Jerarquía Polimórfica

Bobby Woolf

La siguiente es una traducción libre del paper Polymorphic Hierarchy publicado por Bobby Woolf en 1996, cuyo texto original puede encontrarse en el siguiente enlace: https://algoritmos-iii.github.io/assets/bibliografia/polymorphic-hierarchy.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

Me he dado cuenta de que al menos la mitad de los métodos que escribo no son muy originales. Claro, muchos de los métodos que escribo son métodos reales que hacen un trabajo real y tienen nombres reales que son bastante únicos. Sin embargo, en el proceso de la programación basada en la realidad, también escribo una gran cantidad de métodos que no requieren tanta reflexión. Implementó un montón de métodos getter y setter, métodos de inicialización y creación de instancias, y escribo un buen número de métodos que subimplementan métodos ya definidos en una superclase. He descubierto que son estos métodos subimplementors la clave del polimorfismo. Usados de forma activa y consistente, los métodos polimórficos conducen a clases polimórficas y, en última instancia, a jerarquías polimórficas.

Discutiré lo que es una jerarquía polimórfica así como un patrón que llamó "*Clase de la Plantilla*" (*Template Class*). La clase superior en una jerarquía polimórfica es una *Clase de la Plantilla*.

Reutilizando descripciones de métodos

Utilicemos printOn: como ejemplo. *VisualWorks* implementa *printString* en la clase *Object* mediante *printOn*:. Dado que *Object>>printOn:* es genérico, a menudo lo subimplemento en mis clases para que me diga no sólo de qué clase proviene esta instancia, sino también para proporcionar alguna idea sobre qué instancia es. Siendo un buen ciudadano corporativo, comento mi nuevo implementor de printOn: con una descripción para dar al siguiente programador alguna idea sobre lo que hace este método.

La descripción en *Object>>printOn*: dice: "Añade al argumento *aStream* una secuencia de caracteres que describe el receptor". No podría haberlo dicho mejor. De hecho, ¿por qué debería intentarlo? ¿Qué crees que hace este método? Habiendo visto un centenar de otros implementors de *printOn*: en todo el sistema, todos los cuales hacen lo mismo,

¿qué crees que hace éste? Mi filosofía es duplicar con el menor esfuerzo posible, así que ¿por qué debería comentar un método que ya ha sido comentado en otro lugar? En su lugar, simplemente describo mi método diciendo: "Ver *superimplementor*". Es mi manera de que el método diga, "Mira, yo hago lo mismo que hace mi *superimplementor*. ¿Qué otra cosa podría hacer? Si no sabes qué es eso es, vea la descripción en el *superimplementor*".

Otra pista de que mi *implementor* de *printOn*: hace lo mismo que el que está definido en Object es que ambos *implementors* aparecen en el mismo protocolo de método, printing. El protocolo de printing en *Object* dice, "Estas son todas las cosas del tipo printing que el receptor sabe hacer". Al poner mi implementor en el protocolo de printing, estoy diciendo que mi método también se utiliza para imprimir. Eso es algo bueno ya que estoy subimplementando un método que ya se utiliza para imprimir. Un *subimplementor* no sólo debe hacer lo mismo que su superimplementor, también debería aparecer en el mismo protocolo.

Un implementor definitorio

Cuando he escrito todos los *implementors* de un mensaje, sólo hay una descripción, y es en el método de la superclase que define la jerarquía. El *implementor* de la superclase suele ser bastante pobre; sólo devuelve *self*, o el error *subclassResponsibility*, o alguna implementación por defecto que no causará problemas cuando las subclases lo hereden. Pero aunque la implementación sea poco desarrollada, el método sigue definiendo lo que este mensaje hace para toda la jerarquía. Por lo tanto, cualquier *subimplementor* debe hacer lo mismo; sus implementaciones son diferentes, pero sus propósitos son los mismos. Por lo tanto, sus descripciones deben ser: "Ver *superimplementor*".

