De auteurs beschrijven in dit artikel hun ervaringen met testgestuurde softwareontwikkeling (*test-driven development*), een test- en ontwikkeltechniek die gebruikt kan worden om testen naadloos te integreren in het software ontwikkelproces.

thema

Testgestuurde software-ontwikkeling

Laagdrempelige werkwijze maakt ontwikkeling beheersbaar

Testen van software is een belangrijk middel om de kwaliteit van de software die wij bouwen te bepalen en te verbeteren. Goed geteste software is meer waard voor een software-ontwikkelende organisatie. Als een programma niet meer om de haverklap omvalt, kan iedereen zich richten op de zaken die er echt toe doen: ontwikkelaars kunnen zich richten op de klant en de features die hij het belangrijkst vindt, testers kunnen zich richten op de bruikbaarheid van de software, marketeers kunnen de waarde van de software laten zien in plaats van excuses te maken voor bekende defecten, managers houden tijd over om aan hun visie te werken in plaats van steeds brandjes te moeten blussen.

In de praktijk wordt er weinig getest, hetgeen indirect leidt tot hoge kosten voor zowel softwareontwikkelorganisaties als klanten[1]. Naar onze mening is er een viertal redenen waarom testen vaak niet een integraal onderdeel van het ontwikkelproces is.

- *Testen is niet leuk.* Als ontwikkelaar wil je goede software maken. Maar als je programmacode geschreven hebt, zou je daarna nog allerlei tests moeten schrijven om defecten in die code te vinden, terwijl je net je best hebt gedaan om correcte code te schrijven.

- *Testen is moeilijk.* Het is lastig om tests te schrijven voor reeds geschreven code. Je wilt componenten los van elkaar testen, maar allerlei afhankelijkheden zitten in de weg. Sommige componenten kunnen alleen indirect getest worden. Verder kost het bedenken van goede testdata en het in een zinvolle begintoestand brengen van het te testen programma vaak veel tijd.

- *Er is geen tijd om te testen.* Software wordt gemaakt onder druk van klanten en projectleiders die niet kunnen wachten tot het af is. Men beseft wel dat testen leidt tot kwalitatief betere code, wat zich in de toekomst vertaalt naar minder fouten, betere aanpasbaarheid en grotere tevredenheid bij klanten. Dit leidt niet tot gedragsverandering, omdat de lange termijnopbrengst niet is te kwantificeren. Testen lijkt niet direct tastbaar resultaat op te leveren, de investering in testtijd op korte termijn is echter wel zichtbaar.

- *Testen is een sluitpost.* Testen wordt vaak in een aparte testfase aan het eind van het project uitgevoerd. Onder tijdsdruk wordt hier als eerste op bezuinigd. Als er in de testfase meer defecten gevonden worden dan verwacht, kan het project gierend uit de hand lopen, ook omdat dat het repareren van een defect vaak nieuwe defecten introduceert. Uiteindelijk wordt de software dan maar opgeleverd met veel "bekende" defecten.

Samengevat lijkt testen een activiteit die op de lange termijn betere software oplevert, maar die op de korte termijn gezien alleen inspanning kost. Dit maakt het lastig om testen voldoende aandacht te geven in het ontwikkelproces. Testgestuurde softwareontwikkeling is een techniek die hierbij goed kan helpen.

INCREMENTEEL Testgestuurde softwareontwikkeling is een incrementele ontwikkeltechniek waarbij testen en ontwerpen gecombineerd worden. Je werkt in zeer kleine stappen in orde grootte van minuten. Je ontwikkelt telkens één specifiek testgeval. Het testgeval wordt

Verzameling unit-tests bij het NIWI

Wij passen testgestuurd ontwikkelen toe in onze projecten. Een voorbeeld hiervan is een project bij het Nederlands Instituut voor Wetenschappelijke Informatiediensten (NIWI) te Amsterdam, waarbij een webgebaseerd softwaresysteem ontwikkeld wordt. Door deze software testgestuurd te ontwikkelen hebben we een goed dekkende testsuite (verzameling unittests) opgebouwd.

