Code Contract: bewijzen dat je software werkt!

Waardevolle toevoeging .NET-ontwikkelplatform

Pieter-Joost van de Sande

Contracten spelen een belangrijke rol in onze samenleving. Hoogstwaarschijnlijk heb je een contract met je werkgever, of als je voor jezelf werkt bijvoorbeeld met een opdrachtgever. Ook heb je waarschijnlijk een contract met je verzekeraar, energieleverancier en andere partijen. Zonder deze contracten zou onze samenleving een grote chaos worden. Partijen zouden niet meer weten wat ze kunnen verwachten en van wie. Met een contract kun je valideren of één of meerdere partijen zich wel aan afspraken houden. Zou het niet geweldig zijn als we dit ook kunnen toepassen in de wereld van softwareontwikkeling?

Moderne talen als C# stellen ons in staat uitgebreide klassedefinities te maken. Uit de interface van deze definities kunnen we vaak in één oogopslag veel opmaken. Als we kijken naar de volgende publieke klasse interface, waar we, ondanks dat er geen implementatie zichtbaar is, al een hoop uit kunnen opmaken.

```
public class StringQueue
{
    public int Count
    {
        get;
    }

    public int Capacity
    {
        get;
    }

    public void Enqueue(String value);

    public String Dequeue();
}
```

Zo is de naam van deze klasse StringQueue, heeft het twee publieke eigenschappen Count en Capacity, en twee publieke methodes Enqueue en Dequeue, waarbij de methode Enqueue een parameter verwacht van het type String. De Dequeue methode verwacht geen parameters, maar de aanroepende partij kan van deze methode wel een geretourneerde waarde van het type String verwachten.
De compiler helpt ook bij het valideren van deze definities en bijbehorende implementatie. Zo kan er geen int waarde meegeven worden aan de Enqueue methode en kan het resultaat van de Dequeue methode niet toegewezen worden aan een float variable. Gebeurt dit toch, dan zal de compiler dit identificeren en één of meerdere fouten tonen. Hiermee kan de volledige applicatie gevalideerd worden en kunnen fouten opgespoord worden voordat de code daadwerkelijk is uitgevoerd.

Aanname

Toch zegt de definitie in het voorgaande codevoorbeeld niets over de eigenlijke waarde die bijvoorbeeld de Count eigenschap teruggeeft. Gezien het een int waarde is, ligt de waarde ergens tussen de -2.147.483.648 en 2.147.483.647. Ook geeft de definitie ons geen informatie over wat de bepalende factoren zijn voor deze waarde, hoewel het aannemelijk klinkt dat Count het aantal elementen in de queue representeert. Daarmee zouden we kunnen aannemen dat Count de waarde 0 retourneert zolang de Enqueue methode nog niet is aangeroepen en dat elke Enqueue aanroep zorgt voor het ophogen van Count, waarbij de Dequeue methode het tegenovergestelde doet.

Maar wat als de Dequeue methode wordt aangeroepen als de queue leeg is? Krijgen we dan een null waarde geretourneerd of wordt er dan een exception opgeworpen? Natuurlijk kunnen al deze zaken gespecificeerd worden in zo geheten XML-commentaar. Als dit het geval is, komt deze informatie onder andere terug in IntelliSense of kan er zelfs een volledige documentatiebibliotheek uit gegenereerd worden, zoals de MSDN library. Helaas is de softwareontwikkelaar zelf verantwoordelijk voor het lezen en naleven van deze documentatie. Voor de compiler is deze vorm te vrij om te valideren of de code aan de documentatie voldoet.

Code Contracts

Ik zou het bovenstaande natuurlijk niet omschrijven als hier geen betere oplossing voor zou zijn. De oplossing hiervoor heet Code Contract. Code Contracts is Microsofts implementatie van programming contracts, ook wel bekend als Design by Contract. Dit concept omschrijft dat ontwikkelaars in staat moeten zijn formele en verifieerbare interfaces te definiëren. Dit wordt mogelijk door naast een bestaand typesysteem drie extra conditie definities toe te voegen: pre-, post- en object invariante condities. Dit concept vindt haar oorsprong in de wiskunde. Formele specificatie, formele verificatie en Hoare Logic spelen hierbij een belangrijke rol. Eerdere platformen die dit concept implementeer-

den zijn Eiffel, JML en AsmL.
Code Contracts is op dit moment alleen nog beschikbaar als losse download en bevindt zich nog in de CTP fase. De losse download is beschikbaar voor zowel Visual Studio 2008 als Visual Studio 2010. Uiteindelijk zal Code Contracts als onderdeel van Visual Studio 2010 en in het .NET Framework 4.0 op de markt verschijnen.

