El patrón Object Recursion ~ Bobby Woolf

La siguiente es una traducción libre del paper *Object Recursion Pattern* publicado por Bobby Woolf en 1998, cuyo texto original puede encontrarse en el siguiente enlace: https://algoritmos-iii.github.io/assets/bibliografia/object-recusion-pattern.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

Intención

Distribuir el procesamiento de una solicitud sobre una estructura delegando polimórficamente. Object Recursion permite, de forma transparente, que una solicitud se divida varias veces en partes más pequeñas, que son más fáciles de manejar.

También conocido como

Recursive Delegation (Delegación Recursiva).

Motivación

Consideremos la necesidad de determinar si dos objetos son equivalentes. Los objetos simples y las primitivas son fáciles de comparar; simplemente se usan operaciones nativas. La dificultad yace en comparar arbitrariamente objetos complejos.

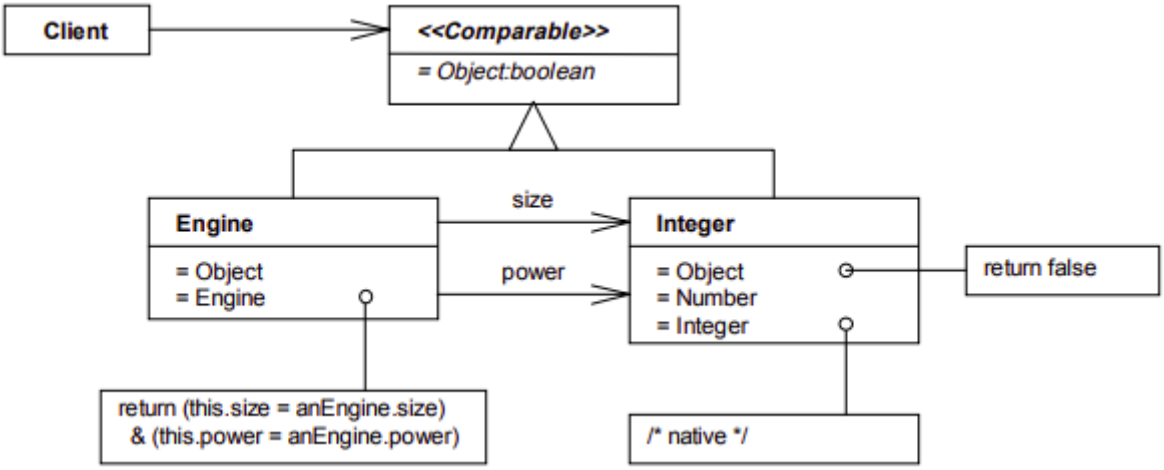
Un enfoque es utilizar un objeto Comparador que acepte dos objetos arbitrariamente complejos y responda si los sujetos son equivalentes. El Comparador toma los sujetos, divide cada uno en partes, y compara cada parte para determinar si son equivalentes. Si alguna de las partes es muy compleja para comparar, el Comparador repite el proceso dividiéndola en partes, y así sucesivamente hasta que todas las partes sean lo suficientemente simples como para compararlas.

Este enfoque del Comparador tiene varias consecuencias indeseables. El Comparador debe reconocer qué tipo de objeto son los sujetos y saber cómo descomponerlos en partes simples que puedan ser comparadas. Cuando más complejo es un sujeto, más complejo será el código para compararlo. Cada vez que un desarrollador implementa una nueva clase cuyas instancias podrían necesitar ser comparadas, tendrá que agregar código al Comparador (o a alguna subclase) para soportar la nueva clase. El código de descomposición en el Comparador depende fuertemente de la implementación de la clase, por lo que cada vez que la implementación cambie el código del Comparador también deberá ser modificado. Los sujetos necesitan proveer de protocolos adicionales para la descomposición, que exponen las implementaciones de los objetos. Considerándolo todo, el Comparador requiere de mucho código complejo, que es difícil de desarrollar y de mantener.

