VISITOR (VISITANTE)

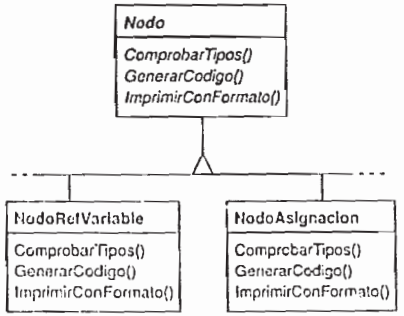
PROPÓSITO

Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

MOTIVACIÓN

Pensemos en un compilador que representa programas como árboles sintácticos abstractos. Necesitaremos realizar operaciones sobre dichos árboles sintácticos abstractos para llevar a cabo el análisis "semántico estático", como comprobar que todas las variables están definidas. También necesitaremos generar código. Por tanto, podríamos definir operaciones para la comprobación de tipos, la optimización de código, el análisis de flujo, comprobar que se asignan valores a las variables antes de su uso, etcétera. Más aún, podríamos usar los árboles sintácticos abstractos para imprimir con formato, reestructurar el programa, instrumentación de código o calcular diferentes métricas de un programa.

La mayoría de estas operaciones necesitarán tratar a los nodos que representan sentencias de asignación de forma distinta que a los que representan variables o expresiones aritméticas. Por tanto, habrá una clase para sentencias de asignación, otra para los accesos a variables, otra para expresiones aritméticas y así sucesivamente. El conjunto de clases de nodos depende del lenguaje que está siendo compilado, pór supuesto, pero no cambia mucho para un lenguaje dado.



Este diagrama muestra parte de la jerarquía de clases de Nodo. El problema aquí es que distribuir todas estas operaciones a través de las distintas clases de nodos conduce a un sistema que es difícil de comprender, mantener y cambiar. Será confuso tener la comprobación de tipos mezclada con el código de impresión o con el de análisis de flujo. Además, añadir una nueva operación normalmente obliga a recompilar todas estas clases. Sería mejor si cada nueva operación pudiera ser añadida por separado y las clases de nodos fuesen independientes de las operaciones que se aplican sobre ellas.

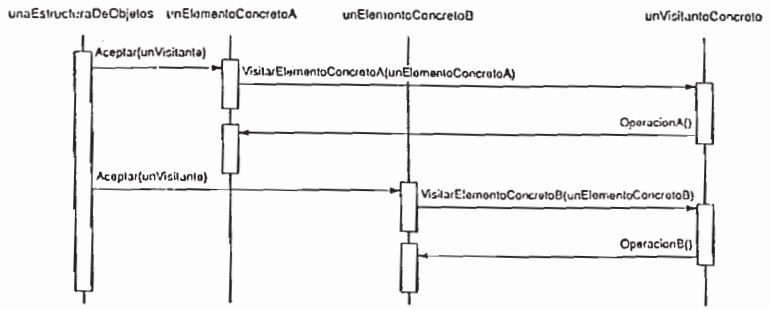
PARTICIPANTES

- **Visitante** (VisitanteNodo)
  - declara una operación Visitar para cada clase de operación ElementoConcreto de la estructura de objetos. El nombre y signatura de la operación identifican a la clase que envía la petición Visitar al visitante. Eso permite al visitante determinar la clase concreta de elemento que está siendo visitada. A continuación el visitante puede acceder al elemento directamente a través de su interfaz particular.
- **VisitanteConcreto** (VisitanteComprobacionDeTipos)
  - implementa cada operación declarada por Visitante. Cada operación implementa un fragmento del algoritmo definido para la clase correspondiente de la estructura. VisitanteConcreto proporciona el contexto para el algoritmo y guarda su estado local. Muchas veces este estado acumula resultados durante el recorrido de la estructura.
- **Elemento** (Nodo)
  - define una operación Aceptar que toma un visitante como argumento.
- **ElementoConcreto** (NodoAsignacion,NodoRefVariable)
  - implementa una operación Aceptar que toma un visitante como argumento.
- **EstructuraDeObjetos** (Programa)
  - puede enumerar sus elementos.
  - puede proporcionar una interfaz de alto nivel para permitir al visitante visitar a sus elementos.
  - puede ser un compuesto (*véase* el patrón Composite (151)) o una colección, como una lista o un conjunto.

COLABORACIONES

- Un cliente que usa el patrón Visitor debe crear un objeto VisitanteConcreto y a continuación recorrer la estructura, visitando cada objeto con el visitante.
- Cada vez que se visita a un elemento, éste llama a la operación del Visitante que se corresponde con su clase. El elemento se pasa a sí mismo como argumento de la operación para permitir al visitante acceder a su estado, en caso de que sea necesario.

