Patrón de diseño: Objeto Nulo ~ Bobby Woolf

La siguiente es una traducción libre del paper *Null Object Pattern* publicado por Bobby Woolf en 1996, cuyo texto original puede encontrarse en el siguiente enlace: https://algoritmos-iii.github.io/assets/bibliografia/null-object-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

Proveer un sustituto para otro objeto que comparte la misma interfaz pero en realidad no hace nada. El Objeto Nulo encapsula las decisiones de implementación de cómo efectivamente "no hacer nada" y esconde esos detalles de sus colaboradores.

Otros nombres

Stub, Active Nothing

Motivación

A veces una clase que requiere un colaborador necesita que el colaborador no haga nada. Pero a la vez, la clase desea tratarlo de la misma manera que trata a uno que sí tenga comportamiento.

Consideremos por ejemplo el paradigma de Modelo-Vista-Controlador en Smalltalk-80. Una vista usa su controlador para recibir datos del usuario. Este es un ejemplo del patrón Strategy [GHJV95, página 315], donde se ve que el Controlador es la estrategia (*strategy*) de la Vista que sabe cómo obtener la información del usuario.

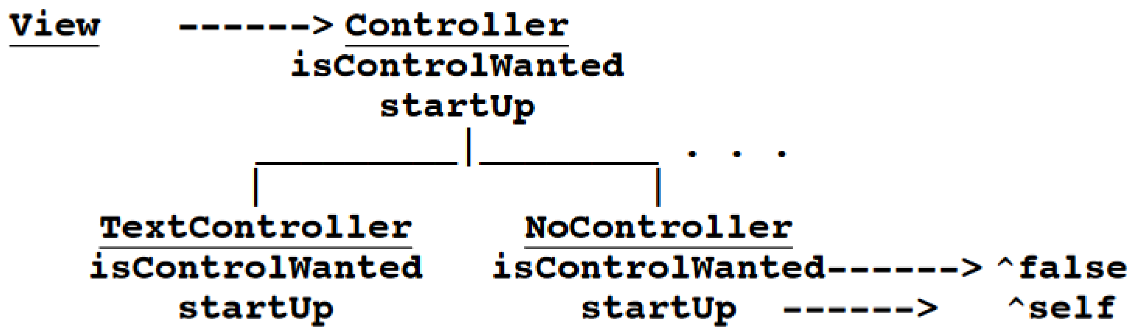
Una vista puede además ser de solo lectura y no obtener ningún dato del usuario. Y por ello, no requiere un controlador, sin embargo, la Vista y sus subclases están implementadas para recibir un controlador y usarlo

Si ninguna instancia o subclase de la Vista requiere un controlador, entonces esa subclase no necesita ser subclase de Vista. Podría estar implementada como una clase visual de forma similar a la Vista que no requiere un controlador. Sin embargo, esto no funciona en un escenario donde algunas subclases requieren un controlador y otras no. En ese caso, la clase sí necesitará ser subclase de Vista y todas sus instancias requerirán un controlador.

Una forma de resolver este problema sería setear el controlador de esa instancia como *nil.* Pero esto no funciona del todo porque la vista constantemente envía mensajes a su controlador que sólo los Controladores pueden responder (como *#isControlWanted* o *#startUp*). Como *UndefinedObject* (la clase a la que pertenece *nil)* no sabe responder estos mensajes de Controlador, la vista debería chequear si existe el controlador antes de usarlo y, en el caso de que no exista, también debería decidir qué hacer. Todo este código condicional ensuciaría un poco la implementación de la Vista. Incluso, si más de una vista necesitará que el controlador sea *nil*, el código condicional sería difícil de reutilizar. Por lo que, usar *nil* como controlador no funciona muy bien.

