ET++ [WGM88] usa pesos ligeros para permitir la independencia de la interfaz de usuario.*[5] El estándar de interfaz de usuario afecta a la disposición de los elementos de la interfaz de usuario (por ejemplo, barras de desplazamiento, botones y menús –lo que se conoce con el nombre colectivo de "útiles"–)** y sus adornos (sombras, bordes, etc.). Un útil delega todo su comportamiento de posicionamiento y dibujado en otro objeto Layout. Cambiar el objeto Layout cambia el aspecto de la interfaz, incluso en tiempo de ejecución.

Para cada clase de útil hay su correspondiente clase Layout (por ejemplo, ScrollbarLayout, MenubarLayout, etc.). Un problema evidente con este enfoque es que usar diferentes objetos Layout duplica el número de objetos de interfaz de usuario: para cada objeto de interfaz de usuario hay un objeto Layout adicional. Para evitar esta penalización, los objetos Layout se implementan como pesos ligeros. Dichos objetos son buenos pesos ligeros porque su principal misión es definir comportamiento, y es fácil pasarles el poco estado extrínseco que necesitan para colocar o dibujar un objeto.

Los objetos Layout son creados y gestionados por objetos Look. La clase Look es una Fabrica Abstracta (79) que recupera un determinado objeto Layout con operaciones como GetButtonLayout, GetMenuBarLayout y otras. Para cada estándar de interfaz de usuario existe la correspondiente subclase de Look (como MotifLook u OpenLook) que proporciona los objetos Layout apropiados.

Por cierto, los objetos Layout son, en esencia, estrategias, (*véase* el patrón Strategy (289)). Son un ejemplo de objeto estrategia implementado como un peso ligero.

PATRONES RELACIONADOS

El patrón Flyweight suele combinarse con el patrón Composite (151) para implementar una estructura lógica jerárquica en términos de un grafo dirigido acíclico con nodos hojas compartidos.

Suele ser mejor implementar los objetos State (279) y Strategy (289) como pesos ligeros.

* Hemos traducido *look and feel* como "interfaz de usuario". *(N. del T.)*

[5] *Véase* el patrón Abstract Factory (79) para otra aproximación a la independencia de la interfaz de usuario.

** *Widgets*, en el original en inglés. *(N. del T.)*

PROXY (APODERADO)

Estructural de Objetos

PROPÓSITO

Proporciona un representante o sustituto de otro objeto para controlar el acceso a éste.

TAMBIÉN CONOCIDO COMO

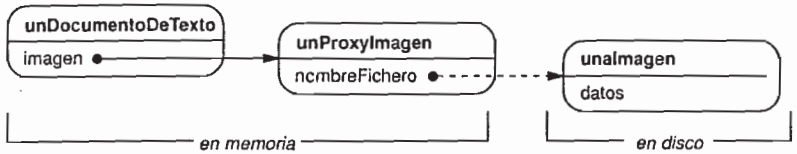
Surrogate (Sustituto)

MOTIVACIÓN

Una razón para controlar el acceso a un objeto es retrasar todo el coste de su creación e inicialización hasta que sea realmente necesario usarlo. Pensemos en un editor de documentos que puede insertar objetos gráficos en un documento. Algunos de estos objetos gráficos, como las grandes imágenes *raster*, pueden ser costosos de crear. Sin embargo, abrir un documento debería ser una operación que se efectuase rápidamente, por lo que se debería evitar crear todos los objetos costosos a la vez en cuanto se abre el documento. Por otro lado, tampoco es necesario, ya que no todos esos objetos serán visibles en el documento al mismo tiempo.

Estas restricciones sugieren que cada objeto concreto se cree *a petición*, lo que en este caso tendrá lugar cuando la imagen se hace visible. Pero, ¿qué ponemos en el documento en lugar de la imagen? ¿Y cómo puede ocultarse el hecho de que la imagen se crea a petición sin que se complique la implementación del editor? Esta optimización no debería influir, por ejemplo, en el código de visualización y formateado.

