Erwin van der Valk is technisch architect bij Qurius ETX

Code analysis in Visual Studio 2005

KWALITEITSVERBETERENDE TOOLS VOOR HET SCHRIJVEN VAN ROBUUSTE EN BETROUWBARE SOFTWARE

Met hulp van de code-analysetools in Visual Studio 2005 kan de ontwikkelaar problemen in broncode sneller vinden en oplossen. Deze tools zijn geïntegreerd in de IDE zodat ze het complete ontwikkel- en testproces van elke ontwikkelaar ondersteunen.

Een klant belt op, witheet! *"Ik kan geen nieuwe orders meer maken! En deze orders moeten **nu** de deur uit!"* Terwijl de boze klant wacht, ben je uren, zo niet dagen bezig het probleem te analyseren, een patch te schrijven en deze te installeren bij alle gebruikers. Uiteindelijk blijkt het probleem te zijn veroorzaakt door een kleine bug die misschien in 5 minuten had kunnen worden opgelost tijdens het programmeren. Zo'n scenario komt de meeste ontwikkelaars wel bekend voor. Ook al maak je een goed ontwerp, schrijf je nette broncode en test je de software goed, het blijft gewoon erg lastig om foutloze software te schrijven. Daarom is het ontzettend belangrijk om problemen in de broncode zo vroeg mogelijk te constateren en op te lossen.

Met de komst van Visual Studio 2005 wordt dat een stuk eenvoudiger. Er is namelijk een aantal tools geïntegreerd in Visual Studio 2005 dat ontwikkelaars actief ondersteunt bij het schrijven van betere code. Deze nieuwe, kwaliteitsverbeterende tools zijn volledig geïntegreerd in de IDE, maar ook in het softwareontwikkelproces dat vrijwel elke ontwikkelaar doorloopt. Afbeelding 1 toont de verschillende fases in het ontwikkeltraject, met daaronder de nieuwe tools in Visual Studio 2005 die deze fases ondersteunen.

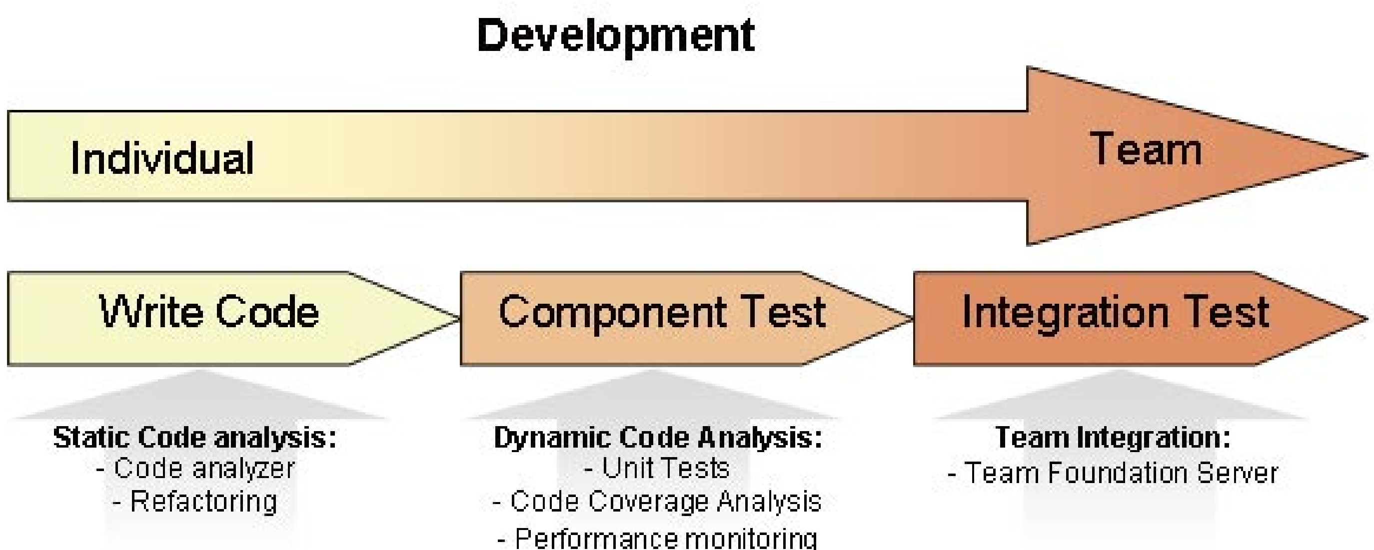
In het begin van het ontwikkelproces ben je als ontwikkelaar bezig met het schrijven van broncode. De Static Code Analyse tools analyseren de broncode en helpen je bij het vinden van problemen en aanbrengen van verbeteringen. Later in het ontwikkelproces moet je als ontwikkelaar de componenten testen. Met de Dynamic Code Analyse tools zul je sneller bugs of performanceproblemen kunnen vinden en oplossen. Aan het eind van het ontwikkelproces helpt Team Foundation Server je om de output van de code-analysetools te gebruiken om het complete ontwikkelteam te ondersteunen.

