Lizenz zum Aufpassen.

Regressionstests mit dem VS 2010 unter Zuhilfenahme von Rhino Mocks

Norbert Eggert und Mike Wiedemann

protected-networks.com

8MAN. Und Berechtigungen bleiben sauber.

protecting companies from the inside out

Haftungsausschluss

Die in diesem Handbuch gemachten Angaben können sich jederzeit ohne vorherige Ankündigung ändern und gelten als nicht rechtsverbindlich.
Die beschriebene Software **8MAN** wird von protected-networks.com im Rahmen einer Nutzungsvereinbarung zur Verfügung gestellt und darf nur in Übereinstimmung mit dieser Vereinbarung eingesetzt werden.
Dieses Dokument darf ohne die vorherige schriftliche Erlaubnis von protected-networks.com weder ganz noch teilweise in irgendeiner Form reproduziert, übermittelt oder übersetzt werden, sei es elektronisch, mechanisch, manuell oder optisch.
Dieses Dokument ist in einer Einheit zu denen auf der Website von protected-networks.com veröffentlichten rechtlichen Hinweisen AGB, EULA und der Datenschutzerklärung zu sehen.

Urheberrecht

8MAN ist eine geschützte Bezeichnung für ein Programm und die entsprechenden Dokumente, dessen Urheberrechte bei protected-networks.com GmbH liegen.
Marken und geschäftliche Bezeichnungen sind – auch ohne besondere Kennzeichnung – Eigentum des jeweiligen Markeninhabers.

protected-networks.com GmbH
Alt-Moabit 73
10555 Berlin
Tel.: +49 (30) 390 63 45 - 0
Web: www.protected-networks.com

Hot-Line
Support: susi.support@protected-networks.com
Tel.-Support: +49 (30) 390 63 45 - 0
Zu den üblichen Bürozeiten

Veröffentlichung: Juli 2010

protected-networks.com

Regressionstests mit dem VS 2010 unter Zuhilfenahme von Rhino Mocks

Der Gegenstand

Eine Hauptproblemstellung beim Programmieren von komplexen Algorithmen ist nicht nur die Gewährleistung der Fehlerfreiheit bei der Ersterstellung der Software, sondern das Sicherstellen der Funktionalität beim Anpassen, Modifizieren und natürlich auch Bugfixing der Algorithmen. Durch fortwährende Änderungs-, Anpassungs- und Feature-Wünsche von Kunden, Vertrieb und Marketing ist die Halbwertzeit einer SW-Version kürzer, als das Leben einer Eintagsfliege. Nüchterner betrachtet ist das jedoch nur „Business as usual". Aus dieser Situation heraus entsteht für die Entwicklungsabteilung eine simple Notwendigkeit – die bestehenden Funktionalitäten zu gewährleisten bei gleichzeitiger Erweiterung von Funktionalitäten. Diese Aufgabe wird im Regelfall umso schwieriger, je mehr Software-Lebenszeit vergangen ist.
Nachfolgend wollen wir eine mögliche Herangehensweise vorstellen, um einen komplexen Algorithmus von der Peripherie zu isolieren und ihm zusätzliche Funktionalität zu geben, die es erlaubt, während der Ausführungszeit die peripheren Bedingungen mitzuschneiden. Wir wollen das mit Hilfe des recht weit verbreiteten Frameworks *Rhino Mocks* [1] in Verbindung mit den Unit Tests von Microsoft (MSTest) im Visual Studio 2010 erreichen. Die zugrunde liegende Programmiersprache ist C#.