El siguiente diagrama de interacción ilustra las colaboraciones entre una estructura de objetos, un visitante y dos elementos:



CONSECUENCIAS

Algunas de las ventajas e inconvenientes del patrón Visitor son las siguientes:

1. *El visitante facilita añadir nuevas operaciones.* Los visitantes facilitan añadir nuevas operaciones que dependen de los componentes de objetos complejos. Podemos definir una nueva operación sobre una estructura simplemente añadiendo un nuevo visitante. Si, por el contrario, extendiésemos la funcionalidad sobre muchas clases, habría que cambiar cada clase para definir una nueva operación.
2. *Un visitante agrupa operaciones relacionadas y separa las que no lo están.* El comportamiento similar no está desperdigado por las clases que definen la estructura de objetos; está localizado en un visitante. Las partes de comportamiento no relacionadas se dividen en sus propias subclases del visitante. Esto simplifica tanto las clases que definen los elementos como los algoritmos definidos por los visitantes. Cualquier estructura de datos específica de un algoritmo puede estar oculta en el visitante.
3. *Es difícil añadir nuevas clases de ElementoConcreto.* El patrón Visitor hace que sea complicado añadir nuevas subclases de Elemento. Cada ElementoConcreto nuevo da lugar a una nueva operación abstracta del Visitante y a su correspondiente implementación en cada clase VisitanteConcreto. A veces se puede proporcionar en Visitante una implementación predeterminada que puede ser heredada por la mayoría de los visitantes concretos, pero esto representa una excepción más que una regla.

   Por tanto la cuestión fundamental a considerar a la hora de aplicar el patrón Visitor es si es más probable que cambie el algoritmo aplicado sobre una estructura de objetos o las clases de los objetos que componen la estructura. La jerarquía de clases de Visitante puede ser difícil de mantener cuando se añaden nuevas clases de ElementoConcreto con frecuencia. En tales casos, es probablemente más fácil definir las operaciones en las clases que componen la estructura. Si la jerarquía de clases de Elemento es estable pero estamos continuamente añadiendo operaciones o cambiando algoritmos, el patrón Visitor nos ayudará a controlar dichos cambios.
4. *Visitar varias jerarquías de clases.* Un iterador (*véase* el patrón Iterator(237)) puede visitar a los objetos de una estructura llamando a sus operaciones a medida que los recorre. Pero un iterador no puede trabajar en varias estructuras de objetos con distintos tipos de elementos. Por ejemplo, la interfaz Iterador definida en la página 242 puede acceder únicamente a objetos del tipo `Elemento`:

```
template <class Elemento>
class Iterador {
    // ...
    Elemento ElementoActual() const;
};
```

Esto implica que todos los elementos que el iterador puede visitar tienen una clase padre común `Elemento`.

El patrón Visitor no tiene esta restricción. Puede visitar objetos que no tienen una clase padre común. Se puede añadir cualquier tipo de objeto a la interfaz de un Visitante. Por ejemplo, en

```
class Visitante {
public:
    // ...
    void VisitarMiTipo(MiTipo*);
    void VisitarTuTipo(TuTipo*);
};
```

MiTipo y TuTipo no tienen por qué estar relacionados en modo alguno a través de la herencia.

5. *Acumular el estado.* Los visitantes pueden acumular estado a medida que van visitando cada elemento de la estructura de objetos. Sin un visitante, este estado se pasaría como argumentos extra a las operaciones que realizan el recorrido, o quizá como variables globales.
6. *Romper la encapsulación.* El enfoque del patrón Visitor asume que la interfaz de ElementoConcreto es lo bastante potente como para que los visitantes hagan su trabajo. Como resultado, el patrón suele obligarnos a proporcionar operaciones públicas que accedan al estado interno de un elemento, lo que puede comprometer su encapsulación.

IMPLEMENTACIÓN

Cada estructura de objetos tendrá una clase Visitante asociada. Esta clase visitante abstracta declara una operación VisitarElementoConcreto para cada clase de ElementoConcreto que define la estructura de objetos. Cada operación Visitar del Visitante declara como argumento un ElementoConcreto en particular, permitiendo al Visitante acceder directamente a la interfaz del ElementoConcreto. Las clases VisitanteConcreto redefinen cada operación Visitar para implementar el comportamiento específico del visitante para la correspondiente clase ElementoConcreto.

La clase Visitante se declararía así en C++:

```
class Visitante {
public:
    virtual void VisitarElementoA(ElementoA*);
    virtual void VisitarElementoB(ElementoB*);

    // y así para otros elementos concretos
protected:
    Visitante();
};
```

Cada clase de ElementoConcreto implementa una operación `Aceptar` que llama a la operación `Visitar...` del visitante correspondiente a ese ElementoConcreto. De modo que la operación que es llamada al final depende tanto de la clase del elemento como de la clase del visitante.[10]

Los elementos concretos se declaran como

```
class Elemento {
public:
    virtual ~Elemento();
    virtual void Aceptar(Visitante&) = 0;
protected:
    Elemento();
};

class ElementoA : public Elemento {
public:
    ElementoA();
    virtual void Aceptar(Visitante& v) { v.VisitarElementoA(this); }
};

class ElementoB : public Elemento {
public:
    ElementoB();
    virtual void Aceptar(Visitante& v) { v.VisitarElementoB(this); }
};
```