Static code analysis

In het begin van een ontwikkeltraject ben je met name bezig met het schrijven van broncode. Het zou ideaal zijn als de IDE tijdens het programmeren direct feedback geeft over de kwaliteit van de broncode en je ook helpt bij het oplossen van eventuele problemen. In Visual Studio 2005 is dat gerealiseerd door het integreren van de Code Analyzer en het beschikbaar stellen van een aantal refactoring patterns.

Code Analyzer

De Code Analyzer in Visual Studio 2005 helpt je bij het vinden van problemen tijdens het programmeren. Je kunt de Analyzer zien als een soort spellingcontrole die niet controleert op spelfouten, maar op mogelijke problemen in de broncode. Dit kunnen problemen zijn op verschillende niveaus, variërend van afwijkingen in naamgevingconventies, verkeerde ontwerpkeuzes of beveiligingspro-



Afbeelding 1. Code-analysetools in Visual Studio 2005

blemen. De Code Analyzer in Visual Studio 2005 wordt ook wel een 'Static Code Analyzer' genoemd, omdat deze de broncode zelf analyseert. Dit in tegenstelling tot een 'Dynamic Code Analyzer' die kijkt naar het gedrag van de broncode als deze wordt uitgevoerd. De validatieregels van de Code Analyzer zijn gebaseerd op de .NET Design Guide Lines. Het doel van deze guide lines is ervoor te zorgen dat oplossingen die met .NET-technologie zijn gerealiseerd op een consistente en voorspelbare manier werken. In deze guide lines zit een enorme hoeveelheid kennis verwerkt over hoe een .NET-applicatie geprogrammeerd zou moeten worden.

Codevoorbeeld 1 toont een stuk C#-code waarmee ik de mogelijkheden van de Code Analyzer zal demonstreren. Dit voorbeeld toont de functie 'GetDaysUntilNow' die, op basis van een datum in stringformaat, uitrekent hoeveel dagen die datum verschilt met de huidige datum. Als de meegegeven datum niet geldig blijkt te zijn, wordt een constante waarde teruggegeven. Deze code compileert zonder problemen en ziet er op het eerste gezicht goed uit. Toch bevat deze code enkele problemen die zonder Code Analyzer misschien wel onopgemerkt zouden blijven. Als je de Code Analyzer aanzet worden de volgende problemen getoond:

- Don't use underscores in identifiers: De variabele days_until_now (regel 3) bevat underscores. Dit is over het algemeen minder leesbaar dan het gebruik van camel casing. Daarom wordt aangeraden geen underscores te gebruiken.
- IFormatProvider not specified: In regel 7 wordt een aanroep naar DateTime.Parse gedaan, zonder een IFormatProvider op te geven, die bepaalt welke landinstellingen worden gebruikt voor het interpreteren van de datum. De default-taalinstellingen die gebruikt worden zijn misschien niet de juiste. Daarom is het beter om expliciet een IFormatProvider mee te geven.
- Don't catch general exceptions: In regel 15 wordt een generieke exceptie afgevangen die niet opnieuw wordt doorgegeven. De bedoeling hiervan is om de exception, die door de DateTime.Parse wordt gegooid bij een ongeldige datum, af te vangen. Het is echter veel veiliger om in plaats van een algemene exception een specifieke exception af te vangen, zoals in dit geval de FormatException. Het afvangen van een algemene exception heeft vaak tot gevolg dat je informatie over onverwachte exceptions kwijt raakt. In zulke gevallen is de oorzaak van fouten bijna niet meer te achterhalen.

