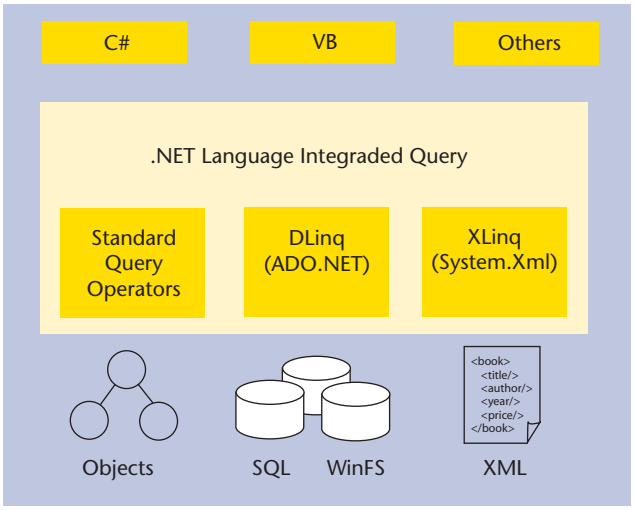
Onlangs werd op de Microsoft PDC 2005 door de .NET-architecten Anders Hejlsberg en Don Box het LINQ (Language Integrated Query) project gepresenteerd. LINQ biedt onder meer een oplossing voor de flinke kloof die er bestaat tussen programmeertalen en databases. In dit artikel gaat Willem Koppenol in op de achtergronden van LINQ en licht aan de hand van voorbeeldcode in C# de werking van LINQ in de praktijk toe.

thema

Het LINQ-project

Uniform programmeermodel in de .NET-omgeving

LINQ is een nieuwe faciliteit die aan de .NET programmeertalen zoals C# en VB.NET zal worden toegevoegd. Het voorziet in een reeks in de taal opgenomen query-expressies en operatoren voor het benaderen van verschillende vormen van data. LINQ biedt daarbij een uniform programmeermodel waarbij het er niet toe doet of de data zich nu bevinden in relationele databases, XML-bestanden of onderdeel is van objecten. LINQ wordt mogelijk gemaakt door een reeks taal innovaties waaronder extension methoden, lambda-expressies en expression-trees. Subprojecten van LINQ zijn DLinq voor database-data en XLinq voor XML-data.



FIGUUR 1. Overzicht van het Linq project.

DE KLOOF Momenteel kun je vanuit een programmeertaal zoals C# weliswaar een query formuleren en meegeven aan de C# class constructor of methode, maar je moet je dan buiten de C# taal begeven en de heel andere wereld van SQL binnenstappen. Typische voorbeeldcode is :

```
SqlCommand cmd;
cmd = New SqlCommand("SELECT * FROM Customers
                      WHERE Country =
@Country
                      AND City = @City",
conn);
```

Opvallend hieraan is dat je twee talen door elkaar heen gebruikt en de SQL query-expressie tussen quotes formuleert. De C# compiler begrijpt niets van SQL en kan dus geen compiletime-typechecking doen. Eventuele fouten in de query zullen zich pas runtime openbaren. Als gevolg hiervan heb je in een visuele omgeving ook geen IntelliSense voor de query.

De relationele wereld van SQL verschilt op nog veel meer punten van het objectgeoriënteerde domein. De gebruikte datatypes verschillen en ook de data zelf worden anders voorgesteld. In de relationele wereld worden data gerepresenteerd door rijen in een tabel, de objectgeoriënteerde wereld gebruikt objecten. Objecten hebben een unieke identiteit met een eigen fysieke locatie en toestandsvariabelen en met mogelijke referenties naar andere objecten. Rijen worden geïdentificeerd door primaire sleutel waarden en hebben mogelijk een link met rijen in een andere tabel door een vreemde sleutel.

GEÏNTEGREERDE QUERY'S In LINQ zijn een reeks standaard query operatoren, waaronder Where, OrderBy en Select, aan programmeertalen zoals VB.NET en C# toegevoegd. Deze operatoren worden gebruikt om zogeheten query expressies mee te initialiseren. Query's

worden door LINQ een integraal onderdeel van de programmeertalen. Query-expressies gaan profiteren van zaken als compiletime-syntaxchecking, statische typering, metadata en IntelliSense, die ze tot nu toe moesten ontberen. LINQ biedt algemene query-faciliteiten, die dus niet specifiek zijn voor relationele of XML-data. En ze werken ook voor interne data in het geheugen. De standaard query-operatoren kunnen door derde partijen worden vervangen door eigen geoptimaliseerde implementaties. Ook domeinspecifieke query-operatoren kunnen worden toegevoegd. De uitbreidbaarheid van de query-architectuur wordt binnen LINQ zelf gebruikt door te voorzien in operatoren die werken op zowel XML- als SQL-data.