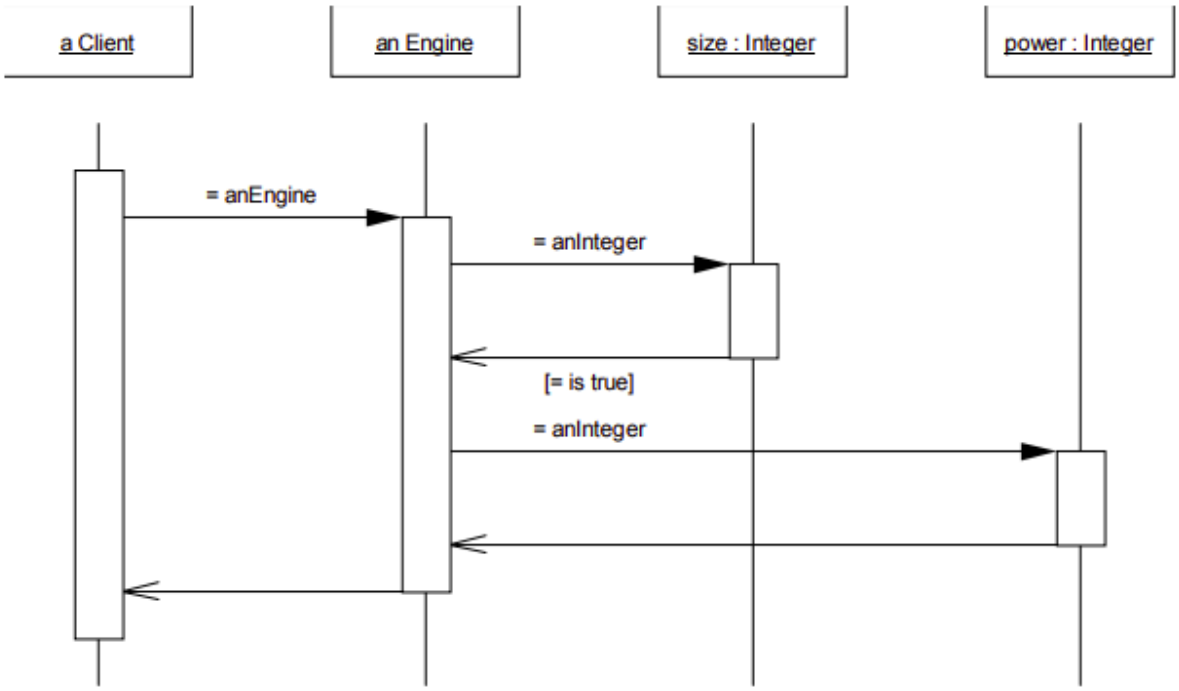
Un enfoque más orientado a objetos es que el Comparador le diga a los sujetos que se comparen entre sí, y los deje decidir cómo hacerlo. De esta manera, el Comparador le está diciendo a los objetos *qué* hacer, pero no *cómo* hacerlo. Con este enfoque, el objeto Comparador ni siquiera es necesario porque cualquier Cliente simplemente puede pedirle a un sujeto que se compare a sí mismo con otro.

Entonces, ¿cómo se comparan los sujetos entre sí? Uno determina si el otro es equivalente a sí mismo. Evalúa qué partes de su estado deben ser equivalentes y luego las compara. Cada una de esas partes evalúan cuáles de sus partes deben ser equivalentes y las compara, y así sucesivamente hasta que todas las partes son objetos simples. A cada paso, el proceso de comparación es relativamente simple. Incluso un objeto muy complejo no necesita saber cómo compararse a sí mismo por completo; solo necesita saber cuáles son sus partes, y ellas deben saber cómo compararse.

Por ejemplo, consideremos un objeto `Motor` que conoce su cilindrada y potencia total. Para compararlo con otro motor, un cliente debe verificar que ambos motores sean del mismo tamaño y de la misma potencia. El siguiente diagrama muestra las clases involucradas y cómo implementar `=`:



El `Cliente` envía `=` al primer `Motor` con el segundo `Motor` como argumento. El primer `Motor` se compara con el segundo comparando su `tamaño` y su `potencia`. Las partes son `Integer`, objetos sencillos que el sistema puede comparar. Si las partes fuesen objetos complejos, continuarían el proceso de comparación comparando sus partes. Eventualmente, las partes más simples son iguales o no lo son.



