Composite Pattern

Introducción y nombre

Composite. Estructural. Permite construir objetos complejos componiendo de forma recursiva objetos similares en una estructura de árbol.

Intención

El patrón Composite sirve para construir algoritmos u objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol. Esto simplifica el tratamiento de los objetos creados, ya que al poseer todos ellos una interfaz común, se tratan todos de la misma manera.

También conocido como

Compuesto.

Motivación

Este patrón propone una solución basada en composición recursiva. Además, describe como usar correctamente esta recursividad. Esto permite tratar objetos individuales y composiciones de objetos de manera uniforme. Este patrón busca representar una jerarquía de objetos conocida como “parte-todo”, donde se sigue la teoría de que las "partes" forman el "todo", siempre teniendo en cuenta que cada "parte" puede tener otras "parte" dentro.

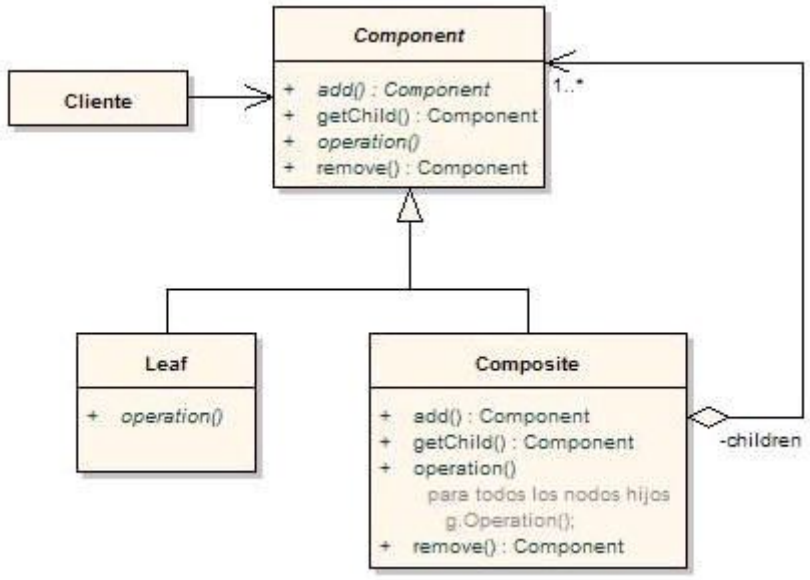
Solución

Se debe utilizar este patrón cuando:

Se busca representar una jerarquía de objetos como “parte-todo”.

Se busca que el cliente puede ignorar la diferencia entre objetos primitivos y compuestos (para que pueda tratarlos de la misma manera).

Diagrama UML



Participantes

Component

Declara la interface para los objetos de la composición.

Implementa un comportamiento común entre las clases.

Declara la interface para acceso y manipulación de hijos.

Declara una interface de manipulación a los padres en la estructura recursiva.

Leaf

Representa los objetos “hoja” (no poseen hijos).

Define comportamientos para objetos primitivos.

Composite

Define un comportamiento para objetos con hijos.

Almacena componentes hijos.

Implementa operaciones de relación con los hijos.

Cliente

Manipula objetos de la composición a través de Component.

Colaboraciones

Los clientes usan la interfaz de Component para interactuar con objetos en la estructura Composite. Si el receptor es una hoja, la interacción es directa. Si es un Composite, se debe llegar a los objetos “hijos”, y puede llevar a utilizar operaciones adicionales.

Consecuencias

Define jerarquías entre las clases.

Simplifica la interacción de los clientes.

Hace más fácil la inserción de nuevos hijos.

Hace el diseño más general.

Implementación

Hay muchas formas de llevar a cabo este patrón. Muchas implementaciones implican referencias explicitas al padre para simplificar la gestión de la estructura.

Si el orden de los hijos provoca problemas utilizar Iterator.

Compartir componentes reduce los requerimientos de almacenamiento.

Para borrar elementos hacer un compuesto responsable de suprimir los hijos.

La mejor estructura para almacenar elementos depende de la eficiencia: si se atraviesa la estructura con frecuencia se puede dejar información en los hijos. Por otro lado, el componente no siempre debería tener una lista de componentes, esto depende de la cantidad de componentes que pueda tener.

Código de muestra

Imaginemos una escuela, que esta compuesta de diferentes sectores (Dirección, Aulas, etc) y personas (profesores, alumnos, portero, etc). Se busca que la escuela pueda identificar las personas que posee y la edad de cada una. Todos deben identificarse con la misma interface:

[code]

```
public interface IPersonal {
   public void getDatosPersonal();
}

public class Composite implements IPersonal {
   private List<IPersonal> values = new ArrayList<IPersonal>();

   public void agrega(IPersonal personal) {
      values.add(personal);
   }

   @Override
   public void getDatosPersonal() {
      for (IPersonal personal : values) {
         personal.getDatosPersonal();
      }
   }
}

public class Escuela extends Composite {
}

public class Direccion extends Composite {
}

public class Aula extends Composite {
}

public class Persona implements IPersonal {

   private String nombre;
   private int edad;

   public Persona(String nombre, int edad) {
      this.edad = edad;
      this.nombre = nombre;
   }
```

```
    @Override
    public void getDatosPersonal() {
        String msg = "Me llamo " + nombre;
        msg = msg + ", tengo " + edad + " años";
        System.out.println(msg);
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}

public class Main {
    public static void main(String[] args) {
        Persona alumno1 = new Persona("Juan Perez", 21);
        Persona alumno2 = new Persona("Vanesa Gonzalez", 23);
        Persona alumno3 = new Persona("Martin Palermo", 26);
        Persona alumno4 = new Persona("Sebastian Paz", 30);
        Persona alumno5 = new Persona("Pepe Pillo", 22);

        Persona profesor1 = new Persona("Jacinto Dalo", 54);
        Persona profesor2 = new Persona("Guillermo Tei", 43);

        Persona director = new Persona("Dr Cito  Maximo", 65);
        Persona portero = new Persona("Don Manolo", 55);

        Escuela escuela = new Escuela();
        escuela.agrega(portero);

        Direccion direccion = new Direccion();
        direccion.agrega(director);

        Aula aula1 = new Aula();
        aula1.agrega(profesor1);
        aula1.agrega(alumno1);
        aula1.agrega(alumno2);
        aula1.agrega(alumno3);

        Aula aula2 = new Aula();
        aula2.agrega(profesor2);
        aula2.agrega(alumno4);
        aula2.agrega(alumno5);

        escuela.agrega(direccion);
        escuela.agrega(aula1);
        escuela.agrega(aula2);

        escuela.getDatosPersonal();
    }
}
```

Y la salida por consola es:
Me llamo Don Manolo, tengo 55 años

Me llamo Dr Cito Maximo, tengo 65 años
Me llamo Jacinto Dalo, tengo 54 años
Me llamo Juan Perez, tengo 21 años
Me llamo Vanesa Gonzalez, tengo 23 años
Me llamo Martin Palermo, tengo 26 años
Me llamo Guillermo Tei, tengo 43 años
Me llamo Sebastian Paz, tengo 30 años
Me llamo Pepe Pillo, tengo 22 años
[/code]

Cuándo utilizarlo

Este patrón busca formar un "todo" a partir de las composiciones de sus "partes". A su vez, estas "partes" pueden estar formadas de otras "partes" o de "hojas". Esta es la idea básica del Composite.

Java utiliza este patrón en su paquete de AWT (interfaces gráficas). En el paquete java.awt.swing el Componente es la clase Component, el Compuesto es la clase Container (sus Compuestos Concretos son Panel, Frame, Dialog y las hojas Label, TextField y Button.

Imaginemos que tenemos un software donde si pinta un gráfico. Algunas partes se pintan, mientras que otras partes de nuestro gráfico, de hecho, son otros gráficos y ciertas partes de estos últimos gráficos son, a su vez, otros gráficos. Este ejemplo es un caso típico para el patrón Composite.

Patrones relacionados

- Decorator: si se usan juntos normalmente tienen una clase común padre.
- Flyweight: permite compartir componentes.
- Iterator: sirve para recorrer las estructuras de los componentes.
- Visitor: localiza comportamientos en componentes y hojas.

Strategy Pattern

Introducción y nombre

Strategy. De Comportamiento. Convierte algoritmos en clases y los vuelve intercambiables.

Intención

Encapsula algoritmos en clases, permitiendo que éstos sean re-utilizados e intercambiables. En base a un parámetro, que puede ser cualquier objeto, permite a una aplicación decidir en tiempo de ejecución el algoritmo que debe ejecutar.

También conocido como Policy, Estrategia.

Motivación

La esencia de este patrón es encapsular algoritmos relacionados que son subclases de una superclase común, lo que permite la selección de un algoritmo que varia según el objeto y también le permite la variación en el tiempo. Esto se define en tiempo de ejecución. Este patrón busca desacoplar bifurcaciones inmensas con algoritmos dificultosos según el camino elegido.

Solución

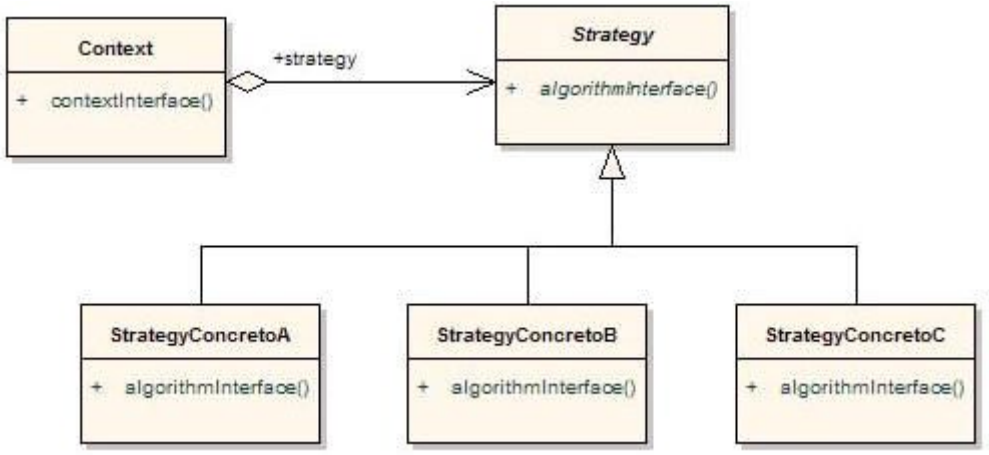
Este patrón debe utilizarse cuando:

Un programa tiene que proporcionar múltiples variantes de un algoritmo o comportamiento.

Es posible encapsular las variantes de comportamiento en clases separadas que proporcionan un modo consistente de acceder a los comportamientos.

Permite cambiar o agregar algoritmos, independientemente de la clase que lo utiliza.

Diagrama UML



Participantes

Strategy: declara una interfaz común a todos los algoritmos soportados.

StrategyConcreto: implementa un algoritmo utilizando la interfaz Strategy. Es la representación de un algoritmo.

Context: mantiene una referencia a Strategy y según las características del contexto, optará por una estrategia determinada..

Colaboraciones

Context / Cliente: solicita un servicio a Strategy y este debe devolver el resultado de un StrategyConcreto.

Consecuencias

Permite que los comportamientos de los Clientes sean determinados dinámicamente sobre un objeto base.

Simplifica los Clientes: les reduce responsabilidad para seleccionar comportamientos o implementaciones de comportamientos alternativos. Esto simplifica el código de los objetos Cliente eliminando las expresiones if y switch.

En algunos casos, esto puede incrementar también la velocidad de los objetos Cliente porque ellos no necesitan perder tiempo seleccionado un comportamiento.

Implementación

Los distintos algoritmos se encapsulan y el cliente trabaja contra el Context. Como hemos dicho, el cliente puede elegir el algoritmo que prefiera de entre los disponibles o puede ser el mismo Context el que elija el más apropiado para cada situación.

Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón Strategy. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución.

Código de muestra

Supongamos un caso donde un instituto educativo tiene una lista de alumnos ordenados por promedio. Dicho instituto suele competir en torneos intercolegiales, en campeonatos nacionales y también en competencias internacionales.

Obviamente la cantidad de gente que participa en cada competencia es distinta: por ejemplo, para los campeonatos locales participan los 3 mejores promedios, pero para las competencias internacionales, sólo se envía al mejor promedio de todos.

```
[code]
public class Alumno {

    private String nombre;
    private double promedio;

    public Alumno(String nombre, double promedio) {
        this.nombre = nombre;
        this.promedio = promedio;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public double getPromedio() {
        return promedio;
    }

    public void setPromedio(double promedio) {
        this.promedio = promedio;
    }
}

public interface ListadoStrategy {
    public List getListado(List lista);
}

public class CompetenciaInternacional implements ListadoStrategy {
    @Override
    public List getListado(List lista) {
        List resultado = new ArrayList();
        resultado.add(lista.get(0));
        return resultado;
    }
}

public class CompetenciaNacional implements ListadoStrategy {
    @Override
    public List getListado(List lista) {
        return lista.subList(0, 3);
    }
}

public class InterColegial implements ListadoStrategy {
    @Override
    public List getListado(List lista) {
        return lista.subList(0, 5);
```

```
    }
}

public class Colegio {

    private List<Alumno> alumnos;

    public Colegio() {
        alumnos = new ArrayList<Alumno>();
        Alumno a1 = new Alumno("Juan", 10);
        Alumno a2 = new Alumno("Sebastian", 9.5);
        Alumno a3 = new Alumno("Mario", 9);
        Alumno a4 = new Alumno("Pedro", 8.5);
        Alumno a5 = new Alumno("Matias", 8);
        Alumno a6 = new Alumno("Diego", 7.8);
        alumnos.add(a1);
        alumnos.add(a2);
        alumnos.add(a3);
        alumnos.add(a4);
        alumnos.add(a5);
        alumnos.add(a6);
    }

    public List<Alumno> getAlumnos() {
        return alumnos;
    }

    public void setPersonas(List<Alumno> alumnos) {
        this.alumnos = alumnos;
    }
}

public class Main {
    public static void main(String[] args) {

        Colegio colegio = new Colegio();
        List<Alumno> alumnos = colegio.getAlumnos();

        ListadoStrategy st = new CompetenciaNacional();
        // Se puede evitar que el cliente conozca los strategy concretos

        List rta = st.getListado(alumnos);

        // Veamos el resultado del patrón
        System.out.println("Los participantes son:");
        for (int i = 0; i < rta.size(); i++) {
            Alumno alumno = (Alumno) rta.get(i);
            System.out.println(alumno.getNombre());
        }
    }
}

La salida por consola es:
Los participantes son:
Juan
Sebastian
Mario
[/code]
```

Cuándo utilizarlo

Este patrón debe ser utilizado cuando un algoritmo es cambiado según un parámetro. Imaginemos una biblioteca de un instituto educativo que presta libros a los alumnos y profesores. Imaginemos que los alumnos pueden asociarse a la biblioteca pagando una mensualidad. Con lo cual un libro puede ser prestado a Alumnos,

Socios y Profesores.
Por decisión de la administración, a los socios se les prestará el libro más nuevo, luego aquel que se encuentre en buen estado y, por último, aquel que estuviese en estado regular.
En cambio, si un Alumno pide un libro, ocurre todo lo contrario. Por último, a los profesores sólo se les puede otorgar libros buenos o recién comprados.
Este caso es ideal para el patrón Strategy, ya que dependiendo de un parámetro (el tipo de persona a la que se le presta el libro) puede realizar una búsqueda con distintos algoritmos.

Patrones relacionados

Flyweight: si hay muchos objetos Cliente, los objetos StrategyConcreto puede estar mejor implementados como Flyweights.
El patrón Template Method maneja comportamientos alternativos a través de subclases más que a través de delegación.

Prototype Pattern

Introducción y nombre

Prototype. Creacional. Los objetos se crean a partir de un modelo.

Intención

Permite a un objeto crear objetos personalizados sin conocer su clase exacta o los detalles de cómo crearlos. El objetivo de este patrón es especificar prototipos de objetos a crear. Los nuevos objetos que se crearan se clonan de dichos prototipos. Vale decir, tiene como finalidad crear nuevos objetos duplicándolos, clonando una instancia creada previamente.
También conocido como: Patrón prototipo.

Motivación

Este patrón es necesario en ciertos escenarios es preciso abstraer la lógica que decide que tipos de objetos utilizará una aplicación, de la lógica que luego usarán esos objetos en su ejecución. Los motivos de esta separación pueden ser variados, por ejemplo, puede ser que la aplicación deba basarse en alguna configuración o parámetro en tiempo de ejecución para decidir el tipo de objetos que se debe crear.
Por otro lado, también aplica en un escenario donde sea necesario la creación de objetos parametrizados como "recién salidos de fábrica" ya listos para utilizarse, con la gran ventaja de la mejora de la performance: clonar objetos es más rápido que crearlos y luego setear cada valor en particular.

Solución

Situaciones en las que resulta aplicable.
Se debe aplicar este patrón cuando:

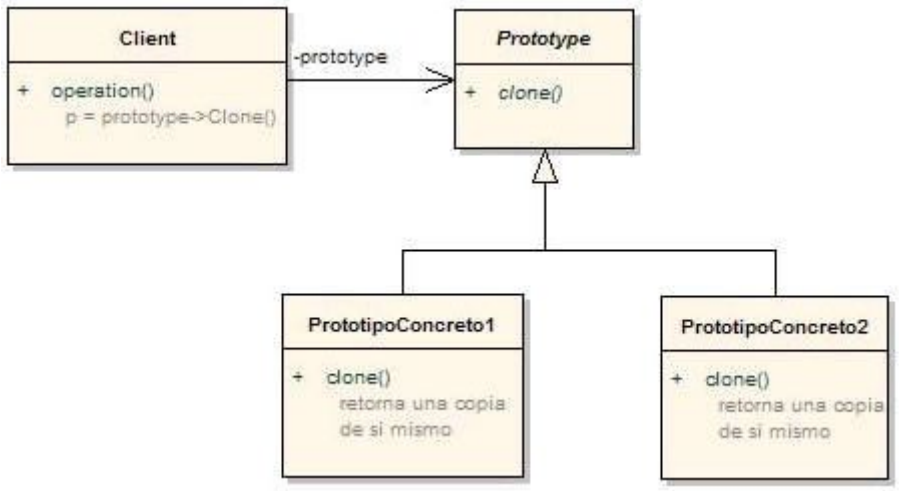
Las clases a instanciar sean especificadas en tiempo de ejecución.

Los objetos a crear tienen características comunes, en cuanto a los valores de sus atributos.

Se quiera evitar la construcción de una jerarquía Factory paralela a la jerarquía de clases de producto.

Diagrama UML

Estructura en un diagramas de clases.



Participantes

Responsabilidad de cada clase participante.

Cliente: solicita nuevos objetos.

Prototype: declara la interface del objeto que se clona. Suele ser una clase abstracta.

PrototypeConcreto: las clases en este papel implementan una operación por medio de la clonación de sí mismo.

Colaboraciones

Cliente: crea nuevos objetos pidiendo al prototipo que se clone.

Los objetos de Prototipo Concreto heredan de Prototype y de esta forma el patrón se asegura de que los objetos prototipo proporcionan un conjunto consistente de métodos para que los objetos clientes los utilicen.

Consecuencias

Un programa puede dinámicamente añadir y borrar objetos prototipo en tiempo de ejecución. Esta es una ventaja que no ofrece ninguno de los otros patrones de creación.

Esconde los nombres de los productos específicos al cliente.

Se pueden especificar nuevos objetos prototipo variando los existentes.

La clase Cliente es independiente de las clases exactas de los objetos prototipo que utiliza. y, además, no necesita conocer los detalles de cómo construir los objetos prototipo.

Clonar un objeto es más rápido que crearlo.

Se desacopla la creación de las clases y se evita repetir la instanciación de objetos con parámetros repetitivos.

Implementación

Dificultades, técnicas y trucos a tener en cuenta al aplicar el PD

Debido a que el patrón Prototye hace uso del método clone(), es necesaria una mínima explicación de su funcionamiento: todas las clases en Java heredan un método de la clase Object llamado clone. Un método clone de un objeto retorna una copia de ese objeto. Esto solamente se hace para instancias de clases que dan permiso para ser clonadas. Una clase da permiso para que su instancia sea clonada si, y solo si, ella implementa el interface Cloneable.

Por otro lado, es importante destacar que si va a variar el número de prototipos se puede utilizar un "administrador de prototipos". Otra opción muy utilizada es un Map como se ve en el ejemplo.

Debe realizarse un gestor de prototipos, para que realice el manejo de los distintos modelos a clonar.

Por último, se debe inicializar los prototipos concretos.

Código de muestra

Imaginemos que estamos haciendo el software para una empresa que vende televisores plasma y LCD. La gran mayoría de los TVs plasma comparten ciertas características: marca, color, precio, etc. Lo mismo ocurre con los LCD. Esto es normal en ciertos rubros ya que compran por mayor y se obtienen datos muy repetitivos en ciertos productos. También es muy normal en las fábricas: salvo algún serial, los productos son todos iguales.

Volviendo a nuestro ejemplo, se decidió realizar una clase genérica TV con los atributos básicos que debe tener un televisor:

[code]

```
public abstract class TV implements Cloneable {
   private String marca;
   private int pulgadas;
   private String color;
   private double precio;

   public TV(String marca, int pulgadas, String color, double precio) {
      this.marca = marca;
      this.pulgadas = pulgadas;
      this.precio = precio;
      this.color = color;
   }

   public String getMarca() {
      return marca;
   }

   public void setMarca(String marca) {
      this.marca = marca;
   }

   public int getPulgadas() {
      return pulgadas;
   }

   public void setPulgadas(int pulgadas) {
      this.pulgadas = pulgadas;
   }

   public String getColor() {
      return color;
   }

   public void setColor(String color) {
      this.color = color;
   }
```

```
    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

También tenemos los televisores específicos:

```
public class Plasma extends TV {
    private double anguloVision;
    private double tiempoRespuesta;

    public Plasma(String marca, int pulgadas, String color, double precio, double anguloVision, double
tiempoRespuesta) {
        super(marca, pulgadas, color, precio);
        this.anguloVision = anguloVision;
        this.tiempoRespuesta = tiempoRespuesta;
    }

    public double getAnguloVision() {
        return anguloVision;
    }

    public void setAnguloVision(double anguloVision) {
        this.anguloVision = anguloVision;
    }

    public double getTiempoRespuesta() {
        return tiempoRespuesta;
    }

    public void setTiempoRespuesta(double tiempoRespuesta) {
        this.tiempoRespuesta = tiempoRespuesta;
    }
}

public class LCD extends TV {
    private double costoFabricacion;

    public LCD(String marca, int pulgadas, String color, double precio, double costoFabricacion) {

        super(marca, pulgadas, color, precio);
        this.costoFabricacion = costoFabricacion;
    }

    public double getCostoFabricacion() {
        return costoFabricacion;
    }

    public void setCostoFabricacion(double costoFabricacion) {
        this.costoFabricacion = costoFabricacion;
    }
}
```

Debido a que se decidió realizar ciertos prototipos, estos se ven plasmados en la clase TvPrototype:

```
public class TvPrototype {
```

```
    private Map<String, TV> prototipos = new HashMap<String, TV>();

    public TvPrototype() {
        Plasma plasma = new Plasma("Sony", 21, "Plateado", 399.99, 90, 0.05);
        LCD lcd = new LCD("Panasonic", 42, "Plateado", 599.99, 290);

        prototipos.put("Plasma", plasma);
        prototipos.put("LCD", lcd);
    }

    public Object prototipo(String tipo) throws CloneNotSupportedException {
        return prototipos.get(tipo).clone();
    }
}
```

La invocación al método prototipo() sería:

```
public class Main {
    public static void main(String[] args) throws Exception {
        TvPrototype tvp = new TvPrototype();
        TV tv = (TV) tvp.prototipo("Plasma");

        System.out.println(tv.getPrecio());
    }
}
```

El resultado de la consola es: 399.99
[/code]

Cuándo utilizarlo

Este patrón debe ser utilizado cuando un sistema posea objetos con datos repetitivos: por ejemplo, si una biblioteca posee una gran cantidad de libros de una misma editorial, mismo idioma, etc.

Por otro lado, el hecho de poder agregar o eliminar prototipos en tiempo de ejecución es una gran ventaja que lo hace muy flexible.

Patrones relacionados

Abstract Factory: el patrón Abstract Factory puede ser una buena alternativa al Prototype donde los cambios dinámicos que Prototype permite para los objetos prototipo no son necesarios. Pueden competir en su objetivo, pero también pueden colaborar entre sí.

Facade: la clase cliente normalmente actúa comúnmente como un facade que separa las otras clases que participan en el patrón Prototype del resto del programa.

Factory Method: puede ser una alternativa al Prototype cuando los objetos prototipo nunca contiene más de un objeto.

Singleton: una clase Prototype suele ser Singleton.

Visitor Pattern

Introducción y nombre

Visitor. De Comportamiento. Busca separar un algoritmo de la estructura de un objeto.

Intención

Este patrón representa una operación que se aplica a las instancias de un conjunto de clases. Dicha operación se implementa de forma que no se modifique el código de las clases sobre las que opera.

También conocido como

Visitante.

Motivación

Si un objeto es el responsable de mantener un cierto tipo de información, entonces es lógico asignarle también la responsabilidad de realizar todas las operaciones necesarias sobre esa información. La operación se define en cada una de las clases que representan los posibles tipos sobre los que se aplica dicha operación, y por medio del polimorfismo y la vinculación dinámica se elige en tiempo de ejecución qué versión de la operación se debe ejecutar. De esta forma se evita un análisis de casos sobre el tipo del parámetro.

Solución

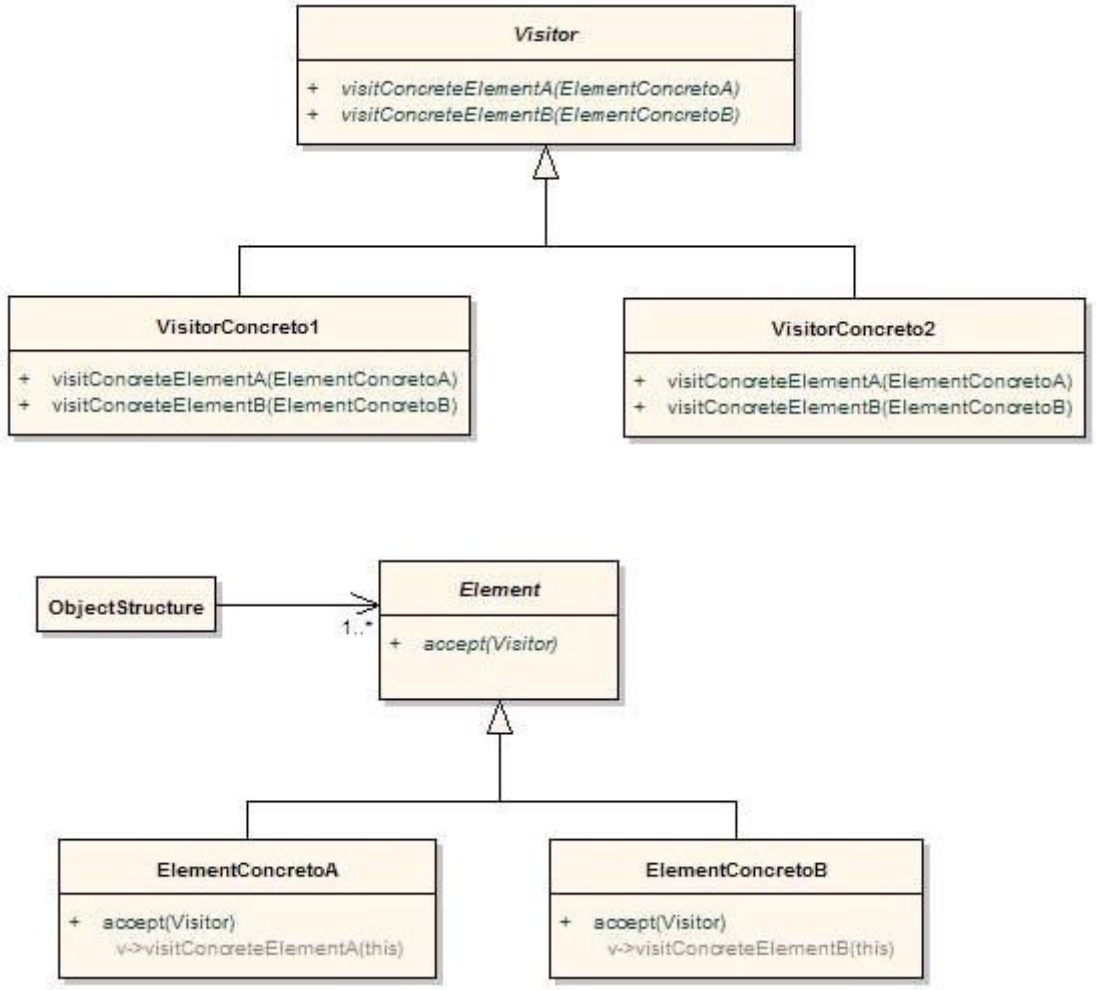
Este patrón debe utilizarse cuando:

Una estructura de objetos contiene muchas clases de objetos con distintas interfaces y se desea llevar a cabo operaciones sobre estos objetos que son distintas en cada clase concreta.

Se quieren llevar a cabo muchas operaciones dispares sobre objetos de una estructura de objetos sin tener que incluir dichas operaciones en las clases.

Las clases que definen la estructura de objetos no cambian, pero las operaciones que se llevan a cabo sobre ellas.

Diagrama UML



Participantes

Visitor: declara una operación de visita para cada uno de los elementos concretos de la estructura de objetos. Esto es, el método visit().

VisitorConcreto : implementa cada una de las operaciones declaradas por Visitor.

Element: define la operación que le permite aceptar la visita de un Visitor.

ConcreteElement: implementa el método accept() que se limita a invocar su correspondiente método del Visitor.

ObjectStructure: gestiona la estructura de objetos y puede ofrecer una interfaz de alto nivel para permitir a los Visitor visitar a sus elementos.

Colaboraciones

El Element ejecuta el método de visitar y se pasa a sí mismo como parámetro.

Consecuencias

Facilita la inclusión de nuevas operaciones.

Agrupa las operaciones relacionadas entre sí.

La inclusión de nuevos ElementsConcretos es una operación costosa.

Posibilita visitar distintas jerarquías de objetos u objetos no relacionados por un padre común.

Implementación

En patrón Visitor posee un conjunto de clases elemento que conforman la estructura de un objeto. Cada una de estas clases elemento tiene un método aceptar (accept()) que recibe al objeto visitador (visitor) como argumento. El visitador es una interfaz que tiene un método visit() diferente para cada clase elemento; por tanto habrá implementaciones de la interfaz visitor de la forma: visitorClase1, visitorClase2... visitorClaseN. El método accept() de una clase elemento llama al método visit de su clase. Los visitadores concretos pueden entonces ser escritas para hacer una operación en particular.

Cada método visit() de un visitador concreto puede ser pensado como un método que no es de una sola clase, sino de un par de clases: el visitador concreto y clase elemento particular. Así el patrón visitor simula el envío doble (en inglés éste término se conoce como Double-Dispatch).

Código de muestra

En Argentina todos los productos pagan IVA. Algunos productos poseen una tasa reducida. Utilizaremos el Visitor para solucionar este problema.

[code]
```
public interface Visitable {
   public double accept(Visitor visitor);
}

public class ProductoDescuento implements Visitable {
   private double precio;

   @Override
   public double accept(Visitor visitor) {
      return visitor.visit(this);
   }

   public double getPrecio() {
      return precio;
   }

   public void setPrecio(double precio) {
      this.precio = precio;
   }
}

public class ProductoDescuento implements Visitable {
   private double precio;

   @Override
   public double accept(Visitor visitor) {
      return visitor.visit(this);
   }

   public double getPrecio() {
      return precio;
   }

   public void setPrecio(double precio) {
      this.precio = precio;
   }
}

public class IVA implements Visitor {
   private final double impuestoNormal = 1.21;
   private final double impuestoReducido = 1.105;

   @Override
```

```
    public double visit(ProductoNormal normal) {
        return normal.getPrecio() * impuestoNormal;
    }

    @Override
    public double visit(ProductoDescuento reducido) {
        return reducido.getPrecio() * impuestoReducido;
    }
}
```

Probemos como funciona este ejemplo:

```
public class Main {
    public static void main(String[] args) {
        ProductoDescuento producto1 = new ProductoDescuento();
        producto1.setPrecio(100);
        ProductoNormal producto2 = new ProductoNormal();
        producto2.setPrecio(100);

        IVA iva = new IVA();
        double resultado1 = producto1.accept(iva);
        double resultado2 = producto2.accept(iva);

        System.out.println(resultado1);
        System.out.println(resultado2);
    }
}
```

La salida por consola es:
110.5
121.0
[/code]

Cuándo utilizarlo

Dado que este patrón separa un algoritmo de la estructura de un objeto, es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general.

Se debe utilizar este patrón si se quiere realizar un cierto número de operaciones, que no están relacionadas entre sí, sobre instancias de un conjunto de clases, y no se quiere “contaminar” a dichas clases.

Patrones relacionados

Interpreter: el visitor puede usarse para realizar la interpretación.

Composite: el visitor puede ayudar al Composite para aplicar una operación sobre la estructura del objeto definido en el composite.

Anti-Patrones

Anti-patrón

Un antipatrón de diseño es un patrón de diseño que conduce a una mala solución. Buscan evitar que los programadores tomen malas decisiones, partiendo de documentación disponible en lugar de simplemente la intuición.

Historia

El término nace con el mencionado libro Design Patterns de GOF, que básicamente son los patrones que vimos a lo largo de este curso. Los autores bautizaron dichas soluciones con el nombre de "patrones de diseño" por analogía con el mismo término, usado en arquitectura.

El libro Anti-Patterns (de William Brown, Raphael Malveau, Skip McCormick, Tom Mowbray y Scott Thomas) describe los antipatrones como la contrapartida natural al estudio de los patrones de diseño. El término fue utilizado por primera vez en 1995, por ello es que los antipatrones no se mencionan en el libro original de Design Patterns, puesto que éste es anterior.

Propósito

Los anti-patrones son ejemplos bien documentados de malas soluciones para problemas. El estudio formal de errores que se repiten permite que el desarrollador pueda reconocerlos más fácilmente y, por ello, podrá encaminarse hacia una mejor solución. Los desarrolladores deben evitar los antipatrones siempre que sea posible.

Utilización

Dado que el término antipatrón es muy amplio, suele ser utilizado en diversos contextos:

Antipatrones de desarrollo de software

- Antipatrones de gestión
- Antipatrones de gestión de proyectos
- Antipatrones generales de diseño de software
- Antipatrones de diseño orientado a objetos
- Antipatrones de programación
- Antipatrones metodológicos
- Antipatrones de gestión de la configuración

Antipatrones organizacionales

Avance del alcance (scope creep): Permitir que el alcance de un proyecto crezca sin el control adecuado.

Bloqueo del vendedor (vendor lock-in): Construir un sistema que dependa en exceso de un componente proporcionado por un tercero.

Diseño de comité (design by committee): Contar con muchas opiniones sobre un diseño, pero adolecer de falta de una visión unificada.

Escalada del compromiso (escalation of commitment): No ser capaz de revocar una decisión a la vista de que no ha sido acertada.

Funcionalitis acechante (creeping featuritis): Añadir nuevas funcionalidades al sistema en detrimento de su calidad.

Gestión basada en números (management by numbers): Prestar demasiada atención a criterios de gestión cuantitativos, cuando no son esenciales o difíciles de cumplir.

Gestión champiñón (mushroom management): Tratar a los empleados sin miramientos, sin informarles de las

decisiones que les afectan (manteniéndolos cubiertos y en la oscuridad, como los champiñones).
Gestión por que lo digo yo (management by perkele): Aplicar una gestión autoritaria con tolerancia nula ante las disensiones.
Migración de coste (cost migration): Trasladar los gastos de un proyecto a un departamento o socio de negocio vulnerable.
Obsolescencia continua (continuous obsolescence): Destinar desproporcionados esfuerzos a adaptar un sistema a nuevos entornos.
Organización de cuerda de violín (violin string organization): Mantener una organización afinada y en buen estado, pero sin ninguna flexibilidad.
Parálisis del análisis (analysis paralysis): Dedicar esfuerzos desproporcionados a la fase de análisis de un proyecto, eternizando el proceso de diseño iterando sobre la búsqueda de mejores soluciones o variantes.
Peligro moral (moral hazard): Aislar a quien ha tomado una decisión a raíz de las consecuencias de la misma.
Sistema de cañerías (stovepipe): Tener una organización estructurada de manera que favorece el flujo de información vertical, pero inhibe la comunicación horizontal.
Te lo dije (I told you so): Permitir que la atención se centre en que la desoída advertencia de un experto se ha demostrado justificada.
Vaca del dinero (cash cow): Pecar de autocomplacencia frente a nuevos productos por disponer de un producto legacy muy lucrativo.

Otros Patrones

Introducción

Dado el éxito que tuvo el famoso libre de Gof, comenzó una serie de epidemia de patrones. Hoy en día hay patrones de todo tipo que busca explicar las mejores prácticas de cada caso.
Los más conocidos son los siguientes.

Patrones de base de datos

Buscan abstraer y encapsular todos los accesos a la fuente de datos. El más conocido es el DAO. Técnicamente el patrón DAO (Data Access Object) pertenece a los patrones JEE, aunque se centra específicamente en la capa de persistencia de los datos.

Patrones de Arquitectura

Se centran en la construcción del software, sobretodo en la ingeniería del mismo. El más conocido es el MVC, que busca separar el diseño de la lógica del negocio.

Patrones JEE

Es un catálogo de patrones para usar en la tecnología JEE, es decir, en la parte empresarial que ofrece Java. Divide los patrones en 3 categorías: Capa de Presentación, de Negocios y de Integración. En esta última capa se encuentra el DAO.