```
class TipoDeElemento {
public:
    void Procesar();
    // ...
};

Coleccion<TipoDeElemento*> unaColeccion;
EstadoIteracion* estado;

estado = unaColeccion.CrearEstadoInicial();

while (!unaColeccion.HaTerminado(estado)) {
    unaColeccion.ElementoActual(estado)->Procesar();
    unaColeccion.Siguiente(estado);
}
delete estado;
```

La interfaz de la iteración basada en el memento tiene dos beneficios interesantes:

1. Puede haber más de un estado para la misma colección (y lo mismo es cierto para el patrón Iterator (237)).
2. No necesita romper la encapsulación para permitir la iteración. El memento sólo es interpretado por la propia colección; nadie más tiene acceso a él. Otros enfoques para iterar requieren romper la encapsulación haciendo a las clases iterador amigas de las clases de sus colecciones (*véase* el patrón Iterator (237)). La situación es a la inversa en la implementación basada en el memento: `Coleccion` es amiga de `IteratorState`.

El toolkit de resolución de problemas QOCA guarda información incremental en mementos [HHMV92]. Los clientes pueden obtener un memento que represente la solución actual a un sistema de ecuaciones. El memento contiene sólo aquellas variables de las ecuaciones que han cambiado desde la última solución. Normalmente, para cada nueva solución sólo cambia un pequeño subconjunto de las variables del resolvente. Este subconjunto es suficiente para devolver el resolvente a su solución precedente; volver a soluciones anteriores requiere almacenar mementos de las soluciones intermedias. Por tanto, no se pueden establecer mementos en cualquier orden; QOCA se basa en un mecanismo de historial para revertir a soluciones anteriores.

PATRONES RELACIONADOS

Command (215): las órdenes pueden usar mementos para guardar el estado de las operaciones que pueden deshacerse.

Iterator (237): puede usar mementos para la iteración, tal y como acabamos de describir.

OBSERVER (OBSERVADOR)

Comportamiento de Objetos

PROPÓSITO

Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.

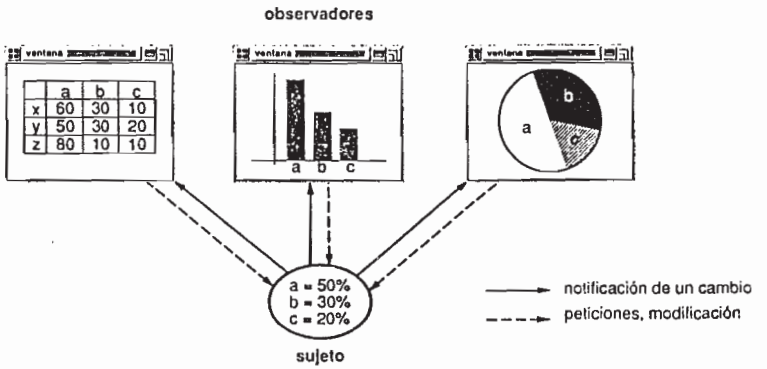
TAMBIÉN CONOCIDO COMO

Dependents (Dependientes), *Publish-Subscribe* (Publicar-Suscribir)

MOTIVACIÓN

Un efecto lateral habitual de dividir un sistema en una colección de clases cooperantes es la necesidad de mantener una consistencia entre objetos relacionados. No queremos alcanzar esa consistencia haciendo a las clases fuertemente acopladas, ya que eso reduciría su reutilización.

Por ejemplo, muchos toolkits de interfaces gráficas de usuario separan los aspectos de presentación de la interfaz de usuario de los datos de aplicación subyacentes [KP88, LVC89, P+88, WGM88]. Las clases que definen los datos de las aplicaciones y las representaciones pueden reutilizarse de forma independiente. También pueden trabajar juntas. Un objeto hoja de cálculo y un gráfico de barras pueden representar la información contenida en el mismo objeto de datos de aplicación usando diferentes representaciones. La hoja de cálculo y el gráfico de barras no se conocen entre sí, permitiendo así reutilizar sólo aquél que se necesite. Pero se *comportan* como si lo hicieran. Cuando el usuario cambia la información de la hoja de cálculo, la barra de herramientas refleja los cambios inmediatamente, y viceversa.



