Program Exploration

Helpende hand bij unit testing

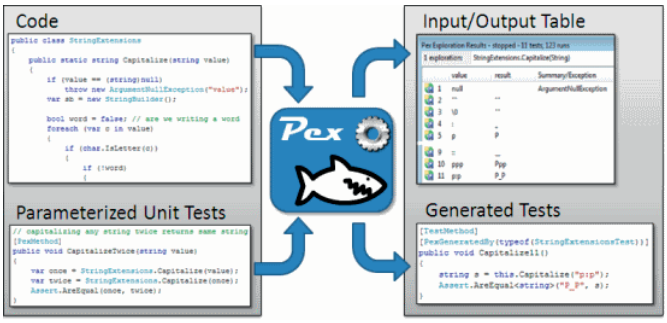
Alexander Nowak

Program Exploration (Pex) is een project van Microsoft Research dat als doel heeft om de generatie van unit tests te automatiseren. Dit gaat verder dan de reeds bestaande unit-test wizard in Visual Studio die op basis van de methode-signatuur een unit test skelet voor je aanmaakt. Pex zoekt namelijk ook de invoerwaarden die eventueel meegegeven moeten worden aan de methode die je aan het testen bent. Als resultaat krijg je een testsuite van unit tests met een hoge code coverage.

Unit testen wordt beschouwd als een 'best practice' om zoveel mogelijk programmeerfouten en eventuele functionele fouten uit je programma te halen, voordat ze andere tests op een hoger niveau blokkeren (of erger: de eindgebruiker blokkeren).
Unit testing is niet compleet nieuw, maar de komst van xUnit frameworks (Junit, Nunit , mbUnit, MS test framework ,etcetera) en XP programmings practices (test-first) hebben unit testing een grotere aandacht gegeven. Naast uniformiteit , automatiseringsmogelijkheden en dergelijke, biedt unit testing de ontwikkelaar ook het voordeel om een veiligheidsnet op te bouwen dat hij bij aanpassingen van de code kan uitvoeren om te zien of alles nog werkt.
Ondanks de voordelen die unit testing kan opleveren, is er in sommige shops niet veel animo voor unit testing om allerlei reden; de moeilijkheid om iets in isolatie te testen, het opstellen van talrijke tests om een zo groot mogelijke coverage te verkrijgen en het onderhoud van al die testen. Het is niet ondenkbaar dat er meer test-code is dan uiteindelijke 'echte' code.
De ontwikkelaar moet zijn designprobleem of zijn reeds geschreven code goed doorgronden om een testsuite te schrijven die alle 'hoeken' van de code onderzoekt. Hij moet ook rekening houden met zogenaamde 'negatieve' testen, het controleren of de code ook in een foutsituatie goed reageert.
Voor bepaalde klassemethodes kan het aantal unit test snel oplopen. Dikwijls is de code repetitief en verschilt enkel op vlak van uitvoerwaarden. MSTest (en MbUnit bijvoorbeeld) heeft de mogelijkheid om zogenaamde 'data-driven' tests op te stellen. Zo kun je in Visual Studio bepalen dat de invoerwaarden en het verwachte resultaat voor een test uit een externe databron moet worden gelezen. In je unit test lees je dan de invoerwaarden en het verwachte resultaat uit. Je voert dan de methode uit met een specifieke 'rij' en vergelijkt het verkregen antwoord met het verwachte resultaat. Zo kun je het aantal unit tests enigszins beperken.
Maar het vinden van de invoerwaarden en verwachte resultaten is nog steeds het werk van de programmeur.
Bij het horen van unit test generatie zullen ontwikkelaars die test-driven development (TDD) bezigen misschien hun wenkbrauwen fronsen, omdat zij de unit tests eerst opstellen om hun design- en codeproces te sturen. Als nevenresultaat hebben ze een volledige suite van unit tests ter beschikking om hun code volledig te testen. Maar zelfs in een test-last (of test-during) manier van werken is unit testing een goede manier om zoveel mogelijk fouten te vinden en er voor te zorgen dat ze er niet inkomen.

