**El Patrón de Objeto Nulo**

***Intención***

Proporcionar un sustituto para otro objeto que comparte la misma interfaz, pero no hace nada. El Objeto Nulo encapsula las decisiones de implementación de cómo "no hacer nada" y oculta esos detalles a sus colaboradores.

También Conocido Como: Stub, Activo Nada

***Motivación***

A veces, una clase que requiere un colaborador no necesita que el colaborador haga nada. Sin embargo, la clase desea tratar a un colaborador que no hace nada de la misma manera que trata a uno que realmente proporciona un comportamiento.

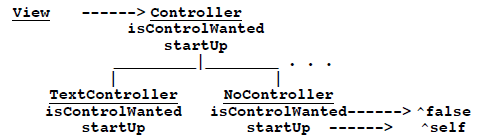
Considere, por ejemplo, el paradigma Modelo-Vista-Controlador en Smalltalk-80. Una Vista utiliza su Controlador para recopilar la entrada del usuario. Este es un ejemplo del patrón Estrategia, ya que el Controlador es la estrategia de la Vista para cómo recopilará la entrada.

Una vista puede ser de solo lectura. Dado que la vista no recopila la entrada del usuario, no necesita un controlador. Sin embargo, la Vista y sus subclases están implementadas para esperar un controlador y utilizan extensamente su controlador.

Si ninguna instancia de la clase "vista" nunca necesitara un controlador, entonces la clase no necesitaría ser una subclase de Vista. Podría implementarse como una clase visual similar a Vista que no requiriera un controlador. Sin embargo, esto no funcionará para una clase que tiene algunas instancias que requieren un controlador y algunas que no. En ese caso, la clase necesita ser una subclase de Vista y todas sus instancias requerirán un controlador. Por lo tanto, la clase de vista requiere un controlador pero una instancia particular no.

Una forma de resolver este problema sería establecer el controlador de la instancia en nil. Sin embargo, esto no funcionaría muy bien porque la vista envía constantemente mensajes a su controlador que solo el Controlador entiende (como isControlWanted y startUp). Dado que el ObjetoNoDefinido (clase nil) no entiende estos mensajes del Controlador, la vista tendría que verificar su controlador antes de enviar esos mensajes. Si el controlador fuera nil, la vista tendría que decidir qué hacer. Todo este código condicional desordenaría la implementación de la vista. Si más de una clase de vista pudiera tener nil como controlador, el código condicional para manejar nil sería difícil de reutilizar. Por lo tanto, usar nil como un controlador no funciona muy bien.

Otra forma de resolver este problema sería usar un controlador de solo lectura. Algunos controladores pueden configurarse en modo de solo lectura para que ignoren la entrada. Un controlador de este tipo aún recopilaría la entrada, pero cuando esté en modo de solo lectura, procesaría la entrada haciendo nada. Si estuviera en modo de edición, procesaría esa misma entrada cambiando el estado del modelo. Esto es excesivo para un controlador que siempre va a ser de solo lectura. Tal controlador no necesita realizar ningún procesamiento dependiendo de su modo actual. Su modo siempre es de solo lectura, por lo que no es necesario ningún procesamiento. Por lo tanto, un controlador que siempre sea de solo lectura debe codificarse para no realizar ningún procesamiento.

En su lugar, lo que necesitamos es un controlador específicamente codificado para ser de solo lectura. Esta subclase especial de Controlador se llama NoController. Implementa toda la interfaz de Controller, pero no hace nada. Cuando se le pregunta isControlWanted, automáticamente responde que no. Cuando se le indica que inicie, automáticamente no hace nada y devuelve self. Hace todo lo que hace un controlador, pero lo hace sin hacer nada.

Este diagrama ilustra cómo una vista requiere un controlador y cómo ese controlador puede ser un NoController. El NoController implementa todo el comportamiento que cualquier controlador hace, pero lo hace sin hacer nada.