Una clase `ElementoCompuesto` podría implementar `Aceptar` como sigue:

```
class ElementoCompuesto : public Elemento {
public:
    virtual void Aceptar(Visitante&);
private:
    Lista<Elemento*>* _hijos;
};

void ElementoCompuesto::Aceptar (Visitante& v) {
    IteradorLista<Elemento*> i(_hijos);

    for (i.Primero(); !i.HaTerminado(); i.Siguiente()) {
        i.ElementoActual()->Aceptar(v);
    }
    v.VisitarElementoCompuesto(this);
}
```

Éstas son otras dos cuestiones de implementacion que surgen al aplicar el patrón Visitor:

1. *Doble despacho.* El patrón Visitor nos permite añadir operaciones a clases sin modificar éstas. Esto se logra mediante una técnica llamada doble-despacho, la cual es muy conocida. De hecho, algunos lenguajes de programación la permiten directamente (por ejemplo, CLOS). Otros lenguajes, como C++ y Smalltalk, permiten el despacho-único.
   En lenguajes de despacho-único, dos son los criterios que determinan qué operación satisfará una petición: el nombre de la petición y el tipo del receptor. Por ejemplo, la operación a la que llamará una petición a GenerarCodigo dependerá del tipo de objeto nodo al que se le pida. En C++, llamar a `GenerarCodigo` sobre una instancia de `NodoRefVariable` llamará a `NodoRefVariable::GenerarCodigo` (que genera código para una referencia a una variable). Llamar a `GenerarCodigo` sobre un `NodoAsignacion` llamará a `NodoAsignacion::GenerarCodigo` (que generará código para una asignación). La operación que se ejecuta depende tanto del tipo del solicitante como del tipo del receptor.
   "Doble-despacho" simplemente significa que la operación que se ejecuta depende del tipo del solicitante y de los tipos de *dos* receptores. `Aceptar` es una operación de doble-despacho. Su significado depende de dos tipos: el del Visitante y el del Elemento. El doble-despacho permite a los visistantes solicitar diferentes operaciones en cada clase de elemento.[11]
   Ésta es la clave del patrón Visitor: la operación que se ejecuta depende tanto del tipo del Visitante como del tipo del Elemento visitado. En vez de enlazar las operaciones estáticamente en

[10] Podríamos usar la sobrecarga de funciones para dar a estas operaciones el mismo nombre, como por ejemplo `Visitar`, ya que las operaciones se diferencian por el parámetro que se les pasa. Esta sobrecarga tiene sus pros y sus contras. Por un lado, refuerza el hecho de que cada operación implica el mismo análisis, sólo que sobre un argumento distinto. Por otro lado, podría hacer menos obvio qué está ocurriendo en el punto de la llamada para alguien que esté leyendo el código. Realmente se reduce a si creemos que la sobrecarga de funciones es buena o no.

[11] Si podemos tener *doble*-despacho, ¿por qué no tener *triple*, *cuádruple* o cualquier otro número? En realidad, el doble-despacho no es más que un caso especial de despacho múltiple, en el cual la operación se elige en función de un número cualquiera de tipos (CLOS permite el despacho múltiple). Los lenguajes que permiten el despacho doble o múltiple ven reducida la necesidad del patrón Visitor.

la interfaz de Elemento, podemos fusionar las operaciones en un Visitante y usar Aceptar para hacer el enlace en tiempo de ejecución. Extender la interfaz de Elemento consiste en definir una nueva subclase de Visitante en vez de muchas nuevas subclases de Elemento.

2. *¿Quién es el responsable de recorrer la estructura de objetos?* Un visitante debe visitar cada elemento de la estructura de objetos. La cuestión es, ¿cómo lo logra? Podemos poner la responsabilidad del recorrido en cualquiera de estos tres sitios: en la estructura de objetos, en el visitante o en un objeto iterador aparte (*véase* el patrón Iterator (237)).
   Muchas veces es la estructura de objetos la responsable de la iteración. Una colección simplemente iterará sobre sus elementos, llamando a la operación Aceptar de cada uno. Un compuesto generalmente se recorrerá a sí mismo haciendo que cada operación Aceptar recorra los hijos del elemento y llame a Aceptar sobre cada uno de ellos, recursivamente.
   Otra solución es usar un iterador para visitar los elementos. En C++, podríamos usar un iterador interno o externo, dependiendo de qué está disponible y qué es más eficiente. En Smalltalk, normalmente usamos un iterador interno mediante do: y un bloque. Puesto que los iteradores internos son implementados por la estructura de objetos, usar un iterador interno se parece mucho a hacer que sea la estructura de objetos la responsable de la iteración. La principal diferencia estriba en que un iterador interno no provocará un doble-despacho —llamará a una operación del *visitante* con un *elemento* como argumento, frente a llamar a una operación del *elemento* con el *visitante* como argumento—. Pero resulta sencillo usar el patrón Visitor con un iterador interno si la operación del visitante simplemente llama a la operación del elemento sin recursividad.
   Incluso se podría poner el algoritmo de recorrido en el visitante, si bien en ese caso acabaríamos duplicando el código del recorrido en cada VisitanteConcreto de cada agregado ElementoConcreto. La principal razón para poner la estrategia de recorrido en el visitante es implementar un recorrido especialmente complejo, que dependa de los resultados de las operaciones de la estructura de objetos. En el Código de Ejemplo se verá un ejemplo para este caso.

