Methodology

Unit testen is inmiddels een bekende methode om in een vroeg stadium te testen. Unit testen is echter moeilijker dan het op het eerste gezicht lijkt. Het vereist veel discipline, moet passen in de architectuur en is ook belangrijk tijdens het onderhoud van het systeem. Unit testen kan daarom niet als een klein detail worden beschouwd, maar is echt een 'way of life', waar het gehele team achter moet staan.

De waarde van goede unit tests

Goed uitgevoerd echt een toevoeging

Peter Hendriks
is ontwikkelaar/ontwerper van Java Enterprise en Eclipse RCP applicaties bij Info Support.

Unit testen is een vorm van testen waarbij een zeer klein onderdeel ('unit') van het systeem geïsoleerd en geautomatiseerd wordt getest. Door een zeer klein onderdeel te testen zijn unit testen eenvoudig, kunnen ze in een heel vroeg stadium al worden uitgevoerd en blijven ze ook werken als het systeem als geheel niet meer functioneert. Dit sluit perfect aan bij de wilde stroom van veranderingen in een typisch softwareontwikkelproject.

Problemen

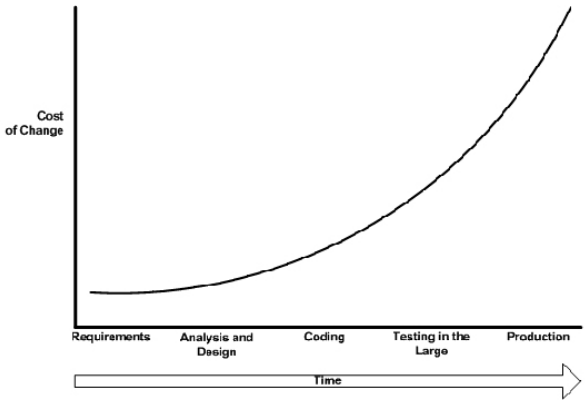
Veel teams die aan de slag gaan met unit testen krijgen na een tijd allerlei problemen met hun testen:

- Testen uitvoeren duurt erg lang.
- Er zijn altijd falende testen.
- Testen schrijven/onderhouden kost veel tijd.
- Het is onduidelijk wat de testen nu eigenlijk wel controleren, en wat niet.
- Niemand weet wat de actuele stand van de testen is.

Een moeilijke eerste stap is erkennen dat deze situaties ook echt een probleem zijn. Er wordt bijvoorbeeld al snel gedacht: "ach, de testen duren wat langer, maar dan voeren we ze gewoon minder vaak uit." Hierdoor neemt echter de tijd tussen het ontstaan van de fout en het detecteren fors toe, waardoor het veel duurder wordt om vast te stellen waar het probleem zit. Het is belangrijk om scherp te hebben wat de voordelen van unit testen zijn en aan welke randvoorwaarden unit testen moeten voldoen om deze voordelen ook echt te behalen.

'Aim small, miss small'

Doordat unit testen in een vroeg stadium worden opgesteld, worden fouten ook zeer snel gedetecteerd. De tijd tussen het ontstaan, het detecteren en het oplossen van een fout heeft veel invloed op de kosten van het oplossen ervan. Hoe eerder de fout wordt gevonden, hoe goedkoper het is om de fout op te lossen.



Figuur 1: Kosten aanpassing nemen snel toe.

Goed uitgevoerde unit testen hebben ook andere voordelen. Ze dwingen een meer ontkoppelde architectuur, die eenvoudiger en flexibeler is af. Daarnaast zijn ze zijn een

5 UIT-BLINKERS GEZOCHT

Wij zoeken:

Java software engineers (3x)

Als Java software engineer speel je een belangrijke rol in de ontwikkeling van ons online marketingplatform. Je bent bezig met het gehele traject van specificatie , ontwerp, realisatie, testen tot implementatie. Je hebt tenminste één jaar werkervaring met het bouwen van Java (web)applicaties . Ook is ervaring met (D)HTML, XML, CSS en Javascript en kennis van Linux, MySQL, Apache en Tomcat vereist.