Pex: helpende hand

Er zijn tools op de markt verschenen die proberen de ontwikkelaar te helpen met het bulk van het werk van het schrijven van unit tests in de vorm van automatisering. We doelen niet enkel op het aanmaken van de skeleton code op basis van de methodesignatuur, maar ook voor het bepalen van de invoerwaarden die nodig zijn om een bepaalde unit test uit te voeren. Niet enkel één paar invoerwaarden, maar alle combinaties, opdat een zo hoog mogelijk code coverage kan bereikt worden.
In de Java-wereld zijn er enkele tools die hieraan sleutelen: TestGen4J (OpenSource), Jtest (Parasoft), CodePro (Instantiations), AppPerfect Unit Tester (AppPerfect) en AgitarOne JUnit Generator (Agitar). Pex is zo'n tool voor Microsoft .NET. Het is een combinatie van .Net libraries, Visual Studio integratiefaciliteiten en een speciale profiler die code onder de loep neemt. Net zoals in klassieke unit testing infrastructuur in Visual Studio werk je met attributes en wizards.



FIGUUR 1: OVERZICHT PEX (BRON: HTTP://RESEARCH.MICROSOFT.COM/PEX,TOELATING VAN PELI DE HALLEUX

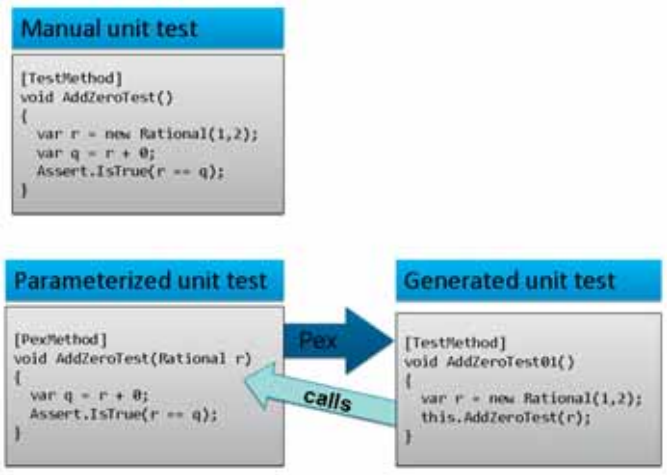
Om met Pex aan de slag te kunnen (zie figuur 1), introduceert het eerst een variant van de data-driven test: de Parameterized Unit test (PUT). Deze test lijkt op een unit test maar bepaalt in zijn methodesignatuur ook de benodigde parameters voor het uitvoeren van de methode die getest wordt. Het verificatieproces moet dan zo opgesteld worden dat het een hele verzameling testen kan nagaan of het verkregen resultaat overeenstemt met de verwachtingen.

De Pex zal dan deze PUT gebruiken om de te testen methode uit te voeren in zijn 'exploratie' fase. Of beter gezegd, meermaals uit te voeren. Afhankelijk wat hij onderweg te weten komt over de code, zal het de invoerwaarden bepalen om zoveel mogelijk testafdekking te hebben. Het kan zijn dat Pex meer dan eens de code moet doorlopen, bijvoorbeeld wegens een conditie op een inputwaarde die daarmee de code in een ander gedeelte van code zal sturen.
Pex is wel zo slim om te weten wanneer hij moet stoppen. Bijvoorbeeld wegens loops of recursieve functies. Diezelfde 'beveiliging' kun je in je programmacode aansturen met behulp van annotaties van Pex attributen.
Als Pex meent dat het werk erop zit, zal het de unit tests genereren waarbij de, in de exploratiefase, gevonden invoerwaarden hard gecodeerd worden en doorgegeven worden aan de PUT die je hebt gemaakt (of hebt laten generen). Deze unit tests kunnen dan via de standaard MSTest framework (in Visual Studio of via commandline) uitgevoerd worden. Door de code 'coverage' optie aan te zetten in de testconfiguratiesettings kun je controleren of er bepaalde delen van de code niet benaderd worden door de unit tests.
Pex helpt ook code te verbeteren. Tijdens de exploratie kan het namelijk zijn dat het een input genereert die het programma een exceptie doet opwerpen, bijvoorbeeld de klassieke 'divsion-by-zero'. Pex suggereert dan een voorstel om de code te verbeteren door een preconditie af te testen en kan zowaar de codesnippet in de code plaatsen. Uiteraard heb je ook de mogelijkheid, zoals in klassieke unit testing, om 'expected exceptions' te bepalen.