CÓDIGO DE EJEMPLO

Como los visitantes suelen asociarse a compuestos, usaremos la clase Equipo que se definió en el Código de Ejemplo del patrón Composite (151) para ilustrar el patrón Visitor. Usaremos el patrón Visitor para definir operaciones para realizar el inventario de materiales y calcular el coste total de un equipo. Las clases Equipo son tan sencillas que realmente no sería necesario usar el patrón Visitor, pero lo haremos así para mostrar qué implicaciones conlleva la implementación de este patrón.

A continuación se muestra de nuevo la clase Equipo del Composite (151). La hemos aumentado con la operación Aceptar para que pueda funcionar con un visitante.

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

    const char* Nombre() { return _nombre; }

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

    virtual void Aceptar(VisitanteEquipo&);
protected:
    Equipo(const char*);
private:
    const char* _nombre;
};
```


Las operaciones de Equipo devuelven los atributos de un equipo, tales como su consumo de potencia y su coste. Las subclases redefinen estas operaciones de forma apropiada a cada tipo de equipo (por ejemplo, chasis, unidades y placas base).

La clase abstracta para todos los visitantes de equipos tiene una función virtual para cada subclase de equipo, como se muestra a continuación. Todas las funciones virtuales no hacen nada por omisión.

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

    virtual void VisitarDisquetera(Disquetera*);
    virtual void VisitarTarjeta(Tarjeta*);
    virtual void VisitarChasis(Chasis*);
    virtual void VisitarBus(Bus*);

    // y así para el resto de subclases concretas de Equipo
protected:
    VisitanteEquipo();
};
```

Las subclases de Equipo definen Aceptar básicamente de la misma forma: llamando a la operación de VisitanteEquipo que se corresponda con la clase que recibe la petición Aceptar, como en:

```
void Disquetera::Aceptar (VisitanteEquipo& visitante) {
   visitor.VisitarDisquetera(this);
}
```

Los equipos que contienen otros equipos (en concreto, las subclases de EquipoCompuesto en el patrón Composite) implementan Aceptar iterando sobre sus hijos y llamando a Aceptar sobre cada uno de ellos. A continuación llama a la operación Visitar como siempre. Por ejemplo, Chasis::Aceptar podría recorrer todas las partes del Chasis como sigue:

```
void Chasis::Aceptar (VisitanteEquipo& visitante) {
    for (
        IteradorLista <Equipo*> i(_partes);
        !i.HaTerminado();
        i.Siguiente()
    ) {
        i.ElementoActual()->Aceptar(visitar);
    }
    visitante.VisitarChasis(this);
}
```

Las subclases de VisitanteEquipo definen algoritmos concretos sobre la estructura de equipos. El VisitantePrecio calcula el coste de la estructura de equipos, calculando el precio neto de todos los equipos simples (por ejemplo, las disqueteras) y el precio con descuento de todos los equipos compuestos (como los chasis y buses).

```
class VisitantePrecio : public VisitanteEquipo {
public:
    VisitantePrecio();

    Moneda& ObtenerPrecioTotal();

    virtual void VisitarDisquetera(Disquetera*);
    virtual void VisitarTarjeta(Tarjeta*);
    virtual void VisitarChasis(Chasis*);
    virtual void VisitarBus(Bus*);
    // ...
private:
    Moneda _total;
};

void VisitantePrecio::VisitarDisquetera (Disquetera* e) {
    _total += e->PrecioNeto();
}

void VisitantePrecio::VisitarChasis (Chasis* e) {
    _total += e->PrecioConDescuento();
}
```

`VisitantePrecio` calculará el coste total de todos los nodos de la estructura de equipos. Nótese que `VisitantePrecio` elige la política de precios apropiada para una clase de equipo despachando a la correspondiente función miembro. Y lo que es más, podemos cambiar la política de precios de una estructura de equipo simplemente cambiando la clase `VisitantePrecio`.