Functie	Query Operator
Restriction	Where
Projection	Select, SelectMany
Ordering	OrderBy, ThenBy
Grouping	GroupBy
Quantifiers	Any, All
Partitioning	Take, Skip, TakeWhile, SkipWhile
Sets	Distinct, Union, Intersect, Except
Elements	First, FirstOrDefault, ElementAt
Aggregation	Count, Sum, Min, Max, Average
Conversion	ToArray, ToList, ToDictionary
Casting	OfType<T>

FIGUUR 2. Lijst met Query Operatoren

IENUMERATOR De standaard query-operatoren kunnen worden toegepast op iedere informatiebron die het `IEnumerable<T>` interface implementeert. Het `IEnumerable<T>` interface heeft een `GetEnumerator()` methode die een `IEnumerator<T>` retourneert. Een IEnumerator kan gebruikt worden om over een collectie te itereren en bevat de volgende methoden:

```
interface IEnumerator {
Object Current();    // Gets current element
boolean MoveNext(); // Advances to next element
void Reset();        // Sets enumerator to
its initial position
}
```

Enumerators kunnen alleen gebruikt worden om data in een collectie te lezen. Een `IEnumerator` is aanvankelijk net voor het eerste element van een collectie gepositioneerd. Door `Reset` wordt hij op deze positie teruggezet. Alvorens je het eerste element van een collectie met `Current` kunt ophalen, moet de `IEnumerator` eerst met `MoveNext` een positie worden opgeschoven.

QUERY SYNTAX Een C# 3.0 programma dat de standaard query-operatoren gebruikt om een array van namen te bevragen, luidt:

```
class SimpleQuery {
  static void Main() {
    string[] carnames = { “Ferrari”, “Fiat”,
“Lamborghini”,
                          “Masarati”, “Alfa
Romeo”};

    IEnumerable<string> expr = from s in car-
names
                               where
s.Length == 4
                               orderby s
                               select s;

    foreach (string item in expr)
      Console.WriteLine(item);
  }
}
```

Met als output:

```
Fiat
```

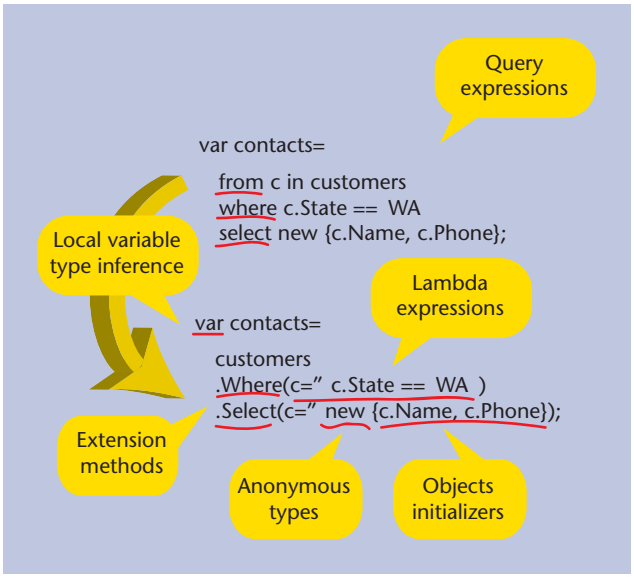
De voorbeeldcode gebruikt query-syntax die net als het `foreach` statement een shortcut is voor code die ook uitgeschreven kan worden. In feite gebeurt op de achtergrond het volgende:

```
IEnumerable<string> expr =
    carnames.Where(s => s.Length ==
4).OrderBy(s => s).Select(s => s);
```

De argumenten van de `Where, OrderBy` en `Select` operatoren zijn hier zogeheten lambda-expressies. Zij maken het mogelijk dat de standaard query-operatoren als methoden worden gedefinieerd en met dot-notatie worden samengevoegd. Ook opvallend is de volgorde van de query-operatoren. Eerst `from`, dan `where` en tenslotte `select.` Dit is afwijkend van de operatorvolgorde die we gewend zijn bij SQL query's. Als je er echter over nadenkt, correspondeert de LINQ-query volgorde meer met een werkelijke volgorde van acties die moeten plaatsvinden.

