Is there a simple way to emulate Objective-C Categories in C#?

Asked 15 years, 11 months ago Modified 15 years, 11 months ago Viewed 597 times

I have a weird design situation that I've never encountered before... If I were using Objective-C, I would solve it with categories, but I have to use C# 2.0.

First, some background. I have two abstraction layers in this class library. The bottom layer implements a plug-in architecture for components that scan content (sorry, can't be more specific than that). Each plug-in will do its scanning in some unique way, but also the plug-ins can vary by what type of content they accept. I didn't want to expose Generics through the plug-in interface for various reasons not relevant to this discussion. So, I ended up with an IScanner interface and a derived interface for each content type.

The top layer is a convenience wrapper that accepts a composite content format that contains various parts. Different scanners will need different parts of the composite, depending on what content type they are interested in. Therefore, I need to have logic specific to each IScanner-derived interface that parses the composite content, looking for the relevant part that is required.

One way to solve this is to simply add another method to IScanner and implement it in each plug-in. However, the whole point of the two-layer design is so that the plug-ins themselves don't need to know about the composite format. The brute-force way to solve this is by having type-tests and downcasts in the upper layer, but these would need to be carefully maintained as support for new content types is added in the future. The Visitor pattern would also be awkward in this situation because there really is only one Visitor, but the number of different Visitable types will only increase with time (i.e. -- these are the opposite conditions for which Visitor is suitable). Plus, double-dispatch feels like overkill when really all I want is to hijack IScanner's single-dispatch!

If I were using Objective-C, I would just define a category on each IScanner-derived interface and add the parseContent method there. The category would be defined in the upper layer, so the plug-ins wouldn't need to change, while simultaneously avoiding the need for type tests. Unfortunately C# extension methods wouldn't work because they are basically static (i.e. -- tied to the compile-time type of the reference used at the call site, not hooked into dynamic dispatch like Obj-C Categories). Not to mention, I have to use C# 2.0, so extension methods are not even available to me. :-P

So is there a clean and simple way to solve this problem in C#, akin to how it could be solved with Objective-C categories?

EDIT: Some pseudo-code to help make the structure of the current design clear:

```
interface IScanner
{ // Nothing to see here...
}

interface IContentTypeAScanner : IScanner
{
    void ScanTypeA(TypeA content);
}

interface IContentTypeBScanner : IScanner
{
    void ScanTypeB(TypeB content);
}

class CompositeScanner
{
    private readonly IScanner realScanner;

    // C-tor omitted for brevity... It takes an IScanner that was created
    // from an assembly-qualified type name using dynamic type loading.

    // NOTE: Composite is defined outside my code and completely outside my
control.
    public void ScanComposite(Composite c)
    {
        // Solution I would like (imaginary syntax borrowed from Obj-C):
        // [realScanner parseAndScanContentFrom: c];
        // where parseAndScanContentFrom: is defined in a category for each
        // interface derived from IScanner.

        // Solution I am stuck with for now:
        if (realScanner is IContentTypeAScanner)
        {
            (realScanner as
IContentTypeAScanner).ScanTypeA(this.parseTypeA(c));
        }
        else if (realScanner is IContentTypeBScanner)
        {
            (realScanner as
IContentTypeBScanner).ScanTypeB(this.parseTypeB(c));
        }
        else
        {
            throw new SomeKindOfException();
        }
    }

    // Private parsing methods omitted for brevity...
}
```

EDIT: To clarify, I have thought through this design a lot already. I have many reasons, most of which I cannot share, for why it is the way it is. I have not accepted any answers yet because although interesting, they dodge the original question.

