Sven Vintges is software-engineer bij Atos Origin

Minder fouten en betere code dankzij generics

OPBOUW STANDAARD LIBRARY IN ORGANISATIE STAP DICHTERBIJ

Microsoft heeft in zijn nieuwe versie van de .NET-programmeertaal een aantal nieuwe taalconstructies geïmplementeerd, waaronder partial classes, anonymous types en lambda-functies. Eén van de fundamentele wijzigingen is generics. In dit artikel wordt deze nieuwe taalconstructie nader toegelicht en wordt de meerwaarde ten aanzien van de code en de toepasbaarheid voor de codebase besproken. Generics bieden de mogelijkheid code te maken met minder fouten en betere kwaliteit.

In .NET Framework 1.1 en de andere voorlopers van .NET Framework 2.0 waren er twee manieren om datastructuren (linked lists, stacks, queues) te implementeren. Een veelgebruikte oplossing is een specialisatie naar het specifieke datatype. In deze oplossing wordt gebruikgemaakt van de class definition van het type dat in de datastructuur wordt opgeslagen. Een veelvoorkomende situatie is dat er een standaard datastructuur wordt overgeërfd. Zo kan er een afgeleide van de datastructuur List worden gemaakt, waarbij door alle methoden waarin het datatype Object wordt gehanteerd, gebruik wordt gemaakt van het specifieke type. Zie codevoorbeeld 1.

Deze oplossing heeft tot gevolg dat de datastructuur voor elk benodigd type geïmplementeerd wordt. In plaats van gebruik te maken van het Object-type wordt het specifieke type gebruikt, bijvoorbeeld String, Int32, enzovoort. Het resultaat is een grote codebase, weinig herbruikbare code en foutgevoeligheid (door copy paste-acties) of het opnieuw implementeren van algoritmen. Een andere, vaak toegepaste oplossing is het gebruik van een generiek datatype. Dit is een oplossing die in het .NET Framework 1.1 is gekozen. In het .NET Framework 1.1 is dit bijvoorbeeld de List. Bij het gebruik van generieke typen bestaat er geen type-safety. Generieke typen accepteren alles van het type Object en wat daarvan afstamt. In het .NET Framework is het type Object het basistype van alle andere typen. Dit betekent dat alle typen in de datastructuur kunnen worden gezet. Hierdoor kan de compiler de programmeur van minder informatie voorzien over eventuele fouten door gebruik van verkeerde typen. Bij het opvragen van een object uit de datastructuur dient een cast plaats te vinden naar het datatype dat gewenst is. Bij casting naar een ander type dan erin is gestopt, mislukt de cast met als gevolg een run-time foutmelding.

Introductie generics

Generics bieden ontwikkelaars de mogelijkheid typen te definiëren waarvoor bepaalde details nog niet zijn ingevuld. Deze details worden later, tijdens de declaratie, ingevuld. Tijdens de definitie van generics wordt het exacte type (de details) open gelaten. Zo kan er een List<T> (List of T) gedefinieerd worden. Deze List is standaard nog niet voorzien van een exact type. Tijdens de instantiëring van dit datatype wordt het type opgegeven, bijvoorbeeld door de regel List<Persoon>. Op deze manier wordt een 'List of Persoon' gedefinieerd. De initiatie van deze list wordt het 'constructed type' van de generic List of T genoemd. Het voordeel hiervan is dat de List met objecten van het type 'Persoon' kan werken. In .NET Framework 1.1 werkt een lijst met objecten van het basistype 'Object'. Hierdoor dient bij het opvragen altijd een expliciete cast naar het verwachte type te worden uitgevoerd. Een voorbeeld waarin dit gebeurt, is te zien in codevoorbeeld 2. In dit voorbeeld wordt bij het gebruik van lijsten met een basistype het probleem uit de inleiding geschetst. Doordat de compiler niets kan afdwingen over het type treedt er niet compile-time, maar run-time een fout op. Dit komt omdat er tijdens het compileren niet bepaald kan worden welk type object in de lijst wordt gezet of is gezet. De applicatie probeert run-time het object te casten. Doordat het object niet van het type is waarnaar gecast moet worden, treedt er een 'InvalidCastException' op.

Het opslaan van data in het basistype Object brengt als gevaar met zich mee dat er een onverwacht objecttype in de datastructuur kan worden gezet. Dit is het gevolg van het feit dat de datastructuur niet type-safe is. De compiler kan niets afdwingen over de typen in de lijst. Doordat er op compilerniveau niets wordt afgedwongen, zal dit run-time pas worden ontdekt. Deze fouten kunnen onverwacht en ongecontroleerd optreden. De meeste applicaties voorzien niet in een mechaniek om deze problemen te onderkennen. Dit kan worden ondervangen door gebruik te maken van reflection. Over het algemeen is dit een vrij dure operatie, helemaal in vergelijking met het afdwingen van het type door de compiler. In ernstige gevallen kan er datacorruptie of een productieverstoring optreden met eventuele langdurige en kostbare analyses tot gevolg.

