Alternative to the visitor pattern?

I am looking for an alternative to the visitor pattern. Let me just focus on a couple of pertinent aspects of the pattern, while skipping over unimportant details. I'll use a Shape example (sorry!):

1. You have a hierarchy of objects that implement the IShape interface
2. You have a number of global operations that are to be performed on all objects in the hierarchy, e.g. Draw, WriteToXml etc...
3. It is tempting to dive straight in and add a Draw() and WriteToXml() method to the IShape interface. This is not necessarily a good thing - whenever you wish to add a new operation that is to be performed on all shapes, each IShape-derived class must be changed
4. Implementing a visitor for each operation i.e. a Draw visitor or a WirteToXml visitor encapsulates all the code for that operation in one class. Adding a new operation is then a matter of creating a new visitor class that performs the operation on all types of IShape
5. When you need to add a new IShape-derived class, you essentially have the same problem as you did in 3 - all visitor classes must be changed to add a method to handle the new IShape-derived type

Most places where you read about the visitor pattern state that point 5 is pretty much the main criteria for the pattern to work and I totally agree. If the number of IShape-derived classes is fixed, then this can be a quite elegant approach.

So, the problem is when a new IShape-derived class is added - each visitor implementation needs to add a new method to handle that class. This is, at best, unpleasant and, at worst, not possible and shows that this pattern is not really designed to cope with such changes.

So, the question is has anybody come across alterative approaches to handling this situation?

oop design-patterns visitor

edited Feb 13 '16 at 23:59

Dave Schweisguth
22.7k 7 60 89

asked Jun 12 '09 at 10:04

Steg
7,847 3 19 17

Just an aside note, as it is unlikely that you can change your language: There are languages that directly support multiple dispatch generic functions. – Svante Jun 12 '09 at 14:21

7 Great question. I just wanted to provide a counterpoint. Sometimes your issue with (5) can be a good thing. I use the visitor pattern when I have some functionality that must be updated when a new IShape subtype is defined. I have a IShapeVisitor interface that defines what methods are needed. As long as that interface is updated with the new subtype, my code doesn't build until the critical functionality is updated. For some situations, this can be very helpful. – oillio Apr 4 '13 at 19:19

1 I agree with @oillio, but then you could also enforce it as an abstract method on IShape. What the Visitor pattern buys you in a pure OO language is locality of the function (vs. locality of class) and thereby a seperation of concerns. In any case, usign the visitor pattern should explicitly break at compile time when you want to force addition of new types to be reviewed carefully! – Johannes Rudolph Jul 18 '15 at 10:37

1 "all visitor classes must be changed to add a method to handle the new IShape-derived type": I would not say that's a "problem". I would say this is a very nice security in your design. It guarantees that the newly added class is considered in every place where you do per-type specific operations (i.e.: in every place where we defined a visitor....if you don't consider the newly added type, compiler won't let you go...). – jpo38 Feb 9 '16 at 19:59

7 Answers

You might want to have a look at the Strategy pattern. This still gives you a separation of concerns while still being able to add new functionality without having to change each class in your hierarchy.

```
class AbstractShape
{
    IXmlWriter _xmlWriter = null;
    IShapeDrawer _shapeDrawer = null;

    public AbstractShape(IXmlWriter xmlWriter,
                IShapeDrawer drawer)
    {
        _xmlWriter = xmlWriter;
        _shapeDrawer = drawer;
    }
```

```
    //...
    public void WriteToXml(IStream stream)
    {
        _xmlWriter.Write(this, stream);

    }

    public void Draw()
    {
        _drawer.Draw(this);
    }

    // any operation could easily be injected and executed
    // on this object at run-time
    public void Execute(IGeneralStrategy generalOperation)
    {
        generalOperation.Execute(this);
    }
}
```

More information is in this related discussion:

> Should an object write itself out to a file, or should another object act on it to perform I/O?

edited May 23 '17 at 12:17

Community ♦

1 1

answered Jun 12 '09 at 10:11

Dirk Vollmar

132k 44 217 281

I've marked this as the answer to my question as I think this, or some minor variation on it, probably fits into what I want to do. For anyone whose interested, I've added an "answer" that describes some of my thoughts on the problem – Steg Jun 22 '09 at 12:51

ok - changed my mind about the answer thing - I'll try to condense it into a comment (following) – Steg Jun 22 '09 at 12:52

