

Hinter die Mythen geschaut

Drei .NET-Mythen entschleiert

von Christian Gross

In der rationalen Welt der IT besitzen Mythen eine lange Tradition: Verwende keine Eigenschaften, vermeide Vererbung und .NET-Software ist praktisch Open Source, da der IL-Code ungeschützt ist – drei Mythen, die in dieser Kolumne widerlegt werden.

Angenommen, ich erzähle Ihnen eine Geschichte: Vielleicht glauben Sie meinen Erzählungen, weil Sie mich kennen, oder Sie tun es als Lüge ab – oder gerade weil Sie mich kennen, glauben Sie mir nicht. Wenn sich Menschen unterhalten, entstehen Geschichten und die können wahr oder falsch sein. Manchmal sind Geschichten zu dem Zeitpunkt wahr, zu dem sie erzählt werden, stellen sich aber später als falsch heraus. Und bei manchen Geschichten gibt es weder eine Bestätigung, dass sie wahr sind, noch dass sie falsch sind – sie werden aber einfach deshalb zum Faktum, weil gegenteilige Informationen fehlen.

Wussten Sie, dass Eskimo das falsche Wort für die Einwohner im Norden von Kanada, Grönland und Alaska ist? Die

Leute aus dem Norden heißen eigentlich Inuit, denn das ist ihr Eigenname. Kanadier haben den neuen Namen sehr schnell übernommen, weil ein berühmter kanadischer Autor „Eskimo“ mit den indianischen Wörtern „Esser von rohem Fleisch“ verbunden hat. Das hat sich als Mythos herausgestellt, zeigt aber, wie wirksam ein Mythos sein kann, selbst wenn er falsch ist. Dieser Artikel beschäftigt sich nun mit drei .NET-Mythen und zeigt, wie sie sich nicht bewahrheiten.

Keine Eigenschaften verwenden

Kürzlich habe ich ein Buch über Entwurfsmuster gelesen, in dem der Autor Eigenschaften als vermeidbares Übel ansieht. Dieser Autor ist nicht allein – er betrachtet Eigenschaften als nicht objektorientiert, sodass man nach Möglichkeit darauf verzichten sollte. Das ist ein Mythos, weil Eigenschaften wesentlicher Bestandteil der Programmierung sind, an denen man nicht vorbeikommt. Eigenschaften stehen deshalb in einem schlechten Licht, weil sie die internen Implementierungsdetails des Objekts aufdecken. Das folgende Beispiel zeigt eine Demonstration zur Ofentemperaturmessung. Es veranschaulicht, wie der Verzicht auf Eigenschaften aussehen würde, verwendet aber trotzdem Eigenschaften. Im Beispiel soll die Tempe-

ratur eines Ofens überwacht werden. Am einfachsten ließe sich das realisieren, wenn man eine Eigenschaft erzeugt, wie es der Quellcode in Listing 1 zeigt.

Der Typ *Oven* legt die Temperatur als Eigenschaft offen, die eine direkte Referenz auf die Variable *_temperature* ist. Die interne Implementierung der Ofentemperatur ist also direkt an ihre externe Schnittstelle gebunden. Guter objektorientierter Stil vermeidet eine derartige Programmierung. Eine bessere *Oven*-Implementierung könnte wie in Listing 2 aussehen.

In dieser Implementierung von *Oven* ist die interne Variable *_temperature* nicht an ihre externe Schnittstelle gebunden. Die Eigenschaft *Temperature* wird also ersetzt durch die Methoden *SetTemperature*, um

kurz & bündig

Inhalt

Mythen bestimmen auch die Software-Entwicklung

Zusammenfassung

Die in der .NET-Welt verbreiteten „Mythen“, die besagen keine Eigenschaften zu verwenden und Vererbung zu vermeiden, lassen sich mit kleinen Codebeispielen schnell widerlegen. Auch der Mythos, nach dem .NET-Software wegen ihrer Ungeschütztigkeit praktisch Open Source ist, hält einer genaueren Betrachtung nicht Stand

Listing 1

```
class Oven {
    private int _temperature;

    public int Temperature {
        get {
            return _temperature;
        }
        set {
            _temperature = value;
        }
    }
}
```

die Temperatur zuzuweisen, und *AreYouPreHeated*, um anzuzeigen, dass die zugewiesene Temperatur erreicht wurde. Das

ist guter objektorientierter Entwurststil, da sich der Ofen um seine eigenen Verantwortlichkeiten zu kümmern hat. Nun sei

angenommen, dass ein Client die Temperatur benötigt, um statistische Angaben zu berechnen. Stellen Sie sich beispielsweise einen Destillationsvorgang vor, bei dem die Geschwindigkeit der Temperaturerhöhung ein Grad pro Minute nicht überschreiten darf. Ein Entwickler würde eine Temperatureigenschaft verlangen, die aber nicht erforderlich ist, da .NET Delegaten definieren kann. Sehen Sie sich dazu den Beispielcode in Listing 3 an.

