Práctico 16 – Ejemplo y Guía de Resolución

Polimorfismo – Binding Dinámico

Objetivo del Ejercicio

Que los alumnos puedan aplicar los conceptos vistos en la teoría de polimorfismo y binding dinámico a la resolución de un ejercicio durante la práctica. Se espera guiar a los alumnos en la correcta comprensión de cómo funcionan los llamados a métodos y cómo los mismos pueden ser resueltos en tiempo de ejecución.
Asimismo, se espera mostrar cómo el polimorfismo y el binding dinámico permiten materializar un diseño orientado a objetos.

Enunciado

Se desea modelar un juego compuesto por héroes y villanos. Cada personaje del juego posee un nombre de Super Héroe y un valor de fuerza. Adicionalmente, el juego debe proveer un mecanismo de agrupamiento de los personajes en ligas para realizar enfrentamientos entre grupos de personajes. Cada liga puede estar compuesta tanto de personajes como de otras ligas. Cada liga tiene su propio nombre de liga y un valor de fuerza. La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman. Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje como de una liga.

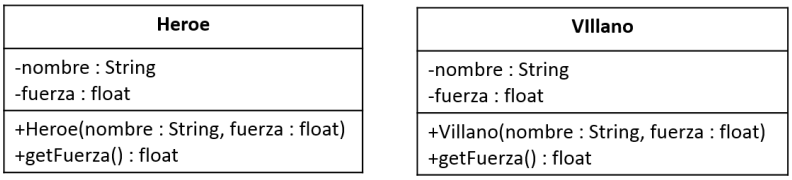
Preguntas para Guiar la Resolución del Ejercicio

1. Analizar el enunciado intentando detectar las diferentes clases intervinientes. Diferenciar instancias de clases. Por ejemplo, ¿Es necesario contar con dos clases una para los héroes y otra para los villanos? ¿Por qué?
2. Al armar la liga, la misma puede contener otras ligas o personajes, ¿Es correcto que la liga entonces tenga dos atributos uno que agrupe personajes y otro que agrupe ligas?
3. ¿Cómo podría modificarse el diseño para unificar el "contenido" de la liga? (Herencia)
4. ¿Qué mecanismo hace posible utilizar la herencia para unificar tipos? (Polimorfismo) ¿Qué tipo de referencia es aquella que permite referenciar instancias de más de una clase? (Referencia polimórfica)
5. ¿Qué sucede si se declara una variable de tipo "Enfrentable"? ¿Cómo resuelve dicha variable la invocación al método getFuerza()? (Binding Dinámico)
6. ¿Qué mecanismo hace posible que la implementación del método a ejecutar varíe de acuerdo al tipo de la instancia que lo invoca?

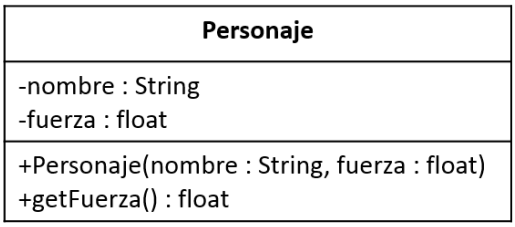
Resolución

"Se desea modelar un juego compuesto por héroes y villanos. Cada personaje del juego posee un nombre de Super Héroe y un valor de fuerza. […] Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje […]."

De acuerdo a ese fragmento del enunciado pareciera necesaria la creación de dos clases, una que represente a los héroes (Héroe) y otra que represente a los villanos (Villano).

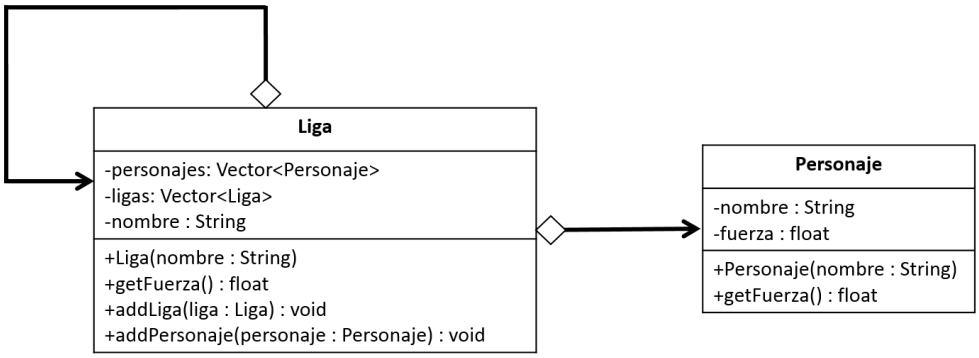


Pero, observando ambas clases, ¿Es realmente necesario? ¿Tienen dichas clases algún elemento que haga necesaria su distinción? ¿Algún atributo particular? ¿Algún método especial? No. Por lo tanto, ambas clases pueden reemplazarse por una única clase "Personaje".