Compiler

Met behulp van generics kan de compiler tijdens het compileren al bepalen of de ingevoerde objecten van een bepaald type zijn en de ontwikkelaar hierover een duidelijke melding geven, zodat hij het probleem kan oplossen. Dit kan doordat tijdens de declaratie van de datastructuur de vrije parameter een type krijgt. De compiler controleert of de opgegeven objecten voldoen aan de parameter die is opgegeven. In codevoorbeeld 3 wordt een implementatie van een generic getoond.

Het .NET Framework kent twee soorten datatypen: het value-type en het reference-type. Een value-type is een elementair type, bijvoorbeeld een int, double, float, enzovoort. Dit basistype staat gedurende het draaien van de applicatie op de stack van de machine. Bij elke

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;


namespace Codevoorbeelden
{
  class MySpecificObject
  {
    // implementatie van de class
  }

  class SpecializedList
  {
    ArrayList myList;

    SpecializedList()
    {
      myList = new ArrayList();
    }

    public void Add(MySpecificObject newObj)
    {
      myList.Add(newObj);
    }

    public MySpecificObject Get(int index)
    {
      return (MySpecificObject)myList[index];
    }
  }
}
```

Codevoorbeeld 1.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Windows.Forms;
using System.Text;

namespace Codevoorbeelden
{
  class Codevoorbeeld2
  {
    public void SomeAction()
    {
      ArrayList lCastList = new ArrayList();

      lCastList.Add(0);
      lCastList.Add(new TextBox());
      lCastList.Add("String");

      //De volgende regel code zal geen fout opleveren
      //op positie 0 staat een integer.
      int i = (int)lCastList[0];

      //De volgende regel code zal zorgdragen
      //voor een InvalidCastException
      int k = (int)lCastList[1];

      //De volgende regel code zal zowel door de compiler als
      // de run-time worden geaccepteerd
      String l = (string)lCastList[2];
    }
  }
}
```

Codevoorbeeld 2

functieaanroep wordt er een kopie van dit type op de stack geplaatst. De manipulaties op deze typen vinden enkel in de scope van de functie plaats (op de eigen stack entry). Het is wel mogelijk een type als reference-type door te geven door gebruik te maken van het 'ref' keyword. De reference-typen worden, in tegenstelling tot de value-typen, op de heap bewaard. Reference-typen zijn de objecten (instantiëring van classes) van de applicatie. Een reference-type wordt als een referentie aan een functie meegegeven. Dit betekent dat bij wijzigingen op het object deze voor de hele applicatie zijn gewijzigd. Ieder value-type heeft een corresponderend reference-type (value-type int heeft reference-type Int32). Het proces van omzetten van een value-type naar een reference-type wordt boxing genoemd. Andersom kan een reference-type weer worden omgezet naar het corresponderende value-type. Dit proces wordt unboxing genoemd.

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using System.Text;

namespace Codevoorbeelden
{
  class Codevoorbeeld3
  {
    public static void DoIet()
    {
      List<int> lIntlist = new List<int>();

      //Onderstaand statement wordt door de compiler geaccepteerd
      lIntlist.Add(0);

      // De volgende regel geeft een compiler error Argument 1 cannot
      // convert from 'System.Windows.Form.Textbox' to int
      lIntlist.Add(new TextBox());

      //De volgende regel kan zonder cast
      int j = lIntlist[0];
    }
  }
}
```

Codevoorbeeld 3.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Codevoorbeelden
{
  // T is de vrije parameter
  class Codevoorbeeld4<T>
  {
    private T codevoorbeeld4;
    // De volgende methode verwacht een parameter van het type T
    public void VoegToe(T newObj)
    {
    // implementatie van de methode
    }

    // De volgende regel leest een object van het type T uit de data
structuur
    public T LeesObject()
    {
      return codevoorbeeld4;
    }

   }
}
```

Codevoorbeeld 4.

Generic class

In de volgende paragrafen worden de vijf generic constructies besproken, inclusief de bijbehorende syntax. De eerste, meest gebruikte, generic is de generic class. Een generic class is, zoals eerder besproken, een class-definitie die een parameter vereist tijdens de instantiëring. In codevoorbeeld 4 zien we een definitie van een generic class.