The fact is, in Obj-C I could solve this problem simply and elegantly. The question is, can I use the same technique in C# and if so, how? I don't mind looking for alternatives, but to be fair that isn't the question I asked. :)

c# objective-c design-patterns oop visitor-pattern

Share
Improve this question
Follow

edited Jan 8, 2009 at 16:36

asked Jan 7, 2009 at 17:00

Bruce Johnston
8,624 ● 3 ● 34 ● 43

Can you point out a good resource where to learn more about Objective-C categories? – Fabrizio C. Jan 7, 2009 at 17:56

en.wikipedia.org/wiki/Objective-C#Categories dotnetaddict.dotnetdevelopersjournal.com/... developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/... – Bruce Johnston Jan 7, 2009 at 21:15

2 Answers

Sorted by: Highest score (default)

1

It sounds like what you're saying is that you have content laid out something like this:

```
+--------+
| part 1 |
| type A |
+--------+
| part 2 |
| type C |
+--------+
| part 3 |
| type F |
+--------+
| part 4 |
| type D |
+--------+
```

and you have readers for each part type. That is, an AScanner knows how to process the data in a part of type A (such as part 1 above), a BScanner knows how to process the data in a part of type B, and so forth. Am I right so far?

Now, if I understand you, the problem that you're having is that the type readers (the `IScanner` implementations) don't know how to locate the part(s) they recognize within

your composite container.

Can your composite container correctly enumerate the separate parts (i.e., does it know where one part ends and another begins), and if so, does each part have some sort of identification that either the scanner or the container can differentiate?

What I mean is, is the data laid out something like this?

```
+-------------+
| part 1      |
| length: 100 |
| type: "A"   |
| data: ...   |
+-------------+
| part 2      |
| length: 460 |
| type: "C"   |
| data: ...   |
+-------------+
| part 3      |
| length: 26  |
| type: "F"   |
| data: ...   |
+-------------+
| part 4      |
| length: 790 |
| type: "D"   |
| data: ...   |
+-------------+
```

If your data layout is similar to this, could the scanners not request of the container all parts with an identifier matching a given pattern? Something like:

```
class Container : IContainer{
    IEnumerable IContainer.GetParts(string type){
        foreach(IPart part in internalPartsList)
            if(part.TypeID == type)
                yield return part;
    }
}

class AScanner : IScanner{
    void IScanner.ProcessContainer(IContainer c){
        foreach(IPart part in c.GetParts("A"))
            ProcessPart(part);
    }
}
```

Or, if perhaps the container wouldn't be able to recognize a part type, but a scanner would be able to recognize its own part type, maybe something like:

```
delegate void PartCallback(IPart part);
```

```
class Container : IContainer{
    void IContainer.GetParts(PartCallback callback){
        foreach(IPart part in internalPartsList)
            callback(part);
    }
}

class AScanner : IScanner{
    void IScanner.ProcessContainer(IContainer c){
        c.GetParts(delegate(IPart part){
            if(IsTypeA(part))
                ProcessPart(part);
        });
    }

    bool IsTypeA(IPart part){
        // determine if part is of type A
    }
}
```

Perhaps I've misunderstood your content and/or your architecture. If so, please clarify, and I'll update.

Comment from OP:

> 1. The scanners should not have any knowledge of the container type.
> 2. The container type has no built-in intelligence. It is as close to plain old data as you can get in C#.
> 3. I can't change the container type; It is part of an existing architecture.

My responses are too long for comments:

1. The scanners have to have *some* way of retrieving the part(s) which they process. If your concern is that the `IScanner` interface should not be aware of the `IContainer` interface so that you have the freedom to alter the `IContainer` interface in the future, then you could compromise in one of a couple of ways:
    - You could pass the scanners an `IPartProvider` interface that `IContainer` derived from (or contained). This `IPartProvider` would only provide the functionality of serving up parts, so it should be pretty stable, and it could be defined in the same assembly as `IScanner`, so that your plug-ins wouldn't need to reference the assembly where `IContainer` was defined.
    - You could pass a delegate to the scanners that they could use to retrieve the parts. The scanners would then not need knowledge of any interface whatsoever (except `IScanner`, of course), only of the delegate.
2. and

3. It sounds like perhaps you need a surrogate class that knows how to communicate with both the container and the scanners. Any of the functionality mentioned above could be implemented in any ol' class, as long as the container already exposes enough functionality publicly (or protectedly [is that a word?]) that an outside/derived class would be able to access the relevant data.