Contractdefinities

Een aantal regels geldt voor alle contractdefinities. Contracten kunnen worden gedefinieerd voor een object, methode en eigenschap. Deze kunnen gedefinieerd worden op een concrete klasse, maar ook op interfaces of een abstracte klasse. Contracten mogen geen side-effects bevatten. Dit houdt in dat ze de status van een object niet mogen wijzigen.
Contracten dienen het doel om tijdens compileertijd te valideren of alle broncode aan de condities voldoet, zodat bugs al voor het uitvoeren van de code kunnen worden ontdekt. Daarom zijn contracten optioneel. Dit houdt in dat je als ontwikkelaar kunt aangeven of de contracten gecontroleerd moeten worden tijdens compileertijd en of de controle hiervan ook geïnjecteerd moet worden in het resultaat van de code, MSIL. Zo is het een gangbare keuze om contracten wel in een debug build mee te nemen, maar weg te laten in een release build. Zeker in scenario's waar contracten verder gaan dan alleen het eenvoudig controleren van parameters.

Het systeem

Om contracten te definiëren, te controleren tijdens compileertijd en te valideren tijdens het uitvoeren van een programma bestaat het Code Contract systeem uit drie hoofdonderdelen.

Het eerste onderdeel is de contract library. Hiermee kunnen contracten gedefinieerd worden door middel van de statische methodes die gedefinieerd zijn in de Contract klasse. Deze is onderdeel van de System.Diagnostics.Contracts namespace uit mscorlib.dll. Hiermee worden contracten gedefinieerd aan het begin van een methode of eigenschap definitie, als onderdeel van de signatuur. Omdat contracten gedefinieerd worden door middel van het gebruik van statische methodes op de Contract klasse is het gebruik van Code Contracts niet taalafhankelijk. Zo kan Code Contract zowel in C# als in VB.NET gebruikt worden, of in elke andere taal die het .NET platform ondersteunt en aan een aantal minimale eisen voldoet zoals het ondersteunen van generics.

Het tweede onderdeel is de binary rewriter, ccrewrite.exe. Deze tool past de MSIL instructies zo aan dat de contractvalidatie wordt toegevoegd op de plekken waar deze thuis hoort. Dit houdt in, ondanks dat de contracten aan het begin van een methode definitie geplaatst worden, dat de daadwerkelijke controle op een andere plek plaatsvindt. Ook worden contracten die bijvoorbeeld op een interface gedefinieerd zijn geïnjecteerd in de klasse die deze implementeren.

Het derde onderdeel is cccheck.exe. Deze tool is in staat code te onderzoeken of aan alle condities voldaan wordt. Dit is vergelijkbaar met het werk dat de compiler doet, deze compileert niet alleen de broncode, maar valideert dit ook.

Preconditions

Preconditions zijn condities die altijd waar moeten zijn, voordat een methode succesvol uitgevoerd kan worden. Met andere woorden: deze specificeren wat een methode verwacht van de aanroepende partij. Pas als alle preconditions waar zijn zal de methode succesvol uitgevoerd kunnen worden.
Voor het definiëren van preconditions wordt de statische Contract.Requires() methode gebruikt. De meest eenvoudige manier voor het definiëren van een preconditie is als volgt:

```
public void Enqueue(String value)
{
    Contract.Requires(value != null);
    Contract.Requires(Count < Capacity);

    // ...
}
```

Hier wordt de meest eenvoudige overload van Contract.Requires() gebruikt, waarbij er alleen een conditie gespecificeerd wordt. In dit geval is de eerste preconditie dat de parameter value geen null waarde mag bevatten. De tweede preconditie specificeert dat de waarde van Count lager moet zijn dan de waarde van Capacity.

In een preconditie kunnen parameterwaardes worden gebruikt, alsmede waardes van eigenschappen of velden die in minimaal de access modifier hebben van de methode. Is een methode dus public, dan kunnen er alleen publieke eigenschappen of velden worden gebruikt. Is deze protected, dan kunnen er alleen protected of hogere eigenschappen of velden worden gebruikt.
Naast de minimale overload van de Requires methode is er ook nog een overload beschikbaar waarin nog een string waarde meegegeven wordt als message parameter, deze waarde wordt gebruikt als boodschap als het contract verbroken wordt. Dit gedrag is vergelijkbaar met de message parameter in de Debug.Assert methode of als in veel unit testing frameworks.
Er is ook een generic methode Requires beschikbaar waarbij een exception type gespecificeerd kan worden. Deze exception zal opgeworpen worden als de conditie niet waar is.

```
public void Enqueue(String value)
{
    Contract.Requires<ArgumentNullExceptio
n>(value != null);
    Contract.Requires<InvalidOperationExce
ption>(Count < MaxSize);

    // ...
}
```