De codeanalyse wordt uitgevoerd tijdens het compileren. Eventuele overtredingen op de validatieregels worden als waarschuwing getoond in het outputvenster. Zo kun je waarschuwingen van de Code Analyzer op dezelfde manier oplossen als foutmeldingen van de compiler. De codeanalyse is dus volledig geïntegreerd in het ontwikkelproces. De Code Analyzer staat niet standaard aan; je kunt deze per project aanzetten. Dit is handig, omdat bij grote projecten de codeanalyse extra tijd kan kosten. Je kunt ervoor kiezen om de codeanalyse alleen aan te zetten bij projecten waar nog actief aan ontwikkeld wordt. Daarnaast kun je precies bepalen welke validatieregels wel of niet uitgevoerd moeten worden op zowel project- als methodeniveau. Dit is handig als je het niet eens bent met de waarschuwingen van de Code Analyzer of als een regel niet binnen de standaarden en richtlijnen van je bedrijf past. Met de Code Analyzer in Visual Studio 2005 kun je dus sneller problemen in de code vinden, deze sneller oplossen en zo de kwaliteit van de applicatie verbeteren.

Refactoring

Naast de Code Analyzer introduceert Visual Studio 2005 nog meer kwaliteitsverbeterende hulpmiddelen. Een voorbeeld zijn de refactoring-opties. Refactoring is het proces van het verbeteren van de kwaliteit van de code, zonder dat de werking ervan verandert. Hierover zijn al vele boeken geschreven, waarin uiteenlopende refactoring-patronen worden beschreven. Visual Studio

```
1  public int GetDaysUntilNow(string dateText)
2  {
3    int days_until_now;
4    try
5    {
6    // Convert to date
7    DateTime dateValue = DateTime.Parse(dateText);
8
9    // Get the difference between now and the date entered
10   TimeSpan span = DateTime.Now.Subtract(dateValue);
11
12   // Get the number of days
13   days_until_now = span.Days;
14   }
15   catch (Exception)
16   {
17   // invalid date entered
18   days_until_now = -1;
19   }
20
21   return days_until_now;
22 }
```

Codevoorbeeld 1.

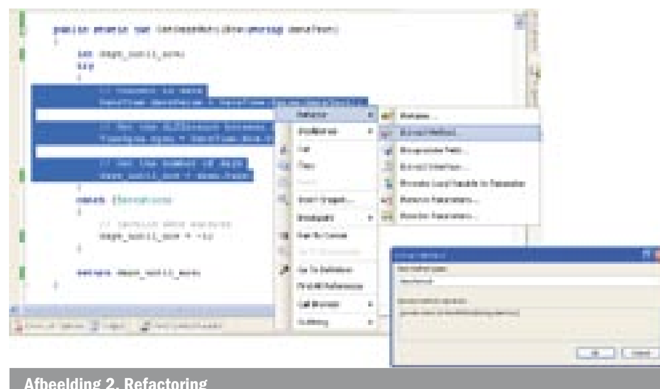
2005 ondersteunt een aantal van deze patronen. De belangrijkste hiervan zijn:

- Extract Method: Hiermee kun je snel een lange methode opsplitsen in kleinere variabelen. Afbeelding 2 toont hier een voorbeeld van.
- Encapsulate Field: Hiermee kun je een field met een paar muisclicks omvormen tot een property, compleet met get- en set-methoden.
- Promote local variable to parameter, Reorder en Remove Parameters: Hiermee kun je eenvoudig parameters toevoegen, verwijderen of de volgorde wijzigen. Dit is vooral handig als de functie op veel plaatsen gebruikt wordt.
- Rename: Hiermee kun je snel een identifier hernoemen. Visual Studio 2005 zoekt zelf alle plaatsen op waar deze gebruikt wordt en hernoemt hem. Dit werkt slimmer dan een zoek & vervangactie, omdat Visual Studio 2005 meer kennis heeft van de broncode dan een gewone tekstverwerker.

