**El Patrón de Objeto Tipo (Type Object)**

***Intención***

Desacoplar las instancias de sus clases para que esas clases puedan implementarse como instancias de una clase. El Objeto Tipo permite crear nuevas "clases" dinámicamente en tiempo de ejecución, permite que un sistema proporcione sus propias reglas de verificación de tipo y puede conducir a sistemas más simples y pequeños.

*También Conocido Como:* Tipo Potencia [MO95], Descriptores de Ítems [Coad93], Metaobjeto [KRB91]

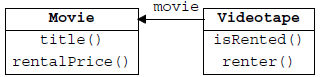
***Motivación***

A veces, una clase no solo requiere un número desconocido de instancias, sino también un número desconocido de subclases. Aunque un sistema de objetos puede crear nuevas instancias según demanda, generalmente no puede crear nuevas clases sin volver a compilar. Un diseño en el que una clase tiene un número desconocido de subclases puede convertirse en uno en el que la clase tiene un número desconocido de instancias.

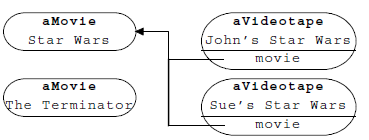
Considera un sistema para el seguimiento de las cintas de video en el inventario de una tienda de alquiler de videos. Obviamente, el sistema requerirá una clase llamada "Cinta de Video". Cada instancia de Cinta de Video representará una de las cintas de video en el inventario de la tienda. Sin embargo, dado que muchas de las cintas de video son muy similares, las instancias de Cinta de Video contendrán mucha información redundante. Por ejemplo, todas las copias de Star Wars tienen el mismo título, precio de alquiler, calificación MPAA1, etc. Esta información es diferente para The Terminator, pero todas las copias de The Terminator tienen los mismos datos. Por lo tanto, repetir esta información a través de todas las copias de Star Wars o todas las copias de The Terminator es redundante.

Una forma de resolver este problema es crear una subclase de Cinta de Video para cada película. Por lo tanto, dos de las subclases serían StarWarsTape y TerminatorTape. La clase en sí misma mantendría la información para esa película. Entonces, la información común a todas las copias de Star Wars se almacenaría solo una vez. Podría codificarse en el lado de la instancia de StarWarsTape o almacenarse en variables en el lado de la clase o en un objeto asignado a la clase para este propósito. Ahora, Cinta de Video sería una clase abstracta; el sistema no crearía instancias de ella. En cambio, cuando la tienda comprara una nueva cinta de video y comenzara a alquilarla, el sistema crearía una instancia de la clase para esa película. Entonces, si la nueva cinta de video fuera para The Terminator, el sistema crearía una instancia de TerminatorTape.

Esta solución funciona, pero no muy bien. Un problema es que si la tienda tiene muchas diferentes películas, Cinta de Video podría requerir un gran número de subclases. Otro problema es qué sucedería cuando el sistema se implementa y la tienda comienza a almacenar una nueva película, tal vez Independence Day. No hay una clase IndependenceDayTape en el sistema. Si el desarrollador no predijo esta situación, tendría que modificar el código para agregar una nueva clase IndependenceDayTape, recompilar el sistema y volver a implementarlo. Si el desarrollador predijo esta situación, podría proporcionar una subclase especial de Cinta de Video, como UnknownTape, y la tienda crearía una instancia de ella para todas las cintas de video de la nueva película. El problema con UnknownTape es que tiene la misma falta de flexibilidad que tenía Cinta de Video. Así como Cinta de Video requería subclases, también las requerirá UnknownTape, por lo que UnknownTape no es una solución muy buena.

