ma de una imagen depende de las características de un dispositivo de visualización, concretamente de su capacidad de color y de su resolución. Sin la ayuda de AppKit los desarrolladores tendrían que determinar qué implementación usar bajo varias circunstancias en cada aplicación.

Para aliviar a los desarrolladores de esta responsabilidad, AppKit proporciona un puente para NXImage/NXImageRep. NXImage define la interfaz para manipular imágenes. La implementación de las imágenes se define en una jerarquía de clases separada NXImageRep que tiene subclases como NXEPSImageRep, NXCachedImageRep y NXBitMapImageRep. NXImage mantiene una referencia a uno o más objetos NXImageRep. Si hay más de una implementación de una imagen, NXImage selecciona la más apropiada para el dispositivo de visualización actual. NXImage es incluso capaz de convertir una implementación en otra si es necesario. El aspecto interesante de esta variante del Bridge es que NXImage puede almacenar más de una implementación de NXImageRep al mismo tiempo.

PATRONES RELACIONADOS

El patrón Abstract Factory (79) puede crear y configurar un Bridge.

El patrón Adapter (131) está orientado a conseguir que trabajen juntas clases que no están relacionadas. Normalmente se aplica a sistemas que ya han sido diseñados. El patrón Bridge, por otro lado, se usa al comenzar un diseño para permitir que abstracciones e implementaciones varíen independientemente unas de otras.

COMPOSITE (COMPUESTO)

Estructural de Objetos

PROPÓSITO

Compone objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

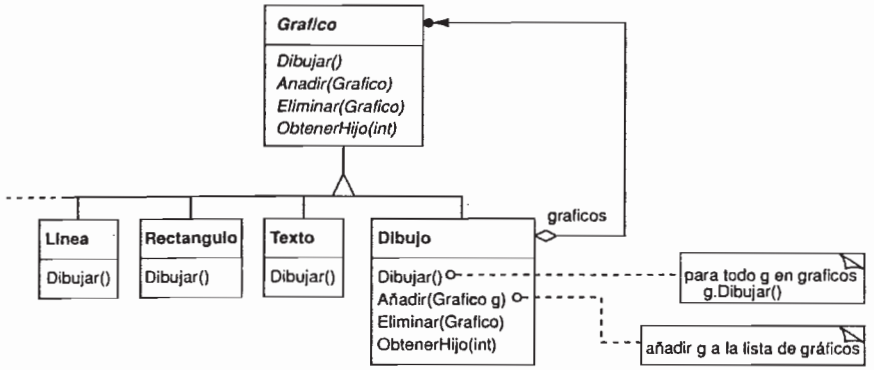
MOTIVACIÓN

Las aplicaciones gráficas como los editores de dibujo y los sistemas de diseño de circuitos permiten a los usuarios construir diagramas complejos a partir de componentes simples. El usuario puede agrupar componentes para formar componentes más grandes, que a su vez pueden agruparse para formar componentes aún mayores. Una implementación simple podría definir clases para primitivas gráficas como Texto y Linea, más otras clases que actúen como contenedoras de estas primitivas.

Pero hay un problema con este enfoque: el código que usa estas clases debe tratar de forma diferente a los objetos primitivos y a los contenedores, incluso aunque la mayor parte del tiempo el usuario los trate de forma idéntica. Tener que distinguir entre estos objetos hace que la aplicación sea más compleja. El patrón Composite describe cómo usar la composición recursiva para que los clientes no tengan que hacer esta distinción.

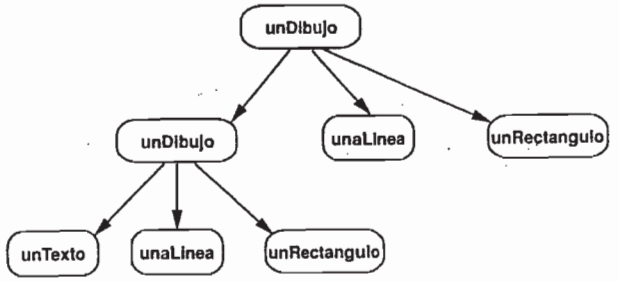
La clave del patrón Composite es una clase abstracta que representa *tanto* a primitivas *como* a sus contenedores. Para el sistema gráfico, esta clase es Grafico. Grafico declara operaciones como Dibujar que son específicas de objetos gráficos. También declara operaciones que comparten todos los objetos compuestos, tales como operaciones para acceder a sus hijos y para gestionarlos.