Este algoritmo de comparación, donde un objeto se compara a sí mismo con otro diciéndole a sus partes que se comparen entre sí sucesivamente es un ejemplo de Object Recursion. Una implementación de un mensaje recursivo envía el mismo mensaje a uno o más de sus objetos relacionados. El mensaje navega por la estructura enlazada hasta alcanzar objetos que simplemente implementan el mensaje y devuelven el resultado.

Claves

Un sistema que incorpora el patrón Object Recursión tiene las siguientes características:

- Dos clases polimórficas, una de ellas maneja una consulta recursivamente y otra que simplemente maneja la consulta sin recursión.
- Un mensaje separado, usualmente en una tercera clase que no es polimórfica con las dos primeras, para iniciar la consulta.

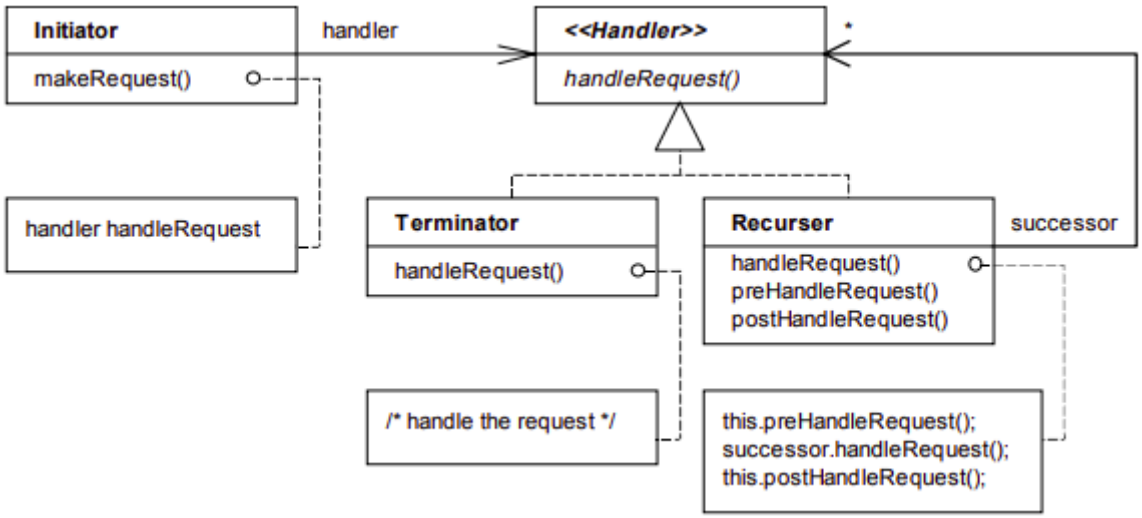
Aplicabilidad

Usaremos el patrón Object Recursion cuando:

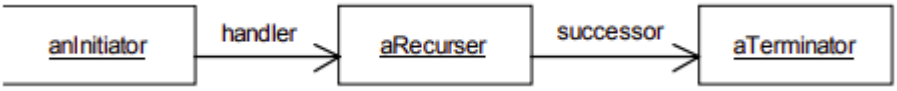
- estamos pasando un mensaje por una estructura enlazada donde el destino final es desconocido.
- estamos enviando un mensaje a todos los nodos que son parte de una estructura enlazada.
- estamos distribuyendo la responsabilidad de un comportamiento a lo largo de una estructura enlazada.

Estructura

El siguiente diagrama de clases exhibe los roles de los participantes:



Un estructura típica de objetos podría lucir así:

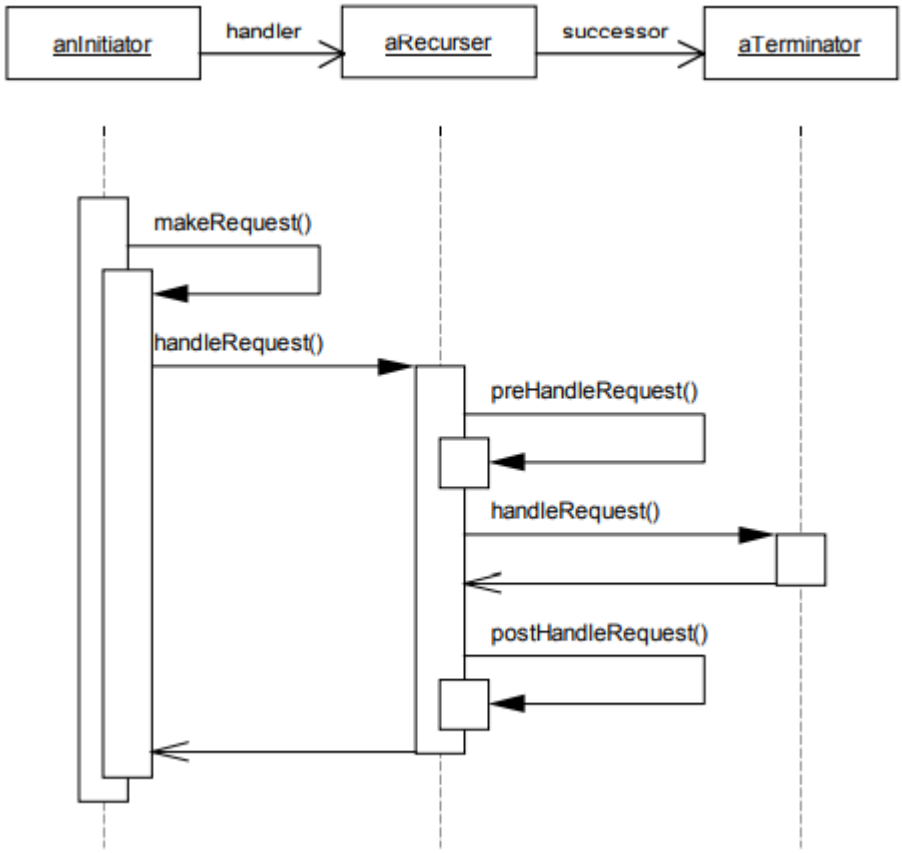


Participantes

- **Initiator** (`Cliente`)
  - Inicia la consulta
  - Usualmente no es un subtipo de `Handler`. `makeRequest()` es un mensaje separado de `handleRequest()`.
- **Handler** (`Comparable`)
  - Define un tipo que puede manejar consultas que los initiators realizan.
- **Recurser** (`Motor`)
  - Define el enlace sucesor.
  - Maneja una solicitud delegándosela a sus sucesores.
  - Los sucesores relevantes para una solicitud pueden variar según la solicitud.
  - Pueden llevar a cabo comportamiento extra antes o después de delegar la solicitud.
  - Puede ser un *Terminator* (enlace terminal) para una solicitud diferente.
- **Terminator** (`Integer`)
  - Termina la solicitud implementándola por completo, no delega ninguna parte de su implementación.
  - Puede ser un *recurser* para otra solicitud diferente.

Colaboración

- El *Initiator* necesita realizar una solicitud. Le pide a su *Handler* que la maneje.
- Cuando el *Handler* es un *Recurser,* hace el trabajo que necesite hacer, le pide a su sucesor -otro *Handler*- que maneje la solicitud, y devuelve un resultado basado en el resultado del sucesor. El trabajo extra puede realizarse antes y/o después de delegar al sucesor. Si el *Recurser* tiene múltiples sucesores, delega a cada uno por turnos, quizás asincrónicamente.
- Cuando el *Handler* es un *Terminator,* maneja la solicitud sin delegarla a ningún otro sucesor y devuelve el resultado (si hubiera uno).



Consecuencias

Las ventajas del patrón Object Recursion son:

- *Procesamiento distribuido*. El procesamiento de la solicitud se distribuye a lo largo de una estructura de *handlers* que pueden ser tan numerosos y complejos como sea necesario para completar la tarea de la mejor manera.
- *Flexibilidad en las responsabilidades*. El *Initiator* no necesita conocer cuántos *Handlers* hay, cómo están organizados, o cómo está distribuido el procesamiento. Simplemente hace la solicitud a su *handler* y deja que los *Handlers* hagan el resto. La organización de los *Handlers* puede cambiar en tiempo de ejecución para reconfigurar dinámicamente el manejo de responsabilidades.
- *Flexibilidad de roles.* Un *handler* que actúa como un *Recurser* para una solicitud puede actuar como *Terminator* para otra solicitud, y viceversa.
- *Aumento del encapsulamiento.* Encapsula las decisiones sobre cómo manejar una solicitud dentro del objeto que la está manejando.

