

Enums im Dutzend

Delphi-Entwickler, die zu .NET wechseln, vermissen in der Regel den Datentyp Set, der den Umgang mit Enumerationen bequem macht. Es gibt zwar Ersatzlösungen, die allerdings nicht überzeugen. Das hat den Autor dazu gebracht, selbst eine Klasse zu entwickeln, um die Leistungsfähigkeit und den Komfort von Sets auch in .NET nicht missen zu müssen.

Auf einen Blick



Bernd Klaiber ist als Softwarearchitekt bei der Firma Höfler Maschinenbau GmbH, Ettlingen, im Bereich Automatisierung tätig. Nach langen Jahren mit Delphi setzt er nun seinen Schwerpunkt auf .NET-Entwicklung in C#. Sie erreichen ihn unter bkdev@bkdev.de

Inhalt

- ➔ In Pascal stellen sogenannte Sets Funktionen bereit, die das Arbeiten mit Enumerationen vereinfachen.
- ➔ .NET kennt keine Sets, Pascal-Compiler für .NET setzen entsprechende Codebefehle aber mehr oder weniger geglückt um.
- ➔ Eine eigene Set-Klasse ist einfach nachzuprogrammieren und stellt Sets auch in C# oder Visual Basic ohne Ballast zur Verfügung.



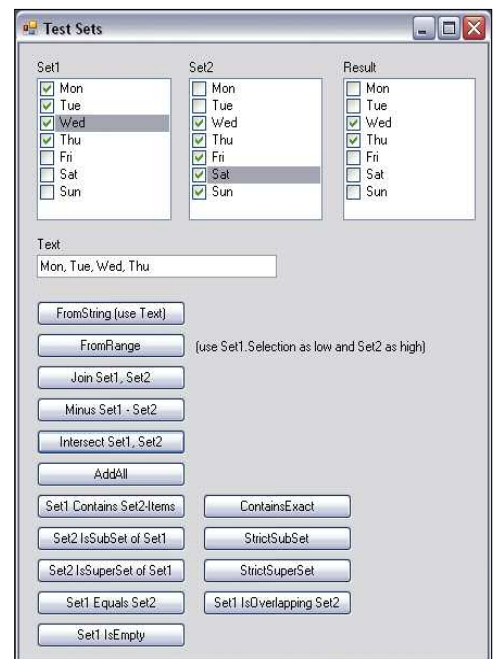
dnpCode
A0905Sets

Enumerationen eignen sich hervorragend, um eine Reihe benannter Zustände oder Werte zu repräsentieren. Anders als bloße Konstanten bilden die Werte eines solchen Aufzählungstyps eine zusammengehörende Gruppe. Dabei kann der Entwickler sich einfach mit den Werten zufriedengeben, welche die Sprache automatisch zuweist, oder sie auch selbst so zuweisen, wie er sie benötigt.

In Pascal lässt sich aus jeder Enumeration auch ein sogenanntes Set bilden. Das Set repräsentiert die Menge der ihm zugrunde liegenden Enumeration und kann mehrere Werte gleichzeitig setzen oder leer sein. Sets des gleichen Typs lassen sich mit den aus der Mathematik bekannten Mengenoperationen wie Schnittmenge oder Vereinigungsmenge verarbeiten und kombinieren.

.NET kennt solche Sets zwar nicht von Haus aus, aber nötig wären sie hier ebenso. Adäquate Funktionen lassen sich mithilfe von Bit-Operationen erreichen. Dazu sind die Werte der Enumerationen binär zu codieren wie zum Beispiel in Listing 1. Auffällig dabei ist das Attribut *Flags*. Es zeigt an, dass die Werte der Enumeration mit binären Operationen zu verwenden sind. Zusätzlich muss der Entwickler die Werte als Potenzen von 2 angeben, also 1, 2, 4, 8 und so weiter. Dabei achtet der Compiler nicht auf die Folge der Werte. Dafür ist der Programmierer alleine zuständig. Er kann aber bei Bedarf auch schon Kombinationen von Werten als eigenständige Werte anlegen.

Das Prinzip an sich ist also ebenso umsetzbar – komfortabel ist es nicht. Denn wie sähe eine



[Abb. 1] Das Beispielprogramm zeigt den Umgang mit der Set-Klasse anhand von Wochentagen.

Prüfung aus, welche Bits tatsächlich gesetzt sind? Der Entwickler müsste alle Möglichkeiten mithilfe von Bit-Operationen testen. Ebenso stört, dass einem solchen „Flag-Grab“ nicht anzusehen ist, ob es ein solches ist oder nicht. Wird ein solcher Typ einer Methode übergeben, ist zunächst nicht klar, ob die Methode einen einzelnen Wert der Enumeration erwartet oder ob sie auch mit einem Bitfeld klarkommt.