Das Ziel

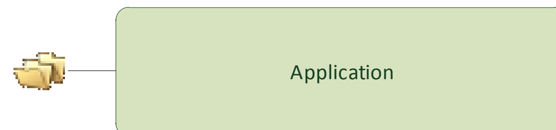
Zielstellung unserer Aktivitäten ist erhöhte Verfügbarkeit der zum Einsatz kommenden Algorithmen. Zu diesem Zweck lassen wir uns die entsprechende Anzahl von Unit-Test-Cases von unserer Software selbst erzeugen. Nach einer genauen manuellen Prüfung der korrekten Funktionalität unseres Algorithmus wird der Test-Case in das Unit-Test-Framework integriert. Durch diese Technik wird die Code-Abdeckung kontinuierlich erhöht. Die Verwendung des Rhino Mocks ermöglicht die weitere Erhöhung der Abdeckung um die Peripheriebedingungen, die meist nur theoretisch eintreten können (z.B. OutOfMemory-Exception). Eingehende Fehler von Kunden werden nach Korrektur als neuer Test zusätzlich in das Unit-Test-Framework überführt.

Der Algorithmus

Für das Beispiel verwenden wir einen Algorithmus, der die Aufgabe hat, Zugriffsrechte auf einen Ordner in einem Dateisystem anzupassen. Ausgangspunkt für den Algorithmus sind die bestehenden Zugriffsrechte des Ordners und vorgegebene zusätzliche Optionen für die Änderungen bzw. die gewünschten neuen Zugriffsrechte für den Ordner. Dies hört sich auf den ersten Blick banal an. Bei genauerer Betrachtung der Variationsmöglichkeiten innerhalb des Algorithmus wird aber schnell deutlich, dass bezüglich der Testabdeckung eine erhebliche Anzahl an einzelnen Testfällen notwendig ist, um eine geeignete Qualität der Software sicherzustellen bzw. aufrechtzuerhalten.

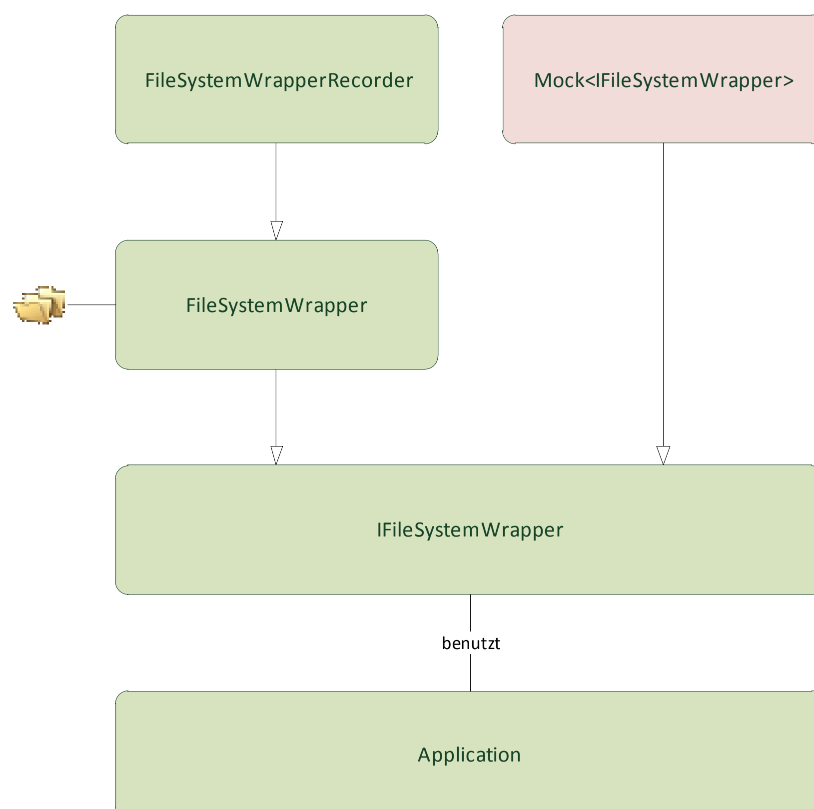
Aufgabe – Isolation von der Peripherie

Die grundsätzliche Problematik beim Testen von „hardwareabhängigen" Algorithmen ist das Vorhandensein der benötigten Hardware. Denkt man an eine Einbindung der Tests im Anschluss an einen Daily-Build-Zyklus, dann muss eine stabile Umgebung mit immer gleichen Ausgangsbedingungen zur Verfügung stehen. Um diese strengen Kriterien zu umgehen, wäre es vorteilhaft, wenn man die hardwarenahen Aufrufe stattdessen simulieren könnte, denn Gegenstand des Tests ist nicht die Peripherie sondern der Algorithmus selbst. Solch eine Simulation kann das Framework „Rhino Mocks" (engl. Attrappe) übernehmen.