Podemos definir un visitante para realizar un inventario como sigue:

```
class VisitanteInventario : public VisitanteEquipo {
public:
    VisitanteInventario();

    Inventario& ObtenerInventario();

    virtual void VisitarDisquetera(Disquetera*);
    virtual void VisitarTarjeta(Tarjeta*);
    virtual void VisitarChasis(Chasis*);
    virtual void VisitarBus(Bus*);
    // ...

private:
    Inventario _inventario;
};
```

El `VisitanteInventario` acumula los totales de cada tipo de equipo de la estructura de objetos. `VisitanteInventario` usa una clase `Inventario` que define una interfaz para añadir equipamiento (que no nos molestaremos en definir aquí).

```
void VisitanteInventario::VisitarDisquetera (Disquetera* e) {
    _inventario.Acumular(e);
}
```

```
    _inventario.Acumular(e);
}
```

Así es como podemos usar un `VisitanteInventario` en una estructura de equipos:

```
Equipo* componente;
VisitanteInventario visitante;

componente->Aceptar(visitor);
cout << "Inventario "
     << componente->Nombre()
     << visitante.ObtenerInventario();
```

Ahora veremos cómo implementar el ejemplo de Smalltalk del patrón Interpreter (*véase* la página 229) con el patrón Visitor. Como en el ejemplo anterior, éste es tan pequeño que el Visitante probablemente no nos aporte gran cosa, pero proporciona un buen ejemplo de cómo usar el patrón. Además, muestra una situación en la que la iteración es responsabilidad del visitante.

La estructura de objetos (expresiones regulares) se compone de cuatro clases, y todas ellas tienen un método `aceptar:` que toma un visitante como argumento. En la clase `ExpresionSecuencia`, el método `aceptar:` se define como

```
aceptar: unVisitante
    ^ unVisitante visitarSecuencia: self
```

En la clase `ExpresionRepetir`, el método `aceptar:` envía el mensaje `visitarRepetir:`. En la clase `ExpresionAlternativa`, envía el mensaje `visitarAlternativa:`. En la clase `ExpresionLiteral`, envía el mensaje `visitarLiteral:`.

Las cuatro clases también deben tener funciones de acceso que pueda usar el visitante. Para `ExpresionSecuencia` éstas son `expresion1` y `expresion2`; para `ExpresionAlternativa` son `alternativa1` y `alternativa2`; para `ExpresionRepetir` es su `repeticion`; y para `ExpresionLiteral` sus `componentes`.

La clase VisitanteConcreto es `VisitanteReconocedorER`. Es la responsable del recorrido porque su algoritmo de recorrido es irregular. La mayor irregularidad es que una `ExpresionRepetir` recorrerá repetidamente su componente. La clase `VisitanteReconocedorER` tiene una variable de instancia `estadoEntrada`. Sus métodos son, en esencia, los mismos que los métodos `reconocer:` de las clases de expresiones del patrón Interpreter, salvo que éstas sustituyen el argumento llamado `estadoEntrada` por el nodo con la expresión que está siendo reconocida. En cualquier caso, siguen devolviendo el conjunto de flujos que puede reconocer la expresión para identificar el estado actual.

```
visitarSecuencia: expSecuencia
    estadoEntrada := expSecuencia expresion1 aceptar: self.
    ^ expSecuencia expresion2 aceptar: self.

visitarRepetir: expRepetir
    | estadoFinal |
    estadoFinal := estadoEntrada copy.
    [estadoEntrada isEmpty]
        whileFalse:
            [estadoEntrada := expRepetir repeticion aceptar: self.
```

```
                estadoFinal addAll: estadoEntrada].
        ^ estadoFinal

    visitarAlternativa: expAlternativa
        | estadoFinal estadoOriginal |
        estadoOriginal := estadoEntrada.
        estadoFinal := expAlternativa alternativa1 aceptar: self.
        estadoEntrada := estadoOriginal.
        estadoFinal addAll: (expAlternativa alternativa2 aceptar: self).
        ^ estadoFinal

    visitarLiteral: expLiteral
        | estadoFinal tStream |
        estadoFinal := Set new.
        estadoEntrada
            do:
                [:stream | tStream := stream copy.
                    (tStream nextAvailable:
                        expLiteral componentes size
                    ) = expLiteral componentes
                        ifTrue: [estadoFinal add: tStream]
                ].
        ^ estadoFinal
```

USOS CONOCIDOS

El compilador Smalltalk-80 tiene una clase Visitante llamada ProgramNodeEnumerator. Se usa sobre todo para los algoritmos que analizan el código fuente. No se usa para generación de código o impresión con formato, aunque se podría.

IRIS Inventor [Str93] es un toolkit para desarrollar aplicaciones gráficas en 3-D. Inventor representa una escena tridimensional como una jerarquía de nodos, cada uno de los cuales representa o bien un objeto geométrico o bien uno de sus atributos. Operaciones como mostrar una escena o establecer una acción para un evento de entrada necesitan recorrer esta estructura de varias formas. Inventor lleva a cabo esto usando visitantes llamados "acciones". Hay diferentes visitantes para la visualización, el manejo de eventos, la búsqueda, guardar en ficheros o determinar las cajas limítrofes.