Otra posible solución podría ser usar un controlador de sólo lectura. Algunos controladores pueden configurarse en modo sólo lectura de forma tal que ignoren el input del usuario. Estos controladores siguen recibiendo información del usuario pero cuando están en sólo lectura procesan estos inputs no haciendo nada. Si estuvieran en modo edición, procesarían esos mismos inputs cambiando el estado del modelo. Esto es demasiado para un controlador que siempre va a ser sólo lectura. Se busca un controlador que no dependa de su modo actual para saber como procesar los inputs. Por lo que, un controlador que siempre es de solo lectura debería estar implementado para no hacer ningún procesamiento.

En su lugar, lo que necesitamos es un controlador específicamente implementado para ser de solo lectura. Esta subclase especial de controlador es llamada *NoControlador* (NoController). Implementa toda la lógica de un Controlador pero no hace nada. Cuando se le envía el mensaje *#isControlWanted* automáticamente responde que no. Cuando se le envía el mensaje *#startUp* no hace nada y se devuelve a sí mismo. Hace lo mismo que hacen otros controladores, pero lo hace no haciendo nada.



Este diagrama ilustra como una vista requiere un controlador y como ese controlador puede ser un NoControlador. El NoControlador implementa la misma lógica que los otros controladores pero lo hace sin hacer nada.

El NoControlador es un ejemplo del patrón de Objeto Nulo. El patrón describe cómo desarrollar una clase que encapsula como un tipo de objeto debería "no hacer nada". Porque el código de cómo "no hacer nada" está encapsulado y su complejidad está oculta para el colaborador y puede ser fácilmente reutilizada por cualquier controlador que la necesite.

La clave del patrón de Objeto Nulo es una clase abstracta que define la interfaz para todos los objetos de este tipo. El Objeto Nulo es implementado como una subclase más de esta clase abstracta. Cómo conserva la interfaz de la clase abstracta, puede usarse en cualquier lugar donde sea requerido este tipo de objeto.

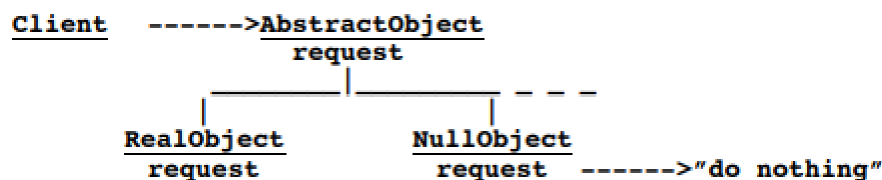
Aplicabilidad

El patrón Objeto Nulo puede usarse cuando:

- un objeto requiere de un colaborador. Este colaborador no es introducido por el patrón, sino que éste utiliza una colaboración que ya existía.
- instancias tienen algunos colaboradores que no hacen nada.
- se quiere que los clientes puedan ignorar la diferencia entre tratar con un colaborador que tiene comportamiento y uno que no hace nada. De esta forma, el cliente no tiene que verificar explícitamente si es nil u otro valor especial.
- se quiere la posibilidad de reutilizar el comportamiento de "no hacer nada", para que varios clientes que lo necesiten puedan funcionar de la misma manera consistentemente.
- todo comportamiento que pueda necesitar ser el comportamiento de no hacer nada está encapsulado dentro de la clase del colaborador. Si parte del

comportamiento de esta clase es el de no hacer nada, la mayoría o todo el comportamiento será de ese mismo tipo. [Coplein96]

Estructura



Participantes

- Cliente (Vista) - *Client (View)*
    - requiere de un colaborador.
- ObjetoAbstracto (Controlador) - *AbstractObject (Controller)*
    - declara la interfaz del colaborador del cliente.
    - implementa el comportamiento por defecto de la interfaz común a todas las clases, según sea apropiado.
- ObjetoReal (ControladorDeTexto) - *RealObject (TextController)*
    - define una subclase concreta del ObjetoAbstracto cuyas instancias tienen un comportamiento útil que el Cliente espera.
- ObjetoNulo (NoControlador) - *NullObject (NoController)*
    - provee de una interfaz idéntica a la del ObjetoAbstracto, de tal forma que el objeto nulo puede ser sustituido por un objeto real.
    - implementa su interfaz para "hacer nada". Lo que realmente significa hacer nada es subjetivo y depende del tipo de comportamiento que el Cliente espera. Algunas llamadas pueden ser respondidas haciendo algo que devuelva un resultado nulo.
    - cuando hay más de una forma de "hacer nada", puede que se requieran más de una clase del ObjetoNulo.