NoController es un ejemplo del patrón Objeto Nulo. El patrón de Objeto Nulo describe cómo desarrollar una clase que encapsula cómo un tipo de objeto no debe hacer nada. Dado que el código de no hacer nada está encapsulado, su complejidad está oculta para el colaborador y puede ser fácilmente reutilizado por cualquier colaborador que lo 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 se implementa como una subclase de esta clase abstracta. Dado que se ajusta a la interfaz de la clase abstracta, se puede usar en cualquier lugar donde se necesite este tipo de objeto.

**Aplicabilidad**

Use el patrón de Objeto Nulo cuando:

- un objeto requiere un colaborador. El patrón de Objeto Nulo no introduce esta colaboración, hace uso de una colaboración que ya existe.

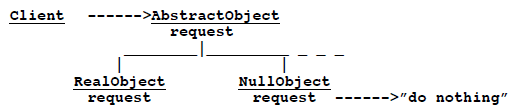
- algunas instancias del colaborador

Quieres que los clientes puedan ignorar la diferencia entre un colaborador que proporciona un comportamiento real y aquel que no hace nada. De esta manera, el cliente no tiene que verificar explícitamente si es nulo o algún otro valor especial.

- Quieres poder reutilizar el comportamiento de no hacer nada para que varios clientes que necesiten este comportamiento funcionen de manera consistente.

- Todo el comportamiento que podría necesitar ser de no hacer nada está encapsulado dentro de la clase colaboradora. Si parte del comportamiento en esa clase es de no hacer nada, la mayor parte o todo el comportamiento de la clase será de no hacer nada.

**Estructura**



**Participantes:**

- Cliente (Vista): requiere un colaborador.

- ObjetoAbstracto (Controlador): declara la interfaz para el colaborador del Cliente. implementa el comportamiento predeterminado para la interfaz común a todas las clases, según corresponda.

- ObjetoReal (ControladorTexto): define una subclase concreta de ObjetoAbstracto cuyas instancias proporcionan un comportamiento útil que el Cliente espera.

- ObjetoNulo (SinControlador): proporciona una interfaz idéntica a la de ObjetoAbstracto para que un objeto nulo pueda ser sustituido por un objeto real. implementa su interfaz para no hacer nada o al menos proporcionar un resultado nulo. cuando hay más de una manera de no hacer nada, puede ser necesario más de una clase ObjetoNulo.

**Colaboraciones:**

- Los Clientes usan la interfaz de la clase ObjetoAbstracto para interactuar con sus colaboradores. Si el receptor es un ObjetoReal, entonces la solicitud se maneja para proporcionar un comportamiento real. Si el receptor es un ObjetoNulo, la solicitud se maneja no haciendo nada o al menos proporcionando un resultado nulo.

**Consecuencias:**

- El patrón del Objeto Nulo

- define jerarquías de clases que consisten en objetos reales y objetos nulos. Los objetos nulos pueden usarse en lugar de objetos reales cuando se espera que el objeto no haga nada. Siempre que el código del cliente espere un objeto real, también puede tomar un objeto nulo.

- simplifica el código del cliente. Los clientes pueden tratar a los colaboradores reales y nulos de manera uniforme. Normalmente, los clientes no saben (y no deberían preocuparse) si están tratando con un colaborador real o nulo. Esto simplifica el código del cliente, ya que evita tener que escribir código de prueba que maneje el colaborador nulo de manera especial.

- encapsula el código de no hacer nada en el objeto nulo. El código de no hacer nada es fácil de encontrar. Su variación con las clases ObjetoAbstracto y ObjetoReal es evidente. Puede codificarse eficientemente para no hacer nada, en lugar de tener que pasar por el proceso de hacer algo, pero en última instancia no hacer nada. No requiere variables que contengan valores nulos porque esos valores pueden codificarse como constantes o el código de no hacer nada puede evitar usar esos valores por completo.