Test engineer

Als Test engineer ben je verantwoordelijk voor het plannen, opzetten en (geautomatiseerd) uitvoeren van testscripts. Je hebt ervaring met het assisteren van testers, met het maken van fysieke testscripts en met het vertalen van handmatige scripts naar geautomatiseerde scripts.

Projectleider

Als Projectleider ben je verantwoordelijk voor complexe klanttrajecten en integratie van het platform met externe systemen (o.a. CRM- en CMS-systemen). Je kunt goed klantproblemen analyseren, bent sterk in het bedenken van praktische oplossingen en weet deze oplossingen zowel schriftelijk als mondeling goed over te brengen.

Blinker is een snel groeiende e-mail marketing service provider in Zoetermeer. We werken met een team van 26 gedreven medewerkers hard aan de ontwikkeling en levering van een innovatief e-mail marketingplatform (SaaS).

Wegens onze ambitieuze plannen zijn we op zoek naar drie goede en gedreven Java software engineers, een Test Engineer en een Projectleider. Je werkt samen met een team van vier zeer ervaren engineers aan de ontwikkeling van onze applicaties die door meer dan 1000 organisaties worden gebruikt.

We verwachten van onze nieuwe medewerker dat hij accuraat is, graag wil samenwerken en open staat voor andere meningen. Met een goed gevoel voor humor en een drive om vakinhoudelijk wat neer te zetten. Voor alle functies vragen we minimaal een opleiding op HBO-niveau.
Kijk voor meer informatie op www.werkenbijblinker.nl

BLINKER
BRENGT KLANTEN DICHTBIJ

BLINKER BV BORIS PASTERNAKLAAN 16, 2719 DA ZOETERMEER T: 079 - 363 70 60 F: 079 - 363 70 61 E: INFO@BLINKER.NL WWW.BLINKER.NL

goede bron van documentatie voor ontwikkelaars die de code onderhouden. Tenslotte reduceren unit testen onzekerheid in de algemene projectplanning, omdat simpele fouten snel worden opgelost.

Randvoorwaarden voor succes

Unit testen moeten aan een reeks van randvoorwaarden voldoen om deze voordelen ook echt te behalen.

De unit testen moeten:

- Zeer vroeg worden opgesteld.
- Voldoen aan dezelfde hoge kwaliteitseisen als productiecode.
- Eenvoudig zijn op te stellen en te onderhouden.
- Snel resultaat opleveren.
- Vaak worden uitgevoerd.
- Snel worden gerepareerd als een test faalt.
- Voor het grootste deel van het systeem de eerste testdekking bieden.

Dit zijn harde voorwaarden en hier moet actief op worden gestuurd. Zonder de hulp van de architect, duidelijke kwaliteitsfocus vanuit het management en een ijzeren discipline binnen het ontwikkelteam is het moeilijk om unit testen succesvol uit te voeren. 'Slechte' unit testen hebben een fors lager rendement en zullen soms zelfs meer kwaad dan goed doen.

Slechte unit tests hebben een fors lager rendement en doen soms zelfs meer kwaad dan goed.

