When should I use the Visitor Design Pattern?

Ask Question

I keep seeing references to the visitor pattern in blogs but I've got to admit, I just don't get it. I read the wikipedia article for the pattern and I understand its mechanics but I'm still confused as to when I'd use it.

As someone who just recently really **got** the decorator pattern and is now seeing uses for it absolutely everywhere I'd like to be able to really understand intuitively this seemingly handy pattern as well.

design-patterns visitor-pattern

edited Feb 19 '16 at 4:24

Ravindra babu

26k 5 134 127

asked Oct 31 '08 at 23:04

George Mauer

45.9k 99 295 514

6 Finally got it after reading this article by Jermey Miller on my blackberry while stuck waiting in a lobby for two hours. It's long but gives a wonderful explanation of double-dispatch, visitor, and composite, and what you can do with these. – George Mauer Jan 20 '09 at 15:03

1 here is a nice article: codeproject.com/Articles/186185/Visitor-Design-Pattern – Seyed Morteza Mousavi Aug 8 '13 at 8:49

1 Visitor Pattern? Which one? The point is: there's a lot of misunderstanding and pure confusion around this design pattern. I've written and article which hopefully puts some order to this chaos: rgomes-info.blogspot.co.uk/2013/01/… – Richard Gomes Aug 8 '13 at 9:38

@George Mauer: I'd like to ask you to reconsider my answer. I've also fixed the broken link. I've written a long article about it, using real life examples. – Richard Gomes Sep 21 '13 at 12:08

20 Answers

I'm not very familiar with the Visitor pattern. Let's see if I got it right. Suppose you have a hierarchy of animals

```
class Animal {  };
class Dog: public Animal {  };
class Cat: public Animal {  };
```

(Suppose it is a complex hierarchy with a well-established interface.)

Now we want to add a new operation to the hierarchy, namely we want each animal to make its sound. As far as the hierarchy is this simple, you can do it with straight polymorphism:

```
class Animal
{ public: virtual void makeSound() = 0; };

class Dog : public Animal
{ public: void makeSound(); };

void Dog::makeSound()
{ std::cout << "woof!\n"; }

class Cat : public Animal
{ public: void makeSound(); };

void Cat::makeSound()
{ std::cout << "meow!\n"; }
```

But proceeding in this way, each time you want to add an operation you must modify the interface to every single class of the hierarchy. Now, suppose instead that you are satisfied with the original interface, and that you want to make the fewest possible modifications to it.

The Visitor pattern allows you to move each new operation in a suitable class, and you need to extend the hierarchy's interface only once. Let's do it. First, we define an abstract operation (the "Visitor" class in GoF) which has a method for every class in the hierarchy:

```
class Operation
{
public:
    virtual void hereIsADog(Dog *d) = 0;
    virtual void hereIsACat(Cat *c) = 0;
};
```

Then, we modify the hierarchy in order to accept new operations:

```
class Animal
{ public: virtual void letsDo(Operation *v) = 0; };

class Dog : public Animal
```

```
{ public: void letsDo(Operation *v); };

void Dog::letsDo(Operation *v)
{ v->hereIsADog(this); }

class Cat : public Animal
{ public: void letsDo(Operation *v); };

void Cat::letsDo(Operation *v)
{ v->hereIsACat(this); }
```

Finally, we implement the actual operation, *without modifying neither Cat nor Dog*:

```
class Sound : public Operation
{
public:
    void hereIsADog(Dog *d);
    void hereIsACat(Cat *c);
};

void Sound::hereIsADog(Dog *d)
{ std::cout << "woof!\n"; }

void Sound::hereIsACat(Cat *c)
{ std::cout << "meow!\n"; }
```

Now you have a way to add operations without modifying the hierarchy anymore. Here is how it works:

```
int main()
{
    Cat c;
    Sound theSound;
    c.letsDo(&theSound);
}
```

edited Feb 15 '12 at 11:32

sthlm58

437 3 15

answered Nov 1 '08 at 0:11

Federico A. Ramponi

33.4k 23 92 126

17 S.Lott, walking a tree is not actually the visitor pattern. (It's the "hierarchical visitor pattern", which is confusingly completely different.) There's no way to show the GoF Visitor pattern without using inheritance or interface implementation. – munificent Jan 28 '10 at 20:52

13 @Knownasilya - Thats not true. The &-Operator gives the address of the Sound-Object, which is needed by the interface. `letsDo(Operation *v)` needs a pointer. – AquilaRapax Feb 1 '13 at 12:58

3 just for clarity sake, is this example of visitor design pattern correct? – godzilla Jul 17 '13 at 14:23

4 After a lot of thinking, I wonder why you called two methods hereIsADog and hereIsACat although you already pass the Dog and the Cat to the methods. I would prefer a simple performTask(Object *obj) and you cast this object in the Operation class. (and in language supporting overriding, no need for casting) – Abdalrahman Shatou Dec 6 '13 at 18:01

5 In your "main" example at the end : `theSound.hereIsACat(c)` would have done the job, how do you justify for all the overhead introduced by the pattern ? double dispatching is the justification. – franssu Sep 18 '14 at 10:46

The reason for your confusion is probably that the Visitor is a fatal misnomer. Many (prominent[1]!) programmers have stumbled over this problem. What it actually does is implement double dispatching in languages that don't support it natively (most of them don't).

[1)] My favourite example is Scott Meyers, acclaimed author of “Effective C++”, who called this one of his most important C++ aha! moments *ever*.

answered Oct 31 '08 at 23:09

Konrad Rudolph
378k 95 746 993

3 +1 "there is no pattern" - the perfect answer. the most upvoted answer proves many c++ programmers are yet to realise the limitations of virtual functions over "adhoc" polymorphism using a type enum and switch case (the c way). It may be neater and invisible to use virtual, but it is still limited to single dispatch. In my personal opinion, this is c++'s biggest flaw. – user3125280 Jan 20 '14 at 5:51

@user3125280 I've read 4/5 articles and the Design Patterns chapter on the Visitor pattern now, and none of them explain the advantage of using this obscure pattern over a case stmt, or when you might use one over the other. Thx for at least bringing it up! – spinkus Feb 16 '14 at 13:36

4 @sam I’m pretty sure they do explain it – it’s the same advantage that you *always* get from subclassing / runtime polymorphism over `switch` : `switch` hard-codes the decision making at the client side (code duplication) and

doesn’t offer static type checking (check for completeness and distinctness of cases etc.). A visitor pattern is verified by the type checker, and usually makes the client code simpler. – Konrad Rudolph Feb 16 '14 at 15:52

@KonradRudolph thanks for that. Noting though, its not addressed explicitly in Patterns or the wikipedia article for instance. I don't disagree with you, but you could argue there are benefits to using a case stmt too so its strange its not generally contrasted: 1. you don't need an accept() method on objects of your collection. 2. The ~visitor can handle objects of unknown type. Thus the case stmt seems a better fit for operating on object structures with a changeable collection of types involved. Patterns does concede that the Visitor pattern is not well suited to such a scenario (p333). – spinkus Feb 17 '14 at 3:05

@SamPinkus konrad's spot on - that's why `virtual` like features are so useful in modern programming languages - they are the basic building block of extensible programs - in my opinion the c way (nested switch or pattern match, etc depending on your language of choice) is far cleaner in code that doesn't need to be extensible and I have been pleasantly surprised to see this style in complicated software like prover 9. More importantly any language that wants to provide extensibility should probably accommodate better dispatch patterns than recursive single dispatch (i.e. visitor). – user3125280 Feb 17 '14 at 10:31

Everyone here is correct, but I think it fails to address the "when". First, from Design Patterns:

> Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Now, let's think of a simple class hierarchy. I have classes 1, 2, 3 and 4 and methods A, B, C and D. Lay them out like in a spreadsheet: the classes are lines and the methods are columns.