3 I think there is a fundamental conflict here - if you have a bunch of things and a bunch of actions that can be performed on these things then adding a new thing means that you must define the effect of all actions on it and vice versa - there is no escaping this. The visitor provides a very simple, elegant way of adding new actions at the expense of making it difficult to add new things. If this constraint must be relaxed, the you have to pay. I was hoping there might be solution that has the elegance and simplicity of the visitor, but as I suspected, I don't think one exists...cont'd... – Steg Jun 22 '09 at 13:06

so, I think that this answer probably suits my current needs and I find it appealing because it is simpler than some of the other suggestions. Anyway, thanks to all for taking time to answer – Steg Jun 22 '09 at 13:09

as a fact with this solution you still need to add corresponding handling for new types, but now you have new fields in base class (how many it may become?). and you have to initialize handlers for new actions (where? when? parameters in constructor?). how many places in code you'll have to trace even if you have factories. the purpose of visitor is combine all newly added concerns in one place. it is not ideal as it could be in a perfect world, but it still is in one place. and it will generate compile time error if there is no handler implemented. also a default visitor, can simplify things – jungle_mole Apr 11 '15 at 15:45

There is the "Visitor Pattern With Default", in which you do the visitor pattern as normal but then define an abstract class that implements your `IShapeVisitor` class by delegating everything to an abstract method with the signature `visitDefault(IShape)`.

Then, when you define a visitor, extend this abstract class instead of implementing the interface directly. You can override the `visit` * methods you know about at that time, and provide for a sensible default. However, if there really isn't any way to figure out sensible default behavior ahead of time, you should just implement the interface directly.

When you add a new `IShape` subclass, then, you fix the abstract class to delegate to its `visitDefault` method, and every visitor that specified a default behavior gets that behavior for the new `IShape`.

A variation on this if your `IShape` classes fall naturally into a hierarchy is to make the abstract class delegate through several different methods; for example, an `DefaultAnimalVisitor` might do:

```
public abstract class DefaultAnimalVisitor implements IAnimalVisitor {
  // The concrete animal classes we have so far: Lion, Tiger, Bear, Snake
  public void visitLion(Lion l)   { visitFeline(l); }
  public void visitTiger(Tiger t) { visitFeline(t); }
  public void visitBear(Bear b)   { visitMammal(b); }
  public void visitSnake(Snake s) { visitDefault(s); }

  // Up the class hierarchy
  public void visitFeline(Feline f) { visitMammal(f); }
  public void visitMammal(Mammal m) { visitDefault(m); }

  public abstract void visitDefault(Animal a);
}
```

This lets you define visitors that specify their behavior at whatever level of specificity you wish.

Unfortunately, there is no way to avoid doing something to specify how visitors will behave with a new class - either you can set up a default ahead of time, or you can't. (See also the second panel of this cartoon)

answered Jun 12 '09 at 10:35

Daniel Martin

18.6k 3 42 64

I maintain a CAD/CAM software for metal cutting machine. So I have some experience with this issues.

When we first converted our software (it was first released in 1985!) to a object oriented designed I did just what you don't like. Objects and Interfaces had Draw, WriteToFile, etc. Discovering and reading about Design Patterns midway through the conversion helped a lot but there were still a lot of bad code smells.

Eventually I realized that none of these types of operations were really the concern of the object. But rather the various subsystems that needed to do the various operations. I handled this by using what is now called a Passive View Command object, and well defined Interface between the layers of software.

Our software is structured basically like this

- The Forms implementing various Form Interface. These forms are a thing shell passing events to the UI Layer.
- UI layer that receives Events and manipulate forms through the Form interface.
- The UI Layer will execute commands that all implement the Command interface
- The UI Object have interfaces of their own that the command can interact with.
- The Commands get the information they need, process it, manipulates the model and then report back to the UI Objects which then does anything needed with the forms.
- Finally the models which contains the various objects of our system. Like Shape Programs, Cutting Paths, Cutting Table, and Metal Sheets.