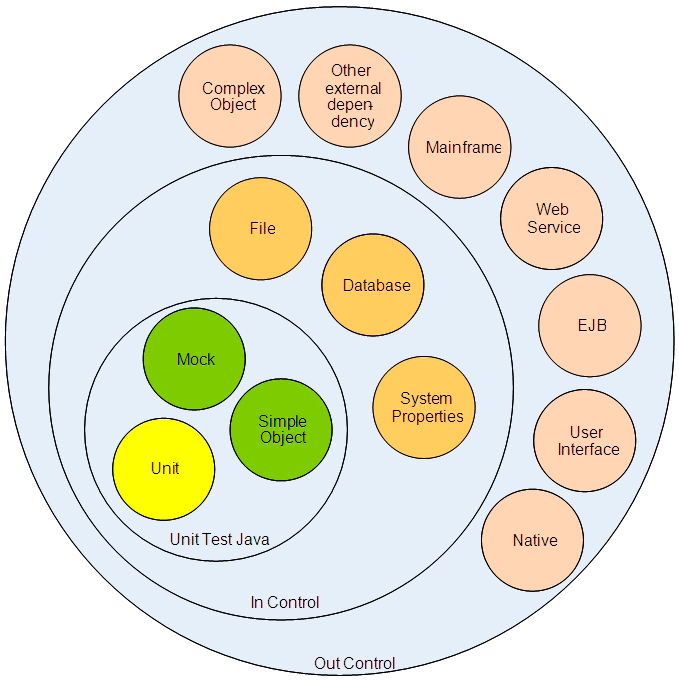
Testbare Architectuur

Het is belangrijk om in de architectuur rekening te houden met de verschillende teststrategieën binnen het project. Omdat unit testen zich niet op de officiële randen van het systeem bevinden, maar midden tussen de code zitten, vereisen ze meer aandacht op een lager niveau. Om goed te kunnen unit testen moet er een goede ontkoppeling zijn tussen klassen. Ook is een goede isolatie van complexe externe componenten, zoals applicatieservers, databases, web services en filesystemen vereist.
Het aanpassen van de architectuur voor een interne projectkeuze als unit testen kan verkeerd aanvoelen. 'Testbaarheid' moet echter simpelweg als een eis worden beschouwd: net zoals een goede performance, security of het gebruik van specifieke frameworks beïnvloedt het de uiteindelijke architectuur.

Afhankelijkheden mocken

Tijdens het unit testen wordt er zo veel mogelijk de focus gelegd op een specifiek stukje code. Deze code zal vaak afhankelijk zijn van andere delen van het systeem.

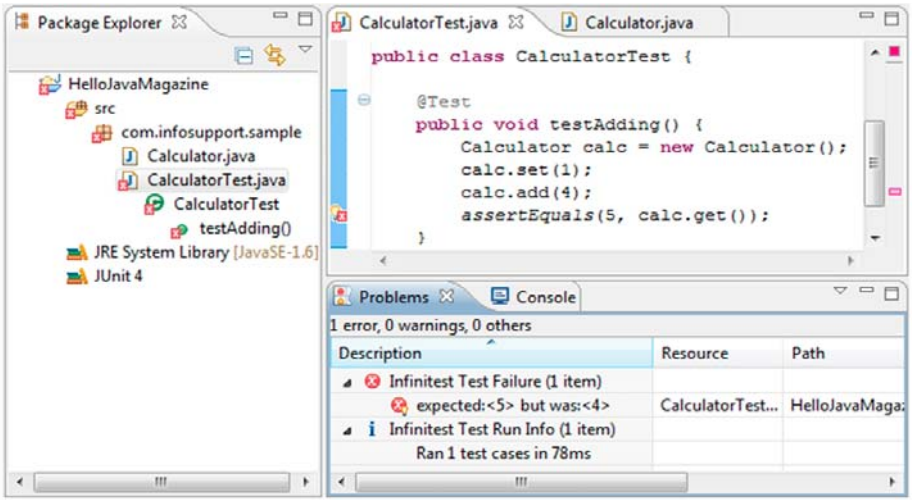
Figuur 2 geeft een overzicht van dependencies die zeer gemakkelijk werken in een unit test (groen), dependencies die lastiger, maar te beheersen zijn (oranje), en dependencies die nooit in een unit test mogen worden gebruikt (rood). Om te voorkomen dat tijdens de test een groot deel van het systeem nodig is worden afhankelijkheden 'gemocked'. Een 'mock' is een simpele vervanging van de afhankelijkheid, die met voorgeprogrammeerd gedrag de test kan voeden zonder dat de echte afhankelijkheid nodig is. Om mocks te maken zijn prima standaardoplossingen beschikbaar, zoals het opensource EasyMock framework [EASYMOCK].
De mock moet ingebracht worden in het systeem tijdens het testen. Een populair concept om dit voor elkaar te krijgen is 'Dependency Injection' (DI) waarbij afhankelijkheden in het object worden 'gespoten' met standaard methode- of constructor parameters. Dit maakt het heel simpel om tijdens unit testen op andere wijze dependencies te maken. DI is daarom heel populair, met ondersteuning in veelgebruikte standaarden en frameworks als Java EE 5, Guice [GUICE] en Spring [SPRING]. Zie het codevoorbeeld in figuur 3.



