What are the actual advantages of the visitor pattern? What are the alternatives?

I read quite a lot about the visitor pattern and its supposed advantages. To me however it seems they are not that much advantages when applied in practice:

- "Convenient" and "elegant" seems to mean lots and lots of boilerplate code
- Therefore, the code is hard to follow. Also 'accept'/'visit' is not very descriptive
- Even uglier boilerplate code if your programming language has no method overloading (i.e. Vala)
- You cannot in general add new operations to an existing type hierarchy without modification of all classes, since you need new 'accept'/'visit' methods *everywhere* as soon as you need an operation with different parameters and/or return value (changes to classes all over the place is one thing this design pattern was supposed to avoid!?)
- Adding a new type to the type hierarchy requires changes to *all* visitors. Also, your visitors cannot simply ignore a type - you need to create an empty visit method (boilerplate again)

It all just seems to be an awful lot of work when all you want to do is actually:

```
// Pseudocode
int SomeOperation(ISomeAbstractThing obj) {
    switch (type of obj) {
        case Foo: // do Foo-specific stuff here
        case Bar: // do Bar-specific stuff here
        case Baz: // do Baz-specific stuff here
        default: return 0; // do some sensible default if type unknown or if we
don't care
    }
}
```

The only real advantage I see (which btw i haven't seen mentioned anywhere): The visitor pattern is probably the fastest method to implement the above code snippet in terms of cpu time (if you don't have a language with double dispatch or efficient type comparison in the fashion of the pseudocode above).

Questions:

- So, what advantages of the visitor pattern have I missed?

- What alternative concepts/data structures could be used to make the above fictional code sample run equally fast?

language-agnostic visitor-pattern

asked Nov 26 '13 at 13:59

Askaga
2,168 4 14 31

IMO you missed an important point...you should **NEVER** (I would write it even bigger) have something like `switch (type of obj)` . No matters if visitor or not. If your code is not like that then you won't write any boilerplate code. – Adriano Repetti Nov 26 '13 at 14:02

Examples of when a visitor may help you? Imagine to write an utility to search text inside files. Search engine will visit each file system item (files will be visited and directories will propagate visit to each child item). File system items can be directories, files, links to FTP sites...search engine will never know with what it's working. Visitor shouldn't know exact type (not because of visitor pattern but because OOP principles...) – Adriano Repetti Nov 26 '13 at 14:07

Visitors are a way too low level. No point in using visitors when you can do something like this: cs.indiana.edu/~dyb/pubs/nano-jfp.pdf – SK-logic Nov 26 '13 at 14:13

@Adriano: You say visitor shouldn't know the exact type, yet when any visit method of the visitor is called then the passed parameter IS an exact type (IMO thats the point of it). Also if you look at the Wikipedia article on visitor pattern you will see that the Scala example works exactly as stated in my pseudocode. – Askaga Nov 26 '13 at 14:14

@BillAskaga it's ONE example and ONE use case (moreover I would overload methods instead of that, again it's about OOP not about visitor pattern). The point of patterns is always...you do not have to use them ALWAYS. When they make your code more complicated then...don't use them. But, again, IMO what makes that code less readable isn't visitor but basic logic around it. – Adriano Repetti Nov 26 '13 at 14:22

4 Answers

By experience, I would say that "Adding a new type to the type hierarchy requires changes to all visitors" is an advantage. Because it definitely forces you to consider the new type added in ALL places where you did some type-specific stuff. It prevents you from forgetting one....

answered Jul 27 '14 at 17:59

jpo38
12.6k 2 29 80

That isn't always required and secondly this goes completely against the basic the Open/Closed fundamentals. – Ron Deijkers Jan 30 '15 at 16:08

In my personal opinion, the visitor pattern is only useful if the interface you want implemented is rather static and doesn't change a lot, while you want to give anyone a chance to implement their own functionality.

Note that you can avoid changing everything every time you add a new method by creating a new interface instead of modifying the old one - then you just have to have some logic handling the case when the visitor doesn't implement all the interfaces.

Basically, the benefit is that it allows you to choose the correct method to call at runtime, rather than at compile time - and the available methods are actually extensible.

For more info, have a look at this article - http://rgomes-info.blogspot.co.uk/2013/01/a-better-implementation-of-visitor.html

answered Nov 26 '13 at 14:10

Luaan
46.8k 4 53 77

For as far as I have seen so far there are two uses / benefits for the visitor design pattern:

1. Double dispatch
2. Separate data structures from the operations on them

Double dispatch

Let's say you have a Vehicle class and a VehicleWasher class. The VehicleWasher has a Wash(Vehicle) method:

```
VehicleWasher
    Wash(Vehicle)

Vehicle
```

Additionally we also have specific vehicles like a car and in the future we'll also have other specific vehicles. For this we have a Car class but also a specific CarWasher class that has an operation specific to washing cars (pseudo code):

```
CarWasher : VehicleWasher
    Wash(Car)

Car : Vehicle
```

Then consider the following client code to wash a specific vehicle (notice that x and washer are declared using their base type because the instances might be dynamically created based on user input or external configuration values; in this example they are simply created with a new operator though):

```
Vehicle x = new Car();
VehicleWasher washer = new CarWasher();

washer.Wash(x);
```

Many languages use single dispatch to call the appropriate function. Single dispatch means that during runtime only a single value is taken into account when determining which method to call. Therefore only the actual type of washer we'll be considered. The actual type of x isn't taken into account. The last line of code will therefore invoke CarWasher.Wash(Vehicle) and *NOT* CarWasher.Wash(Car).