TAAL UITBREIDINGEN C# 3.0 Zoals we in het voorbeeld net zagen zijn het LINQ-project en de query-operatoren gebaseerd op taalinnovaties in C# 3.0 en VB.NET 9. Deze taalinnovaties vormen een coherent onderdeel van het LINQ-project en betreffen extension-methoden, impliciet getypeerde locale variabelen, lambda-expressies, expression-trees, object en collection initializers, anonymous types en query-expressies.

Zij maken op hun beurt weer veel gebruik van de generic types (geparameteriseerde types) die in .NET 2.0 zijn geïntroduceerd. Voor een goed begrip van het LINQ-project is het nodig de belangrijkste van deze taaluitbreidingen nu nader te beschouwen.



FIGUUR 3. Taal innovaties in C# 3.0

EXTENSION METHODEN Extension methoden zijn een belangrijk onderdeel van de query-architectuur. Met extension methoden kunnen ontwikkelaars de functionaliteit van bestaande types uitbreiden met nieuwe methoden. Deze methoden kunnen vervolgens via de standaard-syntax worden aangeroepen. Extension-methoden worden gedefinieerd als statische methoden in statische classes. In C# herken je extension methoden aan het keyword `this` bij de eerste parameter van de methode. Een voorbeeld is de eenvoudigste query-operator `Where` :

```
namespace System.Query {
  using System;
  using System.Collections.Generic;

  public static class Sequence {
    public static IEnumerable<T> Where<T>(
              this IEnumerable<T> source,
                   Func<T, bool> predicate)
{

      foreach (T item in source)
        if (predicate(item))
          yield return item;
    }
  }
}
```

Het type van de eerste parameter van de extensie-methode geeft aan op welk type de extensie betrekking heeft. De `Where` extension-methode is dus een uitbreiding op het type `IEnumerable<T>`. We kunnen `Where` net als iedere andere statische methode direct aanroepen:

```
IEnumerable<string> expr =
     Sequence.Where(carnames, s => s.Length
== 4);
```

Wat extension-methoden echter bijzonder maakt is dat ze ook kunnen worden aangeroepen met instance-syntax:

```
IEnumerable<string> expr =
     carnames.Where(s => s.Length == 4);
```

Extension-methoden die in de statische classes van een namespace zijn gedefinieerd komen in scope als die namespace wordt geïmporteerd (in C# met het `using` statement). De standaard query-operatoren zijn extensie-methoden in `System.Query.Sequence`. Ze zijn vrijwel allen gedefinieerd in termen van het `IEnumerable<T>` interface. Iedere datastructuur die `IEnumerable<T>` implementeert, krijgt dus de beschikking over de standaard query-operatoren door de `System.Query` namespace te importeren. Desgewenst kun je de standaard query-operatoren voor een specific type vervangen door je eigen methoden met dezelfde naam en een compatibele parameterlijst te definiëren.

IMPLICIET GETYPEERDE VARIABELEN Impliciet getypeerde variabelen ontlasten de programmeur ervan om het type van een variabele bij declaratie vast te leggen. Het type van een variabele wordt in plaats daarvan door de compiler afgeleid uit de initialisatie-expressie. Impliciet getypeerde variabelen worden veel gebruikt bij LINQ query-expressies. Impliciete typering vrijwaart de programmeur ervan alle types die een query zou kunnen retourneren expliciet te definiëren, waardoor een hoop rompslomp wordt voorkomen. Het `var` keyword kenmerkt de impliciete variabele declaratie:

```
List<int> numbers = new List<int>();  //
expliciete declaratie
var numbers= new List<int>();          //
impliciete declaratie
```

Bedenk dat dit iets anders is dan de ongetypeerde variabelen zoals we die wel kennen uit scripttalen. Dergelijke variabelen kunnen gedurende hun levensduur verschillende types bevatten hetgeen niet kan bij impliciet getypeerde variabelen.

