteriormente en nuestra clase `Heroe`. Para solucionar este problema, este capítulo introduce el concepto de **herencia**. La herencia es una relación entre clases que permite definir super-clasificaciones y sub-clasificaciones de objetos, generar distintos niveles de abstracción y compartir comportamiento entre distintas clases. Estos distintos niveles de abstracción llevan a la aparición de clases concretas y clases abstractas. Finalmente este capítulo discute distintos criterios de aplicación de herencia.

1.2. Clasificando Objetos con Herencia

Previamente estudiamos cómo las clases nos permiten clasificar objetos similares. Todos los objetos de la misma clase tienen comportamiento y estructura similar. Por ejemplo, Diego y Jorge son dos guerreros que poseen un arma y atacan de la misma manera. De la misma manera podemos encontrar otras clasificaciones de objetos que tienen comportamientos y/o estructuras distintas como son los arqueros en nuestro ejemplo.

Pese a que arqueros y guerreros pertenecen claramente a distintas clases de objetos dado que exhiben distinto comportamiento, ambos tienen también algunas características en común ya que ambos conforman nuestro grupo de héroes. La Figura 1.1 muestra como se relacionan nuestros distintos objetos (Diego, Jorge y Sergio) utilizando un diagrama de Venn. En la figura vemos

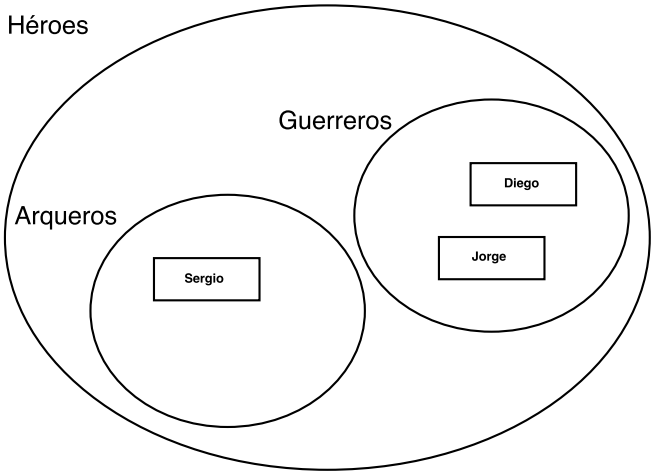

Figura 1.1: Diagrama de Venn presentando las ideas de super-clasificación y sub-clasificaciones en nuestro ejemplo de héroes, guerreros y arqueros.

que Diego y Jorge está incluidos dentro del conjunto de guerreros y Sergio está incluido en el conjunto de arqueros. Ambos conjuntos están incluidos en un conjunto más grande y más abstracto: el conjunto de héroes. Si consideramos que los conjuntos distintos conjuntos definen clases en nuestro programa, entonces decimos que la clase `Guerrero` son subclases de la clase `Héroe`. De manera más general podemos definir esta relación de la siguiente manera:

Definición 1 ***Superclase.*** *Una clase* $C1$ *es superclase de una clase* $C2$ *si* $C2$ *define un subconjunto de* $C1$*.*

De manera recíproca, podemos definir la relación de subclase:

Definición 2 ***Subclase.*** *Una clase* $C1$ *es subclase de* $C2$ *si* $C2$ *es superclase de* $C1$*.*

1.2.1. Herencia

Para representar la relación de sub y superclasificación entre clases usamos *herencia*. La herencia es una relación entre dos clases donde una clase es superclase de la otra. Decimos que la subclase *hereda* la estructura y el comportamiento de la superclase, permitiéndonos combinar el comportamiento definido en distintas clases. En nuestro ejemplo, podemos decir que un guerrero (*i.e.,* una instancia de la clase `Guerrero`) exhibe el comportamiento de un guerrero (*i.e.,* el comportamiento definido en su clase) y el comportamiento