PUT: startpunt voor Pex

Voordat Pex zijn werk kan doen, moeten we een PUT schrijven. In een klassieke unit test ondervraag je de code onder test in een welbepaalde manier. Elke unit test is één welbepaalde manier om bijvoorbeeld het object aan te maken en te voorzien met benodigde gegevens. Ook tijdens de uitvoering van de methode moet je eventueel relevante waarden meegeven. Daarna moet je controleren of het gedrag van die methode overeenstemt met de verwachte waarden. In de meeste gevallen kun je verschillende combinaties maken van instelwaarden, parameters en resultaten afhankelijk van de verschillende code - paden. Die wil je allemaal testen. Daarom heb je ook meerder unit tests. Elke unit test is een 'scenario' om door je code te lopen en te kijken of die overstemt met wat je bedoelde. Dus voor elk scenario moet je nadenken over de relevante invoer die in een welbepaalde manier door je code gebruikt wordt en nadenken hoe controleert of het resultaat van de uitvoering van de code overeenstemt met de verwachtingen.
Een PUT is een testmethode die idealiter ook al die scenario's kan uitvoeren en verifiëren (zie figuur 2). Dit vergt dus een aanpassing om alle variabele elementen voor een enkelvoudig unit test te isoleren aan te reiken aan de testmethode. Dus een PUT is een test die scenarioafhankelijke zaken via parameters aanreikt.

Pex helpt je bij het generen van een skelet van een PUT voor een methode. Die is quasi op dezelfde manier als de skelet generatie



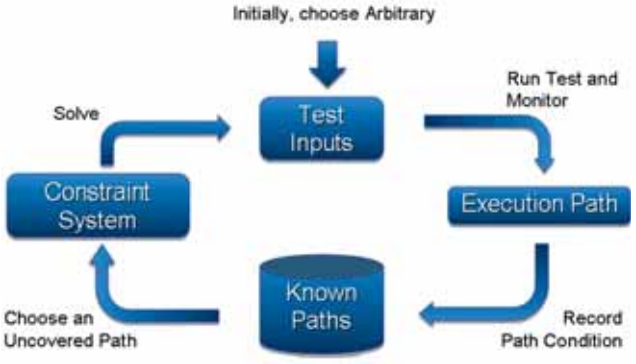
FIGUUR 2: PARAMETRIZED UNIT TEST (BRON: HTTP://RESEARCH.MICROSOFT.COM/PEX , TOELATING VAN PELI DE HALLEUX)

van klassieke unit test via de Visual Studio integratie van MSTest of achter de schermen in 'code digging' scenario (dadelijk vanuit de te testen methode de exploraties starten zonder zelf eerst een PUT te schrijven).
Maar de kracht van Pex is dat hij via zo'n PUT kan nadenken welke instelwaarden en/of parameters voor de methode onder test hij moet generen (zie figuur 3). Pex doet dit niet lukraak (behalve de eerste keer gebeurt er een 'educated guess'), maar voert de code via de PUT uit en kijkt dus welke codepaden er zijn en welke condities er zijn om in een codepad te geraken. Meestal heeft hij meerdere keren nodig om alle paden te doorlopen. De argumentwaarden voor de unit tests worden zo bijgehouden. Uiteindelijk zal Pex de benodigde klassieke unit tests (met vaste waarden) generen op basis van zijn analyse.