LAMBDA-EXPRESSIES Een ander belangrijk onderdeel van de query architectuur zijn de lambda-expres-

sies. Lambda-expressies zijn een evolutie van de anonieme methoden die in C# 2.0 zijn geïntroduceerd. Bij anonieme methoden kan op de plaats waar een delegate wordt verwacht in plaats daarvan een ter plekke een anonieme methode worden opgegeven. Zo verwacht de onderstaande `FindAll` methode een delegate parameter. De functie die wordt uitgevoerd, het retourneren van even getallen, wordt als anonieme methode ter plekke gespecificeerd:

```
List<int> evenNr = list.FindAll(delegate(int
i) { return (i%2) == 0); }
```

Lambda-expressies gebruiken een compactere, functionele programmeersyntax voor anonieme methoden. Een lambda-expressie bestaat uit een parameterlijst, gevolgd door => en afgesloten met een expressie. Bij een impliciet getypeerde parameterlijst, worden de types van de parameters afgeleid van de context waarin de lambda-expressie wordt gebruikt. Voorbeelden van lambda-expressies zijn:

```
 (int x) => x + 1                  // expli-
citly typed variables
(x,y) => return x*y               // implicitly
typed variables
```

Het equivalent van de boven gespecificeerde `FindAll` methode maar nu met impliciet getypeerde variabelen en een lambda-expressie is:

```
var evenNumbers = list.FindAll(i => (i % 2)
== 0);
```

Expression Trees Een heel vernieuwend onderdeel van de query-architectuur zijn de expression-trees. Door expression-trees kunnen ontwikkelaars met lambda-expressies werken zoals met data. Het nieuwe type, `Expression<T>` , stelt een in memory representatie voor van een lambda-expressie. Ter verduidelijking beschouwen we de volgende initialisatiecode:

```
Func<int, bool> nonLambdaExpr = x => (x & 1)
== 0;
Expression<Func<int, bool>> lambdaExpr = x =>
(x & 1) == 0;
```

De eerste regel gebruikt lambda-syntax om een delegate te initialiseren. Het generieke `Func` delegate-type is een onderdeel van LINQ en is simpelweg een handige delegate dat een functie met één parameter representeert:

```
public delegate T Func<T, A>(A a);
```

In de tweede regel is het `Func` delegate de type parameter voor het generieke type `Expression<T>`. De initialisatiecode die volgt is precies hetzelfde als in de eerste regel, maar het resultaat is heel verschillend. De compiler kent het `Expression<T>` type en zal de lambda-expressie vertalen tot een boomstructuur van objecten in de expressie. De lambda-expressie kan nu in code benaderd en gemodificeerd worden.

Dit onderdeel van LINQ maakt het mogelijk dat er bibliotheken worden geschreven die de basis query-abstracties gebruiken. De DLinq data access-implementatie gebruikt deze faciliteit bijvoorbeeld om expression trees te vertalen naar een SQL-statement dat aan de database wordt aangeboden.

DLinq DLinq is het onderdeel van het LINQ-project dat voorziet in de infrastructuur voor het objectgeoriënteerd benaderen van relationele data. DLinq vertaalt de in de taal geïntegreerde query's naar SQL, biedt ze aan voor uitvoering door de database en slaat de resultaten weer op in gespecificeerde objecten. De applicatie kan vervolgens de objecten manipuleren. Eventuele wijzigingen kunnen via DLinq worden opgeslagen. DLinq wordt een nieuw onderdeel wordt van ADO.NET en het is mogelijk bestaande ADO.NET oplossingen stap voor stap naar DLinq om te zetten.

Entity classes DLinq kent het concept entity classes. Dit zijn classes die worden geassocieerd met een tabel in de database. Attributen geven aan met welke tabel en kolommen de class en velden in de class corresponderen:

```
[Table(Name="Cars")]
public class Car {
  [Column(Id=true)]
public string CarID;
[Column]
  public string Brand;
[Column]
  public double Price;
}
```

Alleen objecten van classes die zijn gedeclareerd met het `[Table]` attribuut kunnen in de database worden opgeslagen. En alleen velden die als kolommen voorzien zijn van het `[Column]` attribuut worden opgehaald of opgeslagen. De anderen worden als transient beschouwd.

DataContext De `DataContext` class in DLinq vertaalt de geïntegreerde query's in SQL query's voor de database en construeert objecten uit de resultaten. De `DataContext` wordt net als een ADO-connection geïnitialiseerd met een connectiestring. De DataContext class maak geïntegreerde query's mogelijk door te voor-

zien in een implementatie van de standaard query-operatoren zoals `Where()` en `Select()`. In praktijk zul je vaak werken met een sterk getypeerde afgeleide `DataContext` waarin de tabellen van de betreffende database expliciet zijn opgenomen:

```
public partial class CarDB : DataContext
{
  public Table<Car> Cars;
  public CarDB(string connection):
base(connection) {}
}
```

