Dynamisch und gefährlich?

C# Dynamics in freier Wildbahn

von Tim Bourguignon

Was passiert genau, wenn das C# 'dynamic'-Schlüsselwort verwendet wird? Manche sagen, dass die Hölle ausbrechen wird. Anderen, dass ein Kätzchen für jede Verwendung stirbt... und wir sprechen von echt süßen Kätzchen! Seit dessen Einführung in die Sprache C#4.0 im Jahr 2010 habe ich mich bemüht, die Kätzchen zu retten, also keine 'dynamics' zu verwenden. Aber das gilt seit 2012 nicht mehr. Wenn Kätzchen wirklich im Gefahr sind, wenn man 'dynamics' verwendet, werde ich bald mit Tierrechtsorganisationen Ärger bekommen.

Dynamics?

Rudern wir kurz zurück für diejenigen, die es verpasst haben. C# sollte eine stark typisierte Sprache sein; was ist dann dieses 'dynamic', von dem wir reden?

Es stimmt, C# ist eine stark typisierte Sprache. Die Sprache drückt Typ-Beschränkungen aus, die an Compile-Zeit verifiziert und validiert werden. Folgender Code kompiliert zum Beispiel nicht:

```
string language = "C#";
language++;
```

Es ist klar: der Compiler merkt, dass der "++" Operator nicht auf 'string' Operand appliziert werden kann und wirft einen Fehler zurück. Ebenso - sofern keine 'ToUppper' Erweiterungs-Methode definiert wird - kompiliert Folgendes auch nicht besser:

```
int theAnswer = 42;
theAnswer.ToUpper();
```

Der Compiler kann da keine Methode-Definition für 'ToUpper' finden, der ein Argument von Typ 'int' braucht.

Was passiert, wenn wir den 'string' und 'int' Typen mit 'dynamic' tauschen? Ihr solltet da den Compiler sofort lächeln und alle Fehler verschwinden sehen. In dynamischen Sprachen bekommen wir die 'Typ-Sicherheit' erst zur Laufzeit . Die Beispiele hier oben kompilieren... krachen aber trotzdem wunderbar! Toll!

Warum hat denn dann C# überhaupt ein "dynamic"- Schlüsselwort?

Klauen wir ein einfaches Beispiel von Scott Hanselman [1]:

```
Calculator calc = new Calculator();
int sum = calc.Add(10, 20);
```

Was würde denn passieren, wenn wir 'object' statt 'Calculator' als Variablen-Typ verwenden würden? Dann weiß der Compiler nicht dass eine 'Add' Methode existiert und ein Fehler wird natürlich angezeigt. Wenn wir mehrere Objekte haben, die eine 'Add' Methode besitzen, definieren wir in der Regel eine Schnittstelle, die diese gemeinsame Fähigkeiten zusammenfasst. Dann können wir die unterschiedlichen Objekte über ihre Schnittstellen ansprechen.

Wir können aber ohne Schnittstelle trotzdem klar kommen. Mit Hilfe von 'Reflection' können wir auf die Methode: 'Add' zur Laufzeit trotzdem zugreifen. Es funktioniert, die Syntax ist aber, naja...weniger schön:

```
object calc = new Calculator();
Type calcType = calc.GetType();
object res = calcType.InvokeMember("Add",
        BindingFlags.InvokeMethod, null, new
object[]  { 10, 20 });
int sum = Convert.ToInt32(res);
```

Diese 'Add' Method existiert jedoch. Schön wäre, wenn wir diese Compile-Zeit Prüfung einfach überspringen könnten. Dann würde das Programm zur Laufzeit korrekt reagieren. Das genau erlaubt uns das 'dynamic' -Schlüsselwort:

```
dynamic calc = new Calculator();
int sum = calc.Add(10, 20);
```

Der Compiler erlaubt alles, was nach dem 'dynamic'-Schlüsselwort kommt, solange es sich an die C# Syntax hält.

Die Idee, einen ObjektTyp über seine Methoden und Attribute anstatt über seine Klasse zu definieren, trägt der Name 'Duck-Typing'. Wie James Whitcomb Riley es beschrieben hat: *„Wenn ich einen Vogel sehe, der wie eine Ente läuft, wie eine Ente schwimmt und wie eine Ente schnattert, dann nenne ich diesen Vogel eine Ente.“* [2]. Duck-Typing ist ein Kernkonzept aller dynamischen Sprachen. Damit steigt die Sprach-Flexibilität dramatisch, gleichzeitig sinkt aber auch deren Fehlerrestistenz. Da der Compiler uns nicht mehr Tipp- und Typfehler anzeigt, müssen wir diese explizit mit Unit Tests ausschließen.