En cambio, dado que el número de tipos de cintas de video es desconocido, cada tipo de cinta de video necesita ser una instancia de una clase. Sin embargo, cada cinta de video necesita ser una instancia de un tipo de cinta de video. Los lenguajes de objetos basados en clases brindan soporte para instancias de clases, pero no brindan soporte para instancias de instancias de clases. Entonces, para implementar esta solución en un lenguaje de programación basado en clases típico, necesitas implementar dos clases: una para representar un tipo de cinta de video (Película) y otra para representar una cinta de video (Cinta de Video). Cada instancia de Cinta de Video tendría un puntero a su correspondiente instancia de Película.

Este diagrama de clases ilustra cómo cada instancia de Cinta de Video tiene una instancia correspondiente de Película. Muestra cómo se separan las propiedades definidas por el tipo de cinta de video de aquellas que difieren para cada cinta de video en particular. En este caso, el título de la película y el costo de alquiler se separan de si la cinta está alquilada y quién la está alquilando actualmente.

Este diagrama de instancias muestra cómo hay una instancia de Película para representar cada tipo de cinta de video y una instancia de Cinta de Video para representar cada video que la tienda tiene en stock. Star Wars y The Terminator son películas; las cintas de video son las copias de Star Wars que John está alquilando frente a las que Sue está alquilando. También muestra cómo cada Cinta de Video sabe qué tipo es debido a su relación con una instancia particular de Película.

Si se alquilara una nueva película, como Independence Day, a Jack, el sistema crearía una nueva Película y una nueva Cinta de Video que apuntaría a la Película. La película es Independence Day y la cinta es la copia de Independence Day que Jack termina alquilando.

Cinta de Video, Película y la relación de es-instancia-de entre ellas (una Cinta de Video es una instancia de una Película) es un ejemplo del patrón Objeto Tipo. Se utiliza para crear instancias de un conjunto de clases cuando el número de clases es desconocido. Permite a una aplicación crear nuevas "clases" en tiempo de ejecución porque las clases son realmente instancias de una clase. La aplicación debe mantener entonces la relación entre las instancias reales y sus instancias tipo-clase. La clave del patrón Objeto Tipo son dos clases concretas, una cuyas instancias representan las instancias de la aplicación y otra cuyas instancias representan tipos de instancias de aplicación. Cada instancia de la aplicación tiene un puntero a su tipo correspondiente.

***Aplicabilidad***

Utiliza el patrón de Objeto Tipo cuando:

- Las instancias de una clase necesitan agruparse según sus atributos y/o comportamiento común.

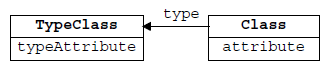
- La clase necesita una subclase para cada grupo para implementar los atributos/comportamiento común de ese grupo.

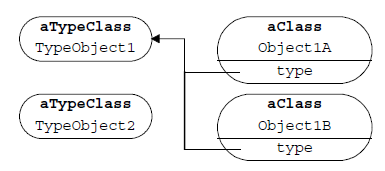
- La clase requiere un gran número de subclases y/o la variedad total de subclases que pueden ser necesarias es desconocida.

- Deseas poder crear nuevas agrupaciones en tiempo de ejecución que no fueron previstas durante el diseño.

- Deseas poder cambiar la subclase de un objeto después de que se haya instanciado sin tener que mutarlo a una nueva clase.

- Deseas poder anidar agrupaciones recursivamente para que un grupo sea en sí mismo un elemento en otro grupo.

***Estructura***

El patrón de Objeto Tipo tiene dos clases concretas, una que representa objetos y otra que representa sus tipos. Cada objeto tiene un puntero a su tipo correspondiente.

Por ejemplo, el sistema utiliza un TipoObjeto para representar cada tipo en el sistema y un Objeto para representar cada una de las instancias de esos TipoObjetos. Cada Objeto tiene un puntero a su TipoObjeto.

***Participantes***

- ClaseTipo (Película)

- Es la clase de TipoObjeto.

- Tiene una instancia separada para cada tipo de Objeto.

- TipoObjeto (Star Wars, The Terminator, Independence Day)

- Es una instancia de ClaseTipo.