- hace que el código de no hacer nada en el objeto nulo sea fácil de reutilizar. Varios clientes que necesiten que sus colaboradores no hagan nada harán todos lo mismo. Si el comportamiento de no hacer nada necesita modificarse, el código se puede cambiar en un solo lugar. A partir de entonces, todos los clientes seguirán utilizando el mismo comportamiento de no hacer nada, que ahora es el comportamiento de no hacer nada modificado.

- hace que el comportamiento de no hacer nada sea difícil de distribuir o mezclar en el comportamiento real de varios objetos colaboradores. El mismo comportamiento de no hacer nada no se puede agregar fácilmente a varias clases a menos que esas clases deleguen el comportamiento en una clase que pueda ser una clase de objeto nulo.

- puede necesitar crear una nueva clase ObjetoNulo para cada nueva clase ObjetoAbstracto.

- puede ser difícil de implementar si varios clientes no están de acuerdo en cómo debería hacer nada el objeto nulo.

- siempre actúa como un objeto de no hacer nada. El objeto nulo no se transforma en un objeto real.

**Implementación:**

- Hay varios problemas a considerar al implementar el patrón del Objeto Nulo:

1. ObjetoNulo como Singleton(Único). La clase ObjetoNulo a menudo se implementa como un Singleton. Dado que un objeto nulo generalmente no tiene ningún estado, su estado no puede cambiar, por lo que múltiples instancias son idénticas. En lugar de usar múltiples instancias idénticas, el sistema puede usar una sola instancia repetidamente.

2. Instancia nula especial del Objeto Real. Como se mencionó en las Consecuencias, el patrón del Objeto Nulo puede hacer que una sola clase Objeto Real se convierta en tres clases: ObjetoAbstracto, ObjetoReal y ObjetoNulo. Por lo tanto, incluso si toda la jerarquía de objetos abstractos se puede implementar con una sola clase ObjetoReal (y sin subclases), al menos se requiere una subclase para implementar la clase ObjetoNulo. Una forma de evitar esta explosión de clases es implementar el objeto nulo como una instancia especial de ObjetoReal en lugar de como una subclase de ObjetoAbstracto. Las variables en esta instancia nula tendrían valores nulos. Esto puede ser suficiente para que la instancia nula haga nada. Por ejemplo, un objeto compuesto cuyos hijos son una lista vacía actúa como un objeto hoja.

3. Los clientes no están de acuerdo sobre el comportamiento nulo. Si algunos clientes esperan que el objeto nulo no haga nada de una manera y otros de otra, se requerirán múltiples clases ObjetoNulo. Si el comportamiento de no hacer nada debe personalizarse en tiempo de ejecución, la clase ObjetoNulo requerirá variables enchufables para que el cliente pueda especificar cómo debería hacer nada el objeto nulo (consulte la discusión de adaptadores enchufables en el patrón Adaptador [GHJV95, página 142]). Nuevamente, una forma de evitar esta explosión de subclases ObjetoNulo de una sola clase ObjetoAbstracto es hacer que los objetos nulos sean instancias especiales de ObjetoReal o una sola subclase ObjetoNulo. Si se utiliza una sola clase ObjetoNulo, su implementación puede convertirse en un ejemplo del patrón Peso Ligero [GHJV95, página 195]. El comportamiento que todos los clientes esperan de un objeto nulo particular se convierte en el comportamiento intrínseco del peso ligero y lo que cada cliente personaliza es el comportamiento extrínseco del peso ligero.

4. Transformación a Objeto Real. Un Objeto Nulo no se transforma en un Objeto Real. Si el objeto puede decidir dejar de proporcionar un comportamiento de no hacer nada y comenzar a proporcionar un comportamiento real, entonces no es un objeto nulo. Puede ser un objeto real con un modo de no hacer nada, como un controlador que puede alternar entre el modo de solo lectura y el modo normal. Si es un objeto único que debe cambiar de ser un objeto de no hacer nada a uno real, debería implementarse con el patrón Proxy. Tal vez el proxy comience usando un objeto nulo y luego cambie a usar un objeto real, o tal vez el comportamiento de no hacer nada esté implementado en el proxy para cuando no tenga un sujeto. El proxy no es necesario si el cliente es consciente de que puede estar usando un colaborador nulo. En este caso, el cliente puede tomar la responsabilidad de cambiar el objeto nulo por uno real cuando sea necesario.