Afbeelding 2 toont een voorbeeld van het toepassen een refactoring-patroon, namelijk de Extract Method-functie. Als je hiervoor kiest, zal Visual Studio 2005 de code analyseren om te bepalen welke scope de nieuwe methode moet krijgen en welke parameters nodig zijn. Je hoeft alleen nog de nieuwe methode een naam te geven. Dankzij de refactoring-opties in Visual Studio 2005 kun je dus sneller verbeteringen aanbrengen in de code.

Dynamic Code Analysis

Na het schrijven van de broncode moet de code getest worden om te kijken of de code wel doet wat het moet doen. Met andere woorden, de code moet worden getest op fouten en op de gestelde



Afbeelding 2. Refactoring

```
[TestMethod()]
void WithdrawTest()
{
   // 100 euro op de bankrekening
   int amount = 110;

   // Maak een nieuwe bankrekening class
   BankAccount b = new BankAccount(amount);

   // Neem 10 euro op.
   int amountToWithdraw = 10;

   b.WithDraw(amountToWithdraw);

   // Controleer of er inderdaad 10 euro is opgenomen
   if (b.Amount != 100)
   Assert.Fail(“Amount invalid, expected 100”);
}
```

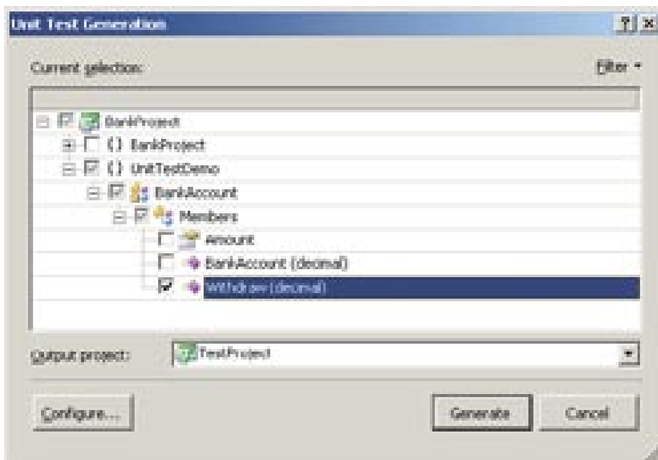
Codevoorbeeld 2. Unit-test

performance-eisen. Aangezien deze fase ontzettend belangrijk is voor het ontwikkeltraject, is er ook voor deze fase een aantal kwaliteitsverbeterende tools in Visual Studio 2005 geïntegreerd. Deze worden ook wel ‘Dynamic Code Analysis’ tools genoemd, omdat deze tools kijken hoe de broncode zich gedraagt als deze wordt uitgevoerd.

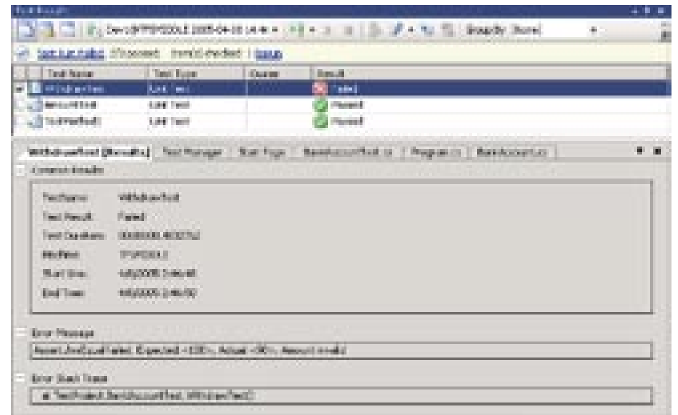
Unit testing

Om te controleren of de code geen fouten bevat moet het getest worden. Vaak wordt hiervoor een uitgebreid handmatig testplan geschreven. Helaas is het uitvoeren van zo’n testplan eentonig en foutgevoelig werk dat onder de druk van deadlines soms niet goed wordt uitgevoerd. Daarom zijn unit-tests in het leven geroepen. Een unit-test is een geautomatiseerde test die wordt geschreven door een programmeur om een code-unit te testen. Het voordeel van zo’n geautomatiseerde test is dat deze eenvoudig herhaald kan worden. Zo wordt het na elke aanpassing van de code mogelijk de unit-test uit te voeren en te kijken of deze geen fouten bevat. Codevoorbeeld 2 laat een vereenvoudigd voorbeeld zien van een unit-test. Deze test de Withdraw-methode van een bankaccount-class. Er wordt een nieuwe bankrekening met een saldo van 110 euro gecreëerd. Er wordt vervolgens 10 euro opgenomen. Daarna wordt gecontroleerd of het verwachte bedrag van 100 euro op de rekening staat.