Elke nacht wordt er automatisch een build gedaan, die de meest recente versie van de broncode compileert en de resulterende software beschikbaar stelt voor testgebruikers. Een onderdeel van deze nightly build is het uitvoeren van de gehele testsuite. De resultaten hiervan worden per e-mail naar alle leden van het ontwikkelteam gestuurd. Zo zien we elke ochtend of er nog openstaande problemen of fouten zijn.

De zeer uitgebreide testsuite die ontstaat als resultaat van het toepassen van testgestuurd ontwikkelen helpt ons verder om portabiliteitsproblemen vroegtijdig op te sporen en te verhelpen. We ontwikkelen namelijk op verschillende platformen (Windows 98 en 2000, Linux); voor de webserver wordt Solaris gebruikt. Platformafhankelijkheden zoals verschil in tekensets kwamen aan het licht doordat de testsuite op de ontwikkelmachines succesvol was maar er bij het uitvoeren van testsuite tijdens de *nightly build* tests faalden. Een testsuite beperkt risico's bij het porten van software naar een nieuw platform. De tests die falen op een nieuw platform geven een goede indicatie van de benodigde hoeveelheid werk.

Het gebruik van testgestuurd ontwikkelen heeft geleid tot een zeer laag aantal opgeleverde defecten. We hebben geen moeilijk voorspelbare testfase nodig en zijn in staat om elke twee weken een release te doen. Per release worden gemiddeld één à twee defecten gevonden door gebruikers. Regressie van defecten komt niet voor, doordat we voor een gevonden defect unittests toevoegen.

Het volgen van de testgestuurde ontwikkelaanpak zorgt ervoor dat we in feite op ieder moment correct werkende en grondig geteste software hebben. Dit maakt het mogelijk om continu gewijzigde en toegevoegde code te integreren (uiteraard na het uitvoeren van de testsuite). We hebben dan ook geen aparte integratiefase in het project. Voor zover we integratieproblemen hebben worden deze altijd binnen enkele minuten opgelost.

Door testgestuurd te ontwikkelen is de code zeer klein gebleven in verhouding tot de functionaliteit. Daarnaast heeft de code een hoge mate van modulariteit gekregen en is de software goed toegankelijk voor ontwikkelaars. Dit bleek uit het feit dat een nieuwe ontwikkelaar zich zeer snel kon inwerken en al binnen enkele dagen productief was. Verder is de software na verloop van tijd uitgebreid met een gedeelte voor databaseontsluiting. Dit gedeelte kon eenvoudig en met minimale afhankelijkheden aan de bestaande code worden toegevoegd.

geschreven als een stuk programmacode. Vervolgens programmeer je de eenvoudigste implementatie die aan de test zou moeten voldoen. Dan voer je de test uit om te controleren of de implementatie inderdaad aan de test voldoet. De test en de implementatie van een component kunnen banaal en nog onvolledig zijn. Stap voor stap ga je de test en de implementatie uitbreiden, totdat er iets ontstaat dat wel iets zinnigs doet. Een ontwikkelaar verwisselt hierbij steeds zijn ontwikkelaarspet voor zijn testerspet. Op het moment dat de component zinnige functionaliteit bevat en de tests slagen, zet de ontwikkelaar zijn onderhoudspet op en herstructureert de component, zodat deze zo eenvoudig mogelijk is en geen duplicatie bevat. Daarna wordt de vers geïntegreerde functionaliteit in het versiebeheersysteem gezet, zodat deze niet verloren kan gaan.

CONTINUE ONTWIKKELING De stappen die we nemen zijn zo klein, dat je activiteiten zoals analyse, ontwerp, implementatie en testen die je in het watervalproces in het groot, en in meeste RAD processen zoals DSDM[2] en Evo[3] in het klein tegenkomt, niet meer als gescheiden, discrete stappen terugziet, omdat ze in de orde grootte van minuten in plaats van dagen of weken plaatsvinden. We noemen dit ook wel *continue ontwikkeling*.