Mithilfe von Delegaten wird die Temperatur des Ofens durch Typen verfolgt, die zur Klasse *Oven* extern sind. Die objektorientierte Entwurfsregel, wonach ein Typ für seine eigenen Daten verantwortlich ist, wird nicht verletzt. Der Delegat *OnTemperature* verteilt den Temperaturwert des Ofens an alle *Listener*, die daran interessiert sind. Die Methode *AddTem-*

Listing 2

```
class Oven {
    private int _temperature;

    public void SetTemperature( int temperature) {
        _temperature = temperature;
    }
    public bool AreYouPreHeated() {
        return false;
    }
}
```

Listing 3

```
delegate void OnTemperature( int temperature);

class Oven {
    private int _temperature;
    OnTemperature _listeners;

    public void BroadcastTemperature() {
        _listeners( _temperature);
    }
    public void AddTemperatureListener
        ( OnTemperature listener) {
        _listeners += listener;
    }
}
class Controller {
    public Controller( Oven oven) {
        oven.AddTemperatureListener
        new OnTemperature( this.OnTemperature);
    }
    public void OnTemperature( int temperature) {
        Console.WriteLine( "Temperature (" + temperature
            + ")");
    }
}
```

Listing 4

```
class Stack extends ArrayList {
    private int topOfStack=0;

    public void push( Object article) {
        add( topOfStack++, article);
    }
    public Object pop() {
        return remove( --topOfStack);
    }
    public void pushMany( Object[] articles) {
        for( int i=0; i<articles.Length; ++i) {
            push( articles[ i]);
        }
    }
}
```

Listing 6

```
class Stack extends ArrayList {
    private int topOfStack=0;

    public void push( Object article) {
        add( topOfStack++, article);
    }
    public Object pop() {
        return remove( --topOfStack);
    }
    public void pushMany( Object[] articles) {
        for( int i=0; i<articles.Length; ++i) {
            add( articles[ i]);
        }
    }
}
```

Listing 7

```
class Stack : ArrayList {
    private int topOfStack = 0;

    public void Push( Object article) {
        this.Add( article);
        topOfStack++;
    }
    public Object Pop() {
        Object retval = this[ --topOfStack];
        this.RemoveAt( topOfStack);
        return retval;
    }
    public void PushMany( Object[] articles) {
        foreach( Object item in articles) {
            Push( item);
        }
    }
}

class MonitorableStack : Stack {
    public int highWaterMark = 0;
    public int lowWaterMark = 0;

    public new void Push( Object item) {
        base.Push( item);
        if( this.Count > highWaterMark) {
            highWaterMark = this.Count;
        }
    }
    public new Object Pop() {
        Object popped = base.Pop();
        if( this.Count < lowWaterMark) {
            lowWaterMark = this.Count;
        }
        return popped;
    }
}
```

Listing 8

```
class Stack : ArrayList {
    private int topOfStack=0;

    public virtual void Push( Object article) {
        this.Add( article);
        topOfStack++;
    }
    public virtual Object Pop() {
        Object retval = this[ --topOfStack];
        this.RemoveAt( topOfStack);
        return retval;
    }
    public void PushMany( Object[] articles) {
        foreach( Object item in articles) {
            Push( item);
        }
    }
}

class MonitorableStack : Stack {
    public int highWaterMark=0;
    public int lowWaterMark=0;

    public override void Push( Object item) {
        base.Push( item);
        if( this.Count > highWaterMark) {
            highWaterMark = this.Count;
        }
    }
    public override Object Pop() {
        Object popped = base.Pop();
        if( this.Count < lowWaterMark) {
            lowWaterMark = this.Count;
        }
        return popped;
    }
}
```

peratureListener fügt in die Liste der *Listener* eine Delegateninstanz ein. Die Methode *BroadcastTemperature* sendet die Temperatur als Rundruf. Es ist wichtig zu erkennen, dass die Methode *BroadcastTemperature* optional und ein Implementierungsdetail ist. Vielleicht läuft die Klasse *Oven* in ihrem eigenen Thread und meldet periodisch eine Temperatur, vielleicht ist eine andere Klasse für die ständige Abfrage des Ofens zuständig – in jedem Fall verwendet die Klasse *Controller* die Methode *Oven.AddTemperatureListener*, um die Ofentemperatur zu empfangen.

