**Propósito**

**Visitor** es un patrón de diseño de comportamiento que te permite separar algoritmos de los objetos sobre los que operan.

**Problema**

Imagina que tu equipo desarrolla una aplicación que funciona con información geográfica estructurada como un enorme grafo. Cada nodo del grafo puede representar una entidad compleja, como una ciudad, pero también cosas más específicas, como industrias, áreas turísticas, etc. Los nodos están conectados con otros si hay un camino entre los objetos reales que representan. Técnicamente, cada tipo de nodo está representado por su propia clase, mientras que cada nodo específico es un objeto.

En cierto momento, te surge la tarea de implementar la exportación del grafo a formato XML. Al principio, el trabajo parece bastante sencillo. Planificaste añadir un método de exportación a cada clase de nodo y después aprovechar la recursión para recorrer cada nodo del grafo, ejecutando el método de exportación. La solución era sencilla y elegante: gracias al polimorfismo, no acoplabas el código que invocaba el método de exportación a clases concretas de nodos.

Lamentablemente, el arquitecto del sistema no te permitió alterar las clases de nodo existentes. Dijo que el código ya estaba en producción y no quería arriesgarse a que se descompusiera por culpa de un potencial error en tus cambios.

Además, cuestionó si tenía sentido tener el código de exportación XML dentro de las clases de nodo. El trabajo principal de estas clases era trabajar con geodatos. El comportamiento de la exportación XML resultaría extraño ahí.

Había otra razón para el rechazo. Era muy probable que, una vez que se implementara esta función, alguien del departamento de marketing te pidiera que incluyeras la capacidad de exportar a otros formatos, o te pidiera alguna otra cosa extraña. Esto te forzaría a cambiar de nuevo esas preciadas y frágiles clases.

**Solución**

El patrón Visitor sugiere que coloques el nuevo comportamiento en una clase separada llamada *visitante*, en lugar de intentar integrarlo dentro de clases existentes. El objeto que originalmente tenía que realizar el comportamiento se pasa ahora a uno de los métodos del visitante como argumento, de modo que el método accede a toda la información necesaria contenida dentro del objeto.

Ahora, ¿qué pasa si ese comportamiento puede ejecutarse sobre objetos de clases diferentes? Por ejemplo, en nuestro caso con la exportación XML, la implementación real probablemente sería un poco diferente en varias clases de nodo. Por lo tanto, la clase visitante puede definir un grupo de métodos en lugar de uno solo, y cada uno de ellos podría tomar argumentos de distintos tipos, así:

**class** **ExportVisitor** **implements** Visitor **is**

**method** doForCity(City c) { ... }

**method** doForIndustry(Industry f) { ... }

**method** doForSightSeeing(SightSeeing ss) { ... }

// ...

Pero, ¿cómo llamaríamos exactamente a estos métodos, sobre todo al lidiar con el grafo completo? Estos métodos tienen distintas firmas, por lo que no podemos utilizar el polimorfismo. Para elegir un método visitante adecuado que sea capaz de procesar un objeto dado, debemos revisar su clase. ¿No suena esto como una pesadilla?

Puede que te preguntes, ¿por qué no utilizar la sobrecarga de métodos? Eso es cuando le das a todos los métodos el mismo nombre, incluso cuando soportan distintos grupos de parámetros. Lamentablemente, incluso asumiendo que nuestro lenguaje de programación la soportara (como Java y C#), no nos ayudaría. Debido a que la clase exacta de un objeto tipo nodo es desconocida de antemano, el mecanismo de sobrecarga no será capaz de determinar el método correcto a ejecutar. Recurrirá por defecto al método que toma un objeto de la clase base Nodo.

Sin embargo, el patrón Visitor ataja este problema. Utiliza una técnica llamada **[Double Dispatch](https://refactoring.guru/es/design-patterns/visitor-double-dispatch)**, que ayuda a ejecutar el método adecuado sobre un objeto sin complicados condicionales. En lugar de permitir al cliente seleccionar una versión adecuada del método a llamar, ¿qué tal si delegamos esta opción a los objetos que pasamos al visitante como argumento? Como estos objetos conocen sus propias clases, podrán elegir un método adecuado en el visitante más fácilmente. “Aceptan” un visitante y le dicen qué método visitante debe ejecutarse.

Lo confieso. Hemos tenido que cambiar las clases de nodo, después de todo. Pero al menos el cambio es trivial y nos permite añadir más comportamientos sin alterar el código otra vez.

Ahora, si extraemos una interfaz común para todos los visitantes, todos los nodos existentes pueden funcionar con cualquier visitante que introduzcas en la aplicación. Si te encuentras introduciendo un nuevo comportamiento relacionado con los nodos, todo lo que tienes que hacer es implementar una nueva clase visitante.

**Aplicabilidad**

 Utiliza el patrón Visitor cuando necesites realizar una operación sobre todos los elementos de una compleja estructura de objetos (por ejemplo, un árbol de objetos).

 El patrón Visitor te permite ejecutar una operación sobre un grupo de objetos con diferentes clases, haciendo que un objeto visitante implemente distintas variantes de la misma operación que correspondan a todas las clases objetivo.

 Utiliza el patrón Visitor para limpiar la lógica de negocio de comportamientos auxiliares.

 El patrón te permite hacer que las clases primarias de tu aplicación estén más centradas en sus trabajos principales extrayendo el resto de los comportamientos y poniéndolos dentro de un grupo de clases visitantes.

 Utiliza el patrón cuando un comportamiento solo tenga sentido en algunas clases de una jerarquía de clases, pero no en otras.

 Puedes extraer este comportamiento y ponerlo en una clase visitante separada e implementar únicamente aquellos métodos visitantes que acepten objetos de clases relevantes, dejando el resto vacíos.

**Pros**

* *Principio de abierto/cerrado*. Puedes introducir un nuevo comportamiento que puede funcionar con objetos de clases diferentes sin cambiar esas clases.
* *Principio de responsabilidad única*. Puedes tomar varias versiones del mismo comportamiento y ponerlas en la misma clase.
* Un objeto visitante puede acumular cierta información útil mientras trabaja con varios objetos. Esto puede resultar útil cuando quieras atravesar una compleja estructura de objetos, como un árbol de objetos, y aplicar el visitante a cada objeto de esa estructura.

## Contras

* Debes actualizar todos los visitantes cada vez que una clase se añada o elimine de la jerarquía de elementos.
* Los visitantes pueden carecer del acceso necesario a los campos y métodos privados de los elementos con los que se supone que deben trabajar.