Testgestuurde ontwikkeling is pragmatische aanpak die zich richt op het op efficiënte wijze opleveren van goede software. Het heeft veel minder het risico van *more planning than doing* dan bijvoorbeeld aan het watervalmodel gerelateerde testmethoden en –modellen als TMAP[4]. Geïntegreerde, continue ontwikkeling van testcode en productiecode beperkt de noodzaak tot uitgebreide testplanning en testontwerp.

Testgestuurde softwareontwikkeling is een techniek die in de loop van de tijd in de praktijk is ontstaan. Het is één van de werkwijzen van Extreme Programming, maar het kan ook heel goed los van Extreme Programming worden toegepast. Sinds kort zijn er ook enkele boeken waarin de techniek uitvoerig wordt beschreven, zie hiervoor (5) en (6) in de literatuurlijst. Zie (7) voor een verslag van een project waarbij we de techniek succesvol hebben toegepast.

ERVARINGEN We hebben in onze projecten een aantal positieve eigenschappen van testgestuurde softwareontwikkeling ervaren.

- *Laagdrempeligheid*
  De principes en concepten zijn eenvoudig te leren. Daarnaast zijn voor vele programmeertalen gratis testgereedschappen beschikbaar, zie[7] voor een overzicht. Wij hebben zelf met succes testraamwerken voor Java, Python, Ruby, Perl en Visual Basic gebruikt. Indien nodig zijn de benodigde gereedschappen eenvoudig in een paar dagen zelf te maken en uit te breiden. Samen met een Perl-programmeur hebben we in ongeveer een uur het beperkte testraamwerk voor Perl (genaamd PerlUnit) uitgebreid zodat hij snel eenvoudig kan controleren of gegenereerde HTML code wel voldoet aan de W3C

Truncate in Python

We willen een functie *truncate* maken die een gegeven string afkapt op een gegeven lengte. Voorbeeld: *truncate('een stuk tekst', 8)* levert op: *'een s...'*. We gebruiken de object-georiënteerde taal Python, waar standaard een testraamwerk meegeleverd wordt.

De verschillende testgevallen maken deel uit van een klasse die afgeleid wordt van de standaardklasse TestCase. We gaan verder niet in op de details van het raamwerk. Elk testgeval wordt geschreven als functie op die klasse.

```
import unittest

class
TruncateTest(unittest.TestCase):

   def testTruncateEmptyString
   (self):
     self.assertEqual('', trun-
     cate('', 4) )

unittest.main()  # voert alle
                 bovenstaande test-
                 gevallen uit
```

We hebben hier het eenvoudigste testgeval geformuleerd: een lege string en een lengte van 4 moet een lege string opleveren. De functie assertEqual controleert of beide argumenten gelijk zijn. De eenvoudigste implementatie die dit testgeval laat slagen is:

```
def truncate( text, length ):
  return ''
```

Uitvoeren van de test levert op:

```
Ran 1 tests in 0.000s

OK
```

We voegen een test toe voor het geval dat de tekst korter is dan de opgegeven maximale lengte. De tekst moet onveranderd worden opgeleverd:

```
def testNoTruncation(self):
  self.assertEqual('een tekst',
  truncate('een tekst', 12) )
```

De test faalt:

```
AssertionError: 'een tekst' != ''

Ran 2 tests in 0.020s

FAILED (failures=1)
```

We moeten de implementatie van de truncate functie dus aanpassen en kiezen weer voor de eenvoudigste oplossing. Deze werkwijze lijkt op het eerste gezicht wellicht simplistisch, maar werkt effectief om onnodige complexiteit te voorkomen.

```
def truncate( text, length ):
  return text
```