- Representa un tipo de Objeto. Establece todas las propiedades de un Objeto que son iguales para todos los Objetos del mismo tipo.

- Clase (Cinta de Video)

- Es la clase de Objeto.

- Representa instancias de ClaseTipo.

- Objeto (Star Wars de John, Star Wars de Sue)

- Es una instancia de Clase.

- Representa un elemento único que tiene un contexto único. Establece todas las propiedades de ese elemento que pueden diferir entre elementos del mismo tipo.

ClaseTipo y Clase son clases. TipoObjeto y Objeto son instancias de sus respectivas clases. Como con cualquier instancia, un TipoObjeto u Objeto sabe cuál es su clase. Además, un Objeto tiene un puntero a su TipoObjeto para que sepa cuál es su TipoObjeto. El Objeto utiliza su TipoObjeto para definir su comportamiento de tipo. Cuando el Objeto recibe solicitudes que son específicas del tipo pero no específicas de la instancia, delega esas solicitudes a su TipoObjeto. Un TipoObjeto también puede tener punteros a todos sus Objetos.

Por lo tanto, Película es una ClaseTipo y Cinta de Video es una Clase. Las instancias de Película como Star Wars, The Terminator e Independence Day son TipoObjetos. Las instancias de Cinta de Video como Star Wars de John y Star Wars de Sue son Objetos. Dado que un Objeto tiene un puntero a su TipoObjeto, la cinta de video de John y la cinta de video de Sue tienen punteros a su Película correspondiente, que en este caso es Star Wars para ambas cintas de video. Así es como las cintas de video saben que contienen Star Wars y no alguna otra película.

***Colaboraciones***

- Un Objeto recibe dos categorías de solicitudes: las definidas por su instancia y las definidas por su tipo. Maneja las solicitudes de instancia por sí mismo y delega las solicitudes de tipo a su TipoObjeto.

- Algunos clientes pueden querer interactuar directamente con los TipoObjetos. Por ejemplo, en lugar de iterar a través de todas las Cintas de Video que tiene la tienda en stock, un arrendador podría querer buscar todas las Películas que ofrece la tienda.

- Si es necesario, el TipoObjeto puede tener un conjunto de punteros a sus Objetos. De esta manera, el sistema puede recuperar fácilmente un Objeto que se ajuste a la descripción de un TipoObjeto. Esto sería similar al mensaje allInstances que implementan las clases de Smalltalk. Por ejemplo, una vez que un arrendador encuentra una Película atractiva, entonces querría saber qué cintas de video tiene la tienda que se ajustan a la descripción.

***Consecuencias***

Las ventajas del patrón de Objeto Tipo son:

- Creación de clases en tiempo de ejecución. El patrón permite crear nuevas "clases" en tiempo de ejecución. Estas nuevas clases no son realmente clases, son instancias llamadas TipoObjetos que son creadas por la ClaseTipo al igual que cualquier instancia es creada por su clase.

- Evita la explosión de subclases. El sistema ya no necesita numerosas subclases para representar diferentes tipos de Objetos. En lugar de numerosas subclases, el sistema puede usar una sola ClaseTipo y numerosos TipoObjetos.

- Oculta la separación de instancia y tipo. Los clientes de un Objeto no necesitan ser conscientes de la separación entre Objeto y TipoObjeto. El cliente realiza solicitudes al Objeto, y el Objeto a su vez decide qué solicitudes enviar al TipoObjeto. Los clientes que son conscientes de los TipoObjetos pueden colaborar con ellos directamente sin pasar por los Objetos.

- Cambio dinámico de tipo. El patrón permite que el Objeto cambie dinámicamente su TipoObjeto, lo que tiene el efecto de cambiar su clase. Esto es más simple que mutar un objeto a una nueva clase.

- Subclasificación independiente. ClaseTipo y Clase pueden ser subclasificadas de forma independiente.

