In de boeken van Kent Beck worden twee rollen besproken die van belang zijn binnen de agile ontwikkelmethodiek eXtreme Programming (XP): die van programmeur en die van klant. Deze polarisatie dwingt respect af vanwege de duidelijkheid, maar veroorzaakt ook verwarring. Want waar blijven de traditionele rollen als projectleider, ontwerper, tester en beheerder? De auteurs bespreken in dit artikel het testen binnen agile projecten op basis van hun ervaringen binnen eXtreme Programming projecten.

Agile testen in de praktijk

Acceptatie- en unittests

Binnen XP worden twee testmomenten expliciet genoemd en onderkend: unittests en acceptatietests. De unittest is door de programmeur en voor de klant, de acceptatietest is voor en door de klant. Daarnaast wordt ook nog eens de nadruk gelegd op het eerst schrijven van de unittests en daarná pas de code. Dit alles wordt binnen andere agile methoden ook toegepast en aangemoedigd. Wat betekent dit in de praktijk voor de kwaliteit van de code, en welke invloed heeft dit op de rol van een tester? We zullen hier op ingaan zowel vanuit het gezichtspunt van de ontwikkelaar als dat van de tester.

Op de pijnbank De tester is iemand die, hoe je het ook wendt of keert, een sceptische kijk op het product heeft. Hij zal zich niet laten verleiden tot “het zal wel” of “dat gaat wel goed”. Hij zal het zeker willen weten, en legt soms het systeem daarvoor op de pijnbank. Dit is een uniek geluid in software ontwikkeling - wat immers probleemoplossend is, en niet probleemtoetsend of zelfs probleemcreërend is. Dit is niet anders voor agile projecten.

Van oudsher is de tester ook iemand die de taal van de klant binnen het project spreekt, dat maakt hem of haar uitermate geschikt voor het doen van acceptatietests. Die tests testen immers de acceptatiecriteria van de klant, en staan dus ook - naast het eindresultaat zelf - het dichtste bij de klant. In de ideale situatie moet een tester zich dus richten op de klant en samen met hem of haar naar het product kijken om te zien of het aan de verwachtingen voldoet. Samen met de klant kan de tester informatie verkrijgen over hoe goed de software voldoet aan de verwachtingen. Met name de niet-functionele eisen zoals performance, gebruiksgemak of beveiliging kunnen hierbij aan de orde komen en als feedback het ontwikkelproces ingaan teneinde de organisatie te laten leren, het product te verbeteren en de klant nog beter van dienst te zijn.

De praktijk Het komt nogal eens voor - in de ervaring van de auteurs gebeurt het zelfs in de meeste gevallen - dat van bovenstaand proces weinig terecht komt. De organisatie leert niet, de software wordt niet beter en de klant wordt niet beter bediend. Eén van de hieraan ten grondslag liggende aspecten is dat zowel de tester als de klant voortdurend bezig zijn met andere zaken. Daar kunnen ze niets aan doen, dat overkomt ze gewoonweg: ze struikelen voortdurend over (voorkombare) bugs. Daardoor is er vaak geen tijd meer over om diepgaand naar bijvoorbeeld de acceptatiecriteria of de niet-functionele aspecten te kijken. Soms neemt deze situatie dusdanig ernstige vormen aan dat de symptomen tot doel worden verheven: de testers denken serieus dat hun taak is om ‘zoveel mogelijk bugs te vinden’, en klanten gaan hun acceptatiecriteria stellen in termen van het maximaal aantal ‘known bugs’ dat in het product mag zitten.

Acceptatietests De acceptatietests zijn in principe de verantwoordelijkheid van de klant. Deze dienen de bedrijfsprocessen te verifiëren, en bevatten de acceptatiecriteria. Ook dat klinkt misschien eenvoudig, maar in de praktijk blijkt het lastig te zijn een softwareapplicatie geautomatiseerd te toetsen aan de acceptatiecriteria. Ook is het niet altijd duidelijk wat je wel en niet kunt testen en hoe je dat moet doen. Tenslotte het lastigste probleem: acceptatietests vinden plaats op hoog niveau - een applicatie als geheel wordt getest - en het aantal mogelijke fouten neemt vaak exponentieel toe met het niveau van abstractie. Als een component 10

functies bevat, en een ander component 15, dan kan de combinatie hiervan (een hoger niveau) 150 (10 x 15) combinatiefouten bevatten. Uiteindelijk komt men dan op getallen als tienduizenden fouten in één programma (zie Humphrey), en die maken het testen tot een eindeloze exercitie. Met name het laatste probleem kan zeer effectief bestreden worden door unittests, hetgeen ruimte schept voor de testers om zich te richten op de eerste problemen, samengevat als: hoe test ik wat?