De test slaagt nu:

```
Ran 2 tests in 0.030s

OK
```

Werkt de huidige implementatie ook voor het randgeval waarbij de lengte van de string precies gelijk is aan de opgegeven lengte?

```
def testTextAsLongAsLength(self):
  self.assertEqual('meer tekst',
  truncate('meer tekst', 10) )
```

Het werkt naar behoren:

```
Ran 3 tests in 0.030s

OK
```

Als de string langer is dan de gegeven lengte, moet deze correct worden afgebroken:

```
def testTruncate(self):
  self.assertEqual('nog meer...',
  truncate('nog meer tekst', 11) )
```

Deze test faalt in eerste instantie, dus we passen de implementatie aan.

```
def truncate( text, length ):
  if len(text) > length:
    return text[0 : length-3] +
    '...'

  return text
```

De tests slagen nu. Er is nog een aantal randgevallen te onderscheiden. Werkt de functie bijvoorbeeld correct als de gegeven lengte net kleiner is dan de lengte van de string?

```
def
testTextAndTruncationAsLongAsLengt
h(self):
  self.assertEqual('meer t...',
  truncate('meer tekst', 9) )
```

Wat als de opgegeven lengte kleiner is dan de lengte van de afbreektekst? We kiezen ervoor dat dan alleen de betreffende tekens van de string opgeleverd moeten worden, zonder afbreektekst.

```
def
testTruncationTextLargerThanLength
(self):
  self.assertEqual('vo', trun-
  cate('voorbeeld', 2) )
```

De huidige implementatie blijkt correct te werken voor het eerste geval maar niet voor het tweede geval. De implementatie van truncate wordt hiervoor aangepast:

```
def truncate( text, length ):
  if length < 3:
  return text[0 : length]

  if len(text) > length:
    return text[0 : length-3] +
    '...'

  return text
```

In de truncate functie wordt op drie plekken verwezen naar de afbreektekst of de lengte ervan, zonder dat de afhankelijkheden tussen deze drie lokaties expliciet is. Dit kunnen we verbeteren door hiervoor constanten te introduceren:

```
truncationText = '...'
truncationTextLength =
len(truncationText)

def truncate( text, length ):
  if length <
truncationTextLength:
  return text[0 : length]

  if len(text) > length:
    return text[0 : length -
    truncationTextLength] +
    truncationText

  return text
```

We voeren de tests nog een keer uit zodat eventuele gemaakte fouten zichtbaar worden.

```
Ran 6 tests in 0.030s

OK
```

Alles werkt zoals het hoort.

standaard. Bij een bijeenkomst van de Nederlandse Extreme Programming gebruikersgroep[9] hebben we in twee uur de basis gemaakt voor een testraamwerk voor de 4GL-taal Progress.

- *Snelle feedback*
  Testgestuurde ontwikkeling kenmerkt zich door snelle en directe feedback. Veel defecten worden zeer snel na het introduceren ervan gevonden of worden zelfs voorkomen. Het gebruik van een debugger om defecten te vinden is zelden meer nodig. Als een debugger wel nodig is, vind je binnen enkele stappen het probleem. Dit kan, omdat er een test faalt, deze al dicht bij de oorzaak zit. Het proces van het lokaliseren en oplossen van defecten wordt hierdoor beter beheersbaar, wat ook voor projectmanagers goed nieuws is. Verder kan regressie van fouten voorkomen worden door voor elk gevonden defect meteen een testgeval te schrijven. Als een defect terugkeert als gevolg van een wijziging, wordt dit meteen zichtbaar bij het uitvoeren van die test.

- *Veranderingsbestendige software.*
  Je kunt wijzigingen aanbrengen zonder dat de software instort. De invloed van wijzigingen wordt direct zichtbaar bij het uitvoeren van de tests. Dit zorgt ervoor dat je beter op veranderende wensen van de klant kunt inspelen. De verzameling tests kunnen bijvoorbeeld gebruikt worden voor een what-if analyse, door wat code aan te passen en te kijken hoeveel tests er dan falen. Op basis hiervan kan een inschatting gemaakt worden van de tijd die de wijziging gaat kosten.