Betrachtet man die grundsätzlichen Fähigkeiten von Mocks im Zusammenhang mit dem Design seiner ganz privaten Klassen, dann könnte man unter Umständen feststellen, was man verbessern kann; die Gewährleistung einer stärkeren Entkopplung von Abhängigkeiten zum Betriebssystem. Interfaces sind ein ideales Werkzeug, um einerseits besagte Entkopplung zu erreichen und andererseits eine Simulation durch ein Mock-Framework zu ermöglichen.

Sollte man das in seinem Design nicht berücksichtigt haben, dann ist der Moment gekommen, in dem man das Nützliche mit dem Angenehmen verbinden kann. Die Pflichtaufgabe lautet: Schaffe dir ein Wrapper-Interface, welches das Dateisystem, das Active Directory oder auch jede andere Ressource (z.B. Datenbank-Tabellen) von der unmittelbaren Benutzung abkoppelt. Die Kür, die wir zusätzlich erbringen wollen, ist eine Funktionalität, die in der Lage ist, das Verhalten der Interfaces für spätere Zwecke aufzuzeichnen.



Was wollen wir mit der Aufzeichnung erreichen?

Die Mitschnittfunktion soll es ermöglichen, einen kompletten Unit-Test-Case bei der regulären Benutzung des Algorithmus aufzuzeichnen, in eine Datei zu schreiben und diesen Test-Case dann unabhängig von der nötigen Peripherie ausführen zu können. Der Test-Case soll anschließend einfach in das Unit Test Projekt übernommen werden können und unmittelbar übersetzbar und ausführbar sein und zwar ohne Anwesenheit der verwendeten Peripherie.

Weiterhin soll es optional möglich sein, und das ist die Herausforderung, ein Problem im echten „Kundenumfeld" aufzuzeichnen und für die Fehleranalyse heranzuziehen, um eine schnelle Lösung für die Korrektur zu finden. Ein in der Entwicklungsabteilung im Debugger nachgestellter Fehler hat eine um Dimensionen kürzere Lebensdauer, als sein verbal beschriebenes Pendant in der E-Mail.

Ein typisches Beispiel ohne Benutzung eines Interfaces:

```
public class FileSystemSecurity
{
    private readonly string path;

    public FileSystemSecurity(string path)
    {
        this.path = path;
    }

    public void ChangeOwner(string owner)
    {
        DirectoryInfo info = new DirectoryInfo(this.path);
        DirectorySecurity sec = info.GetAccessControl();
        NTAccount account = new NTAccount(owner);
        sec.SetOwner(account);
        info.SetAccessControl(sec);
    }
}
```

Der dazu passende Unit Test könnte unter anderem so aussehen:

```
[TestClass()]
public class FileSystemSecurityTest
{…
    [TestMethod()]
    public void ChangeOwnerTest1()
    {
        string path = @"\\server\share\a\b\c";
        string account = "Administratoren";
        NTAccount expected = new NTAccount(account);
        FileSystemSecurity fss = new FileSystemSecurity(path);
        fss.ChangeOwner(account);
        Assert.AreEqual(account, getOwner(path).Value);
    }

    private static NTAccount getOwner(string path)
    {
        DirectoryInfo info = new DirectoryInfo(path);
        DirectorySecurity sec = info.GetAccessControl();
        return (NTAccount)sec.GetOwner(typeof(NTAccount));
    }
}
```

Zum Zwecke der Anschauung wurde hier bei diesem Beispiel auf Sicherheitsabfragen verzichtet. Schon Anhand dieses kleinen Beispiels kann man die Vielfalt erahnen, die zu testen wäre.