Now, Object Oriented design presumes you are more likely to grow new classes than new methods, so adding more lines, so to speak, is easier. You just add a new class, specify what's different in that class, and inherits the rest.

Sometimes, though, the classes are relatively static, but you need to add more methods frequently -- adding columns. The standard way in an OO design would be to add such methods to all classes, which can be costly. The Visitor pattern makes this easy.

By the way, this is the problem that Scala's pattern matches intends to solve.

answered Jan 26 '09 at 2:00

Daniel C. Sobral

247k 73 435 631

The **Visitor** design pattern works really well for "recursive" structures like directory trees, XML structures, or document outlines.

A Visitor object visits each node in the recursive structure: each directory, each XML tag, whatever. The Visitor object doesn't loop through the structure. Instead Visitor methods are applied to each node of the structure.

Here's a typical recursive node structure. Could be a directory or an XML tag. [If your a Java person, imagine of a lot of extra methods to build and maintain the children list.]

```
class TreeNode( object ):
    def __init__( self, name, *children ):
        self.name= name
        self.children= children
    def visit( self, someVisitor ):
        someVisitor.arrivedAt( self )
        someVisitor.down()
        for c in self.children:
            c.visit( someVisitor )
        someVisitor.up()
```

The `visit` method applies a Visitor object to each node in the structure. In this case, it's a top-down visitor. You can change the structure of the `visit` method to do bottom-up or some other ordering.

Here's a superclass for visitors. It's used by the `visit` method. It "arrives at" each node in the structure. Since the `visit` method calls `up` and `down`, the visitor can keep track of the depth.

```
class Visitor( object ):
    def __init__( self ):
        self.depth= 0
    def down( self ):
        self.depth += 1
    def up( self ):
        self.depth -= 1
    def arrivedAt( self, aTreeNode ):
        print self.depth, aTreeNode.name
```

A subclass could do things like count nodes at each level and accumulate a list of nodes, generating a nice path hierarchical section numbers.

Here's an application. It builds a tree structure, `someTree`. It creates a `Visitor`, `dumpNodes`.

Then it applies the `dumpNodes` to the tree. The `dumpNode` object will "visit" each node in the tree.

```
someTree= TreeNode( "Top", TreeNode("c1"), TreeNode("c2"), TreeNode("c3") )
dumpNodes= Visitor()
someTree.visit( dumpNodes )
```

The TreeNode `visit` algorithm will assure that every TreeNode is used as an argument to the Visitor's `arrivedAt` method.

answered Nov 1 '08 at 2:44

S.Lott
305k 64 425 707

5 As others have stated this is the "hierarchical visitor pattern". – PPC-Coder Nov 29 '12 at 16:48

1 @PPC-Coder What's the difference between 'hierarchical visitor pattern' and visitor pattern? – Tim Lovell-Smith Aug 17 '15 at 17:22

3 The hierarchical visitor pattern is more flexible than the classic visitor pattern. For example, with the hierarchical pattern you can track the depth of traversal and decide which branch to traverse or stop traversing all together. The classic visitor doesn't have this concept and will visit all nodes. – PPC-Coder Aug 17 '15 at 17:42

One way to look at it is that the visitor pattern is a way of letting your clients add additional methods to all of your classes in a particular class hierarchy.

It is useful when you have a fairly stable class hierarchy, but you have changing requirements of what needs to be done with that hierarchy.

The classic example is for compilers and the like. An Abstract Syntax Tree (AST) can accurately define the structure of the programming language, but the operations you might want to do on the AST will change as your project advances: code-generators, pretty-printers, debuggers, complexity metrics analysis.

Without the Visitor Pattern, every time a developer wanted to add a new feature, they would need to add that method to every feature in the base class. This is particularly hard when the base classes appear in a separate library, or are produced by a separate team.

(I have heard it argued that the Visitor pattern is in conflict with good OO practices, because it moves the operations of the data away from the data. The Visitor pattern is useful in precisely the situation that the normal OO practices fail.)

answered Oct 31 '08 at 23:13

Oddthinking
13.5k 10 60 108

There are at least three very good reasons for using the Visitor Pattern:

1. Reduce proliferation of code which is only slightly different when data structures change.
2. Apply the same computation to several data structures, without changing the code which implements the computation.
3. Add information to legacy libraries without changing the legacy code.

Please have a look at an article I've written about this.

edited Sep 21 '13 at 12:06

answered Aug 8 '13 at 9:44

Richard Gomes
3,242 24 29

1 I commented on your article with the single biggest use that I've seen for visitor. Thoughts? – George Mauer Sep 24 '13 at 22:40

As Konrad Rudolph already pointed out, it is suitable for cases where we need **double dispatch**

Here is an example to show a situation where we need double dispatch & how visitor helps us in doing so.

Example :

Lets say I have 3 types of mobile devices - iPhone, Android, Windows Mobile.

All these three devices have a Bluetooth radio installed in them.

Lets assume that the blue tooth radio can be from 2 separate OEMs – Intel & Broadcom.

Just to make the example relevant for our discussion, lets also assume that the APIs exposes by Intel radio are different from the ones exposed by Broadcom radio.

This is how my classes look –

Now, I would like to introduce an operation – Switching On the Bluetooth on mobile device.

Its function signature should like something like this –

```
 void SwitchOnBlueTooth(IMobileDevice mobileDevice, IBlueToothRadio
blueToothRadio)
```

So depending upon **Right type of device** and **Depending upon right type of Bluetooth radio**, it can be switched on by **calling appropriate steps or algorithm**.

In principal, it becomes a 3 x 2 matrix, where-in I'm trying to vector the right operation depending upon the right type of objects involved.

A polymorphic behaviour depending upon the type of both the arguments.