Die Lösung mit einem Delegaten ist elegant, weil die Klassen *Oven* und *Controller* locker gekoppelt sind und die Klasse *Controller* die Implementierung der Klasse *Oven* nicht kennen muss. Der Mythos, keine Eigenschaften zu verwenden, scheint demnach eine Tatsache zu sein. Das Problem dabei ist, dass sich der Eigenschaftsmythos immer auf die konkrete Erscheinung und nicht auf das Gesamtbild bezieht. In diesem Beispiel ist es nicht notwendig, eine Eigenschaft zu verwenden (selbst wenn es einfacher gewesen wäre), da das die Entwurfsprinzipien guter Objektorientierung verletzen würde. Diese Veranschaulichung ignoriert von vornherein, woher die Temperatur gekommen ist. Überlegen Sie einmal, wo, wann und wie *Oven* die Variable *_temperature* zuweist. Woher kommt diese Temperatur? Das ist nicht definiert, stellt aber ein wichtiges Detail dar, weil die Anwendung andernfalls nicht funktioniert.

Die Klasse *Oven* muss ihre Temperatur von irgendwoher beziehen – höchstwahrscheinlich von einem Gerätetreiber, der als Schnittstelle zum physischen Ofen fungiert. In den meisten Fällen ist diese Temperatur eine Eigenschaft, die von der *Oven*-Klasse mithilfe eines Polling-Mechanismus abgefragt wird. Mit anderen Worten verbirgt die Klasse *Oven* die Tatsache, dass irgendwo in den von *Oven* verwendeten Typen eine Eigenschaft *Temperatur* existiert. Selbst wenn der Gerätetreiber mit Delegaten arbeitet, muss in der Implementierung des Gerätetreibers ein Typ definiert werden, der eine Eigenschaft *aktuelle Temperatur* besitzt.

Ein zynisch veranlagter Leser könnte sagen, die Objekt-Puristen fügen Komplexitätsebenen hinzu, wo sie nicht notwendig sind. Das stimmt nicht ganz, weil eine

sorglose Verwendung von Eigenschaften die objektorientierten Entwurfsregeln verletzt. Der Leser sollte erkennen, dass Eigenschaften fundamentale Entwurfselemente verkörpern. Sie sollten nur offen gelegt werden, wenn sie einen fundamentalen Bestandteil des Entwurfs repräsentieren, wie zum Beispiel die Temperatur des Ofens. Nachdem ich aber diesen Entwurf ohne Verwendung einer Eigenschaft durchgespielt habe, stehe ich zu dem Ent-

Die Lösung hängt vom Kontext ab

wurf – allerdings würde ich nicht sagen, dass man unbedingt diesen Entwurf verwenden muss und nicht die Eigenschaft *Temperatur*. Die Lösung, die Sie letztlich umsetzen, hängt von Ihrem Kontext ab. Möglicherweise ist die Verwendung der Eigenschaft *Temperatur* einfacher.

Keine Vererbung verwenden

Es gibt einen Mythos, nach dem man Vererbung mit äußerster Vorsicht nutzen und nur mit Schnittstellen arbeiten sollte. Dieser Mythos hat seine Wurzeln im Problem der fragilen Basisklassen. Der Java-Quellcode in Listing 4 soll dies verständlich machen. (Als Beispiel wurde Java gewählt, weil es C# ähnlich ist und die Erläuterung vereinfacht.)

Die Klasse *Stack* legt die drei Methoden *push*, *pop* und *pushMany* offen. Wichtig in diesem Beispiel ist, wie die Methode *pushMany* implementiert ist. Die Schleife, die die einzelnen Elemente des *articles*-Arrays durchläuft, addiert mithilfe der *push*-Methode. Die Implementierung von *pushMany* ist logisch – doch sehen Sie sich jetzt den Quellcode in Listing 5 an.