Zusätzlich zum Verlust der Compile-Time-Verifizierung, können dynamische Sprachen nur schwer Methoden & Eigenschaften ableiten. Daher kann sie auch kein so mächtiges Tool wie zum Beispiel IntelliSense [3] bieten. C# ist da kein Ausnahme, sobald wir 'dynamic' verwenden, wird IntelliSence stumm. Manche sagen, das es eher von Vorteil wäre... Geschmacksache.

.NET Dynamische Sprachen

Das eine Szenario, in welche dieses dynamische Verfahren wirklich Sinn macht, besteht in der Interoperabilität mit dynamischen Sprachen wie Python und Ruby, oder eher dessen .NET Varianten IronPython [4] und IronRuby [5]. In diese Fälle erlaubt das 'dynamic'-Schlüsselwort Python oder Ruby Scripte, auf die CLR und die .NET Plattform zu zugreifen.

Hier ist das Python Script, welches wir gerne verwenden würden:

```
def add(a, b):
    "add(a, b) -> returns a + b"
    return a + b
```

In C# brauchen wir nur eine Python Runtime bauen, das Script in ein 'dynamic' Objekt laden und schon können wir diese 'add' Methode aufrufen:

```
var pythonRuntime = Python.CreateRuntime();
```

```
dynamic pythonScript =
            pythonRuntime.UseFile("script.py");
var result = pythonScript.add(100, 200)));
```

Ja, und?

Das alles ist bisher weder neu noch wirklich spannend. Iron-Interoperabilität machen ernsthaft erst einige wenige Projekte, also wo will ich dann hin?

Benutzer des ASP.NET MVC Framework [6] haben schon mal ein dynamisches Objekt verwendet: der 'ViewBag'. In den letzten Jahren habe ich mehrere Projekte entdeckt wie zum Beispiel NancyFx[7], Simple.Data[8] oder Oak [9] die diese dynamische Konstrukte häufiger verwenden (arme Kätzchen). Es schaut also so aus, als ob es ein paar Szenarien gäbe, in welchen diese dreckige Dynamics schon interessant wären...

APIs

Bevor wir weiter gehen, würde ich gerne kurz über API reden. Ein massiver Nachteil der 'dynamic' ist dies Beschränkung der Entdeckbarkeit. IntelliSense ist ein sehr gutes Tool und hilft sehr gut, sich zu informieren, was mit welche Objekt gerade machbar ist und es beschränkt die Typ-Fehler- Szenarien.

Ein API zu basteln bedeutet, sich viele Gedanken darüber zu machen, was die Benutzer wirklich damit machen wollen. Sie sollte nicht nur Probleme lösen können. Eine gute API sollte einfach zu bedienen sein und das Leben dessen Benutzer erleichtern. Objektorientiert oder funktional? Mit oder ohne Schnittstellen? Anhand von JSON, XML oder sonst was? So viele Fragen, die man berücksichtigen soll ; und dynamische Konzepte sind selten in dieses Gedankenprozessen dabei. Und doch haben sie was anzubieten. Schauen wir einfach weiter!

DynamicObject

Das System.Dynamic.DynamicObject ist die Basis für alle dynamischen Objekte. Ein DynamicObject belichtet seine Members an der Laufzeit über drei Methoden, die man überschreiben kann:

- TrySetMember
- TryGetMember
- TryInvokeMember

Diese Methoden auf eine schlaue Art und Weise zu kombinieren, ist der Schlüssel für diese dynamische Verwendung.

ExpandoObject

Das System.Dynamic.ExpandoObject ist ein Objekt, welches Members zur Laufzeit dynamisch hinzufügen und entfernen kann.

Zum Beispiel:

```
dynamic myExpando = new ExpandoObject();
myExpando.number = 10; //Eigenschaft hinzufügen
myExpando.Increment = (Action)(() =>
        { myExpando.number++; }); //Methode
hinzufügen
Console.WriteLine(myExpando.number); //10
sampleObject.Increment(); //Methoden Aufruf
Console.WriteLine(myExpando.number); //11
```

Anstatt 'Number' oder 'Increment' hätte ich alles mögliche schreiben können. Diese Eigenschaften und Methoden existieren erst einmal nicht und werden zur Laufzeit hinzugefügt. Innerlich verwendet das ExpandoObject ein Art Dictionary, um diese dynamischen Eigenschaften zu speichern. Das Dictionary hat Eigenschafts-Namen als Schlüssel und die Werte können direkt via der *„object.<key>“* z.B. 'expando.number', Syntax aufgerufen werden.

Der ASP.NET MVC ViewBag weist viel Ähnlichkeit zu einem ExpandoObject auf - und für die meistens Entwicklern ist das ExpandoObject ein einfacher 'Eigenschaft-Sack'.
Auf ihre Blog [10] hat Alexandra Rusina aber eine andere Verwendung von ExpandoObjects dargestellt: XML Strukturen Behandlung. Anhand von ExpandoObjects kann man hierarchische Strukturen bauen die sich danach einfacher als XElements behandeln lassen. Der Konvertierung von und zu XML muss per Hand gemacht werden aber ein Gewinn an Komplexität ist es schon.