Colaboraciones

- Los clientes utilizan la interfaz de la clase ObjetoAbstracto para interactuar con sus colaboradores. Si el receptor es un ObjetoReal, entonces el mensaje se

responde con comportamiento real. En cambio, si el receptor es un ObjetoNulo, se responde haciendo nada o devolviendo por lo menos un resultado nulo.

Consecuencias

El patrón de diseño Objeto Nulo:

- define una jerarquía de clases que consiste de objetos reales y objetos nulos. Estos últimos pueden ser usados, en lugar de los objetos reales, cuando se espera que el objeto no haga nada. En el código del cliente, donde sea que se espere un objeto real, puede aceptarse también un objeto nulo.
- simplifica el código del cliente. Este puede tratar de igual manera, tanto a los colaboradores reales, como a los nulos. Normalmente, los clientes no saben (y tampoco debería importarles) si están tratando con un colaborador real o nulo. Al no tener la necesidad de escribir código que compruebe si el colaborador es nulo, se termina simplificando el código del cliente.
- encapsula el código de no hacer nada dentro del objeto nulo. Así, este código es fácil de encontrar. Además, las diferencias con las clases ObjetoAbstracto y ObjetoReal son evidentes. Puede implementarse de manera eficiente para que no haga nada, en lugar de imitar el comportamiento real pero terminar no haciendo nada. No necesita de variables que contengan valores nulos porque, o bien, estos pueden estar *hardcodeados* como constantes, o bien el código de "no hacer nada" puede evitar usar esos valores.
- facilita la reutilización del código de "no hacer nada" del objeto nulo. Puede que varios clientes, que necesitan que sus colaboradores no hagan nada, deseen que estos se comporten de la misma forma. Si este código tiene que ser modificado, sólo habría que cambiarlo en un sólo lugar. En consecuencia, todos los clientes continuarán usando el mismo comportamiento, ahora modificado.
- dificulta la distribución o combinación del comportamiento de "no hacer nada" dentro del comportamiento existente de algunos objetos colaboradores. Para algunas clases, no es tan fácil agregar el mismo código de "no hacer nada", a menos que estas puedan delegar este comportamiento a la clase del objeto nulo.
- puede precisar de la creación de una nueva clase de ObjetoNulo por cada nueva clase de ObjetoAbstracto.
- puede ser difícil de implementar si varios clientes no concuerdan en cómo el objeto nulo debería "hacer nada".

- siempre actúa como un objeto que no hace nada. El Objeto Nulo no se transforma en un Objeto Real.

Implementación

Hay varias aristas a considerar para implementar el patrón de Objeto Nulo:

1. *El Objeto Nulo como Singleton. El Objeto Nulo se implementa normalmente como un Singleton [GHJV95, pág. 127]. Como el Objeto Nulo normalmente no tiene estado, su estado no puede cambiar, por lo que múltiples instancias en realidad son idénticas. En lugar de usar varias instancias idénticas, el sistema podría usar la misma instancia repetidas veces.*
2. *Instancia nula especial de un Objeto Real. Como se menciona en las Consecuencias, el patrón de Objeto Nulo puede causar que una clase simple Objeto Real crezca a 3 clases: ObjetoAbstracto, ObjetoReal y ObjetoNulo. Incluso si toda la jerarquía puede implementarse solo con una clase ObjetoReal (y ninguna subclase), por lo menos una subclase es necesaria para implementar el ObjetoNulo. Una forma de evitar esta explosión de clases es implementar el objeto nulo como una instancia especial de ObjetoReal en lugar de una subclase de ObjetoAbstracto. Las variables en esta instancia nula tendrían valores nulos. Esto debería ser suficiente para causar que la instancia nula no haga nada. Por ejemplo, un objeto compuesto cuyo hijo es una lista vacía, actúa como un elemento hoja.*
3. *Distintos clientes no están de acuerdo con el comportamiento nulo. Si algunos clientes esperan que el objeto nulo "no haga nada" de una forma particular y otros de otra forma, se requerirán múltiples clases distintas de ObjetoNulo. Si el comportamiento de "no hacer nada" debe ser customizado en tiempo de ejecución, el la clase ObjetoNulo va a requerir variables inyectables para que el cliente pueda especificar cómo necesita que el ObjetoNulo no haga nada (ver la discusión de adaptadores inyectables en el patrón Adaptador (Adapter) [GHJV95, pág 142]). Nuevamente, una forma de evitar esta explosión de subclases de ObjetoNulo en una clase ObjetoAbstracto es hacer los objetos nulos como instancias especiales del ObjetoReal o de una única subclase ObjetoNulo. Si se usa una sola clase ObjetoNulo, su implementación puede volverse un ejemplo del patrón Objeto Ligero (Flyweight pattern) [GHJV95, pág. 195]. El comportamiento que todos los clientes esperan de un objeto nulo particular se convierte en el comportamiento intrínseco del objeto ligero y lo que cada cliente customiza es el comportamiento extrínseco del objeto ligero.*

4. *Transformación a ObjetoReal*. Un objeto nulo no se transforma en un ObjetoReal. Si el objeto puede decidir dejar de proveer comportamiento nulo y empezar a proveer comportamiento real, no es un objeto nulo. Puede ser un objeto real con un modo de “no hacer nada”, como un controlador que puede salir o entrar de modo de solo lectura. Si es un único objeto que debe mutar de un objeto nulo a un objeto real, debería estar implementado mediante un patrón Proxy [GHJV95, pág 207]. Quizás el proxy comienza usando un objeto nulo, luego cambia a usar un objeto real, o quizás, el comportamiento de no hacer nada está implementado en el proxy para cuando no tiene un objeto real. No se requiere un proxy si el cliente sabe que puede estar usando un colaborador nulo. En ese caso, el cliente puede tomar responsabilidad de cambiar el objeto nulo por uno real cuando sea necesario.
5. *El ObjetoNulo no es un Proxy. El uso de un objeto nulo puede ser similar al de un proxy [GHJV95, pág 207], pero estos patrones tienen propósitos diferentes. Un Proxy provee un nivel de indirección al acceder al sujeto real, controlando por ende este acceso. Un colaborador nulo no esconde un objeto real ni controla su acceso a él, reemplaza el objeto real. Un proxy podría eventualmente mutar y comenzar a actuar como un objeto real. Un objeto nulo nunca va a mutar para proveer comportamiento real, siempre va a tener el comportamiento de “no hacer nada”:*
6. *Objeto Nulo como una Estrategia especial. Un objeto nulo peude ser un caso especial del patrón Estrategia (Strategy)[GHJV95, 315]. La Estrategia especifica varias clases de EstrategiaConcreta como diferentes maneras de cumplir una misma tarea. Si una de esas maneras es consistentemente no hacer nada, esa EstrategiaConcreta es un Objeto Nulo. Por ejemplo, un Controlador es la Estrategia de la Vista para manejar inputs y NoControlador es la Estrategia que ignora todo input.*
7. *Objeto Nulo como un Estado especial. Un Objeto nulo puede ser un caso especial del patrón Estado (State) [GHJV95, pág 305]. Normalmente, cada EstadoConcreto tiene algunos mensajes de no hacer nada porque no son apropiados para ese estado, De hecho, cierto método es generalmente implementado para hacer algo útil en la mayoría de los estados pero no hacer nada en por lo menos un estado. Si un EstadoConcreto particular implementa la mayoría de sus mensajes no haciendo nada o al menos devolviendo resultados nulos, se convierte en un de no hacer nada y en un Objeto Nulo. Por ejemplo, el estado que representa un usuario que no está logueado solamente permite al usuario loguearse, por lo que es un estado nulo. [Wallingford96]*