Een query voor auto's van het merk Mercedes ziet er dan als volgt uit:

```
CarDB db = new CarDB("c:\\cars.mdf");
var q = from c in db.Cars
    where c.Brand == "Mercedes"
    select c;
foreach (var car in q)
  Console.WriteLine("id = {0}, Car = {1},
Price = {2}",
                          car.CarID, car.
Brand, car.Price);
```

DATA OPSLAG DLinq is niet alleen geschikt voor het opvragen van data maar ook voor het wijzigen van data. Objecten van entity-classes kunnen net als gewone objecten in de applicatie worden benaderd, veranderd of in collecties worden opgenomen. DLinq houdt alle wijzigingen bij en staat klaar de wijzigingen in de database op te slaan zodra het daartoe de opdracht krijgt. In het onderstaande voorbeeld wordt een auto opgezocht via zijn `CarID`, wordt de prijs gewijzigd en worden de wijzigingen opgeslagen:

```
CarDB db = new CarDB("c:\\cars.mdf");
string id = "101";            // Query for a
specific car
var car = db.Cars.First(c => c.CarID == id);
car.Price = 99.99;            // Change the
price
db.SubmitChanges();           // Ask
DataContext to save changes
```

Bij de aanroep van `SubmitChanges()` genereert DLinq automatisch de benodigde SQL-commando's om de wijzigingen op te slaan en voert deze uit. Desgewenst kan voor het opslaan van de wijzigingen ook een stored procedure worden aangesproken.

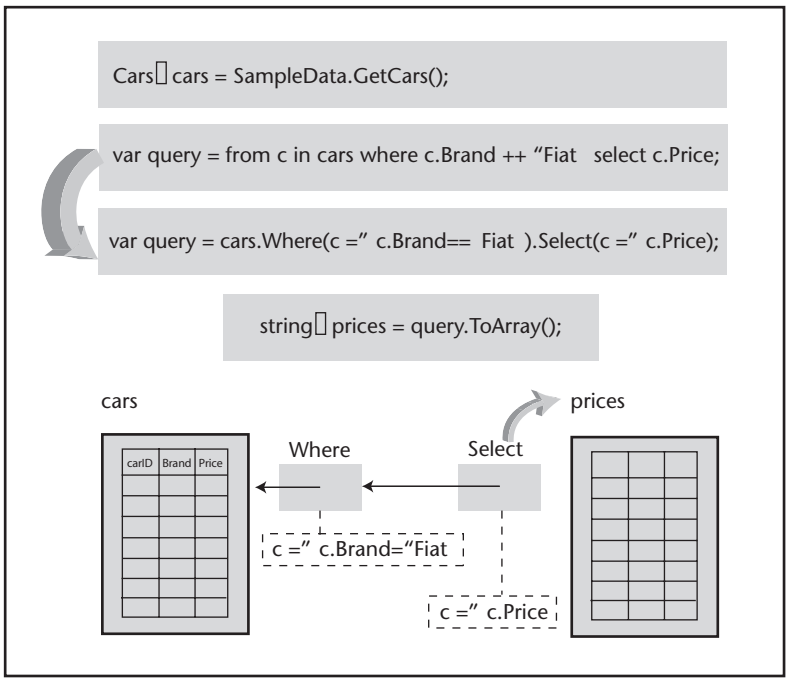
DEFERRED EXECUTION Wanneer de code voor een query-expressie wordt uitgevoerd, betekent dat niet dat de query zelf wordt uitgevoerd. Een query-expressie is een beschrijving van de query. In onderstaande query-expressie refereert de variabele `q` aan een beschrijving van de query, niet aan het resultaat van het uitvoeren ervan.

```
var q = from c in db.Cars
    where c.Brand == "Mercedes"
    select c;
```

Het echte type van `q` is hier `Query<Car>`. Een `Query` object kun je vergelijken met een ADO command-object. Een command object is de houder van een string die een query beschrijft. Op dezelfde manier is een `Query` object de houder van een expression tree. Een expression tree is, zoals eerder beschreven, een in memory boomstructuur bestaande uit objecten in een lambda expressie. En net als een command object een `ExecuteReader()` methode heeft die de query uitvoert en een `DataReader` retourneert, heeft een Query object een `GetEnumerator()` methode die de resultaten teruggeeft als een `IEnumerator<T>`. De query wordt pas echt uitgevoerd als de applicatie vraagt de resultaten van de query op te sommen. Dit gebeurt bij het foreach statement.

```
foreach (Car c in q)
  Console.WriteLine(c.Price);
```

Dit betekent ook dat de query twee keer wordt uitgevoerd als je de resultaten twee keer opsomt in een `foreach`. Dit is natuurlijk niet erg efficiënt en om dit te voorkomen kun je de resultaten van de uitvoering van de query opslaan in een collectie. Dit beschreven gedrag staat bekend onder de naam deferred execution.