Las subclases Linea, Rectangulo y Texto (*véase* el diagrama de clases siguiente) definen objetos gráficos primitivos. Estas clases implementan Dibujar para dibujar líneas, rectángulos y texto, respectivamente. Como los gráficos primitivos no tienen gráficos hijos, ninguna de estas clases implementa operaciones relacionadas con los hijos.



La clase Dibujo define una agregación de objetos Grafico. Dibujo implementa Dibujar para que llame al Dibujar de sus hijos, y añade operaciones relacionadas con los hijos. Como la interfaz de Dibujo se ajusta a la interfaz de Grafico, los objetos Dibujo pueden componer recursivamente otros Dibujos.

El siguiente diagrama muestra una típica estructura de objetos compuestos recursivamente por otros objetos Grafico compuestos:

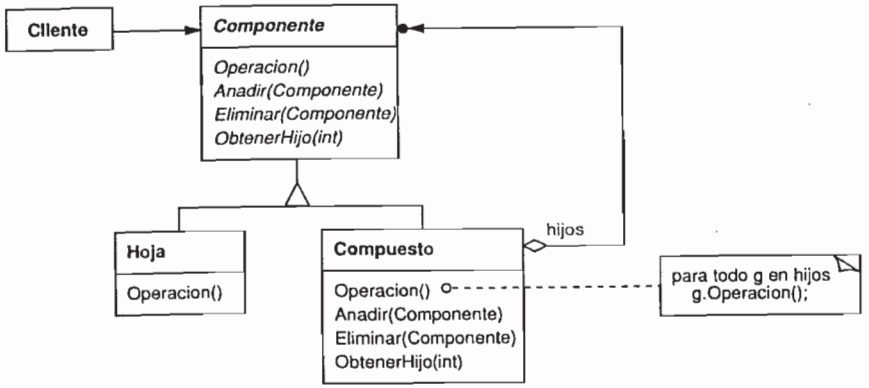


APLICABILIDAD

Use el patrón Composite cuando

- quiera representar jerarquías de objetos parte-todo.
- quiera que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme.

ESTRUCTURA



Una estructura de objetos Compuestos típica puede parecerse a esto:

PARTICIPANTES

- **Componente** (Grafico)
  - declara la interfaz de los objetos de la composición.
  - implementa el comportamiento predeterminado de la interfaz que es común a todas las clases.
  - declara una interfaz para acceder a sus componentes hijos y gestionarlos.
  - (opcional) define una interfaz para acceder al padre de un componente en la estructura recursiva y, si es necesario, la implementa.
- **Hoja** (Rectangulo, Linea, Texto, etc.)
  - representa objetos hoja en la composición. Una hoja no tiene hijos.
  - define el comportamiento de los objetos primitivos de la composición.
- **Compuesto** (Dibujo)
  - define el comportamiento de los componentes que tienen hijos.
  - almacena componentes hijos.
  - implementa las operaciones de la interfaz Componente relacionadas con los hijos.
- **Cliente**
  - manipula objetos en la composición a través de la interfaz Componente.

COLABORACIONES

Los Clientes usan la interfaz de la clase Componente para interactuar con los objetos de la estructura compuesta. Si el recipiente es una Hoja, la petición se trata correctamente. Si es un Compuesto, normalmente redirige las peticiones a sus componentes hijos, posiblemente realizando operaciones adicionales antes o después.