So Drawing is handled in the UI Layer. We have different software for different machines. So while all of our software share the same model and reuse many of the same commands. They handle things like drawing very different. For a example a cutting table is draw different for a router machine versus a machine using a

plasma torch despite them both being esstentially a giant X-Y flat table. This because like cars the two machines are built differently enough so that there is a visual difference to the customer.

As for shapes what we do is as follows

We have shape programs that produce cutting paths through the entered parameters. The cutting path knows which shape program produced. However a cutting path isn't a shape. It just the information needed to draw on the screen and to cut the shape. One reason for this design is that cutting paths can be created without a shape program when they are imported from a external app.

This design allows us to separate the design of the cutting path from the design of the shape which are not always the same thing. In your case likely all you need to package is the information needed to draw the shape.

Each shape program has a number of views implementing a IShapeView Interface. Through the IShapeView interface the shape program can tell the generic shape form we have how to setup itself up to show the parameters of that shape. The generic shape form implements a IShapeForm interface and registers itself with the ShapeScreen Object. The ShapeScreen Object registers itself with our application object. The shape views use whatever shapescreen that registers itself with the application.

The reason for the multiple views that we have customers that like to enter shapes in different ways. Our customer base is split in half between those who like to enter shape parameters in a table form and those who like to enter with a graphical representation of the shape in front of them. We also need to access the parameters at times through a minimal dialog rather than our full shape entry screen. Hence the multiple views.

Commands that manipulate shapes fall in one of two catagories. Either they manipulate the cutting path or they manipulate the shape parameters. To manipulate the shape parameters generally we either throw them back into the shape entry screen or show the minimal dialog. Recalculate the shape, and display it in the same location.

For the cutting path we bundled up each operation in a separate command object. For example we have command objects

ResizePath RotatePath MovePath SplitPath and so on.

When we need to add new functionality we add another command object, find a menu, keyboard short or toolbar button slot in the right UI screen and setup the UI object to ececute that command.

For example

```
    CuttingTableScreen.KeyRoute.Add vbShift+vbKeyF1, New MirrorPath
```

or

```
    CuttingTableScreen.Toolbar("Edit Path").AddButton
 Application.Icons("MirrorPath"),"Mirror Path", New MirrorPath
```

In both instances the Command object MirrorPath is being associated with a desired UI element. In the execute method of MirrorPath is all the code needed to mirror the path in a particular axis. Likely the command will have it's own dialog or use one of the UI elements to ask the user which axis to mirror. None of this is making a visitor, or adding a method to the path.

You will find that a lot can be handled through bundling actions into commands. However I caution that is not a black or white situation. You will still find that certain things work better as methods on the original object. In may experience I found that perhaps 80% of what I used to do in methods were able to be moved into the command. The last 20% just plain work better on the object.

Now some may not like this because it seems to violate encapsulations. From maintaining our software as a object oriented system for the last decade I have to say the MOST important long term thing you can do is clearly document the interactions between the different layers of your software and between the different objects.

Bundling actions into Command objects helps with this goal way better than a slavish devotion to the ideals of encapsulation. Everything that is needs to be done to Mirror a Path is bundled in the Mirror Path Command Object.

answered Jun 12 '09 at 13:16

RS Conley
6,823 13 34

The solution seems interesting, but i would appreciate if you could refer me to example code for such solution to better understand the concept. – Basil Musa Dec 4 '17 at 13:13

Visitor design pattern is a workaround, not a solution to the problem. Short answer would be pattern matching.

edited Dec 23 '15 at 9:50

answered Dec 23 '15 at 9:43

Marko Tunjic

Regardless of what path you take, the implementation of alternate functionality that is currently provided by the Visitor pattern will have to 'know' something about the concrete implementation of the interface that it is working on. So there is no getting around the fact that you are going to have to write addition 'visitor' functionality for each additional implementation. That said what you are looking for is a more flexible and structured approach to creating this functionality.

You need to separate out the visitor functionality from the interface of the shape.