De kunst is nu te verzekeren dat je in de PUT de controles inbouwt om te kijken of de uitkomst wel juist is. Aangezien Pex de invoer aanmaakt op basis van de code-exploratie, moet je een manier vinden om de verificaties te doen voor de talrijke scenario's. In een klassieke unit test kun je de verificatie toespitsen op één combinatie van invoerwaarden.

Voorbeeld

Stel dat je ergens in je programma de regels hebt gecodeerd die bepalen hoeveel korting je geeft (en we hebben maar één enkel product dat we verkopen!) . Deze code is niet in TDD stijl opge-



FIGUUR 3: DYNAMIC SYMBOLIC EXECUTION (BRON: HTTP://RESEARCH.MICROSOFT.COM/PEX , TOELATING VAN PELI DE HALLEUX)

bouwd en we hebben nog geen unit testen ter beschikking (zie code snippet 1).

```
public class bestellingsRegels
 {
    public decimal BepaalKortingsPercentage(uint aantal)
     {
        decimal korting = 0.0M;

        korting = (aantal > 1000 & aantal <= 10000) ? 1.5M :
                (aantal > 10000 & aantal <= 25000) ?
                3.0M :
                (aantal > 25000) ? 4.0M :
                                   0.0M ;

        return korting;
     }
 }
```

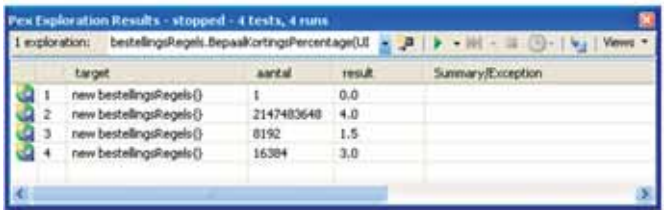
CODE SNIPPET 1: VOORBEELD BESTELKORTING

Via traditionele testtechnieken zoals beslissingstabellen, equivalentie klassebepaling en grenswaardenanalyse kunnen we op zoek gaan naar geschikte invoerwaarden voor de bestelhoeveelheid om een zo groot mogelijke coverage te krijgen. We weten dat we in dit voorbeeld minstens vier testwaarden moeten vinden, zodat we alle codepaden doorlopen.
Maar Pex kan ons ook een handje helpen. Als je in Visual Studio op de methode in kwestie gaat staan met je cursor en op de rechtermuisknop klikt, dan verschijnt de mogelijkheid om Pex dadelijk zijn werk te laten doen, ook bekend als 'code digging' (zie screenshot 1). Je kunt ook expliciet vragen aan Pex om een PUT voor jouw te maken of je kun er handmatig eentje maken.

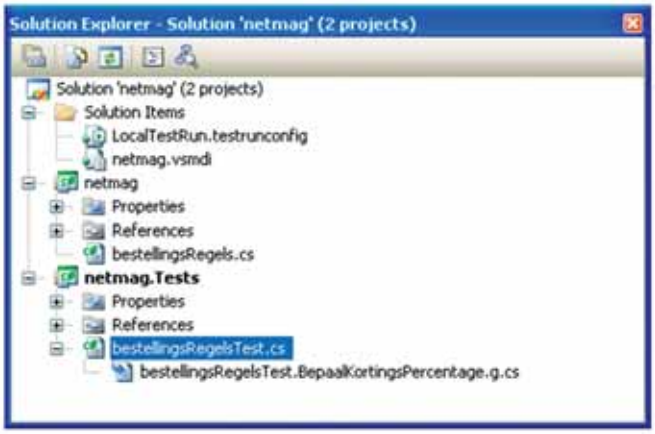
SCREENSHOT 1 : CODE DIGGING

Achter de schermen maakt Pex een PUT aan een voert die uit met een aantal waarden. In screenshot 2 zie het resultaat van deze sessie. In de titel van het venster zie je dat Pex vier maal de methode in test heeft uitgevoerd telkens met een ander invoerwaarde.