El mismo principio se aplica a la costumbre de Smalltalk de hacer que un método simplemente llame a otro método que tiene casi el mismo nombre excepto por un parámetro extra. Por ejemplo, *Object>>changed* envía *changed*: y a su vez *Object>>changed:* envía *changed:with:*. No es necesario describir el propósito de estos tres métodos. Una vez que sepa lo que hace *changed:with:*, sabré qué *changed* y *changed:* funcionan de la misma manera, excepto que utilizan valores por defecto para algunos de los parámetros. Así que mi descripción para *Object>>changed* sería "Ver *changed:*"; "Ver *changed:with:*" describe *Objeto>>changed:*. Si subimplementara cualquiera de estos métodos, la descripción sería "Ver *superimplementor*".

Anatomía de la descripción de un método

Entonces, ¿qué tipo de alardes inmodestos y confesiones secretas acaban en la descripción de un método?

En primer lugar, intentó evitar reformular el nombre del método. Por desgracia, he escrito muchos métodos como *codigoDeProducto* cuya descripción dice: "Devuelve el código del producto del receptor". No, ¿¡en serio!? Este tipo de métodos suelen devolver el valor de una variable de instancia, así que cuando no se me ocurre otra cosa que decir, los describo con "*Getter*". Lo mismo ocurre con los métodos *setter*. Dado que ya deberías saber lo que hace la variable de instancia, no hay mucho que decir sobre los métodos *getter y setter*.

En segundo lugar, intentó utilizar la descripción del método para describirlo en su totalidad. Generalmente no me gusta comentar líneas individuales de código. Es tentador hacerlo cuando escribo una o dos líneas de código que son tan extrañas que otro programador no tiene posibilidad de entender lo que está viendo. Por desgracia, no voy a poder enmendar este desafortunado error incluyendo un comentario igualmente enrevesado. En su lugar, oculto el código extraño en un nuevo método con un nombre descriptivo. El comentario que habría incluido para explicar el código se convierte en la descripción del método.

Propósito y detalles de implementación

En tercer lugar, he dividido la descripción de mi método en dos partes: propósito y detalles de aplicación. El propósito explica lo que hace el método. Lo expresó como: "Si envías este mensaje a este objeto, esto es lo que ocurrirá. Este método hará..." Los ejemplos podrían ser: "Ordenar los elementos del receptor", "Leer el siguiente elemento y devolverlo", y "Responder si el receptor contiene errores". Los detalles de implementación son opcionales. Estos detalles serían mi lista de excusas de por qué el código es tan extraño. Por suerte escribo un código razonablemente bueno la mayor parte del tiempo, así que sólo unos pocos de mis métodos necesitan estas explicaciones.

El propósito de un método es reutilizable. Es mejor que todos los *implementors* de un mensaje dentro de una jerarquía tengan el mismo propósito. Por lo tanto, sólo documento el propósito del mensaje en el *implementor* en la parte de mayor jerarquía. Otros implementors documentan su propósito con "Ver *superimplementor*".

Los detalles de implementación de un método no son reutilizables. Dos *implementors* de un mismo mensaje no deberían tener los mismos detalles de implementación, si los tienen significa que existe código repetido.

De este modo, puede dividir fácilmente sus propias descripciones de métodos en detalles de implementación y propósito. Para ello separe el comentario que explica qué hace el método del que explica cómo. El comentario del “qué” es el propósito del método; debería ser el mismo para cada *implementor* en la jerarquía. El comentario del “cómo” describe los detalles de la implementación, por lo que debería ser diferente (o despreciable) para cada *implementor*. En todos los *implementors* , excepto en el más alto en la jerarquía, cambie el comentario del propósito por "Ver *superimplementor*", ya

que el propósito es siempre el mismo. Añada un comentario de detalles de implementación si lo considera necesario. Cuando leo las descripciones de los métodos de otros programadores (como el código del proveedor), realizó mentalmente esta separación para entender mejor el método.

Reutilización de descripción para el polimorfismo

Una de las cosas que me costó aprender sobre Smalltalk es para qué sirven las subclases. Para mí, cada clase era su propio paquete de detalles de implementación. Cualquier cosa que dos clases tuvieran en común era una coincidencia. La manera de elegir una superclase era seleccionar aquella que permitiera a mi clase heredar la mayor cantidad de cosas "gratis". Yo no creaba jerarquías, sólo grupos de clases.

Ahora no puedo pensar en una clase sin tener que entender también sus superclases. Ya no uso navegadores de clases, sino navegadores de jerarquías. Las superclases son como un conjunto lineal de mezclas y quiero ver lo que se está mezclando en esta clase. Encuentro que tratar de entender una clase sin conocer sus superclases es como escuchar un chiste privado sin conocer el contexto del que proviene.

