Test Driven Development loont

Door de toenemende populariteit van Agile ontwikkelmethodieken groeit ook de bekendheid van het Test Driven Development (TDD). De twee zijn evenwel niet onlosmakelijk met elkaar verbonden, zij het dat ze wel logischerwijs in elkaars verlengde liggen. Het loont om deze manier van werken toe te passen.

Guido van Loon

Bij Agile projecten is alles erop gericht zo snel mogelijk waarde voor de organisatie te creëren. Hierbij geldt 'do the simplest thing that could possibly work' als leidend principe. Als gevolg daarvan kan het gebeuren dat na een aantal releases delen van de software aanpassingen nodig hebben *(refactoring)* omdat een eerder uitgangspunt *(the simplest thing)* niet meer toereikend is voor een volgende release. Een Agile methodiek is dan alleen effectief wanneer er geautomatiseerde testen zijn die efficiënt kunnen aantonen dat de *refactoring*-slag geen functionaliteit heeft beïnvloed.

Met TDD zorg je er niet alleen voor dat deze testen er zijn, de methode helpt ook bij het invullen van het technisch ontwerp, zodat er bijna als vanzelf een *loosely coupled,* op componenten gebaseerd ontwerp ontstaat. TDD krijgt daarom ook wel eens het stempel *'test driven design'*.

Unit testen

Test driven development is een specialisatie van unit testen. Daarom is het van belang om eerst te begrijpen hoe unit testen in zijn werk gaat voordat we test driven development kunnen bespreken.

Het is de taak van een softwareontwikkelaar op basis van functionele specificaties een correct functionerend stuk software te schrijven. Om te bewijzen dat de software naar behoren werkt, is het vaak nodig één of meerdere vormen van testen toe te passen. De meest bekende test is de acceptatietest waarbij de opdrachtgever vaststelt dat alles in orde is en de software accepteert.

Maar wanneer er tijdens zo'n acceptatietest nog fouten aan het licht komen, zijn deze meestal niet eenvoudig meer op te lossen. Wanneer dezelfde fout eerder in het ontwikkelproces zou zijn ontdekt, dan was het vele malen makkelijker en goedkoper geweest om deze te corrigeren. Daarom is het van belang om zo vroeg mogelijk in het proces eventueel gemaakte fouten te ontdekken en te herstellen, aangezien de kosten sterk stijgen naarmate men verder in het proces is.

