Parte Teórica

Tomás Felipe Melli

July 16, 2025

Índice

1 1C2025

1.1 Explique un double dispatch genérico

Si miramos el paper de double dispatch :

> Double dispatching is a way of choosing methods based on the class of the arguments as well as of the receiver. It involves a standard Smalltalk method–lookup, or message dispatch, for each argument, as well as for the receiver. Arithmetic operations usually have one argument, so there are two message dispatches — i.e. double dispatching. Dan Ingalls called this technique multiple polymorphism, which is more accurate but less poetic.
> In general, a double dispatch method for a message op is : argument opWordFromReceiverClass: self.

Podemos decir que un double dispatch es un mecanismo para que dos objetos interactúen (a través de un mensaje) en el cual, uno de ellos se envía como argumento. Esto es necesario para que el objeto receptor, en base a algún criterio, resuelva y luego, responda. Este mecanismo es útil ya que permite eliminar ifs. Esto es posible ya que el objeto receptor puede, a partir del objeto que recibe como argumento, saber de qué clase de objeto se trata. Como el ejemplo de este mismo libro sobre operaciones aritméticas :

> For example, the double-dispatch implementation of the method for "*" does nothing except send a message to the argument. The name of the message encodes the name of the original message (i.e. "*") and the class of the original receiver. The method for "*" in Fraction is :
> ↑ aNumber productFromFraction: self.
> Thus, multiplication of a Fraction and another number causes two message dispatches, one on "*" and one on productFromFraction: . Each subclass of Number must implement the productFromFraction: method. These methods are usually very simple, since the class of the receiver and the argument are both known. Multiplying a Fraction with a Float will cause productFromFraction: to be sent to the Float with the Fraction as the argument. The definition of productFromFraction: in class Float is:.
> productFromFraction: aNumber
> ↑ self productFromFloat: aNumber asFloat
> which directly coerces the Fraction to be a Float. Multiplying two Floats is implemented by a primitive in Smalltalk-80, so the definition of productFromFloat: is
> productFromFloat: aFloat
> ¡primitive: 49¿
> self error: 'The result cannot by represented by a Float.'.

En este ejemplo vemos cómo el mensaje '*' implementado con double dispatch lo que hace es mandar un mensaje al objeto que es argumento de manera de 'encodear' el nombre del mensaje original y su clase receptora. Esto es útil para sacar ifs por ejemplo :).

1.2 Qué diferencia hay entre cohesión y acoplamiento?

Un objeto es cohesivo cuando sus responsabilidades están bien definidas y no asume responsabilidades que no son esenciales a sí. Un objeto está altamente acoplado cuando para cumplir sus responsabilidades depende directamente de muchos otros (cuánto depende de otros para funcionar ?). Este esquema rígido hace difícil hacer modificaciones y es poco extensible.

1.3 Cómo se implementa un wrapper polimórfico en Smalltalk ?

Un wrapper es un objeto que envuelve a otro con la finalidad de delegar mensajes, agregar funcionalidad, controlar el acceso,.. entre otras cosas. Un wrapper polimórfico es básicamente un objeto que evuelve a otro/s y expone una **interfaz polimórfica** es decir, puede responder los mismos mensajes que los objetos envueltos. En smalltalk podemos utilizar el patrón estructural Decorator para lograr esto, delegan todo el comportamiento a sus decoratee.

1.4 Diferencia entre Observer y Active Variable y cuál es la ventaja de este sobre el patrón de Gang of Four

El patrón Observer es un patrón del tipo Behavioural en el que cierto sujeto (subject) mantiene una lista de Observers a los cuáles notificará cuando este sujeto cambie. Este patrón es particularmente útil cuando hay una dependencia entre el sujeto y otros objetos y para desacoplarlo, se utiliza el Observer para ello. Al ocurrir el cambio por tanto, se propaga por los observadores hacia los objetos que reaccionan a estos cambios evitando la dependecia directa con el mismo.
Por otro lado, Active Variable es una variable que puede ser de instancia o temporal que se declara (av1 getFn1 putFn1) donde av1 representa el nombre de la AV, el segundo campo es el mensaje que se manda a self cuando se accede a ella y el tercer campo es el mensaje que se envía cuando su valor cambia.

Podemos decir que las diferencias son, en principio que uno es un patrón de diseño de tipo Behaviuoral que es básicamente un enfoque de diseño y que AV es una implementación propia del lenguaje en la que se otorga un comportamiento especial, reactivo, a una variable del sistema. Si bien hay similitudes en esto de que hay una reacción al cambio, podríamos pensar a la AV como el sujeto que al cambiar propaga ese cambio y alguien lo escucha, como sucede con el sujeto y sus observadores. Tal vez, si no está desacertado decir que el Observer es un patrón que nace como consecuencia de una clase con muchísimas AV como variables de instancia que por cuestiones de extensibilidad se decidió sacarlas para desacoplar la clase de todos esos cambios y simplemente definirle un observer para que propage dichos cambios (no estoy seguro de esto, es un pensamiento!).

1.5 Explique el patrón Facade

El patrón Facade es un patrón de diseño del tipo Behaviuoral en el que se condensa la representación de un subsistema para facilitarle el uso al usuario (interfaz simplificada a las funcionalidades del sistema). Un ejemplo de la clase es el ejercicio de TusLibros.

1.6 Qué es el method lookup y el metamodelo de Smalltalk

El methodLookup es el mecanismo que tiene smalltalk para, a partir del selector del mensaje, buscar en el protocolo del objeto receptor cómo responder a dicho mensaje, en caso de no encontrarlo, subirá por la jerarquía, y en caso de no estar, se le enviará al objeto receptor doesNotUnderstand: selector.
El metamodelo de smalltalk es la estructura del sistema en cuanto a la organización de las clases, metaclases y objetos. Su estructura circular controlada y la posibilidad de meta-programar.