Zoals uit codevoorbeeld 2 blijkt, is het schrijven van een unit-test nog steeds arbeidsintensief werk. Om dat probleem op te lossen helpt Visual Studio 2005 bij het genereren van een testraamwerk dat uit de volgende onderdelen bestaat:



Afbeelding 3. Testcodegeneratie



Afbeelding 4. Testresultaten

- Een testproject waarbinnen de unit-tests zullen vallen. Op deze manier zijn de unit-tests gescheiden van de code van het programma.
- Een of meer test-classes waarin alle individuele tests voor één class staan. Bij het genereren van de testcode kun je selecteren welke functies of properties je wilt testen; zie ook afbeelding 3. Visual Studio 2005 maakt voor elk te testen functie een testraamwerk waarin het aanroepen van deze functies al staat gegenereerd.
- Een façade voor de te testen class. Dit is een wrapper-class waarmee niet alleen alle publieke functies en properties beschikbaar worden gesteld, maar ook alle private properties. Op deze manier kun je de interne state van je class ook controleren.

Het enige dat je zelf moet doen is het veranderen van de waarde van de parameters en het controleren van het resultaat van de functieaanroep. Op deze manier kun je zeer snel nieuwe tests toevoegen. Als je een aantal tests hebt gemaakt, kun je deze bekijken in de testmanager. Hier kun je ook tests selecteren om uit te voeren. Afbeelding 4 toont hoe de resultaten eruit zien van het uitvoeren van een unit-test. Als een unit-test niet gelukt is, kun je door erop te klikken zien waarom deze niet is gelukt. In dit voorbeeld heb ik de Withdraw-methode aangepast, zodat geen geld meer opgenomen wordt, met als gevolg dat de unit-test faalt.

Met unit-tests kun je na elke wijziging van de code opnieuw controleren of deze geen fouten bevat. Hiermee kun je de kwaliteit van de code aanzienlijk verbeteren.

Code Coverage

Het maken van unit-tests kan een prima hulpmiddel zijn om de kwaliteit van code te verbeteren. Echter, als je unit-tests maar 50% van de code testen, kan de andere 50% nog steeds fouten bevatten. Met Code Coverage Analyzer in Visual Studio 2005 wordt het mogelijk om te controleren hoeveel procent van de code daadwerkelijk getest wordt door unit-tests.

Je kunt de Code Coverage-analyse aanzetten in de Test Manager. Als je dat doet zul je per getest bestand zien hoeveel procent van de code afgedekt is met unit-tests. Afbeelding 5 toont hoe Visual Studio precies aangeeft welke regels code wel of niet getest zijn. Deze informatie kun je gebruiken om de tests te verbeteren of aan te vullen, zodat je 100% van de code af kunt dekken met unit-tests. Hiermee kun je dus de kwaliteit van de unit-tests en daarmee ook de kwaliteit van de applicatie verbeteren.

Performancemonitor

Het is niet alleen belangrijk om te testen of code geen fouten bevat, er moet ook gecontroleerd worden of de performance wel voldoet aan de gestelde eisen. Een applicatie met een slechte performance is namelijk net zo vervelend voor een klant als een instabiele appli-

Class1Test.cs* | Class1.cs

ProductData.BankAccountTest | CreditTest()

```
private TestContext m_testContext = null;

public TestContext TestContext
{
    get { return m_testContext; }
    set { m_testContext = value; }
}
[TestMethod()]
public void BalanceTest()
{
    // TODO: Initialize to an appropriate value
    string customerName = null;
    // TODO: Initialize to an appropriate value
    double balance = 0;
    BankAccountNS.BankAccount target = new BankAc
    // TODO: Assign to an appropriate value for
    double val = 0;
    Assert.AreEqual(val, target.Balance);
    Assert.Inconclusive("Look at this code and ma
}
[TestMethod()]
public void ConstructorTest()
{
    // TODO: Initialize to an appropriate value
    string customerName = null;
    // TODO: Initialize to an appropriate value
    double balance = 0;
    BankAccountNS.BankAccount target = new BankAc
    // TODO: Implement code to verify target
    Assert.Inconclusive("Look at this code and ma
    // TODO: Initialize to an appropriate value
```