Cuando creo una subclase, no sólo pienso en cómo debería funcionar la clase, sino en cómo debería diferir de su superclase. La superclase ya debería hacer prácticamente todo lo que la subclase necesita saber hacer (aunque la subclase puede añadir comportamientos adicionales). El problema es que la superclase sabe qué hacer pero no cómo hacerlo. La subclase implementa el cómo.

Considere la jerarquía *Collection*. Una *Collection* puede aceptar peticiones para añadir y eliminar elementos, iterar sobre ellos, y responder cuántos elementos tiene. Pero una *Collection* no sabe cómo satisfacer estas peticiones. El cómo depende de la implementación; ¿es la *Collection* una lista, un árbol, una tabla hash, o qué? Las subclases de *Collection* implementan los detalles del cómo. *Set* (una tabla hash) implementa *add*:, *remove*:, *do*: y *size* de una manera. *OrderedCollection* (una lista) las implementa de otra manera. *Collection* define lo que puede hacer una colección; las subclases definen cómo se hace.

¿Cómo me aseguro de que la subclase haga lo mismo que la superclase? Cada método que la subclase subimplemente (extienda o sobreescriba de la superclase) tiene que hacer lo mismo que el método de la superclase. El *subimplementor* no lo hace de la misma manera, por supuesto, pero debe producir el mismo resultado. En otras palabras, el *subimplementor* debe tener el mismo propósito que el *superimplementor*.

El propósito es polimórfico

Cuando todos los *implementors* en una jerarquía tienen el mismo propósito, son polimórficos. Cuando todos los métodos que las subclases subimplementan son

polimórficos con sus versiones heredadas, la jerarquía es polimórfica. Esto significa que un objeto colaborador puede usar una instancia de la jerarquía con igual facilidad que cualquier otra.

Ya que todas las instancias se comportan de la misma forma, funcionara igual de bien con todas.

Por ejemplo, consideremos un objeto (*Empleado*) que deberá mantener una lista de cosas a hacer (*cosasAHacer*). ¿Cómo debería ordenar esta lista el objeto, por prioridad o por orden de entrada? ¿Quién sabe? Pero sabemos que va a ser algún tipo de *Collection*, probablemente una *OrderedCollection* o una *SortedCollection*. Así que sin importar cómo decidas implementar la *cosasAHacer*, ya sabrás cuál es su comportamiento. Podrás agregar una tarea (*add*:), eliminar una tarea que ya fue completada (*remove*:), preguntar cuántas tareas quedan por ser completadas (*size*), y preguntar cuál es la siguiente tarea (*first*). Luego se pueden determinar los detalles específicos al dominio, como debería ser ordenada la lista, o si deberíamos usar una *OrderedCollection* o una *SortedCollection*.

Definiendo polimorfismo

La definición del polimorfismo es mucho más extensa que la que se escucha normalmente. La definición más “pegadiza” que he aprendido es que dos métodos son polimórficos cuando tienen el mismo nombre. Esto no siempre es verdad. Piensa en los mensajes *value* y *value*: .Tienen muchos *implementors*, ¿pero estos son polimórficos? Si son polimórficos, ¿qué hacen dichos mensajes? Eso depende de qué objeto lo recibe. En *ValueModel*, *value* y *value*: son métodos de acceso (*getter* y *setter*). En un bloque (*BlockClosure*) *value* y *value*: son métodos de evaluación, claramente no son *getters* y *setters*. Estos *implementors* de *value* y *value*: tienen el mismo nombre, pero no son polimórficos.

Para que dos métodos sean polimórficos, no solo deben tener el mismo nombre, sino que también se deben comportar de la misma forma. Esto no solo significa que aceptan la misma cantidad de parámetros, sino también que cada parámetro sea del mismo tipo en ambos métodos. Los dos métodos deben producir los mismos efectos secundarios, como cambiar el estado del objeto que lo recibe de la misma forma. También, ambos métodos deben devolver el mismo tipo de resultado. Solamente luego de cumplirse todo esto dos métodos serán realmente polimórficos.