Para facilitar la adición de nuevos nodos, Inventor implementa un esquema de doble-despacho en C++. Este esquema se basa en la información de tipos en tiempo de ejecución y en una tabla bidimensional en la que las filas representan visitantes y las columnas clases de nodos. Las casillas guardan un puntero a la función asignada a esas clases de visitante y nodo.

Mark Linton acuñó el término "Visitor" en la especificación de Fresco Application Toolkit, de X Consortium [LP93].

PATRONES RELACIONADOS

Composite (151): los visitantes pueden usarse para aplicar una operación sobre una estructura de objetos definida por el patrón Composite.

Interpreter (225): se puede aplicar el patrón Visitor para llevar a cabo la interpretación.

DISCUSIÓN ACERCA DE LOS PATRONES DE COMPORTAMIENTO

ENCAPSULAR LO QUE VARÍA

Encapsular aquello que puede variar es el tema de muchos patrones de comportamiento. Cuando un determinado aspecto de un programa cambia con frecuencia, estos patrones definen un objeto que encapsula dicho aspecto. De esa manera, otras partes del programa pueden colaborar con el objeto siempre que dependan de ese aspecto. Los patrones normalmente definen una clase abstracta que describe el objeto encapsulado, y el patrón toma su nombre de ese objeto.[12] Por ejemplo,

- un objeto Estrategia encapsula un algoritmo (Strategy (289)),
- un objeto Estado encapsula un comportamiento dependiente del estado (State (279)),
- un objeto Mediador encapsula el protocolo entre objetos (Mediator (251)), y
- un objeto Iterador encapsula el modo en que se accede y se recorren los componentes de un objeto agregado (Iterator (237)).

Estos patrones describen aspectos de un programa que es probable que cambien. La mayoría de los patrones tienen dos tipos de objetos: el nuevo objeto que encapsula el aspecto y el objeto existente que usa el nuevo objeto creado. Normalmente, si no fuera por el patrón, la funcionalidad de los nuevos objetos sería una parte integral de los existentes. Por ejemplo, el código de una Estrategia probablemente estaría ligado al Contexto de la estrategia, y el código de un Estado se encontraría implementado directamente en el Contexto del estado.

Pero no todos los patrones de comportamiento de objetos dividen así la funcionalidad. Por ejemplo, el patrón Chain of Responsibility (205) trata con un número indeterminado de objetos (una cadena), cada uno de los cuales puede que ya exista en el sistema.

El patrón Chain of Responsibility muestra otra diferencia entre los patrones de comportamiento: no todos definen relaciones de comunicación estáticas entre las clases. El patrón Chain of Responsibility describe el modo de comunicación entre un número indefinido de objetos. Otros patrones usan objetos que se pasan como argumentos.

OBJETOS COMO ARGUMENTOS

Varios patrones introducen un objeto que *siempre* se usa como argumento. Uno de ellos es el Visitor (305). Un objeto Visitante es el argumento de una operación polimórfica Aceptar del objeto que visita. El visitante nunca se considera parte de estos objetos, incluso aunque la alternativa convencional al patrón consiste en distribuir el código del Visitante entre las clases de la estructura de objetos.

Otros patrones definen objetos que actúan como elementos mágicos que se pasan de un lado a otro y que más tarde pueden ser invocados. Tanto el patrón Command (215) como el Memento (261) entran en esta categoría. En el Command, el elemento representa una petición; en el Memento, representa el estado interno de un objeto en un momento concreto. En ambos casos, el elemento puede tener una representación interna compleja, pero los clientes nunca llegan a percibirla. No obstante, incluso aquí hay diferencias. El polimorfismo es importante en el patrón Command, ya que ejecutar el objeto Orden es una operación polimórfica. Por el contrario, la interfaz del Memento es tan limitada que éste sólo puede pasarse como un valor. Por tanto, es probable que no presente ninguna operación polimórfica a sus clientes.

[12] Este tema también afecta a otros tipos de patrones. Tanto el patrón Abstract Factory (79) como el Builder (89) y el Prototype (109), encapsulan el conocimiento sobre cómo se crean los objetos. El patrón Decorator (161) encapsula la responsabilidad que puede añadirse a un objeto. El patrón Bridge (141) separa una abstracción de su implementación, permitiéndolas variar de forma independiente.

El Mediator (251) y el Observer (269) son patrones rivales. La diferencia entre ellos es que el patrón Observer distribuye la comunicación introduciendo objetos Observador y Sujeto, mientras que un objeto Mediador encapsula la comunicación entre otros objetos.

En el patrón Observer, no hay ningún objeto individual que encapsule una restricción. En vez de eso, el Observador y el Sujeto deben cooperar para mantener la restricción. Los patrones de comunicación se determinan por el modo en que se interconectan los observadores y los sujetos: un sujeto individual normalmente tiene muchos observadores, y a veces el observador de un sujeto es un sujeto de otro observador. El patrón Mediator centraliza más que distribuye, ubicando en el mediador la responsabilidad de mantener la restricción.