Afbeelding 5. Code Coverage-resultaten

catie. Het oplossen van performanceproblemen is lastig. Tijdens het testen van een applicatie kun je wel zien of een applicatie traag reageert, maar de oorzaak ervan is moeilijker te achterhalen. Is de database traag? Wordt er lang gedaan over een bepaalde berekening?
Tijdens het ontwikkelen van een applicatie kun je wel proberen deze te optimaliseren voor performance, maar dat is erg lastig. Je weet namelijk niet altijd waar de performanceproblemen zullen optreden. Vaak is ook hier de 80/20-regel van toepassing. Tachtig procent van de performance van een applicatie wordt bepaald door maar twintig procent van de code. Daarom is het van belang om te kijken waar de meeste performancewinst te halen valt. Hier kan de performancemonitor bij helpen. De performancemonitor bekijkt een applicatie tijdens het uitvoeren van functies en verzamelt allerlei nuttige informatie die je kunt gebruiken voor het optimaliseren van de performance.

Sampling en instrumentation

De performancemonitor kan de applicatie op twee manieren analyseren, namelijk met *sampling* en *instrumentation*. Als je deze manieren vergelijkt met een dokter (performancemonitor) die de gezondheid van zijn patiënt (de applicatie) wil bepalen, kun je sampling vergelijken met een routinecontrole, waarbij de dokter elk uur even komt kijken. Instrumentation werkt meer als een monitor op de intensive care, die continue gegevens verzamelt over de patiënt. Meestal zal je sampling gebruiken om de potentiële performanceproblemen te vinden. Als je deze gevonden hebt, kun je met instrumentation één onderdeel van de applicatie bekijken om het performanceprobleem verder te analyseren.

Sampling gebruik je dus om te kijken waar in de applicatie de performanceproblemen zitten. Sampling werkt op de volgende manier: om de zoveel klokcycli wordt een snapshot van de callstack gemaakt om te bepalen welke functie wordt uitgevoerd en hoeveel geheugen er gebruikt wordt. Het voordeel van deze methode is dat de applicatie zo min mogelijk beïnvloed wordt. Het nadeel is dat kortlopende functies misschien niet gesampled worden. Als je met sampling een performanceprobleem hebt geïdentificeerd, kun je instrumentation gebruiken om een gedetailleerd beeld van het probleem te krijgen. Om heel precies te kunnen meten welke functies er allemaal worden uitgevoerd, voegt instrumentation een klein stukje code toe aan het begin en eind van elke functie. Hiermee kun je een heel accuraat beeld krijgen van de applicatie. Het uitvoeren van dit stukje code beïnvloedt wel de performance. Dit kan vervelend zijn als bijvoorbeeld timing heel belangrijk is in de applicatie.

Het resultaat van een performancemonitor-sessie is een rapport waarin alle informatie over de uitgevoerde functies komt te staan:

- Hoe lang duurt het uitvoeren van een bepaalde functie?
- Hoe vaak wordt een bepaalde methode aangeroepen?
- Hoeveel geheugen wordt gebruikt door een bepaalde methode?
- Hoeveel instanties van een bepaalde class worden gemaakt en hoeveel geheugen nemen die in beslag?

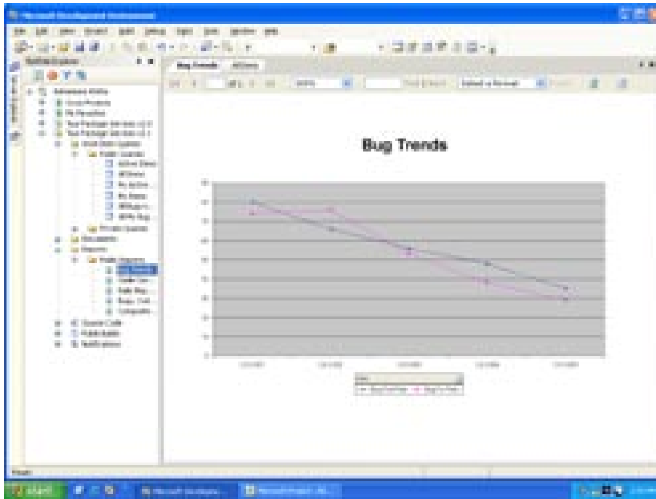
De bovenstaande gegevens kun je gebruiken om te bepalen in welke functie(s) de meeste performanceproblemen zitten en waardoor de problemen worden veroorzaakt. Wanneer je deze problemen oplost heb je de kwaliteit van je applicatie opnieuw verbeterd.