Listing 1

Eine binär codierte Enumeration in C#.

```
[Flags]
public enum DayF { Mon = 1, Tue = 2, Wed = 4, Thu = 8, Fri = 16, Sat = 32, Sun = 64 };
...
bool IsWorkingday(DayF day)
{
    DayF WorkingDays = DayF.Mon | DayF.Tue | DayF.Wed | DayF.Thu | DayF.Fri;
    return (day & WorkingDays) == day;}
}
```

Listing 2

Ein einfaches Pascal-Programm, das Sets definiert.

```

program SetTest;

{$APPTYPE CONSOLE}
uses
  SysUtils;
type
  Day = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  Days = set of Day;
const
  WorkingDays: Days = [Day.Mon..Day.Fri];

function IsWorkingDay(aDay: Day): Boolean;
begin
  result := aDay in WorkingDays;
end;

begin
  Console.WriteLine(System.String.Format(
    'Wed is a Weekday ? {0}',
    IsWorkingDay(Day.Wed).ToString()));
  Console.ReadLine();
end.

```

Das ist bei dem Komfort, den Programmiersprachen heute bieten, nicht hinzunehmen. Aber es ist wenigstens möglich, einen entsprechenden Set-Mechanismus in .NET zu simulieren, um auf einfache Weise die Liste der in der Menge enthaltenen Elemente abzurufen.

.NET-Pascal

Anhand zweier Implementierungen von Pascal für das .NET Framework hat der Autor untersucht, wie die jeweiligen Compiler die Sets für .NET umsetzen: CodeGears RAD Studio 2009 [1] und RemObjects Oxygene [2]. Diesem Zweck dient das kleine typische Pascal-Programm in Listing 2, das eine Enumeration *Day* samt dem zugehörigen Set *Days* und davon die Untermenge *WorkingDays* definiert. Bei der Untersuchung, wie RAD Studio diesen Code für .NET umsetzt, ist das frei verfügbare Tool Reflector [3] von großem Nutzen. Das Ergebnis sieht folgendermaßen aus:

```

public static bool IsWorkingDay(
  Day aDay)
{
  return (((int) aDay) < 8) ?
    (((int) WorkingDays) & (1 <<
      aDay)) : 0) > 0);
}

```

Hoppla – dieses Beispiel dürfte als „Spickvorlage“, wie eine Umsetzung funk-

Listing 3

So setzt der Pascal-Compiler Oxygene den Code von Listing 2 in .NET um.

```

[Serializable, StructLayout(LayoutKind.Sequential), CompilerGenerated]
internal struct Days
{
  private ulong 0000;
  public const int Mon = 0;
  public const int Tue = 1;
  public const int Wed = 2;
  public const int Thu = 3;
  public const int Fri = 4;
  public const int Sat = 5;
  public const int Sun = 6;
  private const int @Set = 1;
  public Days(ulong[] aData);
  public ulong[] ToArray();
  public void Set(int aElement);
  public void SetRange(int aLeft, int aRight);
  public void Clear(int aElement);
  public static bool op_In(int aElement, Days aSource);
  public static bool op_In(Day aElement, Days aSource);
  public override int GetHashCode();
  public static bool operator ==(Days aFirst, Days aSecond);
  public static bool operator !=(Days aFirst, Days aSecond);
  public override bool Equals(object obj);
  public static Days operator +(Days aFirst, Days aSecond);
  public static Days operator *(Days aFirst, Days aSecond);
  public static Days operator -(Days aFirst, Days aSecond);
  public static bool operator >=(Days aFirst, Days aSecond);
  public static bool operator <=(Days aFirst, Days aSecond);
}

```

tionieren könnte, ausscheiden; seine Komplexität und mangelnde Transparenz machen es dafür ungeeignet.

Die zweite Analyse legt den Pascal-Compiler Oxygene zugrunde, den das Softwarehaus RemObjects als Plug-in für Visual Studio anbietet. Er bindet sich vollständig in die Entwicklungsumgebung ein und erweitert die Visual-Studio-Sprachen C# und Visual Basic um Pascal. Künftig will RemObjects übrigens zusammen mit CodeGear eine Lösung unter dem Namen Delphi Prism anbieten.

Oxygene setzt den Code von Listing 2 so um, wie es in Listing 3 zu sehen ist. Der Compiler baut aus der Enumeration eine Set-Klasse in Form einer Struktur auf. Ohne weiter auf die Details dieser Klasse einzugehen, scheint dieser Ansatz flexibler zu sein.

Ziel ist es nun, eine solche Klasse nicht automatisch durch den Compiler erzeugen zu lassen, sondern im Folgenden eine Klasse in C# zu erstellen, die den vollständigen Komfort wie auch die Transparenz der Struktur bietet.