Las desventajas del patrón Object Recursion son:

- *Complejidad para la programación.* La recursividad, ya sea procedural u orientada a objetos, es un concepto difícil de entender. Su sobreuso puede hacer a un sistema más difícil de entender y mantener.

Implementación

Hay algunos problemas a considerar cuando se implementa el patrón Object Recursion:

1. *Tipos separados del Initiator.* El mensaje Initiator.makeRequest() no debe ser polimórfico con el mensaje Recursor.handleRequest(). Los programadores principantes aprenden rápidamente que si un método no tiene *senders* puede ser eliminado, pero asumen erróneamente que siempre y cuando un método tenga *senders* no debería ser eliminado. Esto no es cierto cuando los únicos *senders* de un método son otros implementadores del mismo mensaje, como es el caso con Object Recursion. A menos que haya otro método no polimórfico para empezar la recursión, ninguno de los implementadores será corrido y todos pueden ser eliminados.
2. *Definir el sucesor.* El *Recurser* necesita uno o más sucesores, pero el *Terminator* no los necesita. Si el *Terminator* implementa el enlace con su sucesor (usualmente heredándolo del *Handler*), ignora esa cadena cuando se implementan los mensajes handleRequest(). Si todos los mensajes del *Terminator* ignoran la cadena de sucesores, entonces su valor siempre será nulo y la cadena no es necesaria.

Código de Ejemplo

Veamos como implementar la igualdad recursivamente.

Todos los objetos, sin importar qué tan complejos sean, están implementados de objetos simples (en otras palabras, primitivas) como Enteros, Floats, Booleanos y Caracteres. Determinar la igualdad de dos objetos simples (del mismo tipo) es una tarea trivial hecha nativamente por el sistema operativo o la CPU. Por ejemplo:

```
5 == 5        // comparacion de enteros (true)
5.25 == 5.15  // comparacion de floats (false)
true == false // comparacion de booleanos (false)
'a' == 'b'    // comparacion de caracteres (false)
```

No hay recursión acá, pero estas comparaciones simples forman los casos base de la recursión.

Comparar dos Ordered Collection es casi así de simple: ¿son todos los elementos iguales? De esta forma dos cadenas son iguales y cada uno de sus caracteres son iguales. Por ejemplo, veamos la implementación de `String.equals(String)`:

```
public class String {
  private char value[];
  private int count;

  public boolean equals(String anotherString) {
    int n = count;
    if (n == anotherString.count) {
      char v1[] = value;
      char v2[] = anotherString.value;
      int i = 0;
      int j = 0;
      while (n-- != 0) {
        if (v1[i++] != v2[j++]) {
          return false;
        }
      }
      return true;
    }
    return false;
  }
}
```

Así, si cada uno de los caracteres en las cadenas son iguales, las cadenas son iguales. Para el propósito de implementar la igualdad recursivamente, un *string* es un objeto *Terminator*.

Ahora consideremos un objeto que guarda por separado el nombre y apellido de una persona. Dos nombres son iguales si sus nombres y apellidos son iguales:

```
public class NombreDePersona {
  private String nombre, apellido;

  public boolean equals (NombreDePersona otroNombreDePersona) {
    return   nombre.equals(otroNombreDePersona.nombre)
       && (apellido.equals(otroNombreDePersona.apellido));
  }
}
```