From your pseudocode in your edited question, it looks like you aren't really gaining any benefit from interfaces, and are tightly coupling your plug-ins to your main app, since each scanner type has a unique derivation of `IScanner` that defines a unique "scan" method and the `CompositeScanner` class has a unique "parse" method for each part type.

I would say that this is your primary problem. You need to decouple the plug-ins—which I assume are the implementors of the `IScanner` interface—from the main app—which I assume is where the `CompositeScanner` class lives. One of my earlier suggestions is how *I* would implement that, but exact details depend on how your `parseType`*X* functions work. Can these be abstracted and generalized?

Presumably, your `parseType`*X* functions communicate with the `Composite` class object to get the data they need. Could these not be moved into a `Parse` method on the `IScanner` interface that proxied through the `CompositeScanner` class to get this data from the `Composite` object? Something like this:

```
delegate byte[] GetDataHandler(int offset, int length);

interface IScanner{
    void   Scan(byte[] data);
    byte[] Parse(GetDataHandler getData);
}

class Composite{
    public byte[] GetData(int offset, int length){/*...*/}
}

class CompositeScanner{}
    IScanner realScanner;

    public void ScanComposite(Composite c){
        realScanner.Scan(realScanner.Parse(delegate(int offset, int length){
            return c.GetData(offset, length);
        });
    }
}
```

Of course, this could be simplified by removing the separate `Parse` method on `IScanner` and simply passing the `GetDataHandler` delegate directly to `Scan` (whose implementation could call a private `Parse`, if desired). The code then looks very similar to my earlier examples.

This design provides as much separation of concerns and decoupling that I can think of.

I just thought of something else that you might find more palatable, and indeed, may provide better separation of concerns.

If you can have each plug-in "register" with the application, you can maybe leave parsing within the application, as long as the plug-in can tell the application how to retrieve its data. Examples are below, but since I don't know how your parts are identified, I've implemented two possibilities—one for indexed parts and one for named parts:

```
// parts identified by their offset within the file
class MainApp{
    struct BlockBounds{
        public int offset;
        public int length;

        public BlockBounds(int offset, int length){
            this.offset = offset;
            this.length = length;
        }
    }

    Dictionary<Type, BlockBounds> plugins = new Dictionary<Type, BlockBounds>
();

    public void RegisterPlugin(Type type, int offset, int length){
        plugins[type] = new BlockBounds(offset, length);
    }

    public void ScanContent(Container c){
        foreach(KeyValuePair<Type, int> pair in plugins)
            ((IScanner)Activator.CreateInstance(pair.Key)).Scan(
                c.GetData(pair.Value.offset, pair.Value.length);
    }
}
```

or

```
// parts identified by name, block length stored within content (as in diagram
above)
class MainApp{
    Dictionary<string, Type> plugins = new Dictionary<string, Type>();

    public void RegisterPlugin(Type type, string partID){
        plugins[partID] = type;
    }

    public void ScanContent(Container c){
        foreach(IPart part in c.GetParts()){
            Type type;
            if(plugins.TryGetValue(part.ID, out type))
```

```
                ((IScanner)Activator.CreateInstance(type)).Scan(part.Data);
            }
        }
    }
```

Obviously, I've extremely simplified these examples, but hopefully, you get the idea. Additionally, rather than using `Activator.CreateInstance`, it would be nice if you could pass a factory (or a factory delegate) to the `RegisterPlugin` method.