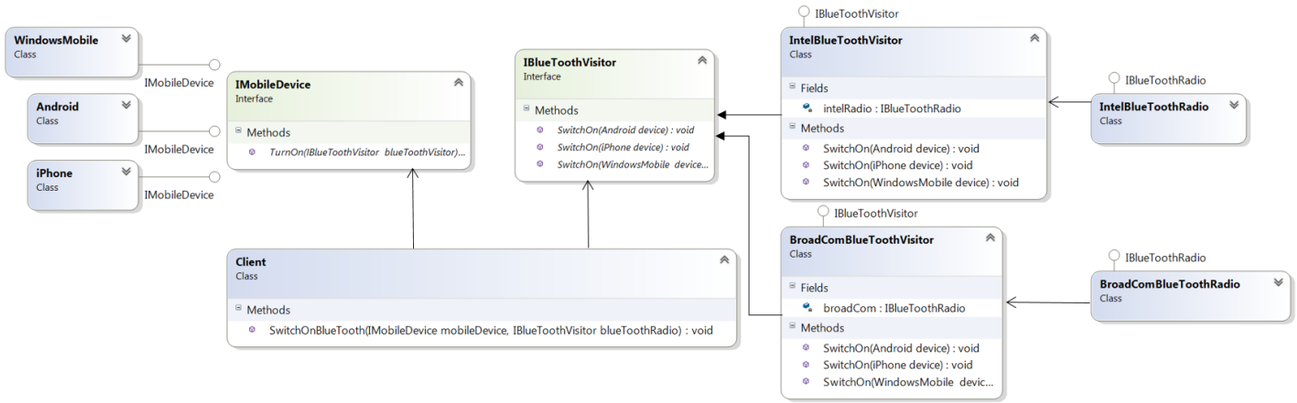
	Android	**WindowsMobile**	**iPhone**
Intel	Intel over Android	Intel over WM	Intel over iPhone
BroadCom	BroadCom over Android	BroadCom over WM	BroadCom over iPhone

Now, Visitor pattern can be applied to this problem. Inspiration comes from the Wikipedia page stating – *"In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate*

specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch."

Double dispatch is a necessity here due to the 3x2 matrix

Here is how the set up will look like -



I wrote the example to answer another question, the code & its explanation is mentioned here.

edited May 23 '17 at 12:02
Community ♦
1 1

answered Jul 13 '16 at 2:58
Kapoor
1,018 5 13

I found it easier in following links:

In http://www.remondo.net/visitor-pattern-example-csharp/ I found an example that shows an mock example that shows what is benefit of visitor pattern. Here you have different container classes for `Pill` :

```
namespace DesignPatterns
{
    public class BlisterPack
    {
        // Pairs so x2
        public int TabletPairs { get; set; }
    }

    public class Bottle
    {
```

```
        // Unsigned
        public uint Items { get; set; }
    }

    public class Jar
    {
        // Signed
        public int Pieces { get; set; }
    }
}
```

As you see in above, You `BilsterPack` contain pairs of Pills' so you need to multiply number of pair's by 2. Also you may notice that `Bottle` use `unit` which is different datatype and need to be cast.

So in main method you may calculate pill count using following code:

```
foreach (var item in packageList)
{
    if (item.GetType() == typeof (BlisterPack))
    {
        pillCount += ((BlisterPack) item).TabletPairs * 2;
    }
    else if (item.GetType() == typeof (Bottle))
    {
        pillCount += (int) ((Bottle) item).Items;
    }
    else if (item.GetType() == typeof (Jar))
    {
        pillCount += ((Jar) item).Pieces;
    }
}
```

Notice that above code violate `Single Responsibility Principle` . That means you must change main method code if you add new type of container. Also making switch longer is bad practice.

So by introducing following code:

```
public class PillCountVisitor : IVisitor
{
    public int Count { get; private set; }

    #region IVisitor Members

    public void Visit(BlisterPack blisterPack)
    {
        Count += blisterPack.TabletPairs * 2;
```

```
    }

    public void Visit(Bottle bottle)
    {
        Count += (int)bottle.Items;
    }

    public void Visit(Jar jar)
    {
        Count += jar.Pieces;
    }

    #endregion
}
```

You moved responsibility of counting number of `Pill` s to class called `PillCountVisitor` (And we removed switch case statement). That mean's whenever you need to add new type of pill container you should change only `PillCountVisitor` class. Also notice `IVisitor` interface is general for using in another scenarios.

By adding Accept method to pill container class:

```
public class BlisterPack : IAcceptor
{
    public int TabletPairs { get; set; }

    #region IAcceptor Members

    public void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }

    #endregion
}
```

we allow visitor to visit pill container classes.

At the end we calculate pill count using following code:

```
var visitor = new PillCountVisitor();

foreach (IAcceptor item in packageList)
{
    item.Accept(visitor);
}
```

That mean's: Every pill container allow the `PillCountVisitor` visitor to see their pills count. He know how to count your pill's.

At the `visitor.Count` has the value of pills.

In http://butunclebob.com/ArticleS.UncleBob.IuseVisitor you see real scenario in which you can not use polymorphism (the answer) to follow Single Responsibility Principle. In fact in:

```
public class HourlyEmployee extends Employee {
  public String reportQtdHoursAndPay() {
    //generate the line for this hourly employee
  }
}
```

the `reportQtdHoursAndPay` method is for reporting and representation and this violate the Single Responsibility Principle. So it is better to use visitor pattern to overcome the problem.

edited May 23 '17 at 12:18
Community ♦
1 1

answered Jul 6 '14 at 12:47
Seyed Morteza Mousavi
2,864 6 26 49

2 Hi Sayed, can you please edit your answer to add parts that you found most enlightening. SO generally discourages link-only answers since the goal is to be a knowledge database and links go down. – George Mauer Jul 6 '14 at 13:32

In my opinion, the amount of work to add a new operation is more or less the same using `Visitor Pattern` or direct modification of each element structure. Also, if I were to add new element class, say `Cow` , the Operation interface will be affected and this propagates to all existing class of elements, therefore requiring recompilation of all element classes. So what is the point?

answered Mar 9 '13 at 6:41
kaosad
693 1 7 13

4 Almost every time I've used Visitor is when you're working with traversing an object hierarchy. Consider a nested tree menu. You want to collapse all nodes. If you don't implement visitor you have to write graph traversal code. Or with visitor: `rootElement.visit (node) -> node.collapse()` . With visitor, each node implements the graph traversal for all its children so you're done. – George Mauer Mar 9 '13 at 20:44

@GeorgeMauer, the concept of double dispatch cleared up the motivation for me: either type-dependent logic is with

the type or, world of pain. The idea of distributing traversal logic still gives me pause. Is it more efficient? Is it more maintainable? What if "fold to level N" is added as a requirement? – nik.shornikov Apr 23 '15 at 18:36

@nik.shornikov efficiency should really not be a concern here. In almost any language, a few function calls is negligible overhead. Anything beyond that is micro-optimization. Is it more maintainable? Well, it depends. I think most times it is, sometimes it is not. As for "fold to level N". Easy pass in a `levelsRemaining` counter as a parameter. Decrement it before calling the next level of children. Inside of your visitor `if(levelsRemaining == 0) return .` – George Mauer Apr 23 '15 at 19:37

@GeorgeMauer, totally agreed on efficiency being a minor concern. But maintainability, e.g. overrides of the accept signature, are exactly what I think the decision should boil down to. – nik.shornikov Apr 29 '15 at 18:55

Visitor Pattern as the same underground implementation to Aspect Object programming..