FIGUUR 4. Deferred Query Executie

XLinq XLinq is het onderdeel van het LINQ-project dat gericht is op XML-data. XLinq heeft de geïntegreerde query-faciliteiten die kenmerkend zijn voor het LINQ-project, maar kan ook gezien worden als een gemoderniseerde in-memory API voor het programmatisch benaderen van XML-data. XLinq belooft dezelfde functionaliteit te brengen als bestaande XML-programmeerinterfaces zoals de DOM API en XQuery/XPath, maar is lichter van gewicht en maakt het XML-programmeren eenvoudiger. Voorts gebruikt XLinq moderne taaleigenschappen zoals generics en sluit aan bij de LINQ-patronen die in het .NET Framework worden geïntegreerd.

XML Programmeren Aangezien XLinq een volledige in-memory XML Programmeer API is, kun je met XLinq alle dingen doen die je zou verwachten benaderen van XML. Je kunt dus met XLinq een XML-bestand inlezen of bewaren, een XML-document vanuit het niets opbouwen en elementen toevoegen en verwijderen. XLinq ondersteunt daarbij voor het opbouwen van een XML-boomstructuur de zogeheten functionele constructie, die te zien is in onderstaand codefragment:

```
XElement cars =
   new XElement("cars",
  new XElement("car",
     new XElement("brand", "citroën",
             new XAttribute("color",
"blue")),
     new XElement("price", "99.99")
  )
   );
```

Door het inspringen en nesten van aanroepen om XML-elementen en attributen aan te maken wordt de structuur van de onderliggende XML-boom zichtbaar. XLinq kent nog vele andere constructies die het omgaan met XML-data vergemakkelijken, zoals de directe creatie van elementen (zonder Document-object) en een vereenvoudigde omgang met prefixes en namespaces. Het voert te ver om hier nu nader op in te gaan.

XML Query's Het voornaamste verschil tussen XLinq en andere XML programmeer-API's zijn natuurlijk de in de taal geïntegreerde query's. XLinq implementeert in ieder geval de Standard LINQ Query Operators en geeft op verschillende plaatsen collecties terug die het `IEnumerable<T>` interface implementeren. Om een indruk te krijgen hoe query's er in XLinq uitzien beschouwen we de volgende XML-data:

```
<cars>
    <car>
        <brand>Mercedes</brand>
    </car>
    <car>
        <brand>Fiat</brand>
    </car>
</cars>
```

Deze XML-data kunnen we transformeren en de `<brand>` elementen direct onder het `<cars>` element plaatsen door de volgende query:

```
new XElement("cars",
  from c in cars.Elements("car")
  select new object[] {
     new XComment("car"),
     new XElement("brand", (string)c.
Element("brand"))
  }
);
```

Een array-initializer wordt gebruikt om een reeks child-elementen te creëren die direct onder het `<cars>` element worden geplaatst. Het resultaat is:

```
<cars>
    <!—car -->
    <brand>Mercedes</brand>
    <!—car -->
    <brand>Fiat</brand>
</cars>
```

Slotwoord Het LINQ- project biedt een uniform programmeermodel in de .NET omgeving voor het benaderen van in-memory, database en XML-data. Query-faciliteiten worden geïntegreerd in de .NET-talen. De grondslag van LINQ wordt gevormd door een reeks taalinnovaties in C# 3.0 en Visual Basic 9.0. De impact van LINQ is naar verwachting aanzienlijk. Het subproject DLinq is een belangrijke stap vooruit in het dichten van de kloof tussen de objectgeoriënteerde .NET-applicatie wereld en de relationeel georiënteerde databasewereld. DLinq gaat zoals de naam doet vermoeden echter niet alleen over query's. Ook het objectpersistentie hoort erbij. Het subproject XLinq is zelfs zonder de geïntegreerde query faciliteiten een flinke stap vooruit in het vanuit .NET benaderen van XML-data.

drs. Willem Koppenol is senior trainer en productspecialist software development bij Twice IT Training (e-mail: wkoppenol@twice.nl).