Noch besser, ExpandoObjects implementiert die *INotifyPropertyChanged* Schnittstelle und kann sogar Ereignisse als Eigenschaften speichern. Damit können ExpandoObjects beobachtet werden und in DataBinding Szenarien verwendet werden ; dies ist ungefähr das, was NancyFx tut.

ElasticObject

Das ElasticObject [11] [11] ist ein OpenSource Cousin des ExpandoObjects. Es versucht, Arbeit mit tiefen Objekt-Graphen zu erleichtern. Es bringt Interessante Erweiterungen zum ExpandoObject und ist tatsächlich in manchen Fällen sehr sinnvoll. Meiner Meinung nach ist es sonst leider schöner auf dem Papier als in der Realität. Ich werde also jetzt nicht weiter drüber reden, aber dessen Aufbau ist es einen Blick Wert.

Gemini

Oak ist ebenfalls OpenSource und versucht den Entwicklungsprozess von 'Single Page Applications' mit .NET zu erleichtern. Es versucht auch, C# und Javascript besser miteinander zu kombinieren, es führt Konvertierungen von JSON zu dynamic-Objekten durch, um ViewModels und DTOs zu vermeiden. Oak bringt weiterhin ähnliche Konstrukte und Tools wie Rails mit, zum Beispiel automatische build- und deploy-Funktionen, um den Entwicklungsprozess an sich 'reibungsloser' zu machen.

Unter anderem verwendet Oak als Kernkomponente eine sogenannte 'Gemini'-Komponente. Ähnlich wie ein ExpandoObject können zu einem Gemini-Objekt Eigenschaften und Methoden hinzugefügt werden. Ein Gemini-Objekt kann aber sich selbst beobachten und seine Members zur Laufzeit analysieren.

```
dynamic gemini = new Gemini(new { FirstName =
"Jane", LastName = "Doe" });
gemini.MiddleInitial = "J";

gemini.RespondsTo("FirstName"); //this would return
true
gemini.RespondsTo("IDontExist"); //this would return
false

gemini.SayHello = new DynamicFunction(() =>
"Hello");
gemini.SayHello(); //calling method

IDictionary<string, object> members =
gemini.Hash(); //Get the members of the object
IEnumerable<string, object> methods =
gemini.Delegates(); //Get the Methods of the object
```

Oak verwendet meistens intern Gemini-Objekte, zum Zweck des Model-Bindings, um Daten hin und zurück vom Views zu reichen, aber auch als Schlüsselkomponente, um Datenbanken anzusprechen, JSON Konvertierungen zu leisten usw.

NancyFx

Nancy [12] ist ein OpenSource MicroFramework und erlaubt es, WebApplikationen zu bauen. Ähnlich wie Sinatra [13] auf der Ruby-Stack bietet es eine leichtgewichtige und zeremonielose Art und Weise, Webseiten zu anbieten. Gegensätzlich zu ASP.NET MVC enthält Nancy nur die aller nötigsten Kernfunktionalitäten 'Out-of-the-box' und muss für alles anderes mit Erweiterungsmodulen ergänzt werden.

Um DataBinding zu leisten, verwendet Nancy dynamische Objekte. Wenn man eine 'Route' definiert, versucht Nancy dafür Parameter zu schnappen und bietet dem Nutzer diese dann in Form eines dynamischen Objekts an.

Zum Beispiel: die folgende Route wird anhand eines HTTP-GET Request auf "/hello/bob" oder "/hello/nancy" URIs aktiviert. Nancy empfängt dann "bob" oder "nancy" und verpackt es in den dynamischen Dictionary 'parameters'.

```
Get["/hello/{name}"] = parameters => {
    return "Hello " + parameters.name;
};
```

Nancy kann unterschiedliche Arten von Parametern fangen: QueryString, CapturedParameters und direkt in das Body der Request. Alle diese Parameter in ein Dictionary zu packen ist eine schlaue Idee. Es ist sehr einfach für den Anwender und auch einfacher für das Framework selbst.

Bei diesem Beispiel: "/hello/nancy?option=default" sieht man gleich, dass parameter.name einen Wert haben wird. Allerdings ist es deutlich schwieriger zu sehen, dass parameter.option auch existiert.