- Múltiples Objetos Tipo. El patrón permite que un Objeto tenga múltiples TipoObjetos donde cada uno define alguna parte del tipo del Objeto. El Objeto debe entonces decidir qué comportamiento de tipo delegar a qué TipoObjeto.

Las desventajas del patrón de Objeto Tipo son:

- Complejidad de diseño. El patrón divide un objeto lógico en dos clases. Su relación, una cosa y su tipo, es difícil de entender. Esto es confuso tanto para modeladores como para programadores. Es difícil reconocer o explicar la relación entre un TipoObjeto y un Objeto. Esta confusión afecta la simplicidad y mantenibilidad. En resumen: "Utiliza la herencia; es más fácil".

- Complejidad de implementación. El patrón traslada las diferencias de implementación fuera de las subclases y hacia el estado de las instancias de TipoObjeto. Mientras que cada subclase podría implementar un método de manera diferente, ahora la ClaseTipo solo puede implementar el método de una manera y el estado de cada TipoObjeto debe hacer que la instancia se comporte de manera diferente.

- Gestión de referencias. Cada Objeto debe mantener una referencia a su TipoObjeto. Así como un objeto sabe cuál es su clase, un Objeto sabe cuál es su TipoObjeto. Pero mientras que el sistema u lenguaje de programación establece y mantiene automáticamente la relación clase-instancia, la aplicación debe establecer y mantener la relación TipoObjeto-Objeto.

***Implementación***

Hay varios problemas que siempre debes abordar al implementar el patrón de Objeto Tipo:

1. El Objeto referencia al TipoObjeto. Cada Objeto tiene una referencia a su TipoObjeto y delega parte de su responsabilidad al TipoObjeto. El TipoObjeto de un Objeto debe especificarse cuando se crea el Objeto.

2. Comportamiento del Objeto vs. Comportamiento del TipoObjeto. El comportamiento de un Objeto puede ser implementado en su clase o puede ser delegado a su TipoObjeto. El TipoObjeto implementa el comportamiento común al tipo, mientras que el Objeto implementa el comportamiento que difiere para cada instancia de un tipo. Cuando el Objeto delega el comportamiento a su TipoObjeto, puede pasar una referencia a sí mismo para que el TipoObjeto pueda acceder a sus datos o comportamiento. El Objeto puede decidir realizar operaciones adicionales antes y después de enviar la solicitud, de manera similar a cómo un Decorador puede mejorar las solicitudes que envía a su Componente.

1. El TipoObjeto no es herencia múltiple. La Clase, no el TipoObjeto, es la plantilla para el nuevo Objeto. Los mensajes que el Objeto entiende están definidos por su Clase, no por su TipoObjeto. La implementación de la Clase decide qué mensajes enviar al TipoObjeto; el Objeto no hereda los mensajes del TipoObjeto. Siempre que agregues un comportamiento a ClaseTipo, también debes agregar un método de delegación a Clase antes de que el comportamiento esté disponible para los Objetos.

Hay otros problemas que debes considerar al implementar el patrón de Objeto Tipo:

3. Creación de Objetos utilizando un TipoObjeto. A menudo, un nuevo Objeto se crea enviando una solicitud al TipoObjeto correspondiente. Esto es notable porque el TipoObjeto es una instancia y las solicitudes de creación de instancias generalmente se envían a una clase, no a una instancia. Pero el TipoObjeto es como una clase para el Objeto, por lo que a menudo tiene la responsabilidad de crear nuevos Objetos.

4. Múltiples TipoObjetos. Un Objeto puede tener más de un TipoObjeto, pero esto es inusual. En este caso, la Clase tendría que decidir a qué TipoObjeto delegar cada solicitud.

5. Cambio de TipoObjeto. El patrón de Objeto Tipo permite que un objeto cambie dinámicamente su "clase", el TipoObjeto. Es más simple para un objeto cambiar su puntero a un TipoObjeto diferente (una instancia diferente de la misma clase) que mutar a una nueva clase.