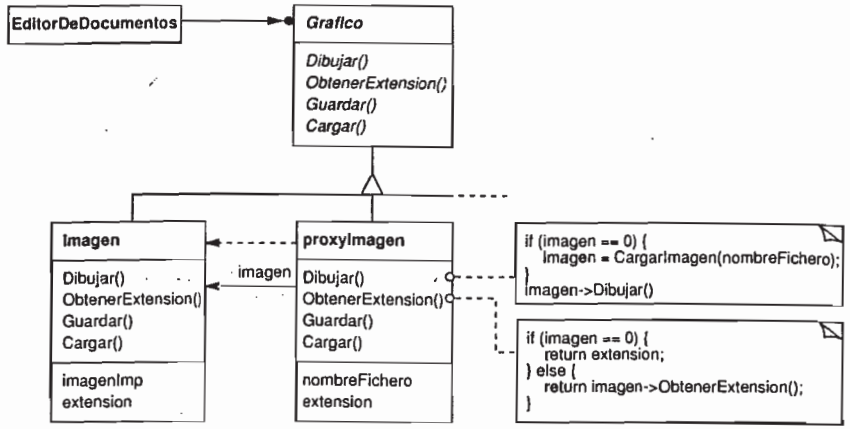
La solución es utilizar otro objeto, un **proxy** de la imagen, que actúe como un sustituto de la imagen real. El proxy se comporta igual que la imagen y se encarga de crearla cuando sea necesario.



El proxy de la imagen crea la imagen real sólo cuando el editor de documentos le pide que se dibuje, invocando a su operación Dibujar. El proxy redirige las siguientes peticiones directamente a la imagen. Debe guardar, por tanto, una referencia a la imagen después de crear ésta.

Supongamos que las imágenes se guardan en ficheros aparte. En este caso podemos usar el nombre del fichero como la referencia al objeto real. El proxy también almacena su **extensión**, esto es, su ancho y su alto. La extensión permite que el proxy pueda responder a las preguntas sobre su tamaño que le haga el formateador sin crear realmente la imagen.

El siguiente diagrama de clases ilustra este ejemplo más en detalle.



El editor de documentos accede a las imágenes insertadas a través de la interfaz definida por la clase abstracta Grafico. ProxyImagen es una clase para las imágenes que se crean a petición. ProxyImagen tiene como referencia a la imagen en el disco el nombre del fichero, el cual recibe como parámetro el constructor de la clase.

ProxyImagen también guarda la caja que circunscribe la imagen y una referencia a la instancia de la Imagen real, la cual no será válida hasta que el proxy cree dicha imagen real. La operación Dibujar se asegura de que la imagen ha sido creada antes de enviarle la petición. ObtenerExtension redirige la petición a la imagen sólo si ésta ha sido creada; en caso contrario, es ProxyImagen quien devuelve su extensión.

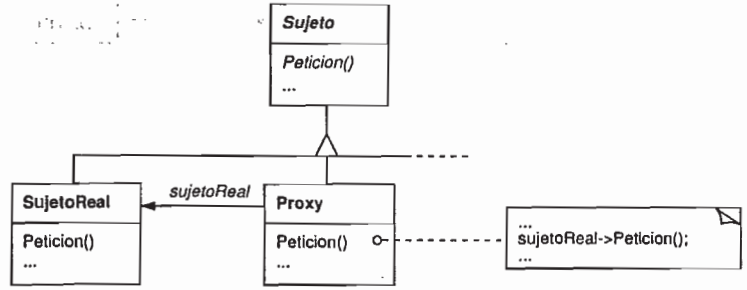
APLICABILIDAD

Este patrón es aplicable cada vez que hay necesidad de una referencia a un objeto más versátil o sofisticada que un simple puntero. Éstas son varias situaciones comunes en las que es aplicable el patrón Proxy:

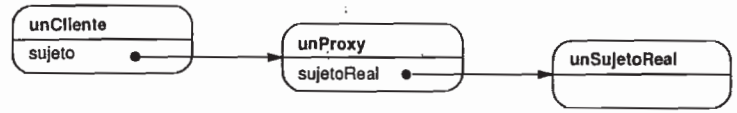
1. Un **proxy remoto** proporciona un representante local de un objeto situado en otro espacio de direcciones. NEXTSTEP [Add94] usa la clase NXProxy con este propósito. Coplien [Cop92] llama "Embajador" a este tipo de proxy.
2. Un **proxy virtual** crea objetos costosos por encargo. El ProxyImagen descrito en la sección de Motivación es un ejemplo de este tipo de proxy.
3. Un **proxy de protección** controla el acceso al objeto original. Los proxies de protección son útiles cuando los objetos debieran tener diferentes permisos de acceso. Por ejemplo, los KernelProxy del sistema operativo Choices [CIRM93] proporcionan un acceso protegido a los objetos del sistema operativo.
4. Una **referencia inteligente** es un sustituto de un simple puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto. Algunos ejemplos de usos típicos son:

   - contar el número de referencias al objeto real, de manera que éste pueda liberarse automáticamente cuando no haya ninguna referencia apuntándole (también se conocen con el nombre de **punteros inteligentes** [Ede92]).
   - cargar un objeto persistente en la memoria cuando es referenciado por primera vez.
   - comprobar que se bloquea el objeto real antes de acceder a él para garantizar que no pueda ser modificado por ningún otro objeto.

ESTRUCTURA



Éste es un posible diagrama de objetos de una estructura de proxies en tiempo de ejecución:



PARTICIPANTES

- **Proxy** (ProxyImagen)
  - mantiene una referencia que permite al proxy acceder al objeto real. El proxy puede referirse a un Sujeto en caso de que las interfaces de SujetoReal y Sujeto sean la misma.
  - proporciona una interfaz idéntica a la de Sujeto, de manera que un proxy pueda ser sustituido por el sujeto real.
  - controla el acceso al sujeto real, y puede ser responsable de su creación y borrado.
  - otras responsabilidades dependen del tipo de proxy:
    - los *proxies remotos* son responsables de codificar una petición y sus argumentos para enviar la petición codificada al sujeto real que se encuentra en un espacio de direcciones diferente.
    - los *proxies virtuales* pueden guardar información adicional sobre el sujeto real, por lo que pueden retardar el acceso al mismo. Por ejemplo, el ProxyImagen de la sección de Motivación guarda la extensión de la imagen real.
    - los *proxies de protección* comprueban que el llamador tenga los permisos de acceso necesarios para realizar una petición.

- **Sujeto** (Grafico)
  - define la interfaz común para el SujetoReal y el Proxy, de modo que pueda usarse un Proxy en cualquier sitio en el que se espere un SujetoReal.
- **SujetoReal** (Imagen)
  - define el objeto real representado.

COLABORACIONES

El Proxy redirige peticiones al SujetoReal cuando sea necesario, dependiendo del tipo de proxy.

CONSECUENCIAS

El patrón Proxy introduce un nivel de indirección al acceder a un objeto. Esta indirección adicional tiene muchos posibles usos, dependiendo del tipo de proxy:

1. Un proxy remoto puede ocultar el hecho de que un objeto reside en un espacio de direcciones diferente.
2. Un proxy virtual puede llevar a cabo optimizaciones tales como crear un objeto por encargo.
3. Tanto los proxies de protección, como las referencias inteligentes, permiten realizar tareas de mantenimiento adicionales cuando se accede a un objeto.

Hay otra optimización que el patrón Proxy puede ocultar al cliente. Se denomina **copia-de-escritura**, y está relacionada con la creación por encargo. Copiar un objeto grande y complejo puede ser una operación costosa. Si dicha copia no se modifica nunca, no hay necesidad de incurrir en ese gasto. Al utilizar un proxy para posponer el proceso de copia nos estamos asegurando de que sólo pagamos el precio de copiar el objeto en caso de que éste sea modificado.

Para realizar una copia-de-escritura el sujeto debe tener un contador de referencias. Copiar el proxy no será más que incrementar dicho contador. Sólo cuando el cliente solicita una operación que modifica el sujeto es cuando el proxy realmente lo copia. En ese caso el proxy también tiene que disminuir el contador de referencias del sujeto. Cuando éste llega a cero se borra el sujeto.