Hemos descubierto que es más fácil hacer Observadores y Sujetos reutilizables que hacer Mediadores reutilizables. El patrón Observer promueve la separación y bajo acoplamiento entre el Observador y el Sujeto, lo que conduce a clases de grano más fino que son más aptas para ser reutilizadas.

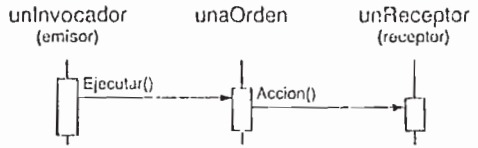
Por otro lado, es más fácil entender el flujo de comunicación en el Mediator que en el Observer. Los observadores y sujetos suelen conectarse nada más crearse, y es difícil ver más tarde en el programa cómo están conectados. Si conocemos el patrón Observer, debemos entender que el modo en que se conectan los observadores y los sujetos es importante, y también sabremos qué conexiones buscar. Sin embargo, la indirección introducida por este patrón sigue haciendo que el sistema sea difícil de entender.

Los observadores pueden parametrizarse en Smalltalk con mensajes para acceder al estado del sujeto, lo que los hace más reutilizables de lo que lo son en C++. Esto hace que en Smalltalk sea más atractivo el Observer que el Mediator. De ahí que un programador de Smalltalk generalmente use el Observer donde un programador de C++ usaría un Mediator.

DESACOPLAR EMISORES Y RECEPTORES

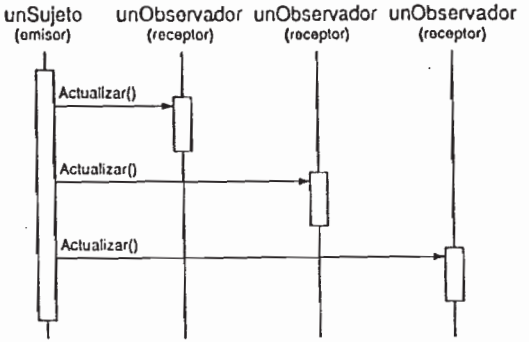
Cuando los objetos que colaboran se refieren unos a otros explícitamente, se vuelven dependientes unos de otros, y eso puede tener un impacto adverso sobre la división en capas y la reutilización de un sistema. Los patrones Command, Observer, Mediator y Chain of Responsibility tratan el problema de cómo desacoplar emisores y receptores, cada uno con sus ventajas e inconvenientes.

El patrón Command permite el desacoplamiento usando un objeto Orden que define un enlace entre un emisor y un receptor:



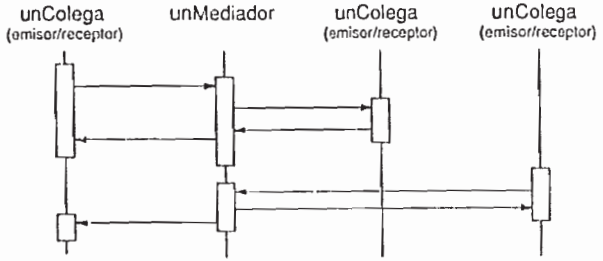
El objeto Orden proporciona una interfaz simple para emitir la petición (es decir, la operación Ejecutar). Definir la conexión emisor-receptor en un objeto aparte permite que el emisor funcione con diferentes receptores. Gracias a mantener al emisor desacoplado de los receptores es más fácil reutilizar los emisores. Más aún, es posible reutilizar el objeto Orden para parametrizar un receptor con diferentes emisores. El patrón Command necesita que haya una subclase por cada conexión emisor-receptor, si bien el patrón describe técnicas de implementación que evitan la herencia.

una interfaz para indicar cambios en los sujetos. Observer define un enlace más débil que Command entre el emisor y el receptor, ya que un sujeto puede tener múltiples observadores, cuyo número puede variar en tiempo de ejecución.



Las interfaces Sujeto y Observador del patrón Observer están diseñadas para posibles cambios en la comunicación. Por tanto, el patrón Observer es mejor para desacoplar objetos cuando hay dependencias de datos entre ellos.

El patrón Mediator desacopla los objetos haciendo que se refieran unos a otros indirectamente, a través del objeto Mediador.



Un objeto Mediador encarnina peticiones entre objetos Colega, y centraliza su comunicación. En consecuencia, los colegas sólo pueden hablar entre sí a través de la interfaz del Mediador. Dado que dicha interfaz es fija, el Mediador podría tener que implementar su propio mecanismo de despacho para una flexibilidad añadida. Las peticiones pueden ser codificadas junto con sus argumentos de tal forma que los compañeros pueden solicitar un conjunto de peticiones ilimitado.