5. Objeto Nulo no es un Proxy. El uso de un objeto nulo puede ser similar al de un Proxy, pero los dos patrones tienen propósitos diferentes. Un proxy proporciona un nivel de indirección al acceder a un sujeto real, controlando así el acceso al sujeto. Un colaborador nulo no oculta un objeto real y controla el acceso a él, lo reemplaza. Un proxy puede eventualmente mutar para comenzar a actuar como un sujeto real. Un objeto nulo no mutará para comenzar a proporcionar un comportamiento real, siempre proporcionará un comportamiento de no hacer nada.

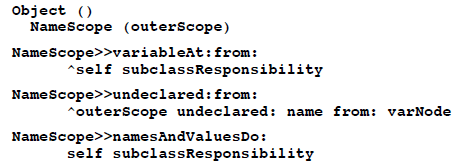
6. Objeto Nulo como Estrategia Especial. Un Objeto Nulo puede ser un caso especial del patrón Estrategia. La Estrategia específica varias clases de EstrategiaConcreta como diferentes enfoques para realizar una tarea. Si uno de esos enfoques consiste en hacer consistentemente nada, esa EstrategiaConcreta es un ObjetoNulo. Por ejemplo, un Controlador es la Estrategia de una Vista para manejar la entrada, y NoControlador es la Estrategia que ignora toda la entrada.

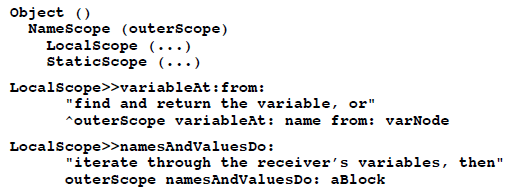
7. Objeto Nulo como Estado Especial. Un Objeto Nulo puede ser un caso especial del patrón Estado. Normalmente, cada EstadoConcreto tiene algunos métodos de no hacer nada porque no son apropiados para ese estado. De hecho, un método dado a menudo se implementa para hacer algo útil en la mayoría de los estados, pero para no hacer nada en al menos un estado. Si un EstadoConcreto particular implementa la mayoría de sus métodos para no hacer nada o al menos dar resultados nulos, se convierte en un estado de no hacer nada y como tal es un Objeto Nulo. Por ejemplo, el estado que representa a un usuario que no ha iniciado sesión permite al usuario no hacer nada excepto iniciar sesión, por lo que es un estado nulo.

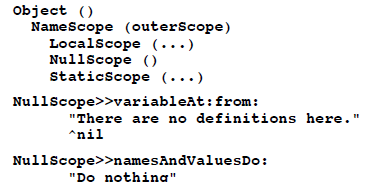
8. La clase Objeto Nulo no es una mezcla. El Objeto Nulo es una clase colaboradora concreta que actúa como el colaborador para un cliente que lo necesita. El comportamiento nulo no está diseñado para mezclarse en un objeto que necesita algún comportamiento de no hacer nada. Está diseñado para una clase que delega en un colaborador todo el comportamiento que puede o no ser un comportamiento de no hacer nada.

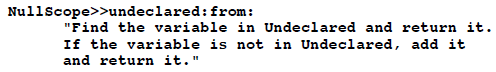
**Código de Ejemplo**

Para una implementación de ejemplo del patrón Objeto Nulo, veamos la implementación de la clase NullScope en VisualWorks Smalltalk (descrita en los Usos Conocidos).

NullScope es una clase especial en la jerarquía de NameScope. Un NameScope sabe cómo buscar una variable con un nombre particular (variableAt:from:), una variable no declarada (undeclared:from:), e iterar a través de sus variables (namesAndValuesDo:). (La forma en que estos mensajes se pasan de un ámbito a otro es un ejemplo del patrón Cadena de Responsabilidad.