"Adicionalmente, el juego debe proveer un mecanismo de agrupamiento de los personajes en ligas para realizar enfrentamientos entre grupos de personajes. Cada liga puede estar compuesta tanto de personajes como de otras ligas. Cada liga tiene su propio nombre de liga y un valor de fuerza."

De acuerdo al fragmento habría que crear una nueva clase que pueda contener tanto ligas como personajes, lo que, de acuerdo a este diseño obliga a la existencia de dos atributos, uno por cada tipo, y de dos métodos que permitan agregar elementos a la liga.



Asimismo el enunciado establece funcionalidad para la liga:

"Se debe proveer funcionalidad que permita retornar la fuerza tanto de un personaje como de una liga."
"La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman."

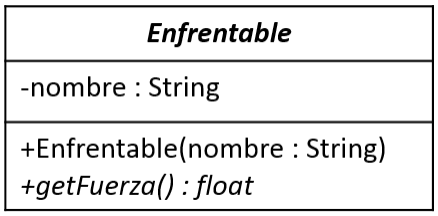
Entonces, considerado este diseño de clases, ¿Cómo quedaría la implementación de dicho método?

```
public float getFuerza(){
	float sum = 0;

	for(Liga l : ligas)
		sum += l.getFuerza();

	for(Personaje p : personajes)
		sum += p.getFuerza();

	return sum/(float)(ligas.size()+personajes.size());
}
```

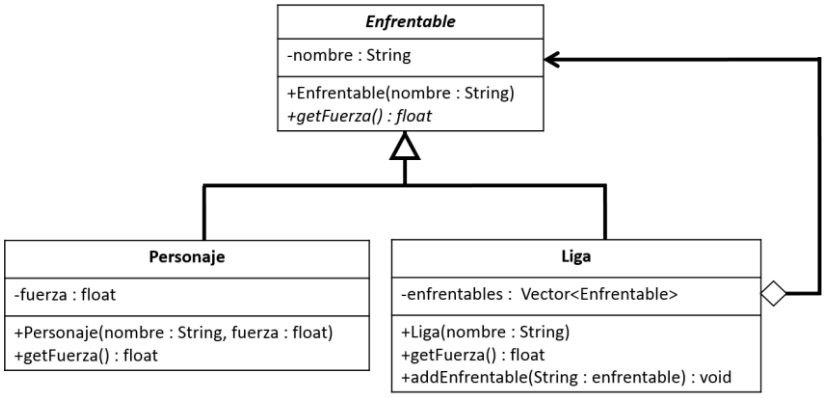
Nótese que debido a la existencia de dos atributos específicos para las ligas y los personajes surge la necesidad de diferenciar en el código entre la obtención de la fuerza tanto por parte de los personajes como de las ligas.

¿Es correcto el diseño de clases?
¿Es correcto que la clase diferencie en su composición entre las ligas y los personajes?
¿Es correcto que el código tenga que diferenciar entre los elementos de las distintas clases?

¿Hay algún otro diseño que permita el tratamiento uniforme de los elementos? Si, es posible modificar el diseño. Incluso puede observarse que hay atributos compartidos. En lugar de hacer que Liga distinga entre ligas y personajes, se puede crear una super-clase ("Enfrentable") de la que hereden tanto Liga como Personaje. El atributo común pasa al padre de la jerarquía, quien también define la interfaz de los métodos comunes.



Entonces, ahora ¿Cómo se conforma la clase Liga? De Liga y Personaje o se puede conformar de Enfrentable? Con este nuevo diseño, la clase Liga puede conformarse con objetos de tipo Enfrentable. Y Enfrentable, ¿Qué puede ser? Puede ser tanto una Liga como un Personaje. Recuérdese que declarando la variable de tipo padre, puede instanciarse dicha variable en cualquiera de los tipos (concretos) de sus hijos.



¿Qué mecanismo es el que permite que una variable pueda cambiar su tipo instanciado en tiempo de ejecución? Polimorfismo. Recordar que polimorfismo es un concepto teórico por el cual un único nombre puede denotar objetos de distintas clases que se encuentran relacionadas por una super-clase común. De esta forma, cualquier objeto denotado por este nombre es capaz de responder a un conjunto común de operaciones. En este caso, al declarar una variable de tipo Enfrentable, se estaría creando una referencia polimórfica, es decir, que puede referenciar a instancias de más de una clase. Recuérdese que no es posible la instanciación de tipos de clase abstractos.