Die Klasse *MonitorableStack* ist eine Subklasse des Typs *Stack*. Sie soll die Anzahl der Operationen verfolgen, die Elemente in den Stack einfügen und aus dem Stack entfernen. Dazu werden die Methoden *pop* und *push* überladen. Bei Aufrufen der Methoden *push*, *pop* und *pushMany* werden dann die Wasserstände gezählt. Der folgende Clientcode veranschaulicht das:

```
Stack cls = new MonitorableStack();
cls.pushMany( new Object[] { 1, 2});
```

Die Variable *cls* ist eine Typinstanz von *MonitorableStack*, allerdings in den Typ *Stack* umgewandelt. Beim Aufruf der Methoden *pop* und *push* werden die überladenen Methoden von *MonitorableStack* aufgerufen. Sehen Sie sich nun die Änderung an der Basisklasse *Stack* in Listing 6 an.

In dieser Variante der Klasse *Stack* ruft die Methode *pushMany* die Methode *push* nicht auf. Diese Änderung könnte aufgrund bestimmter Anforderungen, von Leistungsbetrachtungen oder aus einem anderen Grund erfolgt sein. Diese Basisklassenänderung führt dazu, dass die Klasse *MonitorableStack* nicht mehr richtig funktioniert. Mit dem zuvor angegebenen Clientcode wird die überladene Methode *push* nicht aufgerufen und die Wasserstände werden demnach nicht modifiziert. Dies zeigt das Wesen des Problems der fragilen Basisklassen und warum Vererbung nicht verwendet werden sollte: Durch Änderung der Basisklasse auf höherer Ebene wird die Funktionalität beeinflusst – ein Mythos ist das deshalb, weil .NET dieses Problem gar nicht kennt. Bei anderen Umgebungen ergibt sich das Problem daraus, dass sie nicht zwischen Methoden, die sich überladen lassen, und Methoden, die nicht überladen werden können, unterscheiden. .NET verlangt eine explizite Absichtsdefinition, wie es der Quellcode in Listing 7 für die gleiche Implementierung veranschaulicht.

Beachten Sie, wie in der .NET-Implementierung die Methoden *Push* und *Pop* der Klasse *MonitorableStack* das Attribut *new* verwenden müssen. Das Attribut *new* zeigt an, dass die Funktionalität der Basisklasse ignoriert und eine neue Funktionalität definiert werden soll. Betrachten Sie nun denselben Java-Clientcode in Verbindung mit der .NET-Implementierung:

```
Stack cls = new MonitorableStack();
cls.PushMany( new Object[] { 1, 2});
```

Die .NET-Implementierung funktioniert nicht wie die Java-Implementierung – das hängt mit der Art und Weise zusammen, wie die Methoden *Push* und *Pop* implementiert sind. Wenn in .NET der Typ der Instanz *cls* in die Basisklasse *Stack* umgewandelt wird, führt der Aufruf von *Push* oder *Pop* zum Aufruf der *Push*- bzw. *Pop*-Implementierungen der Basisklasse. Die