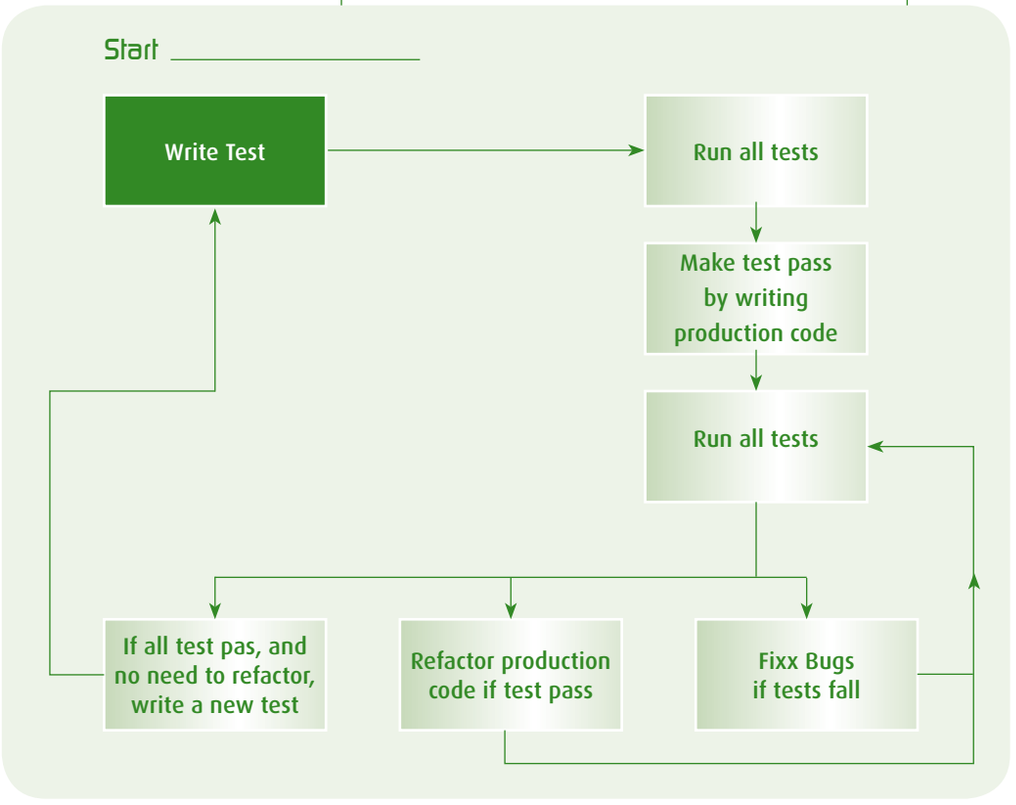
Hoge kwaliteit

Met correct functionerende software alleen zijn we er nog niet. Een goede softwareontwikkelaar streeft ernaar code te schrijven die van een hoge kwaliteit is. Vanzelfsprekend zitten er in kwalitatief goede code weinig fouten. Maar

code is pas echt goed wanneer deze volledig gedocumenteerd is en makkelijk door anderen is te begrijpen. Het moet ook mogelijk op een eenvoudige manier wijzigingen door te voeren zonder dat men bang hoeft te zijn dat er daarna onvoorspelbaar gedrag optreedt. Hoe vaak bekruipt je niet het gevoel dat je een bepaalde aanpassing liever niet zou doen, omdat je simpelweg niet weet wat de impact op het totale systeem zal zijn?

Unit testen is een manier van testen die de ontwikkelaar kan helpen de kwaliteit van zijn eigen code te verbeteren. Een unittest is een test die wordt opgesteld in dezelfde taal als waarin de software wordt ontwikkeld. Het is een test geschreven door de softwareontwikkelaar voor zijn eigen code. Elke unittest is verantwoordelijk voor het testen van een afgebakend stukje van de code (een unit). Unittests zijn autonoom, herhaalbaar en zijn snel en eenvoudig uit te voeren. Een typische unittest is te beschouwen als een klein programmaatje dat de ontwikkelaar tegelijk met de productiecode schrijft. Dit programmaatje (de test) voert een deel van de productiecode uit en controleert de uitkomst aan de hand van de verwachtingen die in de test zijn voorgeprogrammeerd. De test kan van zichzelf aangeven of hij geslaagd is. De meeste unittests worden geschreven met behulp van een framework zoals JUnit voor Java of MSTest voor Microsoft .NET.

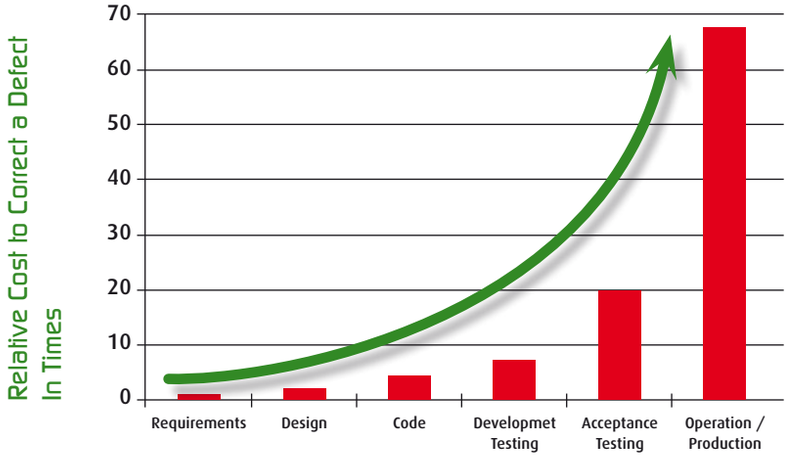
Een schematische weergave van het TDD-proces.



