Abstract Factory Design Pattern

Asked 16 years, 4 months ago Modified 7 years, 3 months ago Viewed 5k times

21

I'm working on an internal project for my company, and part of the project is to be able to parse various "Tasks" from an XML file into a collection of tasks to be ran later.

Because each type of Task has a multitude of different associated fields, I decided it would be best to represent each type of Task with a seperate class.

To do this, I constructed an abstract base class:

```
public abstract class Task
{
    public enum TaskType
    {
        // Types of Tasks
    }

    public abstract TaskType Type
    {
        get;
    }

    public abstract LoadFromXml(XmlElement task);
    public abstract XmlElement CreateXml(XmlDocument currentDoc);
}
```

Each task inherited from this base class, and included the code necessary to create itself from the passed in XmlElement, as well as serialize itself back out to an XmlElement.

A basic example:

```
public class MergeTask : Task
{

    public override TaskType Type
    {
        get { return TaskType.Merge; }
    }

    // Lots of Properties / Methods for this Task

    public MergeTask (XmlElement elem)
    {
        this.LoadFromXml(elem);
    }

    public override LoadFromXml(XmlElement task)
    {
```

```
        // Populates this Task from the Xml.
    }

    public override XmlElement CreateXml(XmlDocument currentDoc)
    {
        // Serializes this class back to xml.
    }
}
```

The parser would then use code similar to this to create a task collection:

```
XmlNode taskNode = parent.SelectNode("tasks");

TaskFactory tf = new TaskFactory();

foreach (XmlNode task in taskNode.ChildNodes)
{
    // Since XmlComments etc will show up
    if (task is XmlElement)
    {
        tasks.Add(tf.CreateTask(task as XmlElement));
    }
}
```

All of this works wonderfully, and allows me to pass tasks around using the base class, while retaining the structure of having individual classes for each task.

However, I am not happy with my code for TaskFactory.CreateTask. This method accepts an XmlElement, and then returns an instance of the appropriate Task class:

```
public Task CreateTask(XmlElement elem)
{
    if (elem != null)
    {
        switch(elem.Name)
        {
            case "merge":
                return new MergeTask(elem);
            default:
                throw new ArgumentException("Invalid Task");
        }
    }
}
```

Because I have to parse the XMLElement, I'm using a huge (10-15 cases in the real code) switch to pick which child class to instantiate. I'm hoping there is some sort of polymorphic trick I can do here to clean up this method.

Any advice?

c# **design-patterns** **factory**

Share

Improve this question

Follow

edited Sep 16, 2008 at 14:37

asked Aug 26, 2008 at 2:35

FlySwat

175k ● 75 ● 248 ● 314

There is a security advantage to checking all the cases. The reflection answer is open to a security flaw where someone injects a Task class (not created by you) into your path and executes it by getting its name into the XML. Harder things have been done by hackers.
– Fuhrmanator Oct 4, 2012 at 18:39

10 Answers

Sorted by: Highest score (default)

12

I use reflection to do this. You can make a factory that basically expands without you having to add any extra code.

make sure you have "using System.Reflection", place the following code in your instantiation method.

```
public Task CreateTask(XmlElement elem)
{
    if (elem != null)
    {
        try
        {
          Assembly a = typeof(Task).Assembly
          string type = string.Format("{0}.
{1}Task",typeof(Task).Namespace,elem.Name);

          //this is only here, so that if that type doesn't exist, this method
          //throws an exception
          Type t = a.GetType(type, true, true);

          return a.CreateInstance(type, true) as Task;
        }
        catch(System.Exception)
        {
          throw new ArgumentException("Invalid Task");
        }
    }
}
```

Another observation, is that you can make this method, a static and hang it off of the Task class, so that you don't have to new up the TaskFactory, and also you get to save yourself a moving piece to maintain.

Share Improve this answer Follow

answered Aug 26, 2008 at 2:45

DevelopingChris

40.7k ● 30 ● 89 ● 119

6

Create a "Prototype" instanace of each class and put them in a hashtable inside the factory , with the string you expect in the XML as the key.

so CreateTask just finds the right Prototype object, by get() ing from the hashtable.

then call LoadFromXML on it.

you have to pre-load the classes into the hashtable,

If you want it more automatic...

You can make the classes "self-registering" by calling a static register method on the factory.

Put calls to register (with constructors) in the static blocks on the Task subclasses. Then all you need to do is "mention" the classes to get the static blocks run.

A static array of Task subclasses would then suffice to "mention" them. Or use reflection to mention the classes.

Share Improve this answer Follow

answered Aug 26, 2008 at 5:00

Tim Williscroft

3,735 ● 25 ● 37

4

How do you feel about Dependency Injection? I use Ninject and the contextual binding support in it would be perfect for this situation. Look at this blog post on how you can use contextual binding with creating controllers with the IControllerFactory when they are requested. This should be a good resource on how to use it for your situation.

Share

Improve this answer

Follow

edited Sep 15, 2017 at 12:02

Vineet Jain

1,565 ● 4 ● 21 ● 32

answered Aug 26, 2008 at 3:19

Dale Ragan

18.3k ● 3 ● 55 ● 71

2

@jholland

> I don't think the Type enum is needed, because I can always do something like this:

Enum?