What I would propose is a creationist approach via an abstract factory to create replacement implementations for visitor functionality.

```
public interface IShape {
  // .. common shape interfaces
}

//
// This is an interface of a factory product that performs 'work' on the shape.
//
public interface IShapeWorker {
     void process(IShape shape);
}

//
// This is the abstract factory that caters for all implementations of
// shape.
//
public interface IShapeWorkerFactory {
    IShapeWorker build(IShape shape);
    ...
}

//
// In order to assemble a correct worker we need to create
// and implementation of the factory that links the Class of
// shape to an IShapeWorker implementation.
// To do this we implement an abstract class that implements IShapeWorkerFactory
//
public AbsractWorkerFactory implements IShapeWorkerFactory {

    protected Hashtable map_ = null;

    protected AbstractWorkerFactory() {
```

```
            map_ = new Hashtable();
            CreateWorkerMappings();
        }

        protected void AddMapping(Class c, IShapeWorker worker) {
               map_.put(c, worker);
        }

        //
        // Implement this method to add IShape implementations to IShapeWorker
        // implementations.
        //
        protected abstract void CreateWorkerMappings();

        public IShapeWorker build(IShape shape) {
             return (IShapeWorker)map_.get(shape.getClass())
        }
    }

    //
    // An implementation that draws circles on graphics
    //
    public GraphicsCircleWorker implements IShapeWorker {

         Graphics graphics_ = null;

         public GraphicsCircleWorker(Graphics g) {
            graphics_ = g;
         }

         public void process(IShape s) {
           Circle circle = (Circle)s;
           if( circle != null) {
              // do something with it.
              graphics_.doSomething();
           }
         }

    }

    //
    // To replace the previous graphics visitor you create
    // a GraphicsWorkderFactory that implements AbstractShapeFactory
    // Adding mappings for those implementations of IShape that you are interested
    in.
    //
    public class GraphicsWorkerFactory implements AbstractShapeFactory {

       Graphics graphics_ = null;
```

```
   public GraphicsWorkerFactory(Graphics g) {
      graphics_ = g;
   }

   protected void CreateWorkerMappings() {
      AddMapping(Circle.class, new GraphicCircleWorker(graphics_));
   }
}


//
// Now in your code you could do the following.
//
IShapeWorkerFactory factory = SelectAppropriateFactory();

//
// for each IShape in the heirarchy
//
for(IShape shape : shapeTreeFlattened) {
    IShapeWorker worker = factory.build(shape);
    if(worker != null)
       worker.process(shape);
}
```

It still means that you have to write concrete implementations to work on new versions of 'shape' but because it is completely separated from the interface of shape, you can retrofit this solution without breaking the original interface and software that interacts with it. It acts as a sort of scaffolding around the implementations of IShape.

edited Aug 14 '15 at 14:51

Jean-François Corbett
27.5k 20 100 146

answered Jun 12 '09 at 15:47

Adrian Regan
2,052 11 11

in AbstractWorkerFactory you still have to do instanceof – Asaf Mesika Dec 25 '13 at 7:46

If you're using Java: Yes, it's called `instanceof` . People are overly scared to use it. Compared to the visitor pattern, it's generally faster, more straightforward, and not plagued by point #5.

answered Sep 13 '13 at 18:02

Andy
3,146 4 27 43

faster? Check this. – ntohl Aug 4 '15 at 9:12

@ntohl In tests I've done (on Java 8, note that test used Java 6) instanceof was faster, so I guess the speed of relative speed of the two must vary based upon subtle details. – Andy Aug 4 '15 at 16:59

If you have n `IShape` s and m operations that behave differently for each shape, then you require n*m individual functions. Putting these all in the same class seems like a terrible idea to me, giving you some sort of God object. So they should be grouped either by `IShape` , by putting m functions, one for each operation, in the `IShape` interface, or grouped by operation (by using the visitor pattern), by putting n functions, one for each `IShape` in each operation/visitor class.

You either have to update multiple classes when you add a new `IShape` or when you add a new operation, there is no way around it.

If you are looking for each operation to implement a default `IShape` function, then that would solve your problem, as in Daniel Martin's answer: https://stackoverflow.com/a/986034/1969638, although I would probably use overloading:

```
interface IVisitor
{
    void visit(IShape shape);
    void visit(Rectangle shape);
    void visit(Circle shape);
}

interface IShape
{
    //...
    void accept(IVisitor visitor);
}
```

edited May 23 '17 at 12:02

Community ♦

1 1

answered Jun 26 '15 at 13:57

Zantier

392 3 14