- Verhalten der Klasse, wenn der Parameter „path“ eine *null*-Referenz ist
- Verhalten der Klasse, wenn der Parameter „account“ eine *null*-Referenz ist
- Verhalten, wenn die nötigen Rechte zum Ändern nicht existieren
- Verhalten, wenn das Verzeichnis nicht existiert

Und gerade im letzten Punkt liegt die Schwäche dieses Unit-Tests. Der Algorithmus für *setOwner* ist darauf angewiesen, dass es dieses Verzeichnis gibt und auch darauf zugegriffen werden kann. Um sowohl die Existenz, als auch das Fehlen des Verzeichnisses testen zu können, bietet Rhino Mocks eine elegante Möglichkeit der Simulation. Zweckmäßig wäre auch hier z.B. ein Interface, um den File-Zugriff von der Anwendungslogik zu trennen.

Interface Beispiel:

```
public interface IFileSystemWrapper
{
    DirectorySecurityHandle StartDirectorySecurityHandling(string path);
    FileSystemChangeResult ConfirmDirectorySecurityHandling(DirectorySecurityHandle dSecHandle);
    FileSystemChangeResult RemoveDirectorySecurity(DirectorySecurityHandle dSecHandle,
                                                   FileSystemAccessRule rule);
    FileSystemChangeResult AddDirectorySecurity(DirectorySecurityHandle dSecHandle,
                                                FileSystemAccessRule rule);
    bool ChangeOwner(DirectorySecurityHandle dSecHandle, string account);
    string GetOwner(DirectorySecurityHandle dSecHandle);
}
```

Nun können wir unter Zuhilfenahme des gerade definierten Interfaces die Klasse entsprechend umarbeiten.

```
public class FileSystemSecurity2
{
    private readonly IFileSystemWrapper fileSystemWrapper;
    private readonly DirectorySecurityHandle pathHandle;

    public FileSystemSecurity2(string path, IFileSystemWrapper fsw)
    {
        this.fileSystemWrapper = fsw;
        this.pathHandle = this.fileSystemWrapper.StartDirectorySecurityHandling(path);
    }

    public bool ChangeOwner(string account)
    {
        this.fileSystemWrapper.ChangeOwner(this.pathHandle, account);
        this.fileSystemWrapper.ConfirmDirectorySecurityHandling(this.pathHandle);
        return true;
    }
}
```

Es folgt ein weiteres angepasstes Unit-Test-Beispiel mit Mock. Die „Abteilung“ *using mock.Record()* spiegelt die gewünschte Verhaltensweise des Simulators wieder. Ein Aufruf von *Expect.Call()* ist exakt als solcher zu verstehen (engl. erwarte Aufruf). Es bedeutet, dass innerhalb des *using mock.Playback()* Teils genau dieser Aufruf mit diesen Parametern einmal erwartet wird. Somit kann ein erwartetes Verhalten für die Peripherie „niedergeschrieben“ werden in dem der Methodenaufruf Return() entsprechend parametriert wird.

```
[TestClass()]
public class FileSystemSecurity2Test
{

    [TestMethod()]
    public void ChangeOwnerTest2()
    {
        MockRepository mock = new MockRepository();
        IFileSystemWrapper fsh = mock.StrictMock<IFileSystemWrapper>();
        using (mock.Record())
        {
            Expect.Call(fsh.StartDirectorySecurityHandling(@"\\myComputer\E$\MyDir"))
                .Return(new DirectorySecurityHandle(@"\\myComputer\E$\MyDir", false));
            Expect.Call(fsh.ChangeOwner(new DirectorySecurityHandle(@"\\myComputer\E$\MyDir", false),
                                        "Administratoren"))
                .Return(true);
        }
        using (mock.Playback())
        {
            const string account = "Administratoren";
            FileSystemSecurity2 underTest = new FileSystemSecurity2(@"\\myComputer\E$\MyDir", fsh);
            bool result = underTest.ChangeOwner(account);
            Assert.IsTrue(result);
        }
    }
}
```

Nun ist es nicht verwunderlich, dass, sollte man diese Technik ernsthaft verwenden wollen, sofort der erhebliche Aufwand erkennbar wird, um die Schnittstellen-Simulation zu formulieren. Aus diesem Grund haben wir uns für eine zusätzliche Aufzeichnungsfunktionalität entschieden.