8. *La clase ObjetoNulo no es una mezcla (mixin). El ObjetoNulo es una clase concreta que funciona como un colaborador para un cliente que necesita un colabordador. El comportamiento nulo no está diseñado para mezclarse dentro de un objeto que necesita algun comportamiento nulo. Está diseñado para una clase que delega a un colaborador todo el comportamiento que puede o no ser el comportamiento nulo. [Coplein96]*

Código de ejemplo

Para una implementación de ejemplo del patrón de Objeto Nulo, daremos una mirada a la implementación de la clase NullScope en VisualWorks Smalltalk (descripta en los Usos Conocidos).

NullScope es una clase especial en la jerarquía de NameScope. Un NameScope sabe como buscar por una variable con un nombre particular (*variableAt:from:*), una variable sin declarar (*undeclared:from:*), e iterar sobre sus variables (*namesAndValuesDo:*). (La forma en la que estos mensajes son enviados de un scope al siguiente es un ejemplo del patrón Cadena de Responsabilidades (Chain of Responsibility)[GHJV95, pág, 223].)

```
Object ()
  NameScope (outerScope)

NameScope>>variableAt:from:
      ^self subclassResponsibility

NameScope>>undeclared:from:
      ^outerScope undeclared: name from: varNode

NameScope>>namesAndValuesDo:
      self subclassResponsibility
```

Un StaticScope representa el scope de clase y variables globales. Un LocalScope representa el scope de variables de instancia y métodos. Ellos implementan *variableAt:from:* y *namesAndValuesDo:* de una forma bastante sencilla que es esencialmente la misma para ambas clases.

```
Object ()
  NameScope (outerScope)
    LocalScope (...)
    StaticScope (...)
LocalScope>>variableAt:from:
      "find and return the variable, or"
      ^outerScope variableAt: name from: varNode
LocalScope>>namesAndValuesDo:
      "iterate through the receiver's variables, then"
      outerScope namesAndValuesDo: aBlock
```

Un NullScope representa el scope más externo. Esto puede ser el scope externo al scope más global o el scope externo de un bloque limpio o de copia (un bloque que no tiene un contexto externo). Hereda la variable de instancia outerScope pero nunca la usa. Nunca contiene declaraciones de variables por lo que su código es bastante simple.

```
Object ()
  NameScope (outerScope)
    LocalScope (...)
    NullScope ()
    StaticScope (...)
NullScope>>variableAt:from:
      "There are no definitions here."
      ^nil
NullScope>>namesAndValuesDo:
      "Do nothing"
```

Lo que es más interesante acerca del NullScope es como implementa *undeclared:from:.* NameScope simplemente pasa la llamada a su scope externo, StaticScope y LocalScope heredan esta implementación. Por lo que ninguno de esas clases hace nada. Sin embargo, NullScope, implementa ese mensaje para devolver la variable del diccionario de variables sin declarar.

```
NullScope>>undeclared:from:
      "Find the variable in Undeclared and return it.
      If the variable is not in Undeclared, add it
      and return it."
```

Así es como las variables se convierten en variables sin declarar: Si *variableAt:from:* falla en encontrar la variable en alguno de los scopes, el cliente llama undeclared:from: para buscarla en Undeclared. Si aún no está en Undeclared, se agrega. La jerarquía debería encapsular esta decisión y esconderla del cliente implementando NullScope>>*variableAt:from:* para enviar undeclared:from:, pero el NullScope no puede hacer eso sin conocer al scope donde inició la búsqueda.

Es importante notar cómo el NullScope factoriza el código especial fuera de las clases reales de NameScope (StaticScope y LocalScope) y lo encapsula dentro de NullScope. Esto evita hacer tests especiales y hace la diferencia entre el comportamiento general (en NameScope) y el comportamiento especial (en NullScope) fácil de ver y al comportamiento especial fácil de reutilizar.