Figuur 2: Voor unit testen moeten dependencies expliciet worden beheerst.

```
package test;
import static org.easymock.EasyMock.*;
import org.junit.*;
public class CustomerServiceTest {
  private CustomerService service;
  private CustomerDao dao;
  @Before public void setUp() {
    service = new CustomerService();
    // Maak een mock voor Data Access Object
    dao = createMock(CustomerDao.class);
    // Injecteer de dao in de service
    // De service zal deze dao gebruiken,
    // ipv zelf een dao maken.
    service.setCustomerDao(dao);
  }
  @Test public void testAdd() {
    Customer customer = new Customer();
    // Mock gedrag "opnemen".
    dao.insert(customer);
    // Mock in afspeelstand zetten.
      // De mock zal nu de opgenomen acties
uitvoeren.
    replay(dao);
    service.add(customer);
   // Controleren of alle acties op de mock zijn
uitgevoerd.
    verify(dao);
  }
}
```

Figuur 3: codevoorbeeld mocken en DI.



Figuur 4: Infinitest toont falende testen net als compileerfouten.

Vroeg, snel en vaak

De unit test dient vroeg te worden opgesteld, liefst vóórdat de productiecode is geschreven. Dit wordt ook wel 'Test Driven Development' genoemd: schrijf één test, schrijf genoeg productiecode om de test te laten slagen en schrijf vervolgens weer een nieuwe test. Veel ontwikkelaars hebben moeite met deze strikte filosofie, omdat ze mentaal gewend zijn nieuwe code te schrijven, maar geen test voor code die niet bestaat. Probeer het in dit geval andersom: schrijf één minimale productiemethode, schrijf hier vervolgens de test voor, enzovoorts. Het gaat er vooral om dat productiecode binnen enkele minuten wordt getest en dat geen code zonder test wordt opgeleverd.

Als de test eenmaal is opgesteld, zal deze talloze malen worden uitgevoerd. Het is belangrijk dat een unit test zeer snel resultaat oplevert, want als teveel testen langzaam zijn zal het lang duren voordat feedback beschikbaar is. Door effectief te mocken en een kleine gegevensset tijdens unit testen te gebruiken zal de test meestal vanzelf zeer snel zijn. Een vervelende uitzondering is testen waarbij een vorm van multi-threading wordt gevalideerd, waarin met 'sleep' statements wordt gewerkt om te garanderen dat bepaalde condities zouden zijn bereikt. Dit is meestal formeel niet correct en vertraagt de testen. Een alternatief is het MulithreadedTC [MTC] framework, dat op basis van een 'metronoom' werkt. Dit is robuuster en veel efficiënter.

Als de test eenmaal is opgesteld, moet deze zo vaak mogelijk worden uitgevoerd, om zo snel te detecteren wanneer een fout in het systeem is geslepen. Een goede, bewezen aanpak is een automatisch buildsysteem, dat bij iedere opgeleverde wijziging van de ontwikkelaar alle testen uitvoert.

Test on Save

Er zijn sinds kort ook experimentele tools die al tijdens het ontwikkelen automatisch de testen uitvoeren binnen de ontwikkel-

Ondanks de eenvoud is succesvol unit testen een uitdaging.

omgeving na iedere aanpassing, waardoor een 'test-on-save' effect ontstaat. De meest uitgebreide tool op dit moment is Infinitest [INFINITEST], die werkt al voor kleinschalige projecten. Met 'test-on-save' is al na enkele seconden automatisch een foutmelding zichtbaar als er een test is gebroken, en biedt hiermee enorm snel feedback.

Afspraken en transparantie

De introductie van unit testen is vaak stroever dan gedacht. Het vereist medewerking van ontwikkelaars, die eerst niet expliciet hoefden te testen en dit soms ook niet als onderdeel van hun werk beschouwen. Ook zal het tijd en moeite kosten genoeg ervaring op te bouwen om comfortabel en vlot testen te schrijven. Het helpt om een ervaren teamlid als coach beschikbaar te hebben. Ook zijn duidelijke afspraken nodig, zodat iedereen op een vergelijkbare manier de testen opstelt.