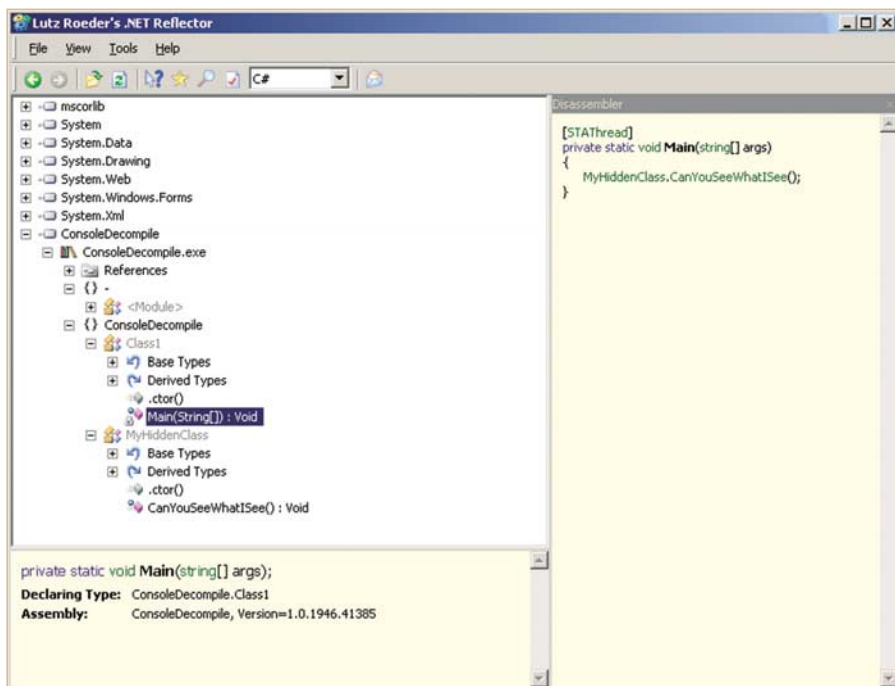


Abb. 1: Assembly mit Reflector disassembliert

Methoden *Push* und *Pop* sind nicht überladen. Beim Aufruf von *PushMany* wird also die *Stack*-Implementierung von *Push* und nicht *MonitorableStack.Push* aufgerufen. Wenn nun der Autor der Basisklasse *Stack* seine Implementierung ändert, wirkt sich das nicht auf die Klasse *MonitorableStack* aus – das Problem der fragilen Basisklassen besteht demnach hier nicht.

Diese Lösung ist natürlich nicht ideal und beseitigt das Problem auch nicht wie in der Java-Version. Ursprünglich wollte ich anhand dieser Lösung demonstrieren,

dass .NET dieses Problem gelöst hat. Um das eigentliche Problem aus der Welt zu schaffen, wäre der Code in Listing 8 geeignet.

Diese Version verwendet die Schlüsselwörter *virtual* und *override* und liefert das gleiche Ergebnis wie die Java-Lösung. Dem aufmerksamen Leser dürfte nicht entgangen sein, dass das Problem der fragilen Basisklassen dennoch existiert, denn wenn man die Methode *Stack.PushMany* dahingehend modifiziert, dass sie ohne die *Push*-Methode auskommt, funktioniert die Klasse *MonitorableStack* nicht mehr korrekt. Der Unterschied zwischen Java und .NET besteht darin, dass das zusätzliche Schlüsselwort *virtual* in .NET aussagt: „Wenn Sie überschreiben wollen, dann nur zu.“ Ändert dann ein Entwickler die Funktionalität der Basisklasse, tritt das Problem wieder zutage, obwohl .NET das Subclassing explizit erlaubt und unterstützt. Mit anderen Worten befindet sich der Entwickler im Unrecht und hat einen Bug eingeführt.

Verwalteter Code (Java, Python, .NET) ist wie Open Source

Viele Leute glauben, dass eine mit .NET geschriebene Anwendung in die Kategorie Open Source fallen muss, weil sich der Code sehr leicht dekompile lässt. Sehen Sie sich zum Beispiel den Quellcode

in Listing 9 an, der gleich kompiliert wird.

Beachten Sie, dass die Klasse *MyHiddenClass* als *internal* markiert ist. Demnach sollte kein Außenstehender in der Lage sein, sich eine Klasse der Assembly anzusehen. Stellen Sie sich beispielsweise vor, dass der Typ *MyHiddenClass* die Logik für einen Lizenzschlüssel enthält, um ein Lizenzierungsschema durchzusetzen. Werfen Sie nun einen Blick auf dieselbe Assembly, die mit dem Dienstprogramm *Reflector* von Lutz Roeder disassembliert wurde (siehe Abbildung 1).

Beachten Sie, wie das Utility Reflector alles disassembliert hat, und zwar einschließlich der „verborgenen“ Klasse *MyHiddenClass*. Mit anderen Worten verhält sich .NET wie Open Source, weil man ohne weiteres alles inspizieren kann, was damit kompiliert worden ist. Der Grund für den Open-Source-Mythos von .NET liegt darin, dass sich alle kompilierten Quellen ohne weiteres disassemblieren lassen. Entstanden ist dieser Mythos, weil sich .NET wesentlich einfacher disassemblieren lässt als die Bytecodes von Java oder Python. Man ist eben besorgt darüber, dass das Reverse Engineering oder Disassemblieren so leicht ist, weil Dritte dann sehen können, wie ein Programm arbeitet. Stellen Sie sich vor, der disassemblierte Code wäre eine deaktivierte Lizenzierungsroutine – Software-Piraten könnten eine Anwendung knacken und dem Entwickler entgeht das Honorar für seine Anstrengungen. Und an diesem Punkt erweist sich der Mythos als falsch. Sprachen wie C oder C++, aber auch Dongles schützen keine Anwendungen, selbst wenn viele glauben, dass es so wäre. Heutzutage lassen sich Anwendungen leicht knacken und viele Shareware-Entwickler verlieren durch Software-Piraterie rund 95% der kalkulierten Nutzungsgebühren. Man sollte daran denken, dass der Lizenzierungscode und der Codeschutz nur so stark wie das schwächste Glied in der Kette sind.