Un StaticScope representa el ámbito para variables de clase y globales. Un LocalScope representa el ámbito para variables de instancia y variables de método. Implementan variableAt: from: y namesAndValuesDo: de una manera bastante directa que es esencialmente la misma para ambas clases.

Un NullScope representa el ámbito exterior. Este es el ámbito exterior de la mayoría global o el ámbito exterior para un bloque limpio o de copia (un bloque que no tiene un contexto exterior). Hereda la variable de instancia outerScope pero nunca la utiliza. Nunca contiene las declaraciones de ninguna variable, por lo que su código es bastante simple.

Lo más interesante de NullScope es cómo implementa undeclared:from:. NameScope simplemente pasa la solicitud a su ámbito exterior. StaticScope y LocalScope heredan esta implementación. Así que ninguna de esas clases hace nada. Pero NullScope implementa el método para devolver la variable del diccionario de variables no declaradas.

Así es como las variables se convierten en no declaradas: Si variableAt:from: no encuentra la variable en ninguno de los ámbitos, el cliente llama a undeclared:from: para encontrarla en Undeclared. Si no está en Undeclared, se agrega. La jerarquía debería encapsular esta decisión y ocultarla del cliente implementando NullScope>>variableAt:from: para enviar undeclared:from:, pero NullScope no puede hacer eso sin saber desde qué ámbito se inició originalmente la búsqueda.

Observa cómo NullScope factoriza el código especial fuera de las clases NameScope reales (StaticScope y LocalScope) y las encapsula en NullScope. Esto evita pruebas especiales, hace que la diferencia entre el comportamiento general (en NameScope) y el comportamiento especial (en NullScope) sea fácil de ver, y facilita la reutilización del comportamiento especial.

**Usos Conocidos**

NoController: la clase de Objeto Nulo en el ejemplo motivador, es una clase en la jerarquía de Controladores de VisualWorks Smalltalk.

NullDragMode: es una clase en la jerarquía de DragMode en VisualWorks Smalltalk. Un DragMode se utiliza para implementar la colocación y arrastre de visuales en el pintor de ventanas. Las subclases representan diferentes formas en que se puede hacer el arrastre. (La jerarquía de DragMode es un ejemplo del patrón de Estrategia. Por ejemplo, una instancia de CornerDragMode representa que se está arrastrando uno de los manejadores de cambio de tamaño visual, por lo que el visual debería mantenerse en el mismo lugar pero su tamaño debería cambiar. Alternativamente, un SelectionDragMode significa que se está arrastrando todo el visual, por lo que su tamaño debería permanecer fijo pero su posición debería seguir al ratón.

Un NullDragMode es un contraparte de CornerDragMode que representa un intento de cambiar el tamaño de un visual que no se puede cambiar de tamaño (como una etiqueta de texto, cuyo tamaño fijo está determinado por los caracteres que contiene y su tamaño de fuente). Los varios modos de arrastre implementan un método, dragObject:startingAt:inController:, que procesa el movimiento de arrastre del ratón. Utiliza un bloque para controlar cómo se realiza el arrastre. En NullDragMode, este método utiliza un bloque vacío que no hace nada. Así que un NullDragMode responde a los movimientos de arrastre del ratón haciendo nada. [VW95]

NullInputManager: es una clase en la jerarquía de InputManager en VisualWorks Smalltalk. Un InputManager proporciona una interfaz de objeto, neutral para la plataforma, a eventos de plataforma que pueden afectar el manejo de la entrada internacionalizada (en idiomas extranjeros). (Dado que envuelve los recursos de la plataforma para darles una interfaz de objeto estándar, este es un ejemplo del patrón de Adaptador [GHJV95, página 142]). Las subclases como X11InputManager representan plataformas específicas. NullInputManager representa plataformas que no admiten internacionalización. Los métodos que implementa hacen poco o nada, mientras que sus contrapartes en X11InputManager hacen un trabajo real. [VW95]