Doordat de unittest zelf ook een stuk programmacode is, is het eenvoudig om hem te herhalen. Stel je een codebase voor waarbij de correcte werking van elk stukje functionaliteit door middel van unittests valt te bewijzen. In zo'n code base kan je met een gerust hart een wijziging doorvoeren. Je hebt immers een complete set unittests waarmee je kan bewijzen dat alle functionaliteit nog steeds in tact is en het kost geen enkele moeite om al deze testen uit te laten voeren omdat ze geautomatiseerd zijn.

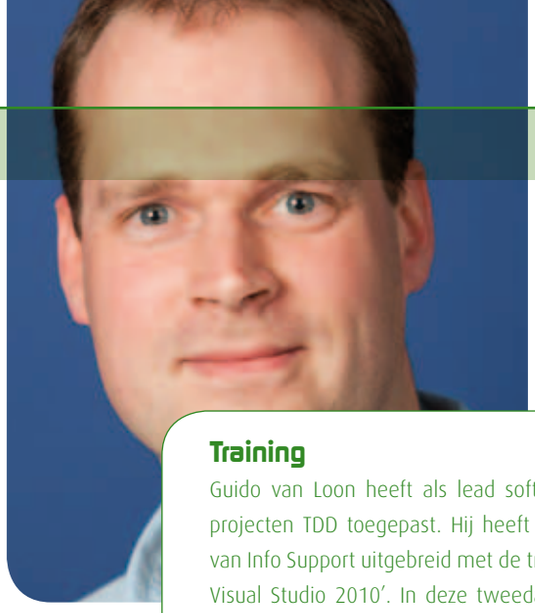
Elke unittest, mits duidelijk geschreven, is ook meteen een stukje documentatie van het systeem. De unittest kan in een paar regels code duidelijk laten zien hoe een methode op een class zich hoort te gedragen wanneer deze bepaalde input krijgt. Dit is waardevolle informatie voor een andere ontwikkelaar die zich op een later moment over de codebase moet ontfermen.

Deze figuur laat de relatieve kosten zien in de tijd die nodig zijn om een defect te herstellen.



Valkuilen

Helaas gebeurt het in de praktijk nog te vaak dat, naarmate de deadline begint te naderen, een projectleider of manager besluit het schrijven van unittests uit te stellen of helemaal te schrappen. Men denkt op deze manier tijd te winnen. Bij een organisatie die werkt met gescheiden ontwikkel- en testteams valt er inderdaad tijdwinst te boeken bij de programmeurs. Maar als



Training

Guido van Loon heeft als lead softwareontwikkelaar in verschillende projecten TDD toegepast. Hij heeft begin dit jaar het trainingsaanbod van Info Support uitgebreid met de training 'Unit Testing & Mocking with Visual Studio 2010'. In deze tweedaagse training leert de cursist TDD toepassen door dieper in te gaan op het schrijven van unittests en het isoleren van de unit die de test ondergaat. Van Loon deelt in deze training niet alleen de theorie van TDD maar ook de praktische ervaring die hij de afgelopen zeven jaar op dit gebied heeft opgedaan. Voor meer informatie ga naar http://www.infosupport.com/Training.

blijkt dat het aantal gevonden defecten daarna steeds in aantal toeneemt, loopt het hele project uiteindelijk toch vertraging op.