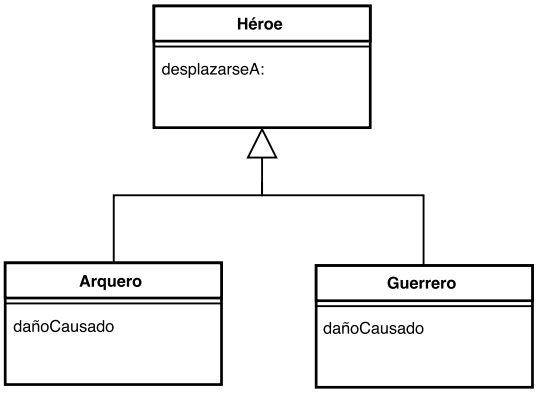

Figura 1.2: Diagrama de clases representando la relación de herencia entre las clases `Héroe`, `Guerreros` y `Arquero`.

de un héroe (*i.e.,* el comportamiento definido en la superclase de su clase). Más aún, al separar el comportamiento de los héroes en la clase `Héroe`, podemos compartirlo entre distintas clases de héroes.

Podemos ilustrar la relación de herencia en un diagrama de clases tal y como aparece en Figura 1.2: una flecha con punta cerrada blanca que se dirige desde una subclase hacia una superclase. **Guille** ►*quizás es mejor mostrar un diagrama con una sola clase/subclase*◄ Para efectuar la relación de herencia entre dos clases en Pharo, debemos modificar la definición de la clase como sigue, indicando que la clase `Heroe` posee las subclases `Arquero` y `Guerrero`.

```
Heroe subclass: #Arquero
   instanceVariableNames: 'flecha'.

Heroe subclass: #Guerrero
   instanceVariableNames: 'arma'.
```

1.2.2. Herencia de comportamiento

method lookup nueva version

1.2.3. Herencia de atributos

1.2.4. `self` y `super`

method lookup revisited 2 self vs super

1.3. Clases concretas vs clases abstractas

Al generar superclases sigo perdiendo información, gano en generalidad. Un Ave quizás no tenga sentido instanciarlo. Si en mi aplicación no voy a instanciar aves (porque representan un concepto demasiado general) entonces la clase es abstracta. Diferencia entre Smalltalk y Java:

1) En Smalltalk no tiene sentido crear un ave, aunque podría. La clase es abstracta cuando no tengo intención de crear instancias de esa clase (porque no tiene sentido).

2) En Java no puedo crear un ave aunque quisiera. La clase es abstracta y el compilador me impide generar instancias de esa clase. Son dos filosofías distintas. public abstract class Ave forma parte de la definición misma de clase

ejemplo de herencia con superclase abstracta ejemplo de herencia con superclase concreta

1.4. Criterios para utilizar herencia

1.4.1. Usar Objetos vs Clases

1.4.2. Especializar vs Generalizar

Otra forma de ver la herencia: subclasificamos un concepto conocido, lo refinamos. Si mi hija no conoce lo que es una tonina, yo le puedo explicar: "Y... es como un delfín pero negro" (una especie de ... pero que ...; mostrando tanto en lo que se parece como en lo que se diferencia).

Tenemos conceptos conocidos: golondrina, colibrí, torcaza, paloma, gorrión. Muchos de estos pájaros tendrán cosas en común pero se diferencian en algo... Una vez que reconocí varios objetos, puedo abstraer una clase. Pierdo información, porque lo que obtengo es más general (una golondrina genérica, en lugar de esta o aquella golondrina, que era de color azul y yo llamaba pepita). Ahora vamos a trabajar con otro tipo de abstracción: tengo varias clases (conceptos) y llego a una jerarquía de clases de la más general a las más particulares:

Otra forma de ver la herencia: subclasificamos un concepto conocido, lo refinamos. Si mi hija no conoce lo que es una tonina, yo le puedo explicar: "Y... es como un delfín pero negro" (una especie de ... pero que ...; mostrando tanto en lo que se parece como en lo que se diferencia).

1.4.3. subclase vs subtipo

1.4.4. Herencia vs composición