NullScope: es una clase en la jerarquía de NameScope en VisualWorks Smalltalk. Un NameScope representa el ámbito de un conjunto particular de variables. El tipo de variable que sea (global, de nivel de clase o de método) define qué tipo de NameScope usará. Por ejemplo, un StaticScope se asigna a variables globales y de clase y un LocalScope se asigna a variables de instancia y variables temporales. Cada ámbito tiene un ámbito exterior. Esto se utiliza para acceder a variables cuyo ámbito es mayor que el nivel actual. Permite al compilador advertir al programador si está declarando una variable con el mismo nombre que otra variable que ya ha sido declarada (generalmente en un ámbito exterior). Así que los NameScopes forman un árbol, con el ámbito global en la raíz y ramas para ámbitos de clase que contienen ramas para ámbitos de método.

Sin embargo, dado que todos los ámbitos tienen un ámbito exterior, ¿cuál es el ámbito exterior del ámbito global? Es un NullScope, un ámbito que nunca contiene variables. Al buscar una declaración de variable, cada ámbito sigue buscando en su ámbito exterior hasta que encuentra la declaración o hasta que alcanza un NullScope. NullScope sabe detener la búsqueda y responder que aparentemente la variable no ha sido declarada (dentro del ámbito del código que inició la búsqueda). Esto podría manejarse como un caso especial en StaticScope, que si es el ámbito global, entonces debería esperar que su ámbito exterior sea nulo, pero el caso especial está codificado de manera más limpia en la clase especial NullScope. Esto permite que NullScope se reutilice en bloques limpios y de copia, aquellos que son tan simples que no tienen ámbito exterior. NullScope se implementa como un Singleton porque el sistema nunca necesita más de una instancia [GHJV95, página 127]. [VW95]

NullLayoutManager: La jerarquía de LayoutManager en el kit de herramientas de Java AWT no tiene una clase de objeto nulo, pero podría usar una como NullLayout. A un Contenedor se le puede asignar un LayoutManager (un ejemplo del patrón de Estrategia [GHJV95, página 315]). Si un Contenedor en particular no requiere un LayoutManager, la variable se puede establecer en nulo. Desafortunadamente, esto significa que el código del Contenedor está lleno de muchas comprobaciones para un LayoutManager nulo. El código del Contenedor sería más simple si usara un objeto nulo como NullLayoutManager en lugar de nulo. [Gamma96]

Null\_Mutex: La clase Null\_Mutex es un mecanismo de exclusión mutua en el marco ASX (ADAPTIVE Service eXecutive) implementado en C++. El marco proporciona varios mecanismos (por ejemplo, Estrategias [GHJV95, página 315]) para el control de concurrencia. La clase Mutex define un bloqueo no recursivo para un hilo que no se llamará a sí mismo mientras se establece el bloqueo. La clase RW\_Mutex define un bloqueo que permite múltiples hilos simultáneos para lectura pero solo un hilo durante la escritura. La clase Null\_Mutex define un bloqueo para un servicio que siempre se ejecuta en un solo hilo y no compite con otros hilos. Dado que el bloqueo no.

Bloqueo Nulo: El Bloqueo Nulo es un tipo de modo de bloqueo (por ejemplo, Estado en el Sistema de Gestión de Bases de Datos de Objetos VERSANT. Tres de los modos de bloqueo que VERSANT utiliza son el bloqueo de escritura, el bloqueo de lectura y el bloqueo nulo. El bloqueo de escritura bloquea otros bloqueos de escritura y bloqueos de lectura en el mismo objeto para que nadie más pueda leer o cambiar el objeto mientras lo estás cambiando. El bloqueo de lectura bloquea los bloqueos de escritura pero permite otros bloqueos de lectura para que otras personas puedan leer el objeto mientras lo estás leyendo, pero no pueden cambiarlo. El bloqueo nulo no bloquea otros bloqueos y no puede ser bloqueado por otros bloqueos. Por lo tanto, garantiza acceso inmediato al objeto, incluso si alguien más ya lo ha bloqueado, pero no te garantiza que el objeto esté en un estado consistente cuando lo accedas. El bloqueo nulo no es realmente un bloqueo porque no realiza ningún bloqueo, pero actúa como un bloqueo para operaciones que requieren algún tipo de bloqueo.