For example if you define a new operation without changing the classes of the elements on which it operates

answered Jun 25 '13 at 22:07

mixturez

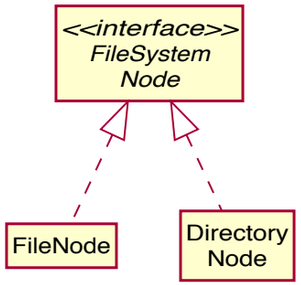
101 1 4

up for mentioning Aspect Object Programming – milesma Apr 8 '15 at 12:37

Cay Horstmann has a great example of where to apply Visitor in his OO Design and patterns book. He summarizes the problem:

> Compound objects often have a complex structure, composed of individual elements. Some elements may again have child elements. ... An operation on an element visits its child elements, applies the operation to them, and combines the results. ... However, it is not easy to add new operations to such a design.

The reason it's not easy is because operations are added within the structure classes themselves. For example, imagine you have a File System:



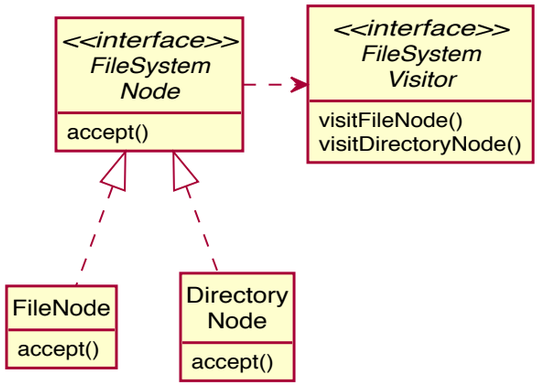
Here are some operations (functionalities) we might want to implement with this structure:

- Display the names of the node elements (a file listing)
- Display the calculated the size of the node elements (where a directory's size includes the size of all its child elements)
- etc.

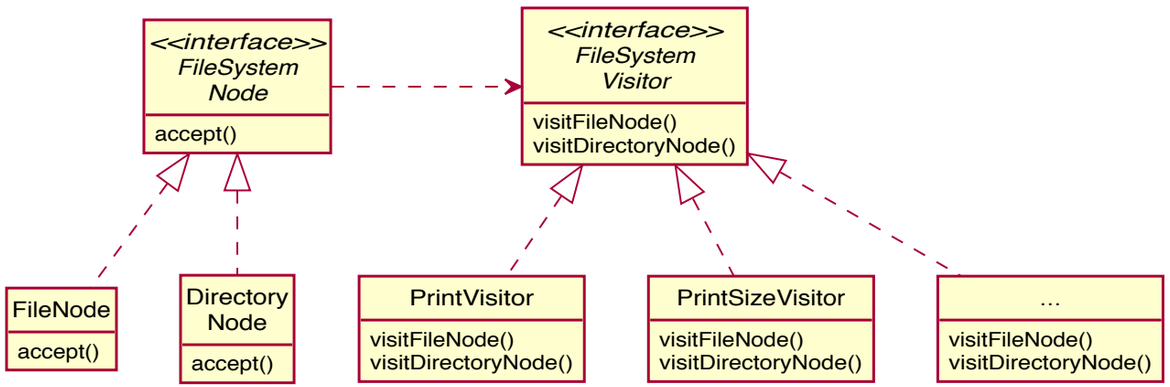
You could add functions to each class in the FileSystem to implement the operations (and people have done this in the past as it's very obvious how to do it). The problem is that whenever you add a new functionality (the "etc." line above), you might need to add more and more methods to the structure classes. At some point, after some number of operations you've added to your software, the methods in those classes don't make sense anymore in terms of the classes' functional cohesion. For example, you have a `FileNode` that has a method `calculateFileColorForFunctionABC()` in order to implement the latest visualization functionality on the file system.

The Visitor Pattern (like many design patterns) was born from the *pain and suffering* of developers who knew there was a better way to allow their code to change without requiring a lot of changes everywhere and also respecting good design principles (high cohesion, low coupling). It's my opinion that it's hard to understand the usefulness of a lot of patterns until you've felt that pain. Explaining the pain (like we attempt to do above with the "etc." functionalities that get added) takes up space in the explanation and is a distraction. Understanding patterns is hard for this reason.

Visitor allows us to decouple the functionalities on the data structure (e.g., `FileSystemNodes`) from the data structures themselves. The pattern allows the design to respect cohesion -- data structure classes are simpler (they have fewer methods) and also the functionalities are encapsulated into `Visitor` implementations. This is done via *double-dispatching* (which is the complicated part of the pattern): using `accept()` methods in the structure classes and `visitX()` methods in the Visitor (the functionality) classes:



This structure allows us to add new functionalities that work on the structure as concrete Visitors (without changing the structure classes).



For example, a `PrintNameVisitor` that implements the directory listing functionality, and a `PrintSizeVisitor` that implements the version with the size. We could imagine one day having an 'ExportXMLVisitor` that generates the data in XML, or another visitor that generates it in JSON, etc. We could even have a visitor that displays my directory tree using a graphical language such as DOT, to be visualized with another program.

As a final note: The complexity of Visitor with its double-dispatch means it is harder to understand, to code and to debug. In short, it has a high geek factor and goes agains the KISS principle. In a survey done by researchers, Visitor was shown to be a controversial pattern (there wasn't a consensus about its usefulness). Some experiments even showed it didn't make code easier to maintain.

edited Feb 8 '17 at 15:02

Community ♦
1 1

answered May 6 '15 at 13:27

Fuhrmanator
4,536 3 29 64

The directory structure I think is a good composite pattern but agree with your last paragraph. – zar May 30 at 17:31

Quick description of the visitor pattern. The classes that require modification must all implement the 'accept' method. Clients call this accept method to perform some new action on that family of classes thereby extending their functionality. Clients are able to use this one accept method to perform a wide range of new actions by passing in a different visitor class for each specific action. A visitor class contains multiple overridden visit methods defining how to achieve that same specific action for every class within the family. These visit methods get passed an instance on which to work.

