How does the visitor pattern not violate the Open Close Priniciple?

Ask Question

> ### From Wikipedia :
>
> The idea was that once completed, the implementation of a class could only be modified to correct errors; new or changed features would require that a different class be created. That class could reuse coding from the original class through inheritance

From what I understand, The Visitor pattern is a powerful technique to traverse similar but different objects that implement the same Interface by the use of double dispatch. In one of my Java examples, I created a composite set of objects that form a tree structure, and each specific implementation of those objects implement the visitable interface. The visitor interface has a method for each of the visitable objects, and the concrete visitor implements what to do for each of those cases.

The thing I'm trying to get my head around is the fact that if I were to add a new implementation to the composite structure that also implements visitable, then I need to reopen the visitor interface and add that case to it, also forcing me to modify each implementation of visitor.

While this is fine, since I would need to do this anyway (What good is adding to your visitables if the visitor can't understand them?) but on an academic level, wouldn't this be violating the Open Closed Principle? Isn't that one of the core reasons for Design Patterns anyway? Trying to show a decent reason for switching to this pattern instead of maintaining a switch statement to end all switch statements, but everyone argues that the code will be the same anyways, with a method for each case instead of a switch block, just broken up and harder to read.

java design-patterns visitor open-closed-principle

edited Feb 13 '16 at 23:59
Dave Schweisguth
22.7k 7 60 89

asked Dec 20 '12 at 19:29
C0M37
354 3 12

2 Answers

A pattern is applicable to certain cases. From the GoF book (p. 333):

> Use the Visitor pattern when
>
> - [...]
> - the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.

If you frequently change the classes of the objects that make up the structure, the Visitor class hierarchy can be hard to maintain. In such a case, it may be easier to define operations on the classes that make up the structure.

edited Dec 20 '12 at 19:56

answered Dec 20 '12 at 19:38
reprogrammer
7,600 12 47 85

So in the event that I have an object structure that changes often, but only really one visiting strategy, then it does not make sense to continue pursuing a visitor pattern. Do you know of any other elegant solutions to this? Should each object in the structure contain it's own logic that would normally be in the visitor? – C0M37 Dec 20 '12 at 19:42

Setting this as the accepted answer, specifically since it addresses the open close principle question, but an straight up honorable mention to dasblinkenlight's answer in this thread for a possible alternative implementation. Thanks guys! – C0M37 Dec 20 '12 at 19:55

See my updated answer. – reprogrammer Dec 20 '12 at 19:56

I also would mention Acyclic Visitor – Fuhrmanator Feb 12 '14 at 14:12

John Vlissides, one of the GoF, wrote an excellent chapter on the subject in his *Patterns Hatching* book. He discusses the very concern that extending the hierarchy is incompatible with maintaining the visitor intact. His solution is a hybrid between a visitor and an `enum` -based (or a type-based) approach, where a visitor is provided with a `visitOther` method called by all classes outside the "base" hierarchy that the visitor understands out of the box. This method provides you an escape way to treat the classes added to the hierarchy after the visitor has been finalized.

```java
abstract class Visitable {
    void accept(Visitor v);
}
class VisitableSubclassA extends Visitable  {
    void accept(Visitor v) {
        v.visitA(this);
    }
}
class VisitableSubclassB extends Visitable {
    void accept(Visitor v) {
        v.visitB(this);
    }
}
interface Visitor {
    // The "boilerplate" visitor
    void visitB(VisitableSubclassA a);
    void visitB(VisitableSubclassB b);
    // The "escape clause" for all other types
    void visitOther(Visitable other);
}
```

When you add this modification, your visitor is no longer in violation of the *Open-Close Principle*, because it is open to extension without the need to modify its source code.

I tried this hybrid method on several projects, and it worked reasonably fine. My main class hierarchy is defined in a separately compiled library which does not need to change. When I add new implementations of `Visitable` , I modify my `Visitor` implementations to expect these new classes in their `visitOther` methods. Since both the visitors and the extending classes are located in the same library, this approach works very well.

P.S. There is another article called *Visitor Revisited* discussing precisely that question. The author concludes that one can go back to an `enum` -based double dispatch, because the original *Visitor Pattern* does not present a significant improvement over the `enum` -based dispatch. I disagree with the author, because in cases when the bulk of your inheritance hierarchy is solid, and the users are expected to provide a few implementations here and there, a hybrid approach provides significant benefits in readability; there is no point in throwing out everything because of a couple of classes that we can fit into the hierarchy with relative ease.

edited Dec 20 '12 at 20:50

answered Dec 20 '12 at 19:44

dasblinkenlight

588k 54 721 1144

I had another idea: Would it make sense at all to implement some sort of composition of visitors that handle different specific visitable types, and simply call the correct visitor in a chain of command style? I suppose at that point I'm just trying to fit a square peg in a round hole, though. – C0M37 Dec 20 '12 at 19:48

Also, thank you for that code example! – C0M37 Dec 20 '12 at 19:49

1 @C0M37 I tried doing something like that (i.e. two levels of visitors) but it quickly got out of hand, so I dropped my experiment, and went back to the "visitor with an escape clause". – dasblinkenlight Dec 20 '12 at 19:52

1 Ever heard of the acyclic visitor? From c2.com: *"The Acyclic Visitor pattern allows new functions to be added to existing class hierarchies without affecting those hierarchies, and without creating the dependency cycles that are inherent to the GangOfFour VisitorPattern."* – Jordão Dec 22 '12 at 13:02

1 @Jordão Nice! I do not quite like the idea of having to cast when jumping through levels of visitors, but the idea looks very cool. – dasblinkenlight Dec 22 '12 at 13:07