Polimorfismo

La habilidad de un elemento del texto del software para denotar, en tiempo de ejecución, dos o más posibles tipos de objeto.

Con esa simple definición Meyer, en su libro Object Oriented Software Construction, nos plantea las bases para el **polimorfismo**.

Otra definición (en extremo minimalista) podemos encontrarla en la wiki de c2:

Misma interfaz, diferente implementación. Sustituibilidad.

La esencia del polimorfismo es esa, y es el pilar más fuerte de la Programación Orientada a Objetos. Si no lo tuviérmos, poco valor agregado tendría toda la ceremonia adicional necesaria para resolver un problema.

Nota: en tanto se nombre el *polimorfismo* en este capítulo, estaremos hablando implícitamente del *polimorfismo de tipos*. Siendo posible otros tipos de polimorfismo existe la necesidad de aclararlo en este punto.

Herencia y polimorfismo: la relación

La simple generación de una taxonomía de tipos puede servirnos para "acumular características", en tanto y en cuanto nuestras subclases tienen por añadidura todas la características de las superclases correspondientes. Sin embargo, habiendo otros mecanismos menos nocivos para la reutilización de código (como ser la composición), no hay suficientes razones para la existencia de esta relación jerárquica entre tipos.

Es por ello que la herencia (de tipos o de comportamiento) establece el mecanismo previo necesario para el polimorfismo: proporciona los primeros lineamientos de sustituibilidad de tipos. Gracias a éstos, enunciados muy claramente por Bárbara Liskov e inmortalizados en el Principio de Sustitución de Liskov, podemos afirmar que cualquier subtipo debe poder sustituír a sus supertipos.

Entidades

Definiremos como *entidad* a un nombre al cual varios valores pueden asignársele en tiempo de ejecución. Es una generalización de la noción tradicional de variable, como podemos leer en el libro de Meyer.

Respecto a las entidades entidades, podemos definir al polimorfismo como la capacidad de una entidad para tener adosados objetos de diversos posibles tipos. Dado que estamos utilizando un lenguaje con tipado estático, esta sustituibilidad estará dada por la herencia.

Bajo la luz del concepto de entidades podemos definir tres variantes claramente distintas del polimorfismo: *asignaciones polimórficas*, *entidades polimórficas* y *estructuras de datos polimórficas* (las tres definidas por Bertrand Meyer).

Asignaciones polimórficas

Una asignación será polimórfica si la variable objetivo y expresión de origen tienen diferentes tipos.

Esto es tan simple como decir que asignamos en una variable de un supertipo un objeto de un subtipo. El siguiente ejemplo lo dejará en claro:

```
class Persona {
  void saludar() { ... }
}

class Estudiante extends Persona {
  void darPresente() { ... }
}
```

Dada esa pequeña jerarquía, tenemos que:

```
Persona persona1 = new Persona();
Persona persona2 = new Estudiante();
Estudiante estudiante1 = new Estudiante();
// estas tres líneas son válidas
Estudiante estudiante2 = new Persona();
// esta línea no es válida, ya que no compila
```

La línea que nos llama la atención es `Persona persona2 = new Estudiante()`, la cual cumple con ser una asignación polimórfica: el objeto del tipo `Estudiante` puede adosarse a una entidad del tipo `Persona`, ya que un `Estudiante` es-una `Persona`.

Un análisis más profundo nos permite evaluar lo siguiente:

```
Persona persona = new Estudiante();
persona.saludar(); // es válido
persona.darPresente(); // no es válido
```

Y esto sucede porque *estamos considerando a la entidad como una Persona, no como un Estudiante, por lo tanto no podemos requerir ninguna de las responsabilidades que no sean de Persona*.

Regla de tipos del polimorfismo: Para que un acoplamiento polimórfico sea válido, *el tipo de orígen debe ser compatible con el tipo del destino*. En pocas palabras, a derecha de la asignación debe haber un tipo igual o más específico que el tipo de la izquierda.

Entidades polimórficas

Una entidad o expresión es polimórfica si, como resultado de asignaciones polimórficas, puede estar adosada a objetos de diferentes tipos en tiempo de ejecución.

En este caso no tenemos control directo sobre la entidad, sino que, por ejemplo, es un parámetro que recibe uno de nuestros métodos. Imaginemos la siguiente clase:

```
class Medico extends Persona {
```

```
  void curar(Persona persona) { ... }
}
```

Nuestros objetos del tipo `Medico` pueden recibir el mensaje `curar(persona:Persona)`, por lo que todas estas invocaciones serán válidas:

```
Medico drMario = new Medico();
drMario.curar(unaPersona);
drMario.curar(unEstudiante);
drMario.curar(otroMedico);
drMario.curar(drMario);
```

En ninguno de esos casos el método `curar(persona:Persona)` conocerá realmente el subtipo de `Persona` que recibirá en tiempo de ejecución. Sin embargo, esa información no es relevante: *un Medico sabe curar entidades del tipo Persona, por lo tanto conforme se invoque el método curar con una entidad que cumpla con la interfaz de Persona, podrá realizarlo*.