Normale .NET-Assemblies könnte man als Open Source auffassen, da es ziemlich trivial ist, die Anwendung zu disassemblieren. Aber mithilfe von Verfahren wie Verschleierung (Obfuscation) ändert sich die Crackbarkeit der Anwendung. Verschleierung ist eine interessante Technik, weil verschleierter Code schwerer zu entschlüsseln ist. Diese Technik wirkt sich

Listing 9

```
namespace ConsoleDecompile
{
    internal class MyHiddenClass {
        public static void CanYouSeeWhatISee() {
            Console.WriteLine("hohum");
        }
    }

    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            MyHiddenClass.CanYouSeeWhatISee();
        }
    }
}
```

aber nicht auf die Funktionsweise eines Programms aus, sondern ändert den Code so, dass er wesentlich schwerer zu verstehen ist. Ein verschleiertes Programm ist wie ein Puzzle einer einzigen Farbe, das in die Luft geworfen wurde. Man sieht zwar die einzelnen Puzzle-Teile, kann sie aber kaum wieder zusammensetzen, weil das Bild fehlt, nach dem sich die Puzzle-Teile zuordnen lassen. Sehen Sie sich Abbildung 2 an, die eine verschleierte Version des zuvor dargestellten Codes wiedergibt.

Dem Bild nach zu urteilen besitzt die verschleierte Version der Anwendung mehrere Typen: *a*, *b*, *<Module>* und *Dotfuscator Attribute*. Die Methode *<Module>.a(String, Int32)* lässt sich in Reflector nicht dekompile und ist mit dem Hinweis versehen, dass eine Verschleierung vorliegt. Die anderen Typen *a* und *b* haben Methoden, die mit *a* bezeichnet sind. Sie lassen sich zwar dekompile, verweisen aber auf andere Typen. Die Zeichenfolge *hobum* ist verschlüsselt worden und lässt sich im dekompliierten Code nicht finden. Praktisch ist die Anwendung durch die Verschleierung kaum noch verständlich. Dennoch lässt sich dieses Beispiel wenigstens ansatzweise verstehen, weil es nur zwei Typen gibt, *MyHiddenClass* und *Class1*. Bei einer „richtigen“ Anwendung mit Hunderten oder Tausenden von Typen und Tausenden Methoden wird das Entziffern des verschleierten Codes äußerst schwierig.

Das bedeutet nicht, dass niemand herausfinden kann, was die Anwendung tut, es gibt aber mehrere Schutzebenen. Ein Verschleierer ...

- verschleiert alle Bezeichner (Typen, Methoden, Eigenschaften usw.),
- entfernt alle unnötigen Symbole,
- entfernt alle unnötigen Metadaten,
- verschlüsselt literale Zeichenfolgen,
- verschleiert mehrere Assemblys und
- stoppt Reverse-Engineering-Tools wie Reflector oder ILDasm.

Es sind unterschiedliche Verschleierer verfügbar, aber der von mir Empfohlene ist *Dotfuscator* von *Preemptive Solutions* (vorgestellt im *dot.net magazin* 06.2005). Zugegebenermaßen bin ich mit einem der Hauptentwickler von *Preemptive* befreundet und weiß, wie engagiert die Firma ist, um den besten Verschleierungsschutz zu