6. Por ejemplo, supongamos que se espera que un envío a la tienda de videos contenga tres copias de The Terminator y dos copias de Star Wars, por lo que esos objetos se ingresan en el sistema. Cuando llega el envío, realmente contiene dos copias de The Terminator y tres copias de Star Wars. Entonces, una de las tres nuevas copias de The Terminator en el sistema necesita cambiarse a una copia de Star Wars. Esto se puede hacer fácilmente cambiando el puntero de la cinta de video de The Terminator a Star Wars.

7. Subclasificación de Clase y ClaseTipo. Es posible subclasificar tanto Clase como ClaseTipo. La tienda de videos podría admitir discos de video haciendo otra Clase llamada Videodisco. Una nueva instancia de Videodisco apuntaría a su instancia de Película al igual que lo haría una Cinta de Video. Si la tienda lleva tres cintas y dos discos del mismo película, tres Cintas de Video y dos Videodiscos compartirían la misma Película.

8. La parte difícil del patrón de Objeto de Tipo ocurre después de que se ha utilizado. Existe un impulso casi irresistible de hacer que los TypeObjects sean más componibles y construir herramientas que permitan a los no programadores especificar nuevos TypeObjects. Estas herramientas pueden volverse bastante complejas, y la estructura de los TypeObjects también puede volverse bastante compleja. Evite cualquier complejidad a menos que traiga una gran recompensa.

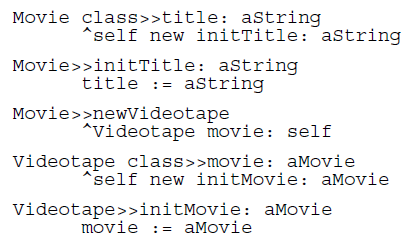
***Código de Ejemplo***

Tienda de Videos

Comenzamos con dos clases, Movie (Película) y Videotape (Cinta de Video).

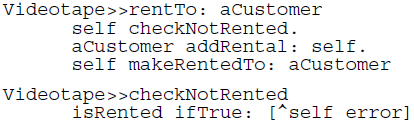
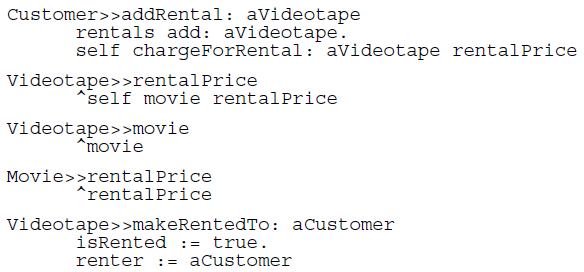
Observa cómo se dividen los atributos entre las dos clases. Si hay varias cintas de video de la misma película, algunas pueden estar alquiladas mientras que otras no lo están. Varias copias ciertamente pueden ser alquiladas a diferentes personas. Por lo tanto, los atributos isRented (está alquilado) y renter (inquilino) se asignan a nivel de Videotape.

Por otro lado, si todas las cintas de video en el grupo contienen la misma película, todas tendrán el mismo nombre, se alquilarán por el mismo precio y tendrán la misma calificación. Por lo tanto, los atributos title, rentalPrice y rating se asignan a nivel de Movie. Esta es la técnica general para separar el TypeObject del Object: Divide los atributos que varían para cada instancia de aquellos que son iguales para un tipo dado.

Se crea una nueva Movie especificando su título. A su vez, una Movie sabe cómo crear una nueva Videotape.

Dado que Movie es el TypeClass de Videotape, Videotape tiene un atributo movie que contiene un puntero a su instancia de Movie correspondiente. Así es como una Videotape sabe cuál es su Movie.

El atributo movie se establece cuando se crea la instancia de Videotape por Videotape class>>movie:.

Una Videotape sabe cómo ser alquilada. Sabe si ya está siendo alquilada. Aunque no conoce su precio directamente, sabe cómo determinar su precio.