Nog moeilijker is het volhouden van unit testen. Even die laatste testen laten zitten om toch vandaag op te kunnen leveren is verleidelijk. En soms is het moeilijk als management om de keus te maken om falende testen op te lossen in plaats van door te werken aan nieuwe features.

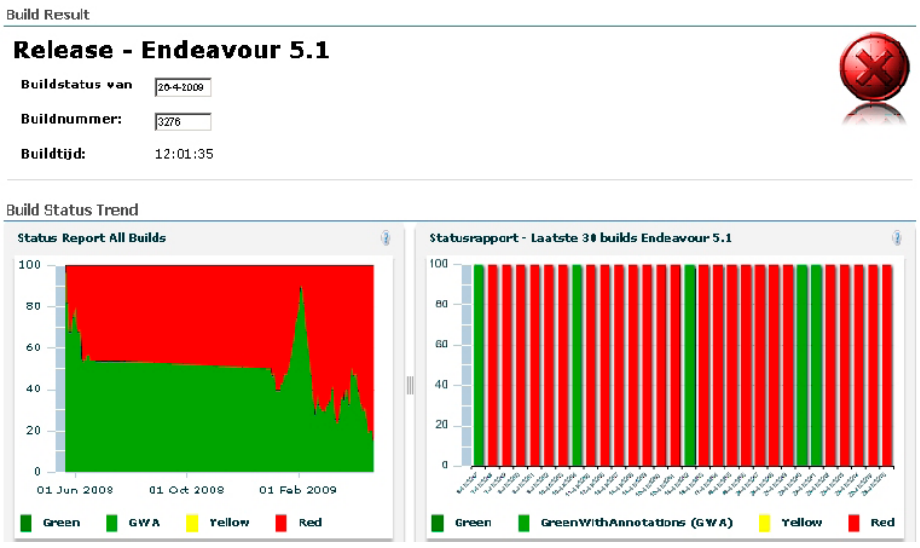
Om hierbij te helpen kan de status en kwaliteit van testen visueel gemaakt worden met tabellen, grafieken en rode/groene ballen. Door de testresultaten van de geautomatiseerde build worden deze overzichten continu bijgewerkt, waardoor steeds zichtbaar is wat de actuele stand van de testen is.

Het is gemakkelijk om even niet naar een diep verstopt testrapport te kijken, maar de sociale druk van een knalrode bal op de project website is een stuk groter. Doordat de status zichtbaar is, wordt ook het oplossen van falende testen een zichtbaar resultaat dat samen gevierd kan worden.

Meten is weten

Om vast te stellen welke code tijdens de unit testen wordt geraakt, wordt coverage tooling gebruikt. Goede gratis tools zijn EclEmma [ECLEMMA] voor Eclipse en Cobertura [COBERTURA] voor geautomatiseerde testen. Metingen voor code-kwaliteit, zoals Checkstyle [CHECKSTYLE] of FindBugs [FINDBUGS] dienen ook voor testcode te worden uitgevoerd. Met deze tools kan worden vastgesteld waar testen helemaal geen dekking bieden, of niet onderhoudbaar zijn opgesteld.

Het meten van unit test resultaten is niet alleen belangrijk om af te dwingen dat het op een consistente manier gebeurt. Op basis van unit test rapportages kunnen al in een vroeg stadium objectieve analyses worden gemaakt over de voortgang en kwaliteit van een project. Bij een stijgend aantal testen is het bijvoorbeeld erg waarschijnlijk dat er succesvol nieuwe functionaliteit wordt gebouwd. Als er in een bepaalde periode regelmatig veel testen falen is dit een teken dat er stevig gesleuteld wordt in de code. Dit kan een geruststellende bevestiging zijn van het algemeen beeld, maar het kan ook verborgen ontwikkelingen boven water halen. Als bijvoorbeeld de verwachting is dat er nieuwe functionaliteit wordt gebouwd, maar het aantal testen neemt niet toe, dan is er iets niet in orde. Een testrapport waarin al maanden geen enkele test heeft gefaald kan twee dingen betekenen: de code is perfect of de testen controleren te weinig.