CONSECUENCIAS

El patrón Composite

- define jerarquías de clases formadas por objetos primitivos y compuestos. Los objetos primitivos pueden componerse en otros objetos más complejos, que a su vez pueden ser compuestos, y así de manera recurrente. Allí donde el código espere un objeto primitivo, también podrá recibir un objeto compuesto.

- simplifica el cliente. Los clientes pueden tratar uniformemente a las estructuras compuestas y a los objetos individuales. Los clientes normalmente no conocen (y no les debería importar) si están tratando con una hoja o con un componente compuesto. Esto simplifica el código del cliente, puesto que evita tener que escribir funciones con instrucciones `if` anidadas en las clases que definen la composición.
- facilita añadir nuevos tipos de componentes. Si se definen nuevas subclases Compuesto u Hoja, éstas funcionarán automáticamente con las estructuras y el código cliente existentes. No hay que cambiar los clientes para las nuevas clases Componente.
- puede hacer que un diseño sea demasiado general. La desventaja de facilitar añadir nuevos componentes es que hace más difícil restringir los componentes de un compuesto. A veces queremos que un compuesto sólo tenga ciertos componentes. Con el patrón Composite, no podemos confiar en el sistema de tipos para que haga cumplir estas restricciones por nosotros. En vez de eso, tendremos que usar comprobaciones en tiempo de ejecución.

IMPLEMENTACIÓN

Hay muchas cuestiones a tener en cuenta al implementar el patrón Composite:

1. *Referencias explícitas al padre.* Mantener referencias de los componentes hijos a sus padres puede simplificar el recorrido y la gestión de una estructura compuesta. La referencia al padre facilita ascender por la estructura y borrar un componente. Las referencias al padre también ayudan a implementar el patrón Chain of Responsibility (205).
   El lugar habitual donde definir la referencia al padre es en la clase Componente. Las clases Hoja y Compuesto pueden heredar la referencia y las operaciones que la gestionan.
   Con referencias al padre, es esencial mantener el invariante de que todos los hijos de un compuesto tienen como padre al compuesto que a su vez los tiene a ellos como hijos. El modo más fácil de garantizar esto es cambiar el padre de un componente *sólo* cuando se añade o se elimina a éste de un compuesto. Si se puede implementar una vez en las operaciones Anadir y Eliminar de la clase Compuesto entonces puede ser heredado por todas las subclases, conservando automáticamente el invariante.
2. *Compartir componentes.* Muchas veces es útil compartir componentes, por ejemplo para reducir los requisitos de almacenamiento. Pero cuando un componente no puede tener más de un padre, compartir componentes se hace más difícil.
   Una posible solución es que los hijos almacenen múltiples padres. Pero eso puede llevarnos a ambigüedades cuando se propaga una petición hacia arriba en la estructura. El patrón Flyweight (179) muestra cómo adaptar un diseño para evitar guardar los padres. Funciona en casos en los que los hijos pueden evitar enviar peticiones al padre externalizando parte de su estado, o todo.
3. *Maximizar la interfaz Componente.* Uno de los objetivos del patrón Composite es hacer que los clientes se despreocupen de las clases Hoja o Compuesto que están usando. Para conseguirlo, la clase Componente debería definir tantas operaciones comunes a las clases Compuesto y Hoja como sea posible. La clase Componente normalmente proporciona implementaciones predeterminadas para estas operaciones, que serán redefinidas por las subclases Hoja y Compuesto.
   No obstante, este objetivo a veces entra en conflicto con el principio de diseño de jerarquías de clases que dice que una clase sólo debería definir operaciones que tienen sentido en sus subclases. Hay muchas operaciones permitidas por Componente que no parecen tener sentido en las clases Hoja. ¿Cómo puede Componente proporcionar una implementación predeterminada para ellas?
   A veces un poco de creatividad muestra cómo una operación que podría parecer que sólo tiene sentido en el caso de los Compuestos puede implementarse para todos los Componentes, moviéndola a la clase Componente. Por ejemplo, la interfaz para acceder a los hijos es una parte fundamental de la clase Compuesto, pero no de las clases Hoja. Pero si vemos a una Hoja como un Componente que *nunca* tiene hijos, podemos definir una operación predeterminada en la clase Componente para acceder a los hijos que nunca *devuelve* ningún hijo. Las clases Hoja pueden usar esa implementación predeterminada, pero las clases Compuesto la reimplementarán para que devuelva sus hijos.
   Las operaciones de gestión de los hijos son más problemáticas, y se tratan en el siguiente punto.
4. *Declarar las operaciones de gestión de los hijos.* Aunque la clase Compuesto *implementa* las operaciones Anadir y Eliminar para controlar los hijos, un aspecto importante del patrón Compuesto es qué clases declaran estas operaciones en la jerarquía de clases Compuesto. ¿Deberíamos declarar estas operaciones en el Componente y hacer que tuvieran sentido en las clases Hoja, o deberíamos declararlas y definirlas sólo en Compuesto y sus subclases?