Pex is begonnen met de waarde 1. Deze waarde voldeed aan de laatste conditie. Maar Pex onthield wel dat er nog een andere tak was van de conditie. Pex gaat dan op zoek naar de waarde om de conditie te 'flippen'. Met andere woorden om in de andere tak te komen. Dus komt het met een waarde om niet aan de conditie te voldoen. En zo gaat hij maar door tot hij alle paden doorlopen heeft, tenzij het op een of andere beperking in tijd of iteratie botst. De voortgang van de zoektocht van Pex kun je altijd natrekken door in de 'views' optie naar de 'output window' te gaan. Daar zie je meer gedetailleerde informatie. Er is ook een uitgebreid rapport dat je kunt bekijken als je de optie activeert via het menu 'options'.
Nu heb je de mogelijkheid om dit resultaat te bewaren in de vorm van een PUT methode en enkele klassieke testmethoden door in-



SCREENSHOT 2 : RESULTATEN CODE DIGGING



SCREENSHOT 3 : TEST PROJECT

dividueel of alle testen samen te selecteren en op 'save' te klikken. Dit resulteert in de generatie van volgende unit test (Mstest) in nieuw of bestaand test project.

De PUT methode wordt in een normale CS file geplaatst. De gegenereerde unit tests worden in een speciale file geplaatst (zie Screenshot 3).

```
[TestClass]
[PexClass(typeof(bestellingsRegels))]
[PexAllowedExceptionFromTypeUnderTest(typeof(Argument-
Exception), true)]
public partial class bestellingsRegelsTest
{
    [PexMethod]
    public void BepaalKortingsPercentage([PexAssumeUnderTest]-
    bestellingsRegels target, uint aantal)
    {
        decimal result = target.BepaalKortingsPercentage(aantal);
        PexStore.Value<decimal>("result", result);
        // TODO: add assertions to method bestellingsRegelsTest.
        BepaalKortingsPercentage(bestellingsRegels, UInt32)
    }

}
```

CODE SNIPPET 2 : GEGENEERDE PUT GEDURENDE CODE DIGGING

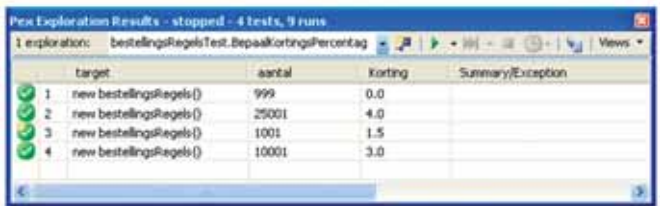
```
public partial class bestellingsRegelsTest
{
    [TestMethod]
    [PexGeneratedBy(typeof(bestellingsRegelsTest))]
    public void BepaalKortingsPercentage01()
    {
        bestellingsRegels bestellingsRegels = new bestellings-
        Regels();
        this.BepaalKortingsPercentage(bestellingsRegels, 1u);
    }

    [TestMethod]
    [PexGeneratedBy(typeof(bestellingsRegelsTest))]
    public void BepaalKortingsPercentage02()
    {
        bestellingsRegels bestellingsRegels = new bestellings-
        Regels();
        this.BepaalKortingsPercentage(bestellingsRegels,
        2147483648u);
    }

    [TestMethod]
    [PexGeneratedBy(typeof(bestellingsRegelsTest))]
    public void BepaalKortingsPercentage03()
    {
        bestellingsRegels bestellingsRegels = new bestellings-
        Regels();
        this.BepaalKortingsPercentage(bestellingsRegels, 8192u);
    }

    [TestMethod]
    [PexGeneratedBy(typeof(bestellingsRegelsTest))]
    public void BepaalKortingsPercentage04()
    {
        bestellingsRegels bestellingsRegels = new bestellings-
        Regels();
        this.BepaalKortingsPercentage(bestellingsRegels, 16384u);
    }

}
```

