Patrones de Diseño

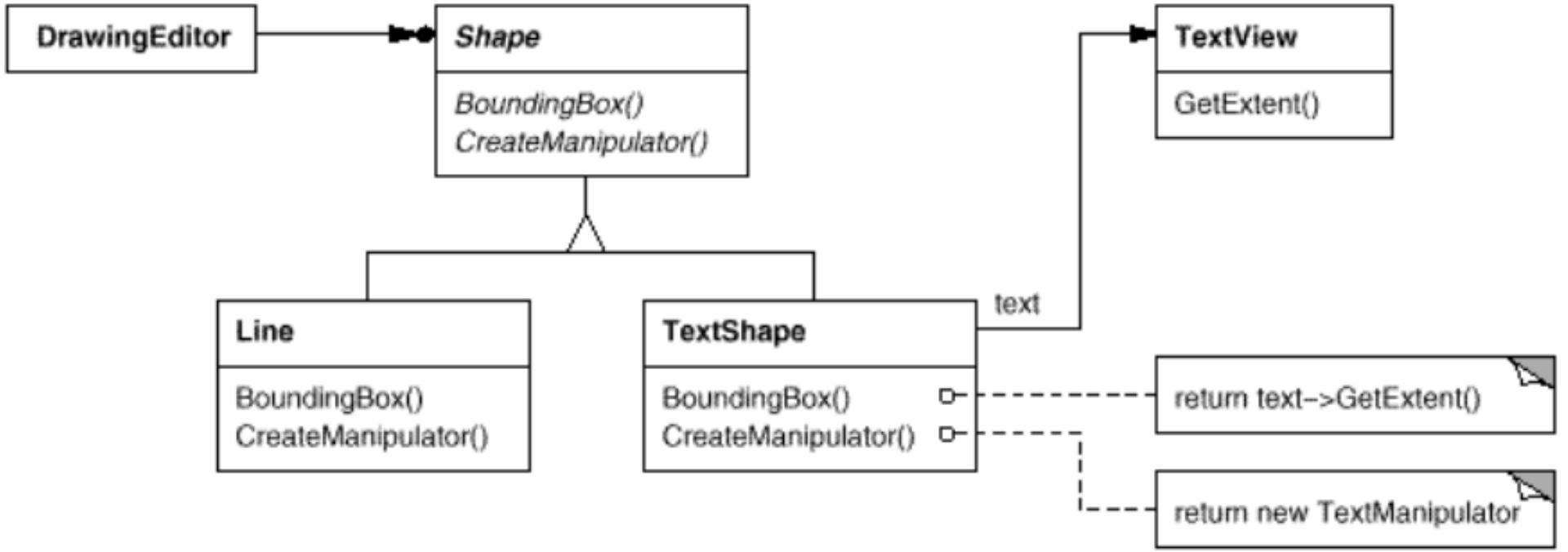
Patrones Estructurales: Adapter

a.k.a: Wrapper

Intención

Convierte la interfaz de una clase en otra interfaz que los clientes esperan.
Adapter permite que las clases trabajen juntas que de otra forma no podrían debido a la incompatibilidad entre interfaces.
De tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.

Motivación

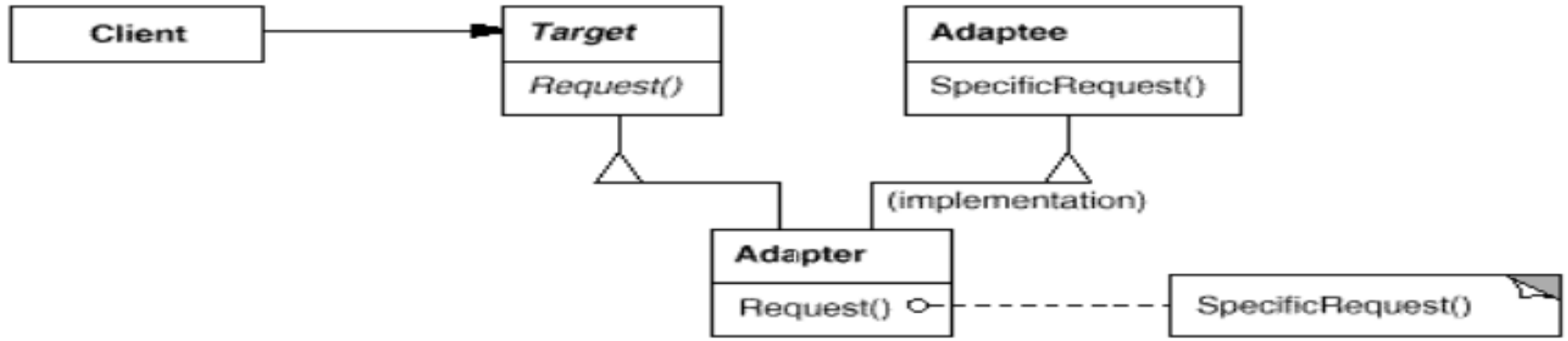

DrawingEditor
Shape
BoundingBox()
CreateManipulator()
TextView
GetExtent()
Line
BoundingBox()
CreateManipulator()
TextShape
BoundingBox()
CreateManipulator()
text
return text->GetExtent()
return new TextManipulator

Aplicación

Usar el patrón Adapter cuando:

- Se desea utilizar una clase existente y su interfaz no coincide con la que se necesita.
- Se desea crear una clase reutilizable que coopere con clases no relacionadas, es decir, clases que no tienen necesariamente interfaces compatibles.

Estructura y Participantes



- Target (Shape)
  define la interfaz específica del dominio que Client utiliza.
- Client (DrawingEditor)
  colabora con objetos que se ajustan a la interfaz de destino.
- Adaptee (TextView)
  define una interfaz existente que necesita adaptación.
- Adapter (TextShape)
  adapta la interfaz de Adaptee a la interfaz de destino.

Colaboraciones y Consecuencias

Client llama a las operaciones en una instancia del adapter. A su vez, el adapter llama a las operaciones de Adaptee que lleva a cabo la solicitud.

Cuestiones a considerar cuando se utiliza el patrón Adapter:
1.¿Cuanta adaptación hace el Adapter? Adapter varía en la cantidad de trabajo que hace para adaptar Adaptee a la interfaz Target. Hay un espectro de trabajo posible, desde una simple conversión (por ejemplo, cambiando los nombres de las operaciones) hasta soportando un conjunto de operaciones enteramente diferentes. La cantidad de trabajo que Adapter hace depende de cuanto de similar tienen la interfaz Target con Adaptee.
2.Adaptadores Pluggables Una clase es más reusable cuando se minimiza la suposición de que otras clases deben utilizarla. Mediante la construcción en una clase de la adaptación de una interfaz, se elimina la suposición de que otras clases ven la misma interfaz. Dicho de otra manera, la adaptación de la interfaz nos permite incorporar a nuestra clase en sistemas existentes que pueden esperar diferentes interfaces de la misma.

Implementación

Aunque la implementación de Adapter es generalmente sencilla, aquí hay algunos cuestiones a tener en cuenta:

- Implementación de adapter de clase en C ++. En una implementación C ++ de una clase adapter, el adapter heredaría públicamente de Target y privado de Adaptee. Así, el Adapter sería un subtipo de Target, pero no de Adaptee.

Ejemplos de Código

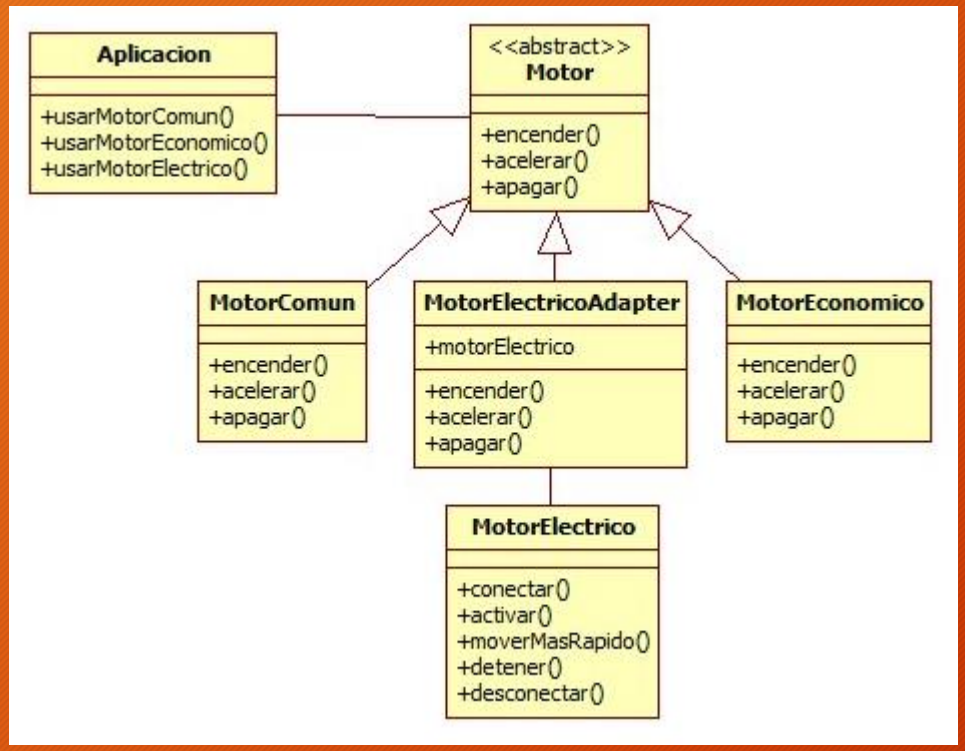
```
public class Client
{
    private ITarget _target;

    public Client(ITarget target)
    {
        _target = target;
    }

    public void MakeRequest()
    {
        _target.MethodA();
    }
}
```

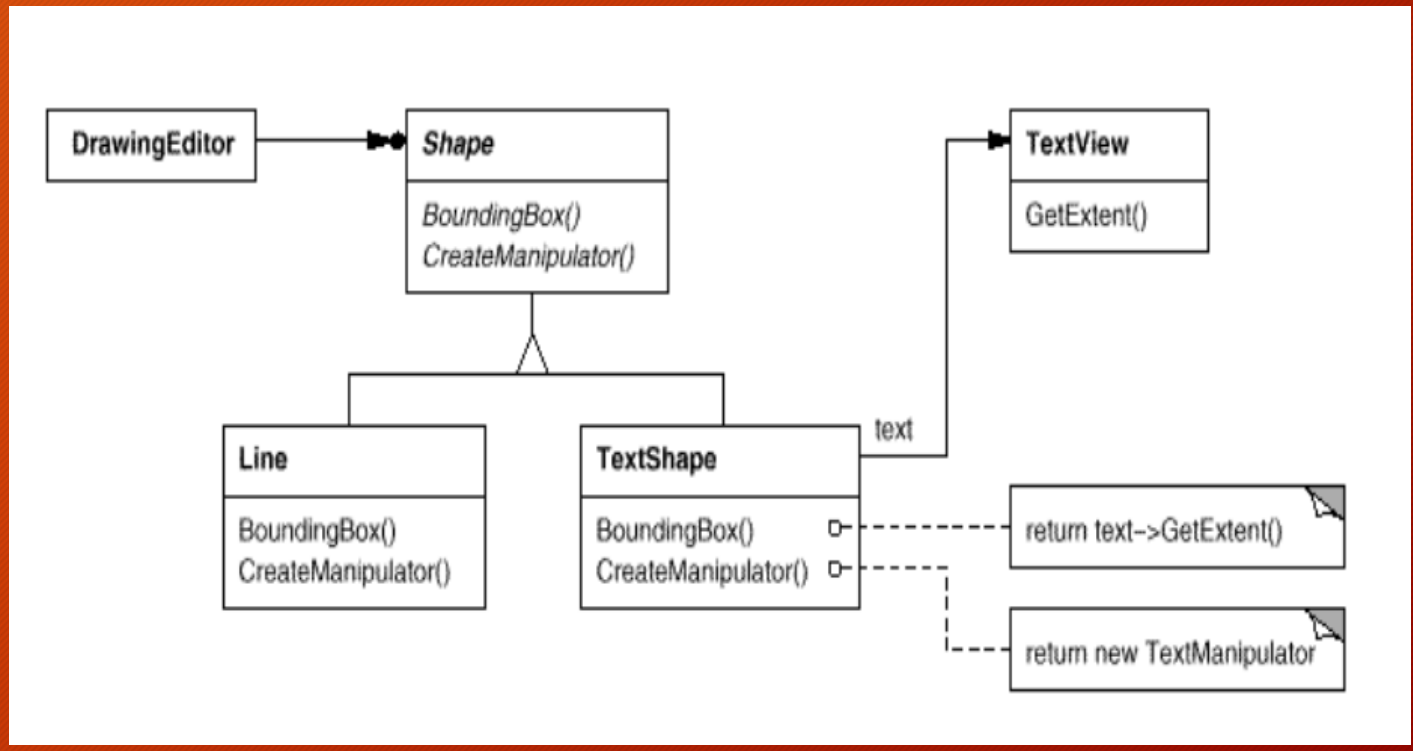
```
public interface ITarget
{
    void MethodA();
}


public class Adaptee
{
    public void MethodB()
    {
        Console.WriteLine("MethodB called");
    }
}
```

Ejemplos de Código

```
public class Adapter : ITarget
{
    Adaptee _adaptee = new Adaptee();

    public void MethodA()
    {
        _adaptee.MethodB();
    }
}
```

Usos conocidos





ET++Draw

Patrones relacionados

Bridge: tiene una estructura similar a un adapter, pero Bridge tiene una diferente intención: Este pretende separar una interfaz de su implementación para que pueda ser variada fácilmente e independientemente. Un adapter está destinado a cambiar la interfaz.

Decorator: mejora otro objeto sin cambiar su interfaz. Un decorator es por lo tanto más transparente a la aplicación que un adapter.

Proxy: Define un representante o sustituto de otro objeto y no cambia su interfaz.