Este comportamiento implica que la hoja de cálculo y el gráfico de barras son dependientes del objeto de datos y, por tanto, se les debería notificar cualquier cambio en el estado de éste. Y no hay razón para limitar a dos el número de objetos dependientes; puede haber cualquier número de interfaces de usuario diferentes para los mismos datos.

El patrón Observer describe cómo establecer estas relaciones. Los principales objetos de este patrón son el **sujeto** y el **observador**. Un sujeto puede tener cualquier número de observadores dependientes de él. Cada vez que el sujeto cambia su estado se notifica a todos sus observadores. En respuesta, cada observador consultará al sujeto para sincronizar su estado con el estado de éste.

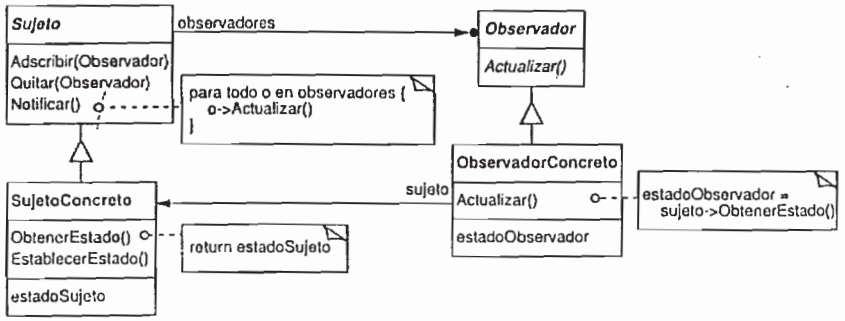
Este tipo de interacción también se conoce como **publicar-suscribir**. El sujeto es quien publica las notificaciones. Envía estas notificaciones sin tener que conocer quiénes son sus observadores. Pueden suscribirse un número indeterminado de observadores para recibir notificaciones.

APLICABILIDAD

Úsese el patrón Observer en cualquiera de las situaciones siguientes:

- Cuando una abstracción tiene dos aspectos y uno depende del otro. Encapsular estos aspectos en objetos separados permite modificarlos y reutilizarlos de forma independiente.
- Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse.
- Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quiénes son dichos objetos. En otras palabras, cuando no queremos que estos objetos estén fuertemente acoplados.

ESTRUCTURA



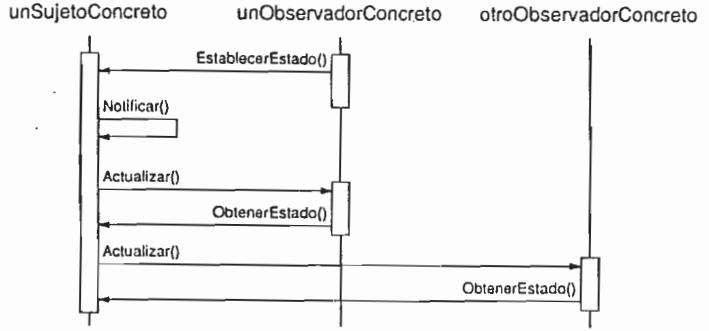
PARTICIPANTES

- **Sujeto**
  - conoce a sus observadores. Un sujeto puede ser observado por cualquier número de objetos Observador.
  - proporciona una interfaz para asignar y quitar objetos Observador.
- **Observador**
  - define una interfaz para actualizar los objetos que deben ser notificados ante cambios en un sujeto.
- **SujetoConcreto**
  - almacena el estado de interés para los objetos ObservadorConcreto.
  - envía una notificación a sus observadores cuando cambia su estado.
- **ObservadorConcreto**
  - mantiene una referencia a un objeto SujetoConcreto.
  - guarda un estado que debería ser consistente con el del sujeto.
  - implementa la interfaz de actualización del Observador para mantener su estado consistente con el del sujeto.