Die Klasse Set<T>

Die Klasse *Set*, die nun zu entwickeln ist, ist generisch und besteht im Wesentlichen aus einer Liste, genauer gesagt aus einem *SortedDictionary*. Seine Elemente stellen die Menge der Enumerationselemente dar. Zusätzlich bietet die Klasse natürlich eine Fülle von Methoden, welche den Umgang mit Mengen komfortabel und einfach machen. Listing 4 zeigt ihre Deklarationen.

Mit dieser *Set*-Klasse lässt sich die kleine Aufgabe aus Listing 1 so implementieren:

```

public enum Day {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun };
...
bool IsWorkingDay(Day day)
{
  Set<Day> WorkingDays =
    Set<Day>.FromRange(Day.Mon,
      Day.Fri);
  return WorkingDays.Contains(day);
}

```

In dem Beispiel wird zunächst die Enumeration *Day* definiert und basierend auf diesem Typ ein Set namens *WorkingDays*. Die Logik empfiehlt hier, die Enumeration

im Singular zu benennen, das *Set* im Plural; das signalisiert später, ob es sich um ein *Enum* oder ein *Set* handelt.

Das *Set* wird mithilfe einer statischen Methode erzeugt, die es gleich mit den Werten füllt. *FromRange()* definiert dabei,

dass die Wochentage von *Mon* bis *Fri* gehen. Es wäre natürlich auch möglich, den normalen Konstruktor zu verwenden und ihm die einzelnen Elemente in einem offenen Array zu übergeben. Neben dem Konstruktor, dem die Elemente mitgege-

ben werden können, gibt es drei weitere Möglichkeiten, ein neues *Set* mithilfe von statischen Methoden zu instanzieren. Eine davon haben Sie gerade mit *FromRange()* kennengelernt:

- *FromRange()*: Hier geben Sie einen Bereich an, dessen Elemente hinzugefügt werden.
- *FromString()*: Füllt das *Set* anhand eines Textes, der Elemente mit Komma getrennt enthält.
- *FromFlaggedEnum()*: Diese Methode eignet sich sehr gut als Schnittstelle zu den .NET-Sets, die unter Verwendung von Bits gebildet werden.

Die *Set*-Klasse unterstützt alle bekannten Mengenoperationen. Je nach Vorliebe des Anwenders können Sie die Operationen mithilfe einer Methode oder mit den überladenen Operatoren nutzen. Die Operatoren und die dazugehörigen Methoden erläutert Tabelle 1.

Um sich mit den Mengenoperatoren vertraut zu machen und sie zu testen, finden Sie auf der CD zu diesem dotnetpro-Heft ein kleines Testprogramm (Abbildung 1), das gleichzeitig als Beispiel dient.

Teste mich!

Die Visual-Studio-Projektmappe enthält auch ein Testprojekt, das alle Methoden der Klasse prüft. Eine derart in sich abgeschlossene Klasse wie *Set* eignet sich hervorragend für automatisierte Tests, da es keine komplexen Interaktionen mit anderen Systemen gibt.

Als Test-Framework dienen hier die Unit-Tests von Microsoft im Namensraum *Microsoft.VisualStudio.TestTools.UnitTesting*. Da es sich um Visual-Studio-2008-Projekte handelt, ist dieses Test-Framework auch in die Professional-Version der Entwicklungsumgebung schon integriert, sodass Umwege über beispielsweise NUnit nicht nötig sind. Wer trotzdem mit NUnit arbeiten will, kann die Testdeklarationen leicht anpassen.

Um sicherzustellen, dass alle Methoden und ihre Verzweigungen durchlaufen werden, hat der Autor das Codeabdeckungs-Tool NCover [4] verwendet. Es protokolliert alle Aufrufe während eines Tests und prüft so, ob alle Methoden aufgerufen wurden. In einer übersichtlichen Grafik zeigt es dann an, welche Zeilen außen vor geblieben sind. Dort sollte der Test erweitert werden, um zu einem 100-Prozent-Ergebnis zu kommen. Da die Methoden bei der Entwicklung ohnehin in irgendeiner Form ge-