   La decisión implica un equilibrio entre seguridad y transparencia:

   - Definir la interfaz de gestión de los hijos en la raíz de la jerarquía de clases nos da transparencia, puesto que podemos tratar a todos los componentes de manera uniforme. Sin embargo, sacrifica la seguridad, ya que los clientes pueden intentar hacer cosas sin sentido, como añadir y eliminar objetos de las hojas.
   - Definir la gestión de los hijos en la clase Compuesto nos proporciona seguridad, ya que cualquier intento de añadir o eliminar objetos de las hojas será detectado en tiempo de compilación en un lenguaje estáticamente tipado, como C++. Pero perdemos transparencia, porque las hojas y los compuestos tienen interfaces diferentes.

   En este patrón hemos dado más importancia a la transparencia que a la seguridad. Si se opta por la seguridad, habrá ocasiones en las que perderemos información sobre el tipo y tendremos que convertir un componente en un compuesto. ¿Cómo podemos hacerlo sin recurrir a una conversión que no sea segura con respecto al tipo?
   Una posibilidad es declarar una operación `Compuesto* ObtenerCompuesto()` en la clase Componente. El Componente proporciona una operación por omisión que devuelve un puntero nulo. La clase Compuesto redefine esta operación para devolverse a sí misma a través de su puntero `this`:

```
class Compuesto;

class Componente {
public:
    //...
    virtual Compuesto* ObtenerCompuesto() { return 0; }
};

class Compuesto : public Componente {
public:
    void Anadir(Componente*);
    // ...
    virtual Compuesto* ObtenerCompuesto() { return this; }
};

class Hoja : public Componente {
    // ...
};
```

`ObtenerCompuesto` nos permite consultar a un componente para ver si es un compuesto. Podemos ejecutar `Anadir` y `Eliminar` con seguridad sobre el compuesto que devuelve.

```
Compuesto* unCompuesto = new Compuesto;
Hoja* unaHoja = new Hoja;

Componente* unComponente;
Compuesto* prueba;

unComponente = unCompuesto;
if (prueba = unComponente->ObtenerCompuesto()) {
    prueba->Anadir(new Hoja);
}

unComponente = unaHoja;

if (prueba = unComponente->ObtenerCompuesto()) {
    prueba->Anadir(new Hoja); // no añadirá a una hoja
}
```

Se pueden hacer comprobaciones similares para un Compuesto usando la construcción de C++ `dynamic_cast`.

Por supuesto, el problema aquí es que no tratamos a todos los componentes de manera uniforme. Tenemos que volver a realizar comprobaciones para diferentes tipos antes de emprender la acción apropiada.

La única forma de proporcionar transparencia es definir operaciones predeterminadas `Anadir` y `Eliminar` en Componente. Eso crea un nuevo problema: no hay modo de implementar `Componente::Anadir` sin introducir así mismo la posibilidad de que falle. Podríamos no hacer nada, pero eso omite una consideración importante; es decir, un intento de añadir algo a una hoja probablemente esté indicando un error. En ese caso, la operación `Anadir` produce basura. Podríamos hacer que borrara su argumento, pero eso no es lo que los clientes esperan.

