Attributed Prototyped Class Factories

Wie klont man Fabriken?

Miroslaw Dobrzanski-Neumann
miroslaw@faneu.de
http://www.faneu.de/miroslaw

Attributed Prototyped Class Factories bieten eine elegante Methode die Vorteile der Beiden Entwurfsmuster „Class Factory“ und „Prototype“ zu vereinigen.

Ersten Vortrag zu diesem Artikel habe ich auf dem Treffen der bonn-to-code.net am 16.05.2006 (http://www.bonn-to-code.net) User Treffen presentiert.

Class Factories

Verwendet man Class Factory ist man in der Lage mehrere Implementierungen für eine geschlossene Menge von Objekttypen bereitzustellen. Ein klassisches Beispiel dafür sind GUI Fabriken, die für einem Satz von Controls verschiedenes „Look And Feel“ geben können. Die Elemente werden indirekt durch eine Class Fatory erstellt.

```
class Factory
{
	ProductA CreateProductA ();
	ProductB CreateProductB ();
	...
}
```

Da die erstellung indirekt passiert, kann die Fabrik selbst entscheiden, welche konkreten Produkte erstellt werden. Die Fabrik könnte z.B. folgende Varianten erzeugen:

```
class Factory
{
	ProductA CreateProductA ()
	{
		if (_lookAndFeel == Mac)
			return MacProductA ();
		else
			return WinProductA ();
	}
}
```

Auch wenn die Vorteile dieses Entwurfs unbestritten sind, stößt man schnell an die gesetzten Grenzen. Was passiert wenn wir einen neues Produkt hinzufügen wollen? Was passiert wenn die Produkte entfernt werden sollen? In beiden Fällen müssen alle Fabriken angepasst werden. Ist die Fabrik offen und existieren mehrere Implementierung stellt sich heraus dass solche Umstellung kaum möglich ist.

Prototypen

Ganz anderer Ansatz verfolgt das „Prototype“ Muster. Hier verzichtet man auf die Fabriken. Verfügbare Produkte werden erst zur Laufzeit aufgezählt. Der gleichen Technik bedient sich auch die Plugin Architektur. Die Problematik der Produkterstellung wurde hier in den Aufzählungsprozess verlagert.

```
class Enumerator
{
	void EnumerateAllProducts (Action<Product> acion);
}
```

Braucht man ein neues Produkt von der Sorte ProduktA lässt man einfach das Muster kopieren.

```
	ProduktA prod = musterA.Clone();
```

Ausgelassen dennoch interassant ist hier die Frage wie die Aufzählung stattfindet. Schließlich muss der Enumerator Musterobjekte erstellen. Eine gängige Variante ist die Anwendung des Fabrik Musters. In dem Fall ist der Enumerator selbst eine Fabrik und kennt alle Produkte. Das Problem wurde hier in den Enumerator verlagert, der alles zentralistisch verwaltet.

Andere Variante bedient sich der Serialisierung. Alle Produkte die man vorhalten will werden in der serialisierter Form bereitgestellt. Die Aufgabe des Enumerators ist nun die Produkte zu deserialisieren und an den Aufrufer zurückzugeben. Damit ist der zentralistische Nachteil behoben. Wir angeln uns jedoch andere Probleme. Vor dem Einsatz müssen alle Produkte serialisiert vorliegen. Zum einen erfordert das eines externen Werkzeges, das die Serialisierung vornimmt. Zum anderen können die serialisierten Prototypen recht groß werden. In einzigen Fällen könnte generische (De)Serialisierung gar unmöglich sein.

Die Normalform

Wenn man das Class Factory Muster beobachtet stellt man sich die Frage wieso eigentlich muss eine Fabrik verschiedene Produkte erstellen, das kommt wahrscheinlich von der Namensgebung und der menschlichen Assoziationen, die damit verbunden sind. Wie wäre es wenn man eine Fabrik so enschränkt, dass man sie nur für eine Sorte von Produkten zuständig macht. Das wäre die Vorgehensweise von Intel und anderen hochspezialisierten Produkthersteller. Noch vor kurzer Zeit sagte man: Intel stellt her Prozessoren und Prozessren und Prozessoren und ... Und wenn die Kapazitäten nicht ausreichen wird die Fabrik 1:1 kopiert.

Das ist die Normalform der Objekterzeugung und Interaktion: *Ich bin nur für ein Sache zuständig und kann das sehr gut*.