Listing 4

Methoden der Set-Klasse.

```
public Set(params T[] enumitems)
public int Count { get { return setenums.Count; } }
protected Type enumtype;
protected SortedDictionary<int, T> setenums = new SortedDictionary<int, T>();
protected virtual void CheckEnumParams(params T[] enumitems)
public T[] Enums
public bool Contains(params T[] enumitems)
public bool ContainsExact(params T[] enumitems)
public bool IsEmpty()
public void Add(params T[] enumitems)
public void Remove(params T[] enumitems)
public void Complement()
public void Clear()
public void AddAll()

public Set<T> Join(Set<T> OtherSet)
public Set<T> Minus(Set<T> OtherSet)
public Set<T> Intersect(Set<T> OtherSet)
public Set<T> XOR(Set<T> OtherSet)
public Set<T> Clone()
public bool IsOverlapping(Set<T> OtherSet)
public bool IsSubsetOf(Set<T> OtherSet)
public bool IsStrictSubsetOf(Set<T> OtherSet)
public bool IsSupersetOf(Set<T> OtherSet)
public bool IsStrictSupersetOf(Set<T> OtherSet)
public new static bool Equals(object objA, object objB)
public override bool Equals(object obj)
public override int GetHashCode()
public static T[] Parse(string value)
public static Set<T> FromString(string value)
public static Set<T> FromRange(T low, T high)
public void ParseAdd(string value)
public static void CheckFlagged()
public static bool IsFlags()
public T ToFlaggedEnum()
public static Set<T> FromFlaggedEnum(T value)
public static implicit operator string(Set<T> s)
public override string ToString()

public static bool operator ==(Set<T> s1, Set<T> s2)
public static bool operator !=(Set<T> s1, Set<T> s2)
public static bool operator <(Set<T> s1, Set<T> s2)
public static bool operator <=(Set<T> s1, Set<T> s2)
public static bool operator >(Set<T> s1, Set<T> s2)
public static bool operator >=(Set<T> s1, Set<T> s2)
public static Set<T> operator +(Set<T> s1, Set<T> s2)
public static Set<T> operator -(Set<T> s1, Set<T> s2)
public static Set<T> operator *(Set<T> s1, Set<T> s2)
public static Set<T> operator ^(Set<T> s1, Set<T> s2)
public static Set<T> operator ~(Set<T> s)
public string ToString(string format, IFormatProvider formatProvider)
IEnumerator<T> IEnumerable<T>.GetEnumerator()
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
```

testet werden müssen, bietet sich der Test-First-Ansatz an.

Auf diese Weise ergeben sich am Ende der Entwicklung die Tests, die jederzeit die volle Funktionsfähigkeit der Klasse prüfen können – ohne großen Mehraufwand. Oft scheuen sich Entwickler, etwas umzustrukturieren – aus Angst, der bis dahin lauffähige Code könnte plötzlich und unbemerkt fehlerhaft werden. Mit einer solchen Testmöglichkeit aber fallen Änderungen deutlich leichter, da sich die Funktionsfähigkeit der Komponente jederzeit prüfen lässt.

Fazit

Die hier vorgestellte *Set*-Klasse erleichtert den Umgang mit Mengen ungemein. Gegenüber den Flags bietet sie deutlich mehr Funktionen und vor allem eine bessere Lesbarkeit im Code. Durch die Verwendung von generischen Funktionen und überladenen Operatoren lässt sich die Klasse sehr intuitiv einsetzen. Zusätzlich setzen die Unit-Tests den Test-First-Ansatz um und zeigen, wie sich Fehler effektiv vermeiden lassen. So halten Pascal-Sets auch in .NET Einzug. [jpl]

Tabelle 1

Liste der Operatoren der Set-Klasse.

Operator	Methode	Beschreibung
A + B	Join	Vereint die Elemente zweier Mengen.
A - B	Minus	Gibt die Differenzmenge zweier Sets zurück, enthält also alle Elemente, die in A, aber nicht in B vorhanden sind.
A * B	Intersect	Ergibt die Schnittmenge zweier Sets.
A < B	IsSubSetOf	Ermittelt eine Teilmenge und ob alle Elemente von A in B enthalten sind.
A <= B	IsStrictSubSetOf	Prüft auf eine echte Teilmenge: ob alle Elemente von A in B enthalten sind, A aber nicht B ist.
A > B	IsSupersetOf	Untersucht, ob alle Elemente von B in A enthalten sind, A also eine Obermenge von B bildet.
A >= B	IsStrictSupersetOf	Prüft, ob alle Elemente von B in A enthalten sind und B gleichzeitig nicht A ist (echte Obermenge).
~A	Complement	Das Ergebnis enthält alle Elemente, die nicht in A enthalten sind.
A ^ B	XOR	Ermittelt alle Elemente, die in A, aber nicht in B enthalten sind, und die Elemente, die in B, aber nicht in A enthalten sind.
A == B	Equals	Prüft die Mengen auf Gleichheit.
A != B	!Equals	Prüft die Mengen auf Ungleichheit.

- [1] CodeGear RAD Studio 2009, [dnplink SL0905Sets2](#)
- [2] RemObjects Oxygene, [dnplink SL0905Sets3](#)
- [3] Red Gate, .NET Reflector (ehemals Lutz Röder), [dnplink SL0905Sets1](#)
- [4] NCover Code Coverage, www.ncover.com

Sie sind der Boss!

Hier programmiert der Chef!

dotnetpro.shirts bestellen unter www.dotnetpro.de