- *Hoge dekkingsgraad*
  Er is altijd werkende software met een hoge test coverage. Doordat tests en code samen groeien, zijn er geen ongeteste componenten. De hoge dekkingsgraad is een neveneffect van testgestuurde softwareontwikkeling. Je hoeft je nauwelijks af te vragen wat je wel en niet gaat testen. Je kunt dus meer tijd besteden aan doen omdat je minder tijd kwijt bent met plannen.
  Als je de tests laat vertellen wat er ontwikkeld moet worden, blijkt de implementatie kleiner en eenvoudiger dan je voor mogelijk hield. Je bouwt alleen componenten die begrijpelijk en bruikbaar zijn en overbodige en niet-gebruikte code wordt voorkomen. De afhankelijkheden tussen componenten worden beperkt en de structuur van de software wordt beter ontkoppeld.

- *Structuur in het werk*
  Als ontwikkelaar weet je waar je aan toe bent. Je kunt in hele kleine stappen werken waarbij je steeds weer iets aan de software toevoegt. Integreren met het werk van andere ontwikkelaars is zo eenvoudig, dat het meerdere keren per dag gedaan kan worden. Bij elke stap weet je zeker dat het werkt en dat het geheel blijft werken. Testgestuurde softwareontwikkeling helpt minder ervaren ontwikkelaars om tot een goede structuur te komen en om beter te leren ontwikkelen. Voor ervaren ontwikkelaars is het een goed hulpmiddel om hun kennis en ervaring bijvoorbeeld op het gebied van ontwerppatronen gerichter en efficiënter in te zetten.

KOSTEN EN RANDVOORWAARDEN Het aanleren van testgestuurd ontwikkelen vraagt enige discipline. Als het eenmaal toegepast wordt, is het makkelijker om het te blijven doen. De discipline is met name nodig om jezelf te dwingen steeds eerst de test te schrijven en om kleine stappen te werken terwijl je toch het beeld van waar je heen wilt vasthoudt. Als je dat niet doet merk je het snel genoeg. Het werken in kleine stappen is in de praktijk lastiger dan het lijkt.

Het leren van de concepten van testgestuurde ontwikkeling vraagt enkele uren van ontwikkelaars om zich in te lezen in de materie. De beste manier om testgestuurd ontwikkelen te leren is het toepassen ervan. Dit vraagt wel enige ruimte om te experimenteren. Een goede oefening is enkele uren te besteden aan het naprogrammeren van een voorbeeld zoals de *test-first*

challenge[10] of het bouwen van een stukje nieuw systeem om het in de eigen praktijk uit te proberen.

Modulariteit Testgestuurde ontwikkeling is lastig toe te passen bij bestaande software die niet goed testbaar is, bijvoorbeeld omdat afhankelijkheden niet goed beheerst zijn. Wij hebben software gezien waarbij alles van alles afhankelijk is en code voor databasetoegang, user-interface en applicatielogica overal door elkaar staat. Dit probleem is echter niet specifiek voor testgestuurde softwareontwikkeling: sterk gekoppelde software is altijd lastig te onderhouden.

Er is een programmeertaal nodig met een zekere mate van modulariteit, zoals Java, Python, Perl, Delphi, C++, C# of Visual Basic. Binnen de programmeeromgeving moet de cyclus van aanpassen, compileren, test uitvoeren voldoende snel uitgevoerd kunnen worden. De hele verzameling tests moet in beperkte tijd uitgevoerd kunnen worden. Bij voorkeur wil je dit vele keren per dag doen om feedback te krijgen op alle wijzigingen en toevoegingen. Op zijn minst moet het uitvoeren van alle tests onderdeel worden van een automatische build-procedure.