Usos Conocidos

NoControlador

NoControlador, la clase de ObjetoNulo en este ejemplo, es una clase de la jerarquía Controller de VisualWorks Smalltalk. [VW95]

NullDragMode

NullDragMode es una clase en la jerarquía DragMode en VisualWorks Smalltalk. Un DragMode se usa para implementar ubicación y arrastramiento de elementos visuales en el dibujador de ventanas. Las subclasesde DragMode representan diferentes formas en las que se puede realizar el arrastramiento. (La jerarquía de DragMode es un ejemplo del patrón Estrategia [GHJV95, página 315]). Por ejemplo, una instancia de CornerDragMode representa cuando uno de los puntos de cambio de tamaño de un elemento visual está siendo arrastrado, por lo que, el elemento visual debería permanecer en el mismo lugar pero su tamaño debería cambiar. Alternativamente, una instancia de SelectionDragMode representa que el elemento visual entero está siendo arrastrado; su tamaño debería permanecer fijo mientras que su posición debería acompañar el cursor.

El NullDragMode es la contraparte de CornerDragMode que representa un intento de cambiar de tamaño un elemento que no puede ser cambiado de tamaño (como un texto cuyo tamaño está determinado por los caracteres que contiene y su tamaño de fuente). Los diferentes DragMode implementan un mensaje *#dragObject:startingAt:inController* que procesa el movimiento de arrastre del mouse. Usa un bloque para controlar cómo se está haciendo el ararstre. En el NullDragMode, este método usa un bloque vacío que no hace nada. Por lo tanto, el

NullDragMode responde a los movimientos de arrastre del mouse no haciendo nada. [VW95]

NullInputManager

NullInputManager es una clase dentro de la jerarquía InputManager de VisualWorks Smalltalk. Un InputManager provee de una interfaz, que no depende de la plataforma, a los eventos relacionados con el manejo de la entrada *internacionalizada* (idiomas extranjeros). (Ya que esta contiene los recursos de la plataforma para darles una comportamiento estándar, esto es un ejemplo del patrón Adapter [GHJV95, página 142].) Subclases, tales como X11InputManager, representan plataformas específicas. NullInputManager representa una plataforma que no soporta la internacionalización. Los métodos que este implementa no hacen casi nada, mientras que sus contrapartes, los métodos del X11InputManager, trabajan realmente. [VW95]

NullScope

NullScope es una clase dentro de la jerarquía NameScope de VisualWorks Smalltalk. Un NameScope representa el alcance (*scope*) de un grupo particular de variables. El tipo de variable (global, de clase, o de método) define qué tipo de NameScope se utilizará. Por ejemplo, un StaticScope es asignado a variables globales y de clase, y un LocalScope a variables de instancia o temporales. Todo scope tiene un nivel externo. Esto se usa para acceder a variables cuyo alcance es mayor al nivel actual. Así, el compilador puede avisar al programador, si está declarando una variable con el mismo nombre de otra existente anteriormente declarada (generalmente en un scope externo). De esta manera, NameScopes forman un árbol, con el scope global como raíz, y a los scopes de clase como sus hijos, que a su vez tienen como hijos a los de métodos.

Sin embargo, como todos los scopes tienen un nivel externo, ¿cuál sería el del global? Es el NullScope, un scope que nunca contiene ninguna variable. Cuando se busca si una variable fue declarada, cada scope se fija en su nivel externo, hasta que se encuentra la declaración o se llega al NullScope. Este último sabe detener la búsqueda y responde que la variable aparentemente no fue declarada (dentro del alcance del código donde se inició la búsqueda). Esto puede tratarse como un caso especial de StaticScope, en el cual si el scope es global, entonces se espera que su nivel externo sea nil. En cambio, se puede escribir de una manera más limpia mediante la clase especial NullScope, permitiéndole ser reutilizado por bloques limpios o de copia (que al ser tan simples no necesitan de un scope externo). NullScope está implementado como un *Singleton*, ya que el sistema nunca necesita más de una instancia del mismo [GHJV95, página 127]. [VW95]