When you might consider using it

1. When you have a family of classes you know your going to have to add many new actions them all, but for some reason you are not able to alter or recompile the family of classes in the future.
2. When you want to add a new action and have that new action entirely defined within one the visitor class rather than spread out across multiple classes.
3. When your boss says you must produce a range of classes which must do something **right now**!... but nobody actually knows exactly what that something is yet.

answered Feb 18 at 16:16

andrew pate
1,061 11 6

Based on the excellent answer of @Federico A. Ramponi.

Just imagine you have this hierarchy:

```
public interface IAnimal
{
    void DoSound();
}

public class Dog : IAnimal
```

```
{
    public void DoSound()
    {
        Console.WriteLine("Woof");
    }
}

public class Cat : IAnimal
{
    public void DoSound(IOperation o)
    {
        Console.WriteLine("Meaw");
    }
}
```

What happen if you need to add a "Walk" method here? That will be painful to the whole design.

At the same time, adding the "Walk" method generate new questions. What about "Eat" or "Sleep"? Must we really add a new method to the Animal hierarchy for every new action or operation that we want to add? That's ugly and most important, we will never be able to close the Animal interface. So, with the visitor pattern, we can add new method to the hierarchy without modifying the hierarchy!

So, just check and run this C# example:

```
using System;
using System.Collections.Generic;

namespace VisitorPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            var animals = new List<IAnimal>
            {
                new Cat(), new Cat(), new Dog(), new Cat(),
                new Dog(), new Dog(), new Cat(), new Dog()
            };

            foreach (var animal in animals)
            {
                animal.DoOperation(new Walk());
                animal.DoOperation(new Sound());
            }

            Console.ReadLine();
        }
```

```
}

public interface IOperation
{
    void PerformOperation(Dog dog);
    void PerformOperation(Cat cat);
}

public class Walk : IOperation
{
    public void PerformOperation(Dog dog)
    {
        Console.WriteLine("Dog walking");
    }

    public void PerformOperation(Cat cat)
    {
        Console.WriteLine("Cat Walking");
    }
}

public class Sound : IOperation
{
    public void PerformOperation(Dog dog)
    {
        Console.WriteLine("Woof");
    }

    public void PerformOperation(Cat cat)
    {
        Console.WriteLine("Meaw");
    }
}

public interface IAnimal
{
    void DoOperation(IOperation o);
}

public class Dog : IAnimal
{
    public void DoOperation(IOperation o)
    {
        o.PerformOperation(this);
    }
}

public class Cat : IAnimal
{
```

```
        public void DoOperation(IOperation o)
        {
            o.PerformOperation(this);
        }
    }
}
```

answered Mar 28 '16 at 23:48

Tomás Escamez

182 2 14

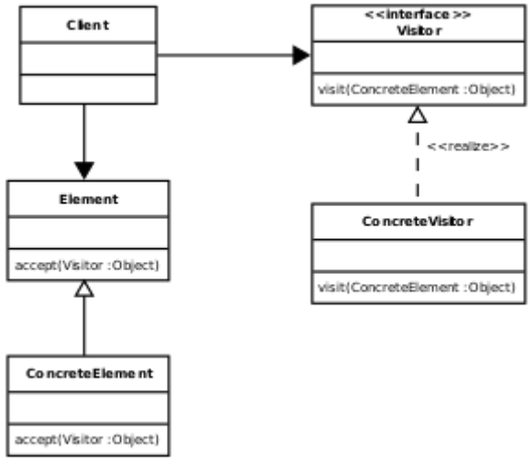
walk, eat aren't suitable examples since they're common to both `Dog` as well as `Cat` . You could've made them in base class so they're inherited or pick a suitable example. – Abhinav Gauniyal Sep 19 '16 at 7:37

sounds are different tho, good sample, but unsure if it has anything to do with the visitor pattern – DAG Mar 8 '17 at 20:03

Visitor

> Visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function

Visitor structure:



Use Visitor pattern if:

1. *Similar operations have to be performed* on objects of different types grouped in a structure
2. You need to execute many distinct and unrelated operations. *It separates Operation from objects Structure*
3. New operations have to be added without change in object structure
4. *Gather related operations into a single class* rather than force you to change or derive classes
5. Add functions to class libraries for which you *either do not have the source or cannot change the source*

Even though *Visitor* pattern provides flexibility to add new operation without changing the existing code in Object, this flexibility has come with a drawback.

If a new Visitable object has been added, it requires code changes in Visitor & ConcreteVisitor classes. There is a workaround to address this issue : Use reflection, which will have impact on performance.

Code snippet:

```
import java.util.HashMap;

interface Visitable{
    void accept(Visitor visitor);
}
```

```
interface Visitor{
    void logGameStatistics(Chess chess);
    void logGameStatistics(Checkers checkers);
    void logGameStatistics(Ludo ludo);
}
class GameVisitor implements Visitor{
    public void logGameStatistics(Chess chess){
        System.out.println("Logging Chess statistics: Game Completion duration,
number of moves etc..");
    }
    public void logGameStatistics(Checkers checkers){
        System.out.println("Logging Checkers statistics: Game Completion
duration, remaining coins of loser");
    }
    public void logGameStatistics(Ludo ludo){
        System.out.println("Logging Ludo statistics: Game Completion duration,
remaining coins of loser");
    }
}

abstract class Game{
    // Add game related attributes and methods here
    public Game(){

    }
    public void getNextMove(){};
    public void makeNextMove(){}
    public abstract String getName();
}
class Chess extends Game implements Visitable{
    public String getName(){
        return Chess.class.getName();
    }
    public void accept(Visitor visitor){
        visitor.logGameStatistics(this);
    }
}
class Checkers extends Game implements Visitable{
    public String getName(){
        return Checkers.class.getName();
    }
    public void accept(Visitor visitor){
        visitor.logGameStatistics(this);
    }
}
class Ludo extends Game implements Visitable{
    public String getName(){
        return Ludo.class.getName();
    }
```

```
    public void accept(Visitor visitor){
        visitor.logGameStatistics(this);
    }
}

public class VisitorPattern{
    public static void main(String args[]){
        Visitor visitor = new GameVisitor();
        Visitable games[] = { new Chess(),new Checkers(), new Ludo()};
        for (Visitable v : games){
            v.accept(visitor);
        }
    }
}
```

Explanation:

1. `Visitable` (`Element`) is an interface and this interface method has to be added to a set of classes.
2. `Visitor` is an interface, which contains methods to perform an operation on `Visitable` elements.
3. `GameVisitor` is a class, which implements `Visitor` interface (`ConcreteVisitor`).
4. Each `Visitable` element accept `Visitor` and invoke a relevant method of `Visitor` interface.
5. You can treat `Game` as `Element` and concrete games like `Chess,Checkers and Ludo` as `ConcreteElements` .