Volgens de definitie van C# 2.0 is de syntax van een generic class de volgende:

```
attributes(opt) class-modifiers(opt) class identifier type-parameter-list(opt) class-base(opt)
    type-parameter-constraints-clauses(opt) class-body ;(opt)
```

Generic method

Een generic method kan een onderdeel zijn van een struct, een class of een interface. Een generic method is - wat deze functionaliteit betreft - dus vergelijkbaar met een normale method. Deze classes, structs of interfaces kunnen generic én niet generic zijn, een generic method kan dus in ieder type class, struct of interface worden gebruikt.
Een generic method kun je declareren door een type parameter achter de naam van de generic method te plaatsen. Volgens de C# 2.0-syntax ziet dit er als volgt uit:

```
generic-method-declaration:
generic-method-header method-body
```

Waarbij generic-method-header en method-body de volgende definities voorstellen:

```
generic-method-header:
attributes(opt) method-modifiersopt return-type member-name type-parameter-list
    ( formal-parameter-list(opt) ) type-parameter-constraints-clauses(opt)
```

```
interface-generic-method-declaration:
attributes(opt) new(opt) return-type identifier type-parameter-list
    ( formal-parameter-list(opt) ) type-parameter-constraints-clauses(opt) ;
```

Generic struct

Een generic struct is een struct waarbij één of meer optionele parameters kunnen worden toegevoegd. Zie codevoorbeeld 6 voor een declaratie van een generic struct en het gebruik hiervan.

Volgens de C#-specificatie is de syntax van een generic struct de volgende:

```
attributes(opt) struct-modifiers(opt) struct identifier type-parame
ter-list(opt) struct-interfaces(opt) type-parameter-constraints-clauses(opt) struct-body ;opt
```

Generic delegates

Generic delegates zijn een van de hoekstenen van C# 2.0. De compiler maakt onder water veel gebruik van deze constructie, waarbij een delegate één of meer parameters krijgt. De werking van een generic delegate is hetzelfde als een normale delegate. Het grote verschil ligt in de vrije parameter van de definitie. In codevoorbeeld 7 is een definitie van een generic delegate te zien.

Volgens de specificatie is de formele syntax:

```
attributes(opt) delegate-modifiers(opt) delegate return-type identifier
type-parameter-list(opt)
    ( formal-parameter-list(opt) ) type-parameter-constraints-clauses(opt) ;
```

Generic interface

C# 2.0 biedt de mogelijkheid van een generic interface. In codevoorbeeld 8 zien we een definitie van een generic interface en het gebruik van een generic interface tijdens de declaratie van een class.
De formele syntax van een generic interface is de volgende:

```
attributes(opt) interface-modifiers(opt) interface identifier type-parameter-
list(opt)
    interface-base(opt) type-parameter-constraints-clauses(opt) interface-body ;(opt)
```

Mogelijkheden en restricties van generics

Generics kun je bij verscheidene taalconstructies gebruiken. Een generic kan één of meer 'vrije' typen bevatten. Doorgaans wordt voor deze typen de letter T en daaropvolgende gebruikt, maar iedere willekeurige reeks kan gebruikt worden. In codevoorbeeld 9 staat de syntax van het definiëren van een generic class. Hierin zijn twee vrije waarden opgegeven, U en V. De parameter V moet een afgeleide zijn van parameter U.
Tijdens het definiëren van een generic kunnen er restricties worden gesteld aan de eigenschappen van het type dat uiteindelijk wordt toegekend aan de generic. Dit kan door het 'WHERE'-keyword. Hiermee kun je afdwingen dat het type voldoet aan bijvoorbeeld een bepaalde interface. Op deze manier is het mogelijk de standaard veel gebruikte datastructuren type-safe te implementeren. Hierbij kun je denken aan een SortedList. Deze datastructuur bevat objecten die op een bepaalde manier gesorteerd zijn. Met generics

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Codevoorbeelden
{
  class Codevoorbeeld5
  {
    public void Copy<T>(T Bron, T Doel)
    {
    // implementatie van de methode
    }
  }
}
```

Codevoorbeeld 5

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Codevoorbeelden
{
  public struct Codevoorbeeld6<T,U>
  {
    public readonly T Item1;
    public readonly U Item2;

    public Codevoorbeeld6(T item1, U item2)
    {
      this.Item1 = item1;
      this.Item2 = item2;
    }
  }
}
```

Codevoorbeeld 6

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Codevoorbeelden
{
  public delegate T genericDelegate<T>(T param1);

  class Codevoorbeeld7
  {
  // implementatie van de class
  }
}
```

Codevoorbeeld 7