Een andere manier, is de manier waarop veel preconditions op dit moment worden gedefinieerd. Code kan gemarkeerd worden als een preconditie blok met de EndContractBlock methode. Met het gebruik van deze methode worden alle bovenstaande statements als preconditions gemarkeerd. Echter stelt dit wel een aantal eisen aan de code die als preconditie gemarkeerd moet worden. Deze code mag alleen uit simpele if-condition-throw statements bestaan.

```
public void Enqueue(String value)
{
    if(value == null) throw new
ArgumentNullException("value");
    if(Count >= Capacity) throw new Invali
dOperationException("Queue is full.");
    Contract.EndContractBlock();

    // ...
}
```

Een groot voordeel hiervan is dat we hiermee ook legacy code kunnen benutten zonder deze opnieuw te schrijven. Echter stelt dit wel de eis dat de code alleen maar uit simpelen if-conditie-throw statements bestaat. Er mogen dus bijvoorbeeld geen toewijzingen van variabele tussen zitten.

Postconditions

Postcondities zijn condities die altijd waar zijn na het succesvol uitvoeren van een methode, ze specificeren wat het resultaat is van een methodeuitvoer. Met andere woorden: wat kan een aanroepende partij van de methode verwachten. Postcondities worden net als preconditions aan het begin van een methode gedefinieerd, ze zijn im-

mers onderdeel van de methodesignatuur. Postcondities mogen ook geen status wijzigen en kunnen alleen gebruik maken van onderdelen die dezelfde visability hebben als de methode waarop ze gedefinieerd zijn.

Voor het definiëren van postcondities gebruiken we de statische Contract.Ensures(). Hier is een simpel voorbeeld:

```
public void Enqueue(String value)
{
    Contract.Ensures(Count == Contract.
OldValue<int>(Count)+1);

    // ...
}
```

In dit voorbeeld wordt de postconditie gespecificeerd met de Contract.Ensures() methode. In deze methode wordt de conditie gespecificeerd dat de waarde van Count bij het verlaten van de methode gelijk moet zijn aan de waarde van Count bij het starten van de methode plus 1. Om de oude waarde – de waarde voordat de methode werd uitgevoerd – te verkrijgen gebruiken we de Contract.OldValue<T> methode. Naast het vergelijken van oude state met de nieuwe state gebruik je in postcondities ook vaak de returnwaarde. Hiervoor gebruiken we de Contract.Result<T>() methode. Deze kan gebruikt worden om de returnwaarde op te nemen in een postconditie. Deze kunnen we bijvoorbeeld gebruiken bij de Dequeue methode:

```
public String Dequeue()
{
    Contract.Ensures(Count == Contract.
OldValue<int>(Count) - 1);
    Contract.Ensures(Contract.
Result<String>() != null);

    // ..
}
```

We kunnen out parameters ook opnemen in postcontract definities, hiervoor gebruiken we de ResultAtReturn<T>() methode.

Object Invariants

Waar pre- en postcondities alleen gelden voor een methode of een eigenschap, gelden invariante voor een volledig object. Dit betekent dat deze condities ten alle tijden waar moeten zijn, in ieder geval na de creatie van het object, bij de start en aan het einde van het uitvoeren van een publieke methode of eigenschap. Voor de buitenwereld zijn de condities dus altijd waar. Deze mogen alleen tijdelijk gebroken worden tijdens de uitvoering van een methode of eigenschap zolang deze na uitvoering maar weer waar zijn.

Object invarianten worden gedefinieerd in een protected methode die geannoteerd is met het ContractInvariantMethod attribuut. Binnen deze methode wordt de Contract.Invariant() methode gebruikt voor het specificeren van object invariant condities.

```
[ContractInvariantMethod]
protected void Invariant()
{
    Contract.Invariant(Count >= 0);
    Contract.Invariant(Count <= Capicity);
    Contract.Invariant(Capicity >= 0);
}
```

Contractvalidatie

Contracten kunnen tijdens compileertijd gevalideerd worden. Hier zorgt cccheck.exe voor. Deze tool analyseert de volledige broncode en bekijkt alle mogelijke paden. Tijdens compileertijd zal het breken van een contract afhankelijk van de instellingen resulteren in een waarschuwing of een fout. Er zal getoond worden op welke plek het contract gebroken wordt, alsmede welk contract gebroken wordt.