If you use a language that does not support multiple dispatch and you do need it (I can honoustly say I have never encountered such a situation though) then you can use the visitor design pattern to enable this. For this two things need to be done. First of all add an Accept method to the Vehicle class (the visitee) that accepts a VehicleWasher as a visitor and then call its operation Wash:

```
Accept(VehicleWasher washer)
    washer.Wash(this);
```

The second thing is to modify the calling code and replace the washer.Wash(x); line with the following:

```
x.Accept(washer);
```

Now for the call to the Accept method the actual type of x is considered (and only that of x since we are assuming to be using a single dispatch language). In the implementation of the Accept method the Wash

method is called on the washer object (the visitor). For this the actual type of the washer is considered and this will invoke CarWasher.Wash(Car). By combining two single dispatches a double dispatch is implemented.

Now to eleborate on your remark of the terms like Accept and Visit and Visitor being very unspecific. That is absolutely true. But it is for a reason.

Consider the requirement in this example to implement a new class that is able to repair vehicles: a VehicleRepairer. This class can only be used as a visitor in this example if it would inherit from VehicleWasher and have its repair logic inside a Wash method. But that ofcourse doesn't make any sense and would be confusing. So I totally agree that design patterns tend to have very vague and unspecific naming but it does make them applicable to many situations. The more specific your naming is, the more restrictive it can be.

Your switch statement only considers one type which is actually a manual way of single dispatch. Applying the visitor design pattern in the above way will provide double dispatch. This way you do not necessarily need additional Visit methods when adding additional types to your hierarchy. Ofcourse it does add some complexity as it makes the code less readable. But ofcourse all patterns come at a price.

Ofcourse this pattern cannot always be used. If you expect lots of complex operations with multiple parameters then this will not be a good option.

An alternative is to use a language that does support multiple dispatch. For instance .NET did not support it until version 4.0 which introduced the dynamic keyword. Then in C# you can do the following:

```
washer.Wash((dynamic)x);
```

Because x is then converted to a dynamic type its actual type will be considered for the dispatch and so both x and washer will be used to select the correct method so that CarWasher.Wash(Car) will be called (making the code work correctly and staying intuitive).

Separate data structures and operations

The other benefit and requirement is that it can separate the data structures from the operations. This can be an advantage because it allows new visitors to be added that have there own operations while it also allows data structures to be added that 'inherit' these operations. It can however be only applied if this seperation can be done / makes sense. The classes that perform the operations (the visitors) do not know the structure of the data structures nor do they have to know that which makes code more maintainable and reusable. When applied for this reason the visitors have operations for the different elements in the data structures.

Say you have different data structures and they all consist of elements of class Item. The structures can be lists, stacks, trees, queues etc.

You can then implement visitors that in this case will have the following method:

```
Visit(Item)
```

The data structures need to accept visitors and then call the Visit method for each Item.

This way you can implement all kinds of visitors and you can still add new data structures as long as they consist of elements of type Item.

For more specific data structures with additional elements (e.g. a Node) you might consider a specific visitor (NodeVisitor) that inherits from your conventional Visitor and have your new data structures accept that visitor (Accept(NodeVisitor)). The new visitors can be used for the new data structures but also for the old data structures due to inheritence and so you do not need to modify your existing 'interface' (the super class in this case).

answered Jan 30 '15 at 15:58

Ron Deijkers

1,056 1 14 25

This is an old question but i would like to answer.

The visitor pattern is useful mostly when you have a composite pattern in place in which you build a tree of objects and such tree arrangement is unpredictable.

Type checking may be one thing that a visitor can do, but say you want to build an expression based on a tree that can vary its form according to a user input or something like that, a visitor would be an effective way for you to validate the tree, or build a complex object according to the items found on the tree.

The visitor may also carry an object that does something on each node it may find on that tree. this visitor may be a composite itself chaining lots of operations on each node, or it can carry a mediator object to mediate operations or dispatch events on each node.

You imagination is the limit of all this. you can filter a collection, build an abstract syntax tree out of an complete tree, parse a string, validate a collection of things, etc.

edited Mar 22 '17 at 8:38 answered Jun 4 '15 at 3:42

prabhakaran S
566 4 12

jhonatan teixeira
142 5