Una técnica simple para manejar el polimorfismo múltiple

Daniel H. H. Ingalls

Mail Stop 22-Y
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014

La siguiente es una traducción libre del paper A Simple Technique for Handling Multiple Polymorphism publicado por Dan Ingalls en 1986, cuyo texto original puede encontrarse en el siguiente enlace: https://github.com/algoritmos-iii/algoritmos-iii.github.io/blob/master/assets/bibliografia/simple-technique-for-handling-multiple-polymorphism.pdf. La misma fue realizada por colaboradores del curso Leveroni de Algoritmos y Programación III, en la Facultad de Ingeniería de la Universidad de Buenos Aires. Comentarios, consultas y reportes de errores pueden enviarse a fiuba-algoritmos-iii-doc@googlegroups.com

Abstracto

Hay ciertas situaciones al programar que llevan a tener expresiones polimórficas múltiples, es decir, expresiones en las que varios de los objetos involucrados son de tipo variable. En tales situaciones, las mecánicas habituales de la programación orientada a objetos convencional dejan de funcionar, llevando a código que no es apropiadamente modular. Este paper describe una simple técnica para enfrentar tales problemas que preserva todos los beneficios de las buenas prácticas de programación orientada a objetos en la cara de cualquier cantidad de polimorfismo. Se da un ejemplo en la sintaxis de Smalltalk-80, pero la técnica es relevante para todos los lenguajes orientados a objetos.

Polimorfismo y mensajes

Las técnicas de programación orientadas a objetos fueron introducidas para sobrepasar la barrera de complejidad del polimorfismo en lenguajes extensibles. Intentos anteriores de crear lenguajes extensibles demostraron mucho poder para describir nuevos campos de información, pero fallaron en proveer la misma economía de descripción cuando el tamaño del sistema incrementaba. Los procedimientos en un lenguaje de programación extensible tenían que ser polimórficos, en otras palabras, tenían que lidiar con argumentos de varios tipos distintos. La solución convencional ante tal polimorfismo era probar por cada tipo conocido y después ejecutar el código apropiado para el identificado. Esta técnica, si bien era adecuada para aplicaciones simples, violaba los principios básicos de la modularidad y llevaba a una explosión combinatoria de la complejidad para programas más grandes.

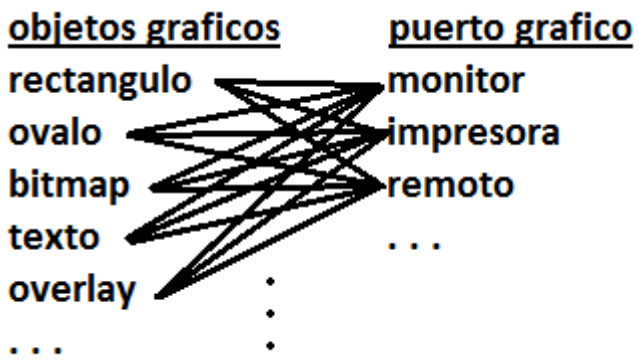
La introducción del paradigma de envío de mensajes para la computación finalmente permitió sobrepasar esta barrera y logró que el potencial de los lenguajes extensibles sea materializado completamente. El proceso del envío de mensajes absorbe la necesidad de verificar la clase del objeto en cuestión. De esta manera los métodos, siendo locales a su propia clase, no son polimórficos, y no dependen de las otras clases del sistema.

Por lo tanto, todos los lenguajes de programación orientados a objetos actuales soportan el polimorfismo simple. Es decir, una variable o expresión que representa el receptor del mensaje puede, dinámicamente, variar en tipo. Diferentes, pero apropiados, resultados serán producidos dependiendo del tipo de cada receptor. Esta capacidad lleva a una gran simplificación de la descripción de los comportamientos de objetos que son diferentes, pero similares al mismo tiempo. Adicionalmente, la mayoría de las implementaciones orientadas a objetos proveen una estructura de envío de mensajes eficiente, de manera que este soporte para receptores polimórficos cuesta apenas un poco más que una llamada convencional a un procedimiento.

El problema

Surgen ciertas situaciones, en cambio, donde más de una variable en una expresión es independientemente polimórfica. Tales casos usualmente llevan a un estilo de programación que se revierte a la prueba explícita de tipos y, en consecuencia, lleva nuevamente a los viejos problemas de la programación procedural.

Utilicemos, como ejemplo, el caso de los objetos gráficos y los distintos tipos de puertos gráficos donde tales objetos pueden ser representados. Claramente, una variable que contiene el objeto gráfico frecuentemente será polimórfica, tomando tales valores como rectángulos, óvalos, líneas, texto, imágenes de mapas de bits u otros objetos gráficos más complejos. Al mismo tiempo, una variable que contiene un puerto gráfico también podrá tomar valores de distintos tipos concretos, como un puerto de un monitor, de una impresora, de un monitor remoto, etc. En consecuencia, tenemos la siguiente interacción polimórfica doble:



Ante esta situación, los programadores frecuentemente escribirán una familia de métodos para cada objeto gráfico con la forma:

```
<Rectangulo>  representarseEn: unPuertoGrafico
        unPuertoGrafico isMemberOf: PuertoDeMonitor
                ifTrue: [“Codigo para representarse en un monitor”].
        unPuertoGrafico isMemberOf: PuertoDeImpresora
                ifTrue: [“Codigo para representarse en una impresora”].
        unPuertoGrafico isMemberOf:  PuertoDeRemoto
                ifTrue: [“Codigo para representarse en un monitor remoto”].
```

… siguiendo de la misma manera para otros objetos gráficos.