UNITTESTS Een unittest test toont aan of de ontwikkelaar de logica van de applicatie technisch correct heeft geïmplementeerd. Een goede unittest strategie kan testers en klanten lucht geven om zich weer te wijden aan hun eigenlijke taak, het accepteren van de functionaliteit in de software. Onze ervaring is dat wanneer deze strategie eenmaal ingezet is het aantal 'bugs' in de software op het moment dat die bij de tester en de klant terecht komt gereduceerd is tot enkele, of enkele tientallen. Het effect hiervan is enorm. De hele keten van programmeur, via tester, beheerder, distributiekanaal, call-centra, eerste, tweede en derde lijnssupport, wordt verlicht doordat technische problemen minder vaak optreden.

Wat zijn de voordelen van unittests?

- Ze dienen als 'gids' voor de te ontwikkelen code
- Ze toetsen of de code de gewenste functionaliteit bevat
- Ze testen of de code ongewenste situaties correct afhandelt
- Ze worden in dezelfde ontwikkeltaal geschreven als waarin gecodeerd wordt, waardoor de leercurve kleiner is
- Ze dienen voor ontwikkelaars als voorbeeld voor het gebruik van de code. Dit is met name in teams met sterk wisselende samenstellingen erg handig: zo dragen de unittests zeer goed de vastgelegde kennis over
- Door het onderbrengen van alle tests in een framework kunnen de tests herhaalbaar worden uitgevoerd zodat eventuele integratie- en regressiefouten zeer snel ontdekt worden
- Het aanbieden van functioneel stabiele code aan testers, of beter: aan de volledig achterliggende organisatie

Overigens: er is niets nieuws onder de zon. Unittests bestaan al heel lang en worden bij life-critical systemen al jarenlang gebruikt. Als er dus meer vraag is naar kwaliteit zal de logische reactie zijn om unittests te gebruiken.

MOEILIJKHEDEN Zoals bij iedere verandering gaat het introduceren niet zonder slag of stoot. Allereerst is er het technische aspect: er moet een unittest framework beschikbaar zijn in de taal waarin ontwikkeld wordt. Als dit niet aanwezig is dan is een paar middagen voldoende om een basaal raamwerk te creëren. Dat framework dient ervoor om tests snel en herhaalbaar te kunnen uitvoeren. Daarom dienen alle unittests in het framework geplaatst te worden. Daarmee ondervangt het ook mogelijke integratieproblemen. Het is namelijk direct zichtbaar wanneer aanpassingen in een unit tot ongewenste fouten elders leiden.

Vervolgens moet men er nog mee leren werken. In een kern is een framework niet moeilijk: het bestaat uit een beperkte basisstructuur die helder en eenvoudig is. De moeilijkheid zit hem in de manier waarop unittests gemaakt worden. Ze worden niet geschreven *nadat* de code geïmplementeerd is, maar ervoor. Deze paradigmaverandering ('eerst tests schrijven en dan de daadwerkelijke code') is pittig.

De ervaring leert dat met name jonge programmeurs dit gemakkelijk oppakken. Zij hebben soms ook de structuur nodig die dit test-driven principe aanbiedt: het ordent hun gedachten over de te realiseren oplossing. Ervaren programmeurs hebben daarentegen vaak het gevoel terug-bij-af te zijn, doordat zij zo gewend zijn aan coderen en achteraf testen. Als ze echter het 'test-driven-development'-paradigma oppakken is hun valkuil om in de test te grote stappen te nemen waardoor er teveel functionaliteit tegelijkertijd getest wordt, en dat leidt tot omvangrijke en weinig overzichtelijke tests. Wanneer er kleinere stapjes genomen worden, zullen de tests overzichtelijker en beter overdraagbaar worden.

Tijdsdruk en softwareontwikkeling horen nu eenmaal bij elkaar, daar kan agile ontwikkelen niet zo heel veel aan veranderen. Juist die vervelende tijdsdruk verleidt ontwikkelaars soms tot halve oplossingen: de discipline van eerst de test maken en daarna coderen, wordt niet altijd volgehouden. Dan wordt het al gauw weer het oude vertrouwde "eerst coderen en daarna (eventueel) testen". Maar ja, dat testen is geen leuk werk. Laat dat dus maar zitten... En als we toch unittests gaan schrijven, tegen welke testgevallen was de code eigenlijk geprobeerd? Erg lastig om alles achteraf te terug te halen en te documenteren.

GOEDE UNITTEST Een unittest heeft als eigenschap dat het slechts één hele kleine eenheid test. Dat betekent niet dat het slechts een testgeval kent, in tegendeel zelfs. Testgevallen worden gegroepeerd in een klasse waarbinnen idealiter één context wordt aangehouden. Als blijkt dat meerdere contexts op één logische klasse ontstaan dan is dat een aanwijzing dat de klasse teveel verschillende verantwoordelijkheden heeft. Om een web van complexe afhankelijkheden te voorkomen, geven wij er de voorkeur aan om productieklassen en testklassen één

op één op elkaar af te beelden. Tests voor functies die triviale operaties uitvoeren hebben in de praktijk weinig zin. Een voorbeeld hiervan zijn getters en setters. Daarnaast zijn deze tests weinig verheffend: je test daar of een variabele een correcte waarde heeft gekregen. Ditzelfde geldt voor inserts en deletes in databases.