In above example, `Chess, Checkers and Ludo` are three different games (and `Visitable` classes). On one fine day, I have encountered with a scenario to log statistics of each game. So without modifying individual class to implement statistics functionality, you can centralise that responsibility in `GameVisitor` class, which does the trick for you without modifying the structure of each game.

output:

```
Logging Chess statistics: Game Completion duration, number of moves etc..
Logging Checkers statistics: Game Completion duration, remaining coins of loser
Logging Ludo statistics: Game Completion duration, remaining coins of loser
```

Refer to

oodesign article

sourcemaking article

for more details

Decorator

> pattern allows behaviour to be added to an individual object, either statically or dynamically, without affecting the behaviour of other objects from the same class

Related posts:

Decorator Pattern for IO

When to Use the Decorator Pattern?

edited Sep 24 '17 at 3:43

answered Feb 15 '16 at 10:19

Ravindra babu

26k 5 134 127

Double dispatch is just **one reason among others to use this pattern**.
But note that it is the single way to implement double or more dispatch in languages that uses a single dispatch paradigm.

Here are reasons to use the pattern :

1) **We want to define new operations without changing the model at each time** because the model doesn't change often wile operations change frequently.

2) **We don't want to couple model and behavior** because **we want to have a reusable model** in multiple applications or **we want to have an extensible model** that allow client classes to define their behaviors with their own classes.

3) We have common operations that depend on the concrete type of the model but **we don't want to implement the logic in each subclass as that would explode common logic in multiple classes and so in multiple places**.

4) We are using a domain model design and **model classes of the same hierarchy perform too many distinct things that could be gathered somewhere else**.

5) **We need a double dispatch**.
We have variables declared with interface types and we want to be able to process them according their runtime type … of course without using `if (myObj instanceof Foo) {}` or any trick.

The idea is for example to pass these variables to methods that declares a concrete type of the interface as parameter to apply a specific processing. This way of doing is not possible out of the box with languages relies on a single-dispatch because the chosen invoked at runtime depends only on the runtime type of the receiver.
Note that in Java, the method (signature) to call is chosen at compile time and it depends on the declared type of the parameters, not their runtime type.

The last point that is a reason to use the visitor is also a consequence because as you implement the visitor (of course for languages that doesn't support multiple dispatch), you necessarily need to introduce a double dispatch implementation.

Note that the traversal of elements (iteration) to apply the visitor on each one is not a reason to use the pattern.
You use the pattern because you split model and processing.
And by using the pattern, you benefit in addition from an iterator ability.
This ability is very powerful and goes beyond iteration on common type with a specific method as `accept()` is a generic method.
It is a special use case. So I will put that to one side.

Example in Java

I will illustrate the added value of the pattern with a chess example where we would like to define processing as player requests a piece moving.

Without the visitor pattern use, we could define piece moving behaviors directly in the pieces subclasses.
We could have for example a `Piece` interface such as :

```
public interface Piece{

    boolean checkMoveValidity(Coordinates coord);

    void performMove(Coordinates coord);

    Piece computeIfKingCheck();

}
```

Each Piece subclass would implement it such as :

```
public class Pawn implements Piece{
```

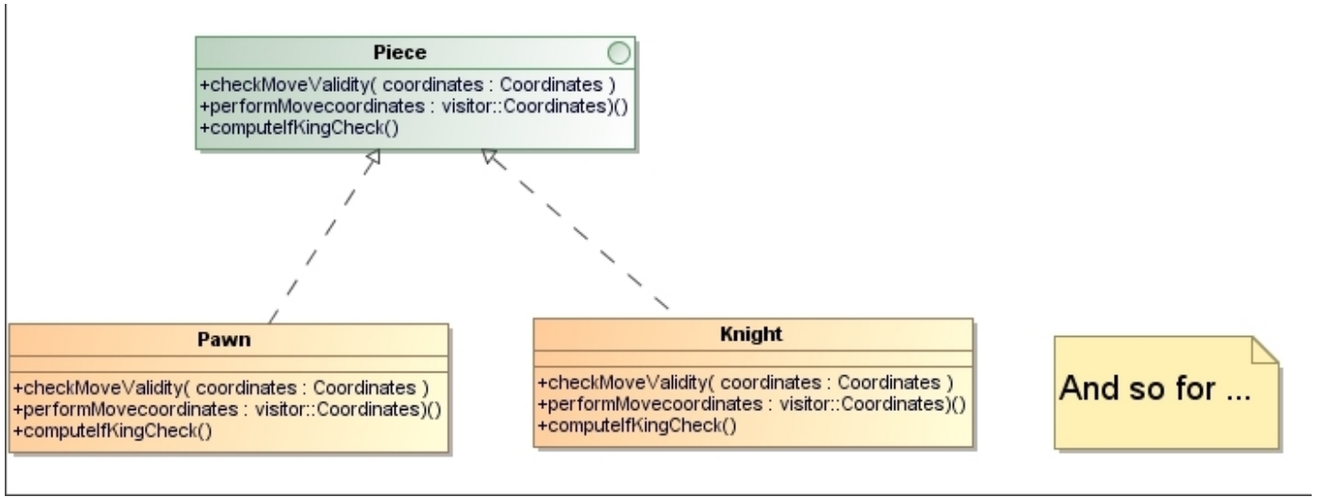
```
    @Override
    public boolean checkMoveValidity(Coordinates coord) {
        ...
    }

    @Override
    public void performMove(Coordinates coord) {
        ...
    }

    @Override
    public Piece computeIfKingCheck() {
        ...
    }

}
```

And the same thing for all Piece subclasses.
Here is a diagram class that illustrates this design :



This approach presents three important drawbacks :

– behaviors such as `performMove()` or `computeIfKingCheck()` will very probably use common logic.
For example whatever the concrete `Piece`, `performMove()` will finally set the current piece to a specific location and potentially takes the opponent piece.
Splitting related behaviors in multiple classes instead of gathering them defeats in a some way the single responsibility pattern. Making their maintainability harder.

– processing as `checkMoveValidity()` should not be something that the `Piece` subclasses may see or change.
It is check that goes beyond human or computer actions. This check is performed at each action requested by a player to ensure that the requested piece move is valid.
So we even don't want to provide that in the `Piece` interface.

– In chess games challenging for bot developers, generally the application provides a standard API (`Piece` interfaces, subclasses, Board, common behaviors, etc…) and let developers enrich their bot strategy.
To be able to do that, we have to propose a model where data and behaviors are not tightly coupled in the `Piece` implementations.