Por lo menos ahora el código está apropiadamente distribuido para que sea local a cada objeto gráfico, y sería fácil agregar un objeto gráfico nuevo o editar uno existente.

En contraposición, en relación a los diferentes puertos gráficos, este código será difícil de extender o incluso mantener. Por supuesto que estos métodos podrían haber sido distribuidos en las clases de los puertos gráficos, pero en ese caso sería complejo extenderlos a nuevos objetos gráficos.

Y es así como el programador fue defraudado por el envío de mensajes convencional, que solo soporta polimorfismo de los receptores y no de los argumentos. El código mostrado más arriba crecerá en complejidad con el grado de polimorfismo, generando así una barrera para cualquier programador bien intencionado que quiera agregar un nuevo tipo de objeto gráfico o puerto gráfico. Además, cualquier error cometido al modificar el código para agregar un nuevo tipo de objeto resultará en la falla del código ya existente, posiblemente llevando a la pérdida completa del soporte del ambiente. Todos estos son problemas que la programación orientada a objetos debería haber resuelto.
[Algunos sistemas orientados a objetos recientes, como CommonLoops[1], proveen métodos que son polimórficos en más de un parámetro. Esto remueve la responsabilidad del programador de tener que implementar la solución propuesta a continuación, pero de todas formas la misma es una forma efectiva de implementar tal funcionamiento.]

La solución

Afortunadamente, la solución para lidiar con el polimorfismo múltiple está disponible en todos los lenguajes orientados a objetos que existen - solo es necesario comprender la conexión entre el polimorfismo y el envío de mensajes para reconocer la forma apropiada de abordarlo. En esencia, cada transmisión de mensaje reduce una variable polimórfica a una monomórfica por el tipo de envío inherente a la búsqueda de mensajes. Por lo general (por diseño), solo el receptor es polimórfico y la situación es simple. Sin embargo, en el ejemplo doblemente polimórfico anterior, el envío del primer mensaje solo hace la mitad del trabajo - el argumento del método de destino sigue siendo polimórfico. Esto sugiere que otro mensaje debe ser enviado para reducir el polimorfismo restante.

Para volver a nuestro ejemplo de objeto de representación, uno definiría un método de retransmisión en cada objeto gráfico para efectuar un próximo envío en el tipo de puerto de la siguiente manera:

```
<Rectangulo>  representarseEn: unPuertoDePantalla
        unPuertoDePantalla representarRectangulo: self
<Ovalo>  representarseEn: unPuertoDePantalla
        unPuertoDePantalla representarOvalo: self
<MapaDeBits>  representarseEn: unPuertoDePantalla
        unPuertoDePantalla representarMapaDeBits: self
```

… siguiendo de la misma manera para otros objetos gráficos.

La información obtenida en el primer envío debe conservarse mediante la introducción de una nueva familia de mensajes específicos para los tipos de objetos gráficos. Ahora solo se necesita definir métodos para esta familia de mensajes en cada una de las clases de interfaces gráficas de la siguiente manera:

```
<PuertoDePantalla>  representarRectangulo: unRec
        “Codigo para representar un rectangulo en una interfaz grafica”
<PuertoDePantalla>  representarOvalo: unRec
        “Codigo para representar un ovalo en una interfaz grafica”
<PuertoDePantalla>  representarMapaDeBits: unRec
        “Codigo para representar un mapa de bits en una interfaz grafica”
```

… siguiendo de la misma manera para otros objetos gráficos.

```
<PuertoDeImpresora>  representarRectangulo: unRec
        “Codigo para representar un rectangulo en un puerto de impresora”
<PuertoDeImpresora>  representarOvalo: unRec
        “Codigo para representar un ovalo en un puerto de impresora”
<PuertoDeImpresora>  representarMapaDeBits: unRec
        “Codigo para representar un mapa de bits en un puerto de
impresora”
```

… siguiendo de la misma manera para otros objetos gráficos.

Esta solución conserva la modularidad del estilo de programación orientada a objetos. Si se desea agregar un nuevo tipo de objeto gráfico, nunca es necesario alterar el código existente, sino solo definir el mensaje de retransmisión en la nueva clase y los métodos de implementación correspondientes en cada una de las actuales clases de puertos. Agregar una nueva clase de puerto es aún más simple, ya que solo equivale a implementar la familia completa de representarX: mensajes.

Por supuesto, la solución inversa en la que los puertos se transmiten a objetos gráficos, tendría propiedades de modularidad igualmente buenas. La elección en este caso depende de una decisión de diseño en cuanto a si los métodos finales pertenecen más a las clases de objetos gráficos o a las clases de puertos de visualización.

La técnica descrita anteriormente también se puede utilizar para reducir grados más altos de polimorfismo. Cada envío de mensajes subsiguientes reduce un mayor grado de polimorfismo. Afortunadamente, así como el polimorfismo doble es mucho menos común que el polimorfismo simple, los grados más altos son mucho más raros aún.

Experiencia

El enfoque mencionado anteriormente ha demostrado su eficacia en varias situaciones además del ejemplo de visualización citado. Uno es la interacción entre diferentes tipos de eventos y controladores de eventos. Otro surge en conexión con la programación lógica donde tanto el receptor como el argumento del mensaje unificarseCon: son polimórficos en constantes, variables, términos y otras formas. Un tercero es una reescritura experimental de la lógica de coerción aritmética en el sistema Smalltalk-80.

Referencias

[1] CommonLoops: Merging Lisp and Object-Oriented Programming, by Daniel Bobrow et aL, Proceedings of OOPSLA '86, September 1986, Portland Oregon