COLABORACIONES

- SujetoConcreto notifica a sus observadores cada vez que se produce un cambio que pudiera hacer que el estado de éstos fuera inconsistente con el suyo.
- Después de ser informado de un cambio en el sujeto concreto, un objeto ObservadorConcreto puede pedirle al sujeto más información. ObservadorConcreto usa esta información para sincronizar su estado con el del sujeto.

El siguiente diagrama de interacción muestra las colaboraciones entre un sujeto y dos observadores:



Nótese cómo el objeto Observador que inicializa la petición de cambio pospone su actualización hasta que obtiene una notificación del sujeto. Notificar no siempre es llamado por el sujeto. Puede ser llamado por un observador o por un tipo de objeto completamente diferente. La sección de Implementación examina algunas variantes comunes.

CONSECUENCIAS

El patrón Observador permite modificar los sujetos y observadores de forma independiente. Es posible reutilizar objetos sin reutilizar sus observadores, y viceversa. Esto permite añadir observadores sin modificar el sujeto u otros observadores.

Otras ventajas e inconvenientes del patrón Observer son los siguientes:

1. *Acoplamiento abstracto entre Sujeto y Observador.* Todo lo que un sujeto sabe es que tiene una lista de observadores, cada uno de los cuales se ajusta a la interfaz simple de la clase abstracta Observador. El sujeto no conoce la clase concreta de ningún observador. Por tanto el acoplamiento entre sujetos y observadores es mínimo.
   Gracias a que Sujeto y Observador no están fuertemente acoplados, pueden pertenecer a diferentes capas de abstracción de un sistema. Un sujeto de bajo nivel puede comunicarse e informar a un observador de más alto nivel, manteniendo de este modo intacta la estructura de capas del sistema. Si juntásemos al Sujeto y al Observador en un solo objeto, entonces el objeto resultante debería dividirse en dos capas (violando así la separación en capas) o estaría obligado a residir en una capa u otra (lo que puede comprometer la abstracción en capas).
2. *Capacidad de comunicación mediante difusión.* A diferencia de una petición ordinaria, la notificación enviada por un sujeto no necesita especificar su receptor. La notificación se envía automáticamente a todos los objetos interesados que se hayan suscrito a ella. Al sujeto no le importa cuántos objetos interesados haya; su única responsabilidad es notificar a sus observadores. Esto nos da la libertad de añadir y quitar observadores en cualquier momento. Se deja al observador manejar u obviar una notificación.
3. *Actualizaciones inesperadas.* Dado que los observadores no saben de la presencia de los otros, pueden no saber el coste último de cambiar el sujeto. Una operación aparentemente inofensiva sobre el sujeto puede dar lugar a una serie de actualizaciones en cascada de los observadores y sus objetos dependientes. Más aún, los criterios de dependencia que no están bien definidos o mantenidos suelen provocar falsas actualizaciones, que pueden ser muy difíciles de localizar.
   Este problema se ve agravado por el hecho de que el protocolo de actualización simple no proporciona detalles acerca de *qué* ha cambiado en el sujeto. Sin protocolos adicionales para ayudar a los observadores a descubrir qué ha cambiado, pueden verse obligados a trabajar duro para deducir los cambios.

IMPLEMENTACIÓN

En esta sección se examinan varias cuestiones relativas a la implementación del mecanismo de dependencia.

1. *Correspondencia entre los sujetos y sus observadores.* El modo más simple de que un sujeto conozca a los observadores a los que debería notificar es guardar referencias a ellos explícitamente en el sujeto. Sin embargo, dicho almacenamiento puede ser demasiado costoso cuando hay muchos sujetos y pocos observadores. Una solución es intercambiar espacio por tiempo usando una búsqueda asociativa (por ejemplo, mediante una tabla de dispersión –en inglés, tabla *hash*–) para mantener la correspondencia sujeto-observador. Así, un sujeto que no tenga observadores no incurrirá en ningún coste de almacenamiento. Por otro lado, este enfoque incrementa el coste de acceder a los observadores.
2. *Observar más de un sujeto.* Puede tener sentido en algunas situaciones que un observador dependa de más de un sujeto. Por ejemplo, una hoja de cálculo puede depender de más de un origen de datos. En tales casos es necesario extender la interfaz de Actualizar para que el observador sepa *qué* sujeto está enviando la notificación. El sujeto puede simplemente pasarse a sí mismo como parámetro en la operación Actualizar, permitiendo así al observador saber qué sujeto examinar.