Die Umkehrung

Geht man so vor verliert man scheinbar, da man nicht nur viele Produkte hat damit aber auch mehrere Fabriken mit im Sack. Beobachtet man die Eigenschaften und Verhalten einer Fabrik und eines Produktes stellt man fest, dass Produkte sich grundsätzlich durch ihre Eigenschaften und Fabriken durch ihres Verhalten auszeichnen. Dieser Unterschied ist sofort bei dem Versuch der Serialisierung sichtbar. Produkte erfordern viel Aufwand um alle ihre Eigenheiten persistent zu halten. Eine Fabrik hat oft nichts zu serialisieren abgesehen von ihrem Datentyp worin auch das Verhalten kodiert ist.

Wir kehren also das „Prototype“ Muster um und lassen anstelle von Produkten die hoch spezialisierten Fabriken serialisieren. Der Enumerator wird dann nicht die Produkte sondern Fabriken aufzählen. Damit ist das Problem der Größe der serialisierten Objekte elegant gelöst.

Serialisierung

Um die Fabriken zu serialisieren bedient man sich eines kleinen Tricks und nutzt man das .NET Framework aus. In den Assemblies kann man auch benutzerdefinierten Metadaten auf vielen Ebenen Speichern und sie später programmatisch abrufen. Das kann man ausnutzen und die Fabriken in Form von Attributen in den Assemblies abzuspeichern. In dem Fall braucht man kein Werkzeug mehr um die Fabriken zu serialisieren. Das erledigt für uns der Compiler für jeder der .NET Sprachen. Die Attribute erfüllen auch die wichtigste Voraussetzung für solche Nutzung. Sie sind vollwertige Klassen und dürfen auch beliebig komplexes Verhalten aufweisen. Die einzige Einschränkung, die .NET Framework an die Attribute stellt ist deren Initialisierung. Wo keine komplexen Objekte überreicht werden dürfen. Da aber die Fabriken kaum Eigenschaften aufweisen passt das genau zu unserem Zweck.

Implementierung (Beispiel)

Zuerst definieren wir Basisklassen für unsere Produkte und dazu Passende Fabriken

```
public abstract class Produkt
{
	protected Produkt()
	{
	}

	public abstract void Perform ();
}

public abstract class FactoryAttribute : Attribute
{
	protected FactoryAttribute (Type t)
	{
```

```
                if (t == null)
                        throw new ArgumentNullException («t»);
                if (!typeof (Produkt).IsAssignableFrom (t)))
                        throw new AgumentException («wrong product type», «t»);
        }
        public abstract Product CreateProduct ();
}
```

Die Fabrik stell sicher, dass ihr die Richtigen Produkttypen überreicht worden sind.

Nun können wird konkrete Produkte und ihre Fabriken erstellen. Fangen wird mit einer Fabrik, die Produkte über ihre Standard Konstruktoren erstellt.

```
[AttributeUsage(AttributeTargets.Assembly, AllowMultiple = true)]
public class StdFactoryAttribute : Attribute
{
        private readonly ConstructorInfo _ci;
        public FactoryAttribute (Type t)
                : base (t)
        {
                _ci = t.GetConstructor (new Type[] {});
                if (_ci == null)
                        throw new ArguentException («needs dflt ctor»);
        }
        public override Product CreateProduct ()
        {
                return (Product)_ci.Invoke (new object[] {});
        }
}
```

Diese Fabrik kann zur Erstellung der Produkte über den Standard Konstruktor verwendet werden.

Und hier noch ein Beispiel:

```
[AttributeUsage(AttributeTargets.Assembly, AllowMultiple = true)]
public class ParamFactoryAttribute : Attribute
{
        private readonly ConstructorInfo _ci;
        private readonly int _param;
        public FactoryAttribute (Type t, int param)
                : base (t)
        {
                _ci = t.GetConstructor (new Type[] {typeof (int)});
                if (_ci == null)
                        throw new ArguentException («needs dflt ctor»);
                _param = param;
        }
        public override Product CreateProduct ()
        {
                return (Product)_ci.Invoke (new object[] {_param});
        }
}
```

Diese Fabrik erstellt Produkte, die einen int Parameter in dem Konstruktor Aufruf erwarten. Auf diese Weise kann man sich dedizierte Fabriken für jede Sorte von Produkten zusammenbauen.