CODE SNIPPET 3 : GEGENEREERDE UNIT TESTS

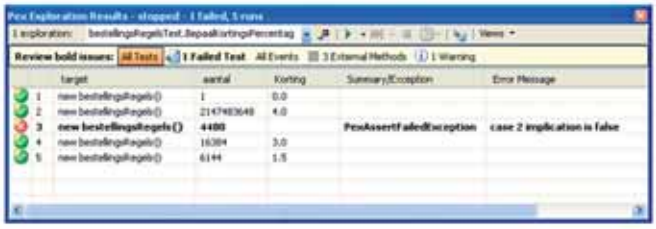
Het is normaal niet de bedoeling dat je in deze laatste wijziging aanbrengt . Als er iets moet worden veranderd aan je code of PUT moet je de exploratie opnieuw uitvoeren van je PUT. De naamgeving van de unit testen is simpelweg een volgnummer. Je moet dus al in de code duiken om te weten wat er exact getest wordt.
Je kunt nu alle tests uitvoeren maar eigenlijk zijn we nog niet klaar. Een ander werk dat we moeten doen is nagaan of de resultaten wel aan de verwachtingen voldoen. Als we dit niet doen, weten we enkel dat de methode niet gecrasht is en een waarde teruggeeft. We moeten dus in de PUT nog een aantal verificaties toevoegen. In een klassieke unit test kon je in ons geval de waarde van korting zelf bepalen en direct in de unit test coderen. In de PUT moeten we dit op één of andere manier generaliseren. Dit is eigenlijk het moeilijkst aan het schrijven van een PUT, want je eindigt helaas snel in het herschrijven van de code zelf zoals in dit voorbeeld. Maar een test zonder verificatie is nu eenmaal geen goede test (zie code snippet 4).

```
[PexMethod]
public void BepaalKortingsPercentage([PexAssumeUnderTest]
bestellingsRegels target, uint aantal)
{
    decimal result = target.BepaalKortingsPercentage(aantal);
    PexAssert.Case(aantal <= 1000).Implies(() => result == 0M).
            Case(aantal > 1000 & aantal <= 10000).Implies(()
            => result == 1.5M).
            Case(aantal > 10000 & aantal <= 25000).Implies(()
            => result == 3.0M).
            Case(aantal > 25000).Implies(() => result == 4.0M);

    PexStore.Value<decimal>("Korting", result);

}
```

CODE SNIPPET 4: VERIFICATIE IN PUT



SCREENSHOT 4: TEST CASE BIJ ASSUMPTIES IN PUT



SCREENSHOT 5 : PEX ASSERT EXCEPTION

Als nu de programmeur een vergissing heeft begaan, bijvoorbeeld in de waarde van een kortingspercentage of bij het bepalen van de verificatie van de testen, is kans groter dat het gedetecteerd wordt bij het uitvoeren van de testen.
Wijzigingen in de code zouden altijd opnieuw getest moeten worden door de ontwikkelaar. In het voorbeeld met een wijziging van de korting zou de uitvoering van alle unit tests dit aan het licht moeten brengen, omdat één van de unit tests andere verwachtingen heeft. Het voldoet aan de verwachtingen zoals die zijn bepaald in de PUT.

```
public decimal BepaalKortingsPercentage(uint aantal)
{
    decimal korting = 0.0M;

    korting = (aantal > 1000 & aantal <= 10000) ? 1.5M :
              (aantal > 10000 & aantal <= 25000) ? 3.25M :
              (aantal > 25000) ? 4.0M :
                                 0.0M ;

    return korting;
}
```

CODE SNIPPET 5A: WIJZIGING KORTING PERCENTAGE (3.0 -> 3.25)