Die Mitschnitt Ableitung

Es ist kein wirklicher magischer Code, der das Aufzeichnen der Interface Aufrufe realisiert. Die eigentliche Herausforderung erkennt man, wenn man versucht, wieder C#-Compiler übersetzbaren Code zu erzeugen. Als Ersatz für die Übergabeparameter der Interface-Aufrufe braucht man im Falle von ordinalen Datentypen natürlich nur die String.Format()-Funktion zu bemühen. Das stimmt gerade mal für numerische Typen. Alle weiteren halten kleine Besonderheiten bereit. Der Datentyp Boolean präsentiert sich beispielsweise mit großen Anfangsbuchstaben, während syntaktisch von C# „true“ und „false“ eingefordert wird. Noch weitaus interessanter verhält es sich mit Enumerations vom Typ [Flags].
Wechselt man jetzt noch die Seite und hat Referenztypen als Übergabeparameter, so erkennt man, dass es sinnvoll sein könnte, für Klassen ein neues Interface zu definieren, das wir *IRecordable* genannt haben. Es ist relativ unspektakulär und beinhaltet die Funktion string ToTestCode(). Diese Funktion muss von jeder Klasse überschrieben werden, die als Übergabeparameter einer Interfacefunktion fungiert. Ihre Implementierung ist die Rückgabe eines Strings, der den C#-Code für den geeigneten Konstruktor-Aufruf enthält. Geeignet könnte heißen, dass es einen alternativen Konstruktor gibt, der, wie hier im Falle von DirectorySecurityHandle, nur den Parameter Path in seine interne Struktur übernimmt aber keine echten Peripherie-Objekte (*DirectoryInfo*) erzeugt. Dieser Konstruktor wurde dem Attribute *“internal“* versehen, um die normale Benutzung zu verhindern. Zu Zwecke der Sichtbarkeit innerhalb des Unit-Tests gibt es in der AssemblyInfo.cs einen geeigneten *InternalsVisibleTo*-Eintrag.

Bleibt noch die Behandlung von C# eigenen Datentypen wie z.B. *FileSystemAccessRule*. Diese Typen benötigen auch die *ToTestCode*() Methode. Eine zweckmäßige Umsetzung wird durch die Verwendung von Extension Methoden ermöglicht.

Die im nachfolgenden Beispielcode benannte statische Klasse RecordingHelper kapselt alle Besonderheiten der Datentypen für ihre C#-Compiler konforme Entsprechung. Sie besteht aus einer Sammlung von statischen Funktionen für unterschiedliche Typen. Sie regelt auch den Umstand der Übergabe einer *null*-Referenz an die Funktionen. Die *String*-Extension-Methode im Anschluss zeigt die Lösung der Funktionserweiterung für Built-In-Typen.

```
public class DirectorySecurityHandle : IRecordable
{
    private readonly string path;
    public DirectoryInfo Info { get; set; }
    public DirectorySecurity Security { get; set; }
    public string Path
    {
        get { return this.path; }
    }
    internal DirectorySecurityHandle(string directoryPath, string distinguishDummmy)
    {
        this.path = directoryPath;
    }
…
    public string ToTestCode()
    {
        return String.Format(" new {0}({1}, String.Empty)",
            this.GetType(), RecorderHelper.ToTestCode(this.Path));
    }
}

public static class StringExtension
{
…
    public static string ToTestCode(this string me)
    {
        return string.Format("@\"{0}\"", me);
    }
}
```

Ein Aufruf von RecordingHelper::ToTestCode(directorySecurityHandle) für das Directory '\\myComputer\E$\MyDir' erzeugt beispielsweise den nachfolgenden String.

```
"new DirectorySecurityHandle(@"\\myComputer\E$\MyDir", String.Empty)"
```

Dieser String findet dann direkte Anwendung als Übergabeparameter beispielsweise der Funktion

`IFileSystemWrapper.ConfirmDirectorySecurityHandling()`.