Figuur 5: Overzicht status testen.

Conclusie

Ondanks de eenvoud is succesvol unit testen een uitdaging. De architectuur moet toestaan dat afhankelijkheden goed kunnen worden gescheiden en de noodzaak voor complexe infrastructuur minimaal is. Er zijn heldere afspraken nodig met duidelijke, actuele overzichten, zodat iedereen weet waar ze aan toe zijn. En de manager moet begrip tonen als ontwikkelaars niet met nieuwe dingen bezig zijn, omdat de testen nog steeds falen.

Als het allemaal lukt levert unit testen echter een schat aan toegevoegde waarde op. Een hogere kwaliteit in een vroeg stadium van het project. Een flexibele, makkelijk te ontkoppelen architectuur, die gemakkelijk kan worden aangepast doordat unit testen continu een goede werking borgen. Een bron aan levende voorbeelden voor ontwikkelaars die willen nakijken wat een stuk code nu écht moet doen. Een soepeler vervolgtraject in latere testen omdat de simpele fouten er al uit zijn. En een actuele graadmeter voor het management over de voortgang en kwaliteit. Goed unit testen kost moeite en vereist samenwerking, maar iedereen heeft er plezier van. «

Referenties

CHECKSTYLE: http://checkstyle.sourceforge.net/
COBERTURA: http://cobertura.sourceforge.net/
EASYMOCK: http://easymock.org/
FINDBUGS: http://findbugs.sourceforge.net/
INFINITEST: http://infinitest.org
GUICE: http://code.google.com/p/google-guice/
MTC: http://code.google.com/p/multithreadedtc/
SPRING: http://www.springsource.org/

30 september en 1 oktober 2009 • Hotel Lapershoek Hilversum

Pragmatisch modelleren met UML

met Sander Hoogendoorn

- U leert het gebruik van use cases
- U krijgt inzicht in de samenhang tussen de verschillende modelleertechnieken
- U gaat zelf aan de slag met UML in een pragmatische case
- Zeer interactieve workshop met maximaal 20 deelnemers
- Voor iedere deelnemer gratis het boek ‘Pragmatisch modelleren met UML 2.0’
- Deelnemers waardeerden de vorige editie met een 8,2!

De modelleertaal UML is uitgegroeid tot de wereldwijde standaard voor het modelleren van requirements, functionaliteit, componenten en services. UML 2.0 kent een groot aantal modelleertechnieken zoals use case diagrammen, activity diagrammen, sequence diagrammen, communication diagrammen, class diagrammen en package diagrammen.

Tijdens de tweedaagse workshop leert u niet alleen een aantal essentiële vaardigheden voor het modelleren met UML (en andere modelleertechnieken), maar u zet tegelijk zo’n pragmatische werkwijze neer. In deze workshop met veel aandacht voor de praktijk kunnen de deelnemers met diverse oefeningen de verschillende gezichtspunten van een prikkelende case uitwerken – een online dating website. Daarbij wordt veel aandacht besteed aan de nauwe relatie tussen de verschillende modelleertechnieken en het gewenste detailniveau van de diagrammen. Bovendien wordt de link gelegd naar de ontwikkeling en het testen van de software op basis van de geproduceerde diagrammen en (servicegeoriënteerde) applicatiearchitecturen.

Kortom: in deze praktische en interactieve workshop wordt UML in al zijn facetten en toepassingsmogelijkheden behandeld.

KORTING!

Profiteer van vroegboekvoordeel en korting bij meerdere deelnemers van één bedrijf!

DATUM	**30 september en 1 oktober 2009**
LOCATIE	**Hotel Lapershoek Hilversum**
TIJD	**Van 9.30 uur tot 17.00 uur**
REGISTRATIE	**www.arrayseminars.nl**

Kijk snel op www.arrayseminars.nl voor het complete programma!

Array SEMINARS

COMPUTABLE