Unittests worden pas echt zinvol wanneer ze een transformatie testen: de omzetting van invoerparameters in een statusverandering of een berekende waarde. In het gebruik van unittests wordt er van uitgegaan dat de tests in elke willekeurige volgorde uitgevoerd moeten kunnen worden. Als unittests gebruik maken van opslagfaciliteiten als een database dan dient er voor gezorgd te worden dat deze tests zelf testrecords inserten en deleten. Zo hebben andere tests geen last van deze testgegevens en kan de testsuite -in het uiterste geval- ook op een productiedatabase gedraaid worden. En ook unittests zijn aan refactoring onderhevig. Dit helpt je namelijk om je tests overzichtelijk te houden, en delen functionaliteit te onderkennen die later naar de applicatie kan verhuizen.

VOORTPLOEGEN Wat mag de tester van dit alles verwachten? Vooral: een betere technische kwaliteit van de software. De aangeleverde code is simpelweg stabieler. Als er dus sprake is van een bestaande teststrategie (gebaseerd op code zonder unittests) zal herijking hiervan dus belangrijk zijn. Anders worden de gezochte fouten niet gevonden. Hij zal zijn aandachtsgebieden moeten gaan verleggen naar zaken op systeemniveau (en niet meer uitsluitend functioneel niveau). Configuraties, complexe uitzonderingssituaties, performance, usability en het gebruik daarbij van tools, zijn essentieel. Dit zal dus meer eisen aan testers gaan stellen dan alleen vakinhoudelijke kennis, iets waar nog weinig gespecialiseerde testbedrijven zich in bekwaamd hebben. Het belangrijkste van dit alles is niet de winst voor de programmeur of de tester, maar voor de klant. Die krijgt tenminste eindelijk eens iets waar hij op kan bouwen.

Links

Unittest frameworks:
http://www.xprogramming.com/software.htm
Testdriven development:
http://www.testdriven.com/modules/news/

Peter Schrier is zelfstandig software development coach en heeft in die hoedanigheid eXtreme programming veelvuldig geïmplementeerd. Hij werkt nauw samen met Exoftware, gespecialiseerd in agile enablement bedrijven. E-mail: pschrier@exoftware.com.

Anko Tijman is senior testspecialist bij de business unit Testen van Ordina System Integration & Development. Hij heeft zich vanuit het vakgebied software testen gespecialiseerd in agile testen en de rol van de tester in agile projecten. E-mail: anko.tijman@ordina.nl.

PATCHES Patches PATCHES Patches PATCHES Patches PATCHES

Gupta naar Linux

Terwijl Borland de hoop op succes voor een Linux-tool (Kylix) eigenlijk heeft opgegeven (zie het interview met Borland-CEO Fuller vanaf pagina 6), zet Gupta nu juist in op een Linux-strategie. Volgens zegslieden van Gupta was Borland misschien wel te vroeg met z'n Linux-strategie. Misschien is de timing van Gupta wel beter, de reacties zijn in ieder geval erg positief: de leverancier van de TeamDeveloper ontwikkelomgeving en de relationele database SQLBase, introduceert nu een TeamDeveloper versie voor gebruik onder Linux. TeamDeveloper 2005 is de eerste echte vierde generatietaal voor het Linux platform, zoals Gupta in het verleden eerder markt verwierf door de vroege voorloper van de ontwikkelomgeving te ontwikkelen voor de net in opmars zijnde Windows-omgeving. Gupta introduceerde toen de taal onder de naam SQLWindows en ging later door het leven als TeamDeveloper. TeamDeveloper is een krachtige ontwikkelomgeving, bestemd voor met name (grote) administratieve toepassingen en is zeer overzichtelijk. De editor kent verschillende varianten om de broncode te bekijken en te bewerken. Voor de 4GL is object-oriented design een vanzelfsprekendheid, waardoor ook grote multi-tiered applicaties eenvoudig en overzichtelijk zijn te bouwen en te onderhouden. Kenmerkend voor de huidige varianten van TeamDeveloper is dat de broncode voor zowel Linux als Windows portabel is. Hierdoor wordt het mogelijk bestaande broncode eenvoudig te migreren naar het Linux platform. TeamDeveloper 2005 wordt medio december op de schappen verwacht.

Nieuwe versie van Borland Delphi

Borland Software Corporation introduceert Borland Delphi 2005, de nieuwste versie van Borland's Rapid Application Development (RAD)-omgeving voor Windows- en .NET applicaties. Delphi 2005 combineert ondersteuning voor Win32, .NET, Delphi en C# in één omgeving, verbetert de ontwikkel- en teamproductiviteit en integreert met Borland's ALM-oplossingen. Delphi 2005 ondersteunt niet alleen meerdere talen en zowel Win32 als .NET SDK's, het bevat ook andere verbeteringen voor ontwikkel- en teamproductiviteit, zoals code refactoring, unit testing en het nieuwe ECO II (Enterprise Core Objects) modelgestuurde bedrijfsapplicatie-framework voor .NET.