Andere wijzigingen in dit voorbeeld (zie code snippet 102) zoals het wijzigen van de condities zijn minder snel te detecteren.

```
public decimal BepaalKortingsPercentage(uint aantal)
{
    decimal korting = 0.0M;

    korting = (aantal > 5000 & aantal <= 10000) ? 1.5M :
              (aantal > 10000 & aantal <= 25000) ? 3.0M :
              (aantal > 25000) ? 4.0M :
                                 0.0M ;

    return korting;
}
```

CODE SNIPPET 5 : WIJZIGING AANTAL (1000 -> 5000)

In dit geval lukken alle tests, omdat de door Pex gevonden invoerwaarden voor de bestelhoeveelheid niet specifiek de grenswaarden aftasten. Je zou Pex kunnen helpen bij het aanmaken van de invoerwaarden door de verzameling waaruit Pex kan kiezen kleiner te maken (zie code snippet 6).

```
[PexMethod]
public void BepaalKortingsPercentage([PexAssumeUnderTest]
bestellingsRegels target, uint aantal)
{
    PexAssume.IsTrue(new List<uint>() { 999, 1000, 1001, 9999,
    10000, 10001, 24999, 25000, 25001 }.Contains(aantal));

    decimal result = target.BepaalKortingsPercentage(aantal);
    PexAssert.Case(aantal <= 1000).Implies(() => result == 0M).
              Case(aantal > 1000 & aantal <= 10000).Implies(()
              => result == 1.5M).
              Case(aantal > 10000 & aantal <= 25000).Implies(()
              => result == 3.0M).
              Case(aantal > 25000).Implies(() => result == 4.0M);

    PexStore.Value<decimal>("Korting", result);

}
```

CODE SNIPPET 6 : PUT MET ASSUMPTIES

Als je initieel op basis van deze PUT de Unit testen door Pex had laten genereren dan (zie screenshot 4) waren die al dichter bij de grenzen en was de kans groter om wijzigingen, zonder aangepaste testverificatie of die helemaal fout zouden kunnen zijn, terug te vinden. Eigenlijk is het best dat de PUT bij codewijzigingen altijd opnieuw uitgevoerd wordt. Dan is de kans groter dat Pex het probleem zelf ontdekt (zie Screenshot 5).

Dit is dan een teken aan de wand om na te gaan of de code juist werd aangepast of dat je de asserts in je PUT moet aanpassen.

We hebben slechts een aantal features van Pex bekijken. Als je Pex eens wilt uitproberen , kun je op http://research.microsoft.com/Pex/ , naast de Pex installatiefile , onder andere een tutorial en samples.

Misschien nog wel enkele richtlijnen.

- Ook al werkt Pex op niveau van IL , de code generatie uitwerking in Visual Studio is enkel uitgewerkt voor C#
- Mocht je reeds TypeMock geïnstalleerd hebben , dan werkt Pex niet om dat op het zelfde niveau van profiling (instrumentation) bevindt.
- Op de blog-site van de auteurs (Nikolai Tillmann, Peli de Halleux) vind je ook een schat aan informatie (http://blogs.msdn.com/nikolait/ en http://blog.dotnetwiki.org/)
- Dit is een research project ! Dit artikel is beschreven op basis van Pex versie 0.8.31021.3 in Visual Studio 2008

Conclusie

Unit tests schrijven is eigenlijk moeilijk werk. Voldoende tests om zo veel mogelijk code af te lopen en het onderhoud van die tests zijn misschien redenen waarom er soms weerstand is tegen het introduceren van unit testing.
Microsoft research tracht met Pex een hulpmiddel te bieden bij unit testing. Het is geen algemene vervanging van zelf geschreven unit tests. De gegenereerde unit tests kunnen zonder probleem naast je eigengemaakte unit tests bestaan. Het enig doel is om ervoor te zorgen dat je zoveel mogelijk fouten uit je code haalt en er om voor te zorgen dat er geen fouten binnensluipen. Hoe meer hulp je krijgt van Visual Studio, des te beter !

Alexander Nowak is softwareontwikkelaar bij Capgemini België