NullIterator: El patrón del Iterador documenta un caso especial llamado NullIterator [GHJV95, páginas 67-68 y 262].

Cada nodo en un árbol podría tener un iterador para sus hijos. Los nodos compuestos devolverían un iterador concreto, pero los nodos hoja devolverían una instancia de NullIterator. Un NullIterator siempre se realiza con recorrido; cuando se le pregunta si está hecho, siempre devuelve verdadero. De esta manera, un cliente siempre puede usar un iterador para recorrer los nodos en una estructura incluso cuando no hay más nodos.

Z-Nodo: Los lenguajes de programación procedural tienen tipos de datos nulos que son como objetos nulos. El nodo Z de Sedgewick es un nodo ficticio que se utiliza como el último nodo en una lista enlazada. Cuando un nodo de árbol requiere un número fijo de nodos secundarios pero no tiene suficientes hijos, utiliza nodos Z como sustitutos de los hijos faltantes. En una lista, el nodo Z protege el procedimiento de eliminación de la necesidad de una prueba especial para eliminar un elemento de una lista vacía. En un árbol binario, un nodo sin dos hijos necesitaría uno o dos enlaces nulos, pero se utiliza el nodo Z nulo en su lugar. De esta manera, un algoritmo de búsqueda puede simplemente saltar las ramas de nodo Z; cuando se haya quedado sin ramas que no sean de nodo Z, sabrá que la búsqueda no encontró el elemento. De esta manera, los nodos Z se utilizan para evitar pruebas especiales de la misma manera que los objetos nulos.

Manejador NULL: El patrón de Referencia Desacoplada muestra cómo acceder a objetos a través de Manjedores para que su verdadera ubicación esté oculta para el cliente. Cuando un cliente solicita un objeto que ya no está disponible, en lugar de permitir que el programa falle, el marco devuelve un Manejador NULL. Este Manejador actúa como otros Manejadores, pero satisface las solicitudes generando excepciones o causando condiciones de error.

Patrones Relacionados: La clase NullObject generalmente puede implementarse como un Singleton [GHJV95, página 127] ya que múltiples instancias actuarían exactamente iguales y no tendrían ningún estado interno que pudiera cambiar.

Cuando se implementan múltiples objetos nulos como instancias de una sola clase NullObject, pueden implementarse como Flyweights [GHJV95, página 195].

NullObject se usa a menudo como una clase en una jerarquía de clases Strategy [GHJV95, página 315]. Representa la estrategia de no hacer nada.

NullObject se usa a menudo como una clase en una jerarquía de clases State [GHJV95, página 305]. Representa el estado en el que el cliente no debe hacer nada.

NullObject puede ser un tipo especial de Iterador [GHJV95, página 257] que no itera sobre nada.

NullObject puede ser una clase especial en una jerarquía de Adaptadores [GHJV95, página 142]. Mientras que un adaptador normalmente envuelve otro objeto y convierte su interfaz, un adaptador nulo fingiría envolver otro objeto sin envolver realmente nada.

Bruce Anderson también ha escrito sobre el patrón Null Object, al que también se refiere como "Active Nothing" [Anderson95].

NullObject es un caso especial del patrón de Valor Excepcional en el Lenguaje de Patrones CHECKS [Cunningham95]. Un Valor Excepcional es un Valor Completo especial (otro patrón) utilizado para representar circunstancias excepcionales. Absorberá todos los mensajes o producirá un Comportamiento Sin Significado (otro patrón). Un NullObject es uno de esos Valores Excepcionales.