Como fue descrito anteriormente, sostengo que dos clases pueden ser polimórficas. Dos clases polimórficas entienden los mismos mensajes y sus *implementors* de dichos

mensajes son polimórficos. Ya que ambas clases comparten la misma interfaz y se comportan de la misma forma, son polimórficos. En la práctica, dos clases no comparten la misma interfaz a menudo, pero si comparten la misma interfaz base y esta es polimórfica. Una interfaz base es una interfaz que varias clases comparten, de forma que puedan ser usadas de forma intercambiable. Mientras un colaborador solo use la interfaz base, podrá usar una instancia de una clase con igual facilidad que una de otras clases que compartan la interfaz.

Haciendo una Jerarquía Polimórfica

Esto me lleva a mi comentario en métodos “Ver *superimplementor*”. En tanto logré acostumbrarme a comentar los métodos de esta forma, también lo hice a pensar en mis métodos de manera polimórfica, lo que me hizo más cuidadoso a la hora de implementarlo polimórficamente. Mientras los métodos de mis jerarquías se vuelven más polimórficos, también lo hacen las jerarquías. Mientras las jerarquías se vuelven más polimórficas, se hacen más flexibles, reusables, extensibles, fáciles de aprender, y todo lo bueno de la orientación a objetos.

El problema surge cuando no hay un *superimplementor* para ver. A menudo estoy implementando un método y recuerdo que ya había implementado otro método con el mismo nombre. Están haciendo lo mismo, por lo tanto, tienen el mismo propósito. Si uno de ellos es un *subimplementor* del otro, describo el *subimplementor* con “Ver *superimplementor*”. Pero no puedo hacer esto cuando dos clases tienen una implementación polimórfica del mismo mensaje, pero no hay un *superimplementor*. Cuando esto sucede, el código me está diciendo “Falta un *superimplementor*”. No quiero repetir un esfuerzo que puedo evitar. Esto incluye definir qué debería hacer este mensaje dentro de esta jerarquía. Así que puedo introducir un *superimplementor*, documentar el propósito allí, y darle una implementación por defecto. Ahora podre documentar los *subimplementors* con “Ver *superimplementor*”.

Desde luego, otro problema con el que a veces me cruzo es que no hay una superclase en la que incluir el *superimplementor*. En este caso, dos clases con implementaciones polimórficas de un mismo mensaje no tienen relación dentro de la jerarquía. Su primera superclase en común es una genérica, tales como *ApplicationModel* u *Object*. Pero definitivamente no voy a agregar un mensaje especifico a mi dominio en una clase tan general, como si todas sus subclases deberían entender dicho mensaje, ¿entonces que puedo hacer? Si estos dos *implementors* son realmente polimórficos, entonces sus clases seguramente deban ser polimórficas también. De ser así, la forma más mantenible de hacer dos clases polimórficas es poniéndolas en la misma jerarquía y hacer toda esta jerarquía polimórfica. Ya que esta jerarquía no existe, debo crear una nueva clase abstracta que describa el comportamiento polimórfico que tiene, luego

subclasificando dos clases concretas a partir de esta. La jerarquía ahora existe, así que podré agregar el *superimplementor* en la superclase. El propósito del mensaje se encontrará en el *superimplementor* y los *subimplementors* tendrán el comentario "Ver *superimplementor*".

Patrón de Clase de la Plantilla (Template Class Pattern)

La clase abstracta que introduje para crear una jerarquía polimórfica es lo que llamo una *Clase de la Plantilla* (*Template Class*). *Clase de la Plantilla* es un patrón que crea jerarquías polimórficas. Es similar al patrón de *Método de la Plantilla*[1] (T*emplate Method*). Mientras que un *Método de la Plantilla* define la interfaz para un método, dejando los detalles a las subclases, una *Clase de la Plantilla* define la interfaz para una clase (un nuevo tipo) dejando los detalles de implementación para las subclases. Esto lleva a una jerarquía de clases que son polimórficas una con la otra. Una *Clase de la Planilla* normalmente está llena de *Métodos de la Plantilla*.

La jerarquía de *ValueModel*

La jerarquía de *ValueModel* en *VisualWorks* es un buen ejemplo de la jerarquía polimórfica. La clase *ValueModel* define la jerarquía diciendo que todas las instancias entenderán mensajes como *value*, *value*: y *onChangeSend: to:*. Todas las subclases implementan estos mensajes de acuerdo a cómo funciona cada clase. Una clase heredará algunas de las implementaciones si ya funcionan de forma adecuada.