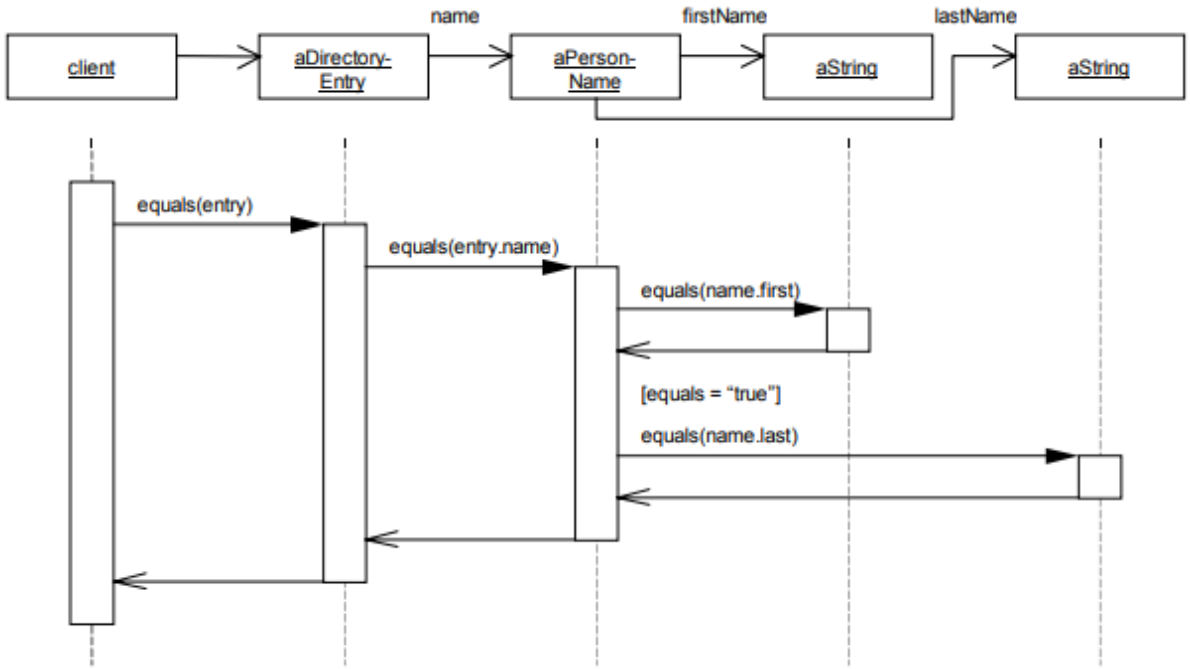
Esto contiene un nivel de recursión—`NombreDePersona.equals()` llama a `String.equals()`— mas la operación primitiva.

Ahora consideremos un objeto de una guía telefónica que guarda el nombre y teléfono de una persona. Dos entradas son consideradas duplicadas si sus nombres son iguales:

```
public class EntradaDeGuia {
  private NombreDePersona nombre;

  public boolean equals (EntradaDeGuia otraEntrada) {
    return nombre.equals(otraEntrada.nombre);
  }
}
```

Aquí tenemos dos niveles de recursión— `EntradaDeGuia.equals()` llama a `NombreDePersona.equals()`, que llama a `String.equals()` .



Al cliente no le importa realmente a cuál implementador de `equals()` está llamando, solo sabe que tiene dos objetos del mismo tipo así que los compara usando `equals()`. Si los dos objetos fueran simples no se necesitaría nada de recursión. Cuando los dos objetos son complejos, `equals()` es implementado recursivamente usando otros implementadores de `equals()` .

Usos Conocidos

Muchos ejemplos de programación usan Object Recursion.

El ejemplo de igualdad descripto en la Motivación usa recursión en un paso: la implementación recursiva del mensaje envía el mismo mensaje directamente a su sucesor. Cada implementador de `equal` también necesita un implementador correspondiente de `hash`, que también es implementado recursivamente.

Un mensaje de copiar o clonar se suele implementar usando recursión en dos pasos: la implementación recursiva le envía un segundo mensaje al receptor, el cual envía el mensaje original a su sucesor. Así los dos mensajes juntos implementan la recursión. En el caso de `copy()`, los dos mensajes son `simpleCopy()` —que solo copia la base del objeto, y `postCopy()` —que solo copia las partes del objeto cuando sean necesarias. `copy()` envía a `simpleCopy()` y `postCopy()` pero `postCopy()` , en cambio, envía `copy()` a las partes, propagando así la recursión.

Los algoritmos de serialización, ya sea que produzcan salida de texto o binaria, usualmente usan recursión. El algoritmo serializa un objeto serializando su base y después serializando recursivamente todas sus partes (persistentes). Cada rama de la recursión termina cuando un objeto simple se serializa a sí mismo y termina.