Normalmente es mejor hacer que `Anadir` y `Eliminar` fallen de manera predeterminada (tal vez lanzando una excepción) si el componente no puede tener hijos o si el argumento de `Eliminar` no es un hijo del componente.

Otra posibilidad es cambiar ligeramente el significado de "eliminar". Si el componente mantiene una referencia al padre podríamos redefinir `Componente::Eliminar` para eliminarse a sí mismo de su padre. No obstante, sigue sin haber una interpretación con sentido para `Anadir`.

5. *¿Debería implementar el Componente una lista de Componentes?* Podríamos estar tentados de definir el conjunto de hijos como una variable de instancia de la clase Componente en la que se declaren las operaciones de acceso y gestión de los hijos. Pero poner el puntero al hijo en la clase base incurre en una penalización de espacio para cada hoja, incluso aunque una hoja nunca tenga hijos. Esto sólo merece la pena si hay relativamente pocos hijos en la estructura.
6. *Ordenación de los hijos.* Muchos diseños especifican una ordenación de los hijos de Compuesto. En el ejemplo del Grafico, la ordenación puede reflejar el orden desde el frente hasta el fondo. Si los objetos Compuesto representan árboles de análisis, entonces las instrucciones compuestas pueden ser instancias de un Compuesto cuyos hijos deben estar ordenados de manera que reflejen el programa.

   Cuando la ordenación de los hijos es una cuestión a tener en cuenta, debemos diseñar las interfaces de acceso y gestión de hijos cuidadosamente para controlar la secuencia de hijos. El patrón Iterador (237) puede servirnos de guía.

7. *Caché para mejorar el rendimiento.* Si necesitamos recorrer composiciones o buscar en ellas con frecuencia, la clase Compuesto puede almacenar información sobre sus hijos que facilite el recorrido o la búsqueda, El Compuesto puede guardar resultados o simplemente información que le permita evitar parte del recorrido o de la búsqueda. Por ejemplo, la clase Dibujo del ejemplo de la sección de Motivación podría guardar la caja limítrofe de sus hijos. Mientras se dibuja o es seleccionado, esta caja previamente guardada permite que Dibujo no tenga que dibujar o realizar búsquedas cuando sus hijos no son visibles en la ventana actual.

   Los cambios en un componente requerirán invalidar la caché de sus padres. Esto funciona mejor cuando los componentes conocen a sus padres. Por tanto, si se va a usar almacenamiento caché se necesita definir una interfaz para decirle a los compuestos que su caché ya no es válida.
8. *¿Quién debería borrar los componentes?* En lenguajes sin recolección de basura, normalmente es mejor hacer que un Compuesto sea el responsable de borrar sus hijos cuando es destruido. Una excepción a esta regla es cuando los objetos Hoja son inmutables y pueden por tanto ser compartidos.
9. *¿Cuál es la mejor estructura de datos para almacenar los componentes?* Los objetos Compuesto pueden usar muchas estructuras de datos diferentes para almacenar sus hijos, incluyendo listas enlazadas, árboles, arrays y tablas de dispersión. La elección de la estructura de datos depende (como siempre) de la eficiencia. De hecho, ni siquiera es necesario usar una estructura de datos de propósito general. A veces los compuestos tienen una variable para cada hijo, aunque esto requiere que cada subclase de Compuesto implemente su propia interfaz de gestión. *Véase* el patrón Interpreter (225) para un ejemplo.

CÓDIGO DE EJEMPLO

Determinados equipos, como computadores y componentes estéreo, suelen estar organizados en jerarquías de parte-todo o de pertenencia. Por ejemplo, un chasis puede contener unidades y placas base, un bus puede contener tarjetas y un armario puede contener chasis, buses, etcétera. Dichas estructuras pueden modelarse de manera natural con el patrón Composite.

La clase `Equipo` define una interfaz para todos los equipos de la jerarquía de parte-todo.