3. *¿Quién dispara la actualización?* El sujeto y sus observadores se basan en el mecanismo de notificación para permanecer consistentes. Pero, ¿qué objeto llama realmente a Notificar para disparar la actualización? He aquí dos posibilidades:
   a. Hacer que las operaciones que establezcan el estado del Sujeto llamen a Notificar después de cambiar el estado del mismo. La ventaja de este enfoque es que los clientes no tienen que acordarse de llamar a Notificar sobre el sujeto. El inconveniente es que varias operaciones consecutivas provocarán varias actualizaciones consecutivas, lo que puede ser ineficiente.
   b. Hacer que los clientes sean responsables de llamar a Notificar en el momento adecuado. La ventaja aquí es que el cliente puede esperar a disparar la actualización hasta que se produzcan una serie de cambios de estado, evitando así las innecesarias actualizaciones intermedias. El inconveniente es que los clientes tienen la responsabilidad añadida de disparar la actualización. Eso hace que sea propenso a errores, ya que los clientes pueden olvidarse de llamar a Notificar.
4. *Referencias perdidas a los sujetos borrados.* Borrar un sujeto no debería producir referencias perdidas en sus observadores. Una manera de evitar esto es hacer que el sujeto notifique a sus observadores cuando va a ser borrado, para que estos puedan inicializar la referencia al sujeto. En general, borrar los observadores no suele ser una opción, ya que puede haber otros objetos que hagan referencia a ellos, y también pueden estar observando a otros sujetos.
5. *Asegurarse de que el estado del Sujeto es consistente consigo mismo antes de la notificación.* Es importante garantizar que el estado del Sujeto es consistente consigo mismo antes de llamar a Notificar, porque los observadores le piden al sujeto su estado actual mientras actualizan su propio estado.
   Es fácil violar involuntariamente esta regla de auto-consistencia cuando las operaciones de las subclases de Sujeto llaman a operaciones heredadas. Por ejemplo, en la siguiente secuencia de código la actualización se dispara cuando el sujeto se encuentra en un estado inconsistente:

```
void MiSujeto::Operacion (int nuevoValor) {
    ClaseBaseSujeto::Operacion(nuevoValor);
        // se dispara la notificación

    _miVar += nuevoValor;
        // se actualiza el estado de la subclase
      // (¡demasiado tarde!)
}
```

   Se puede salvar este escollo enviando notificaciones en métodos plantilla (Template Method (299)) de la clase abstracta Sujeto, definiendo una operación primitiva para que sea redefinida por las subclases y haciendo que Notificar sea la última operación del método plantilla, lo que garantizará que el objeto es consistente consigo mismo cuando las subclases redefinan las operaciones de Sujeto.

```
void Texto::Cortar (SeleccionDeTexto t) {
     SustituirSeleccion(t);   // redefinida en las subclases
     Notificar();
}
```

   Por cierto, siempre es una buena idea documentar qué operaciones del Sujeto disparan notificaciones.