Zu den oben vorgestellten Fabriken definieren wir noch ein paar Produkte

```
public class StdProduktA : Produkt
{
        public Produkt()
        {
        }

        public void Perform ()
        {
                werke werke werke ...
        }
}
public class StdProduktB : Produkt
{
        public Produkt()
        {
        }

        public void Perform ()
        {
                werke werke werke ...
        }
```

```
}
public class ParamProduktC : Produkt
{
	private readonly int _param
	public Produkt(int param)
	{
		_param = param;
	}

	public void Perform ()
	{
		werke werke werke ...
	}
}
```

Nun geschieht der eingentliche Zauber

Wir lassen den Compiler unsere Fabriken in die Metadaten serialisieren:

```
[assembly:StdFactoryAttribute (typeof (StdProduktA))]
[assembly:StdFactoryAttribute (typeof (StdProduktB))]
[assembly:ParamFactoryAttribute (typeof (ParamProduktC),   1)]
[assembly:ParamFactoryAttribute (typeof (ParamProduktC),   6)]
[assembly:ParamFactoryAttribute (typeof (ParamProduktC), 345)]
```

Damit haben wir fünf Fabriken erstellt die alle jeweils für nur ein Produkttyp bzw. dessen Variante verantwortlich sind.

Um die Fabriken aufzuzählen und die Produkte zu erzeugen braucht man keinen eigenen Enumerator. Man bedient sich der Mitteln der .NET Laufzeitumgebung. Wie das geht wird das folgende Beispiel Code zeigen. Das einzige was man dazu braucht der ist richtige Assembly wo die Fabriken serialisiert sind.

```
class Enumerator
{
	void EnumerateAllProducts (Assembly assembly, Action<Product> acion)
	{
		FactoryAttribute[] fas = (FactoryAttribute[]) assembly
			.GetCustomAttributes(typeof (FactoryAttribute), true);
		foreach (FactoryAttribute fa in fas)
		{
			action (fa.CreateProdukt());
		}
	}
}
```

Metafabriken und Bootstraping

Oft besteht die Notwendigkeit die Produkte nach der Erstellung zu modifizieren. Für diese Aufgaben stellt man sinnvollerweise auch Designer bzw. Editoren bereit. Als sehr praktisch erweist sich der Ansatz: Den Editoren auch die Fabrikfunktion zuzuweisen. Die Editoren werden aber oft von Control abgeleitet und zeichnen sich selbst durch viele Eigenschaften aus, die man schlecht billig serialisieren kann. Jedoch alleine die Tatsache, dass man nicht gleichzeitig von Attribute und Control ableiten kann verhinder die Nutzung der oben beschriebenen Technik. Es gibt aber auch eine Lösung für dieses Problem und Sie heißt Metafabrik. Über Fabrikattribute werden (Meta)Fabriken erstellt, deren Aufgabe ist selbst die richtigen Fabriken zu erzeugen. Diesen Prozeß nennt man Bootstraping. In diesem Fall werden nicht die Pordukte Aufgezäht sondern Fabriken, die dann für die Erzeugung und Manipulation der eigentlichen Produkte zuständig sind.

Auf diese Weise hat man sich die Reduktion der Serialisierten Daten durch die Zunahme der Komplexität erkauft.

Alternativem

Solche Nutzung der Attribute könnte man sich ersparen wenn man die Fabriken bzw. Produkte von einer Basisklasse / einem Interface ableiten würde. In diesem Fall würde es reichen alle Klassen in den Assemblies aufzuzählen und schauen welche von der richtigen Basisklasse abgeleitet sind. Natürlich würde das funktionieren aber

- Eine Aufzählung aller Datentypen kann sehr kostspielig sein besonders in sehr ausgewachsenen Assemblies. Die Aufzählung der Attribute ist dagegen enorm günstig
- Es könnte sich herausstellen, dass nicht alle Klassen, die ein Interface implementieren, auch aufgezählt werden sollten. Ein klassisches Beispiel ist hier ein Null Object z.B. NullFactory das zwar von der betroffenen Klasse ableitet aber nur internen Zwecken dient.
- Man könnte alle Klassen, die daran nicht Teilnehmen sollten mit einem Attribut versehen. Das würde aber das Verhalten von Java widerspiegeln: Es ist erlaubt solange es nicht verboten ist. Warum das so schlecht ist könnt ihr in einem anderen Artikel „Warum ich Java nicht mag“ auf meiner Seite nachlesen.

Zusammenfassung

Dediziert man jedem Produkttyp bzw. jeder Produktsorte eine eigene Fabrik, die für die Erzeugung und die Manipulation ihr anvertrauten Produkte spezialisiert ist kann man das „Class Factory“ Entwurfsmuster mit der Eigenschaften des „Prototype“ Muster versehen. Dabei werden die nicht die Produkte serialisiert, was oft aufwendig und kostspielig sein kann, sondern die Fabriken. Dabei nutzt man die wichtigste Eigenheit einer Fabrik: Sie weist fast ausschließlich nur Verhalten aus und keine zu serialisierenden Eigenschaften. Sie ist dann sehr einfach zu serialisieren besonders in einer solchen Umgebung wie .NET wo sich dafür der Attribute bedienen kann.