Estructuras de datos polimórficas

Un estructura de datos contenedora es polimórfica si puede contener referencias a objetos de diversos tipos.

Una lista en la que puede haber cualquier tipo de objetos es un desorden. Sin embargo, si acotamos de algún modo aquellos objetos que puede alojar tendremos una ventaja significativa.

Supongamos que tenemos el siguiente código:

```
class Congreso {
  List<Persona> asistentes = new LinkedList<Persona>();
  void agregarAsistente(Persona persona) {
    this.asistentes.add(persona);
  }
  void saludarATodos() {
    for (Persona cadaUno : this.asistentes) {
      cadaUno.saludar();
    }
  }
}
```

La clase `Congreso` representa un evento al cual pueden asistir todas `Persona`. El método `saludarATodos()` nos permite recorrer la lista de asistentes, pidiéndoles que saluden (una responsabilidad asumida por `Persona`) independientemente sean efectivamente del tipo `Persona`, `Estudiante` O `Medico`.

Esta posibilidad de tratar uniformemente a todos los elementos de una colección brinda gran potencia y flexibilidad.

¡Polimorfismo no es conversión!

A pesar de su nombre, "muchas formas", el polimorfismo no hará cambiar a los objetos para tomar una nueva forma (o tipo, en nuestra jerga) en tiempo de ejecución.

Importante: Los acoplamientos polimórficos sólo son aplicables a tipos de referencia, con el efecto descripto anteriormente: un cambio del tipo de la referencia. No hay cambios en el objeto.

En el siguiente ejemplo vemos cómo cambia la referencia del objeto, pero no el objeto:

```
Estudiante estudianteUno = new Estudiante();
Persona personaUno = estudianteUno;
```

Tenemos dos referencias de diferente tipo, apuntando al mismo objeto. Por lo tanto, el objeto no está cambiando constantemente entre una forma y otra, sino que *dependiendo de cómo sea tratado, responderá a una u otra interfaz*.

Dynamic binding

En palabras de Alan Kay, una de las principales características de la Programación Orientada a Objetos son las ligaduras extremadamente tardías.

Se define **dynamic binding** (o ligadura tardía en fuentes en español) como la propiedad de cualquier ejecución de un método en la que se encuentra su versión más adecuada dependiendo del tipo del objeto sobre el cual se la invoca.

Un ejemplo puede dejar las cosas más en claro:

```
class Persona {
  void saludar() { System.out.println("¡Hola!"); }
}

class Estudiante extends Persona {
  void saludar() { System.out.println("¡Presente!"); }
}
```

Si nuestro código fuera:

```
Persona personaUno = new Persona();
Persona personaDos = new Estudiante();
personaUno.saludar();
personaDos.saludar();
```

Obtendremos las siguientes salidas por pantalla:

```
¡Hola!
¡Presente!
```

Analicemos el caso:

1. El código compila, ya que un `Estudiante` es una `Persona`, entonces la asignación es válida (asignación polimórfica).
2. Al momento de invocar el método `saludar()`, en el caso de que el objeto que recibe el mensaje sea del tipo `Persona`, utilizará la primera versión del método, imprimiendo "¡Hola!". En el otro caso, en que el objeto es en realidad un `Estudiante`, utilizará la segunda versión del método, imprimiendo "¡Presente!".

Esto es posible gracias al dynamic binding, la vinculación extremadamente tardía entre las invocaciones y las implementaciones correspondientes.

Este comportamiento, según la página oficial de Oracle, se llama invocación de métodos virtuales, y demuestra un aspecto de la importante característica que es el polimorfismo para la Programación Orientada a Objetos.

Una diferencia semántica

Existe una diferenciación semántica en el tipo de polimorfismo que implementamos: el de subtipos y el de utilidades.

En el polimorfismo de subtipos, existe la taxonomía nombrada y los subtipos son variaciones mutuamente exclusivas, y poseen una característica que será la misma para todos los tipos de la jerarquía. El ejemplo clásico es la `Forma`, con `Circulo`, `Cuadrado`, y el método `dibujar():void`. En pocas palabras: varían en un tema en común.

El segundo tipo es el de utilidad, en el que clases diferentes por definición, poseen un método en común. Ejemplo clásico es el del método `toString():String`, una utilidad provista por el lenguaje e implementada en toda la jerarquía de objetos que nos otorga una representación en forma de cadena de texto del objeto sobre el que se lo invoca.

El límite entre ambos es difuso muchas veces, pero podría clasificarse el tipo de polimorfismo que se implementa en cada ocasión.

Recursos

Bibliografía

- [MEYER-97] **Object-oriented software construction** *Bertrand Meyer* - Prentice Hall PTR - 1997
- [MEYER-09] **Touch of class: learning to program well with objects and contracts**, *Bertrand Meyer* - Springer - 2009
- Cardelli, Luca. On Understanding Types, Data Abstraction, and Polymorphism. 1985. PDF

Links

- Poly Morphism - c2 wiki
- Polymorphism - Java tutorials