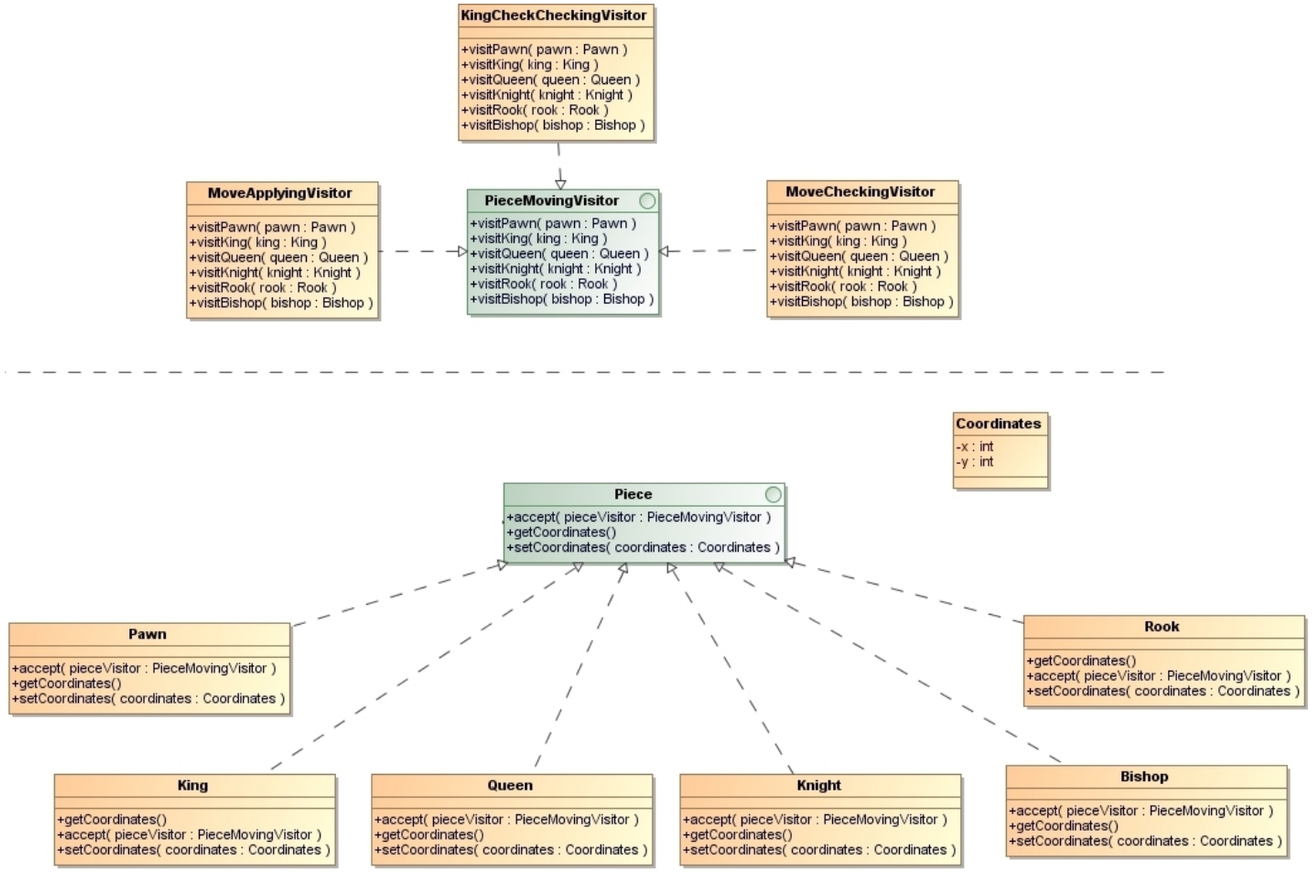
So let's go to use the visitor pattern !

We have two kinds of structure :

– the model classes that accept to be visited (the pieces)

– the visitors that visit them (moving operations)

Here is a class diagram that illustrates the pattern :



In the upper part we have the visitors and in the lower part we have the model classes.

Here is the `PieceMovingVisitor` interface (behavior specified for each kind of `Piece`) :

```
public interface PieceMovingVisitor {

    void visitPawn(Pawn pawn);

    void visitKing(King king);

    void visitQueen(Queen queen);

    void visitKnight(Knight knight);

    void visitRook(Rook rook);
```

```
    void visitBishop(Bishop bishop);

}
```

The Piece is defined now :

```
public interface Piece {

    void accept(PieceMovingVisitor pieceVisitor);

    Coordinates getCoordinates();

    void setCoordinates(Coordinates coordinates);

}
```

Its key method is :

```
void accept(PieceMovingVisitor pieceVisitor);
```

It provides the first dispatch : a invocation based on the `Piece` receiver.
At compile time, the method is bound to the `accept()` method of the Piece interface and at runtime, the bounded method will be invoked on the runtime `Piece` class.
And it is the `accept()` method implementation that will perform a second dispatch.

Indeed, each `Piece` subclass that wants to be visited by a `PieceMovingVisitor` object invokes the `PieceMovingVisitor.visit()` method by passing as argument itself.
In this way, the compiler bounds as soon as the compile time, the type of the declared parameter with the concrete type.
There is the second dispatch.
Here is the `Bishop` subclass that illustrates that :

```
public class Bishop implements Piece {

    private Coordinates coord;

    public Bishop(Coordinates coord) {
        super(coord);
    }

    @Override
    public void accept(PieceMovingVisitor pieceVisitor) {
        pieceVisitor.visitBishop(this);
```

```
    }

    @Override
    public Coordinates getCoordinates() {
        return coordinates;
    }

   @Override
    public void setCoordinates(Coordinates coordinates) {
        this.coordinates = coordinates;
   }

}
```

And here an usage example :

```
// 1. Player requests a move for a specific piece
Piece piece = selectPiece();
Coordinates coord = selectCoordinates();

// 2. We check with MoveCheckingVisitor that the request is valid
final MoveCheckingVisitor moveCheckingVisitor = new MoveCheckingVisitor(coord);
piece.accept(moveCheckingVisitor);

// 3. If the move is valid, MovePerformingVisitor performs the move
if (moveCheckingVisitor.isValid()) {
    piece.accept(new MovePerformingVisitor(coord));
}
```

Visitor drawbacks

The Visitor pattern is a very powerful pattern but it also has some important limitations that you should consider before using it.

1) Risk to reduce/break the encapsulation

In some kinds of operation, the visitor pattern may reduce or break the encapsulation of domain objects.

For example, as the `MovePerformingVisitor` class needs to set the coordinates of the actual piece, the `Piece` interface has to provide a way to do that :

```
void setCoordinates(Coordinates coordinates);
```

The responsibility of `Piece` coordinates changes is now open to other classes than `Piece` subclasses. Moving the processing performed by the visitor in the `Piece` subclasses is not an option either.

It will indeed create another issue as the `Piece.accept()` accepts any visitor implementation. It doesn't know what the visitor performs and so no idea about whether and how to change the Piece state.
A way to identify the visitor would be to perform a post processing in `Piece.accept()` according to the visitor implementation. It would be a very bad idea as it would create a high coupling between Visitor implementations and Piece subclasses and besides it would probably require to use trick as `getClass()`, `instanceof` or any marker identifying the Visitor implementation.