Jetzt fehlt nur noch ein kleiner Schritt bis zum Mitschnitt. Bezüglich der Datensenke haben wir uns an bestehende C#-Klassen gehalten. Die Daten werden über herkömmliche Mittel in eine Textdatei transportiert. Die eigentlichen Interface Funktionsaufrufe werden in einer Recorder-Ableitung der normalen Interface-Implementierung generiert. Das nachfolgende Beispiel zeigt die Umsetzung.

```
public class FileSystemWrapperRecorder : FileSystemWrapper
{
    public override DirectorySecurityHandle StartDirectorySecurityHandling(string path)
    {
        try
        {
            DirectorySecurityHandle retval = base.StartDirectorySecurityHandling(path);
            Log.TestCode(null,
String.Format("Expect.Call(fileSystemHelper.StartDirectorySecurityHandling({0})).Return({1});",
                                                 RecorderHelper.ToTestCode(path),
                                                 RecorderHelper.ToTestCode(retval)));
            return retval;
        }
        catch (Exception e)
        {
            Log.TestCode(null,
String.Format("Expect.Call(fileSystemHelper.StartDirectorySecurityHandling({0})).Throw({1});",
                                                 RecorderHelper.ToTestCode(path),
                                                 RecorderHelper.ToTestCode(e)));
            throw;
        }
    }

    public override FileSystemChangeResult ConfirmDirectorySecurityHandling(DirectorySecurityHandle
dSecHandle)
    {
        try
        {
            FileSystemChangeResult retVal = base.ConfirmDirectorySecurityHandling(dSecHandle);
            Log.TestCode(null,
String.Format("Expect.Call(fileSystemHelper.ConfirmDirectorySecurityHandling({0})).Return({1});",
                                                 RecorderHelper.ToTestCode(dSecHandle),
                                                 retVal.ToTypedString()));
            return retVal;
        }
        catch (Exception e)
        {
            Log.TestCode(null,
String.Format("Expect.Call(fileSystemHelper.ConfirmDirectorySecurityHandling({0})).Throw({1});",
                                                 RecorderHelper.ToTestCode(dSecHandle),
                                                 RecorderHelper.ToTestCode(e)));
            throw;
        }
    }
    …
}
```

Wie man an diesem Beispiel sieht, kann man hier auch beliebige Exceptions berücksichtigen. Die Umsetzung der Aufrufe entspricht den Vorgaben von Rhino Mocks. Die Funktionalität des Businesscodes jedoch bleibt unbeeinflusst, wenn die nötige Sorgfalt bei der Umsetzung der *RecorderHelper* und *ToTestCode*() Methoden stattfindet. Unter allen Umständen müssen Nebenwirkungen, verursacht durch die Interface-Recorder-Funktionalität, vermieden werden.

Um mit dem Rhino Mocks Framework erfolgreich simulieren zu können, ist es nötig, die Equals()-Methode für alle diejenigen Klassen zu überschreiben, die in den Interface-Aufrufen als Übergabeparameter benutzt werden. Leider bedarf es einer Sonderbehandlung, wenn Built-In-Klassen zur Anwendung kommen. Da C# nach unserem Kenntnisstand keine Möglichkeit bietet, die Equals()-Methode einer Klasse wie System.Security.AccessControl.FileSystemAccessRule zu überschreiben, ist es nötig, für die Simulation in solchen Fällen auf eine andere Aufrufmimik auszuweichen. Nachfolgendes Beispiel zeigt diese Möglichkeit unter der Verwendung der Lambda Syntax:

```
    Expect.Call(fileSystemHelper.AddDirectorySecurity(null, null))
        .Callback<DirectorySecurityHandle, FileSystemAccessRule>((d, r) =>
            Equals(d, new DirectorySecurityHandle(@"\\myComputer\E$\MyDir ", String.Empty)) &&
            FileSystemAccessRuleExtension.Equals(r,
                new FileSystemAccessRule("S-1-5-21-xxxxxxxxxx-xxxxxxxxxx-xxxxxxxxxx-2219",
                    FileSystemRights.Modify|FileSystemRights.Synchronize,
                    InheritanceFlags.ContainerInherit|InheritanceFlags.ObjectInherit,
                    PropagationFlags.None,
                    AccessControlType.Allow)))
        .Return(FileSystemChangeResult.Successful);
```