6. *Evitar protocolos específicos del observador: los modelos push y pull.* Las implementaciones del patrón Observer suelen hacer que el sujeto envíe información adicional sobre el cambio. El sujeto pasa esta información como un parámetro de Actualizar. La cantidad de información puede variar mucho.
   En un extremo, al que llamaremos modelo ***push***, el sujeto envía a los observadores información detallada acerca del cambio, ya quieran éstos o no. El otro extremo es el **modelo *pull***; el sujeto no envía nada más que la notificación mínima, y los observadores piden después los detalles explícitamente.
   El modelo pull enfatiza la ignorancia del sujeto respecto a sus observadores, mientras que el modelo push asume que los sujetos saben algo sobre las necesidades de sus observadores. El modelo push puede hacer que los observadores sean menos reutilizables, ya que las clases Sujeto hacen suposiciones sobre las clases Observador que pudieran no ser siempre ciertas. Por otro lado, el modelo pull puede ser ineficiente, ya que las clases Observador deben determinar qué ha cambiado sin ayuda por parte del Sujeto.
7. *Especificar las modificaciones de interés explícitamente.* Se puede mejorar la eficiencia extendiendo la interfaz de registro del sujeto para permitir que los observadores registren sólo a aquellos eventos concretos que les interesen. Cuando ocurre uno de tales eventos, el sujeto informa únicamente a aquellos observadores que se han registrados como interesados en ese evento. Una manera de permitir esto es usar la noción de aspectos en los objetos Sujeto. Para registrarse como interesado en eventos particulares, los observadores se adscriben a sus sujetos usando

```
void Sujeto::Adscribir(Observador*, Aspecto& interes);
```

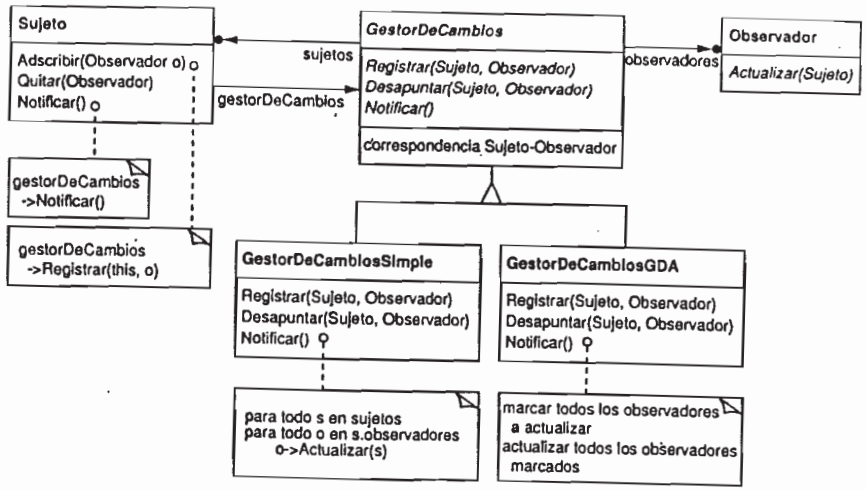
   donde `interes` especifica el evento de interés. En el momento de la notificación, el sujeto proporciona á sus observadores el aspecto que ha cambiado como un parámetro de la operación Actualizar. Por ejemplo:

```
void Observador::Actualizar(Sujeto*, Aspecto& interes);
```

8. *Encapsular la semántica de las actualizaciones complejas.* Cuando la relación de dependencia entre sujetos y observadores es particularmente compleja, puede ser necesario un objeto que mantenga estas relaciones. Llamaremos a este objeto un **GestorDeCambios**. Su propósito es minimizar el trabajo necesario para lograr que los observadores reflejen un cambio en su sujeto. Por ejemplo, si una operación necesita cambiar varios sujetos interdependientes, puede ser necesario asegurarse de que se notifica a sus observadores sólo después de que *todos* los sujetos han sido modificados, para evitar notificar a los observadores más de una vez.
   GestorDeCambios tiene tres responsabilidades:

   (a). Hace corresponder a un sujeto con sus observadores, proporcionando una interfaz para mantener dicha correspondencia. Esto elimina la necesidad de que los sujetos mantengan referencias a otros observadores y viceversa.
   (b). Define una determinada estrategia de actualización.
   (c). Actualiza todos los observadores dependientes a petición de un sujeto.