NullLayoutManager

La jerarquía LayoutManager en el Java AWT toolkit no tiene una clase de objeto nulo pero podría utilizar una como NullLayout. A un Contenedor (Container) se le puede asignar un LayoutManager (esto es un ejemplo del patrón *Strategy* [GHJV95, página 315]). Si un contenedor en particular no necesita de un LayoutManager, la variable puede cambiar a nil. Desafortunadamente, esto significa que el código de Container está repleto de muchos chequeos que verifican si el LayoutManager es nil. Este código se simplificaría si se usara un objeto nulo, como NullLayoutManager, en vez de nil. [Gamma96]

Null_Mutex

La clase NullMutex es un mecanismo de exclusión mutua en el framework ASX (Servicio ejecutivo adaptable) implementado en C++. Este framework provee varios mecanismos de control de concurrencia (por ejemplo Estrategias [GHJV95, página 315]). La clase Mutex define un lock no recursivo para un hilo que no se va a llamar a si mismo cuando el lock se establece. La clase RW_Mutex define un lock que permite múltiples hilos simultaneos para leer pero un solo hilo durante una escritura. La clase NullMutex define un lock para un servicio que siempre corre en un único hilo y que no tiene interacción con otros hilos. Por ello, el lock no es realmente necesario, la clase NullMutex no bloquea nada realmente, sus mensajes de adquirir y liberar no hacen nada. Esto evita el problema de adquirir locks cuando no son realmente necesarios [Schmidt94].

Null Lock

El NullLock es un tipo de lock (por ejemplo un Estado [GHJV95, página 305]) en VERSANT Object Database Management System. Tres de los modos de lock de VERSANT son lock de escritura, lock de lectura y lock nulo. Los locks de escritura bloquean otros locks de escritura y de lectura en el mismo objeto para que nadie pueda leer o cambiar el objeto mientras se está cambiando. Los locks de lectura bloquean los locks de escritura pero permiten que otros puedan leer el objeto mientras es está leyendo (pero no se puede cambiar).

El NullLock no bloquea otros locks ni puede ser bloqueado. Esto garantiza el acceso inmediato al objeto, incluso si alguien tiene un lock sobre el pero no garantiza que el objeto va a estar en un estado consistente al accederlo. El Null Lock no es realmente un lock porque no su mensaje lock no hace nada pero actúa como un lock para operaciones que requieren algún tipo de lock. [Versant95]

NullIterator

El patrón *Iterator* (Iterador) es un caso especial de *NullIterator* (IteradorNulo) [GHJV95, páginas 67-68 y 262]. Cada nodo de un árbol podría tener un iterador para sus hijos. Los nodos con hijos devolverían un iterador concreto, mientras que una hoja devolvería una instancia de NullIterator. Un iterador nulo siempre ya terminó de recorrerse, es decir, cuando se le pregunta si ya terminó (*#isDone)*, siempre devuelve *true*. De esta manera, el cliente puede usar el iterador a lo largo de toda la estructura, aún cuando no quedan más nodos.

Z-Node

Los lenguajes por procedimientos o procedimentales tienen un tipo de dato nulo, que es parecido a los objetos nulos. El *Z-Node* (Nodo-Z) de Sedgewick es un nodo ficticio que se usa como el final de una lista enlazada. Cuando un nodo de un árbol requiere un número fijo de hijos pero no tiene hijos suficientes, usa nodos-z para llenar los huecos de los hijos faltantes. En una lista, el nodo-z evita en el procedimiento de borrado la necesidad de hacer chequeo especial al borrar un elemento de una lista vacía. En un arbol binario, un nodo sin hijos necesitaría uno o dos links nulos pero se usa el nodo-z en su lugar. De esta forma, un algoritmo de búsqueda puede simplemente saltear las ramas de nodos-z; cuando se haya quedado sin ramas de nodo-z, sabría que el elemento buscado no se encontró. De esta manera, los nodos-z se usan para evitar chequeos especiales por un valor nulo. [Sedge88]