Ya que todas las subclases de *ValueModel* son compatibles con la interfaz base, un colaborador puede usar un *ValueModel* sin tener que considerar a qué subclase pertenece la instancia. Sea que la instancia es un *ValueHolder*, un *AspectAdaptor* o un *TypeConverter*, es compatible con la interfaz base de *value*. Este es un beneficio del polimorfismo a nivel de clases. Ya que la jerarquía es polimórfica, es posible usar una clase de la jerarquía con igual facilidad que otras y el código que colabora nunca notará la diferencia.

La mayoría de las jerarquías que conocemos y tanto queremos–*Collection, Magnitude, ArithmeticValue/Number, Boolean, String*, etc.–son jerarquías polimórficas. Esto no es una coincidencia. El hecho de que las conocemos tan bien y las utilizamos tan a menudo es en gran parte gracias a que su polimorfismo las hace tan fáciles de utilizar.

[1] Gamma, Erich, Richard Helm, Ralph Johnson, y John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995, ISBN 0-201-63361-2; http://www.aw.com/cp/Gamma.html.

Conclusión

En resumen:

- Muchos métodos no solo tienen el mismo nombre, sino que también hacen lo mismo. De aquí es de donde viene el polimorfismo.
- Para que dos métodos sean polimórficos, no solo deben tener el mismo nombre, sino que también los mismos tipos de parámetros, los mismos efectos secundarios, y el devolver el mismo tipo de resultado.
- Dos métodos que son polimórficos deberían aparecer en el mismo protocolo de métodos.
- La descripción de un método documenta dos cosas: el propósito y los detalles de implementación.
- Para que dos métodos sean polimórficos, deben tener el mismo propósito. No deberían tener los mismos detalles de implementación.
- Para que dos clases sean polimórficas, deben compartir la misma interfaz base de mensajes polimórficos.
- Un colaborador puede utilizar las dos clases de forma intercambiada si solo utiliza mensajes que pertenecen a su interfaz base.
- Para que una jerarquía sea polimórfica, todas sus clases deben compartir la misma interfaz base polimórfica.
- La jerarquía polimórfica y su interfaz base es definida por una clase abstracta. Llamo a esta clase una *Clase de la Plantilla*.
- Una jerarquía polimórfica encapsula código que es altamente reusable, flexible, extensible, y todo lo que implica la orientación a objetos.

Para aprender más sobre cómo implementar tus propias jerarquías polimórficas, sugiero leer el paper “Reusability Through Self-Encapsulation” por Ken Auer[2]. Es un lenguaje de patrones que describe cómo desarrollar una clase jerárquica que logra ser reusable mediante la herencia mientras mantiene la encapsulación de cada clase. Aunque el polimorfismo no es un objetivo explícito del lenguaje de patrones, las jerarquías desarrolladas de esta manera tienden a ser polimórficas.

Bobby Wolf es un Senior Member of Technical Staff en Knowledge Systems Corp. en Cary, Carolina del Norte.

Enseña a clientes sobre el uso de VisualWorks, ENVY, y patrones de diseño. Agradece sus comentarios en woolf@acm.org o en http://www.ksccary.com.

[2] Coplein, James y Douglas Schmidt, Editores. Pattern Languages of Program Design. Addison-Wesley, Reading, MA, 1995, ISBN 0-201-60734-4; http://heg-school.aw.com/cseng/authors/coplien/patternlang/patternlang.html.

La clave del polimorfismo está en los métodos separados que no solo tienen el mismo nombre, sino que también hacen lo mismo. Igualmente, los métodos polimórficos no son suficientes. Deben estar agrupados en clases polimórficas que forman jerarquías polimórficas. Bobby explora cómo las jerarquías polimórficas surgen durante el proceso de desarrollo.

La descripción para un *subimplementors* polimórfico es, “Ver *superimplementor*.”

La descripción de un método tiene dos partes: el propósito y los detalles de implementación.

El propósito de un método es reusable; los detalles de la implementación no lo son.

La descripción “Ver *superimplementor*” hace que los desarrolladores piensen de manera polimórfica.