Así que elige implementar su comportamiento isRented por sí mismo, pero delega su comportamiento rentalPrice a su Type Object.

Cuando Independence Day se lanza en video doméstico, el sistema crea una Movie para

ello. Reúne la información adecuada sobre la nueva película (título, precio de alquiler, calificación, etc.) a través de una GUI y ejecuta el código necesario. Luego, el sistema crea las nuevas Videotapes usando la nueva Movie.

….

***Usos Conocidos***

Coad: El patrón de Descripción de Ítem de Coad es el patrón de Objeto de Tipo excepto que él solo enfatizó el hecho de que un Tipo contiene valores que todas sus Instancias tienen en común. Utilizó un objeto de "descripción de aeronave" como ejemplo. [Coad92]

Hay: utiliza el patrón de Objeto de Tipo en muchos de sus patrones de modelado de datos, y lo discute como un principio de modelado, pero no lo llama un patrón separado. Lo utiliza para definir tipos para actividades, productos, activos (un supertipo de producto), incidentes, cuentas, pruebas, documentos y secciones de una Ficha de Datos de Seguridad de Materiales. [Hay96]

Fowler: habla sobre los mundos separados de Tipo de Objeto y Objeto, y los llama el "nivel de conocimiento" y el "nivel operativo". Utiliza el Objeto de Tipo para definir tipos para unidades organizativas, relaciones de responsabilidad, partes involucradas en relaciones, contratos, los términos para contratos y medidas, así como muchas de las cosas que Hay discutió. [Fowler97]

Odell: El patrón de Tipo de Poder de Odell es el patrón de Objeto de Tipo. Lo ilustra con el ejemplo de especies de árboles y árbol. Una especie de árbol describe un tipo de árbol como olmo americano, arce de azúcar, albaricoquero o saguaro. Un árbol representa un árbol particular en mi patio delantero o el de tu patio trasero. Cada árbol tiene una especie de árbol correspondiente que describe qué tipo de árbol es. [MO95]

Tipos de Ejemplos y MuestrasP: El patrón de Objeto de Tipo se ha utilizado en el campo médico para modelar muestras médicas. Una muestra tiene cuatro propiedades independientes:

- el sistema del que se toma (por ejemplo, John Doe)

- el subsistema (por ejemplo, sangre, orina, esputo)

- el procedimiento de recolección (aspiración, drenaje, raspado)

- el aditivo de preservación (heparina, EDTA)

Esto se modela fácilmente como un objeto de Muestra con cuatro atributos: sistema, subsistema, procedimiento de recolección y aditivo. Aunque el sistema (la persona que proporcionó la muestra) es diferente para casi todas las muestras, el trío (subsistema, procedimiento de recolección y aditivo) es compartido por muchas muestras. Por ejemplo, los técnicos médicos se refieren a una muestra "de sangre", lo que significa una muestra de sangre/aspiración/EDTA. Por lo tanto, los atributos del trío se pueden agrupar en un solo objeto de Tipo de Muestra.

Un Tipo de Muestra es responsable de crear nuevos objetos de Muestra. Hay alrededor de 5,000 combinaciones de tríos posibles, pero la mayoría de ellas no tienen sentido, por lo que el sistema solo proporciona los Tipo de Muestra más comunes. Si se necesita otro Tipo de Muestra, los usuarios pueden crear uno nuevo especificando su subsistema, procedimiento de recolección y aditivo. Mientras el sistema rastrea decenas de miles de Muestras, solo necesita rastrear alrededor de cien Tipo de Muestra. Entonces, los Tipo de Muestra son Objetos de Tipo y las Muestras son sus Objetos. [DeKezel96]

Señales y Excepciones: El patrón de Objeto de Tipo es más común en marcos de dominio que en marcos de proveedores, pero un ejemplo de proveedor es el marco de Señales/Excepciones en VisualWorks Smalltalk. Cuando el código de Smalltalk encuentra un error, puede generar una Excepción. La Excepción registra el contexto de dónde ocurrió el error con fines de depuración. Sin embargo, la Excepción en sí misma no sabe qué es lo que sucedió.