Die Angabe von Parametern innerhalb des AddDirectorySecurity()-Aufrufs ist nicht nötig, da durch den Funktionsaufruf von Callback() eine benutzerdefinierte Vergleichsmethode zum Einsatz kommt (FileSystemAccessRuleExtension.Equal()). Die Syntax des Aufrufs entspricht

```
Expect.Call(fileSystemHelper.AddDirectorySecurity(new DirectorySecurityHandle(
                                                     @"\\myComputer\E$\MyDir ",
                                                     String.Empty),
    new FileSystemAccessRule("S-1-5-21-xxxxxxxxxx-xxxxxxxxxx-xxxxxxxxxx-2219",
        FileSystemRights.Modify|FileSystemRights.Synchronize,
        InheritanceFlags.ContainerInherit|InheritanceFlags.ObjectInherit,
        PropagationFlags.None,
        AccessControlType.Allow)))
    . Return(FileSystemChangeResult.Successful);
```

welcher aber nicht funktionsfähig ist, weil die Equals()-Methode der FileSystemAccessRule lediglich die Objektadressen vergleichen würde (Object.Equals) und damit niemals die inhaltliche Gleichheit ermitteln könnte. Komplizierter wird das o.g. Beispiel nur durch zwei Übergabeparameter. Prinzipiell ist aber eine beliebige Anzahl möglich. Denkbar wären auch *ref* bzw. *out* Parameter. Auf diese Besonderheit haben wir allerdings verzichtet.

Ein letztes Wort zu Aufzählungen wie List oder Array. Es besteht auch die Möglichkeit, für die Aufzählungen sowohl die Funktion ToTestCode() als auch Equals()-Methoden zu definieren. Anbei ein Beispiel:

```
    private const string EmptyArray = "new {0}[] {{ }}\r\n";
    private const string ArrayHead = "new [] { \r\n\t";
    private const string ArraySeparator = ",\r\n\t";
    private const string ArrayTail = " }";

    public static string ToTestCode<T>(this T[] array) where T : IRecordable
    {
        if (array.Length == 0)
        {
            return String.Format(EmptyArray, typeof(T).ToString().Replace('+', '.'));
        }
        StringBuilder result = new StringBuilder(ArrayHead);
        foreach (T item in array)
        {
            if (typeof(T).IsValueType)
            {
                result.Append(item.ToTestCode());
            }
            else
            {
                result.Append(item != null ? item.ToTestCode() : "null");
            }
            result.Append(ArraySeparator);
        }
        if (result.Length > ArrayHead.Length)
        {
            result.Length -= ArraySeparator.Length;
        }
```

```
        result.Append(ArrayTail);
        return result.ToString();
    }
```

Noch einmal Rhino Mocks

Das Framework Rhino Mocks beinhaltet unterschiedlichste Möglichkeiten um Klassen, Interfaces, Events oder Delegates zu simulieren. Wir verwenden hier eine mögliche Variante; die Simulation des Interfaces mit einem StrictMock. Prinzipiell sieht der Test-Code etwa so aus.

```
using Rhino.Mocks;
using Rhino.Mocks.Exceptions;

    [TestMethod]
    public void TestMethod()
    {
        MockRepository mock = new MockRepository();
        IFileSystemWrapper fsh = mock.StrictMock<IFileSystemWrapper>();
        using (mock.Record())
        {
            Expect.Call(fsh.StartDirectorySecurityHandling(@"\\myComputer\E$\MyDir"))
                  .Return(new DirectorySecurityHandle(@"\\myComputer\E$\MyDir", String.Empty));
        }
        using (mock.Playback())
        {
            UnderTest underTest = new UnderTest(fsh);
            Assert.IsTrue(underTest.HasExplicit());
        }
    }
```