2) Requirement to change the model

Contrary to some other behavioral design patterns as `Decorator` for example, the visitor pattern is intrusive.
We indeed need to modify the initial receiver class to provide an `accept()` method to accept to be visited.
We didn't have any issue for `Piece` and its subclasses as these are **our classes**.
In built-in or third party classes, things are not so easy.
We need to wrap or inherit (if we can) them to add the `accept()` method.

3) Indirections

The pattern creates multiples indirections.
The double dispatch means two invocations instead of a single one :

```
call the visited (piece) -> that calls the visitor (pieceMovingVisitor)
```

And we could have additional indirections as the visitor changes the visited object state.
It may look like a cycle :

```
call the visited (piece) -> that calls the visitor (pieceMovingVisitor) -> that
calls the visited (piece)
```

edited Dec 25 '17 at 23:21

answered Dec 25 '17 at 12:47

davidxxx
49k 5 36 63

While I have understood the how and when, I have never understood the why. In case it helps anyone with a background in a language like C++, you want to read this very carefully.

For the lazy, we use the visitor pattern because **"while virtual functions are dispatched dynamically in C++, function overloading is done statically"**.

Or, put another way, to make sure that CollideWith(ApolloSpacecraft&) is called when you pass in a SpaceShip reference that is actually bound to an ApolloSpacecraft object.

```
class SpaceShip {};
class ApolloSpacecraft : public SpaceShip {};
class ExplodingAsteroid : public Asteroid {
public:
  virtual void CollideWith(SpaceShip&) {
    cout << "ExplodingAsteroid hit a SpaceShip" << endl;
  }
  virtual void CollideWith(ApolloSpacecraft&) {
    cout << "ExplodingAsteroid hit an ApolloSpacecraft" << endl;
  }
}
```

answered Nov 25 '13 at 23:04

Carl

28.6k 6 60 89

1 The use of dynamic dispatch in the visitor pattern completely perplexes me. Suggested uses of the pattern describe branching that could be done at compile time. These cases would seemingly be better off with a function template. – Praxeolitic Dec 20 '14 at 1:50

I really like the description and the example from http://python-3-patterns-idioms-test.readthedocs.io/en/latest/Visitor.html.

> The assumption is that you have a primary class hierarchy that is fixed; perhaps it's from another vendor and you can't make changes to that hierarchy. However, your intent is that you'd like to add new polymorphic methods to that hierarchy, which means that normally you'd have to add something to the base class interface. So the dilemma is that you need to add methods to the base class, but you can't touch the base class. How do you get around this?
>
> The design pattern that solves this kind of problem is called a "visitor" (the final one in the Design Patterns book), and it builds on the double dispatching scheme shown in the last section.
>
> The visitor pattern allows you to extend the interface of the primary type by creating a separate class hierarchy of type Visitor to virtualize the operations performed upon the primary type. The objects of the primary type simply "accept" the visitor, then call the visitor's dynamically-bound member function.

answered Nov 4 '16 at 15:35

wojcikstefan
671 7 7

While technically the Visitor pattern this is really just basic double dispatch from their example. I would argue the usefulness is not particularly visible from this alone. – George Mauer Nov 4 '16 at 16:35

When you want to have function objects on union data types, you will need visitor pattern.

You might wonder what function objects and union data types are, then it's worth reading http://www.ccs.neu.edu/home/matthias/htdc.html

answered Jun 14 '17 at 14:40

Wei Qiu
455 6 13

Thanks for the awesome explanation of @Federico A. Ramponi, I just made this in **java** version. Hope it might be helpful.

Also just as @Konrad Rudolph pointed out, it's actually a **double dispatch** using **two** concrete instances together to determine the run-time methods.

So actually there is no need to create a **common** interface for the *operation* executor as long as we have the *operation* interface properly defined.

```
import static java.lang.System.out;
public class Visitor_2 {
    public static void main(String...args) {
        Hearen hearen = new Hearen();
        FoodImpl food = new FoodImpl();
        hearen.showTheHobby(food);
        Katherine katherine = new Katherine();
        katherine.presentHobby(food);
    }
}

interface Hobby {
    void insert(Hearen hearen);
```

```
    void embed(Katherine katherine);
}


class Hearen {
    String name = "Hearen";
    void showTheHobby(Hobby hobby) {
        hobby.insert(this);
    }
}

class Katherine {
    String name = "Katherine";
    void presentHobby(Hobby hobby) {
        hobby.embed(this);
    }
}

class FoodImpl implements Hobby {
    public void insert(Hearen hearen) {
        out.println(hearen.name + " start to eat bread");
    }
    public void embed(Katherine katherine) {
        out.println(katherine.name + " start to eat mango");
    }
}
```

As you expect, a **common** interface will bring us more clarity though it's actually not the *essential* part in this pattern.

```
import static java.lang.System.out;
public class Visitor_2 {
    public static void main(String...args) {
        Hearen hearen = new Hearen();
        FoodImpl food = new FoodImpl();
        hearen.showHobby(food);
        Katherine katherine = new Katherine();
        katherine.showHobby(food);
    }
}

interface Hobby {
    void insert(Hearen hearen);
    void insert(Katherine katherine);
}

abstract class Person {
    String name;
```

```
    protected Person(String n) {
        this.name = n;
    }
    abstract void showHobby(Hobby hobby);
}

class Hearen extends  Person {
    public Hearen() {
        super("Hearen");
    }
    @Override
    void showHobby(Hobby hobby) {
        hobby.insert(this);
    }
}

class Katherine extends Person {
    public Katherine() {
        super("Katherine");
    }

    @Override
    void showHobby(Hobby hobby) {
        hobby.insert(this);
    }
}

class FoodImpl implements Hobby {
    public void insert(Hearen hearen) {
        out.println(hearen.name + " start to eat bread");
    }
    public void insert(Katherine katherine) {
        out.println(katherine.name + " start to eat mango");
    }
}
```

edited Jun 9 at 3:40

answered May 29 at 3:34

Hearen

801 6 19

---

your question is when to know:

i do not first code with visitor pattern. i code standard and wait for the need to occur & then refactor. so lets say you have multiple payment systems that you installed one at a time. At checkout time you could have

many if conditions (or instanceOf) , for example :

```
//psuedo code
    if(payPal)
    do paypal checkout
    if(stripe)
    do strip stuff checkout
    if(payoneer)
    do payoneer checkout
```

now imagine i had 10 payment methods, it gets kind of ugly. So when you see that kind of pattern occuring visitor comes in handly to seperate all that out and you end up calling something like this afterwards:

```
new PaymentCheckoutVistor(paymentType).visit()
```

You can see how to implement it from the number of examples here, im just showing you a usecase.

answered Jun 25 at 3:15

j2emanue

18.1k 15 115 214

---

protected by DNA Jul 6 '14 at 14:07

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?