Algo salió mal, justo donde. Delega la información sobre qué sucedió a una Señal. Cada Señal describe un posible tipo de problema, como interrupción de usuario, mensaje no entendido y desbordamiento de subíndice. Por lo tanto, dos errores de mensaje no entendido crean dos instancias de Excepción separadas que apuntan a la misma instancia de Señal. Señal es la Clase de Tipo y Excepción es la Clase. [VW95]

***Reflexión***

El patrón de Objeto de Tipo está presente en la mayoría de los sistemas reflectivos, donde un objeto de tipo a menudo se llama metaobjeto. La separación de clase/instancia en Smalltalk es un ejemplo del patrón de Objeto de Tipo. Los programadores pueden manipular clases directamente, agregando métodos, cambiando la jerarquía de clases y creando nuevas clases. Por mucho, el uso más común de una clase es hacer instancias, pero los otros usos son parte de la cultura y se discuten a menudo, aunque no se usen a menudo. [KRB91]

La reflexión tiene una merecida reputación de ser difícil de entender. El patrón de Objeto de Tipo muestra que no tiene que ser difícil y puede ser una entrada fácil al mundo más complejo de la programación reflexiva.

***Patrones Relacionados***

El patrón de Objeto de Tipo es similar a los patrones de Estrategia y Estado [GHJV95, página 315 y página 305]. Los tres patrones dividen un objeto en piezas y el "objeto real" delega en el nuevo objeto, ya sea el Objeto de Tipo, la Estrategia o el Estado. La Estrategia y el Estado suelen ser puramente de comportamiento, mientras que un Objeto de Tipo a menudo contiene mucho estado compartido. Los Estados cambian con frecuencia, mientras que los Objetos de Tipo rara vez cambian. Una Estrategia suele tener una responsabilidad principal, mientras que un Objeto de Tipo suele tener muchas responsabilidades. Por lo tanto, los patrones no son exactamente iguales, aunque sus diagramas de objeto son similares.

Cualquier sistema con un Objeto de Tipo está en camino de tener una Arquitectura Reflexiva [BMRSS96]. A menudo, un Objeto de Tipo contiene Estrategias para sus instancias. Esta es una buena manera de definir el comportamiento en un tipo.

Una implementación de Objeto de Tipo puede volverse lo suficientemente compleja como para tener jerarquías de Clase y Clase de Tipo. Estas jerarquías se parecen mucho a las jerarquías de Abstracción e Implementador en el patrón de Puente [GHJV95, página 151], donde Clase es la abstracción y Clase de Tipo es la implementación. Sin embargo, los clientes pueden colaborar directamente con los Objetos de Tipo, una interacción que generalmente no ocurre con los Implementadores Concretos.

Un Objeto puede parecer un Decorador [GHJV95, página 175] para su Objeto de Tipo. Un Objeto y su Objeto de Tipo tienen interfaces similares y el Objeto elige qué mensajes enviar a su Objeto de Tipo y cuáles mejorar. Sin embargo, un Decorador no se comporta como una instancia de su Componente.

Los Objetos de Tipo pueden parecer Peso Ligero [GHJV95, página 195] para sus Objetos. Dos Objetos que usan el mismo Objeto de Tipo podrían pensar que cada uno tiene su propia copia, pero en cambio están compartiendo la misma. Por lo tanto, es importante que ninguno de los Objetos cambie el estado intrínseco del Objeto de Tipo.

Otra forma de hacer que un objeto actúe como el tipo de otro es con el patrón de Prototipo [GHJV95, página 117], cuando cada objeto realiza un seguimiento de su prototipo y delega las solicitudes a él que no sabe cómo manejar.