Het ontwikkelteam moet de gevonden fouten herstellen waarna dezelfde functionaliteit weer opnieuw naar het testteam gaat. Zonder unittests zijn er bij het herstellen van fouten mogelijk ongemerkt nieuwe fouten geïntroduceerd in functionaliteit die eerder wel correct werkte waardoor er nog een derde ronde herstel- en testwerk nodig is. Indien er wel unittests geschreven zouden zijn, was een aantal defects veel eerder gevonden, was het hersteld en zou het aantal bevindingen van het testteam lager zijn. Een derde ronde herstel- en testwerk voor nieuwe defecten, ontstaan tijdens herstelwerk, is met unit testen vrijwel uitgesloten.

Softwareontwikkelaars die met de beste bedoelingen op een goede manier unittests proberen te schrijven, lopen vroeg of laat tegen problemen aan dat hun unittests niet meer te beheersen zijn als de unit die getest wordt, te groot blijkt. De test krijgt dat in één keer te veel logica te verwerken, waardoor de unittest al snel te complex is. Een kleine wijziging in het systeem zorgt er dan voor dat te veel testen een aanpassingen nodig hebben om de wijziging te reflecteren. Wanneer er te veel van dit soort testen zijn, kan dat ervoor zorgen dat de extra winst die unittesten moet brengen, omslaat in extra kosten. Houd daarom de unit die getest wordt, zo klein mogelijk.

Een ander probleem van veel unittests zijn allerlei externe omgevingsfactoren die de uitkomst van de test beïnvloeden, zoals de inhoud van een database, de instellingen in een configuratiefile of de afhankelijkheid van een extern systeem waarover men geen controle heeft. Deze factoren dienen te worden geabstraheerd achter interfaces, zodat ze tijdens het unittesten zijn te vervangen door iets waarover de test controle heeft. Deze techniek staat bekend als mocking en is een waardevolle toevoeging in het arsenaal van een TDD-ontwikkelaar.

> Het vergt tijd om deze manier van werken in de vingers te krijgen; daarom is het zinvol vandaag al te beginnen.

Test Driven Development

Bij test driven development is het altijd nodig eerst de unittest te schrijven alvorens de functionaliteit te implementeren. De gedachte hierbij is dat de ontwikkelaar tijdens het schrijven van de unittest beter nadenkt over de manier waarop de functionaliteit moet worden geïmplementeerd. De ontwikkelaar krijgt daardoor de stimulans om code te schrijven die eenvoudiger in kleine testbare eenheden is op te delen. Hij vermijdt hiermee automatisch de valkuil van onbeheersbare unittests. Omdat de ontwikkelaar de test eerst schrijft, is het niet mogelijk code te implementeren die niet door een unittest kan worden getest. "Je kan iets niet bouwen als je het niet kan testen."

Het test-driven-development-proces spreekt eigenlijk voor zichzelf. Het is belangrijk elke keer alle testen uit te voeren zodat je zeker weet dat de nieuwe code geen bestaande tests heeft gebroken, dus geen bestaand gedrag heeft gewijzigd.

In de praktijk is TDD niet zo zwart-wit als de theorie ons doet geloven. Om een unittest te kunnen uitvoeren, is het nodig deze eerst te compileren. Hiervoor is het vaak noodzakelijk alvast een deel van de productiecode te implementeren, nog voordat de unittest voor de eerste keer valt uit te voeren. Ik merk zelf dat het schrijven van een unittest en productiecode hand in hand gaan.

Begin vandaag

TDD is niet onlosmakelijk verbonden met een Agile methodiek. De methode is ook prima toe te passen in de meer traditionele ontwikkelmethodieken. Mijn ervaring is dat mensen na een tijdje TDD niet meer anders willen ontwikkelen. Er is enig doorzettingsvermogen vereist en het ontwikkelteam moet er zelf wel achter staan wanneer zijn manier van werken wordt aangepast.

Een traditioneel softwareontwikkelteam kan geleidelijk naar een Agile methodiek toegroeien door te beginnen met het schrijven van unittests, bij voorkeur volgens TDD. Het vergt tijd om deze manier van werken in de vingers te krijgen; daarom is het zinvol vandaag al te beginnen.

Guido van Loon is werkzaam voor Info Support.