El patrón Mediator puede reducir la herencia en un sistema, al centralizar el comportamiento de comunicación en una clase en vez de distribuirlo entre las subclases. Sin embargo, los esquemas de despacho *ad hoc* suelen disminuir la seguridad de tipos.

Por último, el patrón Chain of Responsibility desacopla al emisor del receptor pasándo la petición a lo largo de una cadena de receptores potenciales:

También es útil mostrar qué clases crean instancias de qué otras. Para ello usamos una flecha con la línea punteada, ya que OMT no lo permite. Llamamos a esto la relación "crea". La flecha apunta a la clase de la que se crea la instancia. En la figura B.1c, HerramientaDeCreacion crea objetos FormaLinea.

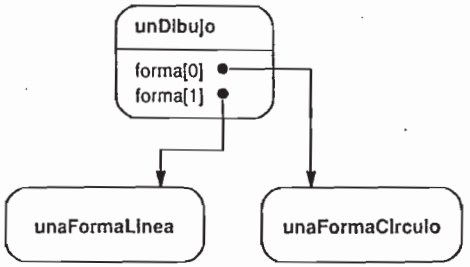
OMT también define un círculo relleno que representa "más de uno". Cuando el círculo aparece en el extremo de una referencia quiere decir que hay muchos objetos referenciados o agregados. La figura B.1c representa un agregado Dibujo que contiene múltiples objetos de tipo Forma.

Por último, hemos aumentado OMT con anotaciones en pseudocódigo que nos permiten esbozar las implementaciones de las operaciones. La figura B.1d muestra la anotación en pseudocódigo para la operación Dibujar de la clase Dibujo.

B.2. DIAGRAMA DE OBJETOS

Un diagrama de objetos muestra exclusivamente instancias. Representa una instantánea de los objetos de un patrón de diseño. Los objetos tienen por nombre "un*Algo*", donde *Algo* es la clase del objeto. Nuestro símbolo de un objeto (modificado ligeramente del estándar OMT) es un rectángulo con los bordes redondeados y una línea que separa el nombre del objeto de las referencias a otros objetos. Las flechas indican el objeto referenciado. La figura B.2 muestra un ejemplo.

Figura B.2: Notación de los diagramas de objetos

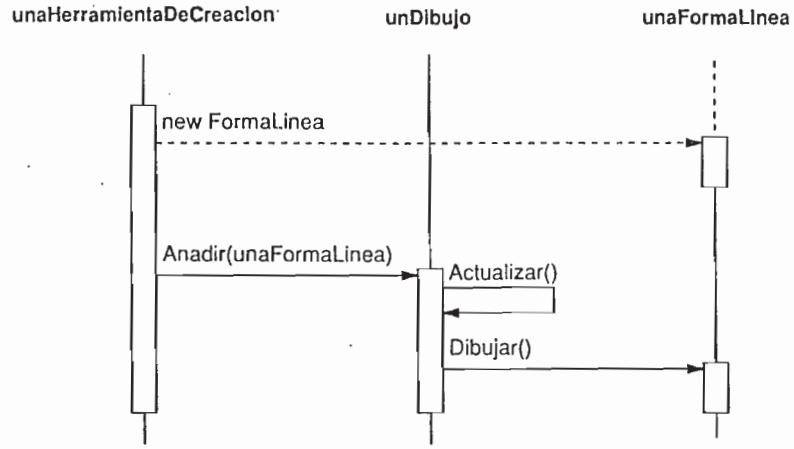


B.3. DIAGRAMA DE INTERACCIÓN

Un diagrama de interacción muestra el orden en que se ejecutan las peticiones entre objetos. La figura B.3 es un diagrama de interacción que muestra cómo se añade una forma a un dibujo.

En un diagrama de interacción, el tiempo fluye de arriba a abajo. Una línea vertical indica el tiempo de vida de un determinado objeto. La convención de nominación de objetos es la misma que la de los diagramas de objetos —el nombre de la clase con los artículos indeterminados "un" o "una" como prefijo (por ejemplo, unaForma) —. Si no se crea ninguna instancia del objeto hasta un tiempo después del instante inicial representado en el diagrama, entonces su línea vertical aparece punteada hasta el momento de la creación.

Figura B.3: Notación de los diagramas de interacción



Un rectángulo vertical indica que un objeto está activo; es decir, que está procesando una petición. La operación puede enviar peticiones a otros objetos; esto se indica con una flecha horizontal que apunta al objeto receptor. El nombre de la petición se muestra encima de la flecha. Una petición de creación de un objeto se indica con una flecha punteada. Una petición al mismo objeto emisor se representa con una flecha hacia sí mismo.

La figura B.3 muestra que la primera petición procede de unaHerramientaDeCreacion para crear unaFormaLinea. A continuación, se añade unaFormaLinea a unDibujo, lo que hace que unDibujo se envíe a sí mismo una petición Actualizar. Nótese que unDibujo envía una petición Dibujar a unaFormaLinea como parte de la operación Actualizar.