La copia-de-escritura puede reducir significativamente el coste de copiar sujetos pesados.

IMPLEMENTACIÓN

El patrón Proxy puede explotar las siguientes características de los lenguajes:

1. *Sobrecargar el operador de acceso a miembros en C++.* C++ admite la sobrecarga de operator->, el operador de acceso a miembros. Sobrecargar este operador permite realizar tareas adicionales cada vez que se desreferencia un objeto, lo que puede resultar útil para implementar algunos tipos de proxies; el proxy se comporta igual que un puntero.
   El siguiente ejemplo ilustra cómo usar esta técnica para implementar un proxy virtual llamado `PunteroImagen`.

```
class Imagen;
extern Imagen* CargarUnFicheroDeImagen(const char*);
    // función externa

class PunteroImagen {
public:
    PunteroImagen(const char* FicheroDeImagen);
    virtual -PunteroImagen();

    virtual Imagen* operator->();
    virtual Imagen& operator*();
private:
    Imagen* CargarImagen();
private:
    Imagen* _imagen;
    const char* _ficheroDeImagen;
};

PunteroImagen::PunteroImagen (const char* elFicheroDeImagen) {
    _ficheroDeImagen = elFicheroDeImagen;
    _imagen = 0;
}

Imagen* PunteroImagen::CargarImagen () {
    if (_imagen == 0) {
        _imagen = CargarUnFicheroDeImagen(_ficheroDeImagen);
    }
    return _imagen;
}
```

Los operadores sobrecargados -> y * usan `CargarImagen` para devolver `_imagen` a sus llamadores (cargando la imagen si es necesario).

```
Imagen* PunteroImagen::operator-> () {
    return CargarImagen();
}

Imagen& PunteroImagen::operator* () {
    return *CargarImagen();
}
```

Este enfoque permite llamar a las operaciones de `Imagen` a través de objetos `PunteroImagen` sin el problema de hacer a esas operaciones parte de la interfaz de `PunteroImagen`:

```
PunteroImagen imagen = PunteroImagen("nombreDeUnFicheroDeImagen");
imagen->Dibujar(Punto(50, 100));
    // (Imagen.operator->())->Dibujar(Punto(50, 100))
```

Nótese cómo el proxy de `imagen` funciona como un puntero, pero no está declarado como un puntero a `Imagen`. Eso significa que no se puede usar exactamente igual que un puntero real a `Imagen`. Por tanto, con este enfoque los clientes deben tratar de forma diferente a los objetos `Imagen` y `PunteroImagen`.

Sobrecargar el operador de acceso a miembros no es una buena solución para todos los tipos de proxies. Algunos necesitan saber exactamente *qué* operación es llamada, y en esos casos no sirve sobrecargar el operador de acceso a miembros.

Piénsese en el ejemplo del proxy virtual de la sección de Motivación. La imagen debería cargarse en un determinado instante –en concreto, cuando se llama a la operación Dibujar–, y no cada vez que se hace referencia a la imagen. Sobrecargar el operador de acceso no permite realizar esta distinción. En ese caso debemos implementar manualmente cada operación del proxy que redirige la petición al sujeto.

Estas operaciones suelen ser muy parecidas unas a otras, como se demuestra en el Código de Ejemplo. Normalmente todas las operaciones verifican que la petición es legal, que existe el objeto original, etc., antes de redirigir la petición al sujeto. Escribir este código una y otra vez es una labor tediosa. Por ese motivo, es frecuente usar un preprocesador para generarlo automáticamente.

2. *Usar* `doesNotUnderstand` *en Smalltalk.* Smalltalk proporciona un enganche que se puede usar para permitir el reenvío automático de peticiones. Smalltalk llama a `doesNotUnderstand: unMensaje` cuando un cliente envía un mensaje a un receptor que no tiene el método correspondiente. La clase Proxy puede redefinir `doesNotUnderstand` para que el mensaje sea reenviado a su sujeto.