NULL Handler

El patrón de diseño *Decoupled Reference* (Referencia Desacoplada) nos permite acceder a objetos a través de *Handlers* (Manejadores), y de esta forma, se le oculta al cliente su verdadera ubicación. Cuando un cliente interactúa con un objeto que ya no está disponible, en vez de que el programa falle, el entorno devuelve un *NULL Handler* (Manejador Nulo). Este Handler actúa como cualquier otro, pero le responde al cliente lanzando excepciones o causando condiciones de error. [Weibel96]

Patrones relacionados

La clase ObjetoNulo puede ser implementada, generalmente, como un *Singleton* [GHJV95, página 127], ya que puede suceder que varias instancias no tengan que cambiar su estado interno y funcionen exactamente igual.

Cuando muchos objetos nulos son implementados como instancias de una sóla clase ObjetoNulo, estos pueden ser implementados como *Flyweights* [GHJV95, página 195].

ObjetoNulo suele usarse como una clase dentro de una jerarquía de clases de Estrategias (*Strategy*) [GHJV95, página 315]. Representa la estrategia de “no hacer nada”.

ObjetoNulo suele utilizarse como una clase dentro de una jerarquía de clases de Estado (*State*) [GHJV95, página 305]. Representa el estado en el cual el cliente no debería hacer nada.

ObjetoNulo puede ser un tipo especial de Iterador (*Iterator*) [GHJV95, página 257], que no itera sobre nada.

ObjetoNulo puede ser una clase especial dentro de una jerarquía de Adaptadores (*Adapters)* [GHJV95, página 142]. Mientras que un adaptador, normalmente, “envuelve” (*wrap*) otro objeto y convierte su interfaz, un adaptador nulo pretende hacer lo mismo pero realmente no adapta nada.

Bruce Anderson también escribió acerca del patrón Objeto Nulo, al que se refirió como “Active Nothing”. [Anderson95]

ObjetoNulo es un caso especial del patrón Valor Excepcional (*Exceptional Value*) según “The CHECKS Pattern Language” [Cunningham95]. Un Valor Excepcional es un caso especial del patrón Valor Completo (*Whole Value*), usado para representar circunstancias excepcionales. Este absorberá todos los mensajes o tendrá un Comportamiento Sin Importancia (*Meaningless Behavior*), que es otro patrón de diseño.

Referencias

[Anderson95] Anderson, Bruce. “Null Object.” Lista de emails referidos a la discusión de patrones de diseño de la UIUC [Universidad de Illinois Urbana-Champaign] (patterns@cs.uiuc.edu), Enero 1995.

[Coplein96] Coplein, James. Correspondencia por email.

[Cunningham95] Ward Cunningham, “The CHECKS Pattern Language of Information Integrity” en [PLoP95].

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

1995; http://www.aw.com/cp/Gamma.html.

[Gamma96] Gamma, Erich. Correspondencia por email.

[PLoP95] Coplien, James and Douglas Schmidt (editores). *Pattern Languages of Program*

Design. Addison-Wesley, Reading, MA, 1995; http://heg-school.aw.com/cseng/authors/coplien/patternlang/patternlang.html.

[Schmidt94] Schmidt, Douglas. “Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Application,” C++ Report. SIGS Publications, Vol. 6, No.

3, Julio 1994.

[Sedge88] Sedgewick, Robert. Algorithms. Addison-Wesley, Reading, MA, 1988.

[Versant95] VERSANT Concepts and Usage Manual. Versant Object Technology, Menlo Park, CA, 1995.

[Wallingford96] Eugene Wallingford. Correspondencia por email.

[Weibel96] Weibel, Peter. “The Decoupled Reference Pattern.” Presentado por EuroPLoP ’96.

[VW95] VisualWorks Release 2.5, ParcPlace-Digitalk, Inc., Sunnyvale, CA, 1995;

http://www.parcplace.com.