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

    const char* Nombre() { return _nombre; }

    virtual Vatio Potencia();
    virtual Moneda PrecioNeto();
    virtual Moneda PrecioConDescuento();

    virtual void Anadir(Equipo*);
    virtual void Eliminar(Equipo*);
    virtual Iterador<Equipo*>* CrearIterador();
protected:
    Equipo(const char*);
private:
    const char* _nombre;
};
```

Equipo declara operaciones que devuelven los atributos de un equipo, como su consumo y coste. Las subclases implementan estas operaciones para determinados tipos de equipos. Equipo también declara una operación CrearIterador que devuelve un Iterador (*véase* el Apéndice C) para acceder a sus partes. La implementación predeterminada de esta operación devuelve un IteradorNulo, que itera sobre el conjunto vacío.

Las subclases de Equipo podrían incluir clases Hoja que representen unidades de disco, circuitos integrados e interruptores:

```
class Disquetera : public Equipo {
public:
    Disquetera(const char*);
    virtual ~Disquetera();

    virtual Vatio Potencia();
    virtual Moneda PrecioNeto();
    virtual Moneda PrecioConDescuento();
};
```

EquipoCompuesto es la clase base de los equipos que contienen otros equipos. Es también una subclase de Equipo.

```
class EquipoCompuesto : public Equipo {
public:
    virtual ~EquipoCompuesto();

    virtual Vatio Potencia();
    virtual Moneda PrecioNeto();
    virtual Moneda PrecioConDescuento();

    virtual void Anadir(Equipo*);
    virtual void Eliminar(Equipo*);
    virtual Iterador<Equipo*>* CrearIterador();

protected:
    EquipoCompuesto(const char*);
private:
    Lista<Equipo*> _equipo;
};
```

EquipoCompuesto define las operaciones para acceder a sus componentes y recorrerlos. Las operaciones Anadir y Eliminar insertan y borran equipos en la lista de equipos almacenados en el miembro _equipo. La operación CrearIterador devuelve un iterador (concretamente, una instancia de IteradorLista) para recorrer la lista.

Una implementación predeterminada de PrecioNeto podría usar CrearIterador para sumar los precios netos de los equipos que lo componen:[2]

```
Moneda EquipoCompuesto::PrecioNeto () {
    Iterador<Equipo*>* i = CrearIterador();
    Moneda total = 0;

    for (i->Primero(); !i->HaTerminado(); i->Siguiente()) {
        total += i->ElementoActual()->PrecioNeto();
    }
    delete i;
    return total;
}
```

Ahora podemos representar un chasis de computadora como una subclase de EquipoCompuesto llamada Chasis. Chasis hereda las operaciones relativas a los hijos de EquipoCompuesto.

```
class Chasis : public EquipoCompuesto {
public:
    Chasis(const char*);
    virtual ~Chasis();

    virtual Vatio Potencia();
    virtual Moneda PrecioNeto();
    virtual Moneda PrecioConDescuento();
};
```

Podemos definir otros contenedores de equipos tales como Armario y Bus de forma similar. Eso nos da todo lo necesario para ensamblar componentes en una computadora personal (bastante sencillo):

```
Armario* armario = new Armario("Armario de PC");
Chasis* chasis = new Chasis("Chasis de PC");

armario->Anadir(chasis);

Bus* bus = new Bus("Bus MCA");
bus->Anadir(new Tarjeta("Token Ring de 16 Mbs "));

chasis->Anadir(bus);
chasis->Anadir(new Disquetera("Disquetera de 3,5 pulgadas"));