   El diagrama siguiente representa una implementación del patrón Observer basada en un GestorDeCambios simple. Hay dos tipos de GestorDeCambios. GestorDeCambiosSimple es simplista en el sentido de que siempre actualiza todos los observadores de cada sujeto. Por el contrario, GestorDeCambiosGDA maneja grafos dirigidos-acíclicos de dependencias entre sujetos y sus observadores. Es preferible un GestorDeCambiosGDA frente a un GestorDeCambiosSimple cuando un observador observa a más de un sujeto. En ese caso, un cambio en dos o más sujetos podría causar actualizaciones redundantes. El GestorDeCambiosGDA garantiza



que el observador sólo recibe una única actualización. GestorDeCambiosSimple está bien cuando las actualizaciones múltiples no constituyen un problema.
GestorDeCambios es una instancia del patrón Mediator (251). En general, sólo hay un único GestorDeCambios, y es conocido globalmente. El patrón Singleton (119) sería aquí de utilidad.

9. *Unir las clases Sujeto y Observador.* Las bibliotecas de clases escritas en lenguajes que carecen de herencia múltiple (como Smalltalk) generalmente no definen clases separadas Sujeto y Observador, sino que juntan sus interfaces en una clase. Eso permite definir un objeto que haga tanto de sujeto como de observador sin usar herencia múltiple. En Smalltalk, por ejemplo, las interfaces de Sujeto y Observador se definen en la clase raíz Object, haciéndolas así disponibles para todas las clases.

CÓDIGO DE EJEMPLO

Una clase abstracta define la interfaz Observador:

```
class Sujeto;

class Observador {
public:
    virtual ~ Observador();
    virtual void Actualizar(Sujeto* elSujetoQueCambio) = 0;
protected:
    Observador();
};
```

Esta implementación permite múltiples sujetos por cada observador. El sujeto que se pasa a la operación `Actualizar` permite que el observador determine qué objeto ha cambiado cuando éste observa más de uno.

De forma similar, una clase abstracta define la interfaz de Sujeto:

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

    virtual void Adscribir(Observador*);
    virtual void Quitar(Observador*);
    virtual void Notificar();
protected:
    Sujeto();
private:
    Lista<Observador*> *_observadores;
};

void Sujeto::Adscribir (Observador* o) {
    _observadores->Insertar(o);
}

void Sujeto::Quitar (Observador* o) {
    _observadores->Eliminar(o);
}

void Sujeto::Notificar () {
    IteradorLista<Observador*> i(_observadores);

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

`Reloj` es un sujeto concreto que almacena y mantiene la hora del día, notificando a sus observadores cada segundo. `Reloj` proporciona la interfaz para obtener unidades de tiempo por separado, como la hora, los minutos o los segundos.

```
class Reloj : public Sujeto {
public:
    Reloj();

    virtual int ObtenerHora();
    virtual int ObtenerMinuto();
    virtual int ObtenerSegundo();

    void Pulso();
};
```

La operación `Pulso` es llamada por un reloj interno a intervalos de tiempo regulares para proporcionar una base de tiempo fiable. `Pulso` actualiza el estado interno de `Reloj` y llama a `Notificar` para informar a los observadores del cambio:

```
void Reloj::Pulso () {
    // actualiza el estado del tiempo interno
    // ...
    Notificar();
}
```


Ahora podemos definir una clase `RelojDigital` que muestra el tiempo. Esta clase hereda su funcionalidad gráfica de una clase `Util`* proporcionada por un toolkit de interfaces de usuario. La interfaz del Observador se combina con la de `RelojDigital` heredando de `Observador`.

```
class RelojDigital: public Util, public Observador {
public:
    RelojDigital(Reloj*);
    virtual ~RelojDigital();

    virtual void Actualizar(Sujeto*);
        // redefine la operación de Observador

    virtual void Dibujar();
        // redefine la operación de Util;
        // define cómo dibujar el reloj digital
private:
    Reloj* _sujeto;
};

RelojDigital::RelojDigital (Reloj* s) {
    _sujeto = s;
    _sujeto->Adscribir(this);
}

RelojDigital:: RelojDigital () {
    _sujeto->Quitar(this);
}
```