I admit that it feels hacky. Reflection feels dirty at first, but once you tame the beast you will enjoy what it allows you to do. (Remember recursion, it feels dirty, but its good)

The trick is to realize, you are analyzing meta data, in this case a string provided from xml, and turning it into run-time behavior. That is what reflection is the best at.

BTW: the is operator, is reflection too.

http://en.wikipedia.org/wiki/Reflection_(computer_science)#Uses

Share Improve this answer Follow

answered Aug 26, 2008 at 3:04

DevelopingChris
40.7k ● 30 ● 89 ● 119

2

@Tim, I ended up using a simplified version of your approach and ChanChans, Here is the code:

```
public class TaskFactory
    {
        private Dictionary<String, Type> _taskTypes = new Dictionary<String,
Type>();

        public TaskFactory()
        {
            // Preload the Task Types into a dictionary so we can look them up
later
            foreach (Type type in typeof(TaskFactory).Assembly.GetTypes())
            {
                if (type.IsSubclassOf(typeof(CCTask)))
                {
                    _taskTypes[type.Name.ToLower()] = type;
                }
            }
        }

        public CCTask CreateTask(XmlElement task)
        {
            if (task != null)
            {
                string taskName = task.Name;
                taskName =  taskName.ToLower() + "task";

                // If the Type information is in our Dictionary, instantiate a
new instance of that task
                Type taskType;
                if (_taskTypes.TryGetValue(taskName, out taskType))
                {
                    return (CCTask)Activator.CreateInstance(taskType, task);
                }
                else
                {
                    throw new ArgumentException("Unrecognized Task:" +
task.Name);
```

```
                }
            }
            else
            {
                return null;
            }
        }
    }
```

Share Improve this answer Follow

answered Aug 26, 2008 at 17:17

FlySwat

175k ● 75 ● 248 ● 314

1

@ChanChan

I like the idea of reflection, yet at the same time I've always been shy to use reflection. It's always struck me as a "hack" to work around something that should be easier. I did consider that approach, and then figured a switch statement would be faster for the same amount of code smell.

You did get me thinking, I don't think the Type enum is needed, because I can always do something like this:

```
if (CurrentTask is MergeTask)
{
    // Do Something Specific to MergeTask
}
```

Perhaps I should crack open my GoF Design Patterns book again, but I really thought there was a way to polymorphically instantiate the right class.

Share

Improve this answer

Follow

edited Aug 26, 2008 at 3:02

answered Aug 26, 2008 at 2:56

FlySwat

175k ● 75 ● 248 ● 314

1

> Enum?

I was referring to the Type property and enum in my abstract class.

Reflection it is then! I'll mark you answer as accepted in about 30 minutes, just to give time for anyone else to weigh in. Its a fun topic.

Share Improve this answer Follow

answered Aug 26, 2008 at 3:08

FlySwat
175k • 75 • 248 • 314

1

Thanks for leaving it open, I won't complain. It is a fun topic, I wish you could polymorphicly instantiate.
Even ruby (and its superior meta-programming) has to use its reflection mechanism for this.

Share Improve this answer Follow

answered Aug 26, 2008 at 3:12

DevelopingChris
40.7k • 30 • 89 • 119

1

@Dale

I have not inspected nInject closely, but from my high level understanding of dependency injection, I believe it would be accomplishing the same thing as ChanChans suggestion, only with more layers of cruft (er abstraction).

In a one off situation where I just need it here, I think using some handrolled reflection code is a better approach than having an additional library to link against and only calling it one place...

But maybe I don't understand the advantage nInject would give me here.

Share Improve this answer Follow

answered Aug 26, 2008 at 3:38

FlySwat
175k • 75 • 248 • 314

1

Some frameworks may rely on reflection where needed, but most of the time you use a boot- strapper, if you will, to setup what to do when an instance of an object is needed. This is usually stored in a generic dictionary. I used my own up until recently, when I started using Ninject.

With Ninject, the main thing I liked about it, is that when it does need to use reflection, it doesn't. Instead it takes advantage of the code generation features of .NET which make it incredibly fast. If you feel reflection would be faster in the context you are using, it also allows you to set it up that way.

I know this maybe overkill for what you need at the moment, but I just wanted to point out dependency injection and give you some food for thought for the future. Visit the dojo for a lesson.

Share

Improve this answer

Follow

edited Aug 26, 2008 at 3:58

answered Aug 26, 2008 at 3:48

Dale Ragan
18.3k 3 55 71

Maybe I'm misunderstanding your comment about Ninject not using reflection - but your blog post ends with a whole class that reflects an entire assembly and registers the compatible types (IController). Don't get me wrong - I think I'll use that code, but I'm not sure how else you would automatically register all the types that implement a desired interface.
– Jeff Meatball Yang Feb 12, 2010 at 16:03

You can register the types by hand if you want: Bind(typeof(IController)).To(typeof(HomeController)).Only(When.ContextVariable(“controllerName”).EqualTo("Home")); The blog post just shows how you can do it programatically, since all controller's inherit off of IController. In a ASP.NET MVC application, you are bound to have a lot of controller's and registering each one by hand could be tedious. – Dale Ragan Feb 13, 2010 at 0:05