Nótese que con el nuevo diseño ahora se tiene un único atributo de tipo Enfrentable y un único método que permite agregar los enfrentables. Permite agregar ligas y personajes? Si! Gracias al mecanismo de polimorfismo.

Por ejemplo, es posible tener las siguientes declaraciones:

```
Enfrentable batman= new Personaje("Batman");

Enfrentable ligaJusticia = new Liga("Liga de la Justicia");
```

Si no existiese el atributo común, la relación entre las clases seguiría siendo de Herencia? No. En el caso en el que la clase padre solo defina el conjunto de operaciones que las clases hijas deban implementar, deja de ser una clase abstracta para convertirse en una interfaz, con lo que la relación para a ser una realización.

¿Qué pasa con la implementación del método getFuerza()? ¿Es la misma para ambas clases? ¿Puede dicha implementación encontrarse en el padre de ja jerarquía? De acuerdo al enunciado:

"La fuerza de la liga se determina como el promedio de la fuerza de cada uno de los personajes y/o ligas que lo conforman."

Entonces, de acuerdo al enunciado, la implementación el getFuerza() sería distinto para cada clase hija de la jerarquía. Pero, ¿Qué pasaría si se decidiese que la implementación del getFuerza() debe estar en Enfrentable? ¿Cómo sabe la clase Enfrentable cómo tiene que calcular la fuerza de acuerdo al tipo de objeto? En este caso, será necesario conocer el tipo de objeto al que se le pide la fuerza para optar por un fragmento de código o el otro. En este caso, como la implementación del método sería única, es necesario contemplar todas las posibles formas de calcular la fuerza de acuerdo a los diferentes tipos de Enfrentable. Es decir, el método quedaría implementado de la siguiente forma:

```
public float getFuerza(){

        if(this instanceof Personaje)
              return this.fuerza;
       else
              if(this instanceof Liga){
                    float sum = 0;
                    for(Enfrentable e : enfrentables)
                          sum += e.getFuerza();
                    return sum/(float)enfrentables.size();
              }
       return 0;
}
```

¿Pero sería correcto tener ese código? No! Se estarían desaprovechando todas las ventajas antes mencionadas tanto del polimorfismo como del binding dinámico. Por otra parte, se estaría reduciendo tanto la extensibilidad como la modificabilidad del diseño.

Entonces, la implementación del método tiene que ser distinta, lo que hace que cada clase hija de la jerarquía deba proveer su propia implementación del método. En este contexto, el padre de la jerarquía solo provee interfaz del método pero no su implementación, es decir, se trata de un método abstracto.

getFuerza() en Enfrentable:

```
public abstract float getFuerza();
```

getFuerza() en Personaje:

```
@Override
public float getFuerza(){
       return this.fuerza();
}
```

getFuerza() en Liga:

```
@Override
public float getFuerza(){
       float sum = 0;

       for(Enfrentable e : enfrentables)
              if(e instanceof Personaje)
                    sum += (Personaje) e.getFuerza();
              else
                     if(e instanceof Liga)
                           sum += (Liga) e.getFuerza();

        return sum/(float)enfrentables.size();
}
```

¿Es correcta la implementación? Ahora que tenemos el Enfrentable, no sabemos a ciencia cierta en que tipo está instanciado cada objeto, pero, ¿Es necesario saber de qué tipo son los objetos? No! Se proveen mecanismos que permiten invocar el método sin saber quién es el que lo invoca hasta el momento de dicha invocación. Entonces, si no es necesario identificar si se trata de un Personaje o una Liga puede cambiarse la implementación:

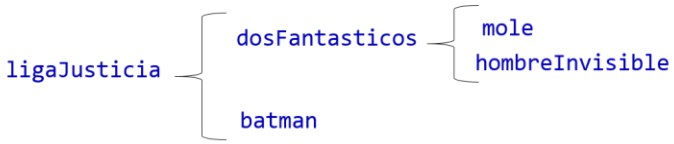
```
@Override
public float getFuerza(){
        float sum = 0;
        for(Enfrentable e : enfrentables)
                sum += e.getFuerza();
         return sum/(float)enfrentables.size();
}
```

Sin saber en qué tipo está instanciado el objeto se puede invocar el método getFuerza() ¿Cuál es el mecanismo que permite que la implementación del método cambie en tiempo de ejecución? Binding dinámico! El binding dinámico es el mecanismo que permite que el código de los métodos sea asociado a una instancia recién en el momento en el cuál esta recibe el mensaje para ejecutar dicho método.