Integratie met Team Foundation

De tools die ik tot nu toe heb beschreven kun je heel goed als individuele ontwikkelaar gebruiken. Met Visual Studio Team Foundation kun je deze tools ook gebruiken om je complete team te ondersteunen. In een ontwikkelteam kun je bijvoorbeeld afspreken dat ingecheckte code moet voldoen aan bepaalde kwaliteitseisen. Een voorbeeld: iedere ontwikkelaar moet eerst de codeanalyse hebben uitgevoerd. Minstens 80 procent van zijn code moet afgedekt zijn met unit-tests. Als je wilt kun je met Visual Studio Team Foundation afdwingen dat code pas ingecheckt mag worden als deze aan de kwaliteitseisen voldoet.
Veel bedrijven gebruiken een nachtelijke build om te kijken of de ingecheckte code van individuele teamleden ook als één geheel werkt. Dit proces wordt ook ondersteund in Visual Studio Team Foundation. Daarnaast is het mogelijk om, na het maken van een build, direct een aantal controles uit te voeren, namelijk: codeanalyse, unit-tests, Code Coverage-analyse en performancetests. De resultaten hiervan kun je inzichtelijk maken met een aantal rapporten, waarvan een voorbeeld getoond wordt in afbeelding 7.

Function Name	Function Addr...	Number of Calls	Elapsed Exclusive Time
WindowsApplication10.EntryPoint.Main	0x0600000A	1	1326179
System.Windows.Forms.Application.Run	0x0A000025	1	3105294104
WindowsApplication10.Form1.Dispose	0x06000009	1	2616
WindowsApplication10.Form1.button1_Click	0x06000007	17	6795654
WindowsApplication10.Form1.test	0x06000006	17000	5504190
WindowsApplication10.Form1..ctor	0x06000005	1	10992785

Afbeelding 6. Performancemonitoring-resultaten



Afbeelding 7. Nachtelijke build-resultaten

Met Visual Studio Team Foundation kun je dus controleren of ingecheckte code voldoet aan de gestelde kwaliteitseisen. Daarnaast kun je de resultaten van de analysetools gebruiken om een beter beeld te krijgen van de kwaliteit van de broncode.

Betere software

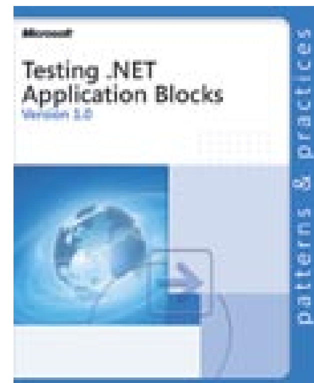
Met de code-analysetools in Visual Studio 2005 kunnen problemen in broncode sneller worden gevonden en sneller worden opgelost. Deze tools zijn geïntegreerd in de IDE zodat ze het complete ontwikkel- en testproces van elke ontwikkelaar ondersteunen. Op deze manier helpt Visual Studio 2005 bij het schrijven van betere software.

Erwin van der Valk is technisch architect bij Qurius ETX. Hij houdt zich voornamelijk bezig met het ontwikkelen van web applicaties met behulp van C# en SQL Server. Voor vragen en opmerkingen kun je mailen naar erwinv@quriusetx.nl

Nuttige internetadressen

.NET Design guidelines: http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp
Refactoring: http://www.refactoring.com
FxCop: http://www.gotdotnet.com/team/fxcop/

(advertentie Microsoft Press)



Testing .NET Application Blocks
ISBN: 0-7356-2220-5
Auteur: Microsoft
Pagina's: 192