Die Testmethode gliedert sich in drei Bereiche. Der grüne Bereich betrifft das Erstellen der gewünschten Interface Mocks. Dieser Code wird von einem speziellen Konstruktor unserer UnderTest-Class erzeugt.
Der rote Bereich kapselt die Aufrufe, die das gewünschte Verhalten des Interfaces darstellt. Dabei ist zu bedenken, dass sowohl die Anzahl, die Reihenfolge als auch die Parametrierung der Aufrufe Berücksichtigung finden. Der passende Code wird durch die Recorder-Implementierung während des normalen Betriebes der Software aufgezeichnet.
Schließlich der blaue Bereich, der die eigentliche zu testende Klasse instanziiert und ausführt. Dieser Code wurde zweckmäßig in die *Dispose*()-Methode der "*UnderTest*"-Klasse platziert. Dies ermöglicht uns dann die Verwendung der „using"-Syntax.

Fazit

Die Firma protected-networks.com GmbH verwendet dieses Framework zur Gewährleistung der Verfügbarkeit der Algorithmen für das Berechtigungsmanagement innerhalb des 8MAN Produkts.

Grenzen der Applikation

Bei der Aufzeichnung von Test Cases im Rahmen der normalen Benutzung können textuell sehr umfangreiche Test Cases zustande kommen.

Wenn Änderungen im Algorithmus selber dazu führen, dass neue Interface-Funktionsaufrufe erzeugt werden, dann ist eine Anpassung der Test Cases notwendig und sollte bei Geringfügigkeit manuell erfolgen. Die korrigierte Version ermöglicht dann wiederum einen Regressionstest. Wenn

sichergestellt ist, dass die zusätzlichen Teile des Algorithmus arbeiten, dann kann man diesen neu aufzeichnen lassen.

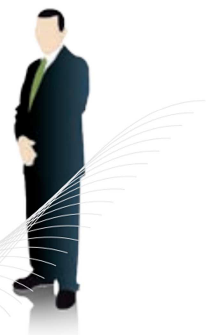
Kundenprobleme, aufgenommen mit „älteren" Versionen, verlieren u.U. dann ihre Existenzberechtigung. Dieser Aspekt ist aber nicht problematisch, da diese Tests „nur" zur Fehleranalyse benutzt wurden.

Quellen

[1] Rhino Mocks, http://www.ayende.com/wiki/Rhino+Mocks+Documentation.ashx

Über protected-networks.com GmbH

Die protected-networks.com GmbH mit Sitz in Berlin entwickelt integrierte Lösungen für das Berechtigungsmanagement in Server-Umgebungen für Unternehmen aller Branchen. Integrated Data Security Management heißt der Ansatz, den protected-networks.com verfolgt und Unternehmen eine einheitliche Lösung für die Verwaltung von Berechtigungen und Daten bietet. Die Lösung **8MAN** ermöglicht das Visualisieren, Administrieren, Dokumentieren und Optimieren aller Berechtigungen, die innerhalb der IT-Umgebung bestehen. Damit erweitert und optimiert **8MAN** nicht nur die Funktionen der Microsoft-Dienste wie Active Directory, Fileserver, SharePoint und Exchange, sondern auch die Funktionen vieler anderer Serversysteme. Die protected-networks.com GmbH wurde Anfang 2009 gegründet und ist durch die Investitionsbank Berlin sowie dem High Tech Gründerfonds finanziert.



protected-networks.com GmbH
Alt-Moabit 73
10555 Berlin
Tel. 030 / 390 63 45 - 50 | Fax 030 / 390 63 45 - 51
info@protected-networks.com
www.protected-networks.com