   Para garantizar que una petición se redirige al sujeto y no es absorbida en silencio por su proxy, puede definirse una clase Proxy que no entienda *ningún* mensaje. Smalltalk permite hacer esto definiendo Proxy como una clase sin superclase.[6]

   El principal inconveniente de `doesNotUnderstand:` es que la mayoría de sistemas Smalltalk tienen unos pocos mensajes especiales que son manejados directamente por la máquina virtual, y para éstos no se lleva a cabo la habitual búsqueda de métodos. El único que suele estar implementado en Object (y que puede por tanto afectar a los proxies) es la operación de identidad, `==`.

   Si se decide usar `doesNotUnderstand:` para implementar Proxy será necesario hacer un diseño que tenga en cuenta este problema. No se debe suponer que la identidad entre proxies significa que sus sujetos reales son también idénticos. Un inconveniente añadido es que `doesNotUnderstand:` fue desarrollada para el tratamiento de errores, no para crear proxies, por lo que no suele ser demasiado rápida.

3. *Los proxies no siempre tienen que conocer el tipo del sujeto real.* Si una clase Proxy puede tratar con su sujeto sólo a través de una interfaz abstracta, entonces no hay necesidad de hacer una clase Proxy para cada clase de SujetoReal; el proxy puede tratar de manera uniforme a todas las clases SujetoReal. Pero si los objetos Proxy van a crear instancias de SujetoReal (como en el caso de los proxies virtuales), entonces tienen que conocer la clase concreta.

Otra cuestión de implementación tiene que ver con cómo referirse al sujeto antes de que se cree una instancia de éste. Algunos proxies tienen que referirse a su sujeto independientemente de que esté en disco o en memoria. Eso significa que deben usar alguna forma de identificadores de objetos independientes del espacio de direcciones. En la sección de Motivación se usó un nombre de fichero para tal fin.

CÓDIGO DE EJEMPLO

El siguiente código implementa dos tipos de proxies: el proxy virtual descrito en la sección de Motivación y un proxy implementado con `doesNotUnderstand:`.[7]

[6] La implementación de objetos distribuidos en NEXSTEP [Add94] (en concreto, la clase NXProxy) usa esta técnica. La implementación redefine `forward`, el enganche equivalente en NEXTSTEP.

[7] El patrón Iterator (237) describe otro tipo de proxy en la página 245.

1. *Un proxy virtual.* La clase `Grafico` define la interfaz de los objetos gráficos:

```
class Grafico {
public:
    virtual ~Grafico();

    virtual void Dibujar(const Punto& en) = 0;
    virtual void ManejarRaton(Evento& evento) = 0;

    virtual const Punto& ObtenerExtension() = 0;

    virtual void Cargar(istream& desde) = 0;
    virtual void Guardar(ostream& en) = 0;
protected:
    Grafico();
};
```

La clase `Imagen` implementa la interfaz `Grafico` para mostrar ficheros de Imagen. `Imagen` redefine `ManejarRaton` para que los usuarios puedan cambiar interactivamente el tamaño de la Imagen.

```
class Imagen : public Grafico {
public:
    Imagen(const char* fichero);  // carga una Imagen desde un fichero
    virtual ~Imagen();

    virtual void Dibujar(const Punto& en);
    virtual void ManejarRaton(Evento& evento);

    virtual const Punto& ObtenerExtension();

    virtual void Cargar(istream& desde);
    virtual void Guardar(ostream& en);
private:
    // ...
};
```

`ProxyImagen` tiene la misma interfaz que `Imagen`:

```
class ProxyImagen : public Grafico {
public:
    ProxyImagen(const char* FicheroDeImagen);
    virtual ~ProxyImagen();

    virtual void Dibujar(const Punto& en);
    virtual void ManejarRaton(Evento& evento);

    virtual const Punto& ObtenerExtension();

    virtual void Cargar(istream& desde);
    virtual void Guardar(ostream& en);
protected:
    Imagen* ObtenerImagen();
private:
```

```
    Imagen* _imagen;
    Punto _extension;
    char* _nombreDeFichero;
};
```

El constructor guarda una copia local del nombre de fichero que contiene la Imagen, e inicializa _extension e _imagen:

```
ProxyImagen::ProxyImagen (const char* nombreDeFichero)  {
    _nombreDeFichero = strdup(nombreDeFichero);
    _extension = Punto::Cero;  // todavía no se conoce la extensión
    _imagen = 0;
}

Imagen* ProxyImagen::GetImagen() {
    if (_imagen == 0) {
        _imagen = new Imagen(_nombreDeFichero);
    }
    return _imagen;
}
```

La implementación de ObtenerExtension devuelve, si es posible, la extensión guardada por el proxy; en otro caso se carga la Imagen desde el fichero. Dibujar carga la Imagen, y ManejarRaton reenvía el evento a la Imagen real.

```
const Punto& ProxyImagen::ObtenerExtension () {
    if (_extension == Punto::Cero) {
        _extension = ObtenerImagen()->ObtenerExtension();
    }
    return _extension;
}

void ProxyImagen::Dibujar (const Punto& en) {
    ObtenerImagen()->Dibujar(en);
}

void ProxyImagen::ManejarRaton (Evento& evento) {
    ObtenerImagen()->ManejarRaton(evento);
}
```

La operación Guardar graba en un flujo de salida la extensión y el nombre del fichero de Imagen almacenados en el proxy. Cargar recupera esta información e inicializa los miembros correspondientes.

```
void ProxyImagen::Guardar (ostream& en) {
    en << _extension << _nombreDeFichero;
}

void ProxyImagen::Cargar (istream& desde) {
    desde >> _extension >> _nombreDeFichero;
}
```


Por último, supongamos que tenemos una clase DocumentoDeTexto que puede contener objetos Grafico:

```
class DocumentoDeTexto {
public:
    DocumentoDeTexto();

    void Insertar(Grafico*);
    // ...
};
```

Podemos insertar un ProxyImagen en un documento de texto como se muestra a continuación:

```
DocumentoDeTexto* texto = new DocumentoDeTexto;
// ...
texto->Insertar(new ProxyImagen("nombreDeUnFicheroDeImagen"));
```

2. *Proxies que usan* doesNotUnderstand. Se pueden crear proxies genéricos en Smalltalk definiendo clases cuya superclase sea nil[8] y definiendo el método doesNotUnderstand: para manejar los mensajes.
El siguiente método presupone que el proxy tiene un método sujetoReal que devuelve su sujeto real. En el caso de ProxyImagen, este método comprobaría si se había creado la Imagen, la crearía si fuese necesario y finalmente la devolvería. Hace uso de perform:withArguments: para responder al mensaje que ha sido atrapado en el sujeto real.

```
doesNotUnderstand: unMensaje
    ^ self sujetoReal
        perform: unMensaje selector
        withArguments: unMensaje arguments
```

El argumento de doesNotUnderstand: es una instancia de Message que representa el mensaje que no entiende el proxy. Por tanto, el proxy responde a todos los mensajes asegurándose de que existe el sujeto real antes de reenviarle el mensaje.
Una de las ventajas de doesNotUnderstand: es que puede realizar un procesamiento arbitrario. Por ejemplo, podríamos obtener un proxy de protección especificando un conjunto mensajesLegales con los mensajes que deben ser aceptados y dándole al proxy el método siguiente:

```
doesNotUnderstand: unMensaje
    ^ (mensajesLegales includes: unMensaje selector)
        ifTrue: [self sujetoReal
            perform: unMensaje selector
            withArguments: unMensaje arguments]
        ifFalse: [self error: 'Operador ilegal']
```

Este método comprueba que un mensaje sea legal antes de redirigirlo al sujeto real. En caso de que no lo sea, enviará error: al proxy, lo que daría como resultado un bucle infinito de errores a menos que el proxy defina error:. Por tanto, debería copiarse la definición de error: de la clase Object junto con cualquier método que use.

[8] Casi todas las clases tienen, en última instancia, a Object como superclase. Por tanto, esto es lo mismo que decir "definir una clase que no tenga a Object como su superclase".

USOS CONOCIDOS

El proxy virtual de la sección de Motivación está tomado de las clases de ET++ para construcción de bloques de texto.

NEXTSTEP [Add94] usa proxies (instancias de la clase NXProxy) como proxies locales de objetos que pueden ser distribuidos. Un servidor crea proxies de objetos remotos cuando los solicitan los clientes. Al recibir un mensaje, el proxy lo codifica junto con sus argumentos y envía el mensaje codificado al sujeto remoto. De forma similar, el sujeto codifica los resultados a devolver y los envía de vuelta al objeto NXProxy.

McCullough [McC87] examina el uso de proxies en Smalltalk para acceder a objetos remotos. Pascoe [Pas86] describe cómo proporcionar efectos laterales y control de acceso en las llamadas a métodos mediante "Encapsuladores".

PATRONES RELACIONADOS

Adapter (131): un adaptador proporciona una interfaz diferente para el objeto que adapta. Por el contrario, un proxy tiene la misma interfaz que su sujeto. No obstante, un proxy utilizado para protección de acceso podría rechazar una operación que el sujeto sí realiza, de modo que su interfaz puede ser realmente un subconjunto de la del sujeto.

Decorator (161): si bien los decoradores pueden tener una implementación parecida a los proxies, tienen un propósito diferente. Un decorador añade una o más responsabilidades a un objeto, mientras que un proxy controla el acceso a un objeto.

Los proxies difieren en el grado de similitud entre su implementación y la de un decorador. Un proxy de protección podría implementarse exactamente como un decorador. Por otro lado, un proxy remoto no contendrá una referencia directa a su sujeto real sino sólo una referencia indirecta, como "un ID de máquina y la dirección local en dicha máquina". Un proxy virtual empezará teniendo una referencia indirecta como un nombre de fichero, pero podrá al final obtener y utilizar una referencia directa.

DISCUSIÓN SOBRE LOS PATRONES ESTRUCTURALES

Tal vez haya notado similitudes entre los patrones estructurales, especialmente en sus participantes y colaboradores. Esto es debido a que los patrones estructurales se basan en un mismo pequeño conjunto de mecanismos del lenguaje para estructurar el código y los objetos: herencia simple y herencia múltiple para los patrones basados en clases, y composición de objetos para los patrones de objetos. Pero estas similitudes ocultan los diferentes propósitos de estos patrones. En esta sección se comparan y contrastan grupos de patrones estructurales para ofrecerle una visión de los méritos de cada uno de ellos.

ADAPTER FRENTE A BRIDGE

Los patrones Adapter (131) y Bridge (141) tienen algunos atributos comunes. Ambos promueven la flexibilidad al proporcionar un nivel de indirección a otro objeto. Ambos implican reenviar peticiones a este objeto desde una interfaz distinta de la suya propia.

La diferencia fundamental entre estos patrones radica en su propósito. El patrón Adapter se centra en resolver incompatibilidades entre dos interfaces existentes. No presta atención a cómo se implementan dichas interfaces, ni tiene en cuenta cómo podrían evolucionar de forma independiente. Es un modo de lograr que dos clases diseñadas independientemente trabajen juntas sin tener que volver a implementar una u otra. Por otro lado, el patrón Bridge une una implementación con sus implementaciones (que pueden ser numerosas). Proporciona una interfaz estable a los clientes permitiendo, no obstante, que cambien las clases que la implementan. También permite incorporar nuevas implementaciones a medida que evoluciona el sistema.

Como resultado de estas diferencias, los patrones Adapter y Bridge suelen usarse en diferentes puntos del ciclo de vida del software. Un adaptador suele hacerse necesario cuando se descubre que deberían trabajar juntas dos clases incompatibles, generalmente para evitar duplicar código, y este acoplamiento no había sido previsto. Por el contrario, el usuario de un puente sabe de antemano que una abstracción debe tener varias implementaciones, y que una y otras pueden variar independientemente. El patrón Adapter hace que las cosas funcionen *después* de que han sido diseñadas; el Bridge lo hace *antes*. Eso no significa que el Adapter sea en modo alguno inferior al Bridge; simplemente, cada patrón resuelve un problema distinto.

Podemos ver una fachada (*véase* el patrón Facade (171)) como un adaptador para un conjunto de objetos. Pero esa interpretación obvia el hecho de que una fachada define una *nueva* interfaz, mientras que un adaptador reutiliza una interfaz existente. Recuérdese que un adaptador hace que trabajen juntas dos interfaces *existentes* en vez de tener que definir una completamente nueva.

COMPOSITE FRENTE A DECORATOR Y A PROXY

Los patrones Composite (151) y Decorator (161) tienen diagramas de estructura parecidos, lo que refleja el hecho de que ambos se basan en la composición recursiva para organizar un número indeterminado de objetos. Esta característica en común puede tentar a pensar en un objeto decorador como un compuesto degenerado, pero eso no representa la esencia del patrón Decorator. El parecido termina en la composición recursiva, al tratarse de nuevo de propósitos diferentes.

El patrón Decorator está diseñado para permitir añadir responsabilidades a objetos sin crear subclases. Esto evita la explosión de subclases a la que puede dar lugar al intentar cubrir cada combinación de responsabilidades estáticamente. El patrón Composite tiene un propósito diferente. Consiste en estructurar subclases para que se puedan tratar de manera uniforme muchos objetos relacionados,

y que múltiples objetos puedan ser tratados como uno solo. Es decir, no se centra en la decoración sino en la representación.

Estos propósitos son distintos pero complementarios. Por tanto, los patrones Composite y Decorador suelen usarse conjuntamente. Ambos llevan a la clase de diseño en la que se pueden construir aplicaciones simplemente poniendo juntos objetos sin definir ninguna clase nueva. Habrá una clase abstracta con algunas subclases que son compuestos, otras que son decoradores y otras que implementan los principales bloques de construcción del sistema. En este caso, tanto compuestos como decoradores tendrán una interfaz común. Desde el punto de vista del patrón Decorator, un compuesto es un ComponenteConcreto. Desde el punto de vista del patrón Composite, un decorador es una Hoja. Por supuesto, no tienen por qué ser usados juntos, y, como hemos visto, sus propósitos son bastante distintos.

Otro patrón con una estructura similar al Decorator es el Proxy (191). Ambos patrones describen cómo proporcionar un nivel de indirección a un objeto, y las implementaciones de los objetos proxy y decorador mantienen una referencia a otro objeto, al cual reenvían peticiones. Una vez más, no obstante, están pensados para propósitos diferentes.

Al igual que el Decorator, el patrón Proxy compone un objeto y proporciona una interfaz idéntica a los clientes. A diferencia del Decorator, el patrón Proxy no tiene que ver con asignar o quitar propiedades dinámicamente, y tampoco está diseñado para la composición recursiva. Su propósito es proporcionar un sustituto para un sujeto cuando no es conveniente o deseable acceder directamente a él, debido, por ejemplo, a residir en una máquina remota, tener acceso restringido o ser persistente.

En el patrón Proxy, el sujeto define la principal funcionalidad, y el proxy proporciona (o rechaza) el acceso al mismo. En el Decorator, el componente proporciona sólo parte de funcionalidad, y uno o más decoradores hacen el resto. El patrón Decorator se encarga de aquellas situaciones en las que no se puede determinar toda la funcionalidad de un objeto en tiempo de compilación, o al menos no resulta conveniente hacerlo. Ése no es el caso del Proxy, ya que éste se centra en una relación —entre el proxy y su sujeto— y dicha relación puede expresarse estáticamente.

Estas diferencias son significativas, ya que representan soluciones a problemas concretos y recurrentes en el diseño orientado a objetos. Pero eso no significa que estos patrones no puedan combinarse. Podríamos pensar en un proxy-decorador que añadiese funcionalidad a un proxy, o en un decorador-proxy que adornase un objeto remoto. Si bien tales híbridos *pueden* ser útiles (no tenemos ejemplos reales a mano), pueden dividirse en patrones que *realmente* son útiles.

CAPÍTULO 5

Patrones de Comportamiento

CONTENIDO DEL CAPÍTULO

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor
Discusión sobre los patrones de comportamiento