cout << "El precio neto es " << chasis->PrecioNeto() << endl;
```

USOS CONOCIDOS

Se pueden encontrar ejemplos del patrón Composite en casi todos los sistemas orientados a objetos. La clase Vista original del Modelo/Vista/Controlador de Smalltalk [KP88] era un Compuesto, y prácticamente todos los toolkits o frameworks de interfaces de usuario han seguido sus pasos, incluyendo ET++ (con VObjects [WGM88]) e InterViews (Styles [LCI+92], Graphics [VL88] y Glyphs [CL90]). Merece la pena destacar que la Vista original del Modelo/Vista/Controlador tenía un conjunto de subvistas; en otras palabras, la clase View era tanto la clase Componente como la Compuesto. La versión 4.0 de Smalltalk-80 revisó el Modelo/Vista/Controlador con una clase VisualComponent que tenía como subclases View y CompositeView.

[2] Es fácil olvidarse de borrar el iterador una vez que se ha usado. El patrón Iterator muestra, en la página 245, cómo protegerse contra tales errores.

El framework para compiladores de Smalltalk RTL [JML92] hace un uso intensivo del patrón Composite. RTLExpression es una clase Componente para árboles de análisis. Tiene subclases, tales como BinaryExpression, que contienen objetos RTLExpression como hijos. Dichas clases definen una estructura compuesta para árboles de análisis. RegisterTransfer es la clase Componente para un programa en la forma intermedia de Single Static Assignment (SSA). Las subclases Hoja de RegisterTransfer definen diferentes asignaciones estáticas, como

- asignaciones primitivas que realizan una operación en dos registros y asignan el resultado a un tercero;
- una asignación con un registro fuente pero sin registro destino, que indica que el registro se usa después del retorno de una rutina; y
- una asignación con un registro destino pero sin origen, lo que indica que al registro se le asigna un valor antes de que empiece la rutina.

Otra subclase, RegisterTransferSet, es una clase Compuesto que representa asignaciones que modifican varios registros a la vez.

Otro ejemplo de este patrón tiene lugar en el dominio de las finanzas, donde una cartera de acciones agrupa valores individuales. Se pueden permitir agregaciones complejas de valores implementando una cartera como un Compuesto que se ajusta a la interfaz de un valor individual [BE93].

El patrón Command (215) describe cómo se pueden componer y secuenciar objetos Orden con una clase Compuesto OrdenMacro.

PATRONES RELACIONADOS

Muchas veces se usa el enlace al componente padre para implementar el patrón Chain of Responsibility (205).

El patrón Decorator (161) suele usarse junto con el Composite. Cuando se usan juntos decoradores y compuestos, normalmente ambos tendrán una clase padre común. Por tanto, los decoradores tendrán que admitir la interfaz Componente con operaciones como Anadir, Eliminar y ObtenerHijo.

El patrón Flyweight (179) permite compartir componentes, si bien en ese caso éstos ya no pueden referirse a sus padres.

Se puede usar el patrón Iterator (237) para recorrer las estructuras definidas por el patrón Composite.

El patrón Visitor (305) localiza operaciones y comportamiento que de otro modo estaría distribuido en varias clases Compuesto y Hoja.

DECORATOR (DECORADOR)

Estructural de Objetos

PROPÓSITO

Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

TAMBIÉN CONOCIDO COMO

Wrapper (Envoltorio)

MOTIVACIÓN

A veces queremos añadir responsabilidades a objetos individuales en vez de a toda una clase. Por ejemplo, un toolkit de interfaces de usuario debería permitir añadir propiedades (como bordes) o comportamientos (como capacidad de desplazamiento) a cualquier componente de la interfaz de usuario.

Un modo de añadir responsabilidades es a través de la herencia. Heredar un borde de otra clase pondría un borde alrededor de todas las instancias de la subclase. Sin embargo, esto es inflexible, ya que la elección del borde se hace estáticamente. Un cliente no puede controlar cómo y cuándo decorar el componente con un borde.

Un enfoque más flexible es encerrar el componente en otro objeto que añada el borde. Al objeto confinante se le denomina **decorador**. El decorador se ajusta a la interfaz del componente que decora de manera que su presencia es transparente a sus clientes. El decorador reenvía las peticiones al componente y puede realizar acciones adicionales (tales como dibujar un borde) antes o después del reenvío. Dicha transparencia permite anidar decoradores recursivamente, permitiendo así un número ilimitado de responsabilidades añadidas.