Antes de que la operación `Actualizar` dibuje la apariencia del reloj, se comprueba que el sujeto de la notificación sea el sujeto del reloj:

```
void RelojDigital::Actualizar (Sujeto* elSujetoQueCambio) {
    if (elSujetoQueCambio == _sujeto) {
        dibujar();
    }
}

void RelojDigital::Dibujar () {
    // obtiene los nuevos valores del sujeto

    int hora = _sujeto->ObtenerHora();
    int minuto = _sujeto->ObtenerMinuto();
    // etc.
    // dibuja el reloj digital
```

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

```
}
```

Se puede definir una clase RelojAnalogico de la misma manera.

```
class RelojAnalogico : public Util, public Observador {
public:
    RelojAnalogico(Reloj*);
    virtual void Actualizar(Sujeto*);
    virtual void Dibujar();
    // ...
};
```

El siguiente código crea un RelojAnalogico y un RelojDigital que siempre muestra el mismo tiempo:

```
Reloj* reloj = new Reloj;
RelojAnalogico* relojAnalogico = new RelojAnalogico(reloj);
RelojDigital* relojDigital = new RelojDigital(reloj);
```

Cada vez que el reloj emite un pulso, los dos relojes se actualizarán y se volverán a dibujar de manera apropiada.

USOS CONOCIDOS

El primer y tal vez más importante ejemplo de patrón Observer aparece en el Modelo/Vista/Controlador de Smalltalk (MVC), el framework de interfaces de usuario en el entorno de Smalltalk [KP88]. La clase Modelo en MCV desempeña el papel de Sujeto, mientras que Vista es la clase base de los observadores. Smalltalk, ET++ [WGM88] y la biblioteca de clases THINK [Sym93b] proporcionan un mecanismo general de dependencia poniendo las interfaces de Sujeto y Observador en la clase base de todas las otras clases del sistema.

Otros toolkits de interfaces de usuario que emplean este patrón son InterViews [LVC89], Andrew Toolkit [P+88] y Unidraw [VL90]. InterViews define explícitamente las clases Observer y Observable (para los sujetos). Andrew las llama "vista" y "objeto de datos", respectivamente. Unidraw divide los objetos del editor gráfico en partes Vista (para los observadores) y Sujeto.

PATRONES RELACIONADOS

Mediator (251): encapsulando semánticas de actualizaciones complejas, el GestorDeCambios actúa como mediador entre sujetos y observadores.

Singleton (119): el GestorDeCambios puede usar el patrón Singleton para que sea único y globalmente accesible.

STATE (ESTADO)

Comportamiento de Objetos

PROPÓSITO

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

TAMBIÉN CONOCIDO COMO

Objects for States (Estados como Objetos)

MOTIVACIÓN

Pensemos en una clase ConexionTCP que representa una conexión de red. Un objeto ConexionTCP puede encontrarse en uno de los siguientes estados: Establecida, Escuchando o Cerrada. Cuando un objeto ConexionTCP recibe peticiones de otros objetos, les responde de distinta forma dependiendo de su estado actual. Por ejemplo, el efecto de una petición Abrir depende de si la conexión se encuentra en su estado Cerrada o en su estado Establecida. El patrón State describe cómo puede ConexionTCP exhibir un comportamiento diferente en cada estado.

La idea clave de este patrón es introducir una clase abstracta llamada EstadoTCP que representa los estados de la conexión de red. La clase EstadoTCP declara una interfaz común para todas las clases que representan diferentes estados operacionales. Las subclases de EstadoTCP implementan comportamiento específico de cada estado. Por ejemplo, las clases TCPEstablecida y TCPCerrada implementan comportamiento concreto de los estados Establecida y Cerrada de una ConexionTCP.

La clase ConexionTCP mantiene un objeto de estado (una instancia de una subclase de EstadoTCP) que representa el estado actual de la conexión TCP. La clase ConexionTCP delega todas las peticiones dependientes del estado en este objeto de estado. ConexionTCP usa su instancia de la subclase de EstadoTCP para realizar operaciones que dependen del estado de la conexión.