Supóngase que ligaJusticia se encuentra compuesta de la siguiente forma:

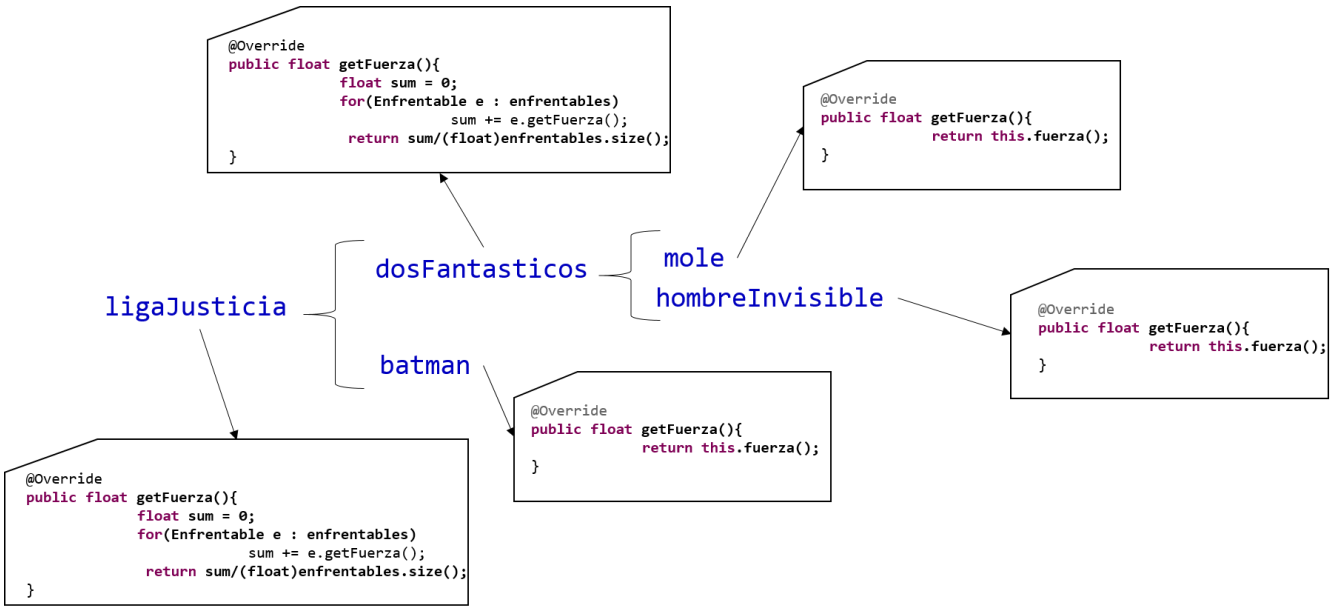
```
Enfrentable dosFantasticos = new Liga("Los Dos Fantásticos");

Enfrentable mole = new Personaje("Mole",4);

Enfrentable hombreInvisible = new Personaje("El Hombre Invisible",3);

dosFantasticos.addEnfrentable(mole);
dosFantasticos.addEnfrentable(hombreInvisible);


Enfrentable ligaJusticia = new Liga("Liga de la Justicia");

Enfrentable batman = new Personaje("Batman",10);

ligaJusticia.addEnfrentable(dosFantasticos);
ligaJusticia.addEnfrentable(batman);
```

Es decir, se tiene la siguiente estructura:



Y que se quiere obtener su valor de fuerza.

Se empieza el for, el primer elemento es "dosFantásticos", se encuentra declarada de tipo Enfrentable y hay que invocar su método de getFuerza(), para realizar dicha invocación importa de qué tipo es realmente dosFantásticos? No! ¿Qué código se va a invocar? ¿Se sabe al momento de escribir el código que método se invocará cuando se llame al getFuerza()? No! El binding es dinámico! Hasta no conocer en qué tipo se encuentra instanciada la variable, no es posible conocer que implementación del método será utilizada. Es decir, hasta la ejecución no se conocerá la implementación. Entonces en el caso de dosFantasticos se invocará la implementación de Liga. Que a su vez deberá recorrer sus elementos para hacer el cálculo de fuerza, invocando a los getFuerza() de mole y hombreInvisible. En ambos casos, se trata de elementos instanciados en Personaje, de modo que se invocará a la implementación de getFuerza() de Personaje. Lo mismo sucede con batman, está instanciado como Personaje, de modo que también se invoca a dicho código.



Cuando los llamados se resuelven: