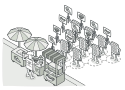
Hey, Have you noticed the great book prices, curt, dolphin? Get up-to-date programming skills for the price of a book! Learn more

Visitor

Intent

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.



Problem

Imagine that your team develops an app which works with geographic information structured in the form of one huge graph. Each node of the graph may represent a complex entity such as a city, but also lower-level details such as industries, roads, and bridges. Initially, the app's workflow was pretty straightforward. It could read a set of data from files and then traverse the graph to execute the needed operations. Different kinds of nodes were represented by its own class, while each specific node is an object.

Importing the graph into XML

At some point, you get a task to implement exporting the graph into XML format. At first, this job seemed pretty straightforward. You planned to add an export method to each node class and then traverse the entire graph to run your export method for every object. The solution was simple and elegant. Thanks to polymorphism, you weren't scaping the code with a hack that depends on concrete classes of nodes.

Unfortunately, the system architect refused to allow you to alter existing node classes. He said that the code was already in production and he didn't want to risk breaking it because of a potential bug in your changes.

The XML export feature is a hard-to-implement cross-cutting concern that affects every application in the class diagram through various unrelated classes.

Besides, he is pessimistic about the chance it makes sense to have the XML export code within the node classes. The primary job of these classes is to work with geodata. The XML export behavior would look alien there.

There was another reason for the refusal. Chances highly likely that after this feature (never implemented) somebody from the marketing department would ask you to provide the ability to export into a different format, or integrate some other partial stuff. This would force you to change these previous working classes again.

Solution

The Visitor pattern suggests that you place the new behavior into a separate class called visitor, instead of trying to integrate it into existing classes. The original object that used to perform the behavior is now passed to one of the visitor's methods as an argument, providing the method access to all necessary data contained within the object.

Now, what if that behavior can be implemented over objects of different classes? For example, in our case with GIS, imagine the actual City/Industry/Road classes probably do not differ too much. They most likely have various node classes. Thus, the visitor class may define not one, but a set of methods, each of which could take an argument of different type, like this:

```
class ExportVisitor implements Visitor {
    method doForCity(City c) {
        // ...export city data...
    }
    method doForIndustry(Industry f) {
        // ...export factory data...
    }
    method doForCoordinates(Coordinates coord) {
        // ...export coordinates...
    }
}
```

But how exactly would we call these methods, especially when dealing with the whole graph? These methods have different signatures, so we can't use polymorphism. To pick a proper visitor method that's able to process a given object, we'd need to check its class. Doesn't this sound like a nightmare?

```
foreach (node node in graph)
    if (node instanceof City)
        exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
        exportVisitor.doForIndustry((Industry) node)
    // ...
```

You might ask, why don't we use method overloading? That's when you give all methods the same name, even if they support different sets of parameters (obviously, even assuming that our programming language supports it at all. Java can't it at all). It wasn't help us. Since the exact type of a node object is unknown in advance, the overloading mechanism won't be able to determine the correct method to execute. It's default is the method that takes an object of the base GeoObject class.

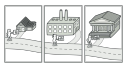
However, the Visitor pattern addresses this problem. It uses a technique called Double Dispatch, which helps to execute the proper method on an object without relying on the object's type. Instead of letting the client code select the proper method, the objects themselves tell a visitor object what method to apply to themselves and the visitor uses argument types to know the exact concrete class they belong to. This way, the proper method can be chosen statically. They "accept" a visitor and tell it what visiting method should be executed.

```
// Client code
foreach (node node in graph)
    node.accept(exportVisitor)

// City.
class City {
    method accept(Visitor v) {
        v.doForCity(this)
    }
    // ...other methods...
}

// Industry.
class Industry {
    method accept(Visitor v) {
        v.doForIndustry(this)
    }
    // ...
}
```

I confess. We had to change the node classes after all. But at least the change is trivial and it lets us add further behaviors without altering the node once again.

Now, if we extract a common interface for all visitors, all existing nodes can work with any visitor you introduce into the app. If you find yourself introducing a new behavior dedicated to nodes, all you have to do is implement a new visitor class.

Real-World Analogy

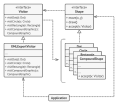


A great insurance agent is always ready to offer either auto policies or various types of organizations.

Imagine a seasoned insurance agent who's eager to get new customers. He can visit every building in a neighborhood, trying to sell insurance to everyone he meets. Depending on the type of organization then occupies the building, he can offer specialized insurance policies:

- If it's a residential building, he sells medical insurance.
- If it's a bank, he sells theft insurance.
- If it's a coffee shop, he sells fire and flood insurance.

Structure



Implementing various types of elements into the Visitor allows a client object

1. The **Visitor** interface declares a set of visiting methods that can take concrete elements of an object structure as arguments. These methods may have the same name if the program is written in a language that supports overloading, but the type of their parameters must be different.
2. Each **Concrete Visitor** implements several versions of the same behaviors, tailored for different concrete element classes.
3. The **Element** interface declares a method for "accepting" visitors. This method should have one parameter declared with the type of the visitor interface.
4. Each **Concrete Element** must implement the acceptance method. The purpose of this method is to redirect the call to the proper visitor's method corresponding to the current element class. Be aware that even if a base element class implements this method, all subclasses must still override this method in their own classes and call the appropriate method on the visitor object.
5. The **Client** usually represents a collection or some other complex object (for example, a Composite tree). Usually, clients aren't aware of all the concrete element classes because they work with objects from that collection via some abstract interface.

Pseudocode

In this example, the **Visitor** pattern adds XML export support to the class hierarchy of geometric shapes.

Importing various types of shapes into XML format via a visitor object

```
// The element interface declares an `accept` method that takes
// the base visitor interface as a argument.
interface Shape {
    method move(x, y)
    method draw()
    method accept(v: Visitor)
}

// Each concrete element class must implement the `accept`
// method in such a way that it calls the visitor's method that
// corresponds to the element's class.
class Dot implements Shape {
    // ...

    // Note that we're calling `visitDot`, which matches the
    // current dot class. This way we let the visitor know the
    // class of the element it deals with.
    method accept(v: Visitor)
        v.doDot(this)
    // ...other methods specific to this class...
}

class Circle implements Shape {
    // ...
    method accept(v: Visitor)
        v.doCircle(this)
    // ...other methods specific to this class...
}

class Rectangle implements Shape {
    // ...
    method accept(v: Visitor)
        v.doRectangle(this)
    // ...other methods specific to this class...
}

class CompoundShape implements Shape {
    // ...
    method accept(v: Visitor)
        v.doCompound(this)
    // ...other methods specific to this class...
}

// The Visitor interface declares a set of visiting methods that
// correspond to element classes. The difference between visiting
// a simple dot and a complex shape(s) may exist since of the
// complexity of the visiting data.
interface Visitor {
    method doDot(d: Dot)
    method doCircle(c: Circle)
    method doRectangle(r: Rectangle)
    method doCompound(cs: CompoundShape)
}

// So we implement the biggest benefit of the Visitor pattern
// even using if and a complex object structure such as a
// compound. In this case, it might be helpful to draw,
// for example, shapes in a specific order when we know that
// the compound contains several objects of the structure.
// Concrete visitor: export into XML format.
class XMLExportVisitor implements Visitor {
    // Export IDs, and a set of visited coordinates.
    method doDot(d: Dot) {
        // ...export the dot's id and center coordinates.
    }

    method doCircle(c: Circle) {
        // ...export the circle's id, center coordinates and
        // radius.
    }

    method doRectangle(r: Rectangle) {
        // ...export the rectangle's id, top left coordinates,
        // width and height.
    }

    method doCompound(cs: CompoundShape) {
        // ...export the shape's id as well as the list of its
        // children.
    }

    // The export code may use some accumulated data any out of
    // standard without forgoing our class structure changes. The
    // visitor can keep state for all its visits. The implementation depends
    // on the calling algorithm.
    // ...other helpers...
    // ...field exportedObjects: array of Shape

    method export(...): ID
        exportedObjects = new ObjectArray(maxSize)

        foreach (shape s: allShapes) do
            shape.accept(exportVisitor)

    // ...
}
```

If you wonder why you need the accept() method in this example, my article Visitor and Double Dispatch will enhance this question in detail.

Applicability

Use the Visitor when you need to perform an operation on all elements of a complex object structure (for example, an object tree).

The Visitor pattern lets you execute an operation over a set of objects with different classes by having a visitor object implement several variants of the same operation, which correspond to all target classes.

Use the Visitor to clean up the business logic of auxiliary behaviors.

The pattern lets you make this primary classes of your app more focused on their main jobs by extracting all other behaviors into a set of visitor classes.

Use this pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.

You can extract this behavior into a separate visitor class and implement only those visiting methods that accept objects of relevant classes, leaving the rest empty.

How to Implement

1. Declare the visitor interface with a set of "visiting" methods, one per each concrete element class that exists in the program.
2. Declare the element interface. If you're working with an existing element class hierarchy, add the abstract "acceptance" method to the base class of the hierarchy. This method should accept a visitor object as an argument.
3. Implement the acceptance methods. In all concrete element classes. These methods must simply redirect the call to a visiting method on the incoming visitor object which matches the class of the current element.
4. The element classes should only work with visitors via the visitor interface. However, treasure may be aware of all concrete element classes, referenced as parameter types of the visiting methods.
5. For each behavior that can't be implemented inside the element hierarchy, create a new concrete visitor class and implement all of the visiting methods.

   You might encounter a situation where the visitor will need access to some private members of the element class. In this case, you can either make these fields or methods public, breaking the element's encapsulation, or nest the visitor class in the element's one. The latter is only possible if you're lucky to work with a programming language that supports nested classes.
6. The client must create visitor objects and pass them into elements via "acceptance" methods.

Pros and Cons

- Open/Closed Principle. You can introduce a new behavior that can work with objects of different class without changing these classes.
- Single Responsibility Principle. You can move multiple versions of the same behavior into the same class.
- A visitor object can accumulate some useful information while working with various objects. This might be handy when you want to traverse some complex object structure, such as an object tree, and apply the visitor to each object of this structure.
- You need to update all visitors each time a class gets added to or removed from the element hierarchy.
- Visitors might lack the necessary access to the private fields and methods of the elements they're supposed to work with.

Relations with Other Patterns

- You can treat Visitor as a powerful version of the Command pattern. Its objects can execute operations over various objects of different classes.
- You can use Visitor to execute an operation over an entire Composite tree.
- You can use Visitor along with Iterator to traverse a complex data structure and execute some operation over its elements, even if they all have different classes.

Code Examples

Extra Content

- Russian only, we can't simply replace the Visitor pattern with method overloading? Read my article Visitor and Double Dispatch to learn about the nasty details.

Support our free website and own the eBook!

- 6 in 1 design patterns and 6 principles explained in depth
- 300+ illustrations
- 1000+ pages (PDF, EPUB, Kindle, Mobi, ready for Kindle Oasis)
- 1.25 years worth high-quality resources and designs
- Lifetime updates, 30-day money-back guarantee
- 24 exclusive appendices with design patterns in practice

Learn more

SECTIONS

Decorator Pattern

READ NEXT

Visitor and Double Dispatch

© 2014-2024 Refactoring.Guru. All rights reserved.
Refactoring.Guru, SOLID.Email

Terms & Conditions
Privacy Policy
Cookie & Usage Policy

Kreditkarten Oskar OÜ, 11 Kesklinna (Kesklinn), Harjumaa, 10145
Support: support@refactoring.guru