Nancy verwendet auch dynamische Objekte in den Ergebnissen von den Routen. Laut der Theorie kann Nancy mit jedem Return-Objekt leben. Nancy ist um eine Adapter-Architektur gebaut. Falls das Objekt einen bekannten Typ hat, wird es an ein spezialisiertes Modul

weitergegeben, sonst wird es als dynamisches Objekt verpackt und an den View weitergereicht. Diese 'dynamic' Verwendung ist einfacher für das Framework. Es werden weniger Interfaces benötigt und Adapter können nachgeliefert werden. So bleibt die Flexibilität so hoch es geht... mit allen Nachteile, die man sich nur ausdenken kann, wenn irgendwas doch schief gehen sollte...

Simple.Data

Simple.Data geht noch einen Schritt weiter. Anhand eines massiven Verbrauchs von der 'TryGetMember' Methode, erlaubt es die Spontan-Definition von Query-ähnlichen Strukturen, um Datenbanken zu manipulieren. Der Syntax ist dann viel näher an einer menschlichen Sprache und lässt sich viel besser bearbeiten. Im folgendem Beispiel ist 'database' ein dynamisches Objekt:

```
database.Customers.FindAllByGenreId(3);

//The produced query: SELECT * FROM Customers
WHERE GenreId = 3
```

Der Eckstein von Simple.Data ist diese schlaue Kapselung von Objekten, Kommandos und Daten in einer sehr lesbaren Form.

Die meisten Klassen in Simple.Data erben von DynamicObject und verwenden 'Regexes', um die Members zu parsen, e.g. die verschiedenen Blöcke in 'TryGetMember' zu identifizieren.

Dieses Verwendung von 'dynamics' erlaubt eine flexible Verkettung von datenbankorientierten Konzepten. Weil der Set an mögliche Kommandos relativ gering ist (Find, FindAllByXXX, OrderByXXX, Select, WithXXX, Where etc.), bleibt die Entdeckbarkeit hoch.

Fazit

In dieses Artikel haben wir verschiedene Konzepte und Verwendungen der 'dynamic' Schlüsselwort kennen gelernt. DynamicObject und ExpandoObject sind Grundsteine der Sprache C#. Sie können zur Manipulation von XML benutzt werden. IhreVarianten, ElasticObject und Gemini Objekte, können auch als flexible DataTransferObjects-verwendet werden. NancyFx verwendet 'dynamics' zusätzlich auch als

architekturelle Bausteine, um Flexibilität zu gewinnen. Simple.Data geht einen Schritt weiter und baut eine Logikebene auf die 'dynamics', um eine bessere API anbieten zu können.

Neben dem gedanklichen Spiel der 'dynamics' ist der API Aufbau der interessante Teil von dieser Verwendung. Welche Tools wollen wir unsere Kunden (e.g. Entwicklern) zukommen lassen? Welche Sprache sollten sie sprechen? Etwas Einfaches für mich? Oder eher etwas Einfaches für sie? Wie kann ich die Flexibilität meine Applikation so hoch wie möglich behalten? Dynamisch Konstrukte können da sehr gut helfen und ich wette, es wird noch andere gute Beispiele in der Zukunft geben.

Diese waren die 'Dynamics in Beispielen aus der freien Wirldbahn, die mir bekannt sind. Falls ihr andere Exemplare kennt, würde ich mich sehr freuen, das zu erfahren. Falls ihr es nicht für mich machen möchtet, macht es bitte für die Kätzchen!

Referenzen

[1] „C#4 and the dynamic keyword whirlwind Tour around .NET 4 (and Visual Studio 2010) Beta 1“ Scott Hanselman http://www.hanselman.com/blog/C4AndTheDynamicKeywordWhirlwindTourAroundNET4AndVisualStudio2010Beta1.aspx

[2] DuckTyping http://de.wikipedia.org/wiki/Duck_Typing

[3] IntelliSense definition http://en.wikipedia.org/wiki/Intelli-sense

[4] IronPython Homepage http://ironpython.net/

[5] IronRuby Homepage http://www.ironruby.net/

[6] Asp.Net MVC Homepage http://www.asp.net/mvc

[7] NancyFx Homepage http://nancyfx.org/

[8] Simple.Data Repository https://github.com/markrendle/Simple.Data

[9] Oak haupt Repository http://amirrajan.github.io/Oak/

[10] „Dynamics in C#4.0: Introducing the ExpandoObject“ Alexandra Rusina http://blogs.msdn.com/b/csharpfaq/archive/2009/10/01/dynamic-in-c-4-0-introducing-the-expandoobject.aspx

[11] ElasticObject haupt Repository https://github.com/amazedsaint/ElasticObject

[12] „Erste Date mit Nancy“ und „Zweite Date mit Nancy“ Tim Bourguignon http://www.bookware.de/kaffeeklatsch/archiv/KaffeeKlatsch-2012-07.pdf - http://www.bookware.de/kaffeeklatsch/archiv/KaffeeKlatsch-2013-02.pdf

[13] Sinatra für Ruby Homepage http://www.sinatrarb.com/