Conclusie Testgestuurde softwareontwikkeling is zowel een testtechniek als een continue ontwikkeltechniek, waarbij testen, ontwerpen en implementeren een geïntegreerde activiteit zijn. Componenten worden incrementeel ontwikkeld, door steeds een testgeval te programmeren voor de component en vervolgens de eenvoudigste implementatie te schrijven die aan die test voldoet. In de inleiding schetsten we een aantal redenen waarom om testen geen integraal onderdeel van het ontwikkelproces is. Hoe helpt testgestuurde softwareontwikkeling deze problemen op te lossen?

Testgestuurde softwareontwikkeling zorgt voor beter gestructureerde software met minder afhankelijkheden. Softwarecomponenten zijn van begin af aan begrijpelijk, testbaar en bruikbaar. Dit maakt testen makkelijker. De tests groeien met de implementatie mee, of beter gezegd, de implementatie groeit met de tests mee.

Met testgestuurde softwareontwikkeling bespaar je tijd. Tijd die eerst werd besteed aan ontwerpen wordt nu besteed aan het ontwikkelen van tests. Het opsporen en repareren van defecten kost veel minder tijd en gebeurt in een zeer vroeg stadium, namelijk op het moment dat de programmacode wordt geschreven.

Testen is volledig geïntegreerd in het ontwikkelproces en vindt continu plaats. Functionaliteit die af is, is ook getest. Bezuinigen op testen is niet meer nodig, omdat testen niet meer een onbeheersbare fase aan het eind van het project is.

Testgestuurd ontwikkelen maakt testen leuk. Het geïntegreerd ontwikkelen van tests helpt ontwikkelaars om vanaf het begin goede software te bouwen. De tests stellen hen in staat om het programma ook goed te houden, zodat ze tijd over houden om voor hun klanten waardevolle functionaliteit toe te voegen.

Samengevat is testgestuurde softwareontwikkeling een laagdrempelige werkwijze die het mogelijk maakt om relatief eenvoudig te beginnen met gestructureerd en grondig testen. Het vangnet van tests en de betere kwaliteit van de software zorgen ervoor dat het bouwen, uitbreiden en wijzigen van software beter beheersbaar wordt, hetgeen testgestuurde softwareontwikkeling aantrekkelijk maakt voor ontwikkelaars, projectmanagers en klanten.

> Testen lijkt niet direct tastbaar resultaat op te leveren, de investering in testtijd op korte termijn is echter wel zichtbaar

Literatuur

1 *Gebruikers dragen meeste kosten van fouten in software*, Automatiseringsgids 5 juli 2002
2 Jennifer Stapleton, DSDM, *Dynamic Systems Development Method - De methode in de praktijk*, Academic Service
3 T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley 1988
4 M. Pol, E. van Veenendaal, R. Teunissen, *Testen volgens TMAP*, Tutein Nolthenius 1995
5 Johannes Link, *Unit Tests Mit Java: Der Test-First Ansatz*, D-Punkt Verlag 2002
6 Kent Beck, *Test Driven Development by Example*, Addison Wesley 2002
7 *SkopeoPro: Extreme Programming in Practice*, http://www.cq2.org/en/SkopeoPro
8 Unit Testing Frameworks: http://www.xprogramming.com/software.htm
9 XP-NL: http://www.xp-nl.org
10 William Wake, *Test-First Challenge*, januari 2002, http://www.xp123.com/xplor/xp0201

Marc Evers en Willem van den Ende

ir. Willem van den Ende (mail@willemvandenende.com) is werkzaam als software development coach bij CQ2 (http://www.cq2.nl)

ir. Marc Evers (evers@livingsoftware.org) is werkzaam als adviseur bij het Software Engineering Research Centre (http://www.serc.nl)

De auteurs willen de volgende personen bedanken voor hun commentaar: Erik Groeneveld, Geert Lobbestael, Frank Niessink, Peter Schrier en Joris van Zundert