Share

Improve this answer

Follow

edited Jan 8, 2009 at 1:50

answered Jan 7, 2009 at 17:35

P Daddy

29.5k ● 9 ● 72 ● 94

There are a few problems with this approach: 1. The scanners should not have any knowledge of the container type. 2. The container type has no built-in intelligence. It is as close to plain old data as you can get in C#. 3. I can't change the container type; It is part of an existing architecture. – Bruce Johnston Jan 7, 2009 at 17:39

IPartProvider is a possibility, although it has other complications I won't go into (internal to my project). The "surrogate class" you mention is the "upper layer" I mentioned in the OP. It is required because scanners and the container are both passive. – Bruce Johnston Jan 7, 2009 at 21:09

This "upper layer" you mention was what I was originally referring to as "IContainer". Perhaps if you show us how you expect the different parts to interact (use bogus names wherever you feel necessary), it will be easier to provide a solution that fits within your model. – P Daddy Jan 7, 2009 at 21:36

You're making an incorrect assumption about the purpose of the IScanner-derived interfaces. They represent scanners for different types of content, but for each content type there will be several unique scanner plug-ins. Also, Composite is ugly and access can't be generalized unfortunately. :-P – Bruce Johnston Jan 8, 2009 at 16:33

1

I'm going to try... ;-) If in your system there is a phase when you populate your "catalog" of `IScanner` objects you could think of decorating your `IScanner`s with an attribute stating which `Part` they are interested in. Then you can map this information and drive the scanning of your `Composite` with the map. This is not a full answer: if I have a bit of time I'll try to elaborate...

Edit: a bit of pseudo-code to support my confused explanation

```
public interface IScanner
{
    void Scan(IPart part);
}

public interface IPart
{
    string ID { get; }
```

```
}

[ScannedPart("your-id-for-A")]
public class AlphaScanner : IScanner
{
    public void Scan(IPart part)
    {
        throw new NotImplementedException();
    }
}

[ScannedPart("your-id-for-B")]
public class BetaScanner : IScanner
{
    public void Scan(IPart part)
    {
        throw new NotImplementedException();
    }
}

public interface IComposite
{
    List<IPart> Parts { get; }
}

public class ScannerDriver
{
    public Dictionary<string, IScanner> Scanners { get; private set; }

    public void DoScan(IComposite composite)
    {
        foreach (IPart part in composite.Parts)
        {
            IScanner scanner = Scanners[part.ID];
            scanner.Scan(part);
        }
    }
}
```

Don't take it as-is: it's for explanation purpose.

Edit: answer to Colonel Kernel comments. I'm glad you find it interesting. :-) In this simple sketch of code reflection should be involved just during the Dictionary initialization (or when needed) and during this phase you can "enforce" the presence of the attribute (or even use other ways of mapping scanners and parts). I say "enforce" because, even if it isn't a compile-time constraint, I think you will run your code *at least once* before putting it into production ;-) so it could be a run-time constraint if needed. I would say that the inspiration is something (very very lightly) similar to MEF or other similar frameworks. Just my 2 cents.

edited Jan 8, 2009 at 8:39

answered Jan 7, 2009 at 17:44

Fabrizio C.

1,564 ● 10 ● 12

I hadn't considered using attributes... However, this doesn't solve the problem. It just converts a type test into a reflection-based attribute check, and it doesn't force scanner classes to include the attribute. Interesting approach though. – Bruce Johnston Jan 7, 2009 at 21:22

See the last "edit" I added for a possible answer to your comment. :-) – Fabrizio C. Jan 8, 2009 at 8:40

In my case, there is no central point of control for registering plug-ins. Their configuration info (including fully-qualified type name) lives in a DB and they are loaded on demand. I can't change this. Therefore, the attribute check would happen too late for my purposes.
– Bruce Johnston Jan 8, 2009 at 16:31

If the configuration is in a DB this could still be ok (metadata, i.e. attribute driven mapping, is just one of the options), but I think I can't be helpful without further details and I don't know if you want to reveal them... ;-) – Fabrizio C. Jan 8, 2009 at 21:24