Un algoritmo para mostrar un objeto como un *string* (por ejemplo: el `toString()` de Java y el `printString` de Smalltalk) usualmente son recursivos. El algoritmo muestra la base como un *string* y después le dice a las partes relevantes que se muestren a sí mismas.

Una estructura de árbol puede usar recursión para pasar un mensaje desde alguna de sus hojas hasta su raíz. Cada nodo en el camino pasa recursivamente el mensaje a su padre, haciendo cualquier trabajo extra en el camino. Un árbol puede usar recursión similarmente para enviar un mensaje de la raíz, a través de todos sus nodos, a sus hojas. Estas técnicas recursivas se suelen usar en árboles gráficos, por ejemplo, para registrar pedidos de invalidación y transmitir oportunidades de redibujar.

Véase la sección de Patrones Relacionados para más usos conocidos.

Patrones Relacionados

Object Recursion vs. Composite y Decorator

Cuando un Decorator [GHJV95, p. 175] delega a su componente o un Composite [GHJV95, p. 163] delega a sus hijos, esto se podría considerar un ejemplo de Object Recursion. Pero Composite y Decorator son ejemplos de patrones estructurales (estructuras de datos) mientras que Object Recursion es un patrón de comportamiento (algorítmico). Si los patrones estructurales incorporan la recursión, es explícita solo en un nivel de profundidad (un *composite* o *decorator* delegando a su componente), mientras que la profundidad de la recursión es ilimitada. En el mejor caso Object Recursion es un patrón que ambos el Composite y Decorator contienen, pero que también se puede usar independientemente del Composite o Decorator.

Object Recursion vs. Cadena de Responsabilidad

La Cadena de Responsabilidad [GHJV95, p. 223] contiene el patrón de Object Recursion. La Cadena de Responsabilidad usa una lista enlazada o árbol organizado por especialización o prioridad. Cuando un pedido es hecho la estructura usa Object Recursion para encontrar un handler adecuado.

Object Recursion y Adapter

Una cadena de Adapters [GHJV95, p. 139] puede parecer que usa una forma no polimórfica de Object Recursion, donde el comportamiento es recursivo aunque el mensaje no lo sea, ya que cada Adapter se delega al siguiente hasta que la delegación termina en el objeto siendo "adaptado", y se retrocede. Pero la falta de polimorfismo va contra el espíritu de Object Recursion.

Object Recursion e Interpreter

En el Interpreter [GHJV95, p. 243] el mensaje `interpret()` viaja por el árbol de sintaxis abstracta usando Object Recursion. Client es el *Initiator*, AbstractExpression es el *Handler*, NonterminalExpression es el *Recurser* y TerminalExpression es el *Terminator*.

Object Recursion e Iterator

Algunas implementaciones de Iterator [GHJV95, p. 257] usan Object Recursion. Los iteradores externos en cualquier estructura no son recursivos, en cambio usan ciclos *while*. Los iteradores internos en estructuras de array también usan ciclos *while*.

Un iterador interno en una estructura de lista enlazada es recursivo. Si el nodo terminante de la lista es un objeto real con el mismo tipo que los otros nodos de la lista, ya que la iteración interna hace los mensajes `next()` e `isDone()` privados, el algoritmo es Object Recursion. Si el fin de la lista esta marcado por null, la recursión es procedural.

Un iterador interno en una estructura con ramas (en otras palabras, un árbol o grafo) debe ser implementado recursivamente. Si los puntos terminales—las hojas y/o la raíz—son objetos, entonces la recursión es Object Recursion.

Object Recursion y Delegación

Otros patrones contienen un objeto que implementa un mensaje delegando el mismo mensaje a un colaborador del mismo tipo: Proxy es un buen ejemplo. Cualquier ejemplo de esta forma es un ejemplo de Object Recursion a un solo nivel

de profundidad, el cual no es un ejemplo muy interesante pero que de todas formas es un ejemplo simple.

Referencias

- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns:
  Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

Agradecimientos

Gracias a Eugene Wallingford por su ayuda para escribir este documento.