In het volgende voorbeeld wordt een simpele klasse Counter gedefinieerd. Deze klasse heeft twee methodes. Eén om de waarde van de eigenschap Value met 1 te verhogen en een tweede om deze waarde met 1 te verlagen. Met commentaar heb ik aangegeven op welke plekken de compiler met foutmeldingen zal komen.

```
public class Counter
{
    private int value;

    public int Value
    {
        get { return value; }
    }

    public void Increment()
    {

Contract.Ensures(Value == Contract.
OldValue<int>(Value) + 1);
        value++;

} // <-- Fout! Er is niet bewezen dat de
invariante waar is!

    public void Decrement()
    {

Contract.Ensures(Value == Contract.
OldValue<int>(Value) - 1);
        value--;

} // <-- Fout! Er is niet bewezen dat de
invariante waar is!
    [ContractInvariantMethod]
    protected void ObjectInvariante()
    {

Contract.Invariant(Value >= -100 && Value
<= 100);
    }
}
```

In dit voorbeeld is te zien dat tijdens het compileren twee fouten worden ontdekt. Zowel de Increment methode als de Decrement methode garanderen niet dat de object invariant niet geschonden wordt. Deze fout kunnen we op twee manieren oplossen. We kunnen een preconditie definiëren of het gedrag binnen de methode aanpassen. Als we de Increment methode als voorbeeld nemen, dan kunnen we hier een preconditie in defineren waarin we specificeren dat Increment alleen aangeroepen mag worden als de waarde van de eigenschap Value lager is dan 100. Hiermee verplaatsen we het probleem naar de aanroepende partij. Als we het oplossen in de methode implementatie zelf, dan kunnen we bijvoorbeeld een if constructie opnemen waarin we eerst controleren of de waarde van de eigenschap Value wel kleiner is dan 100 en als dit het geval is deze met 1 ophogen.
Maar de tool gaat verder, het analyseert volledige broncodes en vind alle mogelijke executiepaden. Dit is goed te zien in het volgende voorbeeld waar de Counter klasse gebruik wordt:

```
class Program
{
    static void Main(string[] args)
    {
        var counter = new Counter();

NeedPositiveNumber(counter.Value); // <--
Fout! Sinds Value is nu 0

        counter.Increment();

NeedPositiveNumber(counter.Value); // <--
Prima! Sinds Value is nu 1


        for (int i = 0; i < 5; i++)
        {
            counter.Increment();
        }
        for (int i = 0; i < 10; i++)
        {
            counter.Decrement();
        }


NeedPositiveNumber(counter.Value); // <--
Fout! Sinds Value is nu -4
    }

    static void NeedPositiveNumber(int
number)
    {
        Contract.Requires(number > 0);
```

```
        // ..
    }
}
```

Als we kijken naar de inhoud van de statische Program.Main() methode, dan zien we dat er verschillende fouten zijn geconstateerd. De eerste is dat de waarde van eigenschap Value direct na initialisatie van de Counter instantie wordt meegegeven aan de statische methode Program.NeedPositiveNumber(). De methode heeft als preconditie dat de parameterwaarde groter moet zijn dan 0. In dit geval geven wij indirect 0 mee, wat volgens de preconditie niet is toegestaan. Pas nadat de Counter.Increment() methode is uitgevoerd, dat als postconditie heeft dat de waarde van de eigenschap Value met 1 is opgehoogd, kan de waarde van de eigenschap Value zonder problemen worden doorgegeven aan de NeedPositiveNumber methode. Hierna worden er twee lussen uitgevoerd waarin er vijf keer Increment wordt uitgevoerd en 10 keer Decrement. Dit resulteert in het feit dat de waarde van de eigenschap Value nu -4 is. Het meegeven van deze waarde aan de NeedPositiveNumber methode resulteert dus weer in een foutmelding. Alle bovengenoemde fouten worden tijdens compileertijd al ontdekt. Tijdens executietijd resulteert het breken van een contract in een ContractException.

Conclusie

Het Code Contracts systeem is een waardevolle toevoeging aan Microsoft's ontwikkelplatform .NET. Het stelt ontwikkelaars in staat, door middel van formele specificatie en verificatie, interface definities uit te breiden met precondities, postcondities en object invariante. Implementaties kunnen aan de hand van deze definities geverifieerd worden, waarmee de correctheid van een applicatie, systeem of individuele broncode gemeten kan worden. Ik ben erg benieuwd wat dit betekent voor Test Driven Development. Methodes als Contract First geven hier wellicht een antwoord op.

Links

1. Code Contracts: http://research.microsoft.com/en-us/projects/contracts/
2. Korte paper over Hoare Logic: http://www.cs.princeton.edu/~appel/papers/subst.pdf
3. Informatie over Eiffel's Design by Contract: http://www.eiffel.com/developers/design_by_contract.html
4. Spec#, Microsoft Research formele taal voor Design by Contract: http://research.microsoft.com/en-us/projects/specsharp/
5. Dotned: http://dotned.nl
6. Devnology: http://devnology.nl/podcasts

Pieter Joost van de Sande, werkt als principal consultant bij Atos Origin. Daarnaast is hij onderdeel van het bestuur van de grootste .NET gebruikersgroep van Nederland DotNed. Voor zijn actieve inzet voor de community heeft Microsoft hem beloond met de MVP award.