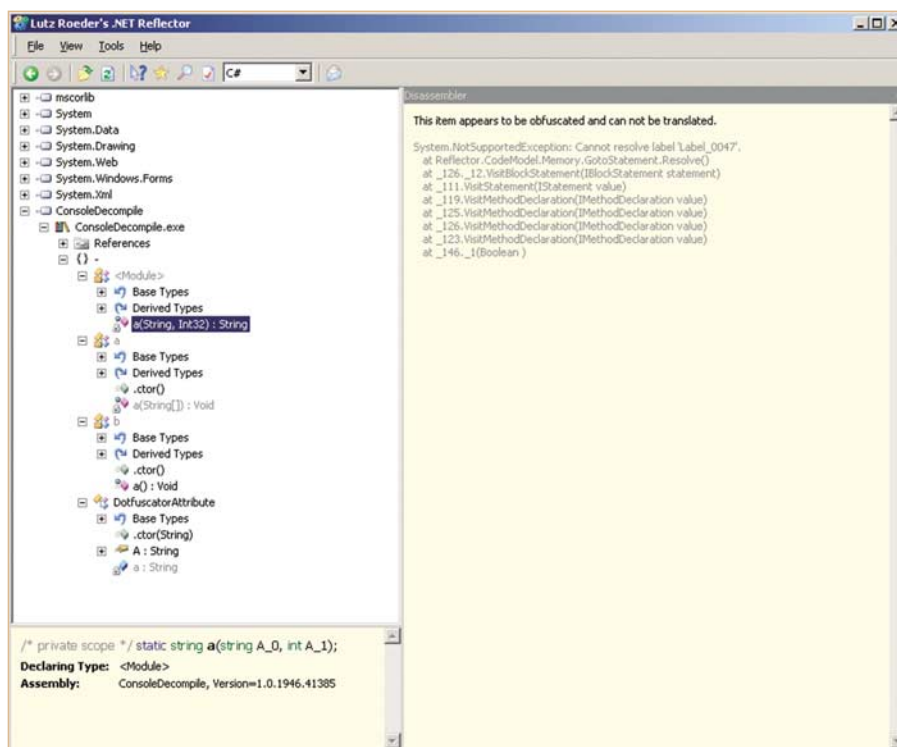


Abb. 2: Verschleierte Version des Codes

entwickeln, der Ihre Investitionen schützt. Wenn Sie mit Verschleierung experimentieren möchten, können Sie die Community-Edition verwenden, die mit Visual Studio vertrieben wird. Verschleierung ist immerhin ein erster Schritt zu einer dauerhaften Lösung, um Ihr geistiges Eigentum zu schützen. Die meisten Leute kennen mich als Open-Source-Vertreter und darauf bin ich auch stolz. Viele Firmen sind allerdings kommerzielle Softwareanbieter. Diese Firmen sollten jedes Recht haben, ihre Software zu verkaufen, ohne dass ihre berechtigten Ansprüche von Piraten hintergangen werden. Denken Sie daran, dass Open Source und Closed Source das Copyright als Legitimation zum Vertrieb von Software verwenden. Piraten, die glauben, dass alle Software kostenlos sein müsse, sind keine Open-Source-Leute. Sogar *Richard Stallman*, bekannt als extremer Verfechter von Free Software und der GPL würde niemals die Aktionen von Piraten billigen.

Zu guter Letzt

Dieser Artikel hat versucht, einige Mythen zu entschleiern – es gibt aber noch eine ganze Menge anderer. Für diejenigen Leser, die sich über meinen Java-Ausflug wundern: Bitte betrachten Sie diesen Verweis

nicht als zynische Respektlosigkeit gegenüber Java – ich mag Java und schreibe Code in Java. Der Mangel, an dem Java leidet, ist auch in Programmiersprachen wie C++ präsent. Der Artikel sollte zeigen, dass .NET viele allgemeine objektorientierte Probleme gelöst hat. .NET ist eine wohlgedachte Umgebung, mit der sich heute alle Arten von Problemen kodieren lassen. In der Tat kann ich nicht verstehen, warum wir nicht alle in einer verwalteten Umgebung wie .NET oder Java kodieren. Sicherlich ist C oder C++ immer noch in bestimmten Kontexten notwendig, doch alles in allem leben wir in einer verwalteten Welt.

Wenn Sie fragen haben, schicken Sie mir bitte eine E-Mail – allerdings habe ich dieses Mal eine zusätzliche Bitte, da ich auf einem Mythen-Trip bin: Senden Sie mir bitte einen Mythos (der sich auf die Software-Entwicklung bezieht), zu dem mein Kommentar gefragt ist. ●

Christian Gross ist ein sehr erfahrener Trainer, der sich für alle Aspekte des Software Engineering interessiert, die mit dem Internet, XML, Java, Apache oder .NET zusammenhängen. Sie erreichen ihn unter contact@devspace.com oder www.devspace.com/Wikka/wikka.php?wakka=BloggerJacksHomePage.